## 1 Minimal Examples

## 2 Basic Documentation

2.1 What’s new in Statsmodels

2.2 Getting started

2.3 Fitting models using R-style formulas

2.4 Installation

2.5 Related Packages

2.6 About Statsmodels

2.7 endog, exog, what’s that?

2.8 Import Paths and Structure

2.9 Pitfalls

2.10 Developer Page

2.11 Internal Classes

## 3 Table of Contents

3.1 Linear Regression

3.2 Generalized Linear Models

3.3 Generalized Estimating Equations

3.4 Robust Linear Models

3.5 Regression with Discrete Dependent Variable

3.6 ANOVA

3.7 Time Series analysis tsa

3.8 Statistics stats

3.9 Nonparametric Methods nonparametric

3.10 Generalized Method of Moments gmm

3.11 Empirical Likelihood emplike

3.12 Other Models miscmodels

3.13 Distributions

3.14 Graphics

3.15 Input-Output iolib

3.16 Tools

3.17 The Datasets Package

3.18 Sandbox

## 4 Indices and tables

### Bibliography

### Python Module Index
statsmodels is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration. An extensive list of result statistics are available for each estimator. The results are tested against existing statistical packages to ensure that they are correct. The package is released under the open source Modified BSD (3-clause) license. The online documentation is hosted at sourceforge.
Since version 0.5.0 of statsmodels, you can use R-style formulas together with pandas data frames to fit your models. Here is a simple example using ordinary least squares:

```python
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

# Load data
url = 'http://vincentarelbundock.github.io/Rdatasets/csv/HistData/Guerry.csv'
dat = pd.read_csv(url)

# Fit regression model (using the natural log of one of the regressors)
results = smf.ols('Lottery ~ Literacy + np.log(Pop1831)', data=dat).fit()

# Inspect the results
print(results.summary())
```

You can also use numpy arrays instead of formulas:

```python
import numpy as np
import statsmodels.api as sm

# Generate artificial data (2 regressors + constant)
nobs = 100
X = np.random.random((nobs, 2))
X = sm.add_constant(X)
beta = [1, .1, .5]
e = np.random.random(nobs)
y = np.dot(X, beta) + e

# Fit regression model
results = sm.OLS(y, X).fit()

# Inspect the results
print(results.summary())
```

Have a look at `dir(results)` to see available results. Attributes are described in `results.__doc__` and results methods have their own docstrings.
2.1 What’s new in Statsmodels

2.1.1 0.5 Release

Release 0.5.0

Statsmodels 0.5 is a large and very exciting release that brings together a year of work done by 38 authors, including over 2000 commits. It contains many new features and a large amount of bug fixes detailed below.

See the list of fixed issues for specific closed issues.

The following major new features appear in this version.

Support for Model Formulas via Patsy

Statsmodels now supports fitting models with a formula. This functionality is provided by patsy. Patsy is now a dependency for statsmodels. Models can be individually imported from the statsmodels.formula.api namespace or you can import them all as:

```python
import statsmodels.formula.api as smf
```

Alternatively, each model in the usual statsmodels.api namespace has a from_formula classmethod that will create a model using a formula. Formulas are also available for specifying linear hypothesis tests using the t_test and f_test methods after model fitting. A typical workflow can now look something like this.

```python
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

url = 'http://vincentarelbundock.github.io/Rdatasets/csv/HistData/Guerry.csv'
data = pd.read_csv(url)

# Fit regression model (using the natural log of one of the regressors)
results = smf.ols('Lottery ~ Literacy + np.log(Pop1831)', data=data).fit()

See here for some more documentation of using formulas in statsmodels

Empirical Likelihood (Google Summer of Code 2012 project)

Empirical Likelihood-Based Inference for moments of univariate and multivariate variables is available as well as EL-based ANOVA tests. EL-based linear regression, including the regression through the origin model. In addition, the
accelerated failure time model for inference on a linear regression model with a randomly right censored endogenous variable is available.

**Analysis of Variance (ANOVA) Modeling**

Support for ANOVA is now available including type I, II, and III sums of squares. See ANOVA.

**Multivariate Kernel Density Estimators (GSoC 2012 project)**

Kernel density estimation has been extended to handle multivariate estimation as well via product kernels. It is available as sm.nonparametric.KDEMultivariate. It supports least squares and maximum likelihood cross-validation for bandwidth estimation, as well as mixed continuous, ordered, and unordered categorical data. Conditional density estimation is also available via sm.nonparametric.KDEMUltivariateConditional.

**Nonparametric Regression (GSoC 2012 project)**

Kernel regression models are now available via sm.nonparametric.KernelReg. It is based on the product kernel mentioned above, so it also has the same set of features including support for cross-validation as well as support for estimation mixed continuous and categorical variables. Censored kernel regression is also provided by kernel_regression.KernelCensoredReg.

**Quantile Regression Model**

Quantile regression is supported via the sm.QuantReg class. Kernel and bandwidth selection options are available for estimating the asymptotic covariance matrix using a kernel density estimator.

**Negative Binomial Regression Model**

It is now possible to fit negative binomial models for count data via maximum-likelihood using the sm.NegativeBinomial class. NB1, NB2, and geometric variance specifications are available.

**l1-penalized Discrete Choice Models**

A new optimization method has been added to the discrete models, which includes Logit, Probit, MNLogit and Poisson, that makes it possible to estimate the models with an l1, linear, penalization. This shrinks parameters towards zero and can set parameters that are not very different from zero to zero. This is especially useful if there are a large number of explanatory variables and a large associated number of parameters. CVXOPT is now an optional dependency that can be used for fitting these models.

**New and Improved Graphics**

- **ProbPlot**: A new ProbPlot object has been added to provide a simple interface to create P-P, Q-Q, and probability plots with options to fit a distribution and show various reference lines. In the case of Q-Q and P-P plots, two different samples can be compared with the *other* keyword argument. sm.graphics.ProbPlot
import numpy as np
import statsmodels.api as sm
x = np.random.normal(loc=1.12, scale=0.25, size=37)
y = np.random.normal(loc=0.75, scale=0.45, size=37)
ppx = sm.ProbPlot(x)
ppy = sm.ProbPlot(y)
fig1 = ppx.qqplot()
fig2 = ppx.qqplot(other=ppy)

- **Mosaic Plot**: Create a mosaic plot from a contingency table. This allows you to visualize multivariate categorical data in a rigorous and informative way. Available with `sm.graphics.mosaic`.

- **Interaction Plot**: Interaction plots now handle categorical factors as well as other improvements. `sm.graphics.interaction_plot`.

- **Regression Plots**: The regression plots have been refactored and improved. They can now handle pandas objects and regression results instances appropriately. See `sm.graphics.plot_fit`, `sm.graphics.plot_regress_exog`, `sm.graphics.plot_partregress`, `sm.graphics.plot_ccpr`, `sm.graphics.abline_plot`, `sm.graphics.influence_plot`, and `sm.graphics.plot_leverage_resid2`.

### Power and Sample Size Calculations

The power module (statsmodel.stats.power) currently implements power and sample size calculations for the t-tests (`sm.stats.TTestPower`, `sm.stats.TTestIndPower`), normal based test (`sm.stats.NormIndPower`), F-tests (`sm.stats.FTestPower`, `class:sm.stats.FTestAnovaPower <FTestAnovaPower>`) and Chisquare goodness of fit (`sm.stats.GofChisquarePower`) test. The implementation is class based, but the module also provides three shortcut functions, `sm.stats.tt_solve_power`, `sm.stats.tt_ind_solve_power` and `sm.stats.zt_ind_solve_power` to solve for any one of the parameters of the power equations. See this blog post for a more in-depth description of the additions.

### Other important new features

- **IPython notebook examples**: Many of our examples have been converted or added as IPython notebooks now. They are available here.

- **Improved marginal effects for discrete choice models**: Expanded options for obtaining marginal effects after the estimation of nonlinear discrete choice models are available. See `get_margeff`.

- **OLS influence outlier measures**: After the estimation of a model with OLS, the common set of influence and outlier measures and a outlier test are now available attached as methods `get_influence` and `outlier_test` to the Results instance. See `OLSInfluence` and `outlier_test`.

- **New datasets**: New `datasets` are available for examples.

- **Access to R datasets**: We now have access to many of the same datasets available to R users through the Rdatasets project. You can access these using the `sm.datasets.get_rdataset` function. This function also includes caching of these datasets.

- **Improved numerical differentiation tools**: Numerical differentiation routines have been greatly improved and expanded to cover all the routines discussed in:


  See the `sm.tools.numdiff` module.
• **Consistent constant handling across models**: Result statistics no longer rely on the assumption that a constant is present in the model.

• **Missing value handling across models**: Users can now control what models do in the presence of missing values via the `missing` keyword available in the instantiation of every model. The options are `'none'`, `'drop'`, and `'raise'`. The default is `'none'`, which does no missing value checks. To drop missing values use `'drop'`. And `'raise'` will raise an error in the presence of any missing data.

• **Ability to write Stata datasets**: Added the ability to write Stata `.dta` files. See `sm.iolib.StataWriter`.

• **ARIMA modeling**: Statsmodels now has support for fitting Autoregressive Integrated Moving Average (ARIMA) models. See `ARIMA` and `ARIMAResults` for more information.

• **Support for dynamic prediction in AR(I)MA models**: It is now possible to obtain dynamic in-sample forecast values in `ARMA` and `ARIMA` models.

• **Improved Pandas integration**: Statsmodels now supports all frequencies available in pandas for time-series modeling. These are used for intelligent dates handling for prediction. These features are available, if you pass a pandas Series or DataFrame with a DatetimeIndex to a time-series model.

• **New statistical hypothesis tests**: Added statistics for calculating interrater agreement including Cohen’s kappa and Fleiss’ kappa (See [Interrater Reliability and Agreement](https://www.statsmodels.org/stable/generated/statsmodels.stats.inter_rater.cohen_kappa.html)), statistics and hypothesis tests for proportions (See [proportion stats](https://www.statsmodels.org/stable/generated/statsmodels.stats.proportion.proportion_confint.html)), Tukey HSD (with plot) was added as an enhancement to the multiple comparison tests (See [sm.stats.multicomp.MultiComparison](https://www.statsmodels.org/stable/generated/statsmodels.stats.multicomp.MultiComparison.html), [sm.stats.multicomp.pairwise_tukeyhsd](https://www.statsmodels.org/stable/generated/statsmodels.stats.multicomp.pairwise_tukeyhsd.html)). Weighted statistics and t tests were enhanced with new options. Tests of equivalence for one sample and two independent or paired samples were added based on t tests and z tests (See [Basic Statistics and t-Tests with frequency weights](https://www.statsmodels.org/stable/generated/statsmodels.stats.weightstats.ztest.html)).

**Major Bugs fixed**

• Post-estimation statistics for weighted least squares that depended on the centered total sum of squares were not correct. These are now correct and tested. See #501.

• Regression through the origin models now correctly use uncentered total sum of squares in post-estimation statistics. This affected the $R^2$ value in linear models without a constant. See #27.

**Backwards incompatible changes and deprecations**

• Cython code is now non-optional. You will need a C compiler to build from source. If building from github and not a source release, you will also need Cython installed. See the [installation documentation](https://www.statsmodels.org/stable/install.html).

• The `q_matrix` keyword to `t_test` and `f_test` for linear models is deprecated. You can now specify linear hypotheses using formulas.

• The `conf_int` keyword to `sm.tsa.acf` is deprecated.

• The `names` argument is deprecated in `sm.tsa.VAR` and `sm.tsa.SVAR <vector_ar.svar_model.SVAR>`. This is now automatically detected and handled.

• The `order` keyword to `sm.tsa.ARMA.fit` is deprecated. It is now passed in during model instantiation.

• The empirical distribution function (sm.distributions.ECDF) and supporting functions have been moved to `statsmodels.distributions`. Their old paths have been deprecated.

• The margeff method of the discrete choice models has been deprecated. Use `get_margeff` instead. See above. Also, the vague resid attribute of the discrete choice models has been deprecated in favor of the more descriptive resid_dev to indicate that they are deviance residuals.

• The class KDE has been deprecated and renamed to `KDEUnivariate` to distinguish it from the new `KDEMultivariate`. See above.
Development summary and credits  The previous version (statsmodels 0.4.3) was released on July 2, 2012. Since then we have closed a total of 380 issues, 172 pull requests and 208 regular issues. The detailed list can be viewed. This release is a result of the work of the following 38 authors who contributed total of 2032 commits. If for any reason, we’ve failed to list your name in the below, please contact us:

- Ana Martinez Pardo <anamartinezpardo-at-gmail.com>
- anov <novikova.go.zoom-at-gmail.com>
- avishaylivne <avishay.livne-at-gmail.com>
- Bruno Rodrigues <rodrigues.bruno-at-aquitania.org>
- Carl Vogel <carlvj-at-gmail.com>
- Chad Fulton <chad-at-chadfulton.com>
- Christian Prinoth <christian-at-prinoth.name>
- Daniel B. Smith <neuromathdan-at-gmail.com>
- denegemann <denis.engemann-at-gmail.com>
- Dieter Vandenbussche <dvandenbussche-at-axioma.com>
- Dougal Sutherland <dougal-at-gmail.com>
- Enrico Giampieri <enrico.giampieri-at-unibo.it>
- evelynmitchell <efm-github-at-linsomniac.com>
- George Panterov <econgpanterov-at-gmail.com>
- Grayson <graysonbadgley-at-gmail.com>
- Jan Schulz <jasc-at-gmx.net>
- Josef Perktold <josef.pktd-at-gmail.com>
- Jeff Reback <jeff-at-reback.net>
- Justin Grana <jg3705a-at-student.american.edu>
- lanmored <ianlangmore-at-gmail.com>
- Matthew Brett <matthew.brett-at-gmail.com>
- Nathaniel J. Smith <njs-at-pobox.com>
- otterb <itoi-at-live.com>
- padarn <padarn-at-wilsonp.anu.edu.au>
- Paul Hobson <pmhobson-at-gmail.com>
- Pietro Battiston <pe-at-pietrobattiston.it>
- Ralf Gommers <ralf.gommers-at-googlemail.com>
- Richard T. Guy <richardtguy84-at-gmail.com>
- Robert Cimrman <cimrman3-at-ntc.zcu.cz>
- Skipper Seabold <jsseabold-at-gmail.com>
- Thomas Haslwanter <thomas.haslwanter-at-fh-linz.at>
- timmie <timmichelsen-at-gmx-topmail.de>
- Tom Augspurger <thomas-augsburger-at-uiowa.edu>

2.1. What's new in Statsmodels
2.1.2 Issues closed in the 0.5.0 development cycle

Issued closed in 0.5.0

GitHub stats for release 0.5.0 (07/02/2012/ - 08/14/2013/).

We closed a total of 380 issues, 172 pull requests and 208 regular issues. This is the full list (generated with the script `tools/github_stats.py`):

This list is automatically generated, and may be incomplete:

Pull Requests (172):

- PR #1010: DOC/RLS: Update release notes workflow. Help Needed!
- PR #1014: DOC: nbgenerate does not like the comment at end of line.
- PR #1012: DOC: Add link to notebook and crosslink ref. Closes #924.
- PR #997: misc, tests, diagnostic
- PR #817: Add 3 new unit tests for arima_process
- PR #1001: BUG include_package_data for install closes #907
- PR #1005: GITHUB: Contributing guidlines
- PR #1007: Cleanup docs for release
- PR #1003: BUG: Workaround for bug in sphinx 1.1.3. See #1002.
- PR #1004: DOC: Update maintainer notes with branching instructions.
- PR #1000: BUG: Support pandas 0.8.0.
- PR #996: BUG: Handle combo of pandas 0.8.0 and dateutils 1.5.0
- PR #995: ENH: Print dateutil version.
- PR #994: ENH: Fail gracefully for version not found.
- PR #993: More conservative error catching in TimeSeriesModel
- PR #992: Misc fixes 12: adjustments to unit test
- PR #985: MAINT: Print versions script.
- PR #986: ENH: Prefer to_offset to get_offset. Closes #964.
• PR #984: COMPAT: Pandas 0.8.1 compatibility. Closes #983.
• PR #982: Misc fixes 11
• PR #978: TST: generic mle pareto disable bsejac tests with estimated loc
• PR #977: BUG python 3.3 fix for numpy str TypeError, see #633
• PR #975: Misc fixes 10 numdiff
• PR #970: BUG: array too long, raises exception with newer numpy closes #967
• PR #965: Vincent summary2 rebased
• PR #933: Update and improve GenericlikelihoodModel and miscmodels
• PR #950: BUG/REF mcnemar fix exact pvalue, allow table as input
• PR #951: Pylint emplike formula genmod
• PR #956: Fix a docstring in KDEMultivariateConditional.
• PR #949: BUG fix lowess sort when nans closes #946
• PR #932: ENH: support basinhopping solver in LikelihoodModel.fit()
• PR #927: DOC: clearer minimal example
• PR #919: Ols summary crash
• PR #918: Fixes10 emplike lowess
• PR #909: Bugs in GLM pvalues, more tests, pylint
• PR #906: ENH: No fmax with Windows SDK so define inline.
• PR #905: MAINT more fixes
• PR #898: Misc fixes 7
• PR #896: Quantreg rebase2
• PR #895: Fixes issue #832
• PR #893: ENH: Remove unneeded restriction on low. Closes #867.
• PR #894: MAINT: Remove broken function. Keep deprecation. Closes #781.
• PR #856: Carljv improved lowess rebased2
• PR #884: Pyflakes cleanup
• PR #887: BUG: Fix kde caching
• PR #883: Fixed pyflakes issue in discrete module
• PR #882: Update prestd.py
• PR #871: Update of sandbox doc
• PR #631: WIP: Correlation positive semi definite
• PR #857: BLD: apt get dependencies from Neurodebian, whitespace cleanup
• PR #855: AnaMP issue 783 mixture rvs tests rebased
• PR #854: Enrico multinear rebased
• PR #849: Tyler tukeyhsd rebased
• PR #848: BLD TravisCI use python-dateutil package

2.1. What’s new in Statsmodels
• PR #784: Misc07 cleanup multipletesting and proportions
• PR #841: ENH: Add load function to main API. Closes #840.
• PR #820: Ensure that tuples are not considered as data, not as data containers
• PR #822: DOC: Update for Cython changes.
• PR #765: Fix build issues
• PR #800: Automatically generate output from notebooks
• PR #802: BUG: Use two- not one-sided t-test in t_test. Closes #740.
• PR #806: ENH: Import formula.api in statsmodels.api namespace.
• PR #803: ENH: Fix arima error message for bad start_params
• PR #801: DOC: Fix ANOVA section titles
• PR #795: Negative Binomial Rebased
• PR #787: Origintests
• PR #794: ENH: Allow pandas-in/pandas-out in tsa.filters
• PR #791: Github stats for release notes
• PR #779: added np.asarray call to durbin_watson in stattools
• PR #772: Anova docs
• PR #776: BUG: Fix dates_from_range with length. Closes #775.
• PR #774: BUG: Attach prediction start date in AR. Closes #773.
• PR #767: MAINT: Remove use of deprecated from examples and docs.
• PR #762: ENH: Add new residuals to wrapper
• PR #754: Fix arima predict
• PR #760: ENH: Adjust for k_trend in information criteria. Closes #324.
• PR #761: ENH: Fixes and tests sign_test. Closes #642.
• PR #759: Fix 236
• PR #758: DOC: Update VAR docs. Closes #537.
• PR #752: Discrete cleanup
• PR #750: VAR with 1d array
• PR #748: Remove reference to new_t_test and new_f_test.
• PR #739: DOC: Remove outdated note in docstring
• PR #732: BLD: Check for patsy dependency at build time + docs
• PR #731: Handle wrapped
• PR #730: Fix opt fulloutput
• PR #729: Get rid of warnings in docs build
• PR #698: update url for hsb2 dataset
• PR #727: DOC: Fix indent and add missing params to linear models. Closes #709.
• PR #726: CLN: Remove unused method. Closes #694
• PR #725: BUG: Should call anova_single. Closes #702.
• PR #723: Rootfinding for Power
• PR #722: Handle pandas.Series with names in make_lags
• PR #714: Fix 712
• PR #668: Allow for any pandas frequency to be used in TimeSeriesModel.
• PR #711: Misc06 - bug fixes
• PR #708: BUG: Fix one regressor case for conf_int. Closes #706.
• PR #700: Bugs rebased
• PR #680: BUG: Swap arguments in fftconvolve for scipy >= 0.12.0
• PR #640: Misc fixes 05
• PR #663: a typo in runs.py doc string for mcnemar test
• PR #652: WIP: fixing pyflakes / pep8, trying to improve readability
• PR #619: DOC: intro to formulas
• PR #648: BF: Make RLM stick to Huber’s description
• PR #649: Bug Fix
• PR #637: Pyflakes cleanup
• PR #634: VAR DOC typo
• PR #623: Slowtests
• PR #621: MAINT: in setup.py, only catch ImportError for pandas.
• PR #590: Cleanup test output
• PR #591: Interrater agreement and reliability measures
• PR #618: Docs fix the main warnings and errors during sphinx build
• PR #610: nonparametric examples and some fixes
• PR #578: Fix 577
• PR #575: MNT: Remove deprecated scikits namespace
• PR #499: WIP: Handle constant
• PR #567: Remove deprecated
• PR #571: Dataset docs
• PR #561: Grab rdatasets
• PR #570: DOC: Fixed links to Rdatasets
• PR #524: DOC: Clean up discrete model documentation.
• PR #506: ENH: Re-use effects if model fit with QR
• PR #556: WIP: L1 doc fix
• PR #564: TST: Use native integer to avoid issues in dtype asserts
• PR #543: Travis CI using M.Brett nipy hack
• PR #558: Plot cleanup

2.1. What’s new in Statsmodels
• PR #541: Replace pandas DataMatrix with DataFrame
• PR #534: Stata test fixes
• PR #532: Compat 323
• PR #531: DOC: Add ECDF to distributions docs
• PR #526: ENH: Add class to write Stata binary dta files
• PR #521: DOC: Add abline plot to docs
• PR #518: Small fixes: interaction_plot
• PR #508: ENH: Avoid taking cholesky decomposition of diagonal matrix
• PR #509: DOC: Add ARIMA to docs
• PR #510: DOC: realdpi is disposable personal income. Closes #394.
• PR #507: ENH: Protect numdifftools import. Closes #45
• PR #504: Fix weights
• PR #498: DOC: Add patys requirement to install docs
• PR #491: Make _data a public attribute.
• PR #494: DOC: Fix pandas links
• PR #492: added intersphinx for pandas
• PR #422: Handle missing data
• PR #485: ENH: Improve error message for pandas objects without dates in index
• PR #428: Remove other data
• PR #483: Arima predict bug
• PR #482: TST: Do array-array comparison when using numpy.testing
• PR #471: Formula rename df -> data
• PR #473: Vincent docs tweak rebased
• PR #468: Docs 050
• PR #462: El aft rebased
• PR #461: TST: numpy 1.5.1 compatibility
• PR #460: Emplike desc reg rebase
• PR #410: Discrete model marginal effects
• PR #417: Numdiff cleanup
• PR #398: Improved plot_corr and plot_corr_grid functions.
• PR #401: BUG: Finish refactoring margeff for dummy. Closes #399.
• PR #400: MAINT: remove lowess.py, which was kept in 0.4.x for backwards compatibi...
• PR #371: BF+TEST: fixes, checks and tests for istestable
• PR #351: ENH: Copy diagonal before write for upcoming numpy changes
• PR #384: REF: Move mixture_rvs out of sandbox.
• PR #368: ENH: Add polished version of acf/pacf plots with confidence intervals
• PR #378: Infer freq
• PR #374: ENH: Add Fair’s extramarital affair dataset. From tobit-model branch.
• PR #358: ENH: Add method to OLSResults for outlier detection
• PR #369: ENH: allow predict to pass through patsy for transforms
• PR #352: Formula integration rebased
• PR #360: REF: Deprecate order in fit and move to ARMA init
• PR #366: Version fixes
• PR #359: DOC: Fix sphinx warnings

Issues (208):
• #1036: Series no longer inherits from ndarray
• #1038: DataFrame with integer names not handled in ARIMA
• #1028: Test fail with windows and Anaconda - Low priority
• #676: acorr_breush_godfrey undefined nlags
• #922: lowess returns inconsistent with option
• #425: no bse in robust with norm=TrimmedMean
• #1025: add_constant incorrectly detects constant column
• #533: py3 compatibility pandas.read_csv(urlopen(...))
• #662: doc: install instruction: explicit about removing scikits.statsmodels
• #910: test failure Ubuntu TestARMLEConstant.test_dynamic_predict
• #80: t_model: f_test, t_test don’t work
• #432: GenericLikelihoodModel change default for score and hessian
• #454: BUG/ENH: HuberScale instance is not used, allow user defined scale estimator
• #98: check connection or connect summary to variable names in wrappers
• #418: BUG: MNLogit loglikeobs, jac
• #1017: nosetests warnings
• #924: DOCS link in notebooks to notebook for download
• #1011: power test endless loop possible
• #907: BLD data_files for stats.libqsturng
• #328: consider moving example scripts into IPython notebooks
• #1002: Docs won’t build with Sphinx 1.1.3
• #69: Make methods like compare_ftest work with wrappers
• #503: summary_old in RegressionResults
• #991: TST precision of normal_power
• #945: Installing statsmodels from github?
• #964: Prefer to_offset not get_offset in tsa stuff
• #983: bug: pandas 0.8.1 incompatibility

2.1. What’s new in Statsmodels
• #899: build_ext inplace doesn’t cythonize
• #923: location of initialization code
• #980: auto lag selection in S_hac_simple
• #968: genericMLE Ubuntu test failure
• #633: python 3.3 compatibility
• #728: test failure for solve_power with fsolve
• #971: numdiff test cases
• #976: VAR Model does not work in 1D
• #972: numdiff: epsilon has no minimum value
• #967: lowes test failure Ubuntu
• #948: nonparametric tests: mcnemar, cochranq unit test
• #963: BUG in runtest_2sample
• #946: Issue with lowess() smoother in statsmodels
• #868: k_vars > nob
• #917: emplike emplikeAFT stray dimensions
• #264: version comparisons need to be made more robust (may be just use LooseVersion)
• #674: failure in test_foreign, pandas testing
• #828: GLMResults inconsistent distribution in pvalues
• #908: RLM missing test for tvalues, pvalues
• #463: formulas missing in docs
• #256: discrete Nbin has zero test coverage
• #831: test errors running bdist
• #733: Docs: interrater cohens_kappa is missing
• #897: lowess failure - sometimes
• #902: test failure tsa.filters precision too high
• #901: test failure stata_writer_pandas, newer versions of pandas
• #900: ARIMA.__new__ errors on python 3.3
• #832: notebook errors
• #867: Baxter King has unneeded limit on value for low?
• #781: discreteResults margeff method not tests, obsolete
• #870: discrete unit tests duplicates
• #630: problems in regression plots
• #885: Caching behavior for KDEUnivariate icdf
• #869: sm.tsa.ARMA(..., order=(p,q)) gives “__init__() got an unexpected keyword argument ‘order’” error
• #783: statsmodelsdistributionsmixture_rvs.py no unit tests
• #824: Multicomparison w/Pandas Series
• #789: presentation of multiple comparison results
• #764: BUG: multipletests incorrect reject for Holm-Sidak
• #766: multipletests - status and tests of 2step FDR procedures
• #763: Bug: multipletests raises exception with empty array
• #840: sm.load should be in the main API namespace
• #830: invalid version number
• #821: Fail gracefully when extensions are not built
• #204: Cython extensions built twice?
• #689: tutorial notebooks
• #740: why does t_test return one-sided p-value
• #804: What goes in statsmodels.formula.api?
• #675: Improve error message for ARMA SVD convergence failure.
• #15: arma singular matrix
• #559: Add Rdatasets to optional dependencies list
• #796: Prediction Standard Errors
• #793: filters are not pandas aware
• #785: Negative R-squared
• #777: OLS residuals returned as Pandas series when endog and exog are Pandas series
• #770: Add ANOVA to docs
• #775: Bug in dates_from_range
• #773: AR model pvalues error with Pandas
• #768: multipletests: numerical problems at threshold
• #355: add draw if interactive to plotting functions
• #625: Exog is not correctly handled in ARIMA predict
• #626: ARIMA summary does not print exogenous variable coefficients
• #657: order (0,1) breaks ARMA forecast
• #736: ARIMA predict problem for ARMA model
• #324: ic in ARResults, aic, bic, hqic, fpe inconsistent definition?
• #642: sign_test check
• #236: AR start_params broken
• #235: tests hang on Windows
• #156: matplotlib deprecated legend? var plots
• #331: Remove stale tests
• #592: test failures in datetools
• #537: Var Models
• #755: Unable to access AR fit parameters when model is estimated with pandas.DataFrame

2.1. What’s new in Statsmodels
• #670: discrete: numerically useless clipping
• #515: MNLogit residuals raise a TypeError
• #225: discrete models only define deviance residuals
• #594: remove skiptest in TestProbitCG
• #681: Dimension Error in discrete_model.py When Running test_dummy_*
• #744: DOC: new_f_test
• #549: Ship released patsy source in statsmodels
• #588: patsy is a hard dependency?
• #716: Tests missing for functions if pandas is used
• #715: statmodels regression plots not working with pandas datatypes
• #450: BUG: full_output in optimizers Likelihood model
• #709: DOCstrings linear models don’t have missing params
• #370: BUG weightstats has wrong cov
• #694: DiscreteMargins duplicate method
• #702: bug, pylint stats.anova
• #423: Handling of constant across models
• #456: BUG: ARMA date handling incompatibility with recent pandas
• #514: NaNs in Multinomial
• #405: Check for existing old version of scikits.statsmodels?
• #586: Segmentation fault with OLS
• #721: Unable to run AR on named time series objects
• #125: caching pinv_wexog broke iterative fit - GLSAR
• #712: TSA bug with frequency inference
• #319: Timeseries Frequencies
• #707: .summary with alpha ignores parsed value
• #673: nonparametric: bug in _kernel_base
• #710: test_power failures
• #706: .conf_int() fails on linear regression without intercept
• #679: Test Baxter King band-pass filter fails with scipy 0.12 beta1
• #552: influence outliers breaks when regressing on constant
• #639: test folders not on python path
• #565: omni_normtest doesn’t propagate the axis argument
• #563: error in doc generation for AR.fit
• #109: TestProbitCG failure on Ubuntu
• #661: from scipy import comb fails on the latest scipy 0.11.0
• #413: DOC: example_discrete.py missing from 0.5 documentation
2.1. What’s new in Statsmodels

- #644: FIX: factor plot + examples broken
- #645: STY: pep8 violations in many examples
- #173: doc sphinx warnings
- #601: bspline.py dependency on old scipy.stats.models
- #103: ecdf and step function conventions
- #18: Newey-West sandwich covariance is missing
- #279: cov_nw_panel not tests, example broken
- #150: precision in test_discrete.TestPoissonNewton.test_jac
- #480: rescale loglike for optimization
- #627: Travis-CI support for scipy
- #622: mark tests as slow in emplike
- #589: OLS F-statistic error
- #572: statsmodels/tools/data.py Stuck looking for la.py
- #580: test errors in graphics
- #577: PatsyData detection buglet
- #470: remove deprecated features
- #573: lazy imports are (possibly) very slow
- #438: New results instances are not in online documentation
- #542: Regression plots fail when Series objects passed to sm.OLS
- #239: release 0.4.x
- #530: 11 docs issues
- #539: test for statwriter (failure)
- #490: Travis CI on PRs
- #252: doc: distributions.rst refers to sandbox only
- #85: release 0.4
- #65: MLE fit of AR model has no tests
- #522: test doesn’t propagate arguments to nose
- #517: missing array conversion or shape in linear model
- #523: test failure with ubuntu decimals too large
- #520: web site documentation, source not updated
- #488: Avoid cholesky decomposition of diagonal matrices in linear regression models
- #394: Definition in macrodata NOTE
- #45: numdifftools dependency
- #501: WLS/GLS post estimation results
- #500: WLS fails if weights is a pandas.Series
- #27: add hasconstant indicator for R-squared and df calculations
• #497: DOC: add patsy?
• #495: ENH: add footer SimpleTable
• #402: model._data -> model.data?
• #477: VAR NaN Bug
• #421: Enhancement: Handle Missing Data
• #489: Expose model._data as model.data
• #315: tsa models assume pandas object indices are dates
• #440: arima predict is broken for steps > q and q != 1
• #458: TST BUG? comparing pandas and array in tests, formula
• #464: from_formula signature
• #245: examples in docs: make nicer
• #466: broken example, pandas
• #57: Unhelpful error from bad exog matrix in model.py
• #271: ARMA.geterrors requires model to be fit
• #350: Writing to array returned np.diag
• #354: example_rst does not copy unchanged files over
• #467: Install issues with Pandas
• #444: ARMA example on stable release website not working
• #377: marginal effects count and discrete adjustments
• #426: “svd” method not supported for OLS.fit()
• #409: Move numdiff out of the sandbox
• #416: Switch to complex-step Hessian for AR(I)MA
• #415: bug in kalman_loglike_complex
• #397: plot_corr axis text labeling not working (with fix)
• #399: discrete errors due to incorrect in-place operation
• #389: VAR test_normality is broken with KeyError
• #388: Add tsaplots to graphics.api as graphics.tsa
• #387: predict date wasn’t getting set with start = None
• #386: p-values not returned from acf
• #385: Allow AR.select_order to work without model being fit
• #383: Move mixture_rvs out of sandbox.
• #248: ARMA breaks with a 1d exog
• #273: When to give order for AR/AR(I)MA
• #363: examples folder -> tutorials folder
• #346: docs in sitepackages
• #353: PACF docs raise a sphinx warning
For an overview of changes that occurred previous to the 0.5.0 release see old_changes.

2.2 Getting started

This very simple case-study is designed to get you up-and-running quickly with statsmodels. Starting from raw data, we will show the steps needed to estimate a statistical model and to draw a diagnostic plot. We will only use functions provided by statsmodels or its pandas and patsy dependencies.

2.2.1 Loading modules and functions

After installing statsmodels and its dependencies, we load a few modules and functions:

```python
In [1]: import statsmodels.api as sm
In [2]: import pandas
In [3]: from patsy import dmatrices
```

pandas builds on numpy arrays to provide rich data structures and data analysis tools. The pandas.DataFrame function provides labelled arrays of (potentially heterogenous) data, similar to the R "data.frame". The pandas.read_csv function can be used to convert a comma-separated values file to a DataFrame object.

patsy is a Python library for describing statistical models and building Design Matrices using R-like formulas.

2.2.2 Data

We download the Guerry dataset, a collection of historical data used in support of Andre-Michel Guerry’s 1833 Essay on the Moral Statistics of France. The data set is hosted online in comma-separated values format (CSV) by the Rdatasets repository. We could download the file locally and then load it using read_csv, but pandas takes care of all of this automatically for us:

```python
In [4]: url = 'http://vincentarelbundock.github.com/Rdatasets/csv/HistData/Guerry.csv'

#the next two lines are not necessary with a recent version of pandas
In [5]: from urllib2 import urlopen
In [6]: url = urlopen(url)
In [7]: df = pandas.read_csv(url)
```

The Input/Output doc page shows how to import from various other formats.

We select the variables of interest and look at the bottom 5 rows:

```python
In [8]: vars = ['Department', 'Lottery', 'Literacy', 'Wealth', 'Region']
In [9]: df = df[vars]
In [10]: df[-5:]
```

<table>
<thead>
<tr>
<th>Department</th>
<th>Lottery</th>
<th>Literacy</th>
<th>Wealth</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vienne</td>
<td>40</td>
<td>25</td>
<td>68</td>
<td>W</td>
</tr>
<tr>
<td>Haute-Vienne</td>
<td>55</td>
<td>13</td>
<td>67</td>
<td>C</td>
</tr>
</tbody>
</table>
2.2.3 Substantive motivation and model

We want to know whether literacy rates in the 86 French departments are associated with per capita wagers on the Royal Lottery in the 1820s. We need to control for the level of wealth in each department, and we also want to include a series of dummy variables on the right-hand side of our regression equation to control for unobserved heterogeneity due to regional effects. The model is estimated using ordinary least squares regression (OLS).

2.2.4 Design matrices (endog & exog)

To fit most of the models covered by statsmodels, you will need to create two design matrices. The first is a matrix of endogenous variable(s) (i.e. dependent, response, regressand, etc.). The second is a matrix of exogenous variable(s) (i.e. independent, predictor, regressor, etc.). The OLS coefficient estimates are calculated as usual:

$$\hat{\beta} = (X'X)^{-1}X'y$$

where $y$ is an $N \times 1$ column of data on lottery wagers per capita ($Lottery$). $X$ is $N \times 7$ with an intercept, the $Literacy$ and $Wealth$ variables, and 4 region binary variables.

The patsy module provides a convenient function to prepare design matrices using R-like formulas. You can find more information here: http://patsy.readthedocs.org

We use patsy’s dmatrices function to create design matrices:

In [13]: y, X = dmatrices(’Lottery ~ Literacy + Wealth + Region’, data=df, return_type=’dataframe’)

The resulting matrices/data frames look like this:

In [14]: y[:3]
Out[14]:
   Lottery
0    41
1    38
2    66

In [15]: X[:3]
Out[15]:
0          1        1          0          0          0      37
Notice that `dmatrices` has
- split the categorical `Region` variable into a set of indicator variables.
- added a constant to the exogenous regressors matrix.
- returned `pandas` DataFrames instead of simple numpy arrays. This is useful because DataFrames allow `statsmodels` to carry-over meta-data (e.g. variable names) when reporting results.

The above behavior can of course be altered. See the `patsy` doc pages.

### 2.2.5 Model fit and summary

Fitting a model in `statsmodels` typically involves 3 easy steps:

1. Use the model class to describe the model
2. Fit the model using a class method
3. Inspect the results using a summary method

For OLS, this is achieved by:

```python
In [16]: mod = sm.OLS(y, X)  # Describe model
In [17]: res = mod.fit()    # Fit model
In [18]: print res.summary() # Summarize model
```

```
OLS Regression Results
==============================================================================
Dep. Variable: Lottery   R-squared: 0.338
Model: OLS              Adj. R-squared: 0.287
Method: Least Squares    F-statistic: 6.636
Date: Sun, 06 Jul 2014   Prob (F-statistic): 1.07e-05
Time: 07:59:53          Log-Likelihood: -375.30
No. Observations: 85    AIC: 764.6
Df Residuals: 78        BIC: 781.7
Df Model: 6             
Covariance Type: nonrobust
==============================================================================
 coef    std err    t     P>|t|      [95.0% Conf. Int.]
-------------------------------------------------------------------------------
Intercept 38.6517  9.456    4.087 0.000   19.826-57.478
Region[T.E] -15.4278 9.727    -1.586 0.117  -34.793  3.938
Region[T.S]  -4.5483 7.279    -0.625 0.534  -19.039  9.943
Region[T.W]  -10.0913 7.196    -1.402 0.165   -24.418  4.235
Literacy     -0.1858  0.210    -0.886 0.378  -0.603  0.232
Wealth       0.4515  0.103    4.390 0.000    0.247 0.656

Omnibus: 3.049  Durbin-Watson: 1.785
Prob(Omnibus): 0.218  Jarque-Bera (JB): 2.694
```

2.2. Getting started
Skew: -0.340  Prob(JB): 0.260
Kurtosis: 2.454  Cond. No. 371.

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

The res object has many useful attributes. For example, we can extract parameter estimates and r-squared by typing:

In [19]: res.params
Out[19]:
Intercept 38.651655
Region[T.E] -15.427785
Region[T.N] -10.016961
Region[T.S] -4.548257
Region[T.W] -10.091276
Literacy -0.185819
Wealth 0.451475
dtype: float64

In [20]: res.rsquared
Out[20]: 0.33795086919288209

Type dir(res) for a full list of attributes.
For more information and examples, see the Regression doc page

2.2.6 Diagnostics and specification tests

statsmodels allows you to conduct a range of useful regression diagnostics and specification tests. For instance, apply the Rainbow test for linearity (the null hypothesis is that the relationship is properly modelled as linear):

In [21]: sm.stats.linear_rainbow(res)
Out[21]: (0.84723399761569096, 0.69979655436216437)

Admittedly, the output produced above is not very verbose, but we know from reading the docstring (also, print sm.stats.linear_rainbow.__doc__) that the first number is an F-statistic and that the second is the p-value.

statsmodels also provides graphics functions. For example, we can draw a plot of partial regression for a set of regressors by:

In [22]: sm.graphics.plot_partregress('Lottery', 'Wealth', ['Region', 'Literacy'],
....:                       data=df, obs_labels=False)
....:
Out[22]: <matplotlib.figure.Figure at 0x64616b0>
2.2.7 More

Congratulations! You’re ready to move on to other topics in the Table of Contents

2.3 Fitting models using R-style formulas

Since version 0.5.0, statsmodels allows users to fit statistical models using R-style formulas. Internally, statsmodels uses the patsy package to convert formulas and data to the matrices that are used in model fitting. The formula framework is quite powerful; this tutorial only scratches the surface. A full description of the formula language can be found in the patsy docs:

- Patsy formula language description

2.3.1 Loading modules and functions

Notice that we called statsmodels.formula.api instead of the usual statsmodels.api. The formula.api hosts many of the same functions found in api (e.g. OLS, GLM), but it also holds lower case counterparts for most of these models. In general, lower case models accept formula and df arguments, whereas upper case ones take endog and exog design matrices. formula accepts a string which describes the model in terms of a patsy formula. df takes a pandas data frame.
dir(smf) will print a list of available models.

Formula-compatible models have the following generic call signature: (formula, data, subset=None, *args, **kwargs)

### 2.3.2 OLS regression using formulas

To begin, we fit the linear model described on the Getting Started page. Download the data, subset columns, and list-wise delete to remove missing observations:

<table>
<thead>
<tr>
<th>Lottery</th>
<th>Literacy</th>
<th>Wealth</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41</td>
<td>37</td>
<td>73</td>
</tr>
<tr>
<td>1</td>
<td>38</td>
<td>51</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>13</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>80</td>
<td>46</td>
<td>76</td>
</tr>
<tr>
<td>4</td>
<td>79</td>
<td>69</td>
<td>83</td>
</tr>
</tbody>
</table>

Fit the model:

```
OLS Regression Results
==============================================================================
Dep. Variable: Lottery             R-squared: 0.338
Model: OLS                            Adj. R-squared: 0.287
Method: Least Squares                F-statistic: 6.636
Date: Sun, 13 Jan 2013                Prob (F-statistic): 0.000
Time: 10:38:36                        Log-Likelihood: -375.30
No. Observations: 85                 AIC: 764.6
Df Residuals: 78                      BIC: 781.7
Df Model: 6                           
===============================================================================
coef    std err          t      P>|t|      [95.0% Conf. Int.]
---------    --------          ------    ------    ----------------------
Intercept    38.6517      9.456       4.087     0.000         19.826    57.478
Region[T.E]  -15.4278      9.727      -1.586     0.117        -34.793     3.938
Region[T.S]  -4.5483      7.279      -0.625     0.534        -19.039     9.943
Region[T.W]  -10.0913      7.196      -1.402     0.165        -24.418     4.235
Literacy     -0.1858      0.210      -0.886     0.378        -0.603     0.232
Wealth       0.4515      0.103       4.390     0.000         0.247     0.656
```

Omnibus: 3.049  Durbin-Watson: 1.785
Prob(Omnibus): 0.218  Jarque-Bera (JB): 2.694
Skew: -0.340  Prob(JB): 0.260
Kurtosis: 2.454  Cond. No. 371.

### 2.3.3 Categorical variables

Looking at the summary printed above, notice that patsy determined that elements of Region were text strings, so it treated Region as a categorical variable. patsy’s default is also to include an intercept, so we automatically dropped one of the Region categories.

If Region had been an integer variable that we wanted to treat explicitly as categorical, we could have done so by using the C() operator:

```
Intercept  38.651655
C(Region)[T.E]  -15.427785
```
C(Region)[T.N]  -10.016961
C(Region)[T.S]  -4.548257
C(Region)[T.W]  -10.091276
Literacy         -0.185819
Wealth           0.451475

Examples more advanced features patsy’s categorical variables function can be found here: Patsy: Contrast Coding Systems for categorical variables

2.3.4 Operators

We have already seen that “~” separates the left-hand side of the model from the right-hand side, and that “+” adds new columns to the design matrix.

Removing variables

The “-” sign can be used to remove columns/variables. For instance, we can remove the intercept from a model by:

C(Region)[C]   38.651655
C(Region)[E]   23.223870
C(Region)[N]   28.634694
C(Region)[S]   34.103399
C(Region)[W]   28.560379
Literacy        -0.185819
Wealth          0.451475

Multiplicative interactions

“::” adds a new column to the design matrix with the product of the other two columns. “*” will also include the individual columns that were multiplied together:

Literacy:Wealth   0.018176
Literacy          0.427386
Wealth            1.080987
Literacy:Wealth   -0.013609

Many other things are possible with operators. Please consult the patsy docs to learn more.

2.3.5 Functions

You can apply vectorized functions to the variables in your model:

Intercept    115.609119
np.log(Literacy) -20.393959

Define a custom function:

Intercept    136.003079
log_plus_1(Literacy) -20.393959
2.3.6 Using formulas with models that do not (yet) support them

Even if a given `statsmodels` function does not support formulas, you can still use `patsy`'s formula language to produce design matrices. Those matrices can then be fed to the fitting function as `endog` and `exog` arguments.

To generate `numpy` arrays:

<table>
<thead>
<tr>
<th>Lottery</th>
<th>Literacy</th>
<th>Wealth</th>
<th>Literacy:Wealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>37</td>
<td>73</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>51</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>13</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>46</td>
<td>76</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>69</td>
<td>83</td>
</tr>
</tbody>
</table>

To generate `pandas` data frames:

<table>
<thead>
<tr>
<th>Lottery</th>
<th>Literacy</th>
<th>Wealth</th>
<th>Literacy:Wealth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>37</td>
<td>73</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>51</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>13</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>46</td>
<td>76</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>69</td>
<td>83</td>
</tr>
</tbody>
</table>

**OLS Regression Results**

| coef     | std err | t   | P>|t|   | [95.0% Conf. Int.] |
|----------|---------|-----|------|------------------|
| Intercept| 38.6348 | 15.825 | 2.441 | 0.017 | 7.149  | 70.121 |
| Literacy | -0.3522 | 0.334 | -1.056 | 0.294 | -1.016 | 0.312 |
| Wealth   | 0.4364  | 0.283 | 1.544  | 0.126 | -0.126 | 0.999 |
| Literacy:Wealth | -0.0005 | 0.006 | -0.085 | 0.933 | -0.013 | 0.012 |

Omnibus: 4.447 Durbin-Watson: 1.953
Prob(Omnibus): 0.108 Jarque-Bera (JB): 3.228
Skew: -0.332 Prob(JB): 0.199
Kurtosis: 2.314 Cond. No. 1.40e+04

---

Chapter 2. Basic Documentation
The condition number is large, $1.4e+04$. This might indicate that there are strong multicollinearity or other numerical problems.

## 2.4 Installation

### 2.4.1 Using setuptools

To obtain the latest released version of statsmodels using setuptools:

```
easy_install -U statsmodels
```

Or follow this link to our PyPI page.

### 2.4.2 Obtaining the Source

We do not release very often but the master branch of our source code is usually fine for everyday use. You can get the latest source from our github repository. Or if you have git installed:

```
git clone git://github.com/statsmodels/statsmodels.git
```

If you want to keep up to date with the source on github just periodically do:

```
git pull
```

in the statsmodels directory.

### 2.4.3 Windows Nightly Binaries

If you are not able to follow the build instructions below, we upload nightly builds of the GitHub repository to http://statsmodels.sourceforge.net/binaries/.

### 2.4.4 Installation from Source

You will need a C compiler installed to build statsmodels. If you are building from the github source and not a source release, then you will also need Cython. You can follow the instructions below to get a C compiler setup for Windows.

**Linux**

Once you have obtained the source, you can do (with appropriate permissions):

```
python setup.py install
```

Or:

```
python setup.py build
python setup.py install
```
Windows

You can build 32-bit version of the code on windows using mingw32.

First, get and install mingw32. Then, you’ll need to edit distutils.cfg. This is usually found somewhere like C:\Python27\lib\distutils\distutils.cfg. Add these lines:

```
[build]
compiler=mingw32
```

Then in the statsmodels directory do:

```
python setup.py build
python setup.py install
```

OR

You can build 32-bit or 64-bit versions of the code using the Microsoft SDK. Detailed instructions can be found on the Cython wiki. The gist of these instructions follow. You will need to download the free Windows SDK C/C++ compiler from Microsoft. You must use the Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1 to be compatible with Python 2.6, 2.7, 3.1, and 3.2. The link for the 3.5 SP1 version is:


For Python 3.3, you need to use the Microsoft Windows SDK for Windows 7 and .NET Framework 4, available from


For 7.0, get the ISO file GRMSDKX_EN_DVD.iso for AMD64. After you install this, open the SDK Command Shell (Start -> All Programs -> Microsoft Windows SDK v7.0 -> CMD Shell). CD to the statsmodels directory and type:

```
set DISTUTILS_USE_SDK=1
```

To build a 64-bit application type:

```
setenv /x64 /release
```

To build a 32-bit application type:

```
setenv /x86 /release
```

The prompt should change colors to green. Then proceed as usual to install:

```
python setup.py build
python setup.py install
```

For 7.1, the instructions are exactly the same, except you use the download link provided above and make sure you are using SDK 7.1.

If you want to accomplish the same without opening up the SDK CMD SHELL, then you can use these commands at the CMD Prompt or in a batch file:

```
setlocal EnableDelayedExpansion
CALL "C:\Program Files\Microsoft SDKs\Windows\v7.0\Bin\SetEnv.cmd" /x64 /release
set DISTUTILS_USE_SDK=1
```

Replace /x64 with /x86 and v7.0 with v7.1 as needed.
2.4.5 Dependencies

- Python >= 2.6, including Python 3.x
- NumPy >= 1.5.0
- SciPy >= 0.7
- Pandas >= 0.7.1
- Patsy >= 0.1.0
- Cython >= 20.1, Needed if you want to build the code from github and not a source distribution. You must use Cython >= 0.20.1 if you’re on Python 3.4. Earlier versions may work for Python < 3.4.

2.4.6 Optional Dependencies

- Matplotlib is needed for plotting functions and running many of the examples.
- Nose is required to run the test suite.
- IPython is required to build the docs locally.

2.5 Related Packages

These are some python packages that have a related purpose and can be useful in combination with statsmodels. The selection in this list is biased towards packages that might be directly useful for data handling and statistical analysis, and towards those that have a BSD compatible license, which implies that we are not restricted in looking at the source to learn of different ways of implementation or of different algorithms. The following descriptions are taken from the websites with small adjustments.

2.5.1 Data Handling

Pandas

https://pypi.python.org/pypi/pandas

“This project aims to provide the following

- A set of fast NumPy-based data structures optimized for panel, time series, and cross-sectional data analysis.
- A set of tools for loading such data from various sources and providing efficient ways to persist the data.
- A robust statistics and econometrics library which closely integrates with the core data structures.”

License: New BSD Language: Python, Cython, binary distribution available for win32-py25, but easy to build with MinGW

Comments

Uses statsmodels as optional dependency for statistical analysis, but has additional statistical and econometrics algorithms that focus on panel data analysis, mostly in the time dimension. It has several data structures that allow dictionary access to the underlying 1, 2, or 3 dimensional arrays. It was initially focused on a two-dimensional representation of the data, but now also allows for different representation of three-dimensional arrays. It allows for arbitrary axis labels, but offers also a convenient time series class.
Tabular

https://pypi.python.org/pypi/tabular

“Tabular data container and associated convenience routines in Python

Tabular is a package of Python modules for working with tabular data. Its main object is the tabarray class, a data structure for holding and manipulating tabular data.

The tabarray object is based on the ndarray object from the Numerical Python package (NumPy), and the Tabular package is built to interface well with NumPy in general. “

License: MIT Language: Python

Comments

Uses numpys structured arrays as basic building block. Focused on spreadsheet-style operations for working with two-dimensional tables and associated data handling and analysis. It is instructive to read the code of tabular for working with structured arrays.

La

https://pypi.python.org/pypi/la

“Label the rows, columns, any dimension, of your NumPy arrays.

The main class of the la package is a labeled array, larry. A larry consists of a data array and a label list. The data array is stored as a NumPy array and the label list as a list of lists. “

License: BSD Language: Python

Comments

The data handling is in intention similar to pandas but closer to working with standard numpy ndarrays. The main addition to numpy arrays are arbitrary labels for each axis of the array. Larry delegates to numpy functions but does not subclass numpy’s ndarrays. It also provides functions for basic descriptive statistics.

2.5.2 Data Analysis

Pymc

https://pypi.python.org/pypi/pymc

“Bayesian estimation, particularly using Markov chain Monte Carlo (MCMC), is an increasingly relevant approach to statistical estimation. PyMC is a python module that implements the Metropolis-Hastings algorithm as a python class, and is extremely flexible and applicable to a large suite of problems.”

License: MIT, Academic Free License (?) Language: Python, C, Fortran binary (bundle ?) installer

Comments This is to some extent the modern Bayesian analog of statsmodels. It is by far the most mature project in this group including statsmodels.

Scikits.talkbox

https://pypi.python.org/pypi/scikits.talkbox

Talkbox is set of python modules for speech/signal processing. The goal of this toolbox is to be a sandbox for features which may end up in scipy at some point.
2.5. Related Packages

2.5.3 Domain-specific Data Analysis

The following packages contain interesting statistical algorithms, however they are tightly focused on their application, and are or might be more difficult to use “from the outside”. (Descriptions are taken from websites)

**Pymvpa**

PyMVPA is a Python module intended to ease pattern classification analyses of large datasets [http://www.pymvpa.org/](http://www.pymvpa.org/)

**Nitime**

[https://github.com/fperez/nitime](https://github.com/fperez/nitime)

“Nitime is a library for time-series analysis of data from neuroscience experiments.

It contains a core of numerical algorithms for time-series analysis both in the time and spectral domains, a set of container objects to represent time-series, and auxiliary objects that expose a high level interface to the numerical machinery and make common analysis tasks easy to express with compact and semantically clear code.”

License: BSD Language: Python

**KF - Kalman Filter**

[https://pypi.python.org/pypi/KF](https://pypi.python.org/pypi/KF)

“This project was started to test different available tools to track mutual funds and hedge fund using Capital Asset Pricing Model (CAPM thereafter) introduced my Sharpe and Arbitrage Pricing Theory (APT thereafter) introduced by Ross.

- License : BSD -check
- Language Python (requires cvxopt)

**Comments** Athough focused on neuroscience, the algorithms for time series analysis are independent of the data representation and can be used with numpy arrays. Current focus is on spectral analysis including coherence between several time series.

**scikit-learn**

Nipy

Nipy aims to provide a complete Python environment for the analysis of structural and functional neuroimaging data
http://nipy.sourceforge.net/ License: BSD

Biopython

Biopython is a set of tools for biological computation http://biopython.org/wiki/Main_Page License:
http://www.biopython.org/DIST/LICENSE similar to MIT (??)

Pysal

A library for exploratory spatial analysis and geocomputation http://code.google.com/p/pysal/ License: BSD

glu-genetics

A broad array of tools to store, clean, and analyze data generated by whole-genome or candidate gene association
scans. http://code.google.com/p/glu-genetics/ License: BSD

2.5.4 Other packages

There exists a large number of machine learning packages in python, many of them with a well established code
base. Unfortunately, none of the packages with a wider coverage of algorithms has a scipy compatible license. A list-
ing can be found at http://mloss.org/software/language/python/. Other packages are available that provide additional
functionality, especially openopt which offers additional optimization routines compared to the ones in scipy.

2.6 About Statsmodels

2.6.1 Background

The models module of scipy.stats was originally written by Jonathan Taylor. For some time it was part of scipy
but was later removed. During the Google Summer of Code 2009, statsmodels was corrected, tested, improved
and released as a new package. Since then, the statsmodels development team has continued to add new models,
plotting tools, and statistical methods.

2.6.2 Testing

Most results have been verified with at least one other statistical package: R, Stata or SAS. The guiding principal for
the initial rewrite and for continued development is that all numbers have to be verified. Some statistical methods are
tested with Monte Carlo studies. While we strive to follow this test driven approach, there is no guarantee that the code
is bug-free and always works. Some auxiliary function are still insufficiently tested, some edge cases might not be
correctly taken into account, and the possibility of numerical problems is inherent to many of the statistical models. We
especially appreciate any help and reports for these kind of problems so we can keep improving the existing models.
Code Stability

The existing models are mostly settled in their user interface and we do not expect many large changes going forward. For the existing code, although there is no guarantee yet on API stability, we have long deprecation periods in all but very special cases, and we try to keep changes that require adjustments by existing users to a minimal level. For newer models we might adjust the user interface as we gain more experience and obtain feedback. These changes will always be noted in our release notes available in the documentation.

2.6.3 Financial Support

We are grateful for the financial support that we obtained for the development of statsmodels:

- AQR www.aqr.com: financial sponsor for the work on Vector Autoregressive Models (VAR) by Wes McKinney

We would also like to thank our hosting providers, github for the public code repository, sourceforge for hosting our documentation and python.org for making our downloads available on PyPi.

Information about the structure and development of statsmodels:

2.7 endog, exog, what’s that?

Statsmodels is using endog and exog as names for the data, the observed variables that are used in an estimation problem. Other names that are often used in different statistical packages or text books are, for example,

<table>
<thead>
<tr>
<th>endog</th>
<th>exog</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>y variable</td>
<td>x variable</td>
</tr>
<tr>
<td>left hand side (LHS)</td>
<td>right hand side (RHS)</td>
</tr>
<tr>
<td>dependent variable</td>
<td>independent variable</td>
</tr>
<tr>
<td>regressand</td>
<td>regressors</td>
</tr>
<tr>
<td>outcome</td>
<td>design</td>
</tr>
<tr>
<td>response variable</td>
<td>explanatory variable</td>
</tr>
</tbody>
</table>

The usage is quite often domain and model specific; however, we have chosen to use endog and exog almost exclusively. A mnemonic hint to keep the two terms apart is that exogenous has an “x”, as in x-variable, in it’s name.

x and y are one letter names that are sometimes used for temporary variables and are not informative in itself. To avoid one letter names we decided to use descriptive names and settled on endog and exog. Since this has been criticized, this might change in future.

2.7.1 Background

Some informal definitions of the terms are

endogenous: caused by factors within the system

exogenous: caused by factors outside the system

Endogenous variables designates variables in an economic/econometric model that are explained, or predicted, by that model. http://stats.oecd.org/glossary/detail.asp?ID=794

Exogenous variables designates variables that appear in an economic/econometric model, but are not explained by that model (i.e. they are taken as given by the model). http://stats.oecd.org/glossary/detail.asp?ID=890

2.7. endog, exog, what’s that?
In econometrics and statistics the terms are defined more formally, and different definitions of exogeneity (weak, strong, strict) are used depending on the model. The usage in statsmodels as variable names cannot always be interpreted in a formal sense, but tries to follow the same principle.

In the simplest form, a model relates an observed variable, y, to another set of variables, x, in some linear or nonlinear form

\[ y = f(x, \beta) + \text{noise} \]
\[ y = x \ast \beta + \text{noise} \]

However, to have a statistical model we need additional assumptions on the properties of the explanatory variables, x, and the noise. One standard assumption for many basic models is that x is not correlated with the noise. In a more general definition, x being exogenous means that we do not have to consider how the explanatory variables in x were generated, whether by design or by random draws from some underlying distribution, when we want to estimate the effect or impact that x has on y, or test a hypothesis about this effect.

In other words, y is exogenous to our model, x is exogenous to our model for the estimation.

As an example, suppose you run an experiment and for the second session some subjects are not available anymore. Is the drop-out relevant for the conclusions you draw for the experiment? In other words, can we treat the drop-out decision as exogenous for our problem.

It is up to the user to know (or to consult a text book to find out) what the underlying statistical assumptions for the models are. As an example, exog in OLS can have lagged dependent variables if the error or noise term is independently distributed over time (or uncorrelated over time). However, if the error terms are autocorrelated, then OLS does not have good statistical properties (is inconsistent) and the correct model will be ARMAX. statsmodels has functions for regression diagnostics to test whether some of the assumptions are justified or not.

2.8 Import Paths and Structure

We offer two ways of importing functions and classes from statsmodels:

1. API import for interactive use
   - Allows tab completion
2. Direct import for programs
   - Avoids importing unnecessary modules and commands

2.8.1 API Import for interactive use

For interactive use the recommended import is:

```python
import statsmodels.api as sm
```

Importing `statsmodels.api` will load most of the public parts of statsmodels. This makes most functions and classes conveniently available within one or two levels, without making the “sm” namespace too crowded.

To see what functions and classes available, you can type the following (or use the namespace exploration features of IPython, Spyder, IDLE, etc.):

```python
>>> dir(sm)
```
>>> dir(sm.graphics)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'abline_plot', 'beanplot', 'fboxplot', 'interaction_plot', 'qqplot',
'rainbow', 'rainbowplot', 'violinplot']

>>> dir(sm.tsa)
['AR', 'ARMA', 'DynamicVAR', 'SVAR', 'VAR', '__builtins__', '__doc__',
'__file__', '__name__', '__package__', 'acf', 'acovf', 'add_lag',
'add_trend', 'adfuller', 'ccf', 'ccovf', 'datetools', 'detrend',
'filters', 'grangercausalitytests', 'interp', 'lagmat', 'lagmat2ds',
'pacf', 'pacf_ols', 'pacf_yw', 'periodogram', 'q_stat', 'stattools',
'tsatoools', 'var']

Notes

The api modules may not include all the public functionality of statsmodels. If you find something that should be added to the api, please file an issue on github or report it to the mailing list.

The subpackages of statsmodels include api.py modules that are mainly intended to collect the imports needed for those subpackages. The subpackage/api.py files are imported into statsmodels api, for example

```
from .nonparametric import api as nonparametric
```

Users do not need to load the subpackage/api.py modules directly.

2.8.2 Direct import for programs

statsmodels submodules are arranged by topic (e.g. discrete for discrete choice models, or tsa for time series analysis). Our directory tree (stripped down) looks something like this:

```
statsmodels/
    __init__.py
    api.py
    discrete/
        __init__.py
        discrete_model.py
        tests/
            results/
    tsa/
        __init__.py
        api.py
        tsatoools.py
        stattools.py
        arima_model.py
        arima_process.py
        vector_ar/
            __init__.py
            var_model.py
            tests/
                results/
                tests/
                results/
    stats/
        __init__.py
        api.py
        stattools.py
```

2.8. Import Paths and Structure 37
The submodules that can be import heavy contain an empty `__init__.py`, except for some testing code for running tests for the submodules. The intention is to change all directories to have an `api.py` and empty `__init__.py` in the next release.

**Import examples**

Functions and classes:

```python
from statsmodels.regression.linear_model import OLS, WLS
from statsmodels.tools.tools import rank, add_constant
```

Modules

```python
from statsmodels.datasets import macrodata
import statsmodels.stats import diagnostic
```

Modules with aliases

```python
import statsmodels.regression.linear_model as lm
import statsmodels.stats.diagnostic as smsdia
import statsmodels.stats.outliers_influence as oi
```

We do not have currently a convention for aliases of submodules.

## 2.9 Pitfalls

This page lists issues which may arise while using statsmodels. These can be the result of data-related or statistical problems, software design, “non-standard” use of models, or edge cases.

statsmodels provides several warnings and helper functions for diagnostic checking (see this blog article for an example of misspecification checks in linear regression). The coverage is of course not comprehensive, but more warnings and diagnostic functions will be added over time.

While the underlying statistical problems are the same for all statistical packages, software implementations differ in the way extreme or corner cases are handled. Please report corner cases for which the models might not work, so we can treat them appropriately.

### 2.9.1 Repeated calls to fit with different parameters

Result instances often need to access attributes from the corresponding model instance. Fitting a model multiple times with different arguments can change model attributes. This means that the result instance may no longer point to the correct model attributes after the model has been re-fit.

It is therefore best practice to create separate model instances when we want to fit a model using different fit function arguments.

For example, this works without problem because we are not keeping the results instance for further use.
```python
mod = AR(endog)
aic = []
for lag in range(1,11):
    res = mod.fit(maxlag=lag)
    aic.append(res.aic)
```

However, when we want to hold on to two different estimation results, then it is recommended to create two separate model instances.

```python
mod1 = RLM(endog, exog)
res1 = mod1.fit(scale_est='mad')
mod2 = RLM(endog, exog)
res2 = mod2.fit(scale_est='stand_mad')
```

### 2.9.2 Unidentified Parameters

**Rank deficient exog, perfect multicollinearity**

Models based on linear models, GLS, RLM, GLM and similar, use a generalized inverse. This means that:

- Rank deficient matrices will not raise an error
- Cases of almost perfect multicollinearity or ill-conditioned design matrices might produce numerically unstable results. Users need to manually check the rank or condition number of the matrix if this is not the desired behavior

Note: Statsmodels currently fails on the NIST benchmark case for Filip if the data is not rescaled, see [this blog](https://example.com)

**Incomplete convergence in maximum likelihood estimation**

In some cases, the maximum likelihood estimator might not exist, parameters might be infinite or not unique (e.g. (quasi-)separation in models with binary endogenous variable). Under the default settings, statsmodels will print a warning if the optimization algorithm stops without reaching convergence. However, it is important to know that the convergence criteria may sometimes falsely indicate convergence (e.g. if the value of the objective function converged but not the parameters). In general, a user needs to verify convergence.

For binary Logit and Probit models, statsmodels raises an exception if perfect prediction is detected. There is, however, no check for quasi-perfect prediction.

### 2.9.3 Other Problems

**Insufficient variation in the data**

It is possible that there is insufficient variation in the data for small datasets or for data with small groups in categorical variables. In these cases, the results might not be identified or some hidden problems might occur.

The only currently known case is a perfect fit in robust linear model estimation. For RLM, if residuals are equal to zero, then it does not cause an exception, but having this perfect fit can produce NaNs in some results (scale=0 and 0/0 division) (issue #55).
2.10 Developer Page

This page explains how you can contribute to the development of statsmodels by submitting patches, statistical tests, new models, or examples. statsmodels is developed on Github using the Git version control system.

2.10.1 Submitting a Bug Report

- Include a short, self-contained code snippet that reproduces the problem
- Specify the statsmodels version used. You can do this with `sm.version.full_version`
- If the issue looks to involve other dependencies, also include the output of `sm.show_versions()`

2.10.2 Making Changes to the Code

For a pull request to be accepted, you must meet the below requirements. This greatly helps the job of maintaining and releasing the software a shared effort.

- **One branch. One feature.** Branches are cheap and github makes it easy to merge and delete branches with a few clicks. Avoid the temptation to lump in a bunch of unrelated changes when working on a feature, if possible. This helps us keep track of what has changed when preparing a release.

- Commit messages should be clear and concise. This means a subject line of less than 80 characters, and, if necessary, a blank line followed by a commit message body. We have an informal commit format standard that we try to adhere to. You can see what this looks like in practice by `git log --oneline -n 10`. If your commit references or closes a specific issue, you can close it by mentioning it in the commit message. *(For maintainers: These suggestions go for Merge commit comments too. These are partially the record for release notes.)*

- Code submissions must always include tests. See our notes on Testing.

- Each function, class, method, and attribute needs to be documented using docstrings. We conform to the numpy docstring standard.

- If you are adding new functionality, you need to add it to the documentation by editing (or creating) the appropriate file in `docs/source`.

- Make sure your documentation changes parse correctly. Change into the top-level `docs/` directory and type:

```
make clean
make html
```

Check that the build output does not have any warnings due to your changes.

- Finally, please add your changes to the release notes. Open the `docs/source/release/versionX.X.rst` file that has the version number of the next release and add your changes to the appropriate section.

2.10.3 How to Submit a Pull Request

So you want to submit a patch to statsmodels but aren’t too familiar with github? Here are the steps you need to take.

1. Fork the statsmodels repository on Github.
2. Create a new feature branch. Each branch must be self-contained, with a single new feature or bugfix.
3. Make sure the test suite passes. This includes testing on Python 3. The easiest way to do this is to either enable Travis-CI on your fork, or to make a pull request and check there.

4. Submit a pull request

2.10.4 Mailing List

Conversations about development take place on the statsmodels mailing list.

2.10.5 License

Statsmodels is released under the Modified (3-clause) BSD license.

2.10.6 Contents

Working with the Statsmodels Code

Github

The statsmodels code base is hosted on Github. To contribute you will need to sign up for a free Github account.

Version Control and Git

We use the Git version control system for development. Git allows many people to work together on the same project. In a nutshell, it allows you to make changes to the code independent of others who may also be working on the code and allows you to easily contribute your changes to the codebase. It also keeps a complete history of all changes to the code, so you can easily undo changes or see when a change was made, by whom, and why.

To install and configure Git, and to setup SSH keys, see this page.

To learn more about Git, you may want to visit:

- Git documentation (book and videos)
- Github help pages
- NumPy documentation
- Matthew Brett’s Pydagogue

Below, we describe the bare minimum git commands you need to contribute to statsmodels.

Statsmodels Git/Github Workflow

Forking and cloning After setting up git, you need to fork the main statsmodels repository. To do this, visit the statsmodels project page and hit the fork button (see this page for details). This should take you to your fork’s page.

Then, you want to clone the fork to your machine:

```
git clone git@github.com:your-user-name/statsmodels.git statsmodels-yourname
cd statsmodels-yourname
git remote add upstream git://github.com/statsmodels/statsmodels.git
```
The first line creates a directory named `statsmodels-yourname`. The third line sets-up a read-only connection to the upstream statsmodels repository. This will allow you to periodically update your local code with changes in the upstream.

**Create a Branch**  All changes to the code should be made in a feature branch. To create a branch, type:

```bash
git branch shiny-new-feature
git checkout shiny-new-feature
```

Doing:

```bash
git branch
```

will give something like:

```
* shiny-new-feature
 master
```

to indicate that you are now on the `shiny-new-feature` branch.

**Making changes**  Hack away! Make any changes that you want, but please keep the work in your branch completely confined to one specific topic, bugfix, or feature implementation. You can work across multiple files and have many commits, but the changes should all be related to the feature of the feature branch, whatever that may be.

Now imagine that you changed the file `foo.py`. You can see your changes by typing:

```bash
git status
```

This will print something like:

```
# On branch shiny-new-feature
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#   modified:   relative/path/to/foo.py
#
# no changes added to commit (use "git add" and/or "git commit -a")
```

Before you can commit these changes, you have to `add`, or `stage`, the changes. You can do this by typing:

```bash
git add path/to/foo.py
```

Then check the status to make sure your commit looks okay:

```bash
git status
```

should give something like:

```
# On branch shiny-new-feature
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   /relative/path/to/foo.py
#```

Chapter 2. Basic Documentation
Pushing your changes  At any time you can push your feature branch (and any changes) to your github (fork) repository by:

git push origin shiny-new-feature

Here origin is the default name given to your remote repository. You can see the remote repositories by:

git remote -v

If you added the upstream repository as described above you will see something like:

origin  git@github.com:yourname/statsmodels.git (fetch)
orign  git@github.com:yourname/statsmodels.git (push)
upstream  git://github.com/statsmodels/statsmodels.git (fetch)
upstream  git://github.com/statsmodels/statsmodels.git (push)

Before you push any commits, however, it is highly recommended that you make sure what you are pushing makes sense and looks clean. You can review your change history by:

git log --oneline --graph

It pays to take care of things locally before you push them to github. So when in doubt, don’t push. Also see the advice on keeping your history clean in Merging vs. Rebasing.

Pull Requests  When you are ready to ask for a code review, we recommend that you file a pull request. Before you do so you should check your changeset yourself. You can do this by using compare view on github.

1. Navigate to your repository on github.
2. Click on Branch List
3. Click on the Compare button for your feature branch, shiny-new-feature.
4. Select the base and compare branches, if necessary. This will be master and shiny-new-feature, respectively.
5. From here you will see a nice overview of your changes. If anything is amiss, you can fix it.

If everything looks good you are read to make a pull request.

1. Navigate to your repository on github.
2. Click on the Pull Request button.
3. You can then click on Commits and Files Changed to make sure everything looks okay one last time.
4. Write a description of your changes in the Preview Discussion tab.
5. Click Send Pull Request.

Your request will then be reviewed. If you need to go back and make more changes, you can make them in your branch and push them to github and the pull request will be automatically updated.

One last thing to note. If there has been a lot of work in upstream/master since you started your patch, you might want to rebase. However, you can probably get away with not rebasing if these changes are unrelated to the work you have done in the shiny-new-feature branch. If you can avoid it, then don’t rebase. If you have to, try to do it once and when you are at the end of your changes. Read on for some notes on Merging vs. Rebasing.

Advanced Topics

Merging vs. Rebasing  This is a topic that has been discussed at great length and with considerable more expertise than we can offer here. This section will provide some resources for further reading and some advice. The focus,
though, will be for those who wish to submit pull requests for a feature branch. For these cases rebase should be preferred.

A rebase replays commits from one branch on top of another branch to preserve a linear history. Recall that your commits were tested against (possibly) older version of master from which you started your branch, so if you rebase, you could introduce bugs. However, if you have only a few commits, this might not be such a concern. One great place to start learning about rebase is *rebasings without tears*. In particular, heed the warnings. Namely, **always make a new branch before doing a rebase**. This is good general advice for working with git. I would also add **never use rebase on work that has already been published**. If another developer is using your work, don’t rebase!!

As for merging, **never merge from trunk into your feature branch**. You will, however, want to check that your work will merge cleanly into trunk. This will help out the reviewers. You can do this in your local repository by merging your work into your master (or any branch that tracks remote master) and *Running the Test Suite*.

**Deleting Branches**  Once your feature branch is accepted into upstream, you might want to get rid of it. First you’ll want to merge upstream master into your branch. That way git will know that it can safely delete your branch:

```bash
git fetch upstream
git checkout master
git merge upstream/master
```

Then you can just do:

```bash
git branch -d shiny-new-feature
```

Make sure you use a lower-case `-d`. That way, git will complain if your feature branch has not actually been merged. The branch will still exist on github however. To delete the branch on github, do:

```bash
git push origin :shiny-new-feature branch
```

**Git for Bzr Users**

```bash
git pull != bzr pull

git pull = git fetch + git merge
```

Of course, you could:

```bash
git pull --rebase = git fetch + git rebase

git merge != bzr merge

git merge == bzr merge + bzr commit

git merge --no-commit == bzr merge
```

**Maintainer Notes**

This is for those with read-write access to upstream. It is recommended to name the upstream remote something to remind you that it is read-write:

```bash
git remote add upstream-rw git@github.com:statsmodels/statsmodels.git

git fetch upstream-rw
```

**Git Workflow**

**Grabbing Changes from Others**  If you need to push changes from others, you can link to their repository by doing:
The rest of the below assumes you are on your or someone else’s branch with the changes you want to push upstream.

**Rebasing**  If there are only a few commits, you can rebase to keep a linear history:

```
git fetch upstream-rw
git rebase upstream-rw/master
```

Rebasing will not automatically close the pull request however, if there is one, so don’t forget to do this.

**Merging**  If there is a long series of related commits, then you’ll want to merge. You may ask yourself, *Merging: To Fast-Forward or Not To Fast-Forward?* See below for more on this choice. Once decided you can do:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

Merging will automatically close the pull request on github.

**Check the History**  This is very important. Again, any and all fixes should be made locally before pushing to the repository:

```
git log --oneline --graph
```

This shows the history in a compact way of the current branch. This:

```
git log -p upstream-rw/master..
```

shows the log of commits excluding those that can be reached from upstream-rw/master, and including those that can be reached from current HEAD. That is, those changes unique to this branch versus upstream-rw/master. See *Pydagogue* for more on using dots with log and also for using *dots with diff*.

**Push Your Feature Branch**  All the changes look good? You can push your feature branch after *Merging or Rebasing* by:

```
git push upstream-rw shiny-new-feature:master
```

**Cherry-Picking**  Say you are interested in some commit in another branch, but want to leave the other ones for now. You can do this with a cherry-pick. Use `git log --oneline` to find the commit that you want to cherry-pick. Say you want commit `dd9ff35` from the `shiny-new-feature` branch. You want to apply this commit to master. You simply do:

```
git checkout master
git cherry-pick dd9ff35
```

And that’s all. This commit is now applied as a new commit in master.

**Merging: To Fast-Forward or Not To Fast-Forward**  By default, `git merge` is a fast-forward merge. What does this mean, and when do you want to avoid this?

The fast-forward merge does not create a merge commit. This means that the existence of the feature branch is lost in the history. The fast-forward is the default for Git basically because branches are cheap and, therefore, usually
Figure 2.1: (source nvie.com, post “A successful Git branching model”)
short-lived. If on the other hand, you have a long-lived feature branch or are following an iterative workflow on the feature branch (i.e. merge into master, then go back to feature branch and add more commits), then it makes sense to include only the merge in the main branch, rather than all the intermediate commits of the feature branch, so you should use:

git merge --no-ff

**Handling Pull Requests** You can apply a pull request through `fetch` and `merge`. In your local copy of the main repo:

git checkout master

git remote add contrib-name git://github.com/contrib-name/statsmodels.git

git fetch contrib-name

git merge contrib-name/shiny-new-feature

Check that the merge applies cleanly and the history looks good. Edit the merge message. Add a short explanation of what the branch did along with a ‘Closes gh-XXX’ string. This will auto-close the pull request and link the ticket and closing commit. To automatically close the issue, you can use any of:

gh-XXX
GH-XXX
#XXX

in the commit message. Any and all problems need to be taken care of locally before doing:

git push origin master

**Releasing**

1. Fix the version number. Open `setup.py` and set:

   ```python
   ISRELEASED = True
   ```

2. Clean the working tree with:

   ```bash
   git clean -xdf
   ```

   But you might want to do a dry-run first:

   ```bash
   git clean -xdfn
   ```

3. Tag the release. For a release candidate, for example:

   ```bash
   git tag -a v0.3.0rc1 -m "Version 0.3.0 Release Candidate 1" 7b2fb29
   ```

4. If on a new minor release (major.minor.micro format) start a new maintenance branch, for example:

   ```bash
   git checkout -b maintenance/0.3.x
   ```

   Any bug fixes and maintenance commits intended for the next micro release should be made against master as usual, but tagged with the milestone for the micro release it is intended for. Then merge into master as usual. When ready to do the backports, use the file `tools/backport_pr.py` to identify which PRs need to be backported and to apply them to the maintenance branch. The tag for the release should be made in the maintenance branch.

5. Upload the source distribution to PyPI:

   ```bash
   python setup.py sdist --formats=gztar,zip register upload
   ```

6. Go back to `setup.py` and set `isreleased = False` and bump the major version in master.
7. Update the version numbers in the statsmodels/statsmodels-website repo. These are in conf.py. Also upload the released version docs to stable/.

8. Make an announcement

9. Profit

Commit Comments

Prefix commit messages in the master branch of the main shared repository with the following:

ENH: Feature implementation
BUG: Bug fix
STY: Coding style changes (indenting, braces, code cleanup)
DOC: Sphinx documentation, docstring, or comment changes
CMP: Compiled code issues, regenerating C code with Cython, etc.
REL: Release related commit
TST: Change to a test, adding a test. Only used if not directly related to a bug.
REF: Refactoring changes

vbench

See vbenchdoc notes for working with the statsmodels vbench suite. The github repository is available here.

Testing

Test Driven Development

We strive to follow a Test Driven Development (TDD) pattern. All models or statistical functions that are added to the main code base are to have tests versus an existing statistical package, if possible.

Introduction to Nose

Like many packages, statsmodels uses the Nose testing system and the convenient extensions in numpy.testing. Nose itself is an extension of Python’s unittest. Nose will find any file, directory, function, or class name that matches the regular expression (?:^(?:[b_./-])?[Tt]est. This is mainly functions that begin with test* and classes that begin with Test*.

Running the Test Suite

You can run all the tests by:

```python
>>> import statsmodels.api as sm
>>> sm.test()
```

You can test submodules by:

```python
>>> sm.discrete.test()
```
How To Write A Test

NumPy provides a good introduction to unit testing with Nose and NumPy extensions here. It is worth a read for some more details. Here, we will document a few conventions we follow that are worth mentioning. Often we want to test a whole model at once rather than just one function, for example. The following is a pared down version test_discrete.py. In this case, several different models with different options need to be tested. The tests look something like

```python
from numpy.testing import assert_almost_equal
import statsmodels.api as sm
from results.results_discrete import Spector

class CheckDiscreteResults(object):
    ""
    res2 are the results. res1 are the values from statsmodels
    ""

def test_params(self):
    assert_almost_equal(self.res1.params, self.res2.params, 4)

def test_tvalues(self):
    decimal_tvalues = 4
    assert_almost_equal(self.res1.params, self.res2.params, self.decimal_tvalues)

# ... as many more tests as there are common results

class TestProbitNewton(CheckDiscreteResults):
    ""
    Tests the Probit model using Newton’s method for fitting.
    ""

    @classmethod
    def setupClass(cls):
        data = sm.datasets.spector.load()
        data.exog = sm.add_constant(data.exog)
        cls.res1 = sm.Probit(data.endog, data.exog).fit(method='newton', disp=0)

        # setup results
        res2 = Spector()
        res2.probit()
        cls.res2 = res2

        # setup precision
        cls.decimal_tvalues = 3

    def test_model_specific(self):
        assert_almost_equal(self.res1.foo, self.res2.foo, 4)

if __name__ == '__main__':
    import nose
    nose.runmodule(argv=['-vvs', '-x', '--pdb'], exit=False)
```

The main workhorse is the `CheckDiscreteResults` class. Notice that we can set the level of precision for `tvalues` to be different than the default in the subclass `TestProbitNewton`. All of the test classes have a `setupClass` classmethod. Otherwise, Nose would reinstantiate the class before every single test method. If the fitting of the model is time consuming, then this is clearly undesirable. Finally, we have a script at the bottom so that we can run the tests should be running the Python file.
Test Results

The test results are the final piece of the above example. For many tests, especially those for the models, there are many results against which you would like to test. It makes sense then to separate the hard-coded results from the actual tests to make the tests more readable. If there are only a few results it’s not necessary to separate the results. We often take results from some other statistical package. It is important to document where you got the results from and why they might differ from the results that we get. Each tests folder has a results subdirectory. Consider the folder structure for the discrete models:

```
tests/
  __init__.py
  test_discrete.py
results/
  __init__.py
  results_discrete.py
  nbinom_resids.csv
```

It is up to you how best to structure the results. In the discrete model example, you will notice that there are result classes based around particular datasets with a method for loading different model results for that dataset. You can also include text files that hold results to be loaded by results classes if it is easier than putting them in the class itself.

Naming Conventions

File and Directory Names

Our directory tree stripped down looks something like:

```
statsmodels/
  __init__.py
  api.py
  discrete/
    __init__.py
    discrete_model.py
  tests/
    results/
  tsa/
    __init__.py
    api.py
    tsatools.py
    stattools.py
    arima_model.py
    arima_process.py
    vector_ar/
      __init__.py
      var_model.py
    tests/
      results/
  tests/
    results/
stats/
  __init__.py
  api.py
  stattools.py
  tests/
  tools/
    __init__.py
```
The submodules are arranged by topic, `discrete` for discrete choice models, or `tsa` for time series analysis. The submodules that can be import heavy contain an empty `__init__.py`, except for some testing code for running tests for the submodules. The namespace to be imported in is `api.py`. That way, we can import selectively and not have to import a lot of code that we don’t need. Helper functions are usually put in files named `tools.py` and statistical functions, such as statistical tests are placed in `stattools.py`. Everything has directories for `tests`.

### endog & exog

Our working definition of a statistical model is an object that has both endogenous and exogenous data defined as well as a statistical relationship. In place of endogenous and exogenous one can often substitute the terms left hand side (LHS) and right hand side (RHS), dependent and independent variables, regressand and regressors, outcome and design, response variable and explanatory variable, respectively. The usage is quite often domain specific; however, we have chosen to use `endog` and `exog` almost exclusively, since the principal developers of statsmodels have a background in econometrics, and this feels most natural. This means that all of the models are objects with `endog` and `exog` defined, though in some cases `exog` is None for convenience (for instance, with an autoregressive process). Each object also defines a `fit` (or similar) method that returns a model-specific results object. In addition there are some functions, e.g. for statistical tests or convenience functions.

See also the related explanation in `endog, exog, what’s that?`.

### Variable Names

All of our models assume that data is arranged with variables in columns. Thus, internally the data is all 2d arrays. By convention, we will prepend a `k_` to variable names that indicate moving over axis 1 (columns), and `n_` to variables that indicate moving over axis 0 (rows). The main exception to the underscore is that `nobs` should indicate the number of observations. For example, in the time-series ARMA model we have:

- `'k_ar'` - The number of AR lags included in the RHS variables
- `'k_ma'` - The number of MA lags included in the RHS variables
- `'k_trend'` - The number of trend variables included in the RHS variables
- `'k_exog'` - The number of exogenous variables included in the RHS variables excluding the trend terms
- `'n_totobs'` - The total number of observations for the LHS variables including the pre-sample values

### Options

We are using similar options in many classes, methods and functions. They should follow a standardized pattern if they recur frequently.

- `'missing' ['none', 'drop', 'raise']` define whether inputs are checked for nans, and how they are treated
- `'alpha'` (float in (0, 1)) significance level for hypothesis tests and confidence intervals, e.g. `'alpha=0.05'`

patterns

- `'return_XXX'`: boolean to indicate optional or different returns
  (not `'ret_XXX'`)
Datasets

For a list of currently available datasets and usage instructions, see the datasets page.

License

To be considered for inclusion in statsmodels, a dataset must be in the public domain, distributed under a BSD-compatible license, or we must obtain permission from the original author.

Adding a dataset: An example

The Nile River data measures the volume of the discharge of the Nile River at Aswan for the years 1871 to 1970. The data are copied from the paper of Cobb (1978).

Step 1: Create a directory datasets/nile/

Step 2: Add datasets/nile/nile.csv and a new file datasets/__init__.py which contains

```python
from data import *
```

Step 3: If nile.csv is a transformed/cleaned version of the original data, create a nile/src directory and include the original raw data there. In the nile case, this step is not necessary.

Step 4: Copy datasets/template_data.py to nile/data.py. Edit nile/data.py by filling-in strings for COPYRIGHT, TITLE, SOURCE, DESCRSHORT, DESCLONG, and NOTE.

```python
COPYRIGHT = """"This is public domain."""
TITLE = """"Nile River Data"""
SOURCE = """
Cobb, G.W. 1978. The Problem of the Nile: Conditional Solution to a Changepoint Problem. Biometrika. 65.2, 243-251,
"""
DESCRSHORT = """"Annual Nile River Volume at Aswan, 1871-1970"
DESCRLONG = """"Annual Nile River Volume at Aswan, 1871-1970. The units of measurement are 1e9 m^3, and there is an apparent changepoint near 1898."""
NOTE = """
Number of observations: 100
Number of variables: 2
Variable name definitions:
  year - Year of observation
  volume - Nile River volume at Aswan
```

The data were originally used in Cobb (1987, See SOURCE). The author acknowledges that the data were originally compiled from various sources by Dr. Barbara Bell, Center for Astrophysics, Cambridge, Massachusetts. The data set is also used as an example in many textbooks and software packages.

```
Step 5: Edit the docstring of the load function in data.py to specify which dataset will be loaded. Also edit the path and the indices for the endog and exog attributes. In the nile case, there is no exog, so everything referencing exog is not used. The year variable is also not used.

Step 6: Edit the datasets/__init__.py to import the directory.

That's it! The result can be found here for reference.
Examples

Examples are invaluable for new users who hope to get up and running quickly with statsmodels, and they are extremely useful to those who wish to explore new features of statsmodels. We hope to provide documentation and tutorials for as many models and use-cases as possible! Please consider submitting an example with any PR that introduces new functionality.

User-contributed examples/tutorials/recipes can be placed on the statsmodels examples wiki page That wiki page is freely editable. Please post your cool tricks, examples, and recipes on there!

If you would rather have your example file officially accepted to the statsmodels distribution and posted on this website, you will need to go through the normal patch submission process and follow the instructions that follow.

File Format

Examples are best contributed as IPython notebooks. Save your notebook with all output cells cleared in examples/notebooks. From the notebook save the pure Python output to examples/python. The first line of the Notebook must be a header cell that contains a title for the notebook, if you want the notebook to be included in the documentation.

The Example Gallery

We have a gallery of example notebooks available here. If you would like your example to show up in this gallery, add a link to the notebook in docs/source/examples/landing.json. For the thumbnail, take a screenshot of what you think is the best “hook” for the notebook. The image will be displayed at 360 x 225 (W x H). It’s best to save the image as a PNG with a resolution that is some multiple of 360 x 225 (720 x 450 is preferred).

Before submitting a PR

To save you some time and to make the new examples nicely fit into the existing ones consider the following points.

Look at examples source code to get a feel for how statsmodels examples should look like.

Build the docs by running make html from the docs directory to see how your example looks in the fully rendered html pages.

Get Involved

Where to Start?

Use grep or download a tool like grin to search the code for TODO notes:

grin -i -I "*.py" todo

This shows almost 700 TODOs in the code base right now. Feel free to inquire on the mailing list about any of these.

Sandbox

We currently have a large amount code in the Sandbox. The medium term goal is to move much of this to feature branches as it gets worked on and remove the sandbox folder. Many of these models and functions are close to done, however, and we welcome any and all contributions to complete them, including refactoring, documentation, and tests. These models include generalized additive models (GAM), information theoretic models such as maximum entropy,
survival models, systems of equation models, restricted least squares, panel data models, and time series models such as (G)ARCH.

Contribute an Example

Contribute an example, add some technical documentation, or contribute a statistics tutorial.

Roadmap to 0.6

Work on any of the big picture ideas is very welcome. Implementing these ideas requires some thought and changes will likely affect all the codebase.

Core Development

- Refactoring models structure to have consistent variable naming, methods, and signatures. Make sure DRY is respected.

Statistics

- Bootstrapping, jackknifing, or re-sampling framework.

Internal Classes

The following summarizes classes and functions that are not intended to be directly used, but of interest only for internal use or for a developer who wants to extend on existing model classes.

Module Reference

Model and Results Classes  These are the base classes for both the estimation models and the results. They are not directly useful, but layout the structure of the subclasses and define some common methods.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>A (predictive) statistical model. Intended to be subclassed not used.</td>
</tr>
<tr>
<td>LikelihoodModel</td>
<td>Likelihood model is a subclass of Model.</td>
</tr>
<tr>
<td>GenericLikelihoodModel</td>
<td>Allows the fitting of any likelihood function via maximum likelihood.</td>
</tr>
<tr>
<td>Results</td>
<td>Class to contain model results</td>
</tr>
<tr>
<td>LikelihoodModelResults</td>
<td>Class to contain results from likelihood models</td>
</tr>
<tr>
<td>ResultMixin</td>
<td></td>
</tr>
<tr>
<td>GenericLikelihoodModelResults</td>
<td>A results class for the discrete dependent variable models.</td>
</tr>
</tbody>
</table>

```python
statsmodels.base.model.Model
class statsmodels.base.model.Model (endog, exog=None, **kwargs)
A (predictive) statistical model. Intended to be subclassed not used.

Parameters
endog : array-like
    1-d endogenous response variable. The dependent variable.

exog : array-like
    A nobs x k array where nobs is the number of observations and k is the number of
```
regressors. An intercept is not included by default and should be added by the user. See statsmodels.tools.add_constant.

missing : str
Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’ hasconst : None or bool Indicates whether the RHS includes a user-supplied constant. If True, a constant is not checked for and k constant is set to 1 and all result statistics are calculated as if a constant is present. If False, a constant is not checked for and k constant is set to 0.

Notes

endog and exog are references to any data provided. So if the data is already stored in numpy arrays and it is changed then endog and exog will change as well.

Methods

statsmodels.base.model.Model.fit
Model.\texttt{fit} ()
Fit a model to data.

statsmodels.base.model.Model.from_formula
classmethod Model.\texttt{from_formula} (formula, data, subset=None, *args, **kwargs)
Create a Model from a formula and dataframe.

Parameters formula : str or generic Formula object
The formula specifying the model
data : array-like
The data for the model. See Notes.
subset : array-like
An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
args : extra arguments
These are passed to the model
kwargs : extra keyword arguments
These are passed to the model.

Returns model : Model instance
Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

```python
statsmodels.base.model.Model.predict
Model.predict (params, exog=None, *args, **kwags)
```

After a model has been fit predict returns the fitted values.

This is a placeholder intended to be overwritten by individual models.

Attributes

```python
endog_names
exog_names
```

```python
statsmodels.base.model.LikelihoodModel
class statsmodels.base.model.LikelihoodModel (endog, exog=None, **kwags)
```

Likelihood model is a subclass of Model.

Methods

```python
fit([start_params, method, maxiter, ...]) Fit method for likelihood based models
from_formula(formula, data[, subset]) Create a Model from a formula and dataframe.
hessian(params) The Hessian matrix of the model
information(params) Fisher information matrix of model
initialize() Initialize (possibly re-initialize) a Model instance. For
loglike(params) Log-likelihood of model.
predict(params[, exog]) After a model has been fit predict returns the fitted values.
score(params) Score vector of model.
```

```python
statsmodels.base.model.LikelihoodModel.fit
LikelihoodModel.fit (start_params=None, method='newton', maxiter=100, full_output=True, disp=True, fargs=(), callback=None, retall=False, **kwags)
```

Fit method for likelihood based models

**Parameters**

**start_params** : array-like, optional

Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

**method** : str, optional

The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:

- ‘newton’ for Newton-Raphson, ‘nm’ for Nelder-Mead
- ‘bfgs’ for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
- ‘lbfgs’ for limited-memory BFGS with optional box constraints
- ‘powell’ for modified Powell’s method
- ‘cg’ for conjugate gradient
- ‘ncg’ for Newton conjugate gradient
- ‘ anneal’ for simulated annealing
- ‘subplex’ for the Nelder-Mead simplex
- ‘control’ for the control method
- ‘basinhopping’ for basin-hopping
- ‘minimize’ for a generic scipy minimize (uses different algorithms)
- ‘SLSQP’ for Sequential Least SQuares Programming
- ‘dogleg’ for interior DogLeg
- ‘trust-ncg’ for truncated Newton-CG
- ‘trust-exact’ for truncated Newton-Type dogleg
- ‘trust-kraus’ for truncated Newton-Type dogleg
- ‘trust-constr’ for truncated Newton-Type dogleg
• ‘cg’ for conjugate gradient
• ‘ncg’ for Newton-conjugate gradient
• ‘basinhopping’ for global basin-hopping solver

The explicit arguments in fit are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

**maxiter** : int, optional
The maximum number of iterations to perform.

**full_output** : bool, optional
Set to True to have all available output in the Results object’s mle_revals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

**disp** : bool, optional
Set to True to print convergence messages.

**fargs** : tuple, optional
Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

**callback** : callable callback(xk), optional
Called after each iteration, as callback(xk), where xk is the current parameter vector.

**retall** : bool, optional
Set to True to return list of solutions at each iteration. Available in Results object’s mle_revals attribute.

**Notes**

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

‘newton’
  tol : float
  Relative error in params acceptable for convergence.
‘nm’ -- Nelder Mead
  xtol : float
  Relative error in params acceptable for convergence
  ftol : float
  Relative error in loglike(params) acceptable for convergence
  maxfun : int
  Maximum number of function evaluations to make.
‘bfgs’
  gtol : float
  Stop when norm of gradient is less than gtol.
  norm : float
  Order of norm (np.Inf is max, -np.Inf is min)
  epsilon
  If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
'lbfgs'
m : int
   This many terms are used for the Hessian approximation.
factr : float
   A stop condition that is a variant of relative error.
pgtol : float
   A stop condition that uses the projected gradient.
epsilon
   If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
maxfun : int
   Maximum number of function evaluations to make.
bounds : sequence
      (min, max) pairs for each element in x,
      defining the bounds on that parameter.
      Use None for one of min or max when there is no bound in that direction.
'cg'
gtol : float
   Stop when norm of gradient is less than gtol.
norm : float
   Order of norm (np.Inf is max, -np.Inf is min)
epsilon
   If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if Likelihoodmodel.score is None.
'ncg'
fhess_p : callable f'(x,*args)
   Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if LikelihoodModel.hessian is None.
avextol : float
   Stop when the average relative error in the minimizer falls below this amount.
epsilon : float or ndarray
   If fhess is approximated, use this value for the step size. Only relevant if Likelihoodmodel.hessian is None.
'powell'
xtol : float
   Line-search error tolerance
ftol : float
   Relative error in loglike(params) for acceptable for convergence.
maxfun : int
   Maximum number of function evaluations to make.
start_direc : ndarray
   Initial direction set.
'basinhopping'
niter : integer
   The number of basin hopping iterations.
niter_success : integer
   Stop the run if the global minimum candidate remains the same for this number of iterations.
T : float
   The "temperature" parameter for the accept or reject criterion. Higher "temperatures" mean that larger jumps in function value will be accepted. For best results 'T' should be comparable to the separation (in function
value) between local minima.

stepsize : float
   Initial step size for use in the random displacement.
interval : integer
   The interval for how often to update the 'stepsize'.
minimizer : dict
   Extra keyword arguments to be passed to the minimizer
   'scipy.optimize.minimize()', for example 'method' - the
   minimization method (e.g. 'L-BFGS-B'), or 'tol' - the
   tolerance for termination. Other arguments are mapped from
   explicit argument of 'fit':
   - 'args' <- 'fargs'
   - 'jac' <- 'score'
   - 'hess' <- 'hess'

statsmodels.base.model.LikelihoodModel.from_formula

classmethod LikelihoodModel.from_formula(formula, data, subset=None, *args, **kwargs)
   Create a Model from a formula and dataframe.

   Parameters
   formula : str or generic Formula object
      The formula specifying the model
   data : array-like
      The data for the model. See Notes.
   subset : array-like
      An array-like object of booleans, integers, or index values that indicate the subset of df
to use in the model. Assumes df is a pandas.DataFrame
   args : extra arguments
      These are passed to the model
   kwargs : extra keyword arguments
      These are passed to the model.

   Returns
   model : Model instance

   Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the
model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

statsmodels.base.model.LikelihoodModel.hessian

LikelihoodModel.hessian(params)
   The Hessian matrix of the model

statsmodels.base.model.LikelihoodModel.information

LikelihoodModel.information(params)
   Fisher information matrix of model

   Returns -Hessian of loglike evaluated at params.
statsmodels.base.model.LikelihoodModel.initialize
LikelihoodModel.initialize()
Initialize (possibly re-initialize) a Model instance. For instance, the design matrix of a linear model may change and some things must be recomputed.

statsmodels.base.model.LikelihoodModel.loglike
LikelihoodModel.loglike(params)
Log-likelihood of model.

statsmodels.base.model.LikelihoodModel.predict
LikelihoodModel.predict(params, exog=None, *args, **kwargs)
After a model has been fit predict returns the fitted values.
This is a placeholder intended to be overwritten by individual models.

statsmodels.base.model.LikelihoodModel.score
LikelihoodModel.score(params)
Score vector of model.
The gradient of logL with respect to each parameter.

Attributes

    endog_names
    exog_names

statsmodels.base.model.GenericLikelihoodModel
class statsmodels.base.model.GenericLikelihoodModel(endog, exog=None, loglike=None, score=None, hessian=None, missing='none', extra_params_names=None, **kwds)
Allows the fitting of any likelihood function via maximum likelihood.
A subclass needs to specify at least the log-likelihood If the log-likelihood is specified for each observation, then results that require the Jacobian will be available. (The other case is not tested yet.)

Notes
Optimization methods that require only a likelihood function are ‘nm’ and ‘powell’
Optimization methods that require a likelihood function and a score/gradient are ‘bfgs’, ‘cg’, and ‘ncg’. A function to compute the Hessian is optional for ‘ncg’.
Optimization method that require a likelihood function, a score/gradient, and a Hessian is ‘newton’
If they are not overwritten by a subclass, then numerical gradient, Jacobian and Hessian of the log-likelihood are calculated by numerical forward differentiation. This might results in some cases in precision problems, and the Hessian might not be positive definite. Even if the Hessian is not positive definite the covariance matrix of the parameter estimates based on the outer product of the Jacobian might still be valid.
Examples

see also subclasses in directory miscmodels

import statsmodels.api as sm
data = sm.datasets.spector.load()  # in this dir from model import GenericLikelihoodModel
probit_mod = sm.Probit(data.endog, data.exog)
probit_res = probit_mod.fit()
loglike = probit_mod.loglike
score = probit_mod.score
mod = GenericLikelihoodModel(data.endog, data.exog, loglike, score)
res = mod.fit(method="nm", maxiter = 500)
import numpy as np
np.allclose(res.params, probit_res.params)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expandparams(params)</td>
<td>expand to full parameter array when some parameters are fixed</td>
</tr>
<tr>
<td>fit([start_params, method, maxiter, ...])</td>
<td>Fit the model using maximum likelihood.</td>
</tr>
<tr>
<td>from_formula(formula, data[, subset])</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>hessian(params)</td>
<td>Hessian of log-likelihood evaluated at params</td>
</tr>
<tr>
<td>information(params)</td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td>initialize()</td>
<td></td>
</tr>
<tr>
<td>jac(params, **kwds)</td>
<td>Jacobian/Gradient of log-likelihood evaluated at params for each</td>
</tr>
<tr>
<td>loglike(params)</td>
<td></td>
</tr>
<tr>
<td>loglikeobs(params)</td>
<td></td>
</tr>
<tr>
<td>nloglike(params)</td>
<td></td>
</tr>
<tr>
<td>predict(params[, exog])</td>
<td>After a model has been fit predict returns the fitted values.</td>
</tr>
<tr>
<td>reduceparams(params)</td>
<td></td>
</tr>
<tr>
<td>score(params)</td>
<td>Gradient of log-likelihood evaluated at params</td>
</tr>
</tbody>
</table>

**statsmodels.base.model.GenericLikelihoodModel.expandparams**

GenericLikelihoodModel.expandparams(params)

expand to full parameter array when some parameters are fixed

**Parameters**

- params : array
  reduced parameter array

**Returns**

- paramsfull : array
  expanded parameter array where fixed parameters are included

**Notes**

Calling this requires that self.fixed_params and self.fixed_paramsmask are defined.

**developer notes:**

This can be used in the log-likelihood to ...

this could also be replaced by a more general parameter transformation.

**statsmodels.base.model.GenericLikelihoodModel.fit**

GenericLikelihoodModel.fit(start_params=None, method='nm', maxiter=500, full_output=1, disp=1, callback=None, retall=0, **kwars)

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit
Create a Model from a formula and dataframe.

**Parameters**

- **formula**: str or generic Formula object
  - The formula specifying the model
- **data**: array-like
  - The data for the model. See Notes.
- **subset**: array-like
  - An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a *pandas.DataFrame*
- **args**: extra arguments
  - These are passed to the model
- **kwargs**: extra keyword arguments
  - These are passed to the model.

**Returns**

- **model**: Model instance

**Notes**

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.
statsmodels.base.model.GenericLikelihoodModel.nloglike

GenericLikelihoodModel.nloglike(params)

statsmodels.base.model.GenericLikelihoodModel.predict

GenericLikelihoodModel.predict(params, exog=None, *args, **kwargs)

After a model has been fit predict returns the fitted values.

This is a placeholder intended to be overwritten by individual models.

statsmodels.base.model.GenericLikelihoodModel.reduceparams

GenericLikelihoodModel.reduceparams(params)

statsmodels.base.model.GenericLikelihoodModel.score

GenericLikelihoodModel.score(params)

Gradient of log-likelihood evaluated at params

Attributes

- endog_names
- exog_names

statsmodels.base.model.Results

class statsmodels.base.model.Results(model, params, **kwd)

Class to contain model results

Parameters

- model : class instance
  - the previously specified model instance
- params : array
  - parameter estimates from the fit model

Methods

- initialize(model, params, **kwd)
- predict([exog, transform])
  - Call self.model.predict with self.params as the first argument.

statsmodels.base.model.Results.initialize

Results.initialize(model, params, **kwd)

statsmodels.base.model.Results.predict

Results.predict(exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

Parameters

- exog : array-like, optional
  - The values for which you want to predict.
- transform : bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model \( y \sim \log(x1) + \log(x2) \), and transform is True, then you can pass a data structure that contains \( x1 \) and \( x2 \) in their original form. Otherwise, you'd need to log the data first.

**Returns** See `self.model.predict`:

```python
statsmodels.base.model.LikelihoodModelResults
class statsmodels.base.model.LikelihoodModelResults(model, params, normalized_cov_params=None, scale=1.0)

Class to contain results from likelihood models

**Parameters**

- **model**: LikelihoodModel instance or subclass instance
  
  LikelihoodModelResults holds a reference to the model that is fit.

- **params**: 1d array_like
  
  Parameter estimates from estimated model

- **normalized_cov_params**: 2d array
  
  Normalized (before scaling) covariance of params. \((\text{dot}(X.T,X))^{-1}\)

- **scale**: float
  
  For (some subset of models) scale will typically be the mean square error from the estimated model (\(\sigma^2\))

**Returns**

**Attributes**:

- **mle_retvals**: dict
  
  Contains the values returned from the chosen optimization method if full_output is True during the fit. Available only if the model is fit by maximum likelihood. See notes below for the output from the different methods.

- **mle_settings**: dict
  
  Contains the arguments passed to the chosen optimization method. Available if the model is fit by maximum likelihood. See LikelihoodModel.fit for more information.

- **model**: model instance
  
  LikelihoodResults contains a reference to the model that is fit.

- **params**: ndarray
  
  The parameters estimated for the model.

- **scale**: float
  
  The scaling factor of the model given during instantiation.

- **tvalues**: array
  
  The t-values of the standard errors.

**Notes**

The covariance of params is given by scale times `normalized_cov_params`.

Return values by solver if full_output is True during fit:
'newton'
  
  **fopt** [float] The value of the (negative) loglikelihood at its minimum.
  
  **iterations** [int] Number of iterations performed.
  
  **score** [ndarray] The score vector at the optimum.
  
  **Hessian** [ndarray] The Hessian at the optimum.
  
  **warnflag** [int] 1 if maxiter is exceeded. 0 if successful convergence.
  
  
  **allvecs** [list] List of solutions at each iteration.

'nm'
  
  **fopt** [float] The value of the (negative) loglikelihood at its minimum.
  
  **iterations** [int] Number of iterations performed.
  
  **warnflag** [int] 1: Maximum number of function evaluations made. 2: Maximum number of iterations reached.
  
  
  **allvecs** [list] List of solutions at each iteration.

'bfgs'
  
  **fopt** [float] Value of the (negative) loglikelihood at its minimum.
  
  **gopt** [float] Value of gradient at minimum, which should be near 0.
  
  **Hinv** [ndarray] value of the inverse Hessian matrix at minimum. Note that this is just an approximation and will often be different from the value of the analytic Hessian.
  
  **fcalls** [int] Number of calls to loglike.
  
  **gcalls** [int] Number of calls to gradient/score.
  
  **warnflag** [int] Warning flag:
  
  - 0 if converged
  - 1 if too many function evaluations or too many iterations
  - 2 if stopped for another reason
  
  
  **allvecs** [list] Results at each iteration.

'lbfgs'
  
  **fopt** [float] Value of the (negative) loglikelihood at its minimum.
  
  **gopt** [float] Value of gradient at minimum, which should be near 0.
  
  **fcalls** [int] Number of calls to loglike.
  
  **warnflag** [int] Warning flag:
  
  - 0 if converged
  - 1 if too many function evaluations or too many iterations
  - 2 if stopped for another reason
  

'powell'
  
  **fopt** [float] Value of the (negative) loglikelihood at its minimum.
iterations  [int] Number of iterations performed.
fcalls  [int] Number of calls to loglike.
warnflag  [int] 1: Maximum number of function evaluations. 2: Maximum number of iterations.
allvecs  [list] Results at each iteration.

'cg'
fopt  [float] Value of the (negative) loglikelihood at its minimum.
fcalls  [int] Number of calls to loglike.
gcalls  [int] Number of calls to gradient/score.
warnflag  [int] 1: Maximum number of iterations exceeded. 2: Gradient and/or function calls not changing.
allvecs  [list] Results at each iteration.

'ncg'
fopt  [float] Value of the (negative) loglikelihood at its minimum.
fcalls  [int] Number of calls to loglike.
gcalls  [int] Number of calls to gradient/score.
hcalls  [int] Number of calls to hessian.
warnflag  [int] 1: Maximum number of iterations exceeded.
allvecs  [list] Results at each iteration.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bse()</td>
<td>Returns the confidence interval of the fitted parameters.</td>
</tr>
<tr>
<td>conf_int([alpha, cols, method])</td>
<td>Returns the variance/covariance matrix.</td>
</tr>
<tr>
<td>cov_params([r_matrix, column, scale, cov_p, ...])</td>
<td>Compute the F-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td>f_test(r_matrix[, q_matrix, cov_p, scale, ...])</td>
<td>Call self.model.predict with self.params as the first argument.</td>
</tr>
<tr>
<td>initialize(model, params, **kwd)</td>
<td>remove data arrays, all nobs arrays from result and model</td>
</tr>
<tr>
<td>llf()</td>
<td>load a pickle, (class method)</td>
</tr>
<tr>
<td>load(fname)</td>
<td>save a pickle of this instance</td>
</tr>
<tr>
<td>normalized_cov_params()</td>
<td>Compute a t-test for a joint linear hypothesis of the form Rb = q</td>
</tr>
<tr>
<td>predict([exog, transform])</td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td>pvalues()</td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td>remove_data()</td>
<td></td>
</tr>
<tr>
<td>save(fname[, remove_data])</td>
<td></td>
</tr>
<tr>
<td>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</td>
<td></td>
</tr>
<tr>
<td>tvalues()</td>
<td></td>
</tr>
<tr>
<td>wald_test(r_matrix[, q_matrix, cov_p, ...])</td>
<td></td>
</tr>
</tbody>
</table>

statsmodels.base.model.LikelihoodModelResults.bse
static LikelihoodModelResults.bse()

statsmodels.base.model.LikelihoodModelResults.conf_int
LikelihoodModelResults.conf_int(alpha=0.05, cols=None, method='default')
Returns the confidence interval of the fitted parameters.

Parameters

alpha : float, optional
The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

cols : array-like, optional
cols specifies which confidence intervals to return

method : string
Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jij” : “jac” : “boot-bse” “boot_quant” “profile”

Returns

conf_int : array
Each row contains [lower, upper] confidence interval

Notes
The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[ -5496529.48322745,  -1467987.78596704],
[    -177.02903529,     207.15277984],
[    -3.12506664,     -0.91539297],
[    -1.51794870,     -0.54850503],
[    -0.56251721,      0.46030900],
[     798.78751530,    2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[ -0.11158110,  0.03994274],
[ -3.12506664, -0.91539297]])

statsmodels.base.model.LikelihoodModelResults.cov_params
LikelihoodModelResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)
Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.
Parameters  

**r_matrix** : array-like

Can be 1d, or 2d. Can be used alone or with other.

**column** : array-like, optional

Must be used on its own. Can be 0d or 1d see below.

**scale** : float, optional

Can be specified or not. Default is None, which means that the scale argument is taken from the model.

**other** : array-like, optional

Can be used when r_matrix is specified.

Returns (The below are assumed to be in matrix notation.) :

**cov** : ndarray

If no argument is specified returns the covariance matrix of a model :

$(\text{scale}) \cdot (X^T X)^{-1}$

If contrast is specified it pre and post-multiplies as follows :

$(\text{scale}) \cdot r_{\text{matrix}} \cdot (X^T X)^{-1} \cdot r_{\text{matrix}}^T$

If contrast and other are specified returns :

$(\text{scale}) \cdot r_{\text{matrix}} \cdot (X^T X)^{-1} \cdot \text{other}^T$

If column is specified returns :

$(\text{scale}) \cdot (X^T X)^{-1}[\text{column}, \text{column}]$ if column is 0d :

OR :

$(\text{scale}) \cdot (X^T X)^{-1}[\text{column}][:, \text{column}]$ if column is 1d :

```
statsmodels.base.model.LikelihoodModelResults.f_test
LikelihoodModelResults.f_test(r_matrix=None, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
```

Compute the F-test for a joint linear hypothesis.

Parameters  

**r_matrix** : array-like, str, or tuple

- **array** : An r x k array where r is the number of restrictions to test and k is the number of regressors.
- **str** : The full hypotheses to test can be given as a string. See the examples.
- **tuple** : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

**q_matrix** : array-like

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

**cov_p** : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

**scale** : float, optional
Default is 1.0 for no scaling.

\texttt{invcov} : array-like, optional

A \( q \times q \) array to specify an inverse covariance matrix based on a restrictions matrix.

\textbf{See also:}

\texttt{statsmodels.contrasts.statsmodels.model.LikelihoodModelResults.wald_test},
\texttt{statsmodels.model.LikelihoodModelResults.t_test}, \texttt{patsy.DesignInfo.linear_constraint}

\textbf{Notes}

The matrix \( r_{matrix} \) is assumed to be non-singular. More precisely,

\[ r_{matrix} (pX pX^T) r_{matrix}^T \]

is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

\textbf{Examples}

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:,:]
This tests that each coefficient is jointly statistically significantly different from zero.

>>> print(results.f_test(A))
<\texttt{F contrast: } F=330.28533923463488, p=4.98403052872e-10, 
df\_denom=9, df\_num=6>

Compare this to

>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10

>>> B = np.array([[0,0,1,-1,0,0,0], [0,0,0,0,0,1,-1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

>>> print(results.f_test(B))
<\texttt{F contrast: } F=9.740461873303655, p=0.00560528853174, df\_denom=9, 
df\_num=2>

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
```
```python
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```
Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

**statsmodels.base.model.LikelihoodModelResults.save**

```py
LikelihoodModelResults.save(fname, remove_data=False)
```

save a pickle of this instance

**Parameters**

- **fname**: string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- **remove_data**: bool
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

**statsmodels.base.model.LikelihoodModelResults.t_test**

```py
LikelihoodModelResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)
```

Compute a t-test for a joint linear hypothesis of the form \( Rb = q \)

**Parameters**

- **r_matrix**: array-like, str, tuple
  
  - array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  
  - str : The full hypotheses to test can be given as a string. See the examples.
  
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like or scalar, optional
  
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p**: array-like, optional
  
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale**: float, optional
  
  An optional scale to use. Default is the scale specified by the model fit.

- **use_t**: bool, optional
  
  If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

**See also:**
**tvalues**  individual t statistics

**f_test**  for F tests

`patsy.DesignInfo.linear_constraint`

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T Test = results.t_test(r) >>>print(T_test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> data = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, data).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

statsmodels.base.model.LikelihoodModelResults.tvalues

`statsmodels.base.model.LikelihoodModelResults.tvalues()`  
Return the t-statistic for a given parameter estimate.

statsmodels.base.model.LikelihoodModelResults.wald_test

`LikelihoodModelResults.wald_test (r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)`  
Compute a Wald-test for a joint linear hypothesis.

**Parameters**  
`r_matrix` : array-like, str, or tuple

- array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

`q_matrix` : array-like

This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.
cov_p : array-like, optional
An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
Default is 1.0 for no scaling.

invcov : array-like, optional
A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool
If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes
The matrix $r_matrix$ is assumed to be non-singular. More precisely,

$r_matrix (pX pX.T) r_matrix.T$

is assumed invertible. Here, $pX$ is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
statsmodels.base.model.ResultMixin.bootstrap

ResultMixin.bootstrap(nrep=100, method='nm', disp=0, store=1)

simple bootstrap to get mean and variance of estimator

see notes

Parameters

nrep : int
    number of bootstrap replications

method : str
    optimization method to use

disp : bool
    If true, then optimization prints results

store : bool
    If true, then parameter estimates for all bootstrap iterations are attached in self.bootstrap_results

Returns

mean : array
    mean of parameter estimates over bootstrap replications

std : array
    standard deviation of parameter estimates over bootstrap replications

Notes

This was mainly written to compare estimators of the standard errors of the parameter estimates. It uses independent random sampling from the original endog and exog, and therefore is only correct if observations are independently distributed.

This will be moved to apply only to models with independently distributed observations.

statsmodels.base.model.ResultMixin.bsejac

static ResultMixin.bsejac()

standard deviation of parameter estimates based on covjac

statsmodels.base.model.ResultMixin.bsejhj

static ResultMixin.bsejhj()

standard deviation of parameter estimates based on covHJH

statsmodels.base.model.ResultMixin.covjac

static ResultMixin.covjac()

covariance of parameters based on outer product of jacobian of log-likelihood

statsmodels.base.model.ResultMixin.covjhj

static ResultMixin.covjhj()

covariance of parameters based on HJH
    dot product of Hessian, Jacobian, Jacobian, Hessian of likelihood

name should be covhjh
statsmodels.base.model.ResultMixin.df_modelwc
static ResultMixin.df_modelwc()

statsmodels.base.model.ResultMixin.get_nlfun
ResultMixin.get_nlfun(fun)

statsmodels.base.model.ResultMixin.hessv
static ResultMixin.hessv()
cached Hessian of log-likelihood

statsmodels.base.model.ResultMixin.jacv
static ResultMixin.jacv()
cached Jacobian of log-likelihood

statsmodels.base.model.GenericLikelihoodModelResults
class statsmodels.base.model.GenericLikelihoodModelResults(model, mlefit)
A results class for the discrete dependent variable models.
..Warning :
The following description has not been updated to this version/class. Where are AIC, BIC, ....? docstring looks like copy from discretemod

Parameters model : A DiscreteModel instance
mlefit : instance of LikelihoodResults
    This contains the numerical optimization results as returned by LikelihoodModel.fit(),
    in a superclass of GnericLikelihoodModels

Returns *Attributes* :

Warning most of these are not available yet :

aic : float
    Akaike information criterion. -2*(llf - p) where p is the number of regressors including
    the intercept.

bic : float
    Bayesian information criterion. -2*llf + ln(nobs)*p where p is the number of regres-
    sors including the intercept.

bse : array
    The standard errors of the coefficients.

df_resid : float
    See model definition.

df_model : float
    See model definition.

fitted_values : array
    Linear predictor XB.

llf : float
Value of the loglikelihood

**llnull**: float

Value of the constant-only loglikelihood

**llr**: float

Likelihood ratio chi-squared statistic; \(-2*(llnull - llf)\)

**llr_pvalue**: float

The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr has a chi-squared distribution with degrees of freedom *df_model*.

**prssquared**: float

McFadden’s pseudo-R-squared. \(1 - (llf/llnull)\)

### Methods

- **aic()**
- **bic()**
- **bootstrap**([Inrep, method, disp, store])
- **bse()**
- **bsejac()**
- **bsejhj()**
- **conf_int([alpha, cols, method])**
- **cov_params([r_matrix, column, scale, cov_p, ...])**
- **covjac()**
- **covjhj()**
- **df_modelwc()**
- **f_test(r_matrix[, q_matrix, cov_p, scale, ...])**
- **get_nlfun(fun)**
- **hessv()**
- **initialize(model, params, **kwds)**
- **jacv()**
- **llf()**
- **load(fname)**
- **normalized_cov_params()**
- **predict([exog, transform])**
- **pvalues()**
- **remove_data()**
- **save(fname[, remove_data])**
- **summary([yname, xname, title, alpha])**
- **t_test(r_matrix[, q_matrix, cov_p, scale, use_t])**
- **tvalues()**
- **wald_test(r_matrix[, q_matrix, cov_p, ...])**

---

statsmodels.base.model.GenericLikelihoodModelResults.aic

**static** GenericLikelihoodModelResults.aic()

statsmodels.base.model.GenericLikelihoodModelResults.bic

**static** GenericLikelihoodModelResults.bic()
GenericLikelihoodModelResults.bootstrap

**Parameters**

- `nrep`: int
  - number of bootstrap replications
- `method`: str
  - optimization method to use
- `disp`: bool
  - If true, then optimization prints results
- `store`: bool
  - If true, then parameter estimates for all bootstrap iterations are attached in self.bootstrap_results

**Returns**

- `mean`: array
  - mean of parameter estimates over bootstrap replications
- `std`: array
  - standard deviation of parameter estimates over bootstrap replications

**Notes**

This was mainly written to compare estimators of the standard errors of the parameter estimates. It uses independent random sampling from the original endog and exog, and therefore is only correct if observations are independently distributed.

This will be moved to apply only to models with independently distributed observations.

**static**

- GenericLikelihoodModelResults.bse()
- GenericLikelihoodModelResults.bsejac()
- GenericLikelihoodModelResults.bsejhj()

**static**

- GenericLikelihoodModelResults.conf_int(alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

**Parameters**

- `alpha`: float, optional
  - The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.
cols : array-like, optional

*cols* specifies which confidence intervals to return

method : string

Not Implemented Yet Method to estimate the confidence_interval. “Default”: uses self.bse which is based on inverse Hessian for MLE “jhj”: “jac”: “boot-bse” “boot_quant” “profile"

Returns conf_int : array

Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[-5496529.48322745, -1467987.78596704],
       [ -177.02903529, 207.15277984],
       [ -0.1115811 ,  0.03994274],
       [-3.12506664, -0.91539297],
       [-1.5179487 , -0.54850503],
       [-0.56251721,  0.460309  ],
       [ 798.7875153 , 2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[-0.1115811 ,  0.03994274],
       [-3.12506664, -0.91539297]])
```

statsmodels.base.model.GenericLikelihoodModelResults.cov_params

GenericLikelihoodModelResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters r_matrix : array-like

Can be 1d, or 2d. Can be used alone or with other.

column : array-like, optional

Must be used on its own. Can be 0d or 1d see below.

scale : float, optional

Can be specified or not. Default is None, which means that the scale argument is taken from the model.
other : array-like, optional

Can be used when r_matrix is specified.

Returns (The below are assumed to be in matrix notation.) :

cov : ndarray

If no argument is specified returns the covariance matrix of a model :

(scale)*((X.T * X)^(-1)) :

If contrast is specified it pre and post-multiplies as follows :

(scale) * r_matrix (X.T * X)^(-1) r_matrix.T :

If contrast and other are specified returns :

(scale) * r_matrix (X.T * X)^(-1) other.T :

If column is specified returns :

(scale) * (X.T * X)^(-1)[column,column] if column is 0d :

OR :

(scales) * (X.T * X)^(-1)[column][:,column] if column is 1d :

statsmodels.base.model.GenericLikelihoodModelResults.covjac

static GenericLikelihoodModelResults.covjac()
covariance of parameters based on outer product of jacobian of log-likelihood

statsmodels.base.model.GenericLikelihoodModelResults.covjhj

static GenericLikelihoodModelResults.covjhj()
covariance of parameters based on HJJH
dot product of Hessian, Jacobian, Jacobian, Hessian of likelihood
name should be covhjh

statsmodels.base.model.GenericLikelihoodModelResults.df_modelwc

static GenericLikelihoodModelResults.df_modelwc()

statsmodels.base.model.GenericLikelihoodModelResults.f_test

GenericLikelihoodModelResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
Compute the F-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple

• array : An r x k array where r is the number of restrictions to test and k is the number of regressors.

• str : The full hypotheses to test can be given as a string. See the examples.

• tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like

This is deprecated. See r_matrix and the examples for more information on new usage.
Can be either a scalar or a length p row vector. If omitted and r_matrix is an array,
q_matrix is assumed to be a conformable array of zeros.
cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

Default is 1.0 for no scaling.

invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely, r_matrix (pX pX.T) r_matrix.T is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:,:]

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
< F contrast: F=330.28533923463488, p=4.98403052872e-10, 
  df_denom=9, df_num=6>
```

Compare this to

```python
>>> results.F
330.285392346658
>>> results.F_p
4.98403096572e-10
```

```python
>>> B = np.array(([0,0,1,-1,0,0,0], [0,0,0,0,0,1,-1]))

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
< F contrast: F=9.740461873303655, p=0.00560528853174, 
  df_denom=9, df_num=2>
```

Alternatively, you can specify the hypothesis tests using a string
```python
from statsmodels.datasets import longley
from statsmodels.formula.api import ols

dta = longley.load_pandas().data

formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
results = ols(formula, dta).fit()

hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
f_test = results.f_test(hypotheses)
print(f_test)
```

Statsmodels base model GenericLikelihoodModelResults.get_nlfun

GenericLikelihoodModelResults.get_nlfun(fun)

Statsmodels base model GenericLikelihoodModelResults.hessv

static GenericLikelihoodModelResults.hessv()

cached Hessian of log-likelihood

Statsmodels base model GenericLikelihoodModelResults.initialize

GenericLikelihoodModelResults.initialize(model, params, **kwds)

Statsmodels base model GenericLikelihoodModelResults.jacv

static GenericLikelihoodModelResults.jacv()

cached Jacobian of log-likelihood

Statsmodels base model GenericLikelihoodModelResults.llf

static GenericLikelihoodModelResults.llf()

Statsmodels base model GenericLikelihoodModelResults.load

classmethod GenericLikelihoodModelResults.load(fname)

load a pickle, (class method)

Parameters fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance :

Statsmodels base model GenericLikelihoodModelResults.normalized_cov_params

GenericLikelihoodModelResults.normalized_cov_params()

Statsmodels base model GenericLikelihoodModelResults.predict

GenericLikelihoodModelResults.predict(exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

Parameters exog : array-like, optional

The values for which you want to predict.

transform : bool, optional

2.10. Developer Page 81
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model \( y \sim \log(x1) + \log(x2) \), and transform is True, then you can pass a data structure that contains \( x1 \) and \( x2 \) in their original form. Otherwise, you'd need to log the data first.

Returns
See self.model.predict:

```
statsmodels.base.model.GenericLikelihoodModelResults.pvalues
static GenericLikelihoodModelResults.pvalues()
```

```
statsmodels.base.model.GenericLikelihoodModelResults.remove_data
GenericLikelihoodModelResults.remove_data()
```

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

```
statsmodels.base.model.GenericLikelihoodModelResults.save
GenericLikelihoodModelResults.save(fname=None, remove_data=False)
```

save a pickle of this instance

**Parameters**

- `fname` : string or filehandle
  fname can be a string to a file path or filename, or a filehandle.
- `remove_data` : bool
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

```
statsmodels.base.model.GenericLikelihoodModelResults.summary
GenericLikelihoodModelResults.summary(yname=None, xname=None, title=None, alpha=0.05)
```

Summarize the Regression Results

**Parameters**

- `yname` : string, optional
  Default is \( y \)
xname : list of strings, optional
    Default is var_## for # in p the number of regressors

title : string, optional
    Title for the top table. If not None, then this replaces the default title

alpha : float
    significance level for the confidence intervals

Returns smry : Summary instance
    this holds the summary tables and text, which can be printed or converted to various
    output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.base.model.GenericLikelihoodModelResults.t_test
GenericLikelihoodModelResults.t_test(r_matrix, q_matrix=None, cov_p=None,
    scale=None, use_t=None)
Compute a t-test for a joint linear hypothesis of the form Rb = q

Parameters r_matrix : array-like, str, tuple
    • array : If an array is given, a p x k 2d array or length k 1d array specifying the linear
      restrictions.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional
    This is deprecated. See r_matrix and the examples for more information on new usage.
    Can be either a scalar or a length p row vector. If omitted and r_matrix is an array,
    q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
    An alternative estimate for the parameter covariance matrix. If None is given,
    self.normalized_cov_params is used.

scale : float, optional
    An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional
    If use_t is None, then the default of the model is used. If use_t is True, then the p-values
    are based on the t distribution. If use_t is False, then the p-values are based on the
    normal distribution.

See also:

tvalues individual t statistics

f_test for F tests

patsy.DesignInfo.linear_constraint
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1, -1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```

This tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>> print(T_Test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string.

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

The `statsmodels` library provides functions to perform hypothesis testing, including the `t_test` method for testing individual coefficients and the `wald_test` method for testing joint linear hypotheses.

```python
statsmodels.base.model.GenericLikelihoodModelResults.tvalues
static GenericLikelihoodModelResults.tvalues()
Return the t-statistic for a given parameter estimate.
```

```python
statsmodels.base.model.GenericLikelihoodModelResults.wald_test
GenericLikelihoodModelResults.wald_test(r_matrix=None, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.
```

**Parameters**

- `r_matrix` : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- `q_matrix` : array-like
  This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.

- `cov_p` : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, `self.normalized_cov_params` is used.

- `scale` : float, optional
  Default is 1.0 for no scaling.
**invcov** : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**use_f** : bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

**See also:**

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

**Notes**

The matrix $r\_matrix$ is assumed to be non-singular. More precisely, $r\_matrix (pX pX\_T) r\_matrix\_T$

is assumed invertible. Here, $pX$ is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
Linear Model
Generalized Linear Model
Discrete Model

Robust Model
Testing on Build Machines

There are currently several places that statsmodels is automatically built and tested against different dependency and Python versions and architectures. Check these logs periodically, make sure everything looks okay, and fix any failures:

- Travis CI
- Daily testing on Ubuntu via Python(x,y)
- NiPy testing on SPARC Boxes

The test coverage pages are here.

- Coveralls

2.11 Internal Classes

The following summarizes classes and functions that are not intended to be directly used, but of interest only for internal use or for a developer who wants to extend on existing model classes.

2.11.1 Module Reference

Model and Results Classes

These are the base classes for both the estimation models and the results. They are not directly useful, but layout the structure of the subclasses and define some common methods.

<table>
<thead>
<tr>
<th>Class Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model(endog[, exog])</td>
<td>A (predictive) statistical model.</td>
</tr>
<tr>
<td>LikelihoodModel(endog[, exog])</td>
<td>Likelihood model is a subclass of Model.</td>
</tr>
<tr>
<td>GenericLikelihoodModel(endog[, exog, ...])</td>
<td>Allows the fitting of any likelihood function via maximum likelihood.</td>
</tr>
<tr>
<td>Results(model, params, **kwds)</td>
<td>Class to contain model results.</td>
</tr>
<tr>
<td>LikelihoodModelResults(model, params[, ...])</td>
<td>Class to contain results from likelihood models</td>
</tr>
<tr>
<td>ResultMixin</td>
<td></td>
</tr>
<tr>
<td>GenericLikelihoodModelResults(model, mlefit)</td>
<td>A results class for the discrete dependent variable models.</td>
</tr>
</tbody>
</table>
class statsmodels.base.model.Model (endog, exog=None, **kwargs)
A (predictive) statistical model. Intended to be subclassed not used.

Parameters

endog : array-like

1-d endogenous response variable. The dependent variable.

exog : array-like

A nobs x k array where nobs is the number of observations and k is the number of
regressors. An intercept is not included by default and should be added by the user. See
statsmodels.tools.add_constant.

missing : str

Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If
‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default
is ‘none.’ hasconst : None or bool
Indicates whether the RHS includes a user-supplied constant. If True, a constant is not checked for
and k_constant is set to 1 and all result
statistics are calculated as if a constant is present. If False, a constant is not checked for
and k_constant is set to 0.

Notes

endog and exog are references to any data provided. So if the data is already stored in numpy arrays and it is
changed then endog and exog will change as well.

Methods

fit() Fit a model to data.

from_formula(formula, data[, subset]) Create a Model from a formula and dataframe.
predict(params[, exog]) After a model has been fit predict returns the fitted values.

statsmodels.base.model.Model.fit
Model.fit()
Fit a model to data.

statsmodels.base.model.Model.from_formula
classmethod Model.from_formula (formula, data, subset=None, *args, **kwargs)
Create a Model from a formula and dataframe.

Parameters

formula : str or generic Formula object

The formula specifying the model
data : array-like

The data for the model. See Notes.

subset : array-like

An array-like object of booleans, integers, or index values that indicate the subset of df
to use in the model. Assumes df is a pandas.DataFrame
statsmodels Documentation, Release 0.6.0

args : extra arguments
These are passed to the model

kwargs : extra keyword arguments
These are passed to the model.

Returns model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the
model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

statsmodels.base.model.Model.predict

Model.predict (params, exog=None, *args, **kwargs)

After a model has been fit predict returns the fitted values.

This is a placeholder intended to be overwritten by individual models.

Attributes

    endog_names
    exog_names

statsmodels.base.model.LikelihoodModel

class statsmodels.base.model.LikelihoodModel (endog, exog=None, **kwargs)

Likelihood model is a subclass of Model.

Methods

    fit([start_params, method, maxiter, ...]) Fit method for likelihood based models
    from_formula(formula, data[, subset]) Create a Model from a formula and dataframe.
    hessian(params) The Hessian matrix of the model
    information(params) Fisher information matrix of model
    initialize() Initialize (possibly re-initialize) a Model instance. For
    loglike(params) Log-likelihood of model.
    predict(params[, exog]) After a model has been fit predict returns the fitted values.
    score(params) Score vector of model.

statsmodels.base.model.LikelihoodModel.fit

LikelihoodModel.fit (start_params=None, method='newton', maxiter=100, full_output=True,
disp=True, fargs=(), callback=None, retall=False, **kwargs)

Fit method for likelihood based models

Parameters start_params : array-like, optional

    Initial guess of the solution for the loglikelihood maximization. The default is an array
    of zeros.

2.11. Internal Classes

91
**method** : str, optional

The `method` determines which solver from `scipy.optimize` is used, and it can be chosen from among the following strings:

- ‘newton’ for Newton-Raphson, ‘nm’ for Nelder-Mead
- ‘bfgs’ for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
- ‘lbfgs’ for limited-memory BFGS with optional box constraints
- ‘powell’ for modified Powell’s method
- ‘cg’ for conjugate gradient
- ‘ncg’ for Newton-conjugate gradient
- ‘basinhopping’ for global basin-hopping solver

The explicit arguments in `fit` are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

**maxiter** : int, optional

The maximum number of iterations to perform.

**full_output** : bool, optional

Set to True to have all available output in the Results object’s `mle_retvals` attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

**disp** : bool, optional

Set to True to print convergence messages.

**fargs** : tuple, optional

Extra arguments passed to the likelihood function, i.e., `loglike(x,*args)`

**callback** : callable callback(xk), optional

Called after each iteration, as `callback(xk)`, where `xk` is the current parameter vector.

**retall** : bool, optional

Set to True to return list of solutions at each iteration. Available in Results object’s `mle_retvals` attribute.

**Notes**

The ‘basinhopping’ solver ignores `maxiter`, `retall`, `full_output` explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

- **newton**
  - `tol` : float
    Relative error in params acceptable for convergence.
  - **nm** -- Nelder Mead
    - `xtol` : float
      Relative error in params acceptable for convergence
    - `ftol` : float
      Relative error in loglike(params) acceptable for
convergence
maxfun : int
   Maximum number of function evaluations to make.
'bfgs'
gtol : float
   Stop when norm of gradient is less than gtol.
norm : float
   Order of norm (np.Inf is max, -np.Inf is min)
epsilon
   If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
'lbfgs'
m : int
   This many terms are used for the Hessian approximation.
factr : float
   A stop condition that is a variant of relative error.
pgtol : float
   A stop condition that uses the projected gradient.
epsilon
   If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
maxfun : int
   Maximum number of function evaluations to make.
bounds : sequence
   (min, max) pairs for each element in x, defining the bounds on that parameter.
   Use None for one of min or max when there is no bound in that direction.
'cg'
gtol : float
   Stop when norm of gradient is less than gtol.
norm : float
   Order of norm (np.Inf is max, -np.Inf is min)
epsilon
   If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if LikelihoodModel.score is None.
'ncg'
fhess_p : callable f'(x,*args)
   Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if LikelihoodModel.hessian is None.
avextol : float
   Stop when the average relative error in the minimizer falls below this amount.
epsilon : float or ndarray
   If fhess is approximated, use this value for the step size. Only relevant if LikelihoodModel.hessian is None.
'powell'
xtol : float
   Line-search error tolerance
ftol : float
   Relative error in loglike(params) for acceptable for convergence.
maxfun : int
   Maximum number of function evaluations to make.
start_direc : ndarray
   Initial direction set.
'basinhopping'

niter : integer
The number of basin hopping iterations.
niter_success : integer
Stop the run if the global minimum candidate remains the same for this number of iterations.

T : float
The "temperature" parameter for the accept or reject criterion. Higher "temperatures" mean that larger jumps in function value will be accepted. For best results 'T' should be comparable to the separation (in function value) between local minima.

stepsize : float
Initial step size for use in the random displacement.

interval : integer
The interval for how often to update the 'stepsize'.

minimizer : dict
Extra keyword arguments to be passed to the minimizer 'scipy.optimize.minimize()', for example 'method' - the minimization method (e.g. 'L-BFGS-B'), or 'tol' - the tolerance for termination. Other arguments are mapped from explicit argument of 'fit':
- 'args' <- 'fargs'
- 'jac' <- 'score'
- 'hess' <- 'hess'

statsmodels.base.model.LikelihoodModel.from_formula

classmethod LikelihoodModel.from_formula(formula, data, subset=None, *args, **kwargs)
Create a Model from a formula and dataframe.

Parameters formula : str or generic Formula object
The formula specifying the model
data : array-like
The data for the model. See Notes.
subset : array-like
An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
args : extra arguments
These are passed to the model
kwargs : extra keyword arguments
These are passed to the model.

Returns model : Model instance

Notes
data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.
statsmodels.base.model.LikelihoodModel.hessian
LikelihoodModel.hessian(params)
The Hessian matrix of the model

statsmodels.base.model.LikelihoodModel.information
LikelihoodModel.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

statsmodels.base.model.LikelihoodModel.initialize
LikelihoodModel.initialize()
Initialize (possibly re-initialize) a Model instance. For instance, the design matrix of a linear model may change and some things must be recomputed.

statsmodels.base.model.LikelihoodModel.loglike
LikelihoodModel.loglike(params)
Log-likelihood of model.

statsmodels.base.model.LikelihoodModel.predict
LikelihoodModel.predict(params, exog=None, *args, **kwargs)
After a model has been fit predict returns the fitted values. This is a placeholder intended to be overwritten by individual models.

statsmodels.base.model.LikelihoodModel.score
LikelihoodModel.score(params)
Score vector of model.
The gradient of logL with respect to each parameter.

Attributes

```
endog_names
exog_names
```

statsmodels.base.model.GenericLikelihoodModel
class statsmodels.base.model.GenericLikelihoodModel(endog, exog=None, loglike=None, score=None, hessian=None, missing='none', extra_params_names=None, **kwds)

Allows the fitting of any likelihood function via maximum likelihood.

A subclass needs to specify at least the log-likelihood. If the log-likelihood is specified for each observation, then results that require the Jacobian will be available. (The other case is not tested yet.)

Notes

Optimization methods that require only a likelihood function are ‘nm’ and ‘powell’

2.11. Internal Classes 95
Optimization methods that require a likelihood function and a score/gradient are ‘bfgs’, ‘cg’, and ‘ncg’. A function to compute the Hessian is optional for ‘ncg’.

Optimization method that require a likelihood function, a score/gradient, and a Hessian is ‘newton’

If they are not overwritten by a subclass, then numerical gradient, Jacobian and Hessian of the log-likelihood are calculated by numerical forward differentiation. This might results in some cases in precision problems, and the Hessian might not be positive definite. Even if the Hessian is not positive definite the covariance matrix of the parameter estimates based on the outer product of the Jacobian might still be valid.

Examples

see also subclasses in directory miscmodels

import statsmodels.api as sm data = sm.datasets.spector.load() data.exog = sm.add_constant(data.exog) # in this dir from model import GenericLikelihoodModel probit_mod = sm.Probit(data.endog, data.exog) probit_res = probit_mod.fit() loglike = probit_mod.loglike score = probit_mod.score mod = GenericLikelihoodModel(data.endog, data.exog, loglike, score) res = mod.fit(method="nm", maxiter = 500) import numpy as np np.allclose(res.params, probit_res.params)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expandparams</td>
<td>expand to full parameter array when some parameters are fixed</td>
</tr>
<tr>
<td>fit</td>
<td>Fit the model using maximum likelihood.</td>
</tr>
<tr>
<td>from_formula</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>hessian</td>
<td>Hessian of log-likelihood evaluated at params</td>
</tr>
<tr>
<td>information</td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td>initialize</td>
<td></td>
</tr>
<tr>
<td>jac</td>
<td>Jacobian/Gradient of log-likelihood evaluated at params for each</td>
</tr>
<tr>
<td>loglike</td>
<td></td>
</tr>
<tr>
<td>loglikeobs</td>
<td></td>
</tr>
<tr>
<td>nloglike</td>
<td></td>
</tr>
<tr>
<td>predict</td>
<td>After a model has been fit predict returns the fitted values.</td>
</tr>
<tr>
<td>reduceparams</td>
<td></td>
</tr>
<tr>
<td>score</td>
<td>Gradient of log-likelihood evaluated at params</td>
</tr>
</tbody>
</table>

```
statsmodels.base.model.GenericLikelihoodModel.expandparams
GenericLikelihoodModel.expandparams (params)
expand to full parameter array when some parameters are fixed

Parameters params : array

Returns paramsfull : array
expanded parameter array where fixed parameters are included

Notes

Calling this requires that self.fixed_params and self.fixed_paramsmask are defined.
```
This can be used in the log-likelihood to ...
this could also be replaced by a more general parameter transformation.

```
statsmodels.base.model.GenericLikelihoodModel.fit
GenericLikelihoodModel.fit(start_params=None, method='nm', maxiter=500, full_output=1, disp=1, callback=None, retall=0, **kwargs)
```

Fit the model using maximum likelihood.
The rest of the docstring is from statsmodels.LikelihoodModel.fit

```
statsmodels.base.model.GenericLikelihoodModel.from_formula
classmethod GenericLikelihoodModel.from_formula(formula, data, subset=None, *args, **kwargs)
```

Create a Model from a formula and dataframe.

**Parameters**

- **formula**: str or generic Formula object
  - The formula specifying the model

- **data**: array-like
  - The data for the model. See Notes.

- **subset**: array-like
  - An array-like object of bools, integers, or index values that indicate the subset of df
    to use in the model. Assumes df is a pandas.DataFrame

- **args**: extra arguments
  - These are passed to the model

- **kwargs**: extra keyword arguments
  - These are passed to the model.

**Returns**

- **model**: Model instance

**Notes**

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

```
statsmodels.base.model.GenericLikelihoodModel.hessian
GenericLikelihoodModel.hessian(params)
```

Hessian of log-likelihood evaluated at params

```
statsmodels.base.model.GenericLikelihoodModel.information
GenericLikelihoodModel.information(params)
```

Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

```
statsmodels.base.model.GenericLikelihoodModel.initialize
GenericLikelihoodModel.initialize()

2.11. Internal Classes
"
statsmodels.base.model.GenericLikelihoodModel.jac
GenericLikelihoodModel.jac(params, **kwds)
Jacobian/Gradient of log-likelihood evaluated at params for each observation.

statsmodels.base.model.GenericLikelihoodModel.loglike
GenericLikelihoodModel.loglike(params)

statsmodels.base.model.GenericLikelihoodModel.loglikeobs
GenericLikelihoodModel.loglikeobs(params)

statsmodels.base.model.GenericLikelihoodModel.nloglike
GenericLikelihoodModel.nloglike(params)

statsmodels.base.model.GenericLikelihoodModel.predict
GenericLikelihoodModel.predict(params, exog=None, *args, **kwargs)
After a model has been fit predict returns the fitted values.
This is a placeholder intended to be overwritten by individual models.

statsmodels.base.model.GenericLikelihoodModel.reduceparams
GenericLikelihoodModel.reduceparams(params)

statsmodels.base.model.GenericLikelihoodModel.score
GenericLikelihoodModel.score(params)
Gradient of log-likelihood evaluated at params

Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>endog_names</td>
</tr>
<tr>
<td>exog_names</td>
</tr>
</tbody>
</table>

statsmodels.base.model.Results

class statsmodels.base.model.Results(model, params, **kw)
Class to contain model results

Parameters

- **model**: class instance
  the previously specified model instance
- **params**: array
  parameter estimates from the fit model

Methods

- initialize(model, params, **kw)
- predict([exog, transform])
  Call self.model.predict with self.params as the first argument.
statsmodels.base.model.Results.initialize
Results.initialize(model, params, **kwds)

statsmodels.base.model.Results.predict
Results.predict(exog=None, transform=True, *args, **kwargs)
Call self.model.predict with self.params as the first argument.

Parameters
exog : array-like, optional
The values for which you want to predict.

transform : bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model \( y \sim \log(x_1) + \log(x_2) \), and transform is True, then you can pass a data structure that contains \( x_1 \) and \( x_2 \) in their original form. Otherwise, you’d need to log the data first.

Returns
See self.model.predict:

statsmodels.base.model.LikelihoodModelResults
class statsmodels.base.model.LikelihoodModelResults(model, params, normalized_cov_params=None, scale=1.0)
Class to contain results from likelihood models

Parameters
model : LikelihoodModel instance or subclass instance
LikelihoodModelResults holds a reference to the model that is fit.

params : 1d array_like
parameter estimates from estimated model

normalized_cov_params : 2d array
Normalized (before scaling) covariance of params. \((\text{dot}(X,T,X))^{-1}\)

scale : float
For (some subset of models) scale will typically be the mean square error from the estimated model \((\sigma^2)\)

Returns
**Attributes**:

mle_retvals : dict
Contains the values returned from the chosen optimization method if full_output is True during the fit. Available only if the model is fit by maximum likelihood. See notes below for the output from the different methods.

mle_settings : dict
Contains the arguments passed to the chosen optimization method. Available if the model is fit by maximum likelihood. See LikelihoodModel.fit for more information.

model : model instance
LikelihoodResults contains a reference to the model that is fit.

params : ndarray
The parameters estimated for the model.
scale : float
    The scaling factor of the model given during instantiation.

tvalues : array
    The t-values of the standard errors.

Notes

The covariance of params is given by scale times normalized_cov_params.
Return values by solver if full_output is True during fit:

‘newton’

fopt [float] The value of the (negative) loglikelihood at its minimum.
iterations [int] Number of iterations performed.
score [ndarray] The score vector at the optimum.
Hessian [ndarray] The Hessian at the optimum.
warnflag [int] 1 if maxiter is exceeded. 0 if successful convergence.
allvecs [list] List of solutions at each iteration.

‘nm’

fopt [float] The value of the (negative) loglikelihood at its minimum.
iterations [int] Number of iterations performed.
warnflag [int] 1: Maximum number of function evaluations made. 2: Maximum number of
iterations reached.
allvecs [list] List of solutions at each iteration.

‘bfgs’

fopt [float] Value of the (negative) loglikelihood at its minimum.
gopt [float] Value of gradient at minimum, which should be near 0.
Hinv [ndarray] value of the inverse Hessian matrix at minimum. Note that this is just an ap-
proximation and will often be different from the value of the analytic Hessian.
fcalls [int] Number of calls to loglike.
gcalls [int] Number of calls to gradient/score.
warnflag [int] 1: Maximum number of iterations exceeded. 2: Gradient and/or function calls
are not changing.
allvecs [list] Results at each iteration.

‘lbfgs’

fopt [float] Value of the (negative) loglikelihood at its minimum.
gopt [float] Value of gradient at minimum, which should be near 0.
statsmodels Documentation, Release 0.6.0

fcalls  [int] Number of calls to loglike.

warnflag  [int] Warning flag:
  • 0 if converged
  • 1 if too many function evaluations or too many iterations
  • 2 if stopped for another reason


‘powell’

fopt  [float] Value of the (negative) loglikelihood at its minimum.

iterations  [int] Number of iterations performed.

fcalls  [int] Number of calls to loglike.

warnflag  [int] 1: Maximum number of function evaluations. 2: Maximum number of iterations.


allvecs  [list] Results at each iteration.

‘cg’

fopt  [float] Value of the (negative) loglikelihood at its minimum.

fcalls  [int] Number of calls to loglike.

gcalls  [int] Number of calls to gradient/score.

warnflag  [int] 1: Maximum number of iterations exceeded. 2: Gradient and/ or function calls not changing.


allvecs  [list] Results at each iteration.

‘ncg’

fopt  [float] Value of the (negative) loglikelihood at its minimum.

fcalls  [int] Number of calls to loglike.

gcalls  [int] Number of calls to gradient/score.

hcalls  [int] Number of calls to hessian.

warnflag  [int] 1: Maximum number of iterations exceeded.


allvecs  [list] Results at each iteration.

Methods

bse()

conf_int([alpha, cols, method])

Returns the confidence interval of the fitted parameters.

cov_params([r_matrix, column, scale, cov_p, ...])

Returns the variance/covariance matrix.

f_test(r_matrix[, q_matrix, cov_p, scale, ...])

Compute the F-test for a joint linear hypothesis.

Continued on next page
Table 2.20 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>initialize</code></td>
<td>Call self.model.initialize with self.params as the first argument.</td>
</tr>
<tr>
<td><code>llf()</code></td>
<td>Call self.loglike with self.params as the first argument.</td>
</tr>
<tr>
<td><code>load(fname)</code></td>
<td>Load a pickle. (class method)</td>
</tr>
<tr>
<td><code>normalized_cov_params()</code></td>
<td>Compute the normalized covariance parameters.</td>
</tr>
<tr>
<td><code>predict([exog, transform])</code></td>
<td>Call self.model.predict with self.params as the first argument.</td>
</tr>
<tr>
<td><code>pvalues()</code></td>
<td>Compute the p-values.</td>
</tr>
<tr>
<td><code>remove_data()</code></td>
<td>Remove data arrays, all nobs arrays from result and model.</td>
</tr>
<tr>
<td><code>save(fname[, remove_data])</code></td>
<td>Save a pickle of this instance.</td>
</tr>
<tr>
<td><code>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</code></td>
<td>Compute a t-test for a joint linear hypothesis of the form Rb = q</td>
</tr>
<tr>
<td><code>tvalues()</code></td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td><code>wald_test(r_matrix[, q_matrix, cov_p, ...])</code></td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
</tbody>
</table>

```python
statsmodels.base.model.LikelihoodModelResults.bse

static LikelihoodModelResults.bse()
```

```python
statsmodels.base.model.LikelihoodModelResults.conf_int

LikelihoodModelResults.conf_int (alpha=0.05, cols=None, method=’default’)

Returns the confidence interval of the fitted parameters.

**Parameters**

- `alpha` : float, optional
  
The `alpha` level for the confidence interval. i.e., The default `alpha` = .05 returns a 95% confidence interval.

- `cols` : array-like, optional
  
  `cols` specifies which confidence intervals to return

- `method` : string
  
  Not Implemented Yet Method to estimate the confidence interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

**Returns**

- `conf_int` : array
  
  Each row contains [lower, upper] confidence interval

**Notes**

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

**Examples**

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[5496529.48322745, 1467987.78596704],
       [177.02903529, 207.15277984],
       [0.1115811, 0.03994274],
       [-3.12506664, -0.91539297])
```
```python
>>> results.conf_int(cols=(2,3))
array([[-0.1115811 , 0.03994274],
       [-3.12506664, -0.91539297]])
```

### statsmodels.base.model.LikelihoodModelResults.cov_params

```python
statsmodels.base.model.LikelihoodModelResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)
```

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

**Parameters**

- `r_matrix`: array-like
  - Can be 1d, or 2d. Can be used alone or with other.
- `column`: array-like, optional
  - Must be used on its own. Can be 0d or 1d see below.
- `scale`: float, optional
  - Can be specified or not. Default is None, which means that the scale argument is taken from the model.
- `other`: array-like, optional
  - Can be used when `r_matrix` is specified.

**Returns** (The below are assumed to be in matrix notation.):

- `cov`: ndarray
  - If no argument is specified returns the covariance matrix of a model:
  
  \[(\text{scale})^2 (X^T X)^{-1}\]

  If contrast is specified it pre and post-multiplies as follows:

  \[(\text{scale}) \times \text{r_matrix} \times (X^T X)^{-1} \times \text{r_matrix}^T\]

  If contrast and other are specified returns:

  \[(\text{scale}) \times \text{r_matrix} \times (X^T X)^{-1} \times \text{other}^T\]

  If column is specified returns:

  \[(\text{scale}) \times (X^T X)^{-1} \times \text{column}\] if column is 0d:

  OR:

  \[(\text{scale}) \times (X^T X)^{-1} \times \text{column}[;\text{column}]\] if column is 1d:

### statsmodels.base.model.LikelihoodModelResults.f_test

```python
statsmodels.base.model.LikelihoodModelResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
```

Compute the F-test for a joint linear hypothesis.

**Parameters**

- `r_matrix`: array-like, str, or tuple
• array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
• str : The full hypotheses to test can be given as a string. See the examples.
• tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like

This is deprecated. See r_matrix and the examples for more information on new usage.
Can be either a scalar or a length p row vector. If omitted and r_matrix is an array,
q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given,
self.normalized_cov_params is used.

scale : float, optional

Default is 1.0 for no scaling.

invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,

r_matrix (pX pX.T) r_matrix.T

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]

This tests that each coefficient is jointly statistically significantly different from zero.

>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
df_denom=9, df_num=6>

Compare this to
>>> results.F
330.285392346658
>>> results.F_p
4.98403096572e-10

>>> B = np.array((
    [0, 0, 1, -1, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, -1]
))

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the
5th and 6th regressors are equal.

>>> print(results.f_test(B))
<

Alternatively, you can specify the hypothesis tests using a string

>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols

>>> dta = longley.load_pandas().data

>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'

>>> results = ols(formula, dta).fit()

>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'

>>> f_test = results.f_test(hypotheses)

statsmodels.base.model.LikelihoodModelResults.initialize
LikelihoodModelResults.initialize(model, params, **kwds)

statsmodels.base.model.LikelihoodModelResults.llf
static LikelihoodModelResults.llf()

statsmodels.base.model.LikelihoodModelResults.load
classmethod LikelihoodModelResults.load(fname)
    load a pickle, (class method)

    Parameters fname : string or filehandle
        fname can be a string to a file path or filename, or a filehandle.

    Returns unpickled instance :

statsmodels.base.model.LikelihoodModelResults.normalized_cov_params
LikelihoodModelResults.normalized_cov_params()

statsmodels.base.model.LikelihoodModelResults.predict
LikelihoodModelResults.predict(exog=None, transform=True, *args, **kwargs)
    Call self.model.predict with self.params as the first argument.

    Parameters exog : array-like, optional
        The values for which you want to predict.

    transform : bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model \( y \sim \log(x1) + \log(x2) \), and transform is True, then you can pass a data structure that contains \( x1 \) and \( x2 \) in their original form. Otherwise, you’d need to log the data first.

Returns See self.model.predict:

```python
class LikelihoodModelResults:
    @staticmethod
    def pvalues():
        return ...

    remove_data():
        remove data arrays, all nobs arrays from result and model

        Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

        Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

        The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

    save(fname, remove_data=False): save a pickle of this instance

    Parameters
    ----------
    fname : string or filehandle
        fname can be a string to a file path or filename, or a filehandle.
    remove_data : bool
        If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

    Notes
    -----
    If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

    t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None): Compute a t-test for a joint linear hypothesis of the form \( Rb = q \)

    Parameters
    ----------
    r_matrix : array-like, str, tuple
        • array : If an array is given, a \( p \times k \) 2d array or length \( k \) 1d array specifying the linear restrictions.
• `str` : The full hypotheses to test can be given as a string. See the examples.

• `tuple` : A tuple of arrays in the form `(R, q)`, since `q_matrix` is deprecated.

`q_matrix` : array-like or scalar, optional

This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length `p` row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.

`cov_p` : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, `self.normalized_cov_params` is used.

`scale` : float, optional

An optional `scale` to use. Default is the scale specified by the model fit.

`use_t` : bool, optional

If `use_t` is None, then the default of the model is used. If `use_t` is True, then the p-values are based on the t distribution. If `use_t` is False, then the p-values are based on the normal distribution.

See also:

`tvalues` individual t statistics

`f_test` for F tests

`patsy.DesignInfo.linear_constraint`

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```

Tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_test = results.t_test(r) >>> print(T_test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9> >>> T_test.effect -1829.2025687192481 >>> T_test.sd 455.39079425193762 >>> T_test.tvalue -4.0167754636411717 >>> T_test.pvalue 0.0015163772380899498
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```
statsmodels.base.model.LikelihoodModelResults.tvalues

static LikelihoodModelResults.tvalues()  
Return the t-statistic for a given parameter estimate.

statsmodels.base.model.LikelihoodModelResults.wald_test
LikelihoodModelResults.wald_test( r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)

Compute a Wald-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple

• array : An r x k array where r is the number of restrictions to test and k is the number of regressors.

• str : The full hypotheses to test can be given as a string. See the examples.

• tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

Default is 1.0 for no scaling.

invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely, 

r_matrix (pX pX.T) r_matrix.T

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
Methods

- `aic()`
- `bic()`
- `bootstrap([nrep, method, disp, store])` simple bootstrap to get mean and variance of estimator
- `bsejac()` standard deviation of parameter estimates based on covjac
- `bsejhj()` standard deviation of parameter estimates based on covHJH
- `covjac()` covariance of parameters based on outer product of jacobian of
- `covjhj()` covariance of parameters based on HJJH
- `df_modelwc()`
- `get_nlfun(fun)`
- `hessv()` cached Hessian of log-likelihood
- `jacv()` cached Jacobian of log-likelihood

**statsmodels.base.model.ResultMixin.aic**
```
static ResultMixin.aic()
```

**statsmodels.base.model.ResultMixin.bic**
```
static ResultMixin.bic()
```

**statsmodels.base.model.ResultMixin.bootstrap**
```
ResultMixin.bootstrap(nrep=100, method='nm', disp=0, store=1)
```
simple bootstrap to get mean and variance of estimator
see notes

**Parameters**
- `nrep` : int
  number of bootstrap replications
- `method` : str
  optimization method to use
- `disp` : bool
  If true, then optimization prints results
- `store` : bool
  If true, then parameter estimates for all bootstrap iterations are attached in
  self.bootstrap_results

**Returns**
- `mean` : array
  mean of parameter estimates over bootstrap replications
- `std` : array
  standard deviation of parameter estimates over bootstrap replications

**Notes**

This was mainly written to compare estimators of the standard errors of the parameter estimates. It uses in-
dependent random sampling from the original endog and exog, and therefore is only correct if observations
are independently distributed.
This will be moved to apply only to models with independently distributed observations.

```python
statsmodels.base.model.ResultMixin.bsejac
static ResultMixin.bsejac()
    standard deviation of parameter estimates based on covjac

statsmodels.base.model.ResultMixin.bsejhj
static ResultMixin.bsejhj()
    standard deviation of parameter estimates based on covHJH

statsmodels.base.model.ResultMixin.covjac
static ResultMixin.covjac()
    covariance of parameters based on outer product of jacobian of log-likelihood

statsmodels.base.model.ResultMixin.covjhj
static ResultMixin.covjhj()
    covariance of parameters based on HJJH
dot product of Hessian, Jacobian, Jacobian, Hessian of likelihood
    name should be covjhj

statsmodels.base.model.ResultMixin.df_modelwc
static ResultMixin.df_modelwc()

statsmodels.base.model.ResultMixin.get_nlfun
ResultMixin.get_nlfun(fun)

statsmodels.base.model.ResultMixin.hessv
static ResultMixin.hessv()
    cached Hessian of log-likelihood

statsmodels.base.model.ResultMixin.jacv
static ResultMixin.jacv()
    cached Jacobian of log-likelihood
```

```python
statsmodels.base.model.GenericLikelihoodModelResults

class statsmodels.base.model.GenericLikelihoodModelResults(model, mlefit)
A results class for the discrete dependent variable models.

.. Warning ::

    The following description has not been updated to this version/class. Where are AIC, BIC, ....? docstring looks like copy from discretemod

    Parameters
        model : A DiscreteModel instance
        mlefit : instance of LikelihoodResults

```
This contains the numerical optimization results as returned by LikelihoodModel.fit(), in a superclass of GenericLikelihoodModels.

**Returns** *Attributes*:

- **aic**: float
  - Akaike information criterion. \(-2^*(\text{llf} - p)\) where \(p\) is the number of regressors including the intercept.

- **bic**: float
  - Bayesian information criterion. \(-2^*\text{llf} + \ln(nobs)*p\) where \(p\) is the number of regressors including the intercept.

- **bse**: array
  - The standard errors of the coefficients.

- **df_resid**: float
  - See model definition.

- **df_model**: float
  - See model definition.

- **fitted_values**: array
  - Linear predictor XB.

- **llf**: float
  - Value of the loglikelihood

- **llnull**: float
  - Value of the constant-only loglikelihood

- **llr**: float
  - Likelihood ratio chi-squared statistic; \(-2*(\text{llnull} - \text{llf})\)

- **llr_pvalue**: float
  - The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr has a chi-squared distribution with degrees of freedom df_model.

- **prsquared**: float
  - McFadden’s pseudo-R-squared. \(1 - (\text{llf}/\text{llnull})\)

**Methods**

- **aic()**
- **bic()**
- **bootstrap([nrep, method, disp, store])**
  - simple bootstrap to get mean and variance of estimator
- **bse()**
- **bsejac()**
- **bsejh()**
- **conf_int([alpha, cols, method])**
  - standard deviation of parameter estimates based on covjac
  - standard deviation of parameter estimates based on covHJH
  - Returns the confidence interval of the fitted parameters.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cov_params([r_matrix, column, scale, cov_p, ...])</code></td>
<td>Returns the variance/covariance matrix.</td>
</tr>
<tr>
<td><code>covjac()</code></td>
<td>covariance of parameters based on outer product of jacobian of</td>
</tr>
<tr>
<td><code>covjhj()</code></td>
<td>covariance of parameters based on HJJH</td>
</tr>
<tr>
<td><code>df_modelwc()</code></td>
<td></td>
</tr>
<tr>
<td><code>f_test(r_matrix[, q_matrix, cov_p, scale, ...])</code></td>
<td>Compute the F-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td><code>get_nlfun(fun)</code></td>
<td>cached Hessian of log-likelihood</td>
</tr>
<tr>
<td><code>hessv()</code></td>
<td>cached Jacobian of log-likelihood</td>
</tr>
<tr>
<td><code>initialize(model, params, **kwd)</code></td>
<td></td>
</tr>
<tr>
<td><code>jacv()</code></td>
<td>load a pickle, (class method)</td>
</tr>
<tr>
<td><code>llf()</code></td>
<td>Call self.model.predict with self.params as the first argument.</td>
</tr>
<tr>
<td><code>load(fname)</code></td>
<td>remove data arrays, all nobs arrays from result and model</td>
</tr>
<tr>
<td><code>normalized_cov_params()</code></td>
<td>save a pickle of this instance</td>
</tr>
<tr>
<td><code>predict([exog, transform])</code></td>
<td>Summarize the Regression Results</td>
</tr>
<tr>
<td><code>pvalues()</code></td>
<td>Compute a t-test for a joint linear hypothesis of the form Rb = q</td>
</tr>
<tr>
<td><code>remove_data()</code></td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td><code>save(fname[, remove_data])</code></td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td><code>summary([yname, xname, title, alpha])</code></td>
<td></td>
</tr>
<tr>
<td><code>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</code></td>
<td></td>
</tr>
<tr>
<td><code>tvalues()</code></td>
<td></td>
</tr>
<tr>
<td><code>wald_test(r_matrix[, q_matrix, cov_p, ...])</code></td>
<td></td>
</tr>
</tbody>
</table>

```python
class GenericLikelihoodModelResults:
    @classmethod
    def aic(cls):
        return cls.aic()
    @classmethod
    def bic(cls):
        return cls.bic()
    def bootstrap(self, nrep=100, method='nm', disp=0, store=1):
        return self.bootstrap_results
```

**Parameters**
- `nrep`: int
  - number of bootstrap replications
- `method`: str
  - optimization method to use
- `disp`: bool
  - If true, then optimization prints results
- `store`: bool
  - If true, then parameter estimates for all bootstrap iterations are attached in self.bootstrap_results

**Returns**
- `mean`: array
  - mean of parameter estimates over bootstrap replications
- `std`: array
  - standard deviation of parameter estimates over bootstrap replications
Notes

This was mainly written to compare estimators of the standard errors of the parameter estimates. It uses independent random sampling from the original endog and exog, and therefore is only correct if observations are independently distributed.

This will be moved to apply only to models with independently distributed observations.

statsmodels.base.model.GenericLikelihoodModelResults.bse
static GenericLikelihoodModelResults.bse()

statsmodels.base.model.GenericLikelihoodModelResults.bsejac
static GenericLikelihoodModelResults.bsejac()
standard deviation of parameter estimates based on covjac

statsmodels.base.model.GenericLikelihoodModelResults.bsejhj
static GenericLikelihoodModelResults.bsejhj()
standard deviation of parameter estimates based on covHJH

statsmodels.base.model.GenericLikelihoodModelResults.conf_int
GenericLikelihoodModelResults.conf_int(alpha=0.05, cols=None, method='default')
Returns the confidence interval of the fitted parameters.

Parameters alpha : float, optional
The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

cols : array-like, optional
cols specifies which confidence intervals to return

method : string
Not Implemented Yet Method to estimate the confidence interval. “Default”: uses self.bse which is based on inverse Hessian for MLE “jhj”: “jac”: “boot-bse” “boot_quant” “profile”

Returns conf_int : array
Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
```
array([[ 5496529.48322745, 1467987.78596704],
[ -177.02903529, 207.15277984],
[ -0.1115811, 0.03994274],
[ -3.12506664, -0.91539297],
[ -1.5179487 , -0.54850503],
[ -0.56251721, 0.460309 ],
[ 798.7875153 , 2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[-0.1115811 , 0.03994274],
[ -3.12506664, -0.91539297]])

statsmodels.base.model.GenericLikelihoodModelResults.cov_params

GenericLikelihoodModelResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters

r_matrix : array-like
Can be 1d, or 2d. Can be used alone or with other.

column : array-like, optional
Must be used on its own. Can be 0d or 1d see below.

scale : float, optional
Can be specified or not. Default is None, which means that the scale argument is taken from the model.

other : array-like, optional
Can be used when r_matrix is specified.

Returns

(The below are assumed to be in matrix notation.) :

cov : ndarray
If no argument is specified returns the covariance matrix of a model :
(scale)*(X.T X)^(-1) :

If contrast is specified it pre and post-multiplies as follows :
(scale) * r_matrix (X.T X)^(-1) r_matrix.T :

If contrast and other are specified returns :
(scale) * r_matrix (X.T X)^(-1) other.T :

If column is specified returns :
(scale) * (X.T X)^(-1)[column,column] if column is 0d :

OR :
(scale) * (X.T X)^(-1)[column][:,column] if column is 1d :
statsmodels.base.model.GenericLikelihoodModelResults.covjac

static GenericLikelihoodModelResults.covjac()
covariance of parameters based on outer product of jacobian of log-likelihood

statsmodels.base.model.GenericLikelihoodModelResults.covjhj

static GenericLikelihoodModelResults.covjhj()
covariance of parameters based on HJJH
dot product of Hessian, Jacobian, Jacobian, Hessian of likelihood
name should be covhjh

statsmodels.base.model.GenericLikelihoodModelResults.df_modelwc

static GenericLikelihoodModelResults.df_modelwc()

statsmodels.base.model.GenericLikelihoodModelResults.f_test

GenericLikelihoodModelResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
Compute the F-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
  This is deprecated. See r_matrix and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
  Default is 1.0 for no scaling.

invcov : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,
r_matrix (pX pX.T) r_matrix.T
is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>
```  

Compare this to

```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10
```  

```python
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>
```  

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```
statsmodels.base.model.GenericLikelihoodModelResults.get_nlfun
GenericLikelihoodModelResults.get_nlfun (fun)
```

```
statsmodels.base.model.GenericLikelihoodModelResults.hessv
static GenericLikelihoodModelResults.hessv ()
cached Hessian of log-likelihood
```

```
statsmodels.base.model.GenericLikelihoodModelResults.initialize
GenericLikelihoodModelResults.initialize (model, params, **kwds)
```
statsmodels.base.model.GenericLikelihoodModelResults.jacv
static GenericLikelihoodModelResults.jacv()
cached Jacobian of log-likelihood

statsmodels.base.model.GenericLikelihoodModelResults.llf
static GenericLikelihoodModelResults.llf()

statsmodels.base.model.GenericLikelihoodModelResults.load
classmethod GenericLikelihoodModelResults.load(fname)
load a pickle, (class method)

Parameters
fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

Returns
unpickled instance :

statsmodels.base.model.GenericLikelihoodModelResults.normalized_cov_params
GenericLikelihoodModelResults.normalized_cov_params()

statsmodels.base.model.GenericLikelihoodModelResults.predict
GenericLikelihoodModelResults.predict (exog=None, transform=True, *args, **kwargs)
Call self.model.predict with self.params as the first argument.

Parameters
exog : array-like, optional
The values for which you want to predict.
transform : bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model \( y \sim \log(x1) + \log(x2) \), and transform is True, then you can pass a data structure that contains \( x1 \) and \( x2 \) in their original form. Otherwise, you’d need to log the data first.

Returns
See self.model.predict :

statsmodels.base.model.GenericLikelihoodModelResults.pvalues
static GenericLikelihoodModelResults.pvalues()

statsmodels.base.model.GenericLikelihoodModelResults.remove_data
GenericLikelihoodModelResults.remove_data()
remove data arrays, all nobs arrays from result and model
This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.
The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

`statsmodels.base.model.GenericLikelihoodModelResults.save`

`GenericLikelihoodModelResults.save(fname, remove_data=False)`

Save a pickle of this instance

**Parameters**

- **fname**: string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- **remove_data**: bool
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

`statsmodels.base.model.GenericLikelihoodModelResults.summary`

`GenericLikelihoodModelResults.summary(yname=None, xname=None, title=None, alpha=0.05)`

Summarize the Regression Results

**Parameters**

- **yname**: string, optional
  
  Default is y

- **xname**: list of strings, optional
  
  Default is var_## for ## in p the number of regressors

- **title**: string, optional
  
  Title for the top table. If not None, then this replaces the default title

- **alpha**: float
  
  significance level for the confidence intervals

**Returns**

- **smry**: Summary instance
  
  this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

- `statsmodels.iolib.summary.Summary` class to hold summary results

`statsmodels.base.model.GenericLikelihoodModelResults.t_test`

`GenericLikelihoodModelResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)`

Compute a t-test for a joint linear hypothesis of the form Rb = q

**Parameters**

- **r_matrix**: array-like, str, tuple
array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.

str : The full hypotheses to test can be given as a string. See the examples.

tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics
f_test for F tests
patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]

r tests that the coefficients on the 5th and 6th independent variable are the same.
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> data = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, data).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
```
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)

statsmodels.base.model.GenericLikelihoodModelResults.tvalues
static GenericLikelihoodModelResults.tvalues()
Return the t-statistic for a given parameter estimate.

statsmodels.base.model.GenericLikelihoodModelResults.wald_test
GenericLikelihoodModelResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
    • array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
    This is deprecated. See r_matrix and the examples for more information on new usage.
    Can be either a scalar or a length p row vector. If omitted and r_matrix is an array,
    q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
    An alternative estimate for the parameter covariance matrix. If None is given,
    self.normalized_cov_params is used.

scale : float, optional
    Default is 1.0 for no scaling.

invcov : array-like, optional
    A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool
    If True, then the F-distribution is used. If False, then the asymptotic distribution,
    chisquare is used. The test statistic is proportionally adjusted for the distribution by
    the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes
The matrix r_matrix is assumed to be non-singular. More precisely,
r_matrix (pX pX.T) r_matrix.T
is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
Linear Model

Generalized Linear Model

2.11. Internal Classes
3.1 Linear Regression

Linear models with independently and identically distributed errors, and for errors with heteroscedasticity or autocorrelation. This module allows estimation by ordinary least squares (OLS), weighted least squares (WLS), generalized least squares (GLS), and feasible generalized least squares with autocorrelated AR(p) errors.

See Module Reference for commands and arguments.

3.1.1 Examples

```python
# Load modules and data
import numpy as np
import statsmodels.api as sm
spector_data = sm.datasets.spector.load()
spector_data.exog = sm.add_constant(spector_data.exog, prepend=False)

# Fit and summarize OLS model
mod = sm.OLS(spector_data.endog, spector_data.exog)
res = mod.fit()
print res.summary()
```

Detailed examples can be found here:

3.1.2 Technical Documentation

The statistical model is assumed to be

\[ Y = X\beta + \mu, \text{ where } \mu \sim N(0, \sigma^2 \Sigma) \]

depending on the assumption on \( \Sigma \), we have currently four classes available

- GLS: generalized least squares for arbitrary covariance \( \Sigma \)
- OLS: ordinary least squares for i.i.d. errors \( \Sigma = I \)
- WLS: weighted least squares for heteroskedastic errors \( \text{diag}(\Sigma) \)
- GLSAR: feasible generalized least squares with autocorrelated AR(p) errors \( \Sigma = \Sigma(\rho) \)

All regression models define the same methods and follow the same structure, and can be used in a similar fashion. Some of them contain additional model specific methods and attributes.

GLS is the superclass of the other regression classes.
References

General reference for regression models:


Econometrics references for regression models:


Attributes

The following is more verbose description of the attributes which is mostly common to all regression classes

pinv_wexog  [array]

$\text{pinv\_wexog}$ is the $p \times n$ Moore-Penrose pseudoinverse of the whitened design matrix. Approximately equal to 

$$(X^T \Sigma^{-1} X)^{-1} X^T \Psi$$

where $\Psi$ is given by $\Psi \Psi^T = \Sigma^{-1}$

cholsimgainv  [array]

$n \times n$ upper triangular matrix such that 

$\Psi \Psi^T = \Sigma^{-1}$

$\text{cholsimgainv} = \Psi^T$

df_model  [float] The model degrees of freedom is equal to $p - 1$, where $p$ is the number of regressors. Note that the intercept is not counted as using a degree of freedom here.

df_resid  [float] The residual degrees of freedom is equal to the number of observations $n$ less the number of parameters $p$. Note that the intercept is counted as using a degree of freedom here.

llf  [float] The value of the likelihood function of the fitted model.

nobs  [float] The number of observations $n$

normalized_cov_params  [array]

A $p \times p$ array

$$(X^T \Sigma^{-1} X)^{-1}$$

sigma  [array]

$\text{sigma}$ is the $n \times n$ structure of the covariance matrix of the error terms

$\mu \sim N(0, \sigma^2 \Sigma)$

wexog  [array]

$\text{wexog}$ is the whitened design matrix.

$\Psi^T X$

wendog  [array]

The whitened response variable.

$\Psi^T Y$
## 3.1.3 Module Reference

### Model Classes

- **OLS**(endog[, exog, missing, hasconst])
  A simple ordinary least squares model.
- **GLS**(endog, exog[, sigma, missing, hasconst])
  Generalized least squares model with a general covariance structure.
- **WLS**(endog, exog[, weights, missing, hasconst])
  A regression model with diagonal but non-identity covariance structure.
- **GLSAR**(endog[, exog, rho, missing])
  A regression model with an AR(p) covariance structure.
- **yule_walker**(X[, order, method, df, inv, demean])
  Estimate AR(p) parameters from a sequence X using Yule-Walker equation.

```python
class statsmodels.regression.linear_model.OLS
(endog, exog=None, missing='none', hasconst=None)
```

A simple ordinary least squares model.

**Parameters**
- **endog**: array-like
  1-d endogenous response variable. The dependent variable.
- **exog**: array-like
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant`.
- **missing**: str
  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’
- **hasconst**: None or bool
  Indicates whether the RHS includes a user-supplied constant. If True, a constant is not checked for and k_constant is set to 1 and all result statistics are calculated as if a constant is present. If False, a constant is not checked for and k_constant is set to 0.

**Notes**

No constant is added by the model unless you are using formulas.

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> Y = [1,3,4,5,2,3,4]
>>> X = range(1,8)
>>> X = sm.add_constant(X)
>>> model = sm.OLS(Y,X)
>>> results = model.fit()
>>> results.params
array([ 2.14285714, 0.25])
>>> results.tvalues
```

---

### 3.1. Linear Regression

---

127
array([ 1.87867287, 0.98019606])
>>> print(results.t_test([1, 0]))
<\text{T test: effect=\text{array([ 2.14285714])}, sd=\text{array([[[ 1.14062282]]]), t=\text{array([[ 1.87867287]])}}, p=\text{array([[ 0.05953974]])}, df\text{\text{denom}}=5, df\text{\text{num}}=2>}

Attributes

<table>
<thead>
<tr>
<th>weights</th>
<th>scalar</th>
<th>Has an attribute weights = array(1.0) due to inheritance from WLS.</th>
</tr>
</thead>
</table>

Methods

<table>
<thead>
<tr>
<th>fit(method)</th>
<th>Full fit of the model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_formula(formula, data[, subset])</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>hessian(params)</td>
<td>The Hessian matrix of the model</td>
</tr>
<tr>
<td>information(params)</td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td>initialize()</td>
<td></td>
</tr>
<tr>
<td>loglike(params)</td>
<td>The likelihood function for the classical OLS model.</td>
</tr>
<tr>
<td>predict(params[, exog])</td>
<td>Return linear predicted values from a design matrix.</td>
</tr>
<tr>
<td>score(params)</td>
<td>Score vector of model.</td>
</tr>
<tr>
<td>whiten(Y)</td>
<td>OLS model whitener does nothing: returns Y.</td>
</tr>
</tbody>
</table>

\text{statsmodels.regression.linear_model.OLS.fit}

\text{OLS\text{.}fit(method=\text{\textquotesingle pinv\textquotesingle}, **kwargs)}

Full fit of the model.

The results include an estimate of covariance matrix, (whitened) residuals and an estimate of scale.

- **Parameters**
  - method : str
    - Can be \text{\textquotesingle pinv\textquotesingle}, \text{\textquotesingle qr\textquotesingle}. \text{\textquotesingle pinv\textquotesingle} uses the Moore-Penrose pseudoinverse to solve the least squares problem. \text{\textquotesingle qr\textquotesingle} uses the QR factorization.

- **Returns**
  - A \text{RegressionResults} class instance.

- See also:
  - \text{regression\text{.}RegressionResults}

\text{Notes}

The fit method uses the pseudoinverse of the design/exogenous variables to solve the least squares minimization.

\text{statsmodels.regression.linear_model.OLS.from_formula}

\text{classmethod OLS\text{.from_formula(formula, data, subset=\text{None}, *args, **kwargs)}}

Create a Model from a formula and dataframe.

- **Parameters**
  - formula : str or generic Formula object
    - The formula specifying the model
**data**: array-like

The data for the model. See Notes.

**subset**: array-like

An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a *pandas.DataFrame*

**args**: extra arguments

These are passed to the model

**kwargs**: extra keyword arguments

These are passed to the model.

**Returns**

**model**: Model instance

**Notes**

data must define `__getitem__` with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

```python
statsmodels.regression.linear_model.OLS.hessian

OLS.hessian(params)

The Hessian matrix of the model
```

```python
statsmodels.regression.linear_model.OLS.information

OLS.information(params)

Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.
```

```python
statsmodels.regression.linear_model.OLS.initialize

OLS.initialize()
```

```python
statsmodels.regression.linear_model.OLS.loglike

OLS.loglike(params)

The likelihood function for the classical OLS model.

**Parameters**

**params**: array-like

The coefficients with which to estimate the log-likelihood.

**Returns**

The concentrated likelihood function evaluated at params.

```python
statsmodels.regression.linear_model.OLS.predict

OLS.predict(params, exog=None)

Return linear predicted values from a design matrix.

**Parameters**

**params**: array-like

Parameters of a linear model

**exog**: array-like, optional.

Design / exogenous data. Model exog is used if None.

```

### 3.1. Linear Regression
Returns An array of fitted values:

Notes

If the model has not yet been fit, params is not optional.

```
statsmodels.regression.linear_model.OLS.score
OLS.score(params)
```

Score vector of model.
The gradient of logL with respect to each parameter.

```
statsmodels.regression.linear_model.OLS.whiten
OLS.whiten(Y)
```

OLS model whitener does nothing: returns Y.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>df_model</td>
<td>The model degree of freedom, defined as the rank of the regressor</td>
</tr>
<tr>
<td>df_resid</td>
<td>The residual degree of freedom, defined as the number of observations</td>
</tr>
<tr>
<td>endog_names</td>
<td></td>
</tr>
<tr>
<td>exog_names</td>
<td></td>
</tr>
</tbody>
</table>

```
statsmodels.regression.linear_model.GLS
```

class `statsmodels.regression.linear_model.GLS` *(endog, exog, sigma=None, missing='none', hasconst=None)*

Generalized least squares model with a general covariance structure.

**Parameters**

- **endog** : array-like
  1-d endogenous response variable. The dependent variable.

- **exog** : array-like
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant`.

- **sigma** : scalar or array
  `sigma` is the weighting matrix of the covariance. The default is None for no scaling. If `sigma` is a scalar, it is assumed that `sigma` is an n x n diagonal matrix with the given scalar, `sigma` as the value of each diagonal element. If `sigma` is an n-length vector, then `sigma` is assumed to be a diagonal matrix with the given `sigma` on the diagonal. This should be the same as WLS.

- **missing** : str
  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’

- **hasconst** : None or bool
  Indicates whether the RHS includes a user-supplied constant. If True, a constant is not checked for and k_constant is set to 1 and all result
**Attributes**:  

- `pinv_wexog`: array  
  `pinv_wexog` is the p x n Moore-Penrose pseudoinverse of `wexog`.  
- `cholsimgainv`: array  
  The transpose of the Cholesky decomposition of the pseudoinverse.  
- `df_model`: float  
  p - 1, where p is the number of regressors including the intercept. of freedom.  
- `df_resid`: float  
  Number of observations n less the number of parameters p.  
- `llf`: float  
  The value of the likelihood function of the fitted model.  
- `nobs`: float  
  The number of observations n.  
- `normalized_cov_params`: array  
  p x p array \( (X^T \Sigma^{-1} X)^{-1} \)  
- `results`: RegressionResults instance  
  A property that returns the RegressionResults class if fit.  
- `sigma`: array  
  `sigma` is the n x n covariance structure of the error terms.  
- `wexog`: array  
  Design matrix whitened by `cholsigmainv`  
- `wendog`: array  
  Response variable whitened by `cholsigmainv`

**Notes**

If sigma is a function of the data making one of the regressors a constant, then the current postestimation statistics will not be correct.

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> ols_resid = sm.OLS(data.endog, data.exog).fit().resid
>>> res_fit = sm.OLS(ols_resid[1:], ols_resid[:-1]).fit()
>>> rho = res_fit.params
```
rho is a consistent estimator of the correlation of the residuals from an OLS fit of the longley data. It is assumed that this is the true rho of the AR process data.

```python
>>> from scipy.linalg import toeplitz
>>> order = toeplitz(np.arange(16))
>>> sigma = rho**order
```

sigma is an n x n matrix of the autocorrelation structure of the data.

```python
>>> gls_model = sm.GLS(data.endog, data.exog, sigma=sigma)
>>> gls_results = gls_model.fit()
>>> print(gls_results.summary())
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit(method)</td>
<td>Full fit of the model.</td>
</tr>
<tr>
<td>from_formula(formula, data[, subset])</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>hessian(params)</td>
<td>The Hessian matrix of the model.</td>
</tr>
<tr>
<td>information(params)</td>
<td>Fisher information matrix of model.</td>
</tr>
<tr>
<td>initialize()</td>
<td></td>
</tr>
<tr>
<td>loglike(params)</td>
<td>Returns the value of the Gaussian log-likelihood function at params.</td>
</tr>
<tr>
<td>predict(params[, exog])</td>
<td>Return linear predicted values from a design matrix.</td>
</tr>
<tr>
<td>score(params)</td>
<td>Score vector of model.</td>
</tr>
<tr>
<td>whiten(X)</td>
<td>GLS whiten method.</td>
</tr>
</tbody>
</table>

**statsmodels.regression.linear_model.GLS.fit**

GLS.fit(method='pinv', **kwargs)

Full fit of the model.

The results include an estimate of covariance matrix, (whitened) residuals and an estimate of scale.

**Parameters**

- method : str
  Can be “pinv”, “qr”. “pinv” uses the Moore-Penrose pseudoinverse to solve the least squares problem. “qr” uses the QR factorization.

**Returns**

- RegressionResults class instance.

**See also:**

- regression.RegressionResults

**Notes**

The fit method uses the pseudoinverse of the design/exogenous variables to solve the least squares minimization.

**statsmodels.regression.linear_model.GLS.from_formula**

GLS.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

**Parameters**

- formula : str or generic Formula object
  The formula specifying the model

- data : array-like

---

132 Chapter 3. Table of Contents
The data for the model. See Notes.

subset : array-like

An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

args : extra arguments

These are passed to the model

kwargs : extra keyword arguments

These are passed to the model.

Returns model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

statsmodels.regression.linear_model.GLS.hessian

GLS.hessian(params)

The Hessian matrix of the model

statsmodels.regression.linear_model.GLS.information

GLS.information(params)

Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

statsmodels.regression.linear_model.GLS.initialize

GLS.initialize()

statsmodels.regression.linear_model.GLS.loglike

GLS.loglike(params)

Returns the value of the Gaussian log-likelihood function at params.

Given the whitened design matrix, the log-likelihood is evaluated at the parameter vector params for the dependent variable endog.

Parameters params : array-like

The parameter estimates

Returns loglike : float

The value of the log-likelihood function for a GLS Model.
Notes

The log-likelihood function for the normal distribution is

\[-\frac{n}{2} \log \left( (Y - \hat{Y})' (Y - \hat{Y}) \right) - \frac{n}{2} \left( 1 + \log \left( \frac{2\pi}{n} \right) \right) - \frac{1}{2} \log (|\Sigma|)\]

Y and \(\hat{Y}\) are whitened.

statsmodels.regression.linear_model.GLS.predict

GLS.predict (params, exog=None)

Return linear predicted values from a design matrix.

Parameters
- **params**: array-like
  Parameters of a linear model
- **exog**: array-like, optional.
  Design / exogenous data. Model exog is used if None.

Returns
- An array of fitted values:

Notes

If the model has not yet been fit, params is not optional.

statsmodels.regression.linear_model.GLS.score

GLS.score (params)

Score vector of model.

The gradient of \(\log L\) with respect to each parameter.

statsmodels.regression.linear_model.GLS.whiten

GLS.whiten (X)

GLS whiten method.

Parameters
- **X**: array-like
  Data to be whitened.

Returns
- np.dot(cholsigmainv,X):

See also:

regression.GLS

Attributes

- **df_model**: The model degree of freedom, defined as the rank of the regressor
- **df_resid**: The residual degree of freedom, defined as the number of observations
- **endog_names**
- **exog_names**
class `statsmodels.regression.linear_model.WLS`(endog, exog, weights=1.0, missing='none', hasconst=None)

A regression model with diagonal but non-identity covariance structure.

The weights are presumed to be (proportional to) the inverse of the variance of the observations. That is, if the variables are to be transformed by 1/sqrt(W) you must supply weights = 1/W.

**Parameters**

- **endog**: array-like
  1-d endogenous response variable. The dependent variable.
- **exog**: array-like
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant`.
- **weights**: array-like, optional
  1d array of weights. If you supply 1/W then the variables are pre-multiplied by 1/sqrt(W). If no weights are supplied the default value is 1 and WLS results are the same as OLS.
- **missing**: str
  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’
- **hasconst**: None or bool
  Indicates whether the RHS includes a user-supplied constant. If True, a constant is not checked for and k_constant is set to 1 and all result statistics are calculated as if a constant is present. If False, a constant is not checked for and k_constant is set to 0.

**Notes**

If the weights are a function of the data, then the post estimation statistics such as fvalue and mse_model might not be correct, as the package does not yet support no-constant regression.

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> Y = [1,3,4,5,2,3,4]
>>> X = range(1,8)
>>> X = sm.add_constant(X)
>>> wls_model = sm.WLS(Y,X, weights=list(range(1,8)))
>>> results = wls_model.fit()
>>> results.params
array([ 2.91666667, 0.09523810])
>>> results.tvalues
array([2.06526520, 0.35684428])
>>> print(results.t_test([1, 0]))
<T test: effect=array([ 2.91666667]), sd=array([[ 1.41224801]]), t=array([[ 2.06526520]]), p=array([[ 0.04690139]])>
>>> print(results.f_test([0, 1]))
<F test: F=array([[ 0.12733784]]), p=[[ 0.73577409]], df_denom=5, df_num=1>
```
Attributes

weights
array
The stored weights supplied as an argument.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit([method])</code></td>
<td>Full fit of the model.</td>
</tr>
<tr>
<td><code>from_formula(formula, data[, subset])</code></td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td><code>hessian(params)</code></td>
<td>The Hessian matrix of the model.</td>
</tr>
<tr>
<td><code>information(params)</code></td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td><code>initialize()</code></td>
<td></td>
</tr>
<tr>
<td><code>loglike(params)</code></td>
<td>Returns the value of the gaussian log-likelihood function at params.</td>
</tr>
<tr>
<td><code>predict(params[, exog])</code></td>
<td>Return linear predicted values from a design matrix.</td>
</tr>
<tr>
<td><code>score(params)</code></td>
<td>Score vector of model.</td>
</tr>
<tr>
<td><code>whiten(X)</code></td>
<td>Whitener for WLS model, multiplies each column by sqrt(self.weights)</td>
</tr>
</tbody>
</table>

**statsmodels.regression.linear_model.WLS.fit**

WLS.fit (method='pinv', **kwargs)

Full fit of the model.

Parameters

- **method**: str
  
  Can be “pinv”, “qr”. “pinv” uses the Moore-Penrose pseudoinverse to solve the least squares problem. “qr” uses the QR factorization.

Returns

- A RegressionResults class instance.

See also:

- regression.RegressionResults

Notes

The fit method uses the pseudoinverse of the design/exogenous variables to solve the least squares minimization.

**statsmodels.regression.linear_model.WLS.from_formula**

classmethod WLS.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

Parameters

- **formula**: str or generic Formula object
  
  The formula specifying the model

- **data**: array-like
  
  The data for the model. See Notes.

- **subset**: array-like
  
  An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
args : extra arguments
These are passed to the model

kwargs : extra keyword arguments
These are passed to the model.

Returns model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

statsmodels.regression.linear_model.WLS.hessian
WLS.hessian(params)
The Hessian matrix of the model

statsmodels.regression.linear_model.WLS.information
WLS.information(params)
Fisher information matrix of model
Returns -Hessian of loglike evaluated at params.

statsmodels.regression.linear_model.WLS.initialize
WLS.initialize()

statsmodels.regression.linear_model.WLS.loglike
WLS.loglike(params)
Returns the value of the gaussian log-likelihood function at params.
Given the whitened design matrix, the log-likelihood is evaluated at the parameter vector params for the dependent variable Y.

Parameters params : array-like
The parameter estimates.

Returns llf : float
The value of the log-likelihood function for a WLS Model.

Notes

\[-\frac{n}{2} \log (Y - \hat{Y}) - \frac{n}{2} \left( 1 + \log \left( \frac{2\pi}{n} \right) \right) - \frac{1}{2} \log (\lvert W \rvert)\]

where W is a diagonal matrix

3.1. Linear Regression
statsmodels.regression.linear_model.WLS.predict

WLS.predict(params, exog=None)

Return linear predicted values from a design matrix.

Parameters
- **params**: array-like
  Parameters of a linear model

- **exog**: array-like, optional.
  Design / exogenous data. Model exog is used if None.

Returns
- **An array of fitted values**

Notes

If the model has not yet been fit, params is not optional.

statsmodels.regression.linear_model.WLS.score

WLS.score(params)

Score vector of model.

The gradient of logL with respect to each parameter.

statsmodels.regression.linear_model.WLS.whiten

WLS.whiten(X)

Whitener for WLS model, multiplies each column by sqrt(self.weights)

Parameters
- **X**: array-like
  Data to be whitened

Returns
- **sqrt(weights)*X**

Attributes

- **df_model**: The model degree of freedom, defined as the rank of the regressor
- **df_resid**: The residual degree of freedom, defined as the number of observations
- **endog_names**
- **exog_names**

statsmodels.regression.linear_model.GLSAR

class statsmodels.regression.linear_model.GLSAR(endog, exog=None, rho=1, missing='none')

A regression model with an AR(p) covariance structure.

Parameters
- **endog**: array-like
  1-d endogenous response variable. The dependent variable.

- **exog**: array-like
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See statsmodels.tools.add_constant.
rho : int

Order of the autoregressive covariance

missing : str

Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If
‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default
is ‘none.’ hasconst : None or bool Indicates whether the RHS includes a user-supplied
constant. If True, a constant is not checked for and k_constant is set to 1 and all result
statistics are calculated as if a constant is present. If False, a constant is not checked for
and k_constant is set to 0.

Notes

GLSAR is considered to be experimental. The linear autoregressive process of order p–AR(p)–is defined as:

Examples

```python
>>> import statsmodels.api as sm
>>> X = range(1,8)
>>> X = sm.add_constant(X)
>>> Y = [1,3,4,5,8,10,9]
>>> model = sm.GLSAR(Y, X, rho=2)
>>> for i in range(6):
...    results = model.fit()
...    print("AR coefficients: {0}".format(model.rho))
...    rho, sigma = sm.regression.yule_walker(results.resid,
...               order=model.order)
...    model = sm.GLSAR(Y, X, rho)
...'
AR coefficients: [ 0.  0.]
AR coefficients: [-0.52571491 -0.84496178]
AR coefficients: [-0.6104153  -0.86656458]
AR coefficients: [-0.60439494 -0.857867 ]
AR coefficients: [-0.60482188 -0.85846157]
AR coefficients: [-0.60479146 -0.85841922]
>>> results.params
array([-0.66661205, 1.60850853])
>>> results.tvalues
array([-2.10304127, 21.8047269 ])  
>>> print(results.t_test([1, 0]))
< T test: effect=array([-0.66661205]), sd=array([[ 0.31697526]])
 t=array([-2.10304127]), p=array([ 0.06309969])
>>> print(results.f_test(np.identity(2)))
<F test: F=array([[1815.23061844]]), p=[[ 0.00002372]], den=3, df_num=2>
```

Or, equivalently

```python
>>> model2 = sm.GLSAR(Y, X, rho=2)
>>> res = model2.iterative_fit(maxiter=6)
>>> model2.rho
array([-0.60479146, -0.85841922])
```

Methods
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit([method])</code></td>
<td>Full fit of the model.</td>
</tr>
<tr>
<td><code>from_formula(formula, data[, subset])</code></td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td><code>hessian(params)</code></td>
<td>The Hessian matrix of the model.</td>
</tr>
<tr>
<td><code>information(params)</code></td>
<td>Fisher information matrix of model.</td>
</tr>
<tr>
<td><code>initialize()</code></td>
<td></td>
</tr>
<tr>
<td><code>iterative_fit([maxiter])</code></td>
<td>Perform an iterative two-stage procedure to estimate a GLS model.</td>
</tr>
<tr>
<td><code>loglike(params)</code></td>
<td>Returns the value of the Gaussian log-likelihood function at params.</td>
</tr>
<tr>
<td><code>predict(params[, exog])</code></td>
<td>Return linear predicted values from a design matrix.</td>
</tr>
<tr>
<td><code>score(params)</code></td>
<td>Score vector of model.</td>
</tr>
<tr>
<td><code>whiten(X)</code></td>
<td>Whiten a series of columns according to an AR(p)</td>
</tr>
</tbody>
</table>

```python
statsmodels.regression.linear_model.GLSAR.fit

GLSAR.fit (method='pinv', **kwargs)

Full fit of the model.

The results include an estimate of covariance matrix, (whitened) residuals and an estimate of scale.

Parameters

- **method**: str
  - Can be “pinv”, “qr”. “pinv” uses the Moore-Penrose pseudoinverse to solve the least squares problem. “qr” uses the QR factorization.

Returns

- A RegressionResults class instance.

See also:

- regression.RegressionResults

Notes

The fit method uses the pseudoinverse of the design/exogenous variables to solve the least squares minimization.

```python
statsmodels.regression.linear_model.GLSAR.from_formula

classmethod GLSAR.from_formula (formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

Parameters

- **formula**: str or generic Formula object
  - The formula specifying the model
- **data**: array-like
  - The data for the model. See Notes.
- **subset**: array-like
  - An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
- **args**: extra arguments
  - These are passed to the model
- **kwargs**: extra keyword arguments
  - These are passed to the model.

Returns

- **model**: Model instance
```
Notes

data must define \_getitem\_ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**statsmodels.regression.linear_model.GLSAR.hessian**

GLSAR.hessian(params)

The Hessian matrix of the model

**statsmodels.regression.linear_model.GLSAR.information**

GLSAR.information(params)

Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

**statsmodels.regression.linear_model.GLSAR.initialize**

GLSAR.initialize()

**statsmodels.regression.linear_model.GLSAR.iterative_fit**

GLSAR.iterative_fit(maxiter=3)

Perform an iterative two-stage procedure to estimate a GLS model.

The model is assumed to have AR(p) errors, AR(p) parameters and regression coefficients are estimated iteratively.

Parameters maxiter : integer, optional

the number of iterations

**statsmodels.regression.linear_model.GLSAR.loglike**

GLSAR.loglike(params)

Returns the value of the Gaussian log-likelihood function at params.

Given the whitened design matrix, the log-likelihood is evaluated at the parameter vector params for the dependent variable endog.

Parameters params : array-like

The parameter estimates

Returns loglike : float

The value of the log-likelihood function for a GLS Model.

Notes

The log-likelihood function for the normal distribution is

\[-\frac{n}{2} \log \left( (Y - \hat{Y})' (Y - \hat{Y}) \right) - \frac{n}{2} \left( 1 + \log \left( \frac{2\pi}{n} \right) \right) - \frac{1}{2} \log |\Sigma| \]

Y and Y-hat are whitened.
statsmodels.regression.linear_model.GLSAR.predict
GLSAR.predict(params, exog=None)
Return linear predicted values from a design matrix.

Parameters
params : array-like
Parameters of a linear model
exog : array-like, optional.
Design / exogenous data. Model exog is used if None.

Returns
An array of fitted values:

Notes
If the model has not yet been fit, params is not optional.

statsmodels.regression.linear_model.GLSAR.score
GLSAR.score(params)
Score vector of model.
The gradient of logL with respect to each parameter.

statsmodels.regression.linear_model.GLSAR.whiten
GLSAR.whiten(X)
Whiten a series of columns according to an AR(p) covariance structure. This drops initial p observations.

Parameters
X : array-like
The data to be whitened,

Returns
whitened array:

Attributes

df_model The model degree of freedom, defined as the rank of the regressor
df_resid The residual degree of freedom, defined as the number of observations
endog_names exog_names

statsmodels.regression.linear_model.yule_walker
statsmodels.regression.linear_model.yule_walker(X, order=1, method='unbiased',
df=None, inv=False, demean=True)
Estimate AR(p) parameters from a sequence X using Yule-Walker equation.
Unbiased or maximum-likelihood estimator (mle)
See, for example:

Parameters
X : array-like
1d array
order : integer, optional
The order of the autoregressive process. Default is 1.

method : string, optional
Method can be “unbiased” or “mle” and this determines denominator in estimate of
autocorrelation function (ACF) at lag k. If “mle”, the denominator is n=X.shape[0], if
“unbiased” the denominator is n-k. The default is unbiased.

df : integer, optional
Specifies the degrees of freedom. If df is supplied, then it is assumed the X has df
degrees of freedom rather than n. Default is None.

inv : bool
If inv is True the inverse of R is also returned. Default is False.

demean : bool
True, the mean is subtracted from X before estimation.

Returns rho :
The autoregressive coefficients

sigma :

Examples

```python
>>> import statsmodels.api as sm
>>> from statsmodels.datasets.sunspots import load
>>> data = load()
>>> rho, sigma = sm.regression.yule_walker(data.endog,
                                           order=4, method="mle")

>>> rho
array([ 1.28310031, -0.45240924, -0.20770299, 0.04794365])
>>> sigma
16.808022730464351
```

---

 STATS MODELS Regression Quantile Regression

class statsmodels.regression.quantile_regression.QuantReg

Quantile Regression
Estimate a quantile regression model using iterative reweighted least squares.

Parameters endog : array or dataframe
endogenous/response variable

exog : array or dataframe
exogenous/explanatory variable(s)
Notes

The Least Absolute Deviation (LAD) estimator is a special case where quantile is set to 0.5 (q argument of the fit method).

The asymptotic covariance matrix is estimated following the procedure in Greene (2008, p.407-408), using either the logistic or gaussian kernels (kernel argument of the fit method).

References

General:

Kernels (used by the fit method):
• Green (2008) Table 14.2

Bandwidth selection (used by the fit method):

Keywords: Least Absolute Deviation (LAD) Regression, Quantile Regression, Regression, Robust Estimation.

Methods

```python
statsmodels.regression.quantile_regression.QuantReg.fit
QuantReg.fit(q=0.5,  vcov='robust', kernel='epa', bandwidth='hsheather',  max_iter=1000,  p_tol=1e-06, **kwargs)
```

Solve by Iterative Weighted Least Squares

Parameters
q : float
Quantile must be between 0 and 1
vcov : string, method used to calculate the variance-covariance matrix of the parameters. Default is robust:
  • robust : heteroskedasticity robust standard errors (as suggested in Greene 6th edition)
  • iid : iid errors (as in Stata 12)

kernel : string, kernel to use in the kernel density estimation for the asymptotic covariance matrix:
  • epa: Epanechnikov
  • cos: Cosine
  • gau: Gaussian
  • par: Parzene

bandwidth: string, Bandwidth selection method in kernel density estimation for asymptotic covariance estimate (full references in QuantReg docstring):
  • hsheather: Hall-Sheather (1988)
  • bofinger: Bofinger (1975)
  • chamberlain: Chamberlain (1994)

...
statsmodels.regression.quantile_regression.QuantReg.hessian

QuantReg.hessian(params)
The Hessian matrix of the model

statsmodels.regression.quantile_regression.QuantReg.information

QuantReg.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

statsmodels.regression.quantile_regression.QuantReg.initialize

QuantReg.initialize()

statsmodels.regression.quantile_regression.QuantReg.loglike

QuantReg.loglike(params)
Log-likelihood of model.

statsmodels.regression.quantile_regression.QuantReg.predict

QuantReg.predict(params, exog=None)
Return linear predicted values from a design matrix.

Parameters params : array-like
Parameters of a linear model

exog : array-like, optional.
Design / exogenous data. Model exog is used if None.

Returns An array of fitted values:

Notes

If the model has not yet been fit, params is not optional.

statsmodels.regression.quantile_regression.QuantReg.score

QuantReg.score(params)
Score vector of model.

The gradient of logL with respect to each parameter.

statsmodels.regression.quantile_regression.QuantReg.whiten

QuantReg.whiten(data)
QuantReg model whitener does nothing: returns data.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>df_model</td>
<td>The model degree of freedom, defined as the rank of the regressor</td>
</tr>
<tr>
<td>df_resid</td>
<td>The residual degree of freedom, defined as the number of observations</td>
</tr>
<tr>
<td>endog_names</td>
<td></td>
</tr>
<tr>
<td>exog_names</td>
<td></td>
</tr>
</tbody>
</table>
Results Classes

Fitting a linear regression model returns a results class. OLS has a specific results class with some additional methods compared to the results class of the other linear models.

\[
\text{RegressionResults}(\text{model, params[, ...]}) \quad \text{This class summarizes the fit of a linear regression model.}
\]

\[
\text{OLSResults}(\text{model, params[, ...]}) \quad \text{Results class for for an OLS model.}
\]

`statsmodels.regression.linear_model.RegressionResults`

class `statsmodels.regression.linear_model.RegressionResults` (model, params, normalized_cov_params=None, scale=1.0)

This class summarizes the fit of a linear regression model.

It handles the output of contrasts, estimates of covariance, etc.

Returns **Attributes**:

- **aic**:
  - Aikake’s information criteria. For a model with a constant \(-2llf + 2(df_{model} + 1)\).
  - For a model without a constant \(-2llf + 2(df_{model}).\)

- **bic**:
  - Bayes’ information criteria For a model with a constant \(-2llf + \log(n)(df_{model} + 1).\)
  - For a model without a constant \(-2llf + \log(n)(df_{model}).\)

- **bse**:
  - The standard errors of the parameter estimates.

- **pinv_wexog**:
  - See specific model class docstring

- **centered_tss**:
  - The total (weighted) sum of squares centered about the mean.

- **cov_HC0**:
  - Heteroscedasticity robust covariance matrix. See HC0_se below.

- **cov_HC1**:
  - Heteroscedasticity robust covariance matrix. See HC1_se below.

- **cov_HC2**:
  - Heteroscedasticity robust covariance matrix. See HC2_se below.

- **cov_HC3**:
  - Heteroscedasticity robust covariance matrix. See HC3_se below.

- **cov_type**:
  - Parameter covariance estimator used for standard errors and t-stats

- **df_model**:
  - Model degrees of freedom. The number of regressors \(p\). Does not include the constant if one is present.
df_resid:
Residual degrees of freedom. \( n - p - 1 \) if a constant is present. \( n - p \) if a constant is not included.

ess:
Explained sum of squares. If a constant is present, the centered total sum of squares minus the sum of squared residuals. If there is no constant, the uncentered total sum of squares is used.

fvalue:
F-statistic of the fully specified model. Calculated as the mean squared error of the model divided by the mean squared error of the residuals.

f_pvalue:
p-value of the F-statistic

fittedvalues:
The predicted values for the original (unwhitened) design.

het_scale:
adjusted squared residuals for heteroscedasticity robust standard errors. Is only available after HC#_se or cov_HC# is called. See HC#_se for more information.

HC0_se:
White’s (1980) heteroskedasticity robust standard errors. Defined as \( \sqrt{\text{diag}(X.T X)^{-1} X.T \text{diag}(e_i^2) X(X.T X)^{-1}} \) where \( e_i = \text{resid}[i] \) HC0_se is a cached property. When HC0_se or cov_HC0 is called the RegressionResults instance will then have another attribute \( \text{het_scale} \), which is in this case is just resid**2.

HC1_se:
MacKinnon and White’s (1985) alternative heteroskedasticity robust standard errors. Defined as \( \sqrt{\text{diag}(n/(n-p)*HC_0)} \) HC1_se is a cached property. When HC1_se or cov_HC1 is called the RegressionResults instance will then have another attribute \( \text{het_scale} \), which is in this case is \( n/(n-p)*\text{resid}**2 \).

HC2_se:
MacKinnon and White’s (1985) alternative heteroskedasticity robust standard errors. Defined as \( (X.T X)^{-1} X.T \text{diag}(e_i^2/(1-h_{ii})) X(X.T X)^{-1} \) where \( h_{ii} = x_i(X.T X)^{-1}x_i.T \) HC2_se is a cached property. When HC2_se or cov_HC2 is called the RegressionResults instance will then have another attribute \( \text{het_scale} \), which is in this case is \( \text{resid}^2/(1-h_{ii}) \). HCCM matrices are only appropriate for OLS.

HC3_se:
MacKinnon and White’s (1985) alternative heteroskedasticity robust standard errors. Defined as \( (X.T X)^{-1} X.T \text{diag}(e_i^2/(1-h_{ii})^2) X(X.T X)^{-1} \) where \( h_{ii} = x_i(X.T X)^{-1}x_i.T \) HC3_se is a cached property. When HC3_se or cov_HC3 is called the RegressionResults instance will then have another attribute \( \text{het_scale} \), which is in this case is \( \text{resid}^2/(1-h_{ii})^2 \).

model:
A pointer to the model instance that called fit() or results.

mse_model:
Mean squared error the model. This is the explained sum of squares divided by the
model degrees of freedom.

**mse_resid** :

Mean squared error of the residuals. The sum of squared residuals divided by the resid-
ual degrees of freedom.

**mse_total** :

Total mean squared error. Defined as the uncentered total sum of squares divided by n
the number of observations.

**nobs** :

Number of observations n.

**normalized_cov_params** :

See specific model class docstring

**params** :

The linear coefficients that minimize the least squares criterion. This is usually called
Beta for the classical linear model.

**pvalues** :

The two-tailed p values for the t-stats of the params.

**resid** :

The residuals of the model.

**resid_pearson** :

\( \text{wresid} \) normalized to have unit variance.

**rsquared** :

R-squared of a model with an intercept. This is defined here as \( 1 - \frac{\text{ssr}}{\text{centered_tss}} \) if the constant is included in the model and \( 1 - \frac{\text{ssr}}{\text{uncentered_tss}} \) if the constant is omitted.

**rsquared_adj** :

Adjusted R-squared. This is defined here as \( 1 - \frac{(\text{nobs}-1)\text{df_resid} \times (1-\text{rsquared})}{\text{if a constant is included}} \) and \( 1 - \frac{\text{nobs}}{\text{df_resid}} \times (1-\text{rsquared}) \) if no constant is included.

**scale** :

A scale factor for the covariance matrix. Default value is \( \text{ssr}/(n-p) \). Note that the square
root of \( \text{scale} \) is often called the standard error of the regression.

**ssr** :

Sum of squared (whitened) residuals.

**uncentered_tss** :

Uncentered sum of squares. Sum of the squared values of the (whitened) endogenous response variable.

**wresid** :

The residuals of the transformed/whitened regressand and regressor(s)
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC0_se()</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>HC1_se()</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>HC2_se()</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>HC3_se()</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>aic()</td>
<td>Use F test to test whether restricted model is correct</td>
</tr>
<tr>
<td>bic()</td>
<td>Use Lagrange Multiplier test to test whether restricted model is correct</td>
</tr>
<tr>
<td>bse()</td>
<td>Likelihood ratio test to test whether restricted model is correct</td>
</tr>
<tr>
<td>centered_tss()</td>
<td>Return condition number of exogenous matrix, calculated as ratio of largest to smallest eigenvalue.</td>
</tr>
<tr>
<td>compare_f_test(restricted)</td>
<td>Returns the confidence interval of the fitted parameters.</td>
</tr>
<tr>
<td>compare_lm_test(restricted[, demean, use_lr])</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>compare_lr_test(restricted[, large_sample])</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>condition_number()</td>
<td>Returns the variance/covariance matrix.</td>
</tr>
<tr>
<td>conf_int([alpha, cols])</td>
<td>Return eigenvalues sorted in decreasing order.</td>
</tr>
<tr>
<td>cov_HC0()</td>
<td>Compute the F-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td>cov_HC1()</td>
<td>create new results instance with robust covariance as default</td>
</tr>
<tr>
<td>cov_HC2()</td>
<td>load a pickle, (class method)</td>
</tr>
<tr>
<td>cov_HC3()</td>
<td>Call self.model.predict with self.params as the first argument.</td>
</tr>
<tr>
<td>cov_params([r_matrix, column, scale, cov_p, ...])</td>
<td>remove data arrays, all nobs arrays from result and model</td>
</tr>
<tr>
<td>eigenvals()</td>
<td>Residuals, normalized to have unit variance.</td>
</tr>
<tr>
<td>ess()</td>
<td>save a pickle of this instance</td>
</tr>
<tr>
<td>f_pvalue()</td>
<td>Summarize the Regression Results</td>
</tr>
<tr>
<td>f_test(r_matrix[, q_matrix, cov_p, scale, ...])</td>
<td>Experimental summary function to summarize the regression results</td>
</tr>
<tr>
<td>fittedvalues()</td>
<td>Compute a t-test for a joint linear hypothesis of the form Rb = q</td>
</tr>
<tr>
<td>fvalue()</td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td>get_robustcov_results([cov_type, use_t])</td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td>initialize(model, params, **kwd)</td>
<td></td>
</tr>
<tr>
<td>llf()</td>
<td></td>
</tr>
<tr>
<td>load(fname)</td>
<td></td>
</tr>
<tr>
<td>mse_model()</td>
<td></td>
</tr>
<tr>
<td>mse_resid()</td>
<td></td>
</tr>
<tr>
<td>mse_total()</td>
<td></td>
</tr>
<tr>
<td>nobs()</td>
<td></td>
</tr>
<tr>
<td>normalized_cov_params()</td>
<td></td>
</tr>
<tr>
<td>predict([exog, transform])</td>
<td></td>
</tr>
<tr>
<td>pvalues()</td>
<td></td>
</tr>
<tr>
<td>remove_data()</td>
<td></td>
</tr>
<tr>
<td>resid()</td>
<td></td>
</tr>
<tr>
<td>resid_pearson()</td>
<td></td>
</tr>
<tr>
<td>rsquared()</td>
<td></td>
</tr>
<tr>
<td>rsquared_adj()</td>
<td></td>
</tr>
<tr>
<td>save(fname[, remove_data])</td>
<td></td>
</tr>
<tr>
<td>scale()</td>
<td></td>
</tr>
<tr>
<td>ssr()</td>
<td></td>
</tr>
<tr>
<td>summary([yname, xname, title, alpha])</td>
<td></td>
</tr>
<tr>
<td>summary2([yname, xname, title, alpha, ...])</td>
<td></td>
</tr>
<tr>
<td>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</td>
<td></td>
</tr>
<tr>
<td>tvalues()</td>
<td></td>
</tr>
<tr>
<td>uncentered_tss()</td>
<td></td>
</tr>
<tr>
<td>wald_test(r_matrix[, q_matrix, cov_p, ...])</td>
<td></td>
</tr>
<tr>
<td>wresid()</td>
<td></td>
</tr>
</tbody>
</table>
statsmodels.regression.linear_model.RegressionResults.HC0_se

static RegressionResults.HC0_se()

See statsmodels.RegressionResults

statsmodels.regression.linear_model.RegressionResults.HC1_se

static RegressionResults.HC1_se()

See statsmodels.RegressionResults

statsmodels.regression.linear_model.RegressionResults.HC2_se

static RegressionResults.HC2_se()

See statsmodels.RegressionResults

statsmodels.regression.linear_model.RegressionResults.HC3_se

static RegressionResults.HC3_se()

See statsmodels.RegressionResults

statsmodels.regression.linear_model.RegressionResults.aic

static RegressionResults.aic()

statsmodels.regression.linear_model.RegressionResults.bic

static RegressionResults.bic()

statsmodels.regression.linear_model.RegressionResults.bse

static RegressionResults.bse()

statsmodels.regression.linear_model.RegressionResults.centered_tss

static RegressionResults.centered_tss()

statsmodels.regression.linear_model.RegressionResults.compare_f_test

RegressionResults.compare_f_test(restricted)

use F test to test whether restricted model is correct

Parameters

restricted : Result instance

The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, ssr, residual degrees of freedom, df_resid.

Returns

f_value : float

test statistic, F distributed

p_value : float

p-value of the test statistic

df_diff : int

degrees of freedom of the restriction, i.e. difference in df between models

3.1. Linear Regression 151
Notes

See mailing list discussion October 17.

This test compares the residual sum of squares of the two models. This is not a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results under the assumption of homoscedasticity and no autocorrelation (sphericity).

```python
statsmodels.regression.linear_model.RegressionResults.compare_lm_test
RegressionResults.compare_lm_test(restricted, demean=True, use_lr=False)
```

Use Lagrange Multiplier test to test whether restricted model is correct

**Parameters**

- **restricted**: Result instance
  - The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, \(ssr\), residual degrees of freedom, \(df\text{\_}resid\).
- **demean**: bool
  - Flag indicating whether the demean the scores based on the residuals from the restricted model. If True, the covariance of the scores are used and the LM test is identical to the large sample version of the LR test.

**Returns**

- **lm_value**: float
  - test statistic, \(\chi^2\) distributed
- **p_value**: float
  - p-value of the test statistic
- **df_diff**: int
  - degrees of freedom of the restriction, i.e. difference in df between models

Notes

TODO: explain LM text

```python
statsmodels.regression.linear_model.RegressionResults.compare_lr_test
RegressionResults.compare_lr_test(restricted, large_sample=False)
```

Likelihood ratio test to test whether restricted model is correct

**Parameters**

- **restricted**: Result instance
  - The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, \(ssr\), residual degrees of freedom, \(df\text{\_}resid\).
- **large_sample**: bool
  - Flag indicating whether to use a heteroskedasticity robust version of the LR test, which is a modified LM test.

**Returns**

- **lr_stat**: float
  - likelihood ratio, \(\chi^2\) distributed with \(df\text{\_}diff\) degrees of freedom
- **p_value**: float
p-value of the test statistic

def_df_diff : int
    degrees of freedom of the restriction, i.e. difference in df between models

Notes

The exact likelihood ratio is valid for homoskedastic data, and is defined as

$$D = -2 \log \left( \frac{L_{null}}{L_{alternative}} \right)$$

where $L$ is the likelihood of the model. With $D$ distributed as chisquare with df equal to difference in number of parameters or equivalently difference in residual degrees of freedom.

The large sample version of the likelihood ratio is defined as

$$D = ns^tS^{-1}s$$

where $s = n^{-1} \sum_{i=1}^{n} s_i$

$$s_i = x_{i,alternative} \epsilon_{i, null}$$

is the average score of the model evaluated using the residuals from null model and the regressors from the alternative model and $S$ is the covariance of the scores, $s_i$. The covariance of the scores is estimated using the same estimator as in the alternative model.

This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results without taking unspecified heteroscedasticity or correlation into account.

This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results without taking unspecified heteroscedasticity or correlation into account.

is the average score of the model evaluated using the residuals from null model and the regressors from the alternative model and $S$ is the covariance of the scores, $s_i$. The covariance of the scores is estimated using the same estimator as in the alternative model.

TODO: put into separate function, needs tests
cols : array-like, optional

*cols* specifies which confidence intervals to return

**Notes**

The confidence interval is based on Student’s t-distribution.

```python
statsmodels.regression.linear_model.RegressionResults.cov_HC0
static RegressionResults.cov_HC0()
See statsmodels.RegressionResults
```

```python
statsmodels.regression.linear_model.RegressionResults.cov_HC1
static RegressionResults.cov_HC1()
See statsmodels.RegressionResults
```

```python
statsmodels.regression.linear_model.RegressionResults.cov_HC2
static RegressionResults.cov_HC2()
See statsmodels.RegressionResults
```

```python
statsmodels.regression.linear_model.RegressionResults.cov_HC3
static RegressionResults.cov_HC3()
See statsmodels.RegressionResults
```

```python
statsmodels.regression.linear_model.RegressionResults.cov_params
RegressionResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)
Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multi-
plied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

**Parameters**

- **r_matrix** : array-like
  
  Can be 1d, or 2d. Can be used alone or with other.

- **column** : array-like, optional
  
  Must be used on its own. Can be 0d or 1d see below.

- **scale** : float, optional
  
  Can be specified or not. Default is None, which means that the scale argument is taken
  from the model.

- **other** : array-like, optional
  
  Can be used when r_matrix is specified.

**Returns** (The below are assumed to be in matrix notation.) :

- **cov** : ndarray
  
  If no argument is specified returns the covariance matrix of a model :

  (scale)*(X.T X)^(-1) :
If contrast is specified it pre and post-multiplies as follows:

\[(\text{scale}) \cdot r\_\text{matrix} \cdot (X^\text{T} \cdot X)^{(-1)} \cdot r\_\text{matrix}\_T:\]

If contrast and other are specified returns:

\[(\text{scale}) \cdot r\_\text{matrix} \cdot (X^\text{T} \cdot X)^{(-1)} \cdot \text{other}\_T:\]

If column is specified returns:

\[(\text{scale}) \cdot (X^\text{T} \cdot X)^{(-1)}[\text{column, column}] \text{ if column is 0d}:\]

\[\text{OR} :\]

\[(\text{scale}) \cdot (X^\text{T} \cdot X)^{(-1)}[\text{column}][:, \text{column}] \text{ if column is 1d}:\]

```python
statsmodels.regression.linear_model.RegressionResults.eigenvals
static RegressionResults.eigenvals()  
Return eigenvalues sorted in decreasing order.
```

```python
statsmodels.regression.linear_model.RegressionResults.ess
static RegressionResults.ess()  
```

```python
statsmodels.regression.linear_model.RegressionResults.f_pvalue
static RegressionResults.f_pvalue()  
```

```python
statsmodels.regression.linear_model.RegressionResults.f_test
RegressionResults.f_test(r\_\text{matrix}, q\_\text{matrix}=None, cov\_p=None, scale=1.0, invcov=None)  
Compute the F-test for a joint linear hypothesis.
```

**Parameters**

- **r\_\text{matrix}**: array-like, str, or tuple
  - array: An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q\_\text{matrix} is deprecated.

- **q\_\text{matrix}**: array-like
  - This is deprecated. See r\_\text{matrix} and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r\_\text{matrix} is an array, q\_\text{matrix} is assumed to be a conformable array of zeros.

- **cov\_p**: array-like, optional
  - An alternative estimate for the parameter covariance matrix. If None is given, self.normalized\_cov\_params is used.

- **scale**: float, optional
  - Default is 1.0 for no scaling.

- **invcov**: array-like, optional
  - A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**See also:**

- statsmodels.contrasts.statsmodels.model.LikelihoodModelResults.wald_test,
- statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint
Notes

The matrix \( r_{matrix} \) is assumed to be non-singular. More precisely,

\[ r_{matrix} (pX \ pX^T) r_{matrix}^T \]

is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm

>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.

>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>

Compare this to

```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10

```python
>>> B = np.array(([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols

>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'

```python
>>> results = ols(formula, dta).fit()
```

\[ (\text{GNPDEFL} = \text{GNP}), \ (\text{UNEMP} = 2), \ (\text{YEAR}/1829 = 1) \]

```python
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```
RegressionResults.get_robustcov_results (cov_type='HC1', use_t=None, **kwds)
create new results instance with robust covariance as default

Parameters cov_type : string
the type of robust sandwich estimator to use. see Notes below

use_t : bool
If true, then the t distribution is used for inference. If false, then the normal distribution is used.

kwds : depends on cov_type
Required or optional arguments for robust covariance calculation. see Notes below

Returns results : results instance
This method creates a new results instance with the requested robust covariance as the default covariance of the parameters. Inferential statistics like p-values and hypothesis tests will be based on this covariance matrix.

Notes

The following covariance types and required or optional arguments are currently available:

•‘HC0’, ‘HC1’, ‘HC2’, ‘HC3’ and no keyword arguments: heteroscedasticity robust covariance

•‘HAC’ and keywords
  – maxlag integer (required) : number of lags to use
  – kernel string (optional) : kernel, default is Bartlett
  – use_correction bool (optional) [If true, use small sample] correction

•‘cluster’ and required keyword groups, integer group indicator
  – groups array_like, integer (required) : index of clusters or groups
  – use_correction bool (optional) : If True the sandwich covariance is calculated with a small sample correction. If False the the sandwich covariance is calculated without small sample correction.

  – df_correction bool (optional) If True (default), then the degrees of freedom for the inferential statistics and hypothesis tests, such as pvalues, f_pvalue, conf_int, and t_test and f_test, are based on the number of groups minus one instead of the total number of observations minus the number of explanatory variables. df_resid of the results instance is adjusted. If False, then df_resid of the results instance is not adjusted.

•‘hac-groupsum’ Driscoll and Kraay, heteroscedasticity and autocorrelation robust standard errors in panel data keywords
  – time array_like (required) : index of time periods
  – maxlag integer (required) : number of lags to use
  – kernel string (optional) : kernel, default is Bartlett
  – use_correction False or string in [‘hac’, ‘cluster’] (optional) : If False the the sandwich covariance is calculated without small sample correction. If use_correction = ‘cluster’ (default), then the same small sample correction as in the case of ‘covtype=’cluster” is used.
- **df_correction** bool (optional)  adjustment to df_resid, see cov_type ‘cluster’ above #TODO: we need more options here

**‘hac-panel’ heteroscedasticity and autocorrelation robust standard** errors in panel data. The data needs to be sorted in this case, the time series for each panel unit or cluster need to be stacked. keywords

- **time** array_like (required) : index of time periods
- **maxlag** integer (required) : number of lags to use
- **kernel** string (optional) : kernel, default is Bartlett
- **use_correction** False or string in ['hac', ‘cluster’) (optional) : If False the the sandwich co-variance is calculated without small sample correction.

- **df_correction** bool (optional)  adjustment to df_resid, see cov_type ‘cluster’ above #TODO: we need more options here

Reminder: **use_correction** in “nw-groupsum” and “nw-panel” is not bool, needs to be in [False, ‘hac’, ‘cluster’]

TODO: Currently there is no check for extra or misspelled keywords, except in the case of cov_type $HC_x$

```python
statsmodels.regression.linear_model.RegressionResults.initialize
RegressionResults.initialize(model, params, **kwd)
```

```python
statsmodels.regression.linear_model.RegressionResults.llf
static RegressionResults.llf()
```

```python
statsmodels.regression.linear_model.RegressionResults.load
classmethod RegressionResults.load(fname)
load a pickle, (class method)

Parameters
fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

Returns
unpickled instance:
```
```python
statsmodels.regression.linear_model.RegressionResults.mse_model
static RegressionResults.mse_model()
```

```python
statsmodels.regression.linear_model.RegressionResults.mse_resid
static RegressionResults.mse_resid()
```

```python
statsmodels.regression.linear_model.RegressionResults.mse_total
static RegressionResults.mse_total()
```

```python
statsmodels.regression.linear_model.RegressionResults.nobs
static RegressionResults.nobs()
```

```python
statsmodels.regression.linear_model.RegressionResults.normalized_cov_params
RegressionResults.normalized_cov_params()
```
RegressionResults.predict(exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

Parameters
exog : array-like, optional
The values for which you want to predict.
transform : bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model $y \sim \log(x1) + \log(x2)$, and transform is True, then you can pass a data structure that contains $x1$ and $x2$ in their original form. Otherwise, you’d need to log the data first.

Returns
See self.model.predict:

RegressionResults.pvalues

RegressionResults.remove_data

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

RegressionResults.resid

RegressionResults.resid_pearson

Residuals, normalized to have unit variance.

Returns
An array $wresid/sqrt(scale)$:

RegressionResults.rsquared

RegressionResults.rsquared_adj

3.1. Linear Regression
RegressionResults.save(fname, remove_data=False)

save a pickle of this instance

Parameters

- **fname** : string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- **remove_data** : bool
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

RegressionResults.scale()

RegressionResults.ssr()

RegressionResults.summary(yname=None, xname=None, title=None, alpha=0.05)

Summarize the Regression Results

Parameters

- **yname** : string, optional
  
  Default is y

- **xname** : list of strings, optional
  
  Default is var_## for ## in p the number of regressors

- **title** : string, optional
  
  Title for the top table. If not None, then this replaces the default title

- **alpha** : float
  
  significance level for the confidence intervals

Returns

- **smry** : Summary instance
  
  this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results
Experimental summary function to summarize the regression results

**Parameters**

- **xname**: List of strings of length equal to the number of parameters
  
  Names of the independent variables (optional)

- **yname**: string
  
  Name of the dependent variable (optional)

- **title**: string, optional
  
  Title for the top table. If not None, then this replaces the default title

- **alpha**: float
  
  significance level for the confidence intervals

- **float_format**: string
  
  print format for floats in parameters summary

**Returns**

- **smry**: Summary instance
  
  this holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

-  `statsmodels.iolib.summary.Summary` class to hold summary results

`statsmodels.regression.linear_model.RegressionResults.t_test`

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

**Parameters**

- **r_matrix**: array-like, str, tuple
  
  - array : If an array is given, a $p \times k$ 2d array or length $k$ 1d array specifying the linear restrictions.
  
  - str : The full hypotheses to test can be given as a string. See the examples.
  
  - tuple : A tuple of arrays in the form $(R, q)$, since $q$ matrix is deprecated.

- **q_matrix**: array-like or scalar, optional
  
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length $p$ row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p**: array-like, optional
  
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale**: float, optional
  
  An optional scale to use. Default is the scale specified by the model fit.

- **use_t**: bool, optional
If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics

f_test for F tests

patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm

>>> data = sm.datasets.longley.load()

>>> data.exog = sm.add_constant(data.exog)

>>> results = sm.OLS(data.endog, data.exog).fit()

>>> r = np.zeros_like(results.params)

>>> r[5:] = [1,-1]

>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]

r tests that the coefficients on the 5th and 6th independent variable are the same.

>>> T_Test = results.t_test(r)

>>> print(T_test)
<T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>

Alternatively, you can specify the hypothesis tests using a string

```python
>>> data = sm.datasets.longley.load_pandas().data

>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'

>>> results = ols(formula, dta).fit()

>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'

>>> t_test = results.t_test(hypotheses)

>>> print(t_test)
```
• str: The full hypotheses to test can be given as a string. See the examples.
• tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix: array-like

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p: array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale: float, optional

Default is 1.0 for no scaling.

invcov: array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f: bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,
r_matrix (pX pX.T) r_matrix.T
is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
•conf_int_el

See also:

RegressionResults

Methods

HC0_se() See statsmodels.RegressionResults
HC1_se() See statsmodels.RegressionResults
HC2_se() See statsmodels.RegressionResults
HC3_se() See statsmodels.RegressionResults
aic() use F test to test whether restricted model is correct
bic() Use Lagrange Multiplier test to test whether restricted model is correct
bse() Likelihood ratio test to test whether restricted model is correct
centered_tss() Returns the confidence interval of the fitted parameters.
conf_int([alpha, cols]) Computes the confidence interval for the parameter given by param_num
conf_int_el(param_num[, sig, upper_bound, ...]) See statsmodels.RegressionResults
cov_HC0() See statsmodels.RegressionResults
cov_HC1() See statsmodels.RegressionResults
cov_HC2() See statsmodels.RegressionResults
cov_HC3() See statsmodels.RegressionResults
cov_params([r_matrix, column, scale, cov_p, ...]) Returns the variance/covariance matrix.
cov_params_centered_tss() Returns eigenvalues sorted in decreasing order.
eigenvals() Tests single or joint hypotheses of the regression parameters using
el_test(b0_vals, param_nums[, ...]) Compute the F-test for a joint linear hypothesis.
f_pvalue() get an instance of Influence with influence and outlier measures
f_test(r_matrix[, q_matrix, cov_p, scale, ...]) create new results instance with robust covariance as default
fittedvalues() load a pickle, (class method)
fvalue() get_robustcov_results([cov_type, use_t]) load a pickle, (class method)
get_influence() Test observations for outliers according to method
get_robustcov_results() Call self.model.predict with self.params as the first argument.
initialize(model, params, **kwds) remove data arrays, all nobs arrays from result and model
llf() outlier_test([method, alpha]) Residuals, normalized to have unit variance.
load(fname) predict([exog, transform]) save a pickle of this instance
mse_model() remove_data() resid()
nobs() resid_pseudo() resid_pearson()
mse_resid() rsquared() rsquared_adj()
nmse_total() save(fname[, remove_data])
Table 3.15 – continued from previous page

- **scale()**
- **ssr()**
- **summary([yname, xname, title, alpha])**
  - Summarize the Regression Results
- **summary2([yname, xname, title, alpha, ...])**
  - Experimental summary function to summarize the regression results
- **t_test(r_matrix[, q_matrix, cov_p, scale, use_t])**
  - Compute a t-test for a joint linear hypothesis of the form $Rb = q$
  - Return the t-statistic for a given parameter estimate.
- **tvalues()**
- **uncentered_tss()**
- **wald_test(r_matrix[, q_matrix, cov_p, ...])**
  - Compute a Wald-test for a joint linear hypothesis.

### Class Methods

- **statsmodels.regression.linear_model.OLSResults.HC0_se**
  - `static OLSResults.HC0_se()`
  - See `statsmodels.RegressionResults`

- **statsmodels.regression.linear_model.OLSResults.HC1_se**
  - `static OLSResults.HC1_se()`
  - See `statsmodels.RegressionResults`

- **statsmodels.regression.linear_model.OLSResults.HC2_se**
  - `static OLSResults.HC2_se()`
  - See `statsmodels.RegressionResults`

- **statsmodels.regression.linear_model.OLSResults.HC3_se**
  - `static OLSResults.HC3_se()`
  - See `statsmodels.RegressionResults`

- **statsmodels.regression.linear_model.OLSResults.aic**
  - `static OLSResults.aic()`

- **statsmodels.regression.linear_model.OLSResults.bic**
  - `static OLSResults.bic()`

- **statsmodels.regression.linear_model.OLSResults.bse**
  - `static OLSResults.bse()`

- **statsmodels.regression.linear_model.OLSResults.centered_tss**
  - `static OLSResults.centered_tss()`

- **statsmodels.regression.linear_model.OLSResults.compare_f_test**
  - `OLSResults.compare_f_test(restricted)`
  - Use F test to test whether restricted model is correct
  
  **Parameters**
  - `restricted` : Result instance
    
    The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, `ssr`, residual degrees of freedom, `df_resid`.

  **Returns**
  - `f_value` : float
test statistic, F distributed

**p_value** : float

p-value of the test statistic

**df_diff** : int

degrees of freedom of the restriction, i.e. difference in df between models

**Notes**

See mailing list discussion October 17,

This test compares the residual sum of squares of the two models. This is not a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results under the assumption of homoscedasticity and no autocorrelation (sphericity).

`statsmodels.regression.linear_model.OLSResults.compare_lm_test`

`OLSResults.compare_lm_test (restricted, demean=True, use_lr=False)`

Use Lagrange Multiplier test to test whether restricted model is correct

**Parameters** restricted : Result instance

The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, `ssr`, residual degrees of freedom, `df_resid`.

demean : bool

Flag indicating whether the demean the scores based on the residuals from the restricted model. If True, the covariance of the scores are used and the LM test is identical to the large sample version of the LR test.

**Returns** lm_value : float

test statistic, chi2 distributed

**p_value** : float

p-value of the test statistic

**df_diff** : int

degrees of freedom of the restriction, i.e. difference in df between models

**Notes**

TODO: explain LM text

`statsmodels.regression.linear_model.OLSResults.compare_lr_test`

`OLSResults.compare_lr_test (restricted, large_sample=False)`

Likelihood ratio test to test whether restricted model is correct

**Parameters** restricted : Result instance

The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, `ssr`, residual degrees of freedom, `df_resid`. 

166 Chapter 3. Table of Contents
large_sample : bool
Flag indicating whether to use a heteroskedasticity robust version of the LR test, which
is a modified LM test.

Returns
lr_stat : float
likelihood ratio, chisquare distributed with df_diff degrees of freedom

p_value : float
p-value of the test statistic

df_diff : int
degrees of freedom of the restriction, i.e. difference in df between models

Notes
The exact likelihood ratio is valid for homoskedastic data, and is defined as

\[ D = -2 \log \left( \frac{\mathcal{L}_{null}}{\mathcal{L}_{alternative}} \right) \]

where \( \mathcal{L} \) is the likelihood of the model. With \( D \) distributed as chisquare with df equal to difference in
number of parameters or equivalently difference in residual degrees of freedom.

The large sample version of the likelihood ratio is defined as

\[ D = n s^T S^{-1} s \]

where \( s = n^{-1} \sum_{i=1}^{n} s_i \)

\[ s_i = x_{i,alternative} \epsilon_{i,null} \]

is the average score of the model evaluated using the residuals from null model and the regressors from
the alternative model and \( S \) is the covariance of the scores, \( s_i \). The covariance of the scores is estimated
using the same estimator as in the alternative model.

This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified
differences in residual degrees of freedom. This method will issue a warning if this is detected but still return the
results without taking unspecified heteroscedasticity or correlation into account.

This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified
differences in residual degrees of freedom. This method will issue a warning if this is detected but still return the
results without taking unspecified heteroscedasticity or correlation into account.

is the average score of the model evaluated using the residuals from null model and the regressors from
the alternative model and \( S \) is the covariance of the scores, \( s_i \). The covariance of the scores is estimated
using the same estimator as in the alternative model.

TODO: put into separate function, needs tests

```
statsmodels.regression.linear_model.OLSResults.condition_number
static OLSResults.condition_number()
Return condition number of exogenous matrix, calculated as ratio of largest to smallest eigenvalue.
```

3.1. Linear Regression 167
statsmodels.regression.linear_model.OLSResults.conf_int

OLSResults.conf_int(alpha=0.05, cols=None)

Returns the confidence interval of the fitted parameters.

Parameters

alpha : float, optional

The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

cols : array-like, optional

cols specifies which confidence intervals to return

Notes

The confidence interval is based on Student’s t-distribution.

statsmodels.regression.linear_model.OLSResults.conf_int_el

OLSResults.conf_int_el(param_num, sig=0.05, upper_bound=None, lower_bound=None, method='nm', stochastic_exog=1)

Computes the confidence interval for the parameter given by param_num using Empirical Likelihood

Parameters

param_num : float

The parameter for which the confidence interval is desired

sig : float

The significance level. Default is .05

upper_bound : float

The maximum value the upper limit can be. Default is the 99.9% confidence value under OLS assumptions.

lower_bound : float

The minimum value the lower limit can be. Default is the 99.9% confidence value under OLS assumptions.

method : string

Can either be ‘nm’ for Nelder-Mead or ‘powell’ for Powell. The optimization method that optimizes over nuisance parameters. Default is ‘nm’

Returns

ci : tuple

The confidence interval

See also:

el_test

Notes

This function uses brentq to find the value of beta where test_beta([beta], param_num)[1] is equal to the critical value.

The function returns the results of each iteration of brentq at each value of beta.

The current function value of the last printed optimization should be the critical value at the desired significance level. For alpha=.05, the value is 3.841459.
To ensure optimization terminated successfully, it is suggested to do `el_test([lower_limit], [param_num])`

If the optimization does not terminate successfully, consider switching optimization algorithms.

If optimization is still not successful, try changing the values of `start_int_params`. If the current function value repeatedly jumps from a number between 0 and the critical value and a very large number (>50), the starting parameters of the interior minimization need to be changed.

```python
statsmodels.regression.linear_model.OLSResults.cov_HC0
static OLSResults.cov_HC0()
    See statsmodels.RegressionResults

statsmodels.regression.linear_model.OLSResults.cov_HC1
static OLSResults.cov_HC1()
    See statsmodels.RegressionResults

statsmodels.regression.linear_model.OLSResults.cov_HC2
static OLSResults.cov_HC2()
    See statsmodels.RegressionResults

statsmodels.regression.linear_model.OLSResults.cov_HC3
static OLSResults.cov_HC3()
    See statsmodels.RegressionResults

statsmodels.regression.linear_model.OLSResults.cov_params
OLSResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)
    Returns the variance/covariance matrix.

    The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

    **Parameters**
    
    **r_matrix** : array-like
        Can be 1d, or 2d. Can be used alone or with other.
    
    **column** : array-like, optional
        Must be used on its own. Can be 0d or 1d see below.
    
    **scale** : float, optional
        Can be specified or not. Default is None, which means that the scale argument is taken from the model.
    
    **other** : array-like, optional
        Can be used when r_matrix is specified.

    **Returns**
    
    **cov** : ndarray
        If no argument is specified returns the covariance matrix of a model:
        
        (scale)*(X.T X)^(-1) :

        If contrast is specified it pre and post-multiplies as follows:
```
(scale) * r_matrix (X.T X)^(-1) r_matrix.T :
If contrast and other are specified returns :
(scale) * r_matrix (X.T X)^(-1) other.T :
If column is specified returns :
(scale) * (X.T X)^(-1)[column,column] if column is 0d :
OR :
(scale) * (X.T X)^(-1)[column][:,column] if column is 1d :

statsmodels.regression.linear_model.OLSResults.eigenvals
static OLSResults.eigenvals()
Return eigenvalues sorted in decreasing order.

statsmodels.regression.linear_model.OLSResults.el_test
OLSResults.el_test(b0_vals, param_nums, return_weights=0, ret_params=0, method='nm',
stochastic_exog=1, return_params=0)
Tests single or joint hypotheses of the regression parameters using Empirical Likelihood.

Parameters b0_vals : 1darray
   The hypothesized value of the parameter to be tested
param_nums : 1darray
   The parameter number to be tested
print_weights : bool
   If true, returns the weights that optimize the likelihood ratio at b0_vals. Default is False
ret_params : bool
   If true, returns the parameter vector that maximizes the likelihood ratio at b0_vals. Also
   returns the weights. Default is False
method : string
   Can either be ‘nm’ for Nelder-Mead or ‘powell’ for Powell. The optimization method
   that optimizes over nuisance parameters. Default is ‘nm’
stochastic_exog : bool
   When TRUE, the exogenous variables are assumed to be stochastic. When the regres-
   sors are nonstochastic, moment conditions are placed on the exogenous variables. Confid-
   ence intervals for stochastic regressors are at least as large as non-stochastic regressors.
   Default = TRUE

Returns res : tuple
   The p-value and -2 times the log-likelihood ratio for the hypothesized values.

Examples

>>> import statsmodels.api as sm
>>> data = sm.datasets.stackloss.load()
>>> endog = data.endog
>>> exog = sm.add_constant(data.exog)
```python
>>> model = sm.OLS(endog, exog)
>>> fitted = model.fit()
>>> fitted.params
array([-39.91967442, 0.7156402 , 1.29528612, -0.15212252])
>>> fitted.rsquared
0.91357690446068196
>>> # Test that the slope on the first variable is 0
>>> fitted.test_beta([0], [1])
(1.7894660442330235e-07, 27.248146353709153)
```

```python
statsmodels.regression.linear_model.OLSResults.ess
static OLSResults.ess()

statsmodels.regression.linear_model.OLSResults.f_pvalue
static OLSResults.f_pvalue()

statsmodels.regression.linear_model.OLSResults.f_test
OLSResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
Compute the F-test for a joint linear hypothesis.

Parameters:
- r_matrix : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.
- q_matrix : array-like
  This is deprecated. See r_matrix and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.
- cov_p : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
- scale : float, optional
  Default is 1.0 for no scaling.
- invcov : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy DesignInfo.linear_constraint

Notes
The matrix r_matrix is assumed to be non-singular. More precisely,
```
is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
```

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>
```

Compare this to

```python
>>> results.F 330.2853392346658
>>> results.F_p 4.98403096572e-10
```

```python
>>> B = np.array([[0,0,1,-1,0,0,0], [0,0,0,0,0,1,-1]])
```

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```
statsmodels.regression.linear_model.OLSResults.fittedvalues
static OLSResults.fittedvalues()

statsmodels.regression.linear_model.OLSResults.fvalue
static OLSResults.fvalue()

statsmodels.regression.linear_model.OLSResults.get_influence
OLSResults.get_influence()
get an instance of Influence with influence and outlier measures
```
Returns infl: Influence instance

the instance has methods to calculate the main influence and outlier measures for the
OLS regression

```
statsmodels.regression.linear_model.OLSResults.get_robustcov_results
```

create new results instance with robust covariance as default

Parameters:

- cov_type: string
  the type of robust sandwich estimator to use. see Notes below

- use_t: bool
  If true, then the t distribution is used for inference. If false, then the normal distribution
  is used.

- kwds: depends on cov_type
  Required or optional arguments for robust covariance calculation. see Notes below

Returns:

results instance

This method creates a new results instance with the requested robust covariance as the
default covariance of the parameters. Inferential statistics like p-values and hypothesis
tests will be based on this covariance matrix.

Notes

The following covariance types and required or optional arguments are currently available:

- ‘HC0’, ‘HC1’, ‘HC2’, ‘HC3’ and no keyword arguments: heteroscedasticity robust covariance
- ‘HAC’ and keywords
  - maxlag integer (required): number of lags to use
  - kernel string (optional): kernel, default is Bartlett
  - use_correction bool (optional): [If true, use small sample] correction
- ‘cluster’ and required keyword groups, integer group indicator
  - groups array_like, integer (required): index of clusters or groups
  - use_correction bool (optional): If True the sandwich covariance is calculated with a small sample correction. If False the the sandwich covariance is calculated without small sample correction.
  - df_correction bool (optional): If True (default), then the degrees of freedom for the inferential statistics and hypothesis tests, such as pvalues, f_pvalue, conf_int, and t_test and f_test, are based on the number of groups minus one instead of the total number of observations minus the number of explanatory variables. df_resid of the results instance is adjusted. If False, then df_resid of the results instance is not adjusted.
- ‘hac-groupsum’ Driscoll and Kraay, heteroscedasticity and autocorrelation robust standard errors in panel data keywords
  - time array_like (required): index of time periods
  - maxlag integer (required): number of lags to use
– **kernel** string (optional) : kernel, default is Bartlett

– **use_correction False or string in ['hac', ‘cluster’] (optional) :** If False the the sandwich co-
  variance is calculated without small sample correction. If use_correction = ‘cluster’ (default),
  then the same small sample correction as in the case of ‘covtype=‘cluster’” is used.

– **df_correction bool (optional)** adjustment to df_resid, see cov_type ‘cluster’ above #TODO:
  we need more options here

• **hac-panel’ heteroscedasticity and autocorrelation robust standard errors in panel data.** The
  data needs to be sorted in this case, the time series for each panel unit or cluster need to be
  stacked. keywords

  – **time array_like (required) :** index of time periods

  – **maxlag integer (required) :** number of lags to use

  – **kernel string (optional) :** kernel, default is Bartlett

  – **use_correction False or string in ['hac', ‘cluster’] (optional) :** If False the the sandwich co-
    variance is calculated without small sample correction.

  – **df_correction bool (optional)** adjustment to df_resid, see cov_type ‘cluster’ above #TODO:
    we need more options here

Reminder: **use_correction** in “nw-groupsum” and “nw-panel” is not bool, needs to be in [False, ‘hac’,
‘cluster’]

TODO: Currently there is no check for extra or misspelled keywords, except in the case of cov_type HCx

```python
statsmodels.regression.linear_model.OLSResults.initialize

OLSResults.initialize(model, params, **kwd)
```

```python
statsmodels.regression.linear_model.OLSResults.llf

static OLSResults.llf()
```

```python
statsmodels.regression.linear_model.OLSResults.load

classmethod OLSResults.load(fname)
    load a pickle, (class method)
```

Parameters **fname** : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

Returns **unpickled instance** :

```python
statsmodels.regression.linear_model.OLSResults.mse_model

static OLSResults.mse_model()
```

```python
statsmodels.regression.linear_model.OLSResults.mse_resid

static OLSResults.mse_resid()
```

```python
statsmodels.regression.linear_model.OLSResults.mse_total

static OLSResults.mse_total()
```
statsmodels.regression.linear_model.OLSResults.nobs

static OLSResults.nobs()

statsmodels.regression.linear_model.OLSResults.normalized_cov_params

OLSResults.normalized_cov_params()

statsmodels.regression.linear_model.OLSResults.outlier_test

OLSResults.outlier_test(method='bonf', alpha=0.05)

Test observations for outliers according to method

Parameters method : str

• bonferroni : one-step correction
• sidak : one-step correction
• holm-sidak :
• holm :
• simes-hochberg :
• hommel :
• fdr_bh : Benjamini/Hochberg
• fdr_by : Benjamini/Yekutieli

See statsmodels.stats.multitest.multipletests for details.

alpha : float

familywise error rate

Returns table : ndarray or DataFrame

Returns either an ndarray or a DataFrame if labels is not None. Will attempt to get
labels from model_results if available. The columns are the Studentized residuals, the
unadjusted p-value, and the corrected p-value according to method.

Notes

The unadjusted p-value is stats.t.sf(abs(resid), df) where df = df_resid - 1.

statsmodels.regression.linear_model.OLSResults.predict

OLSResults.predict(exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

Parameters exog : array-like, optional

The values for which you want to predict.

transform : bool, optional

If the model was fit via a formula, do you want to pass exog through the formula. Default
is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you
can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d
need to log the data first.

Returns See self.model.predict :
statsmodels.regression.linear_model.OLSResults.pvalues
static OLSResults.pvalues()

statsmodels.regression.linear_model.OLSResults.remove_data
OLSResults.remove_data()
remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

statsmodels.regression.linear_model.OLSResults.resid
static OLSResults.resid()

statsmodels.regression.linear_model.OLSResults.resid_pearson
static OLSResults.resid_pearson()
Residuals, normalized to have unit variance.

**Returns** An array $wresid/sqrt(scale)$:

statsmodels.regression.linear_model.OLSResults.rsquared
static OLSResults.rsquared()

statsmodels.regression.linear_model.OLSResults.rsquared_adj
static OLSResults.rsquared_adj()

statsmodels.regression.linear_model.OLSResults.save
OLSResults.save(fname, remove_data=False)
save a pickle of this instance

**Parameters**

fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

remove_data : bool
If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.
statsmodels.regression.linear_model.OLSResults.scale
    static OLSResults.scale()

statsmodels.regression.linear_model.OLSResults.ssr
    static OLSResults.ssr()

statsmodels.regression.linear_model.OLSResults.summary
    OLSResults.summary(yname=None, xname=None, title=None, alpha=0.05)
    Summarize the Regression Results

    Parameters
    yname : string, optional
        Default is y
    xname : list of strings, optional
        Default is var_## for ## in p the number of regressors
    title : string, optional
        Title for the top table. If not None, then this replaces the default title
    alpha : float
        significance level for the confidence intervals

    Returns
    smry : Summary instance
        this holds the summary tables and text, which can be printed or converted to various output formats.

    See also:

    statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.regression.linear_model.OLSResults.summary2
    OLSResults.summary2(yname=None, xname=None, title=None, alpha=0.05, float_format='%.4f')
    Experimental summary function to summarize the regression results

    Parameters
    xname : List of strings of length equal to the number of parameters
        Names of the independent variables (optional)
    yname : string
        Name of the dependent variable (optional)
    title : string, optional
        Title for the top table. If not None, then this replaces the default title
    alpha : float
        significance level for the confidence intervals
    float_format : string
        print format for floats in parameters summary

    Returns
    smry : Summary instance
        this holds the summary tables and text, which can be printed or converted to various output formats.
See also:

**statsmodels.iolib.summary.Summary** class to hold summary results

 statsmodels.regression.linear_model.OLSResults.t_test

OLSResults.t_test(*r_matrix*, *q_matrix=None, cov_p=None, scale=None, use_t=None)

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

**Parameters**

- **r_matrix** : array-like, str, tuple
  - array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like or scalar, optional
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p** : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale** : float, optional
  An optional scale to use. Default is the scale specified by the model fit.

- **use_t** : bool, optional
  If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

**tvalues** individual t statistics

**f_test** for F tests

patsy.DesignInfo.linear_constraint

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```
r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>> print(T_test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9> >>> T_test.effect -1829.2025687192481 >>> T_test.sd 455.39079425193762 >>> T_test.tvalue -4.0167754636411717 >>> T_test.pvalue 0.0015163772380899498
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data >>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR' >>> results = ols(formula, dta).fit() >>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1' >>> t_test = results.t_test(hypotheses) >>> print(t_test)
```

```python
statsmodels.regression.linear_model.OLSResults.tvalues
```

```
static OLSResults.tvalues()
Return the t-statistic for a given parameter estimate.
```

```
statsmodels.regression.linear_model.OLSResults.uncentered_tss
```

```
static OLSResults.uncentered_tss()
```

```
statsmodels.regression.linear_model.OLSResults.wald_test
```

```
OLSResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.
```

**Parameters**

- **r_matrix** : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like
  This is deprecated. See `r_matrix` and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.

- **cov_p** : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, `self.normalized_cov_params` is used.

- **scale** : float, optional
  Default is 1.0 for no scaling.

- **invcov** : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

- **use_f** : bool
  If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

3.1. Linear Regression 179
See also:

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix \( r_{matrix} \) is assumed to be non-singular. More precisely,

\( r_{matrix} (pX pX^T) r_{matrix}^T \)

is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be
problems in non-OLS models where the rank of the covariance of the noise is not full.

statsmodels.regression.linear_model.OLSResults.wresid

```
static OLSResults.wresid()
```

QuantRegResults(model, params[, ...]) Results instance for the QuantReg model

statsmodels.regression.quantile_regression.QuantRegResults

```
class statsmodels.regression.quantile_regression.QuantRegResults
```

Results instance for the QuantReg model

Methods

HC0_se()
HC1_se()
HC2_se()
HC3_se()
aic()
bic()
bse()
centered_tss()
compare_f_test(restricted)
compare_lm_test(restricted[, demean, use_lr])
compare_lr_test(restricted[, large_sample])
condition_number()
conf_int([alpha, cols])
cov_HC0()
cov_HC1()
cov_HC2()
cov_HC3()
cov_params([r_matrix, column, scale, cov_p, ...])
eigenvals()

f_pvalue()
### Table 3.17 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f_test(r_matrix[, q_matrix, cov_p, scale, ...])</code></td>
<td>Compute the F-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td><code>fittedvalues()</code></td>
<td></td>
</tr>
<tr>
<td><code>fvalue()</code></td>
<td></td>
</tr>
<tr>
<td><code>get_robustcov_results([cov_type, use_t])</code></td>
<td>create new results instance with robust covariance as default</td>
</tr>
<tr>
<td><code>initialize(model, params, **kwd)</code></td>
<td></td>
</tr>
<tr>
<td><code>llf()</code></td>
<td></td>
</tr>
<tr>
<td><code>load(fname)</code></td>
<td>load a pickle, (class method)</td>
</tr>
<tr>
<td><code>mle()</code></td>
<td></td>
</tr>
<tr>
<td><code>mle_model()</code></td>
<td></td>
</tr>
<tr>
<td><code>mle_resid()</code></td>
<td></td>
</tr>
<tr>
<td><code>mse()</code></td>
<td></td>
</tr>
<tr>
<td><code>mse_total()</code></td>
<td></td>
</tr>
<tr>
<td><code>nobs()</code></td>
<td></td>
</tr>
<tr>
<td><code>normalized_cov_params()</code></td>
<td></td>
</tr>
<tr>
<td><code>predict([exog, transform])</code></td>
<td>Call self.model.predict with self.params as the first argument.</td>
</tr>
<tr>
<td><code>prsquared()</code></td>
<td></td>
</tr>
<tr>
<td><code>pvalues()</code></td>
<td></td>
</tr>
<tr>
<td><code>remove_data()</code></td>
<td>remove data arrays, all nobs arrays from result and model</td>
</tr>
<tr>
<td><code>resid()</code></td>
<td>Residuals, normalized to have unit variance.</td>
</tr>
<tr>
<td><code>resid_pearson()</code></td>
<td></td>
</tr>
<tr>
<td><code>rsquared()</code></td>
<td></td>
</tr>
<tr>
<td><code>rsquared_adj()</code></td>
<td></td>
</tr>
<tr>
<td><code>save(fname[, remove_data])</code></td>
<td>save a pickle of this instance</td>
</tr>
<tr>
<td><code>scale()</code></td>
<td></td>
</tr>
<tr>
<td><code>ssr()</code></td>
<td></td>
</tr>
<tr>
<td><code>summary([yname, xname, title, alpha])</code></td>
<td>Summarize the Regression Results</td>
</tr>
<tr>
<td><code>summary2([yname, xname, title, alpha, ...])</code></td>
<td>Experimental summary function to summarize the regression results</td>
</tr>
<tr>
<td><code>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</code></td>
<td>Compute a t-test for a joint linear hypothesis of the form Rb = q</td>
</tr>
<tr>
<td><code>tvalues()</code></td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td><code>uncentered_tss()</code></td>
<td></td>
</tr>
<tr>
<td><code>wald_test(r_matrix[, q_matrix, cov_p, ...])</code></td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td><code>wresid()</code></td>
<td></td>
</tr>
</tbody>
</table>

```python
statsmodels.regression.quantile_regression.QuantRegResults.HC0_se
static QuantRegResults.HC0_se()
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.HC1_se
static QuantRegResults.HC1_se()
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.HC2_se
static QuantRegResults.HC2_se()
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.HC3_se
static QuantRegResults.HC3_se()
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.aic
static QuantRegResults.aic()
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.bic
static QuantRegResults.bic()
```
statsmodels.regression.quantile_regression.QuantRegResults.bse
static QuantRegResults.bse()

statsmodels.regression.quantile_regression.QuantRegResults.centered_tss
static QuantRegResults.centered_tss()

statsmodels.regression.quantile_regression.QuantRegResults.compare_f_test
QuantRegResults.compare_f_test (restricted)
use F test to test whether restricted model is correct

Parameters restricted : Result instance
   The restricted model is assumed to be nested in the current model. The result instance
   of the restricted model is required to have two attributes, residual sum of squares, ssr,
   residual degrees of freedom, df_resid.

Returns f_value : float
   test statistic, F distributed

   p_value : float
      p-value of the test statistic

   df_diff : int
      degrees of freedom of the restriction, i.e. difference in df between models

Notes

See mailing list discussion October 17,
This test compares the residual sum of squares of the two models. This is not a valid test, if there is
unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still
return the results under the assumption of homoscedasticity and no autocorrelation (sphericity).

statsmodels.regression.quantile_regression.QuantRegResults.compare_lm_test
QuantRegResults.compare_lm_test (restricted, demean=True, use_lr=False)
Use Lagrange Multiplier test to test whether restricted model is correct

Parameters restricted : Result instance
   The restricted model is assumed to be nested in the current model. The result instance
   of the restricted model is required to have two attributes, residual sum of squares, ssr,
   residual degrees of freedom, df_resid.

demean : bool
   Flag indicating whether the demean the scores based on the residuals from the restricted
   model. If True, the covariance of the scores are used and the LM test is identical to the
   large sample version of the LR test.

Returns lm_value : float
   test statistic, chi2 distributed

   p_value : float
      p-value of the test statistic
df_diff : int

degrees of freedom of the restriction, i.e. difference in df between models

Notes

TODO: explain LM text

statsmodels.regression.quantile_regression.QuantRegResults.compare_lr_test

QuantRegResults.compare_lr_test(restricted, large_sample=False)

Likelihood ratio test to test whether restricted model is correct

Parameters restricted : Result instance

The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, ssr, residual degrees of freedom, df_resid.

large_sample : bool

Flag indicating whether to use a heteroskedasticity robust version of the LR test, which is a modified LM test.

Returns lr_stat : float

likelihood ratio, chisquare distributed with df_diff degrees of freedom

p_value : float

p-value of the test statistic

df_diff : int

degrees of freedom of the restriction, i.e. difference in df between models

Notes

The exact likelihood ratio is valid for homoskedastic data, and is defined as

\[ D = -2 \log \left( \frac{\mathcal{L}_{null}}{\mathcal{L}_{alternative}} \right) \]

where \( \mathcal{L} \) is the likelihood of the model. With \( D \) distributed as chisquare with df equal to difference in number of parameters or equivalently difference in residual degrees of freedom.

The large sample version of the likelihood ratio is defined as

\[ D = ns' S^{-1} s \]

where \( s = n^{-1} \sum_{i=1}^{n} s_i \)

\[ s_i = x_{i, alternative} \epsilon_{i, null} \]

is the average score of the model evaluated using the residuals from null model and the regressors from the alternative model and \( S \) is the covariance of the scores, \( s_i \). The covariance of the scores is estimated using the same estimator as in the alternative model.
This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results without taking unspecified heteroscedasticity or correlation into account.

This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results without taking unspecified heteroscedasticity or correlation into account.

\( \frac{s}{s_i} \) is the average score of the model evaluated using the residuals from null model and the regressors from the alternative model and \( S \) is the covariance of the scores, \( s_i \). The covariance of the scores is estimated using the same estimator as in the alternative model.

TODO: put into separate function, needs tests

```python
statsmodels.regression.quantile_regression.QuantRegResults.condition_number
static QuantRegResults.condition_number()
Return condition number of exogenous matrix, calculated as ratio of largest to smallest eigenvalue.
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.conf_int
QuantRegResults.conf_int(alpha=0.05, cols=None)
Returns the confidence interval of the fitted parameters.

Parameters

- alpha : float, optional
  The alpha level for the confidence interval. ie., The default \( \alpha = .05 \) returns a 95% confidence interval.
- cols : array-like, optional
  cols specifies which confidence intervals to return
```

**Notes**

The confidence interval is based on Student’s t-distribution.

```python
statsmodels.regression.quantile_regression.QuantRegResults.cov_HC0
static QuantRegResults.cov_HC0()
See statsmodels.RegressionResults
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.cov_HC1
static QuantRegResults.cov_HC1()
See statsmodels.RegressionResults
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.cov_HC2
static QuantRegResults.cov_HC2()
See statsmodels.RegressionResults
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.cov_HC3
static QuantRegResults.cov_HC3()
See statsmodels.RegressionResults
```
statsmodels.regression.quantile_regression.QuantRegResults.cov_params

QuantRegResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

**Parameters**

- **r_matrix**: array-like
  - Can be 1d, or 2d. Can be used alone or with other.
- **column**: array-like, optional
  - Must be used on its own. Can be 0d or 1d see below.
- **scale**: float, optional
  - Can be specified or not. Default is None, which means that the scale argument is taken from the model.
- **other**: array-like, optional
  - Can be used when r_matrix is specified.

**Returns**

(The below are assumed to be in matrix notation.)

- **cov**: ndarray
  - If no argument is specified returns the covariance matrix of a model:
    
    (scale)*(X.T X)^(-1):
  - If contrast is specified it pre and post-multiplies as follows:
    
    (scale) * r_matrix (X.T X)^(-1) r_matrix.T:
  - If contrast and other are specified returns:
    
    (scale) * r_matrix (X.T X)^(-1) other.T:
  - If column is specified returns:
    
    (scale) * (X.T X)^(-1)[column,column] if column is 0d:
    
    OR:
    
    (scale) * (X.T X)^(-1)[column][;,column] if column is 1d:

statsmodels.regression.quantile_regression.QuantRegResults.eigenvals

static QuantRegResults.eigenvals()

Return eigenvalues sorted in decreasing order.

statsmodels.regression.quantile_regression.QuantRegResults.ess

static QuantRegResults.ess()

statsmodels.regression.quantile_regression.QuantRegResults.f_pvalue

static QuantRegResults.f_pvalue()
Statsmodels Documentation, Release 0.6.0

Statsmodels.regression.quantile_regression.QuantRegResults.f_test

QuantRegResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
Computes the F-test for a joint linear hypothesis.

Parameters
- **r_matrix**: array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.
- **q_matrix**: array-like
  This is deprecated. See r_matrix and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.
- **cov_p**: array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
- **scale**: float, optional
  Default is 1.0 for no scaling.
- **invcov**: array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts.statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,
r_matrix (pX pX.T) r_matrix.T
is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
```
This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>
```

186 Chapter 3. Table of Contents
Compare this to

```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10
```nn

```python
>>> B = np.array([[0,0,1,-1,0,0,0], [0,0,0,0,0,1,-1]])
```nn

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>
```nn

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
```nn

```python
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

In the `statsmodels.regression.quantile_regression` module, the methods are defined as follows:

```python
statsmodels.regression.quantile_regression.QuantRegResults.fittedvalues
```nn

```python
static QuantRegResults.fittedvalues()
```nn

```python
statsmodels.regression.quantile_regression.QuantRegResults.fvalue
```nn

```python
static QuantRegResults.fvalue()
```nn

```python
statsmodels.regression.quantile_regression.QuantRegResults.get_robustcov_results
```nn

```python
QuantRegResults.get_robustcov_results(cov_type='HC1', use_t=None, **kwds)
```nn

create new results instance with robust covariance as default

**Parameters**

- `cov_type` : string
  the type of robust sandwich estimator to use. see Notes below

- `use_t` : bool
  If true, then the t distribution is used for inference. If false, then the normal distribution is used.

- `kwds` : depends on cov_type
  Required or optional arguments for robust covariance calculation. see Notes below

**Returns**

- `results` : results instance
  This method creates a new results instance with the requested robust covariance as the default covariance of the parameters. Inferential statistics like p-values and hypothesis tests will be based on this covariance matrix.
Notes

The following covariance types and required or optional arguments are currently available:

- **HC0**, **HC1**, **HC2**, **HC3** and no keyword arguments: heteroscedasticity robust covariance
- **HAC** and keywords
  - `maxlag` integer (required): number of lags to use
  - `kernel` string (optional): kernel, default is Bartlett
  - `use_correction` bool (optional): [If true, use small sample] correction
- **cluster** and required keyword `groups`, integer group indicator
  - `groups` array_like, integer (required): index of clusters or groups
  - `use_correction` bool (optional): If True the sandwich covariance is calculated with a small sample correction. If False the sandwich covariance is calculated without small sample correction.
  - `df_correction` bool (optional): If True (default), then the degrees of freedom for the inferential statistics and hypothesis tests, such as pvalues, f_pvalue, conf_int, and t_test and f_test, are based on the number of groups minus one instead of the total number of observations minus the number of explanatory variables. `df_resid` of the results instance is adjusted. If False, then `df_resid` of the results instance is not adjusted.
- **hac-groupsum** Driscoll and Kraay, heteroscedasticity and autocorrelation robust standard errors in panel data keywords
  - `time` array_like (required): index of time periods
  - `maxlag` integer (required): number of lags to use
  - `kernel` string (optional): kernel, default is Bartlett
  - `use_correction` False or string in ['hac', 'cluster'] (optional): If False the sandwich covariance is calculated without small sample correction. If `use_correction = 'cluster'` (default), then the same small sample correction as in the case of `cov_type='cluster'` is used.
  - `df_correction` bool (optional): adjustment to df_resid, see cov_type 'cluster' above #TODO: we need more options here
- **hac-panel** heteroscedasticity and autocorrelation robust standard errors in panel data. The data needs to be sorted in this case, the time series for each panel unit or cluster need to be stacked. keywords
  - `time` array_like (required): index of time periods
  - `maxlag` integer (required): number of lags to use
  - `kernel` string (optional): kernel, default is Bartlett
  - `use_correction` False or string in ['hac', 'cluster'] (optional): If False the sandwich covariance is calculated without small sample correction.
  - `df_correction` bool (optional): adjustment to df_resid, see cov_type 'cluster' above #TODO: we need more options here

Reminder: `use_correction` in “nw-groupsum” and “nw-panel” is not bool, needs to be in [False, ‘hac’, ‘cluster’]

TODO: Currently there is no check for extra or misspelled keywords, except in the case of cov_type HCx
statsmodels.regression.quantile_regression.QuantRegResults.initialize
QuantRegResults.initialize(model, params, **kwds)

statsmodels.regression.quantile_regression.QuantRegResults.llf
static QuantRegResults.llf()

statsmodels.regression.quantile_regression.QuantRegResults.load
classmethod QuantRegResults.load(fname)
load a pickle, (class method)

Parameters fname: string or filehandle
fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance:

statsmodels.regression.quantile_regression.QuantRegResults.mse
static QuantRegResults.mse()

statsmodels.regression.quantile_regression.QuantRegResults.mse_model
static QuantRegResults.mse_model()

statsmodels.regression.quantile_regression.QuantRegResults.mse_resid
static QuantRegResults.mse_resid()

statsmodels.regression.quantile_regression.QuantRegResults.mse_total
static QuantRegResults.mse_total()

statsmodels.regression.quantile_regression.QuantRegResults.nobs
static QuantRegResults.nobs()

statsmodels.regression.quantile_regression.QuantRegResults.normalized_cov_params
QuantRegResults.normalized_cov_params()

statsmodels.regression.quantile_regression.QuantRegResults.predict
QuantRegResults.predict(exog=None, transform=True, *args, **kwargs)
Call self.model.predict with self.params as the first argument.

Parameters exog: array-like, optional
The values for which you want to predict.

transform: bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

Returns See self.model.predict:
QuantRegResults.prssquared

QuantRegResults.pvalues

QuantRegResults.remove_data

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

QuantRegResults.resid

QuantRegResults.resid_pearson

Residuals, normalized to have unit variance.

Returns An array wresid/sqrt(scale):

QuantRegResults.rsquared

QuantRegResults.rsquared_adj

QuantRegResults.save

save a pickle of this instance

Parameters fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

remove_data : bool

If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.
Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

```python
statsmodels.regression.quantile_regression.QuantRegResults.scale
QuantRegResults.scale()
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.ssr
static QuantRegResults.ssr()
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.summary
QuantRegResults.summary(yname=None, xname=None, title=None, alpha=0.05)
```

Summarize the Regression Results

**Parameters**
- `yname` : string, optional
  Default is `y`
- `xname` : list of strings, optional
  Default is `var_##` for `##` in `p` the number of regressors
- `title` : string, optional
  Title for the top table. If not None, then this replaces the default title
- `alpha` : float
  significance level for the confidence intervals

**Returns**
- `smry` : Summary instance
  this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

```python
statsmodels.iolib.summary.Summary class to hold summary results
```

```python
statsmodels.regression.quantile_regression.QuantRegResults.summary2
QuantRegResults.summary2(yname=None, xname=None, title=None, alpha=0.05, float_format='%.4f')
```

Experimental summary function to summarize the regression results

**Parameters**
- `xname` : List of strings of length equal to the number of parameters
  Names of the independent variables (optional)
- `yname` : string
  Name of the dependent variable (optional)
- `title` : string, optional
  Title for the top table. If not None, then this replaces the default title
- `alpha` : float
  significance level for the confidence intervals
**float_format**: string

  print format for floats in parameters summary

**Returns**

  **smry**: Summary instance

  this holds the summary tables and text, which can be printed or converted to various output formats.

**See also**:

statsmodels.iolib.summary.Summary class to hold summary results

**statsmodels.regression.quantile_regression.QuantRegResults.t_test**

QuantRegResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

**Parameters**

  **r_matrix**: array-like, str, tuple

  - array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

  **q_matrix**: array-like or scalar, optional

  This is deprecated. See r_matrix and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

  **cov_p**: array-like, optional

  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

  **scale**: float, optional

  An optional scale to use. Default is the scale specified by the model fit.

  **use_t**: bool, optional

  If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

**See also**:

tvalues individual t statistics

f_test for F tests

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
```
```python
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```

\( r \) tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_test = results.t_test(r) >>> print(T_test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
```

statsmodels.regression.quantile_regression.QuantRegResults.tvalues

`static QuantRegResults.tvalues()`

Return the t-statistic for a given parameter estimate.

statsmodels.regression.quantile_regression.QuantRegResults.uncentered_tss

`static QuantRegResults.uncentered_tss()`

statsmodels.regression.quantile_regression.QuantRegResults.wald_test

`QuantRegResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)`

Compute a Wald-test for a joint linear hypothesis.

**Parameters**

- \( r_{\text{matrix}} \) : array-like, str, or tuple
  - array : An \( r \times k \) array where \( r \) is the number of restrictions to test and \( k \) is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form \((R, q)\), since \( q_{\text{matrix}} \) is deprecated.

- \( q_{\text{matrix}} \) : array-like
  
  This is deprecated. See \( r_{\text{matrix}} \) and the examples for more information on new usage. Can be either a scalar or a length \( p \) row vector. If omitted and \( r_{\text{matrix}} \) is an array, \( q_{\text{matrix}} \) is assumed to be a conformable array of zeros.

- \( \text{cov}_p \) : array-like, optional
  
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- scale : float, optional
  
  Default is 1.0 for no scaling.

- \( \text{invcov} \) : array-like, optional

---

3.1. Linear Regression

193
A \( q \times q \) array to specify an inverse covariance matrix based on a restrictions matrix.

\[ \text{use}_f : \text{bool} \]

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix \( r_{\text{matrix}} \) is assumed to be non-singular. More precisely,

\[ r_{\text{matrix}} \left( \mathbf{pX} \mathbf{pX}^T \right) r_{\text{matrix}}^T \]

is assumed invertible. Here, \( \mathbf{pX} \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

3.2 Generalized Linear Models

Generalized linear models currently supports estimation using the one-parameter exponential families

See Module Reference for commands and arguments.

3.2.1 Examples

```python
# Load modules and data
import statsmodels.api as sm
data = sm.datasets.scotland.load()
data.exog = sm.add_constant(data.exog)

# Instantiate a gamma family model with the default link function.
gamma_model = sm.GLM(data.endog, data.exog, family=sm.families.Gamma())
gamma_results = gamma_model.fit()
```

Detailed examples can be found here:

3.2.2 Technical Documentation

References

3.2.3 Module Reference

Model Class

GLM(endog, exog[, family, offset, exposure, ...]) Generalized Linear Models class

GLM inherits from statsmodels.LikelihoodModel

Parameters
- endog : array-like
  1d array of endogenous response variable. This array can be 1d or 2d. Binomial family models accept a 2d array with two columns. If supplied, each observation is expected to be [success, failure].
- exog : array-like
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See statsmodels.tools.add_constant.
- family : family class instance
  The default is Gaussian. To specify the binomial distribution family = sm.family.Binomial() Each family can take a link instance as an argument. See statsmodels.family.family for more information.
- missing : str
  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’

See also:
statsmodels.families.

Notes

Only the following combinations make sense for family and link

+ ident log logit probit cloglog pow opow nbinom loglog logc
Gaussian | x | x | x
inv Gaussian | x | x | x
binomial | x | x | x | x | x | x | x | x
Poisson | x | x | x | x
Not all of these link functions are currently available.

Endog and exog are references so that if the data they refer to are already arrays and these arrays are changed, endog and exog will change.

Attributes

- **df_model** [float] Model degrees of freedom is equal to p - 1, where p is the number of regressors. Note that the intercept is not reported as a degree of freedom.

- **df_resid** [float] Residual degrees of freedom is equal to the number of observation n minus the number of regressors p.

- **endog** [array] See above. Note that endog is a reference to the data so that if data is already an array and it is changed, then endog changes as well.

- **exposure** [array-like] Include ln(exposure) in model with coefficient constrained to 1. Can only be used if the link is the logarithm function.

- **exog** [array] See above. Note that endog is a reference to the data so that if data is already an array and it is changed, then endog changes as well.

- **iteration** [int] The number of iterations that fit has run. Initialized at 0.

- **family** [family class instance] The distribution family of the model. Can be any family in statsmodels.families. Default is Gaussian.

- **mu** [array] The mean response of the transformed variable. \( \mu \) is the value of the inverse of the link function at eta, where eta is the linear predicted value of the WLS fit of the transformed variable. \( \mu \) is only available after fit is called. See statsmodels.families.family.fitted of the distribution family for more information.

- **normalized_cov_params** [array] The p x p normalized covariance of the design / exogenous data. This is approximately equal to \((X.T X)^{-1}\)

- **offset** [array-like] Include offset in model with coefficient constrained to 1.

- **pinv_wexog** [array] The pseudoinverse of the design / exogenous data array. Note that GLM has no whiten method, so this is just the pseudo inverse of the design. The pseudoinverse is approximately equal to \((X.T X)^{-1}X.T\)

- **scale** [float] The estimate of the scale / dispersion of the model fit. Only available after fit is called. See GLM.fit and GLM.estimate_scale for more information.

- **scaletype** [str] The scaling used for fitting the model. This is only available after fit is called. The default is None. See GLM.fit for more information.

- **weights** [array] The value of the weights after the last iteration of fit. Only available after fit is called. See statsmodels.families.family for the specific distribution weighting functions.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.scotland.load()
>>> data.exog = sm.add_constant(data.exog)
```

Instantiate a gamma family model with the default link function.
>>> gamma_model = sm.GLM(data.endog, data.exog,
...    family=sm.families.Gamma())

>>> gamma_results = gamma_model.fit()
>>> gamma_results.params
array([-0.01776527, 0.00004962, 0.00203442, -0.00007181, 0.00011185,
        -0.00000015, -0.00051868, -0.00000243])

>>> gamma_results.scale
0.0035842831734919055
>>> gamma_results.deviance
0.087388516416999198
>>> gamma_results.pearson_chi2
0.086022796163805704
>>> gamma_results.llf
-83.017202161073527

Attributes

| df_model       | float       | $p - 1$, where $p$ is the number of regressors including the intercept. |
| df_resid       | float       | The number of observation $n$ minus the number of regressors $p$. |
| endog          | array       | See Parameters. |
| exog           | array       | See Parameters. |
| family         | family class instance | A pointer to the distribution family of the model. |
| mu             | array       | The estimated mean response of the transformed variable. |
| normalized_cov_params | array | $p \times p$ normalized covariance of the design / exogenous data. |
| pinv_wexog     | array       | For GLM this is just the pseudo inverse of the original design. |
| scale          | float       | The estimate of the scale / dispersion. Available after fit is called. |
| scaletype      | str         | The scaling used for fitting the model. Available after fit is called. |
| weights        | array       | The value of the weights after the last iteration of fit. |

Methods

estimate_scale(mu) Estimates the dispersion/scale.
fit([start_params, maxiter, method, tol, scale]) Fits a generalized linear model for a given family.
from_formula(formula, data[, subset]) Create a Model from a formula and dataframe.
heessian(params) The Hessian matrix of the model.
information(params) Fisher information matrix.
initialize() Initialize a generalized linear model.
loglike(*args) Loglikelihood function.
predict(params[, exog, exposure, offset, linear]) Return predicted values for a design matrix.
score(params) Score matrix.

statsmodels.genmod.generalized_linear_model.GLM.estimate_scale
GLM.estimate_scale (mu)
Estimates the dispersion/scale.

Type of scale can be chose in the fit method.

**Parameters**

- mu : array
  
  mu is the mean response estimate

**Returns**

Estimate of scale :

See also:

statsmodels.glm.fit

Notes

The default scale for Binomial and Poisson families is 1. The default for the other families is Pearson’s Chi-Square estimate.

```python
GLM.fit(start_params=None, maxiter=100, method='IRLS', tol=1e-08, scale=None)
```

Fits a generalized linear model for a given family.

**Parameters**

- maxiter : int, optional
  
  Default is 100.

- method : string
  
  Default is ‘IRLS’ for iteratively reweighted least squares. This is currently the only method available for GLM fit.

- scale : string or float, optional
  
  `scale` can be ‘X2’, ‘dev’, or a float. The default value is None, which uses X2 for Gamma, Gaussian, and Inverse Gaussian. X2 is Pearson’s chi-square divided by df_resid. The default is 1 for the Binomial and Poisson families. dev is the deviance divided by df_resid

- tol : float
  
  Convergence tolerance. Default is 1e-8.

- start_params : array-like, optional
  
  Initial guess of the solution for the loglikelihood maximization. The default is family-specific and is given by the family.starting_mu(endog). If start_params is given then the initial mean will be calculated as np.dot(exog, start_params).

```python
GLM.from_formula(formula, data, subset=None, *args, **kwargs)
```

Create a Model from a formula and dataframe.

**Parameters**

- formula : str or generic Formula object
  
  The formula specifying the model

- data : array-like
  
  The data for the model. See Notes.

- subset : array-like
An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

args : extra arguments
These are passed to the model

kwargs : extra keyword arguments
These are passed to the model.

Returns model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

statsmodels.genmod.generalized_linear_model.GLM.hessian
GLM.hessian (params)
The Hessian matrix of the model

statsmodels.genmod.generalized_linear_model.GLM.information
GLM.information (params)
Fisher information matrix. Not yet implemented.

statsmodels.genmod.generalized_linear_model.GLM.initialize
GLM.initialize ()
Initialize a generalized linear model.

statsmodels.genmod.generalized_linear_model.GLM.loglike
GLM.loglike (*args)
Loglikelihood function.
Each distribution family has its own loglikelihood function. See statsmodels.families.family

statsmodels.genmod.generalized_linear_model.GLM.predict
GLM.predict (params, exog=None, exposure=None, offset=None, linear=False)
Return predicted values for a design matrix

Parameters params : array-like
Parameters / coefficients of a GLM.

exog : array-like, optional
Design / exogenous data. Is exog is None, model exog is used.

exposure : array-like, optional
Exposure time values, only can be used with the log link function. See notes for details.

offset : array-like, optional
Offset values. See notes for details.

linear : bool
If True, returns the linear predicted values. If False, returns the value of the inverse of the model’s link function at the linear predicted values.

Returns An array of fitted values:

Notes

Any exposure and offset provided here take precedence over the exposure and offset used in the model fit. If exog is passed as an argument here, then any exposure and offset values in the fit will be ignored.

Exposure values must be strictly positive.

statsmodels.genmod.generalized_linear_model.GLM.score

GLM.score(params)

Score matrix. Not yet implemented

Attributes

- endog_names
- exog_names

Results Class

GLMResults(model, params, ...) Class to contain GLM results.

statsmodels.genmod.generalized_linear_model.GLMResults

class statsmodels.genmod.generalized_linear_model.GLMResults(model, params, normalized_cov_params, scale)

Class to contain GLM results.

GLMResults inherits from statsmodels.LikelihoodModelResults

Parameters See statsmodels.LikelihoodModelResults:

Returns **Attributes**:

- aic: float
  - Akaike Information Criterion \(-2 \times llf + 2 \times (df\_model + 1)\)
- bic: float
  - Bayes Information Criterion \(\text{deviance} - df\_resid \times \log(nobs)\)
- deviance: float
  - See statsmodels.families.family for the distribution-specific deviance functions.
- df_model: float
  - See GLM.df_model
- df_resid: float
See GLM.df_resid

**fit_history** : dict
Contains information about the iterations. Its keys are *iterations*, *deviance* and *params*.

**fittedvalues** : array
Linear predicted values for the fitted model. dot(exog, params)

**llf** : float
Value of the loglikelihood function evaluated at params. See statsmodels.families.family for distribution-specific loglikelihoods.

**model** : class instance
Pointer to GLM model instance that called fit.

**mu** : array
See GLM docstring.

**nobs** : float
The number of observations n.

**normalized_cov_params** : array
See GLM docstring

**null_deviance** : float
The value of the deviance function for the model fit with a constant as the only regressor.

**params** : array
The coefficients of the fitted model. Note that interpretation of the coefficients often depends on the distribution family and the data.

**pearson_chi2** : array
Pearson’s Chi-Squared statistic is defined as the sum of the squares of the Pearson residuals.

**pinv_wexog** : array
See GLM docstring.

**pvalues** : array
The two-tailed p-values for the parameters.

**resid_anscombe** : array
Anscombe residuals. See statsmodels.families.family for distribution-specific Anscombe residuals.

**resid_deviance** : array
Deviance residuals. See statsmodels.families.family for distribution-specific deviance residuals.

**resid_pearson** : array
Pearson residuals. The Pearson residuals are defined as \((endog - mu) / \sqrt{VAR(mu)})\) where VAR is the distribution specific variance function. See statsmodels.families.family and statsmodels.families.varfuncs for more information.

3.2. Generalized Linear Models
resid_response : array
Response residuals. The response residuals are defined as endog - fittedvalues

resid_working : array
Working residuals. The working residuals are defined as resid_response / link'(mu). See statsmodels.family.links for the derivatives of the link functions. They are defined analytically.

scale : float
The estimate of the scale / dispersion for the model fit. See GLM.fit and GLM.estimate_scale for more information.

stand_errors : array
The standard errors of the fitted GLM. #TODO still named bse

See also:
statsmodels.LikelihoodModelResults

Methods

aic()
bic()
bse()
conf_int([alpha, cols, method]) Returns the confidence interval of the fitted parameters.
cov_params([r_matrix, column, scale, cov_p, ...]) Returns the variance/covariance matrix.
deviance()
f_test([r_matrix, q_matrix, cov_p, scale, ...]) Compute the F-test for a joint linear hypothesis.
fittedvalues()
initialize(model, params, **kwd)
llf()
load(fname) load a pickle, (class method)
normalized_cov_params()
null()
null_deviance()
pearson_chi2()
predict([exog, transform]) Call self.model.predict with self.params as the first argument.
pvalues()
remove_data() remove data arrays, all nobs arrays from result and model
resid_anscombe()
resid_deviance()
resid_pearson()
resid_response()
resid_working()
save(fname[, remove_data]) save a pickle of this instance
summary([yname, xname, title, alpha]) Summarize the Regression Results
summary2([yname, xname, title, alpha, ...]) Experimental summary for regression Results
t_test([r_matrix, q_matrix, cov_p, scale, use_t]) Compute a t-test for a joint linear hypothesis of the form Rb = q
tvalues()
wald_test([r_matrix, q_matrix, cov_p, ...]) Compute a Wald-test for a joint linear hypothesis.
statsmodels.genmod.generalized_linear_model.GLMResults.aic

statsmodels.genmod.generalized_linear_model.GLMResults.bic

statsmodels.genmod.generalized_linear_model.GLMResults.conf_int

Returns the confidence interval of the fitted parameters.

Parameters

- **alpha**: float, optional
  The alpha level for the confidence interval. i.e., The default alpha = .05 returns a 95% confidence interval.

- **cols**: array-like, optional
  cols specifies which confidence intervals to return

- **method**: string
  Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

Returns **conf_int**: array
Each row contains [lower, upper] confidence interval

Notes
The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[ 5496529.48322745, -1467987.78596704],
        [ -177.02903529,  207.15277984],
        [ -0.1115811 ,   0.03994274],
        [ -3.12506664,  -0.91539297],
        [ -1.5179487 ,  -0.54850503],
        [ -0.56251721,   0.460309  ],
        [  798.7875153 ,  2859.51541392]])
```

```python
>>> results.conf_int(cols=(2,3))
array([[ 0.1115811 ,   0.03994274],
        [ -3.12506664,  -0.91539297]])
```

3.2. Generalized Linear Models
GLMResults.cov_params

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters
- **r_matrix** : array-like
  - Can be 1d, or 2d. Can be used alone or with other.

- **column** : array-like, optional
  - Must be used on its own. Can be 0d or 1d see below.

- **scale** : float, optional
  - Can be specified or not. Default is None, which means that the scale argument is taken from the model.

Other : array-like, optional
  - Can be used when r_matrix is specified.

Returns

- **cov** : ndarray
  - If no argument is specified returns the covariance matrix of a model:
    - (scale)*(X.T X)^(-1)
  - If contrast is specified it pre and post-multiplies as follows:
    - (scale) * r_matrix (X.T X)^(-1) r_matrix.T
  - If contrast and other are specified returns:
    - (scale) * r_matrix (X.T X)^(-1) other.T
  - If column is specified returns:
    - (scale) * (X.T X)^(-1)[column,column] if column is 0d
    - OR:
    - (scale) * (X.T X)^(-1)[column][:,column] if column is 1d

GLMResults.deviance

Compute the F-test for a joint linear hypothesis.

Parameters
- **r_matrix** : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like
This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.

**cov_p**: array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

**scale**: float, optional

Default is 1.0 for no scaling.

**invcov**: array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix `r_matrix` is assumed to be non-singular. More precisely, `r_matrix (pX pX.T) r_matrix.T` is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
```

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>
```

Compare this to

```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10
```

```python
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
```

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>

Alternatively, you can specify the hypothesis tests using a string

>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>>
dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
GLMResults.predict(exog=None, transform=True, **args, **kwargs)

Call self.model.predict with self.params as the first argument.

Parameters:
- **exog**: array-like, optional
  The values for which you want to predict.
- **transform**: bool, optional
  If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model $y \sim \log(x1) + \log(x2)$, and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you'd need to log the data first.

Returns:
See self.model.predict:

---

GLMResults.pvalues()

GLMResults.remove_data()

remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

GLMResults.resid_anscombe()

GLMResults.resid_deviance()

GLMResults.resid_pearson()

GLMResults.resid_response()

GLMResults.resid_working()
GLMResults.save(fname, remove_data=False)

save a pickle of this instance

**Parameters**

- **fname** : string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- **remove_data** : bool
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

GLMResults.summary(yname=None, xname=None, title=None, alpha=0.05)

Summarize the Regression Results

**Parameters**

- **yname** : string, optional
  
  Default is \(y\)

- **xname** : list of strings, optional
  
  Default is \(\text{var}_{##}\) for ## in p the number of regressors

- **title** : string, optional
  
  Title for the top table. If not None, then this replaces the default title

- **alpha** : float
  
  significance level for the confidence intervals

**Returns**

- **smry** : Summary instance
  
  this holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

- **statsmodels.iolib.summary.Summary** class to hold summary results

GLMResults.summary2(yname=None, xname=None, title=None, alpha=0.05, float_format='%.4f')

Experimental summary for regression Results

**Parameters**

- **yname** : string
  
  Name of the dependent variable (optional)

- **xname** : List of strings of length equal to the number of parameters
  
  Names of the independent variables (optional)

- **title** : string, optional
Title for the top table. If not None, then this replaces the default title

**alpha** : float

significance level for the confidence intervals

**float_format** : string :

print format for floats in parameters summary

**Returns** smry : Summary instance

this holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

*statsmodels.iolib.summary.Summary* class to hold summary results

**statsmodels.genmod.generalized_linear_model.GLMResults.t_test**

GLMResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)

Compute a t-test for a joint linear hypothesis of the form Rb = q

**Parameters** r_matrix : array-like, str, tuple

- array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

**See also:**

tvalues individual t statistics

f_test for F tests

patsy.DesignInfo.linear_constraint
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]
```

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_test)
<T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> data = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, data).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

```
statsmodels.genmod.generalized_linear_model.GLMResults.tvalues
static GLMResults.tvalues()  
Return the t-statistic for a given parameter estimate.
```

```
statsmodels.genmod.generalized_linear_model.GLMResults.wald_test
GLMResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
  • array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  • str : The full hypotheses to test can be given as a string. See the examples.
  • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.
q_matrix : array-like
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.
cov_p : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
scale : float, optional
  Default is 1.0 for no scaling.
```
invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,

\[ r_matrix (pX pX^T) r_matrix^T \]

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Families

The distribution families currently implemented are

<table>
<thead>
<tr>
<th>Family(link, variance)</th>
<th>The parent class for one-parameter exponential families.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binomial((link))</td>
<td>Binomial exponential family distribution.</td>
</tr>
<tr>
<td>Gamma((link))</td>
<td>Gamma exponential family distribution.</td>
</tr>
<tr>
<td>Gaussian((link))</td>
<td>Gaussian exponential family distribution.</td>
</tr>
<tr>
<td>InverseGaussian((link))</td>
<td>InverseGaussian exponential family.</td>
</tr>
<tr>
<td>NegativeBinomial((link, alpha))</td>
<td>Negative Binomial exponential family.</td>
</tr>
<tr>
<td>Poisson((link))</td>
<td>Poisson exponential family.</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.family.Family

class statsmodels.genmod.families.family.Family(link, variance)

The parent class for one-parameter exponential families.

Parameters

link : a link function instance

Link is the linear transformation function. See the individual families for available links.

variance : a variance function

Measures the variance as a function of the mean probabilities. See the individual families for the default variance function.

Methods

deviance(Y, mu[, scale])  Deviance of (Y, mu) pair.
fitted(eta)               Fitted values based on linear predictors eta.
Table 3.24 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loglike(Y, mu[, scale])</code></td>
<td>The loglikelihood function.</td>
</tr>
<tr>
<td><code>predict(mu)</code></td>
<td>Linear predictors based on given mu values.</td>
</tr>
<tr>
<td><code>resid_anscombe(Y, mu)</code></td>
<td>The Anscombe residuals.</td>
</tr>
<tr>
<td><code>resid_dev(Y, mu[, scale])</code></td>
<td>The deviance residuals</td>
</tr>
<tr>
<td><code>starting_mu(y)</code></td>
<td>Starting value for mu in the IRLS algorithm.</td>
</tr>
<tr>
<td><code>weights(mu)</code></td>
<td>Weights for IRLS steps</td>
</tr>
</tbody>
</table>

**statsmodels.genmod.families.family.Family.deviance**

Family. `deviance(Y, mu, scale=1.0)`

Deviance of (Y, mu) pair.

Deviance is usually defined as twice the loglikelihood ratio.

**Parameters**

- `Y` : array_like
  - The endogenous response variable
- `mu` : array-like
  - The inverse of the link function at the linear predicted values.
- `scale` : float, optional
  - An optional scale argument

**Returns**

- `DEV` : array
  - The value of deviance function defined below.

**Notes**

\[
DEV = \frac{(\text{sum}_i(2\cdot\text{loglike}(Y_i,Y_i) - 2\cdot\text{loglike}(Y_i,mu_i)))}{\text{scale}}
\]

The deviance functions are analytically defined for each family.

**statsmodels.genmod.families.family.Family.fitted**

Family. `fitted(eta)`

Fitted values based on linear predictors eta.

**Parameters**

- `eta` : array
  - Values of the linear predictor of the model. dot(X, beta) in a classical linear model.

**Returns**

- `mu` : array
  - The mean response variables given by the inverse of the link function.

**statsmodels.genmod.families.family.Family.loglike**

Family. `loglike(Y, mu, scale=1.0)`

The loglikelihood function.

**Parameters**

- `Y` : array
  - Usually the endogenous response variable.
- `mu` : array
  - Usually but not always the fitted mean response variable.

**Returns**

- `llf` : float
The value of the loglikelihood evaluated at $(Y, \mu)$.

**Notes**:

This is defined for each family. $Y$ and $\mu$ are not restricted to ‘$Y$’ and ‘$\mu$’ respectively. For instance, the deviance function calls both $\loglike(Y, Y)$ and $\loglike(Y, \mu)$ to get the likelihood ratio.

**statsmodels.genmod.families.family.Family.predict**

Family.predict($\mu$)

Linear predictors based on given $\mu$ values.

**Parameters** $\mu$ : array
The mean response variables

**Returns** $\eta$ : array
Linear predictors based on the mean response variables. The value of the link function at the given $\mu$.

**statsmodels.genmod.families.family.Family.resid_anscombe**

Family.resid_anscombe($Y, \mu$)

The Anscombe residuals.

**See also**:
statsmodels.families.family.Family

**statsmodels.genmod.families.family.Family.resid_dev**

Family.resid_dev($Y, \mu, scale=1.0$)

The deviance residuals

**Parameters** $Y$ : array
The endogenous response variable

$\mu$ : array
The inverse of the link function at the linear predicted values.

scale : float, optional
An optional argument to divide the residuals by scale

**Returns** Deviance residuals.

**Notes**

The deviance residuals are defined for each family.
statsmodels.genmod.families.family.Family.starting_mu

Family.starting_mu(y)

Starting value for mu in the IRLS algorithm.

Parameters:
- y : array
  The untransformed response variable.

Returns:
- mu_0 : array
  The first guess on the transformed response variable.

Notes

Only the Binomial family takes a different initial value.

statsmodels.genmod.families.family.Family.weights

Family.weights(mu)

Weights for IRLS steps

Parameters:
- mu : array-like
  The transformed mean response variable in the exponential family

Returns:
- w : array
  The weights for the IRLS steps

Notes

\[ w = \frac{1}{(\text{link}'(mu)^2 \times \text{variance}(mu))} \]

statsmodels.genmod.families.family.Binomial

class statsmodels.genmod.families.family.Binomial(link=<class 'statsmodels.genmod.families.links.logit'>)

Binomial exponential family distribution.

Parameters:
- link : a link instance, optional
  The default link for the Binomial family is the logit link. Available links are logit, probit, cauchy, log, and cloglog. See statsmodels.family.links for more information.

See also:

statsmodels.genmod.families.family.Family

Notes

endog for Binomial can be specified in one of three ways.

Attributes

<table>
<thead>
<tr>
<th>Binomial.link</th>
<th>a link instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binomial.variance</td>
<td>varfunc instance</td>
</tr>
</tbody>
</table>

The link function of the Binomial instance

\[ variance \text{ is an instance of statsmodels.family.varfuncs.binary} \]
statsmodels Documentation, Release 0.6.0

Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviance(Y, mu[, scale])</td>
<td>Deviance function for either Bernoulli or Binomial data.</td>
</tr>
<tr>
<td>fitted(eta)</td>
<td>Fitted values based on linear predictors eta.</td>
</tr>
<tr>
<td>initialize(Y)</td>
<td>Initialize the response variable.</td>
</tr>
<tr>
<td>loglike(Y, mu[, scale])</td>
<td>Loglikelihood function for Binomial exponential family distribution.</td>
</tr>
<tr>
<td>predict(mu)</td>
<td>Linear predictors based on given mu values.</td>
</tr>
<tr>
<td>resid_anscombe(Y, mu)</td>
<td>The Anscombe residuals</td>
</tr>
<tr>
<td>resid_dev(Y, mu[, scale])</td>
<td>Binomial deviance residuals</td>
</tr>
<tr>
<td>starting_mu(y)</td>
<td>The starting values for the IRLS algorithm for the Binomial family.</td>
</tr>
<tr>
<td>weights(mu)</td>
<td>Weights for IRLS steps</td>
</tr>
</tbody>
</table>

```
statsmodels.genmod.families.family.Binomial.deviance
Binomial.deviance(Y, mu, scale=1.0)

Deviance function for either Bernoulli or Binomial data.

Parameters

Y : array-like
   Endogenous response variable (already transformed to a probability if appropriate).
mu : array
   Fitted mean response variable
scale : float, optional
   An optional scale argument

Returns

deviance : float
   The deviance function as defined below

Notes

If the endogenous variable is binary:

deviance = -2*sum(I_one * log(mu) + (I_zero)*log(1-mu))

where I_one is an indicator function that evaluates to 1 if Y_i == 1. and I_zero is an indicator function
that evaluates to 1 if Y_i == 0.

If the model is binomial:

deviance = 2*sum(log(Y/mu) + (n-Y)*log((n-Y)/(n-mu))) where Y and n are as defined in Binomial.initialize.
```

```
statsmodels.genmod.families.family.Binomial.fitted
Binomial.fitted(eta)

Fitted values based on linear predictors eta.

Parameters

eta : array
   Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

Returns

mu : array
   The mean response variables given by the inverse of the link function.
```
Binomial.initialize(Y)
Initialize the response variable.

Parameters Y : array
   Endogenous response variable

Returns If `Y` is binary, returns `Y`:
   If `Y` is a 2d array, then the input is assumed to be in the format:
   (successes, failures) and:
   successes/(success + failures) is returned. And n is set to:
   successes + failures.

Binomial.loglike(Y, mu, scale=1.0)
Loglikelihood function for Binomial exponential family distribution.

Parameters Y : array-like
   Endogenous response variable
   mu : array-like
      Fitted mean response variable
   scale : float, optional
      The default is 1.

Returns llf : float
   The value of the loglikelihood function evaluated at (Y, mu, scale) as defined below.

Notes
If Y is binary: \( llf = scale \cdot \sum(Y \cdot \log(mu/(1-mu)) + \log(1-mu)) \)
If Y is binomial: \( llf = scale \cdot \sum(\text{gammaln}(n+1) - \text{gammaln}(y+1) - \text{gammaln}(n-y+1) + y \cdot \log(mu/(1-mu)) + n \cdot \log(1-mu)) \)
where gammaln is the log gamma function and y = Y*n with Y and n as defined in Binomial initialize.
This simply makes y the original number of successes.

Binomial.predict(mu)
Linear predictors based on given mu values.

Parameters mu : array
   The mean response variables

Returns eta : array
   Linear predictors based on the mean response variables. The value of the link function
   at the given mu.
Statsmodels Documentation, Release 0.6.0

statsmodels.genmod.families.family.Binomial.resid_anscombe

Binomial.resid_anscombe(Y, mu)

The Anscombe residuals

**Parameters**

Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

**Returns**

resid_anscombe : array

The Anscombe residuals as defined below.

**Notes**

sqrt(n)*(cox_snell(Y)-cox_snell(mu))/(mu**(1/6.)*(1-mu)**(1/6.))

where cox_snell is defined as cox_snell(x) = betainc(2/3., 2/3., x)*betainc(2/3.,2/3.) where betainc is the incomplete beta function

The name ‘cox_snell’ is idiosyncratic and is simply used for convenience following the approach suggested in Cox and Snell (1968). Further note that cox_snell(x) = x**(2/3.)/(2/3.)*hyp2f1(2/3.,1/3.,5/3.,x) where hyp2f1 is the hypergeometric 2f1 function. The Anscombe residuals are sometimes defined in the literature using the hyp2f1 formulation. Both betainc and hyp2f1 can be found in scipy.

**References**


statsmodels.genmod.families.family.Binomial.resid_dev

Binomial.resid_dev(Y, mu, scale=1.0)

Binomial deviance residuals

**Parameters**

Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

scale : float, optional

An optional argument to divide the residuals by scale

**Returns**

resid_dev : array

Deviance residuals as defined below.

3.2. Generalized Linear Models

217
Notes

If $Y$ is binary:
\[
\text{resid}_d = \text{sign}(Y - \mu) \cdot \sqrt{-2 \log(I_{\text{one}} \cdot \mu + I_{\text{zero}} \cdot (1 - \mu))}
\]
where $I_{\text{one}}$ is an indicator function that evaluates as 1 if $Y == 1$ and $I_{\text{zero}}$ is an indicator function that evaluates as 1 if $Y == 0$.

If $Y$ is binomial:
\[
\text{resid}_d = \text{sign}(Y - \mu) \cdot \sqrt{2n \cdot (Y \log(Y/\mu) + (1 - Y) \log((1 - Y)/(1 - \mu)))}
\]
where $Y$ and $n$ are as defined in Binomial.initialize.

**statsmodels.genmod.families.family.Binomial.starting_mu**

Binomial.starting_mu($y$)

The starting values for the IRLS algorithm for the Binomial family.

A good choice for the binomial family is

\[\text{starting}_\mu = (y + .5)/2\]

**statsmodels.genmod.families.family.Binomial.weights**

Binomial.weights($\mu$)

Weights for IRLS steps

**Parameters**

- $\mu$: array-like

  The transformed mean response variable in the exponential family

**Returns**

- $w$: array

  The weights for the IRLS steps

**Notes**

\[w = 1 / (\text{link}(\mu)^2 \cdot \text{variance}(\mu))\]

**statsmodels.genmod.families.family.Gamma**

class statsmodels.genmod.families.family.Gamma(**link=statsmodels.genmod.families.links.inverse_power**)

Gamma exponential family distribution.

**Parameters**

- $\text{link}$: a link instance, optional

  The default link for the Gamma family is the inverse link. Available links are log, identity, and inverse. See statsmodels.family.links for more information.

**See also:**

statsmodels.genmod.families.family.Family
Attributes

<table>
<thead>
<tr>
<th>Gamma.link</th>
<th>a link instance</th>
<th>The link function of the Gamma instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gamma.variance</td>
<td>varfunc instance</td>
<td>variance is an instance of statsmodels.family.varfuncs.mu_squared</td>
</tr>
</tbody>
</table>

Methods

- **deviance**(*Y*, *mu*, scale=1.0)  
  Gamma deviance function
  - Parameters
    - **Y**: array-like
      Endogenous response variable
    - **mu**: array-like
      Fitted mean response variable
    - **scale**: float, optional
      An optional scale argument
  - Returns
    - **deviance**: float
      Deviance function as defined below

  
  \[deviance = 2 \times \text{sum}((Y - \mu)/\mu - \log(Y/\mu))\]

- **fitted**(*eta*)  
  Fitted values based on linear predictors eta.
  - Parameters **eta**: array
  - Returns **mu**: array

- **loglike**(*Y*, *mu*, scale)  
  Loglikelihood function for Gamma exponential family distribution.
  - Parameters
    - **Y**: array-like
    - **mu**: array-like
    - **scale**: float, optional
  - Returns **deviance**: float

- **predict**(*mu*)  
  Linear predictors based on given mu values.
  - Parameters **mu**: array
  - Returns **eta**: array

- **resid_anscombe**(*Y*, *mu*)  
  The Anscombe residuals for Gamma exponential family distribution
  - Parameters
    - **Y**: array-like
    - **mu**: array-like
  - Returns **resid_anscombe**: array

- **resid_dev**(*Y*, *mu*, scale)  
  Gamma deviance residuals
  - Parameters
    - **Y**: array-like
    - **mu**: array-like
    - **scale**: float, optional
  - Returns **resid_dev**: array

- **starting_mu**(*y*)  
  Starting value for mu in the IRLS algorithm.
  - Parameters **y**: array
  - Returns **mu**: array

- **weights**(*mu*)  
  Weights for IRLS steps
  - Parameters **mu**: array-like
  - Returns **weights**: array

---

3.2. Generalized Linear Models

**statsmodels.genmod.families.family.Gamma.deviance**  
Gamma deviance function

**Parameters**
- **Y**: array-like  
  Endogenous response variable
- **mu**: array-like  
  Fitted mean response variable
- **scale**: float, optional  
  An optional scale argument

**Returns**
- **deviance**: float  
  Deviance function as defined below

\[deviance = 2 \times \text{sum}((Y - \mu)/\mu - \log(Y/\mu))\]

**statsmodels.genmod.families.family.Gamma.fitted**  
Fitted values based on linear predictors eta.

**Parameters**
- **eta**: array  
  Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

**Returns**
- **mu**: array  
  The mean response variables given by the inverse of the link function.
statsmodels.genmod.families.family.Gamma.loglike

Gamma.loglike(Y, mu, scale=1.0)
Loglikelihood function for Gamma exponential family distribution.

**Parameters**

- **Y**: array-like
  Endogenous response variable
- **mu**: array-like
  Fitted mean response variable
- **scale**: float, optional
  The default is 1.

**Returns**

- **llf**: float
  The value of the loglikelihood function evaluated at (Y,mu,scale) as defined below.

**Notes**

\[
llf = -1/scale \times \sum (Y/mu + \log(mu) + (scale-1)*\log(Y) + \log(scale) + scale*\text{gammaln}(1/scale))
\]
where \text{gammaln} is the log gamma function.

statsmodels.genmod.families.family.Gamma.predict

Gamma.predict(mu)
Linear predictors based on given mu values.

**Parameters**

- **mu**: array
  The mean response variables

**Returns**

- **eta**: array
  Linear predictors based on the mean response variables. The value of the link function at the given mu.

statsmodels.genmod.families.family.Gamma.resid_anscombe

Gamma.resid_anscombe(Y, mu)
The Anscombe residuals for Gamma exponential family distribution.

**Parameters**

- **Y**: array
  Endogenous response variable
- **mu**: array
  Fitted mean response variable

**Returns**

- **resid_anscombe**: array
  The Anscombe residuals for the Gamma family defined below

**Notes**

\[
\text{resid_anscombe} = 3*(Y^{(1/3.)}-\text{mu}^{(1/3.)})/\text{mu}^{(1/3.)}
\]
Gamma.resid_dev(Y, mu, scale=1.0)

Gamma deviance residuals

Parameters

Y : array-like
   Endogenous response variable
mu : array-like
   Fitted mean response variable
scale : float, optional
   An optional argument to divide the residuals by scale

Returns resid_dev : array
   Deviance residuals as defined below

Notes

\[ resid_{dev} = \text{sign}(Y - \mu) \times \sqrt{-2 \times \left(-\frac{Y-\mu}{\mu} + \log\left(\frac{Y}{\mu}\right)\right)} \]

Gamma.starting_mu(y)

Starting value for \( \mu \) in the IRLS algorithm.

Parameters

y : array
   The untransformed response variable.

Returns mu_0 : array
   The first guess on the transformed response variable.

Notes

Only the Binomial family takes a different initial value.

Gamma.weights(mu)

Weights for IRLS steps

Parameters

mu : array-like
   The transformed mean response variable in the exponential family

Returns

w : array
   The weights for the IRLS steps

Notes

\[ w = \frac{1}{(\text{link}'(\mu)^2 \times \text{variance}(\mu))} \]
statsmodels.genmod.families.family.Gaussian

class statsmodels.genmod.families.family.Gaussian(link=<class 'statsmodels.genmod.families.links.identity'>)

Gaussian exponential family distribution.

Parameters link : a link instance, optional

The default link for the Gaussian family is the identity link. Available links are log,
identity, and inverse. See statsmodels.family.links for more information.

See also:
statsmodels.genmod.families.family.Family

Attributes

Gaussian.link : a link instance

Gaussian.variance : varfunc instance

variance is an instance of statsmodels.family.varfuncs.constant

Methods

deviance(Y, mu[, scale]) : Gaussian deviance function

fitted(eta) : Fitted values based on linear predictors eta.

loglike(Y, mu[, scale]) : Loglikelihood function for Gaussian exponential family distribution.

predict(mu) : Linear predictors based on given mu values.

resid_anscombe(Y, mu) : The Anscombe residuals for the Gaussian exponential family distribution

resid_dev(Y, mu[, scale]) : Gaussian deviance residuals

starting_mu(y) : Starting value for mu in the IRLS algorithm.

weights(mu) : Weights for IRLS steps

statsmodels.genmod.families.family.Gaussian.deviance

Gaussian.deviance(Y, mu, scale=1.0)

Gaussian deviance function

Parameters Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

scale : float, optional

An optional scale argument

Returns deviance : float

The deviance function at (Y, mu) as defined below.

Notes

deviance = sum((Y-mu)**2)
statsmodels.genmod.families.family.Gaussian.fitted

Gaussian.fitted(eta)

Fitted values based on linear predictors eta.

Parameters eta : array

Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

Returns mu : array

The mean response variables given by the inverse of the link function.

statsmodels.genmod.families.family.Gaussian.loglike

Gaussian.loglike(Y, mu, scale=1.0)

Loglikelihood function for Gaussian exponential family distribution.

Parameters Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

scale : float, optional

Scales the loglikelihood function. The default is 1.

Returns llf : float

The value of the loglikelihood function evaluated at (Y, mu, scale) as defined below.

Notes

If the link is the identity link function then the loglikelihood function is the same as the classical OLS model. llf = -(nobs/2)*(log(SSR) + (1 + log(2*pi/nobs))) where SSR = sum((Y-link^(-1)(mu))^2)

If the links is not the identity link then the loglikelihood function is defined as llf = sum((Y'*mu-mu'*mu/2)/scale - Y'^*Y/2*scale) - (1/2.)*log(2*pi*scale))

statsmodels.genmod.families.family.Gaussian.predict

Gaussian.predict(mu)

Linear predictors based on given mu values.

Parameters mu : array

The mean response variables

Returns eta : array

Linear predictors based on the mean response variables. The value of the link function at the given mu.

statsmodels.genmod.families.family.Gaussian.resid_anscombe

Gaussian.resid_anscombe(Y, mu)

The Anscombe residuals for the Gaussian exponential family distribution

Parameters Y : array

Endogenous response variable
mu : array
    Fitted mean response variable

Returns resid_anscombe : array
    The Anscombe residuals for the Gaussian family defined below

Notes

resid_anscombe = Y - mu

statsmodels.genmod.families.family.Gaussian.resid_dev
Gaussian.resid_dev(Y, mu, scale=1.0)
Gaussian deviance residuals

Parameters Y : array-like
    Endogenous response variable
mu : array-like
    Fitted mean response variable
scale : float, optional
    An optional argument to divide the residuals by scale

Returns resid_dev : array
    Deviance residuals as defined below

Notes

resid_dev = (Y - mu)/sqrt(variance(mu))

statsmodels.genmod.families.family.Gaussian.starting_mu
Gaussian.starting_mu(y)
Starting value for mu in the IRLS algorithm.

Parameters y : array
    The untransformed response variable.

Returns mu_0 : array
    The first guess on the transformed response variable.

Notes

Only the Binomial family takes a different initial value.
Gaussian.weights

Gaussian.weights(mu)

Weights for IRLS steps

**Parameters**

mu : array-like

The transformed mean response variable in the exponential family

**Returns**

w : array

The weights for the IRLS steps

**Notes**

\[ w = \frac{1}{(\text{link}'(\mu)^2 \cdot \text{variance}(\mu))} \]

InverseGaussian

class statsmodels.genmod.families.family.InverseGaussian

InverseGaussian exponential family.

**Parameters**

link : a link instance, optional

The default link for the inverse Gaussian family is the inverse squared link. Available links are inverse_squared, inverse, log, and identity. See statsmodels.family.links for more information.

**Notes**

The inverse Guassian distribution is sometimes referred to in the literature as the wald distribution.

**Attributes**

<table>
<thead>
<tr>
<th>InverseGaussian.link</th>
<th>a link instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>InverseGaussian.variance</td>
<td>varfunc instance</td>
</tr>
</tbody>
</table>

**Methods**

<table>
<thead>
<tr>
<th>deviance(Y, mu[, scale])</th>
<th>Inverse Gaussian deviance function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fitted(eta)</td>
<td>Fitted values based on linear predictors eta.</td>
</tr>
<tr>
<td>loglike(Y, mu[, scale])</td>
<td>Loglikelihood function for inverse Gaussian distribution.</td>
</tr>
<tr>
<td>predict(mu)</td>
<td>Linear predictors based on given mu values.</td>
</tr>
<tr>
<td>resid_anscombe(Y, mu)</td>
<td>The Anscombe residuals for the inverse Gaussian distribution</td>
</tr>
<tr>
<td>resid_dev(Y, mu[, scale])</td>
<td>Returns the deviance residuals for the inverse Gaussian family.</td>
</tr>
<tr>
<td>starting_mu(mu)</td>
<td>Starting value for mu in the IRLS algorithm.</td>
</tr>
<tr>
<td>weights(mu)</td>
<td>Weights for IRLS steps</td>
</tr>
</tbody>
</table>

3.2. Generalized Linear Models
**Invers Gaussian deviance**

\[ \text{deviance} = \sum \left( \frac{(y - \mu)^2}{y \mu^2} \right) \]

**Fitted values**

\[ \mu = \frac{\eta}{\alpha} \]

**Loglikelihood function**

\[ \ell = \frac{-1}{2} \sum \left( \frac{(y - \mu)^2}{y \mu^2 \alpha} \right) + \log(\alpha y^3) + \log(2\pi) \]
**statsmodels.genmod.families.family.InverseGaussian.predict**

InverseGaussian.predict(mu)

Linear predictors based on given mu values.

**Parameters**

- mu : array
  The mean response variables

**Returns**

- eta : array
  Linear predictors based on the mean response variables. The value of the link function at the given mu.

---

**statsmodels.genmod.families.family.InverseGaussian.resid_anscombe**

InverseGaussian.resid_anscombe(Y, mu)

The Anscombe residuals for the inverse Gaussian distribution

**Parameters**

- Y : array
  Endogenous response variable
- mu : array
  Fitted mean response variable

**Returns**

- resid_anscombe : array
  The Anscombe residuals for the inverse Gaussian distribution as defined below

**Notes**

\[
resid_{anscombe} = \log(Y/mu)/\sqrt{mu}
\]

---

**statsmodels.genmod.families.family.InverseGaussian.resid_dev**

InverseGaussian.resid_dev(Y, mu, scale=1.0)

Returns the deviance residuals for the inverse Gaussian family.

**Parameters**

- Y : array-like
  Endogenous response variable
- mu : array-like
  Fitted mean response variable
- scale : float, optional
  An optional argument to divide the residuals by scale

**Returns**

- resid_dev : array
  Deviance residuals as defined below

**Notes**

\[
dev_{resid} = \text{sign}(Y-mu) \times \sqrt{(Y-mu)^2/(Y\times mu^2)}
\]

---

3.2. Generalized Linear Models 227
**statsmodels.genmod.families.family.InverseGaussian.starting_mu**

**InverseGaussian.starting_mu(y)**

Starting value for mu in the IRLS algorithm.

**Parameters**

- **y**: array
  
  The untransformed response variable.

**Returns**

- **mu_0**: array
  
  The first guess on the transformed response variable.

**Notes**

Only the Binomial family takes a different initial value.

**statsmodels.genmod.families.family.InverseGaussian.weights**

**InverseGaussian.weights(mu)**

Weights for IRLS steps

**Parameters**

- **mu**: array-like
  
  The transformed mean response variable in the exponential family

**Returns**

- **w**: array
  
  The weights for the IRLS steps

**Notes**

\[ w = 1 / (\text{link}'(\mu)^2 \times \text{variance}(\mu)) \]

**statsmodels.genmod.families.family.NegativeBinomial**

**class statsmodels.genmod.families.family.NegativeBinomial(link=<class 'statsmodels.genmod.families.links.log'>, alpha=1.0)**

Negative Binomial exponential family.

**Parameters**

- **link**: a link instance, optional
  
  The default link for the negative binomial family is the log link. Available links are log, cloglog, identity, nbinom and power. See statsmodels.family.links for more information.

- **alpha**: float, optional
  
  The ancillary parameter for the negative binomial distribution. For now alpha is assumed to be nonstochastic. The default value is 1. Permissible values are usually assumed to be between 0.01 and 2.

**See also:**

statsmodels.genmod.families.family.Family

**Notes**

Support for Power link functions is not yet supported.
Attributes

<table>
<thead>
<tr>
<th>NegativeBinomial.link</th>
<th>a link instance</th>
<th>The link function of the negative binomial instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>NegativeBinomial.variance</td>
<td>varfunc instance</td>
<td>variance is an instance of statsmodels.family.varfuncs.nbinom</td>
</tr>
</tbody>
</table>

Methods

- `deviance(Y, mu[, scale])` Returns the value of the deviance function.
- `fitted(eta)` Fitted values based on linear predictors eta.
- `loglike(Y[, fittedvalues])` The loglikelihood function for the negative binomial family.
- `predict(mu)` Linear predictors based on given mu values.
- `resid_anscombe(Y, mu)` The Anscombe residuals for the negative binomial family
- `resid_dev(Y, mu[, scale])` Negative Binomial Deviance Residual
- `starting_mu(y)` Starting value for mu in the IRLS algorithm.
- `weights(mu)` Weights for IRLS steps

```
statsmodels.genmod.families.family.NegativeBinomial.deviance

NegativeBinomial.deviance(Y, mu, scale=1.0)

Returns the value of the deviance function.

Parameters

- **Y** : array-like
  Endogenous response variable
- **mu** : array-like
  Fitted mean response variable
- **scale** : float, optional
  An optional scale argument

Returns **deviance** : float
Deviance function as defined below

Notes

\[
deviance = \text{sum}(\text{piecewise})
\]
where piecewise is defined as
if \( Y_i = 0 \):
  piecewise_i = \frac{2 \log (1 + \alpha \mu)}{\alpha}
if \( Y_i > 0 \):
  piecewise_i = 2Y (\log(Y/\mu) - \frac{2}{\alpha (1 + \alpha Y)} \log((1 + \alpha Y)/(1 + \mu))

```

```
statsmodels.genmod.families.family.NegativeBinomial.fitted

NegativeBinomial.fitted(eta)
Fitted values based on linear predictors eta.

Parameters

- **eta** : array
  Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

3.2. Generalized Linear Models
Returns mu : array

The mean response variables given by the inverse of the link function.

```
statsmodels.genmod.families.family.NegativeBinomial.loglike
```

NegativeBinomial.loglike(Y, fittedvalues=None)

The loglikelihood function for the negative binomial family.

Parameters Y : array-like

Endogenous response variable

fittedvalues : array-like

The linear fitted values of the model. This is dot(exog.params).

Returns llf : float

The value of the loglikelihood function evaluated at (Y,mu,scale) as defined below.

Notes

\[ \text{sum}(Y \times \log(\alpha \times \exp(fittedvalues)/(1+\alpha \times \exp(fittedvalues)) - \log(1+\alpha \times \exp(fittedvalues))/\alpha + \text{constant}) \]

where constant is defined as constant = \( \text{gammaln}(Y + 1/\alpha) - \text{gammaln}(Y + 1) - \text{gammaln}(1/\alpha) \)

```
statsmodels.genmod.families.family.NegativeBinomial.predict
```

NegativeBinomial.predict(mu)

Linear predictors based on given mu values.

Parameters mu : array

The mean response variables

Returns eta : array

Linear predictors based on the mean response variables. The value of the link function at the given mu.

```
statsmodels.genmod.families.family.NegativeBinomial.resid_anscombe
```

NegativeBinomial.resid_anscombe(Y, mu)

The Anscombe residuals for the negative binomial family

Parameters Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

Returns resid_anscombe : array

The Anscombe residuals as defined below.
Notes

\[ \text{resid}_\text{anscombe} = \frac{(\text{hyp2f1}(-\alpha Y) - \text{hyp2f1}(-\alpha \mu) + 1.5(Y^{(2/3.)} - \mu^{(2/3.)})/((\mu+\alpha\mu^2)^{1/6.})}{(\mu+\alpha\mu^2)^{1/6.}} \]

where \( \text{hyp2f1} \) is the hypergeometric 2f1 function parameterized as \( \text{hyp2f1}(x) = \text{hyp2f1}(2/3., 1/3., 5/3., x) \)

\[ \text{statsmodels.genmod.families.family.NegativeBinomial.resid_dev} \]
\[ \text{NegativeBinomial}.\text{resid}_\text{dev}(Y, \mu, \text{scale}=1.0) \]

Negative Binomial Deviance Residual

Parameters

- \( Y \): array-like
  - \( Y \) is the response variable
- \( \mu \): array-like
  - \( \mu \) is the fitted value of the model
- \( \text{scale} \): float, optional
  - An optional argument to divide the residuals by scale

Returns

- \( \text{resid}_\text{dev} \): array
  - The array of deviance residuals

Notes

\[ \text{resid}_\text{dev} = \text{sign}(Y-\mu) \times \sqrt{\text{piecewise}} \]

where \( \text{piecewise} \) is defined as:
- if \( Y_i = 0 \): \( \text{piecewise}_i = 2 \times \log(1 + \alpha \mu) / \alpha \)
- if \( Y_i > 0 \): \( \text{piecewise}_i = 2 \times Y \times \log(Y/\mu) - 2/\alpha \times (1 + \alpha \times Y) \times \log((1 + \alpha \times Y)/(1 + \alpha \times \mu)) \)

\[ \text{statsmodels.genmod.families.family.NegativeBinomial.starting_mu} \]
\[ \text{NegativeBinomial}.\text{starting}_\text{mu}(y) \]

Starting value for \( \mu \) in the IRLS algorithm.

Parameters

- \( y \): array
  - The untransformed response variable.

Returns

- \( \mu_0 \): array
  - The first guess on the transformed response variable.

Notes

Only the Binomial family takes a different initial value.

\[ \text{statsmodels.genmod.families.family.NegativeBinomial.weights} \]
\[ \text{NegativeBinomial}.\text{weights}(\mu) \]

Weights for IRLS steps

Parameters

- \( \mu \): array-like
  - The transformed mean response variable in the exponential family

3.2. Generalized Linear Models
Returns  w : array

The weights for the IRLS steps

Notes

\[ w = \frac{1}{(\text{link}'(\mu)^2 \ast \text{variance}(\mu))} \]

statsmodels.genmod.families.family.Poisson

class statsmodels.genmod.families.family.Poisson(link=<class 'statsmodels.genmod.families.links.log'>)

Poisson exponential family.

Parameters  link : a link instance, optional

The default link for the Poisson family is the log link. Available links are log, identity, and sqrt. See statsmodels.family.links for more information.

See also:

statsmodels.genmod.families.family.Family

Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson.link</td>
<td>a link instance</td>
</tr>
<tr>
<td>Poisson.variance</td>
<td>varfuncs instance</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviance</td>
<td>Poisson deviance function</td>
</tr>
<tr>
<td>fitted</td>
<td>Fitted values based on linear predictors eta.</td>
</tr>
<tr>
<td>loglike</td>
<td>Loglikelihood function for Poisson exponential family distribution.</td>
</tr>
<tr>
<td>predict</td>
<td>Linear predictors based on given mu values.</td>
</tr>
<tr>
<td>resid_anscombe</td>
<td>Anscombe residuals for the Poisson exponential family distribution</td>
</tr>
<tr>
<td>resid_dev</td>
<td>Poisson deviance residual</td>
</tr>
<tr>
<td>starting_mu</td>
<td>Starting value for mu in the IRLS algorithm.</td>
</tr>
<tr>
<td>weights</td>
<td>Weights for IRLS steps</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.family.Poisson.deviance

Poisson.deviance \((Y, \mu, \text{scale}=1.0)\)

Poisson deviance function

Parameters  Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

scale : float, optional

An optional scale argument
Returns deviance : float

The deviance function at (Y, mu) as defined below.

Notes

If a constant term is included it is defined as

\[ deviance = 2 \sum_i (Y \times \log(Y/\mu)) \]

.. statistics.models.genmod.families.family.Poisson.fitted

Poisson.fitted(eta)

Fitted values based on linear predictors eta.

Parameters eta : array

Values of the linear predictor of the model. dot(X, beta) in a classical linear model.

Returns mu : array

The mean response variables given by the inverse of the link function.

.. statistics.models.genmod.families.family.Poisson.loglike

Poisson.loglike(Y, mu, scale=1.0)

Loglikelihood function for Poisson exponential family distribution.

Parameters Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

scale : float, optional

The default is 1.

Returns llf : float

The value of the loglikelihood function evaluated at (Y, mu, scale) as defined below.

Notes

\[ llf = scale \times \text{sum(-mu + Y * \log(mu) - gammaln(Y+1))} \]

where gammaln is the log gamma function

.. statistics.models.genmod.families.family.Poisson.predict

Poisson.predict(mu)

Linear predictors based on given mu values.

Parameters mu : array

The mean response variables

Returns eta : array

Linear predictors based on the mean response variables. The value of the link function at the given mu.
Poisson.resid_anscombe(Y, mu)

Anscombe residuals for the Poisson exponential family distribution

Parameters:
- **Y**: array-like
  - Endogenous response variable
- **mu**: array-like
  - Fitted mean response variable

Returns:
- **resid_anscombe**: array
  - The Anscombe residuals for the Poisson family defined below

Notes:

\[
\text{resid_anscombe} = \left(\frac{3}{2}\right) \times \left(Y^{2/3} - \mu \times (2/3)\right)/\mu^{1/6}.
\]

Poisson.resid_dev(Y, mu, scale=1.0)

Poisson deviance residual

Parameters:
- **Y**: array-like
  - Endogenous response variable
- **mu**: array-like
  - Fitted mean response variable
- **scale**: float, optional
  - An optional argument to divide the residuals by scale

Returns:
- **resid_dev**: array
  - Deviance residuals as defined below

Notes:

\[
\text{resid_dev} = \text{sign}(Y-mu) \times \sqrt{(2 \times Y \times \log(Y/mu) - 2 \times (Y-mu))}
\]

Poisson.starting_mu(y)

Starting value for mu in the IRLS algorithm.

Parameters:
- **y**: array
  - The untransformed response variable.

Returns:
- **mu_0**: array
  - The first guess on the transformed response variable.

Notes:

Only the Binomial family takes a different initial value.
statsmodels.genmod.families.family.Poisson.weights

Poisson.weights(mu)
Weights for IRLS steps

Parameters
mu : array-like
The transformed mean response variable in the exponential family

Returns
w : array
The weights for the IRLS steps

Notes

\[ w = 1 / (\text{link}'(mu)^2 \times \text{variance}(mu)) \]

Link Functions

The link functions currently implemented are the following. Not all link functions are available for each distribution family. The list of available link functions can be obtained by

```python
>>> sm.families.family.<familyname>.links
```

<table>
<thead>
<tr>
<th>Link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A generic link function for one-parameter exponential family.</td>
<td></td>
</tr>
<tr>
<td>CDFLink([dbn])</td>
<td>The use the CDF of a scipy.stats distribution</td>
</tr>
<tr>
<td>CLogLog</td>
<td>The complementary log-log transform</td>
</tr>
<tr>
<td>Log</td>
<td>The log transform ..</td>
</tr>
<tr>
<td>Logit</td>
<td>The logit transform ..</td>
</tr>
<tr>
<td>NegativeBinomial([alpha])</td>
<td>The negative binomial link function</td>
</tr>
<tr>
<td>Power([power])</td>
<td>The power transform</td>
</tr>
<tr>
<td>cauchy()</td>
<td>The Cauchy (standard Cauchy CDF) transform</td>
</tr>
<tr>
<td>cloglog</td>
<td>The CLogLog transform link function.</td>
</tr>
<tr>
<td>identity()</td>
<td>The identity transform ..</td>
</tr>
<tr>
<td>inverse_power()</td>
<td>The inverse transform ..</td>
</tr>
<tr>
<td>inverse_squared()</td>
<td>The inverse squared transform ..</td>
</tr>
<tr>
<td>log</td>
<td>The log transform ..</td>
</tr>
<tr>
<td>logit</td>
<td>The logit transform ..</td>
</tr>
<tr>
<td>nbinom([alpha])</td>
<td>The negative binomial link function.</td>
</tr>
<tr>
<td>probit([dbn])</td>
<td>The probit (standard normal CDF) transform</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.links.Link

class statsmodels.genmod.families.links.Link
A generic link function for one-parameter exponential family.

Link does nothing, but lays out the methods expected of any subclass.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of the link function g'(p).</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of the link function.</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse link function g^(-1)(z).</td>
</tr>
</tbody>
</table>
statsmodels.genmod.families.links.Link.deriv

\texttt{Link.deriv}(p)

Derivative of the link function \( g'(p) \). Just a placeholder.

\textbf{Parameters} \( p \) : array-like

\textbf{Returns} The value of the derivative of the link function \( g'(p) \).

statsmodels.genmod.families.links.Link.inverse

\texttt{Link.inverse}(z)

Inverse of the link function. Just a placeholder.

\textbf{Parameters} \( z \) : array-like

\( z \) is usually the linear predictor of the transformed variable in the IRLS algorithm for GLM.

\textbf{Returns} The value of the inverse of the link function \( g^{-1}(z) = p \).

statsmodels.genmod.families.links.Link.inverse_deriv

\texttt{Link.inverse_deriv}(z)

Derivative of the inverse link function \( g^{-1}(z) \).

\textbf{Parameters} \( z \) : array-like

\( z \) is usually the linear predictor for a GLM or GEE model.

\textbf{Returns} The value of the derivative of the inverse of the link function.

\textbf{Notes}

This reference implementation gives the correct result but it inefficient, so it can be overridden in subclasses.

statsmodels.genmod.families.links.CDFLink

class \texttt{statsmodels.genmod.families.links.CDFLink} (\texttt{dbn=\textless \texttt{scipy.stats.distributions.norm_gen}}

\texttt{object at 0x0534E8D0})

The use the CDF of a scipy.stats distribution

CDFLink is a subclass of logit in order to use its \_\texttt{clean} method for the link and its derivative.

\textbf{Parameters} \( \texttt{dbn} \) : scipy.stats distribution

Default is \( \texttt{dbn=\texttt{scipy.stats.norm}} \)

\textbf{Notes}

The CDF link is untested.

\textbf{Methods}

\begin{itemize}
  \item \texttt{deriv}(p) Derivative of CDF link
  \item \texttt{inverse}(z) The inverse of the CDF link
\end{itemize}

Continued on next page
Table 3.33 – continued from previous page

<table>
<thead>
<tr>
<th>inverse_deriv(z)</th>
<th>Derivative of the inverse of the CDF transformation link function</th>
</tr>
</thead>
</table>

statsmodels.genmod.families.links.CDFLink.deriv

CDFLink.deriv(p)

Derivative of CDF link

Parameters p : array-like

mean parameters

Returns g'(p) : array

The derivative of CDF transform at p

Notes

g'(p) = 1 / dbn.pdf(dbn.ppf(p))

statsmodels.genmod.families.links.CDFLink.inverse

CDFLink.inverse(z)

The inverse of the CDF link

Parameters z : array-like

The value of the inverse of the link function at p

Returns p : array

Mean probabilities. The value of the inverse of CDF link of z

Notes

g^(-1)(z) = dbn.cdf(z)

statsmodels.genmod.families.links.CDFLink.inverse_deriv

CDFLink.inverse_deriv(z)

Derivative of the inverse of the CDF transformation link function

Parameters z : array

The inverse of the link function at p

Returns The value of the derivative of the inverse of the logit function :

statsmodels.genmod.families.links.CLogLog

class statsmodels.genmod.families.links.CLogLog

The complementary log-log transform

CLogLog inherits from Logit in order to have access to its _clean method for the link and its derivative.

Notes

CLogLog is untested.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of C-Log-Log transform link function</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of C-Log-Log transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the C-Log-Log transform link function</td>
</tr>
</tbody>
</table>

\[ g'(p) = \frac{-1}{\log(p) \cdot p} \]

\[ g^\prime(z) = 1 - \exp(-\exp(z)) \]

\[ g^\prime(p) = \frac{-1}{\log(p) \cdot p} \]

**statsmodels.genmod.families.links.CLogLog.deriv**

CLogLog.deriv(p)

Derivative of C-Log-Log transform link function

**Parameters**

- p : array-like
  
  Mean parameters

**Returns**

- g'(p) : array
  
  The derivative of the CLogLog transform link function

**Notes**

\[ g'(p) = \frac{-1}{\log(p) \cdot p} \]

**statsmodels.genmod.families.links.CLogLog.inverse**

CLogLog.inverse(z)

Inverse of C-Log-Log transform link function

**Parameters**

- z : array-like
  
  The value of the inverse of the CLogLog link function at p

**Returns**

- p : array
  
  Mean parameters

**Notes**

\[ g^\prime(-1)(z) = 1 - \exp(-\exp(z)) \]

**statsmodels.genmod.families.links.CLogLog.inverse_deriv**

CLogLog.inverse_deriv(z)

Derivative of the inverse of the C-Log-Log transform link function

**Parameters**

- z : array-like
  
  The value of the inverse of the CLogLog link function at p

**Returns**

The derivative of the inverse of the CLogLog link function:

**statsmodels.genmod.families.links.Log**

class statsmodels.genmod.families.links.Log

The log transform
Notes

call and derivative call a private method _clean to trim the data by machine epsilon so that p is in (0,1). log is an alias of Log.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of log transform link function</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of log transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the log transform link function</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.links.Log.deriv
Log.deriv(p)
Derivative of log transform link function

Parameters p : array-like
Mean parameters

Returns g'(p) : array
derivative of log transform of x

Notes

g(x) = 1/x

statsmodels.genmod.families.links.Log.inverse
Log.inverse(z)
Inverse of log transform link function

Parameters z : array
The inverse of the link function at p

Returns p : array
The mean probabilities given the value of the inverse z

Notes

g^{-1}(z) = \exp(z)

statsmodels.genmod.families.links.Log.inverse_deriv
Log.inverse_deriv(z)
Derivative of the inverse of the log transform link function

Parameters z : array
The inverse of the link function at p

Returns The value of the derivative of the inverse of the logit function :
statsmodels.genmod.families.links.Logit

class statsmodels.genmod.families.links.Logit
   The logit transform

   Notes

   call and derivative use a private method _clean to make trim p by machine epsilon so that p is in (0,1)
   Alias of Logit: logit = Logit()

   Methods

   __deriv__(p)  Derivative of the logit transform
   __inverse__(z)  Inverse of the logit transform
   __inverse_deriv__(z)  Derivative of the inverse of the logit transform

statsmodels.genmod.families.links.Logit.deriv
Logit.__deriv__(p)  Derivative of the logit transform

   Parameters p: array-like :
   Probabilities

   Returns g'(p): array
   Value of the derivative of logit transform at p

   Notes

   g'(p) = 1 / (p * (1 - p))
   Alias for Logit: logit = Logit()

statsmodels.genmod.families.links.Logit.inverse
Logit.__inverse__(z)  Inverse of the logit transform

   Parameters z: array-like
   The value of the logit transform at p

   Returns p: array
   Probabilities

   Notes

   g^(-1)(z) = exp(z)/(1+exp(z))
Logit inverse derivative

\[ \text{Inverse derivative of the logit transform} \]

**Parameters**
- **z**: array-like

  *z* is usually the linear predictor for a GLM or GEE model.

**Returns**
- The value of the derivative of the inverse of the logit function:

Negative Binomial link function

**Parameters**
- **alpha**: float, optional

  Alpha is the ancillary parameter of the Negative Binomial link function. It is assumed to be nonstochastic. The default value is 1. Permissible values are usually assumed to be in (.01,2).

**Methods**

- **deriv(p)**: Derivative of the negative binomial transform
- **inverse(z)**: Inverse of the negative binomial transform
- **inverse_deriv(z)**: Derivative of the inverse of the negative binomial transform

Negative Binomial derivative

**Parameters**
- **p**: array-like

  Mean parameters

**Returns**
- **g'(p)**: array

  The derivative of the negative binomial transform link function

**Notes**

\[ g'(x) = \frac{1}{x+\alpha x^2} \]

Negative Binomial inverse

**Parameters**
- **z**: array-like

  The value of the inverse of the negative binomial link at *p*.

**Returns**
- **p**: array
Mean parameters

Notes

\[
g^{-1}(z) = \frac{\exp(z)}{\alpha(1-\exp(z))}
\]

`statsmodels.genmod.families.links.NegativeBinomial.inverse_deriv`

`NegativeBinomial.inverse_deriv(z)`
Derivative of the inverse of the negative binomial transform

**Parameters**
- `z`: array-like
  Usually the linear predictor for a GLM or GEE model

**Returns**
The value of the inverse of the derivative of the negative binomial link:

`statsmodels.genmod.families.links.Power`

**class**
`statsmodels.genmod.families.links.Power(power=1.0)`
The power transform

**Parameters**
- `power`: float
  The exponent of the power transform

Notes

Aliases of Power:
- `inverse = Power(power=-1)`
- `sqrt = Power(power=.5)`
- `inverse_squared = Power(power=-2.)`
- `identity = Power(power=1.)`

Methods

- `deriv(p)`
  Derivative of the power transform

- `inverse(z)`
  Inverse of the power transform link function

- `inverse_deriv(z)`
  Derivative of the inverse of the power transform

`statsmodels.genmod.families.links.Power.deriv`

`Power.deriv(p)`
Derivative of the power transform

**Parameters**
- `p`: array-like
  Mean parameters

**Returns**
- `g'(p)`: array
  Derivative of power transform of `p`

Notes

\[
g'(p) = power * p^{power-1)
\]
**Power**

`Power.inverse(z)`

Inverse of the power transform link function

**Parameters**
- `z`: array-like
  Value of the transformed mean parameters at `p`

**Returns**
- `p`: array
  Mean parameters

**Notes**

\[ g^{-1}(z') = z'^{1/power} \]

**Power.inverse_deriv**

`Power.inverse_deriv(z)`

Derivative of the inverse of the power transform

**Parameters**
- `z`: array-like
  `z` is usually the linear predictor for a GLM or GEE model.

**Returns**
- The value of the derivative of the inverse of the power transform function:

**Cauchy**

`Cauchy` is an alias of CDFLink with `dbn=scipy.stats.cauchy`

**Methods**

- `deriv(p)` Derivative of CDF link
- `inverse(z)` The inverse of the CDF link
- `inverse_deriv(z)` Derivative of the inverse of the CDF transformation link function

**Cauchy.deriv**

`Cauchy.deriv(p)` Derivative of CDF link

**Parameters**
- `p`: array-like
  mean parameters

**Returns**
- `g'(p)`: array
  The derivative of CDF transform at `p`
Notes

\[ g'(p) = \frac{1}{dbn.pdf(dbn.ppf(p))} \]

\textbf{statsmodels.genmod.families.links.cauchy.inverse}

\texttt{cauchy.inverse(z)}

The inverse of the CDF link

\textbf{Parameters} \( z \) : array-like

The value of the inverse of the link function at \( p \)

\textbf{Returns} \( p \) : array

Mean probabilities. The value of the inverse of CDF link of \( z \)

Notes

\[ g^{-1}(z) = dbn.cdf(z) \]

\textbf{statsmodels.genmod.families.links.cauchy.inverse_deriv}

\texttt{cauchy.inverse_deriv(z)}

Derivative of the inverse of the CDF transformation link function

\textbf{Parameters} \( z \) : array

The inverse of the link function at \( p \)

\textbf{Returns} The value of the derivative of the inverse of the logit function:

\textbf{statsmodels.genmod.families.links.CLogLog}

\textbf{class} \texttt{statsmodels.genmod.families.links.CLogLog}

The complementary log-log transform

CLogLog inherits from Logit in order to have access to its _clean method for the link and its derivative.

Notes

CLogLog is untested.

Methods

\begin{verbatim}
| deriv(p)         | Derivative of C-Log-Log transform link function |
| inverse(z)      | Inverse of C-Log-Log transform link function   |
| inverse_deriv(z)| Derivative of the inverse of the C-Log-Log transform link function |
\end{verbatim}

\textbf{statsmodels.genmod.families.links.CLogLog.deriv}

\texttt{CLogLog.deriv(p)}

Derivative of C-Log-Log transform link function

\textbf{Parameters} \( p \) : array-like
Mean parameters

Returns $g'(p)$: array

The derivative of the CLogLog transform link function

Notes

$g'(p) = -1 / (\log(p) * p)$

statsmodels.genmod.families.links.CLogLog.inverse

CLogLog.inverse($z$)

Inverse of C-Log-Log transform link function

Parameters $z$: array-like

The value of the inverse of the CLogLog link function at $p$

Returns $p$: array

Mean parameters

Notes

$g^\leftarrow(1)(z) = 1-\exp(-\exp(z))$

statsmodels.genmod.families.links.CLogLog.inverse_deriv

CLogLog.inverse_deriv($z$)

Derivative of the inverse of the C-Log-Log transform link function

Parameters $z$: array-like

The value of the inverse of the CLogLog link function at $p$

Returns The derivative of the inverse of the CLogLog link function:

statsmodels.genmod.families.links.identity

class statsmodels.genmod.families.links.identity

The identity transform

Notes

$g(p) = p$

Alias of statsmodels.family.links.Power(power=1.)

Methods

| deriv(p) | Derivative of the power transform |
| inverse(z) | Inverse of the power transform link function |
| inverse_deriv(z) | Derivative of the inverse of the power transform |
statsmodels.genmod.families.links.identity.deriv
identity.deriv(p)
Derivative of the power transform

Parameters p : array-like
Mean parameters

Returns g'(p) : array
Derivative of power transform of p

Notes

g'(p) = power * p**('power - 1)

statsmodels.genmod.families.links.identity.inverse
identity.inverse(z)
Inverse of the power transform link function

Parameters ‘z‘ : array-like
Value of the transformed mean parameters at p

Returns ‘p‘ : array
Mean parameters

Notes

g'^(-1)(z') = z**('1/power)

statsmodels.genmod.families.links.identity.inverse_deriv
identity.inverse_deriv(z)
Derivative of the inverse of the power transform

Parameters z : array-like
z is usually the linear predictor for a GLM or GEE model.

Returns The value of the derivative of the inverse of the power transform function :

statsmodels.genmod.families.links.inverse_power

class statsmodels.genmod.families.links.inverse_power
The inverse transform

Notes

g(p) = 1/p
Alias of statsmodels.family.links.Power(power=-1.)
Methods
Derivative of the power transform

\[ g'(p) = \text{power} \times p'^{('\text{power} - 1)} \]

Inverse of the power transform link function

\[ g^{-1}(z') = z'^{1/\text{power}} \]

Derivative of the inverse of the power transform

\[ \text{z is usually the linear predictor for a GLM or GEE model.} \]

The inverse squared transform
### Notes

\[ g(p) = 1/(p^{**2}) \]

Alias of `statsmodels.family.links.Power(power=2.)`

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of the power transform</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of the power transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the power transform</td>
</tr>
</tbody>
</table>

#### `statsmodels.genmod.families.links.inverse_squared.deriv`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverse_squared.deriv(p)</td>
<td>Derivative of the power transform</td>
</tr>
</tbody>
</table>

**Parameters**

- `p` : array-like
  - Mean parameters

**Returns**

- `g'(p)` : array
  - Derivative of power transform of `p`

#### Notes

\[ g'(p) = power * p^{*(power - 1)} \]

#### `statsmodels.genmod.families.links.inverse_squared.inverse`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverse_squared.inverse(z)</td>
<td>Inverse of the power transform link function</td>
</tr>
</tbody>
</table>

**Parameters**

- `z` : array-like
  - Value of the transformed mean parameters at `p`

**Returns**

- `p` : array
  - Mean parameters

#### Notes

\[ g^\wedge(-1)(z') = z'^{*(1/power)} \]

#### `statsmodels.genmod.families.links.inverse_squared.inverse_deriv`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverse_squared.inverse_deriv(z)</td>
<td>Derivative of the inverse of the power transform</td>
</tr>
</tbody>
</table>

**Parameters**

- `z` : array-like
  - `z` is usually the linear predictor for a GLM or GEE model.

**Returns**

- The value of the derivative of the inverse of the power transform function:
class statsmodels.genmod.families.links.Log

The log transform

Notes

call and derivative call a private method _clean to trim the data by machine epsilon so that p is in (0,1). log is an alias of Log.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of log transform link function</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of log transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the log transform link function</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.links.Log.deriv

Log.deriv(p)

Derivative of log transform link function

Parameters:
- p : array-like
  Mean parameters

Returns:
- g'(p) : array
g' is the derivative of log transform of x

Notes

g(x) = 1/x

statsmodels.genmod.families.links.Log.inverse

Log.inverse(z)

Inverse of log transform link function

Parameters:
- z : array
  The inverse of the link function at p

Returns:
- p : array
  The mean probabilities given the value of the inverse z

Notes

g^{-1}(z) = \exp(z)

statsmodels.genmod.families.links.Log.inverse_deriv

Log.inverse_deriv(z)

Derivative of the inverse of the log transform link function
Parameters  z : array

The inverse of the link function at \( p \)

Returns  The value of the derivative of the inverse of the logit function :

\[
\text{statsmodels.genmod.families.links.Logit}
\]

class \text{statsmodels.genmod.families.links.Logit}

The logit transform

\textbf{Notes}

call and derivative use a private method \_clean to make trim \( p \) by machine epsilon so that \( p \) is in (0,1)

Alias of Logit: logit = Logit()

\textbf{Methods}

\begin{tabular}{ll}
\texttt{deriv(p)} & Derivative of the logit transform \\
\texttt{inverse(z)} & Inverse of the logit transform \\
\texttt{inverse\_deriv(z)} & Derivative of the inverse of the logit transform \\
\end{tabular}

\text{statsmodels.genmod.families.links.Logit.deriv}

\texttt{Logit.deriv(p)}

Derivative of the logit transform

Parameters  \( p \) : array-like :

Probabilities

Returns  \( g'(p) \) : array

Value of the derivative of logit transform at \( p \)

\textbf{Notes}

\[ g'(p) = 1 / (p \times (1 - p)) \]

Alias for \texttt{Logit}: logit = Logit()

\text{statsmodels.genmod.families.links.Logit.inverse}

\texttt{Logit.inverse(z)}

Inverse of the logit transform

Parameters  \( z \) : array-like

The value of the logit transform at \( p \)

Returns  \( p \) : array

Probabilities
Notes

g\(^{-1}\)(z) = \exp(z)/(1+\exp(z))

```
statsmodels.genmod.families.links.Logit.inverse_deriv

Logit.inverse_deriv(z)
  Derivative of the inverse of the logit transform

  Parameters  z : array-like
    z is usually the linear predictor for a GLM or GEE model.

  Returns  The value of the derivative of the inverse of the logit function :
```

```
statsmodels.genmod.families.links.nbinom

class statsmodels.genmod.families.links.nbinom(alpha=1.0)
  The negative binomial link function.

Notes

g(p) = \log(p/(p + 1/alpha))
nbinom is an alias of NegativeBinomial. nbinom = NegativeBinomial(alpha=1.)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of the negative binomial transform</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of the negative binomial transform</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the negative binomial transform</td>
</tr>
</tbody>
</table>

```
statsmodels.genmod.families.links.nbinom.deriv

nbinom.deriv(p)
  Derivative of the negative binomial transform

  Parameters  p : array-like
    Mean parameters

  Returns  g'(p) : array
    The derivative of the negative binomial transform link function

Notes

g'(x) = 1/(x+alpha*x^2)

```
statsmodels.genmod.families.links.nbinom.inverse

nbinom.inverse(z)
  Inverse of the negative binomial transform

  Parameters  z : array-like
The value of the inverse of the negative binomial link at \( p \).

**Returns**:

- \( p \): array
  Mean parameters

**Notes**

\[ g^{-1}(z) = \frac{\exp(z)}{(\alpha \cdot (1 - \exp(z)))} \]

```python
statsmodels.genmod.families.links.nbinom.inverse_deriv
nbinom.inverse_deriv(z)
```
Derivative of the inverse of the negative binomial transform

**Parameters**

- \( z \) : array-like
  Usually the linear predictor for a GLM or GEE model

**Returns**

The value of the inverse of the derivative of the negative binomial link:

```python
statsmodels.genmod.families.links.probit
```

**class**

```python
class statsmodels.genmod.families.links.probit (dbn=<scipy.stats.distributions.norm_gen
go\object at 0x0534E8D0>)
```
The probit (standard normal CDF) transform

**Notes**

\[ g(p) = \text{scipy.stats.norm.ppf}(p) \]

probit is an alias of CDFLink.

**Methods**

- `deriv(p)` : Derivative of CDF link
- `inverse(z)` : The inverse of the CDF link
- `inverse_deriv(z)` : Derivative of the inverse of the CDF transformation link function

```python
statsmodels.genmod.families.links.probit.deriv
probit.deriv(p)
```
Derivative of CDF link

**Parameters**

- \( p \) : array-like
  mean parameters

**Returns**

- \( g'(p) \) : array
  The derivative of CDF transform at \( p \)
Notes

\[ g'(p) = \frac{1}{dbn.pdf(dbn.ppf(p))} \]

\texttt{statsmodels.genmod.families.links.probit.inverse}
\texttt{probit.inverse(z)}

The inverse of the CDF link

**Parameters**

- \( z \) : array-like
  
  The value of the inverse of the link function at \( p \)

**Returns**

- \( p \) : array
  
  Mean probabilities. The value of the inverse of CDF link of \( z \)

Notes

\[ g^\Gamma(-1)(z) = dbn.cdf(z) \]

\texttt{statsmodels.genmod.families.links.probit.inverse_deriv}
\texttt{probit.inverse_deriv(z)}

Derivative of the inverse of the CDF transformation link function

**Parameters**

- \( z \) : array
  
  The inverse of the link function at \( p \)

**Returns**

- The value of the derivative of the inverse of the logit function:

3.3 Generalized Estimating Equations

Generalized Estimating Equations estimate generalized linear models for panel, cluster or repeated measures data when the observations are possibly correlated withing a cluster but uncorrelated across clusters. It supports estimation of the same one-parameter exponential families as Generalized Linear models \((GLM)\).

See Module Reference for commands and arguments.

3.3.1 Examples

```python
# TODO update for GEE
# Load modules and data
import statsmodels.api as sm
data = sm.datasets.scotland.load()
data.exog = sm.add_constant(data.exog)

# Instantiate a gamma family model with the default link function.
gamma_model = sm.GLM(data.endog, data.exog, family=sm.families.Gamma())
gamma_results = gamma_model.fit()
```

Detailed examples can be found here:
References


3.3.2 Module Reference

Model Class

```python
generate_estimating_equations.GEE
class generate_estimating_equations.GEE (
    endog, exog, groups[, time, family, ...]) Generalized Estimating Equations Models
```

statsmodels.genmod.generalized_estimating_equations.GEE

class statsmodels.genmod.generalized_estimating_equations.GEE (endog, exog, groups, time=None, family=None, cov_struct=None, missing='none', offset=None, dep_data=None, constraint=None)

Generalized Estimating Equations Models

GEE estimates Generalized Linear Models when the data have a grouped structure, and the observations are possibly correlated within groups but not between groups.

**Parameters**

- **endog**: array-like
  - 1d array of endogenous response values.
- **exog**: array-like
  - A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant`.
- **groups**: array-like
  - A 1d array of length nobs containing the group labels.
- **time**: array-like
  - A 2d array of time (or other index) values, used by some dependence structures to define similarity relationships among observations within a cluster.
- **family**: family class instance
The default is Gaussian. To specify the binomial distribution family = sm.family.Binomial(). Each family can take a link instance as an argument. See statsmodels.family.family for more information.

cov_struct : CovStruct class instance

The default is Independence. To specify an exchangeable structure use cov_struct = Exchangeable(). See statsmodels.genmod.dependence_structures.CovStruct for more information.

offset : array-like

An offset to be included in the fit. If provided, must be an array whose length is the number of rows in exog.

dep_data : array-like

Additional data passed to the dependence structure.

constraint : (ndarray, ndarray)

If provided, the constraint is a tuple (L, R) such that the model parameters are estimated under the constraint L * param = R, where L is a q x p matrix and R is a q-dimensional vector. If constraint is provided, a score test is performed to compare the constrained model to the unconstrained model.

missing : str

Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’

See also:
statsmodels.families.

Notes

Only the following combinations make sense for family and link

+ ident log logit probit cloglog pow opow nbinom loglog logc

Gaussian | x  x    x
inv Gaussian | x  x   x
binomial | x  x  x  x  x  x  x  x  x
Poisson | x  x   x
neg binomial | x  x    x
gamma | x  x   x

Not all of these link functions are currently available.

Endog and exog are references so that if the data they refer to are already arrays and these arrays are changed, endog and exog will change.

The “robust” covariance type is the standard “sandwich estimator” (e.g. Liang and Zeger (1986)). It is the default here and in most other packages. The “naive” estimator gives smaller standard errors, but is only correct if the working correlation structure is correctly specified. The “bias reduced” estimator of Mancl and DeRouen (Biometrics, 2001) reduces the downward bias of the robust estimator.

Methods
**estimate_scale()**
Returns an estimate of the scale parameter \( \phi \) at the current parameter value.

**fit([maxiter, ctol, start_params, ...])**
Fits a GEE model.

**from_formula(formula, data[, subset])**
Create a Model from a formula and dataframe.

**predict(params[, exog, offset, linear])**
Return predicted values for a design matrix.

**update_cached_means(mean_params)**
Cached_means should always contain the most recent calculation.

---

**statsmodels.genmod.generalized_estimating_equations.GEE.estimate_scale**

**GEE.estimate_scale()**
Returns an estimate of the scale parameter \( \phi \) at the current parameter value.

**statsmodels.genmod.generalized_estimating_equations.GEE.fit**

**GEE.fit(maxiter=60, ctol=1e-06, start_params=None, params_niter=1, first_dep_update=0, covariance_type='robust')**
Fits a GEE model.

**Parameters**
- **maxiter**: integer
  - The maximum number of iterations

- **ctol**: float
  - The convergence criterion for stopping the Gauss-Seidel iterations

- **start_params**: array-like
  - A vector of starting values for the regression coefficients. If None, a default is chosen.

- **params_niter**: integer
  - The number of Gauss-Seidel updates of the mean structure parameters that take place prior to each update of the dependence structure.

- **first_dep_update**: integer
  - No dependence structure updates occur before this iteration number.

- **covariance_type**: string
  - One of “robust”, “naive”, or “bias_reduced”.

**Returns**
An instance of the GEEResults class:

**Notes**
If convergence difficulties occur, increase the values of **first_dep_update** and/or **params_niter**. Setting **first_dep_update** to a greater value (e.g. ~10-20) causes the algorithm to move close to the GLM solution before attempting to identify the dependence structure.

For the Gaussian family, there is no benefit to setting **params_niter** to a value greater than 1, since the mean structure parameters converge in one step.

**statsmodels.genmod.generalized_estimating_equations.GEE.from_formula**

**classmethod GEE.from_formula(formula, data, subset=None, *args, **kwargs)**
Create a Model from a formula and dataframe.

**Parameters**
- **formula**: str or generic Formula object
  - The formula specifying the model
**data**: array-like

The data for the model. See Notes.

**subset**: array-like

An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

**args**: extra arguments

These are passed to the model

**kwargs**: extra keyword arguments

These are passed to the model.

**Returns**

**model**: Model instance

**Notes**

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

---

**statsmodels.genmod.generalized_estimating_equations.GEE.predict**

GEE.predict (params, exog=None, offset=None, linear=False)

Return predicted values for a design matrix

**Parameters**

**params**: array-like

Parameters / coefficients of a GLM.

**exog**: array-like, optional

Design / exogenous data. If exog is None, model exog is used.

**offset**: array-like, optional

Offset for exog if provided. If offset is None, model offset is used.

**linear**: bool

If True, returns the linear predicted values. If False, returns the value of the inverse of the model’s link function at the linear predicted values.

**Returns**

An array of fitted values:

---

**statsmodels.genmod.generalized_estimating_equations.GEE.update_cached_means**

GEE.update_cached_means (mean_params)

cached_means should always contain the most recent calculation of the group-wise mean vectors. This function should be called every time the regression parameters are changed, to keep the cached means up to date.

**Attributes**

<table>
<thead>
<tr>
<th>cached_means</th>
</tr>
</thead>
<tbody>
<tr>
<td>endog_names</td>
</tr>
<tr>
<td>exog_names</td>
</tr>
</tbody>
</table>

Continued on next page
Results Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEEResults(model, params, cov_params, scale)</td>
<td>Class to contain GEE results. GEEResults inherits from statsmodels.LikelihoodModelResults</td>
</tr>
<tr>
<td>GEEMargins(results, args[, kwargs])</td>
<td>Estimate the marginal effects of a model fit using generalized estimating equations</td>
</tr>
</tbody>
</table>

**class statsmodels.genmod.generalized_estimating_equations.GEEResults**

Class to contain GEE results.

GEEResults inherits from statsmodels.LikelihoodModelResults

**Parameters** See statsmodels.LikelihoodModelReesults:

**Returns** **Attributes**:

- **naive_covariance**: ndarray
  - covariance of the parameter estimates that is not robust to correlation or variance mis-specification

- **robust_covariance_bc**: ndarray
  - covariance of the parameter estimates that is robust and bias reduced

- **converged**: bool
  - indicator for convergence of the optimization. True if the norm of the score is smaller than a threshold

- **covariance_type**: string
  - string indicating whether a “robust”, “naive” or “bias_reduced” covariance is used as default

- **fit_history**: dict
  - Contains information about the iterations.

- **fittedvalues**: array
  - Linear predicted values for the fitted model. dot(exog, params)

- **model**: class instance
  - Pointer to GEE model instance that called fit.

- **normalized_cov_params**: array
  - See GEE docstring

- **params**: array
  - The coefficients of the fitted model. Note that interpretation of the coefficients often depends on the distribution family and the data.

- **scale**: float

3.3. Generalized Estimating Equations
The estimate of the scale / dispersion for the model fit. See GEE.fit for more information.

**score_norm** : float

norm of the score at the end of the iterative estimation.

**bse** : array

The standard errors of the fitted GEE parameters.

**See also:**

statsmodels.LikelihoodModelResults, GEE

**Methods**

-bse()

-centered_resid()

-conf_int([alpha, cols, covariance_type])

-cov_params([r_matrix, column, scale, cov_p, ...])

-f_test([r_matrix, q_matrix, cov_p, scale, ...])

-fittedvalues()

-initialize(model, params, **kwd)

-load(fname)

-normalized_cov_params()

-predict([exog, transform])

-pvalues()

-remove_data()

-resid()

-save(fname, remove_data)

-standart_errors([covariance_type])

-summary([xname, title, alpha, ...])

-t_test([r_matrix, q_matrix, cov_p, scale, use_t])

-tvalues()

-wald_test([r_matrix, q_matrix, cov_p, ...])

---

statsmodels.genmod.generalized_estimating_equations.GEEResults.bse
statsmodels.genmod.generalized_estimating_equations.GEEResults.bse()

statsmodels.genmod.generalized_estimating_equations.GEEResults.centered_resid
statsmodels.genmod.generalized_estimating_equations.GEEResults.centered_resid()

Returns the residuals centered within each group.

statsmodels.genmod.generalized_estimating_equations.GEEResults.conf_int
GEEResults.conf_int(alpha=0.05, cols=None, covariance_type='robust')

Returns confidence intervals for the fitted parameters.

---

Parameters

**alpha** : float, optional

The alpha level for the confidence interval. i.e., The default alpha = .05 returns a 95% confidence interval.

**cols** : array-like, optional
cols specifies which confidence intervals to return.

covariance_type : string

The covariance type used for computing standard errors; must be one of ‘robust’, ‘naive’, and ‘bias reduced’. See GEE for details.

Notes

The confidence interval is based on the Gaussian distribution.

statsmodels.genmod.generalized_estimating_equations.GEEResults.cov_params

GEEResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters r_matrix : array-like

Can be 1d, or 2d. Can be used alone or with other.

column : array-like, optional

Must be used on its own. Can be 0d or 1d see below.

scale : float, optional

Can be specified or not. Default is None, which means that the scale argument is taken from the model.

other : array-like, optional

Can be used when r_matrix is specified.

Returns (The below are assumed to be in matrix notation.) :

cov : ndarray

If no argument is specified returns the covariance matrix of a model :

(scale)*(X.T X)^(-1)

If contrast is specified it pre and post-multiplies as follows :

(scale) * r_matrix (X.T X)^(-1) r_matrix.T

If contrast and other are specified returns :

(scale) * r_matrix (X.T X)^(-1) other.T

If column is specified returns :

(scale) * (X.T X)^(-1)[column,column] if column is 0d :

OR :

(scale) * (X.T X)^(-1)[column][:,column] if column is 1d :
statsmodels.genmod.generalized_estimating_equations.GEEResults.f_test

GEEResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)

Compute the F-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of
    regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
  This is deprecated. See r_matrix and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array,
  q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given,
  self.normalized_cov_params is used.

scale : float, optional
  Default is 1.0 for no scaling.

invcov : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts.statsmodels.model.LikelihoodModelResults.wald_test
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,

r_matrix (pX pX.T) r_matrix.T

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be
problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]

This tests that each coefficient is jointly statistically significantly different from zero.

>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
 df_denom=9, df_num=6>
```

262 Chapter 3. Table of Contents
Compare this to

```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10

>>> B = np.array(([0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]))
```

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the
5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
 df_num=2>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols

>>> dta = longley.load_pandas().data

>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'

>>> results = ols(formula, dta).fit()

>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'

>>> f_test = results.f_test(hypotheses)

>>> print(f_test)
```

### 3.3. Generalized Estimating Equations

#### static GEEResults.fittedvalues() Returns the fitted values from the model.

#### GEEResults.initialize(model, params, **kwd)

#### GEEResults.llf()

#### classmethod GEEResults.load(fname) load a pickle, (class method)

- Parameters **fname**: string or filehandle
- Parameters **fname** can be a string to a file path or filename, or a filehandle.

- Returns **unpickled instance**:

#### GEEResults.normalized_cov_params()

#### GEEResults.predict(exog=None, transform=True, *args, **kwargs) Call self.model.predict with self.params as the first argument.

- Parameters **exog**: array-like, optional
The values for which you want to predict.

**transform**: bool, optional

If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model $y \sim \log(x1) + \log(x2)$, and transform is True, then you can pass a data structure that contains $x1$ and $x2$ in their original form. Otherwise, you’d need to log the data first.

**Returns** See `self.model.predict`:

```python
statsmodels.genmod.generalized_estimating_equations.GEEResults.pvalues
```

```python
static GEEResults.pvalues()
```

```python
statsmodels.genmod.generalized_estimating_equations.GEEResults.remove_data
```

```python
GEEResults.remove_data()
```

remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning**: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

```python
statsmodels.genmod.generalized_estimating_equations.GEEResults.resid
```

```python
static GEEResults.resid()
```

Returns the residuals, the endogeneous data minus the fitted values from the model.

```python
statsmodels.genmod.generalized_estimating_equations.GEEResults.save
```

```python
GEEResults.save(fname, remove_data=False)
```

dsaves a pickle of this instance

**Parameters**

- `fname` : string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- `remove_data` : bool
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.
statsmodels.genmod.generalized_estimating_equations.GEEResults.standard_errors
GEEResults.standard_errors(covariance_type='robust')
This is a convenience function that returns the standard errors for any covariance type. The value of \( bse \) is the standard errors for whichever covariance type is specified as an argument to \( fit \) (defaults to “robust”).

statsmodels.genmod.generalized_estimating_equations.GEEResults.summary
GEEResults.summary(yname=None, xname=None, title=None, alpha=0.05, covariance_type='robust')
Summarize the Regression Results

Parameters
yname : string, optional
Default is \( y \)
xname : list of strings, optional
Default is \( var_{##} \) for \( ## \) in \( p \) the number of regressors
title : string, optional
Title for the top table. If not None, then this replaces the default title
alpha : float
significance level for the confidence intervals
covariance_type : string
The covariance type used to compute the standard errors; one of ‘robust’ (the usual robust sandwich-type covariance estimate), ‘naive’ (ignores dependence), and ‘bias reduced’ (the Mancl/DeRouen estimate).

Returns
smry : Summary instance
this holds the summary tables and text, which can be printed or converted to various output formats.

See also:
statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.genmod.generalized_estimating_equations.GEEResults.t_test
GEEResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)
Compute a t-test for a joint linear hypothesis of the form \( Rb = q \)

Parameters
r_matrix : array-like, str, tuple
- array : If an array is given, a \( p \times k \) 2d array or length \( k \) 1d array specifying the linear restrictions.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form \((R, q)\), since \( q \_matrix \) is deprecated.
q_matrix : array-like or scalar, optional
This is deprecated. See \( r \_matrix \) and the examples for more information on new usage. Can be either a scalar or a length \( p \) row vector. If omitted and \( r \_matrix \) is an array, \( q \_matrix \) is assumed to be a conformable array of zeros.
cov_p : array-like, optional
An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues  individual t statistics

f_test  for F tests

patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]
```

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_Test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

```
statsmodels.genmod.generalized_estimating_equations.GEEResults.tvalues
static GEEResults.tvalues()

Return the t-statistic for a given parameter estimate.

statsmodels.genmod.generalized_estimating_equations.GEEResults.wald_test
GEEResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=False)

Compute a Wald-test for a joint linear hypothesis.
```
Parameters  

**r_matrix** : array-like, str, or tuple

- **array** : An r x k array where r is the number of restrictions to test and k is the number of regressors.
- **str** : The full hypotheses to test can be given as a string. See the examples.
- **tuple** : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

**q_matrix** : array-like

This is deprecated. See *r_matrix* and the examples for more information on new usage.
Can be either a scalar or a length p row vector. If omitted and *r_matrix* is an array,
*q_matrix* is assumed to be a conformable array of zeros.

**cov_p** : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given,
self.normalized_cov_params is used.

**scale** : float, optional

Default is 1.0 for no scaling.

**invcov** : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**use_f** : bool

If True, then the F-distribution is used. If False, then the asymptotic distribution,
chisquare is used. The test statistic is proportionally adjusted for the distribution by
the number of constraints in the hypothesis.

See also:

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix *r_matrix* is assumed to be non-singular. More precisely,

\[ r_{matrix} (pX pX^T) r_{matrix}.T \]

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be
problems in non-OLS models where the rank of the covariance of the noise is not full.

**statsmodels.genmod.generalized_estimating_equations.GEEMargins**

class **statsmodels.genmod.generalized_estimating_equations.GEEMargins** (results,

args,

kwargs=)

Estimate the marginal effects of a model fit using generalized estimating equations.

Parameters  **results** : GEEResults instance

The results instance of a fitted discrete choice model

**args** : tuple

Args are passed to *get_margeff*. This is the same as results.get_margeff. See there for
more information.
**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conf_int([alpha])</code></td>
<td>Returns the confidence intervals of the marginal effects</td>
</tr>
<tr>
<td><code>get_margeff([at, method, atexog, dummy, count])</code></td>
<td>Get marginal effects of the fitted model.</td>
</tr>
<tr>
<td><code>pvalues()</code></td>
<td>Returns a summary table for marginal effects</td>
</tr>
<tr>
<td><code>summary([alpha])</code></td>
<td>Returns a DataFrame summarizing the marginal effects.</td>
</tr>
<tr>
<td><code>summary_frame([alpha])</code></td>
<td></td>
</tr>
<tr>
<td><code>tvalues()</code></td>
<td></td>
</tr>
</tbody>
</table>

**statsmodels.genmod.generalized_estimating_equations.GEEMargins.conf_int**

`GEEMargins.conf_int (alpha=0.05)`

Returns the confidence intervals of the marginal effects

**Parameters**

- **alpha** : float
  - Number between 0 and 1. The confidence intervals have the probability 1-alpha.

**Returns**

- **conf_int** : ndarray
  - An array with lower, upper confidence intervals for the marginal effects.

**statsmodels.genmod.generalized_estimating_equations.GEEMargins.get_margeff**

`GEEMargins.get_margeff (at='overall', method='dydx', atexog=None, dummy=False, count=False)`

Get marginal effects of the fitted model.

**Parameters**

- **at** : str, optional
  - Options are:
    - ‘overall’, The average of the marginal effects at each observation.
    - ‘mean’, The marginal effects at the mean of each regressor.
    - ‘median’, The marginal effects at the median of each regressor.
    - ‘zero’, The marginal effects at zero for each regressor.
    - ‘all’, The marginal effects at each observation. If at is all only margeff will be available.
  - Note that if `exog` is specified, then marginal effects for all variables not specified by `exog` are calculated using the `at` option.

- **method** : str, optional
  - Options are:
    - ‘dydx’ - dy/dx - No transformation is made and marginal effects are returned. This is the default.
    - ‘eyex’ - estimate elasticities of variables in `exog` – d(lny)/d(lnx)
    - ‘dyex’ - estimate semielasticity – dy/d(lnx)
    - ‘eydx’ - estimate semeilasticity – d(lny)/dx
Note that transformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. ‘dyex’ and ‘eyex’ do not make sense for discrete variables.

**atexog** : array-like, optional

Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

**dummy** : bool, optional

If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.

**count** : bool, optional

If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

Returns **effects** : ndarray

the marginal effect corresponding to the input options

**Notes**

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

```python
statsmodels.genmod.generalized_estimating_equations.GEEMargins.pvalues
statsmodels.genmod.generalized_estimating_equations.GEEMargins.summary
_statsmodels.genmod.generalized_estimating_equations.GEEMargins.summary_frame
```

Returns a summary table for marginal effects

Parameters **alpha** : float

Number between 0 and 1. The confidence intervals have the probability 1-alpha.

Returns **Summary** : SummaryTable

A SummaryTable instance

Returns a DataFrame summarizing the marginal effects.

Parameters **alpha** : float

Number between 0 and 1. The confidence intervals have the probability 1-alpha.

Returns **frame** : DataFrames

A DataFrame summarizing the marginal effects.
Dependence Structures

The dependence structures currently implemented are

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CovStruct()</td>
<td>A base class for correlation and covariance structures of grouped data.</td>
</tr>
<tr>
<td>Autoregressive([dist_func])</td>
<td>An autoregressive working dependence structure.</td>
</tr>
<tr>
<td>Exchangeable()</td>
<td>An exchangeable working dependence structure.</td>
</tr>
<tr>
<td>GlobalOddsRatio(endog_type)</td>
<td>Estimate the global odds ratio for a GEE with ordinal or nominal</td>
</tr>
<tr>
<td>Independence()</td>
<td>An independence working dependence structure.</td>
</tr>
<tr>
<td>Nested()</td>
<td>A nested working dependence structure.</td>
</tr>
</tbody>
</table>

Statsmodels Documentation, Release 0.6.0

statsmodels.genmod.dependence_structures.covstruct.CovStruct

class statsmodels.genmod.dependence_structures.covstruct.CovStruct

A base class for correlation and covariance structures of grouped data.

Each implementation of this class takes the residuals from a regression model that has been fitted to grouped data, and uses them to estimate the within-group dependence structure of the random errors in the model.

The state of the covariance structure is represented through the value of the class variable dep_params. The default state of a newly-created instance should correspond to the identity correlation matrix.

Methods

covariance_matrix(endog_expval, index) Returns the working covariance or correlation matrix for a given cluster of data.
initialize(model) Called by GEE, used by implementations that need additional
summary() Returns a text summary of the current estimate of the
update(params) Updates the association parameter values based on the current regression coefficients

statsmodels.genmod.dependence_structures.covstruct.CovStruct.covariance_matrix

CovStruct.covariance_matrix(endog_expval, index)

Returns the working covariance or correlation matrix for a given cluster of data.

Parameters endog_expval: array-like:

The expected values of endog for the cluster for which the covariance or correlation matrix will be returned

index: integer:

The index of the cluster for which the covariance or correlation matrix will be returned

Returns M: matrix:

The covariance or correlation matrix of endog

is_cor: bool:

True if M is a correlation matrix, False if M is a covariance matrix

statsmodels.genmod.dependence_structures.covstruct.CovStruct.initialize
CovStruct.initialize(model)
Called by GEE, used by implementations that need additional setup prior to running fit.

Parameters model: GEE class
A reference to the parent GEE class instance.

CovStruct.summary()
Returns a text summary of the current estimate of the dependence structure.

CovStruct.update(params)
Updates the association parameter values based on the current regression coefficients.

Parameters params: array-like
Working values for the regression parameters.

Attributes

CovStruct.dep_params

Autoregressive
class statsmodels.genmod.dependence_structures.covstruct.Autoregressive(dist_func=None)
An autoregressive working dependence structure.

The dependence is defined in terms of the time component of the parent GEE class. Time represents a potentially multidimensional index from which distances between pairs of observations can be determined. The correlation between two observations in the same cluster is dep_params^distance, where dep_params is the autocorrelation parameter to be estimated, and distance is the distance between the two observations, calculated from their corresponding time values. time is stored as an n_obs x k matrix, where k represents the number of dimensions in the time index.

The autocorrelation parameter is estimated using weighted nonlinear least squares, regressing each value within a cluster on each preceding value in the same cluster.

Parameters dist_func: function from R^k x R^k to R^+, optional:
A function that computes the distance between the two observations based on their time values.

Methods
covariance_matrix(endog_expval, index) Returns the working covariance or correlation matrix for a given cluster of data.
initialize(model) Called by GEE, used by implementations that need additional
summary() Updates the association parameter values based on the current regression coefficients.

Autoregressive.covariance_matrix(endog_expval, index)
Returns the working covariance or correlation matrix for a given cluster of data.

**Parameters**

`endog_expval`: array-like

The expected values of endog for the cluster for which the covariance or correlation matrix will be returned

`index`: integer

The index of the cluster for which the covariance or correlation matrix will be returned

**Returns**

`M`: matrix

The covariance or correlation matrix of endog

`is_cor`: bool

True if M is a correlation matrix, False if M is a covariance matrix

---

**statsmodels.genmod.dependence_structures.covstruct.Autoregressive.initialize**

Autoregressive.initialize(model)

Called by GEE, used by implementations that need additional setup prior to running fit.

**Parameters**

`model`: GEE class

A reference to the parent GEE class instance.

**statsmodels.genmod.dependence_structures.covstruct.Autoregressive.summary**

Autoregressive.summary()

**statsmodels.genmod.dependence_structures.covstruct.Autoregressive.update**

Autoregressive.update(params)

Updates the association parameter values based on the current regression coefficients.

**Parameters**

`params`: array-like

Working values for the regression parameters.

**Attributes**

- Autoregressive.dep_params
- Autoregressive.designx
- Autoregressive.dist_func

---

**statsmodels.genmod.dependence_structures.covstruct.Exchangeable**

class statsmodels.genmod.dependence_structures.covstruct.Exchangeable

An exchangeable working dependence structure.

**Methods**

- covariance_matrix(expval, index)
  Returns the working covariance or correlation matrix for a given cluster of data.

- initialize(model)
  Called by GEE, used by implementations that need additional
Table 3.59 – continued from previous page

summary()
update(params) Updates the association parameter values based on the current regression coefficients.

```python
statsmodels.genmod.dependence_structures.covstruct.Exchangeable.covariance_matrix
Exchangeable.covariance_matrix(expval, index)
Returns the working covariance or correlation matrix for a given cluster of data.

Parameters
endog_expval: array-like :
The expected values of endog for the cluster for which the covariance or correlation
matrix will be returned

index: integer :
The index of the cluster for which the covariance or correlation matrix will be returned

Returns
M: matrix :
The covariance or correlation matrix of endog

is_cor: bool :
True if M is a correlation matrix, False if M is a covariance matrix
```

```python
statsmodels.genmod.dependence_structures.covstruct.Exchangeable.initialize
Exchangeable.initialize(model)
Called by GEE, used by implementations that need additional setup prior to running fit.

Parameters
model: GEE class
A reference to the parent GEE class instance.
```

```python
statsmodels.genmod.dependence_structures.covstruct.Exchangeable.summary
Exchangeable.summary()
```

```python
statsmodels.genmod.dependence_structures.covstruct.Exchangeable.update
Exchangeable.update(params)
Updates the association parameter values based on the current regression coefficients.

Parameters
params: array-like
Working values for the regression parameters.
```

Attributes

```python
Exchangeable.dep_params
```
References


Methods

### statsmodels.genmod.dependence_structures.covstruct.GlobalOddsRatio.covariance_matrix

GlobalOddsRatio.covariance_matrix(expected_value, index)

Returns the working covariance or correlation matrix for a given cluster of data.

**Parameters**

- **endog_expval**: array-like
  - The expected values of endog for the cluster for which the covariance or correlation matrix will be returned

- **index**: integer
  - The index of the cluster for which the covariance or correlation matrix will be returned

**Returns**

- **M**: matrix
  - The covariance or correlation matrix of endog

- **is_cor**: bool
  - True if M is a correlation matrix, False if M is a covariance matrix

### statsmodels.genmod.dependence_structures.covstruct.GlobalOddsRatio.get_eyy

GlobalOddsRatio.get_eyy(endog_expval, index)

Returns a matrix V such that V[i,j] is the joint probability that endog[i] = 1 and endog[j] = 1, based on the marginal probabilities of endog and the odds ratio current_or.

### statsmodels.genmod.dependence_structures.covstruct.GlobalOddsRatio.initialize

GlobalOddsRatio.initialize(model)

### statsmodels.genmod.dependence_structures.covstruct.GlobalOddsRatio.observed_crude_oddsratio

GlobalOddsRatio.observed_crude_oddsratio()

The crude odds ratio is obtained by pooling all data corresponding to a given pair of cut points (c,c'), then forming the inverse variance weighted average of these odds ratios to obtain a single OR. Since the covariate effects are ignored, this OR will generally be greater than the stratified OR.

---

274 Chapter 3. Table of Contents
statsmodels.genmod.dependence_structures.covstruct.GlobalOddsRatio.pooled_odds_ratio

GlobalOddsRatio.pooled_odds_ratio(Tables)

Returns the pooled odds ratio for a list of 2x2 tables.

The pooled odds ratio is the inverse variance weighted average of the sample odds ratios of the tables.

statsmodels.genmod.dependence_structures.covstruct.GlobalOddsRatio.summary

GlobalOddsRatio.summary()

statsmodels.genmod.dependence_structures.covstruct.GlobalOddsRatio.update

GlobalOddsRatio.update(params)

Updates the association parameter values based on the current regression coefficients.

Parameters params : array-like

Working values for the regression parameters.

Attributes

GlobalOddsRatio.cpp
GlobalOddsRatio.crude_or
GlobalOddsRatio.dep_params
GlobalOddsRatio.ibd

statsmodels.genmod.dependence_structures.covstruct.Independence

class statsmodels.genmod.dependence_structures.covstruct.Independence

An independence working dependence structure.

Methods

covariance_matrix(expval, index) Returns the working covariance or correlation matrix for a given cluster of data.
initialize(model) Called by GEE, used by implementations that need additional
summary() Updates the association parameter values based on the current regression coefficients.
update(params)

statsmodels.genmod.dependence_structures.covstruct.Independence.covariance_matrix

Independence.covariance_matrix(expval, index)

Returns the working covariance or correlation matrix for a given cluster of data.

Parameters endog_expval: array-like :

The expected values of endog for the cluster for which the covariance or correlation matrix will be returned

index: integer :

The index of the cluster for which the covariance or correlation matrix will be returned

Returns M: matrix :

The covariance or correlation matrix of endog
is_cor : bool
True if M is a correlation matrix, False if M is a covariance matrix

statsmodels.genmod.dependence_structures.covstruct.Independence.initialize
Independence.initialize(model)
Called by GEE, used by implementations that need additional setup prior to running fit.

Parameters model : GEE class
A reference to the parent GEE class instance.

statsmodels.genmod.dependence_structures.covstruct.Independence.summary
Independence.summary()

statsmodels.genmod.dependence_structures.covstruct.Independence.update
Independence.update(params)
Updates the association parameter values based on the current regression coefficients.

Parameters params : array-like
Working values for the regression parameters.

Attributes

Independence.dep_params

statsmodels.genmod.dependence_structures.covstruct.Nested
class statsmodels.genmod.dependence_structures.covstruct.Nested
A nested working dependence structure.

A working dependence structure that captures a nested hierarchy of groups, each level of which contributes to
the random error term of the model.

When using this working covariance structure, dep_data of the GEE instance should contain a n_obs x k matrix
of 0/1 indicators, corresponding to the k subgroups nested under the top-level groups of the GEE instance. These
subgroups should be nested from left to right, so that two observations with the same value for column j
of dep_data should also have the same value for all columns j' < j (this only applies to observations in the same
top-level cluster given by the groups argument to GEE).

Notes

The calculations for this dependence structure involve all pairs of observations within a group (that is, within
the top level group structure passed to GEE). Large group sizes will result in slow iterations.

The variance components are estimated using least squares regression of the products r*r', for standardized
residuals r and r' in the same group, on a vector of indicators defining which variance components are shared by
r and r'.

Methods
statsmodels.genmod.dependence_structures.covstruct.Nested covariance_matrix

Returns the working covariance or correlation matrix for a given cluster of data.

Parameters endog_expval: array-like:
   The expected values of endog for the cluster for which the covariance or correlation
   matrix will be returned

index: integer:
   The index of the cluster for which the covariance or correlation matrix will be returned

Returns M: matrix:
   The covariance or correlation matrix of endog

is_cor: bool:
   True if M is a correlation matrix, False if M is a covariance matrix

statsmodels.genmod.dependence_structures.covstruct.Nested initialize

Called on the first call to update

ilabels is a list of n_i x n_i matrices containing integer labels that correspond to specific correlation pa-
rameters. Two elements of ilabels[i] with the same label share identical variance components.

designx is a matrix, with each row containing dummy variables indicating which variance components are
associated with the corresponding element of QY.

statsmodels.genmod.dependence_structures.covstruct.Nested summary

Returns a summary string describing the state of the dependence structure.

statsmodels.genmod.dependence_structures.covstruct.Nested update

Updates the association parameter values based on the current regression coefficients.

Parameters params: array-like
   Working values for the regression parameters.

Attributes

- Nested.dep_params
- Nested.designx
- Nested.designx_s
- Nested.designx_u

Continued on next page
Families

The distribution families are the same as for GLM, currently implemented are

| Family(link, variance) | The parent class for one-parameter exponential families. |
| Binomial(link) | Binomial exponential family distribution. |
| Gamma(link) | Gamma exponential family distribution. |
| Gaussian(link) | Gaussian exponential family distribution. |
| InverseGaussian(link) | InverseGaussian exponential family. |
| NegativeBinomial(link, alpha) | Negative Binomial exponential family. |
| Poisson(link) | Poisson exponential family. |

**statsmodels.genmod.families.family.Family**

class statsmodels.genmod.families.family.Family(link, variance)
The parent class for one-parameter exponential families.

Parameters:

- link : a link function instance
  
  Link is the linear transformation function. See the individual families for available links.

- variance : a variance function
  
  Measures the variance as a function of the mean probabilities. See the individual families for the default variance function.

Methods:

- deviance(Y, mu[, scale]) Deviance of (Y,mu) pair.
- fitted(eta) Fitted values based on linear predictors eta.
- loglike(Y, mu[, scale]) The loglikelihood function.
- predict(mu) Linear predictors based on given mu values.
- resid_anscombe(Y, mu) The Anscombe residuals.
- resid_dev(Y, mu[, scale]) The deviance residuals.
- starting_mu(y) Starting value for mu in the IRLS algorithm.
- weights(mu) Weights for IRLS steps

**statsmodels.genmod.families.family.Family.deviance**

Family.deviance(Y, mu, scale=1.0) Deviance of (Y,mu) pair.

Deviance is usually defined as twice the loglikelihood ratio.

Parameters:

- Y : array-like
  
  The endogenous response variable

- mu : array-like
The inverse of the link function at the linear predicted values.

**scale**: float, optional

An optional scale argument

Returns **DEV**: array

The value of deviance function defined below.

**Notes**

\[
DEV = \frac{\sum_i(2\loglike(Y_i,Y_i) - 2\loglike(Y_i,mu_i))}{\text{scale}}
\]

The deviance functions are analytically defined for each family.

**statsmodels.genmod.families.family.Family.fitted**

Family.fitted(eta)

Fitted values based on linear predictors eta.

**Parameters**

**eta**: array

Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

**Returns**

**mu**: array

The mean response variables given by the inverse of the link function.

**statsmodels.genmod.families.family.Family.loglike**

Family.loglike(Y, mu, scale=1.0)

The loglikelihood function.

**Parameters**

`Y`: array

Usually the endogenous response variable.

`mu`: array

Usually but not always the fitted mean response variable.

**Returns**

**llf**: float

The value of the loglikelihood evaluated at (Y, mu).

**Notes**

---

This is defined for each family. Y and mu are not restricted to:

`Y` and `mu` respectively. For instance, the deviance function calls:

both loglike(Y,Y) and loglike(Y,mu) to get the likelihood ratio.

**statsmodels.genmod.families.family.Family.predict**

Family.predict(mu)

Linear predictors based on given mu values.

**Parameters**

**mu**: array

The mean response variables

**Returns**

**eta**: array
Linear predictors based on the mean response variables. The value of the link function at the given mu.

```
statsmodels.genmod.families.family.Family.resid_anscombe
Family.resid_anscombe(Y, mu)
    The Anscombe residuals.
```

See also:
```
statsmodels.families.family.Family, the
```

```
statsmodels.genmod.families.family.Family.resid_dev
Family.resid_dev(Y, mu, scale=1.0)
    The deviance residuals
Parameters
    Y : array
        The endogenous response variable
    mu : array
        The inverse of the link function at the linear predicted values.
    scale : float, optional
        An optional argument to divide the residuals by scale
Returns
    Deviance residuals.
```

Notes

The deviance residuals are defined for each family.

```
statsmodels.genmod.families.family.Family.starting_mu
Family.starting_mu(y)
    Starting value for mu in the IRLS algorithm.
Parameters
    y : array
        The untransformed response variable.
Returns
    mu_0 : array
        The first guess on the transformed response variable.
```

Notes

Only the Binomial family takes a different initial value.

```
statsmodels.genmod.families.family.Family.weights
Family.weights(mu)
    Weights for IRLS steps
Parameters
    mu : array-like
        The transformed mean response variable in the exponential family
```
Returns \( w \): array

The weights for the IRLS steps

Notes

\[
w = 1 / (\text{link}'(\mu)^2 \times \text{variance}(\mu))
\]

\[\text{statsmodels.genmod.families.family.Binomial}\]

\text{class statsmodels.genmod.families.family.Binomial(link=<class 'statsmodels.genmod.families.links.logit'>)}

Binomial exponential family distribution.

Parameters **link** : a link instance, optional

The default link for the Binomial family is the logit link. Available links are logit, probit, cauchy, log, and cloglog. See \text{statsmodels.family.links} for more information.

See also:

\[\text{statsmodels.genmod.families.family.Family}\]

Notes

endog for Binomial can be specified in one of three ways.

Attributes

<table>
<thead>
<tr>
<th>Binomial.link</th>
<th>a link instance</th>
<th>The link function of the Binomial instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binomial.variance</td>
<td>varfunc instance</td>
<td>\textit{variance} is an instance of \text{statsmodels.family.varfuncs.binary}</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviance(Y, mu[, scale])</td>
<td>Deviance function for either Bernoulli or Binomial data.</td>
</tr>
<tr>
<td>fitted(eta)</td>
<td>Fitted values based on linear predictors eta.</td>
</tr>
<tr>
<td>initialize(Y)</td>
<td>Initialize the response variable.</td>
</tr>
<tr>
<td>loglike(Y, mu[, scale])</td>
<td>Loglikelihood function for Binomial exponential family distribution.</td>
</tr>
<tr>
<td>predict(mu)</td>
<td>Linear predictors based on given mu values.</td>
</tr>
<tr>
<td>resid_anscombe(Y, mu)</td>
<td>The Anscombe residuals</td>
</tr>
<tr>
<td>resid_dev(Y, mu[, scale])</td>
<td>Binomial deviance residuals</td>
</tr>
<tr>
<td>starting_mu(y)</td>
<td>The starting values for the IRLS algorithm for the Binomial family.</td>
</tr>
<tr>
<td>weights(mu)</td>
<td>Weights for IRLS steps</td>
</tr>
</tbody>
</table>

\[\text{statsmodels.genmod.families.family.Binomial.deviance}\]

\[\text{Binomial.deviance}(Y, mu, scale=1.0)\]

Deviance function for either Bernoulli or Binomial data.

Parameters Y : array-like

Endogenous response variable (already transformed to a probability if appropriate).
mu : array
    Fitted mean response variable
scale : float, optional
    An optional scale argument

Returns deviance : float
    The deviance function as defined below

Notes

If the endogenous variable is binary:
\[
deviance = -2 \times \sum (I_{\text{one}} \times \log(mu) + (I_{\text{zero}}) \times \log(1-mu))
\]
where I_{\text{one}} is an indicator function that evaluates to 1 if \(Y_i == 1\). and I_{\text{zero}} is an indicator function that evaluates to 1 if \(Y_i == 0\).

If the model is binomial:
\[
deviance = 2 \times \sum (\log(Y/mu) + (n-Y) \times \log((n-Y)/(n-mu)))
\]
where \(Y\) and \(n\) are as defined in Binomial.initialize.

statsmodels.genmod.families.family.Binomial.fitted
    Binomial.fitted(eta)
    Fitted values based on linear predictors eta.

Parameters eta : array
    Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

Returns mu : array
    The mean response variables given by the inverse of the link function.

statsmodels.genmod.families.family.Binomial.initialize
    Binomial.initialize(Y)
    Initialize the response variable.

Parameters Y : array
    Endogenous response variable

Returns If ‘Y’ is binary, returns ‘Y’ :
    If ‘Y’ is a 2d array, then the input is assumed to be in the format :
    (successes, failures) and :
    successes/(success + failures) is returned. And n is set to :
    successes + failures. :

statsmodels.genmod.families.family.Binomial.loglike
    Binomial.loglike(Y, mu, scale=1.0)
    Loglikelihood function for Binomial exponential family distribution.

Parameters Y : array-like
Endogenous response variable

\[ \textbf{mu} : \text{array-like} \]

Fitted mean response variable

\[ \textbf{scale} : \text{float, optional} \]

The default is 1.

Returns \( \textbf{llf} : \text{float} \)

The value of the loglikelihood function evaluated at \((Y, \mu, \text{scale})\) as defined below.

Notes

If \( Y \) is binary:  
\[ llf = \text{scale}\cdot\text{sum}(Y\cdot \log(\mu/(1-\mu)) + \log(1-\mu)) \]

If \( Y \) is binomial:  
\[ llf = \text{scale}\cdot\text{sum}(\text{gammaln}(n+1) - \text{gammaln}(y+1) - \text{gammaln}(n-y+1) + y\cdot \log(\mu/(1-\mu)) + n\cdot \log(1-\mu)) \]

where \text{gammaln} is the log gamma function and \( y = Y\cdot n \) with \( Y \) and \( n \) as defined in Binomial initialize. This simply makes \( y \) the original number of successes.

\texttt{statsmodels.genmod.families.family.Binomial.predict}

\begin{verbatim}
Binomial.predict \( \text{(mu)} \)
Linear predictors based on given mu values.

Parameters \textbf{mu} : array

The mean response variables

Returns \textbf{eta} : array

Linear predictors based on the mean response variables. The value of the link function at the given mu.
\end{verbatim}

\texttt{statsmodels.genmod.families.family.Binomial.resid_anscombe}

\begin{verbatim}
Binomial.resid_anscombe \( \text{(Y, mu)} \)
The Anscombe residuals

Parameters \textbf{Y} : array-like

Endogenous response variable

\textbf{mu} : array-like

Fitted mean response variable

Returns \textbf{resid_anscombe} : array

The Anscombe residuals as defined below.
\end{verbatim}

Notes

\[ \sqrt{n}\cdot(\text{cox_snell}(Y) - \text{cox_snell}(\mu))/(\mu^{*}(1/6.)*(1-\mu)^{*}(1/6.)) \]

where \text{cox_snell} is defined as \text{cox_snell}(x) = \text{betainc}(2/3., 2/3., x)*\text{betainc}(2/3., 2/3.) where \text{betainc} is the incomplete beta function
The name ‘cox_snell’ is idiosyncratic and is simply used for convenience following the approach suggested in Cox and Snell (1968). Further note that cox_snell(x) = x**(2/3.)/(2/3.)*hyp2f1(2/3.,1/3.,5/3.,x) where hyp2f1 is the hypergeometric 2f1 function. The Anscombe residuals are sometimes defined in the literature using the hyp2f1 formulation. Both betainc and hyp2f1 can be found in scipy.

References


statsmodels.genmod.families.family.Binomial.resid_dev

Binomial.resid_dev(Y, mu, scale=1.0)

Binomial deviance residuals

Parameters

Y : array-like
    Endogenous response variable

mu : array-like
    Fitted mean response variable

scale : float, optional
    An optional argument to divide the residuals by scale

Returns

resid_dev : array
    Deviance residuals as defined below

Notes

If Y is binary:

resid_dev = sign(Y-mu)*sqrt(-2*log(I_one*mu + I_zero*(1-mu)))

where I_one is an indicator function that evaluates as 1 if Y == 1 and I_zero is an indicator function that evaluates as 1 if Y == 0.

If Y is binomial:

resid_dev = sign(Y-mu)*sqrt(2*n*(Y*log(Y/mu)+(1-Y)*log((1-Y)/(1-mu))))

where Y and n are as defined in Binomial.initialize.

statsmodels.genmod.families.family.Binomial.starting_mu

Binomial.starting_mu(y)

The starting values for the IRLS algorithm for the Binomial family.

A good choice for the binomial family is

starting_mu = (y + .5)/2
**statsmodels.genmod.families.family.Binomial.weights**

*Binomial.weights(mu)*

Weights for IRLS steps

**Parameters**

`mu` : array-like

The transformed mean response variable in the exponential family

**Returns**

`w` : array

The weights for the IRLS steps

**Notes**

\[ w = \frac{1}{(\text{link}'(\mu)^2 \times \text{variance}(\mu))} \]

**statsmodels.genmod.families.family.Gamma**

*class statsmodels.genmod.families.family.Gamma(link=<class 'statsmodels.genmod.families.links.inverse_power'>)*

Gamma exponential family distribution.

**Parameters**

`link` : a link instance, optional

The default link for the Gamma family is the inverse link. Available links are log, identity, and inverse. See statsmodels.family.links for more information.

**See also:**

statsmodels.genmod.families.family.Family

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gamma.link</td>
<td>a link instance</td>
</tr>
<tr>
<td>Gamma.variance</td>
<td>varfunc instance</td>
</tr>
<tr>
<td>variance</td>
<td><code>variance</code> is an instance of statsmodels.family.varfuncs.mu_squared</td>
</tr>
</tbody>
</table>

**Methods**

- **deviance(Y, mu[, scale])**
  - Gamma deviance function
- **fitted(eta)**
  - Fitted values based on linear predictors eta.
- **loglike(Y, mu[, scale])**
  - Loglikelihood function for Gamma exponential family distribution.
- **predict(mu)**
  - Linear predictors based on given mu values.
- **resid_anscombe(Y, mu)**
  - The Anscombe residuals for Gamma exponential family distribution
- **resid_dev(Y, mu[, scale])**
  - Gamma deviance residuals
- **starting_mu(y)**
  - Starting value for mu in the IRLS algorithm.
- **weights(mu)**
  - Weights for IRLS steps

**statsmodels.genmod.families.family.Gamma.deviance**

*Gamma.deviance(Y, mu, scale=1.0)*

Gamma deviance function

**Parameters**

`Y` : array-like

Endogenous response variable
mu : array-like
Fitted mean response variable

scale : float, optional
An optional scale argument

Returns deviance : float
Deviance function as defined below

Notes

\[ deviance = 2 \sum \left( \frac{Y - \mu}{\mu} - \log \left( \frac{Y}{\mu} \right) \right) \]

statsmodels.genmod.families.family.Gamma.fitted

Gamma.fitted(eta)
Fitted values based on linear predictors eta.

Parameters eta : array
Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

Returns mu : array
The mean response variables given by the inverse of the link function.

statsmodels.genmod.families.family.Gamma.loglike

Gamma.loglike(Y, mu, scale=1.0)
Loglikelihood function for Gamma exponential family distribution.

Parameters Y : array-like
Endogenous response variable

mu : array-like
Fitted mean response variable

scale : float, optional
The default is 1.

Returns llf : float
The value of the loglikelihood function evaluated at (Y,mu,scale) as defined below.

Notes

\[ llf = -\frac{1}{\text{scale}} \sum \left( \frac{Y}{\mu} + \log(\mu) + (\text{scale}-1)\log(Y) + \log(\text{scale}) + \text{scale}\cdot\text{gammaln}(1/\text{scale}) \right) \] where \text{gammaln} is the log gamma function.

statsmodels.genmod.families.family.Gamma.predict

Gamma.predict(mu)
Linear predictors based on given mu values.

Parameters mu : array
The mean response variables
Returns eta: array

Linear predictors based on the mean response variables. The value of the link function at the given mu.

\[ \text{resid_anscombe}(Y, \mu) \]

The Anscombe residuals for Gamma exponential family distribution

**Parameters**

- **Y**: array
  - Endogenous response variable
- **mu**: array
  - Fitted mean response variable

**Returns resid_anscombe**: array

The Anscombe residuals for the Gamma family defined below

**Notes**

\[
\text{resid_anscombe} = 3*(Y^{1/3} - \mu^{1/3})/\mu^{1/3}
\]

\[ \text{resid_dev}(Y, \mu, \text{scale}=1.0) \]

Gamma deviance residuals

**Parameters**

- **Y**: array-like
  - Endogenous response variable
- **mu**: array-like
  - Fitted mean response variable
- **scale**: float, optional
  - An optional argument to divide the residuals by scale

**Returns resid_dev**: array

Deviance residuals as defined below

**Notes**

\[
\text{resid_dev} = \text{sign}(\text{Y} - \mu) \times \sqrt{2 \times \left(-\frac{\text{Y} - \mu}{\mu} + \log(\frac{\text{Y}}{\mu})\right)}
\]

\[ \text{starting_mu}(y) \]

Starting value for mu in the IRLS algorithm.

**Parameters**

- **y**: array
  - The untransformed response variable.

**Returns mu_0**: array

The first guess on the transformed response variable.
Notes

Only the Binomial family takes a different initial value.

```
statsmodels.genmod.families.family.Gamma.weights
Gamma.weights(mu)
   Weights for IRLS steps
   Parameters mu : array-like
       The transformed mean response variable in the exponential family
   Returns w : array
       The weights for the IRLS steps

Notes

w = 1 / (link'(mu)**2 * variance(mu))
```

```
statsmodels.genmod.families.family.Gaussian

class statsmodels.genmod.families.family.Gaussian(link=<class 'statsmodels.genmod.families.links.identity'>)
Gaussian exponential family distribution.

   Parameters link : a link instance, optional
       The default link for the Gaussian family is the identity link. Available links are log, identity, and inverse. See statsmodels.family.links for more information.

See also:

statsmodels.genmod.families.family.Family

Attributes

| Gaussian.link | a link instance
<table>
<thead>
<tr>
<th>Gaussian.variance</th>
<th>varfunc instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>The link function of the Gaussian instance</td>
<td></td>
</tr>
</tbody>
</table>

variance is an instance of statsmodels.family.varfuncs.constant

Methods

| deviance(Y, mu[, scale]) | Gaussian deviance function |
| fitted(eta) | Fitted values based on linear predictors eta. |
| loglike(Y, mu[, scale]) | Loglikelihood function for Gaussian exponential family distribution. |
| predict(mu) | Linear predictors based on given mu values. |
| resid_anscombe(Y, mu) | The Anscombe residuals for the Gaussian exponential family distribution |
| resid_dev(Y, mu[, scale]) | Gaussian deviance residuals |
| starting_mu(y) | Starting value for mu in the IRLS algorithm. |
| weights(mu) | Weights for IRLS steps |

statsmodels.genmod.families.family.Gaussian.deviance
**Gaussian.deviance** $(Y, mu, scale=1.0)$

Gaussian deviance function

**Parameters**

- $Y$: array-like
  - Endogenous response variable
- $mu$: array-like
  - Fitted mean response variable
- $scale$: float, optional
  - An optional scale argument

**Returns**

- $deviance$: float
  - The deviance function at $(Y, mu)$ as defined below.

**Notes**

$$deviance = \text{sum}((Y-mu)^2)$$

**Gaussian.fitted**

Fitted values based on linear predictors $eta$.

**Parameters**

- $eta$: array
  - Values of the linear predictor of the model. $\text{dot}(X, \beta)$ in a classical linear model.

**Returns**

- $mu$: array
  - The mean response variables given by the inverse of the link function.

**Gaussian.loglike**

Loglikelihood function for Gaussian exponential family distribution.

**Parameters**

- $Y$: array-like
  - Endogenous response variable
- $mu$: array-like
  - Fitted mean response variable
- $scale$: float, optional
  - Scales the loglikelihood function. The default is 1.

**Returns**

- $llf$: float
  - The value of the loglikelihood function evaluated at $(Y, mu, scale)$ as defined below.

**Notes**

If the link is the identity link function then the loglikelihood function is the same as the classical OLS model. $llf = -(\text{obs}/2)*\text{log(SSR)} + (1 + \text{log}(2*pi/\text{obs}))$ where $SSR = \text{sum}((Y-\text{link}^{-1}(mu))^2)$
If the link is not the identity link then the loglikelihood function is defined as
\[ \text{llf} = \sum \left( \frac{Y^\prime \mu - \mu^\prime \mu^2/2}{\text{scale}} - \frac{Y^2}{2 \cdot \text{scale}} - \frac{1}{2} \cdot \log(2\pi \cdot \text{scale}) \right) \]

**statsmodels.genmod.families.family.Gaussian.predict**

Gaussian.predict(mu)

Linear predictors based on given mu values.

**Parameters**

mu : array

The mean response variables

**Returns**

eta : array

Linear predictors based on the mean response variables. The value of the link function at the given mu.

**statsmodels.genmod.families.family.Gaussian.resid_anscombe**

Gaussian.resid_anscombe(Y, mu)

The Anscombe residuals for the Gaussian exponential family distribution

**Parameters**

Y : array

Endogenous response variable

mu : array

Fitted mean response variable

**Returns**

resid_anscombe : array

The Anscombe residuals for the Gaussian family defined below

**Notes**

\[ \text{resid_anscombe} = Y - \mu \]

**statsmodels.genmod.families.family.Gaussian.resid_dev**

Gaussian.resid_dev(Y, mu, scale=1.0)

Gaussian deviance residuals

**Parameters**

Y : array-like

Endogenous response variable

mu : array-like

Fitted mean response variable

scale : float, optional

An optional argument to divide the residuals by scale

**Returns**

resid_dev : array

Deviance residuals as defined below

**Notes**

\[ \text{resid_dev} = (Y - \mu) / \sqrt{\text{variance}(\mu)} \]
statsmodels.genmod.families.family.Gaussian.starting_mu
Gaussian.starting_mu(y)
Starting value for mu in the IRLS algorithm.

**Parameters**

- **y**: array
  The untransformed response variable.

**Returns**

- **mu_0**: array
  The first guess on the transformed response variable.

**Notes**

Only the Binomial family takes a different initial value.

---

statsmodels.genmod.families.family.Gaussian.weights
Gaussian.weights(mu)
Weights for IRLS steps

**Parameters**

- **mu**: array-like
  The transformed mean response variable in the exponential family

**Returns**

- **w**: array
  The weights for the IRLS steps

**Notes**

\[
    w = \frac{1}{(\text{link}'(mu)^2 \ast \text{variance}(mu))}
\]

---

statsmodels.genmod.families.family.InverseGaussian
class statsmodels.genmod.families.family.InverseGaussian
InverseGaussian exponential family.

**Parameters**

- **link**: a link instance, optional
  The default link for the inverse Gaussian family is the inverse squared link. Available links are inverse_squared, inverse, log, and identity. See statsmodels.family.links for more information.

**See also:**

- statsmodels.genmod.families.family.Family

**Notes**

The inverse Guassian distribution is sometimes referred to in the literature as the wald distribution.
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InverseGaussian.link</td>
<td>a link instance</td>
</tr>
<tr>
<td>InverseGaussian.variance</td>
<td>The link function of the inverse Gaussian instance</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviance(Y, mu[, scale])</td>
<td>Inverse Gaussian deviance function</td>
</tr>
<tr>
<td>fitted(eta)</td>
<td>Fitted values based on linear predictors eta.</td>
</tr>
<tr>
<td>loglike(Y, mu[, scale])</td>
<td>Loglikelihood function for inverse Gaussian distribution.</td>
</tr>
<tr>
<td>predict(mu)</td>
<td>Linear predictors based on given mu values.</td>
</tr>
<tr>
<td>resid_anscombe(Y, mu)</td>
<td>The Anscombe residuals for the inverse Gaussian distribution.</td>
</tr>
<tr>
<td>resid_dev(Y, mu[, scale])</td>
<td>Returns the deviance residuals for the inverse Gaussian distribution.</td>
</tr>
<tr>
<td>starting_mu(y)</td>
<td>Starting value for mu in the IRLS algorithm.</td>
</tr>
<tr>
<td>weights(mu)</td>
<td>Weights for IRLS steps</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.family.InverseGaussian.deviance

InverseGaussian.deviance(Y, mu, scale=1.0)
Inverse Gaussian deviance function

Parameters

- **Y**: array-like
  Endogenous response variable
- **mu**: array-like
  Fitted mean response variable
- **scale**: float, optional
  An optional scale argument

Returns** deviance**: float
Deviance function as defined below

**Notes**

\[
deviance = \sum((Y-mu)^2/(Y*mu^2))
\]

statsmodels.genmod.families.family.InverseGaussian.fitted

InverseGaussian.fitted(eta)
Fitted values based on linear predictors eta.

Parameters **eta**: array
Values of the linear predictor of the model. dot(X,beta) in a classical linear model.

Returns **mu**: array
The mean response variables given by the inverse of the link function.
InverseGaussian.loglike(Y, mu, scale=1.0)

Loglikelihood function for inverse Gaussian distribution.

Parameters

Y : array-like
Endogenous response variable
mu : array-like
Fitted mean response variable
scale : float, optional
The default is 1.

Returns

llf : float
The value of the loglikelihood function evaluated at (Y, mu, scale) as defined below.

Notes

\[
llf = -(1/2.)*\sum(\frac{(Y-mu)^2}{Y*mu^2*scale}) + \log(scale*Y^3) + \log(2*pi)
\]

InverseGaussian.predict(mu)

Linear predictors based on given mu values.

Parameters

mu : array
The mean response variables

Returns

eta : array
Linear predictors based on the mean response variables. The value of the link function at the given mu.

InverseGaussian.resid_anscombe(Y, mu)

The Anscombe residuals for the inverse Gaussian distribution

Parameters

Y : array
Endogenous response variable
mu : array
Fitted mean response variable

Returns

resid_anscombe : array
The Anscombe residuals for the inverse Gaussian distribution as defined below

Notes

\[
resid_anscombe = \log(Y/mu)/\sqrt{mu}
\]
InverseGaussian.resid_dev($Y, \mu, scale=1.0$)

Returns the deviance residuals for the inverse Gaussian family.

**Parameters**
- $Y$: array-like
  - Endogenous response variable
- $\mu$: array-like
  - Fitted mean response variable
- $scale$: float, optional
  - An optional argument to divide the residuals by scale

**Returns**
- resid_dev: array
  - Deviance residuals as defined below

**Notes**

\[
dev_{resid} = \text{sign}(Y-\mu)\sqrt{(Y-\mu)^2/(Y\mu^2)}
\]

InverseGaussian.starting_mu($y$)

Starting value for $\mu$ in the IRLS algorithm.

**Parameters**
- $y$: array
  - The untransformed response variable.

**Returns**
- $\mu_0$: array
  - The first guess on the transformed response variable.

**Notes**

Only the Binomial family takes a different initial value.

InverseGaussian.weights($\mu$)

Weights for IRLS steps

**Parameters**
- $\mu$: array-like
  - The transformed mean response variable in the exponential family

**Returns**
- $w$: array
  - The weights for the IRLS steps

**Notes**

\[
w = 1 / (\text{link}'(\mu)^2 * \text{variance}(\mu))
\]
Negative Binomial exponential family.

**Parameters**

- **link**: a link instance, optional
  
  The default link for the negative binomial family is the log link. Available links are log, cloglog, identity, nbinom and power. See statsmodels.family.links for more information.

- **alpha**: float, optional
  
  The ancillary parameter for the negative binomial distribution. For now alpha is assumed to be nonstochastic. The default value is 1. Permissible values are usually assumed to be between .01 and 2.

**See also:**

statsmodels.genmod.families.family.Family

**Notes**

Support for Power link functions is not yet supported.

**Attributes**

- **NegativeBinomial.link**: a link instance
  
  The link function of the negative binomial instance

- **NegativeBinomial.variance**: varfunc instance
  
  variance is an instance of statsmodels.family.varfuncs.nbinom

**Methods**

- **deviance(Y, mu[, scale])**
  
  Returns the value of the deviance function.

- **fitted(eta)**
  
  Fitted values based on linear predictors eta.

- **loglike(Y[, fittedvalues])**
  
  The loglikelihood function for the negative binomial family.

- **predict(mu)**
  
  Linear predictors based on given mu values.

- **resid_anscombe(Y, mu)**
  
  The Anscombe residuals for the negative binomial family

- **resid_dev(Y, mu[, scale])**
  
  Negative Binomial Deviance Residual

- **starting_mu(y)**
  
  Starting value for mu in the IRLS algorithm.

- **weights(mu)**
  
  Weights for IRLS steps

**statsmodels.genmod.families.family.NegativeBinomial.deviance**

NegativeBinomial.deviance(Y, mu, scale=1.0)

Returns the value of the deviance function.

**Parameters**

- **Y**: array-like
  
  Endogenous response variable

- **mu**: array-like
  
  Fitted mean response variable

- **scale**: float, optional
An optional scale argument

**Returns deviance**: float

Deviance function as defined below

**Notes**

\[
\text{deviance} = \text{sum}(\text{piecewise})
\]

where piecewise is defined as

if \(Y_i == 0\) :

\[
\text{piecewise}_i = \frac{2 \log (1 + \alpha \cdot \mu)}{\alpha}
\]

if \(Y_i > 0\):

\[
\text{piecewise}_i = 2Y \log(Y/\mu) - \frac{2}{\alpha} \frac{1}{1 + \alpha Y} \frac{1}{\log((1 + \alpha Y)/(1 + \alpha \mu))}
\]

**statsmodels.genmod.families.family.NegativeBinomial.fitted**

NegativeBinomial.fitted(\(\eta\))

Fitted values based on linear predictors \(\eta\).

**Parameters** \(\eta\) : array

Values of the linear predictor of the model. \(\text{dot}(X, \beta)\) in a classical linear model.

**Returns** \(\mu\) : array

The mean response variables given by the inverse of the link function.

**statsmodels.genmod.families.family.NegativeBinomial.loglike**

NegativeBinomial.loglike(\(Y\), fittedvalues=None)

The loglikelihood function for the negative binomial family.

**Parameters** \(Y\) : array-like

Endogenous response variable

\(\text{fittedvalues}\) : array-like

The linear fitted values of the model. This is \(\text{dot}(\text{exog}, \text{params})\).

**Returns** \(\text{llf}\) : float

The value of the loglikelihood function evaluated at \((Y, \mu, \text{scale})\) as defined below.

**Notes**

\[
\text{sum}(Y \log(\alpha \exp(\text{fittedvalues})/(1 + \alpha \exp(\text{fittedvalues}))) - \log(1 + \alpha \exp(\text{fittedvalues})))/\alpha + \text{constant})
\]

where constant is defined as constant = \(\text{gammaln}(Y + 1/\alpha) - \text{gammaln}(Y + 1) - \text{gammaln}(1/\alpha)\)
statsmodels.genmod.families.family.NegativeBinomial.predict

NegativeBinomial.predict (mu)

Linear predictors based on given mu values.

Parameters
mu : array
  The mean response variables

Returns
eta : array
  Linear predictors based on the mean response variables. The value of the link function at the given mu.

statsmodels.genmod.families.family.NegativeBinomial.resid_anscombe

NegativeBinomial.resid_anscombe (Y, mu)

The Anscombe residuals for the negative binomial family

Parameters
Y : array-like
  Endogenous response variable
mu : array-like
  Fitted mean response variable

Returns
resid_anscombe : array
  The Anscombe residuals as defined below.

Notes

\[
\text{resid}_\text{anscombe} = \left( \frac{\text{hyp2f1}(-\alpha*Y) - \text{hyp2f1}(-\alpha*mu) + 1.5*(Y^{(2/3)}-mu^{(2/3)})}{(mu+alpha*mu^{(2/3)})^{(1/6.)}} \right)
\]

where \( \text{hyp2f1} \) is the hypergeometric 2f1 function parameterized as \( \text{hyp2f1}(x) = \text{hyp2f1}(2/3.,1/3.,5/3.,x) \)

statsmodels.genmod.families.family.NegativeBinomial.resid_dev

NegativeBinomial.resid_dev (Y, mu, scale=1.0)

Negative Binomial Deviance Residual

Parameters
Y : array-like
  Y is the response variable
mu : array-like
  mu is the fitted value of the model
scale : float, optional
  An optional argument to divide the residuals by scale

Returns
resid_dev : array
  The array of deviance residuals

3.3. Generalized Estimating Equations 297
Notes

\[ resid\_dev = \text{sign}(Y - \mu) \times \sqrt{\text{piecewise}} \]

where piecewise is defined as if \( Y_i = 0 \): \[ \text{piecewise}_i = 2 \times \log(1 + \alpha \times \mu)/\alpha \]
if \( Y_i > 0 \): \[ \text{piecewise}_i = 2 \times Y \times \log(Y/\mu) - 2/\alpha \times (1 + \alpha \times Y) \times \log((1 + \alpha \times Y)/(1 + \alpha \times \mu)) \]

**statsmodels.genmod.families.family.NegativeBinomial.starting_mu**

NegativeBinomial.starting_mu(y)

Starting value for \( \mu \) in the IRLS algorithm.

**Parameters**

- **y**: array
  
The untransformed response variable.

**Returns**

- **mu_0**: array
  
The first guess on the transformed response variable.

Notes

Only the Binomial family takes a different initial value.

**statsmodels.genmod.families.family.NegativeBinomial.weights**

NegativeBinomial.weights(mu)

Weights for IRLS steps

**Parameters**

- **mu**: array-like
  
The transformed mean response variable in the exponential family

**Returns**

- **w**: array
  
The weights for the IRLS steps

Notes

\[ w = 1 / (\text{link}'(\mu)^2 \times \text{variance}(\mu)) \]

**statsmodels.genmod.families.family.Poisson**

class statsmodels.genmod.families.family.Poisson(link=class 'statsmodels.genmod.families.links.log')

Poisson exponential family.

**Parameters**

- **link**: a link instance, optional
  
The default link for the Poisson family is the log link. Available links are log, identity, and sqrt. See statsmodels.family.links for more information.

**See also:**

statsmodels.genmod.families.family.Family
Attributes

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson.link</td>
<td>a link instance</td>
</tr>
<tr>
<td>Poisson.variance</td>
<td>The link function of the Poisson instance. variance is an instance of statsmodels.genmod.families.family.varfuncs.mu</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviance(Y, mu[, scale])</td>
<td>Poisson deviance function</td>
</tr>
<tr>
<td>fitted(eta)</td>
<td>Fitted values based on linear predictors eta.</td>
</tr>
<tr>
<td>loglike(Y, mu[, scale])</td>
<td>Loglikelihood function for Poisson exponential family distribution.</td>
</tr>
<tr>
<td>predict(mu)</td>
<td>Linear predictors based on given mu values.</td>
</tr>
<tr>
<td>resid_anscombe(Y, mu)</td>
<td>Anscombe residuals for the Poisson exponential family distribution</td>
</tr>
<tr>
<td>resid_dev(Y, mu[, scale])</td>
<td>Poisson deviance residual</td>
</tr>
<tr>
<td>starting_mu(y)</td>
<td>Starting value for mu in the IRLS algorithm.</td>
</tr>
<tr>
<td>weights(mu)</td>
<td>Weights for IRLS steps</td>
</tr>
</tbody>
</table>

```
statsmodels.genmod.families.family.Poisson.deviance
Poisson.deviance(Y, mu, scale=1.0)
```

Parameters

- **Y**: array-like
  - Endogenous response variable
- **mu**: array-like
  - Fitted mean response variable
- **scale**: float, optional
  - An optional scale argument

Returns

- **deviance**: float
  - The deviance function at (Y, mu) as defined below.

Notes

If a constant term is included it is defined as

\[ deviance = 2 \sum_i (Y \log(Y/\mu)) \]

```
statsmodels.genmod.families.family.Poisson.fitted
Poisson.fitted(eta)
```

Parameters

- **eta**: array
  - Values of the linear predictor of the model. dot(X, beta) in a classical linear model.

Returns

- **mu**: array
  - The mean response variables given by the inverse of the link function.
Poisson.loglike(Y, mu, scale=1.0)

Parameters  
Y : array-like
Endogenous response variable  
mu : array-like
Fitted mean response variable  
scale : float, optional
The default is 1.

Returns  
llf : float
The value of the loglikelihood function evaluated at (Y, mu, scale) as defined below.

Notes

llf = scale * sum(-mu + Y*log(mu) - gammaln(Y+1)) where gammaln is the log gamma function

Poisson.predict(mu)

Parameters  
mu : array
The mean response variables

Returns  
eta : array
Linear predictors based on the mean response variables. The value of the link function at the given mu.

Poisson.resid_anscombe(Y, mu)

Parameters  
Y : array-like
Endogenous response variable  
mu : array-like
Fitted mean response variable

Returns  
resid_anscombe : array
The Anscome residuals for the Poisson family defined below

Notes

resid_anscombe = (3/2.* (Y**(2/3.) - mu**(2/3.)))/mu**(1/6.).
statsmodels.genmod.families.family.Poisson.resid_dev

Poisson.resid_dev(Y, mu, scale=1.0)

Poisson deviance residual

Parameters

- **Y** : array-like
  - Endogenous response variable
- **mu** : array-like
  - Fitted mean response variable
- **scale** : float, optional
  - An optional argument to divide the residuals by scale

Returns

**resid_dev** : array
  - Deviance residuals as defined below

Notes

\[
\text{resid}_\text{dev} = \text{sign}(Y - \mu) * \sqrt{2 * Y \log(Y/\mu) - 2 * (Y - \mu)}
\]

statsmodels.genmod.families.family.Poisson.starting_mu

Poisson.starting_mu(y)

Starting value for \( \mu \) in the IRLS algorithm.

Parameters

- **y** : array
  - The untransformed response variable.

Returns

**mu_0** : array
  - The first guess on the transformed response variable.

Notes

Only the Binomial family takes a different initial value.

statsmodels.genmod.families.family.Poisson.weights

Poisson.weights(mu)

Weights for IRLS steps

Parameters

- **mu** : array-like
  - The transformed mean response variable in the exponential family

Returns

**w** : array
  - The weights for the IRLS steps

Notes

\[
w = 1 / (\text{link}'(\mu)^2 * \text{variance}(\mu))
\]
## Link Functions

The link functions are the same as for GLM, currently implemented are the following. Not all link functions are available for each distribution family. The list of available link functions can be obtained by

```python
>>> sm.families.family.<familyname>.links
```

<table>
<thead>
<tr>
<th>Link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDFFLink((dbn))</td>
<td>The use the CDF of a scipy.stats distribution</td>
</tr>
<tr>
<td>CLogLog</td>
<td>The complementary log-log transform</td>
</tr>
<tr>
<td>Log</td>
<td>The log transform ..</td>
</tr>
<tr>
<td>Logit</td>
<td>The logit transform ..</td>
</tr>
<tr>
<td>NegativeBinomial([alpha])</td>
<td>The negative binomial link function</td>
</tr>
<tr>
<td>Power([power])</td>
<td>The power transform</td>
</tr>
<tr>
<td>cauchy()</td>
<td>The Cauchy (standard Cauchy CDF) transform</td>
</tr>
<tr>
<td>cloglog</td>
<td>The CLogLog transform link function.</td>
</tr>
<tr>
<td>identity()</td>
<td>The identity transform ..</td>
</tr>
<tr>
<td>inverse_power()</td>
<td>The inverse transform ..</td>
</tr>
<tr>
<td>inverse_squared()</td>
<td>The inverse squared transform ..</td>
</tr>
<tr>
<td>log</td>
<td>The log transform ..</td>
</tr>
<tr>
<td>logit</td>
<td></td>
</tr>
<tr>
<td>nbinom([alpha])</td>
<td>The negative binomial link function.</td>
</tr>
<tr>
<td>probit((dbn))</td>
<td>The probit (standard normal CDF) transform</td>
</tr>
</tbody>
</table>

```python
class statsmodels.genmod.families.links.Link

A generic link function for one-parameter exponential family.

_A Link_ does nothing, but lays out the methods expected of any subclass.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of the link function $g'(p)$. Just a placeholder.</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of the link function.</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse link function $g^a(-1)(z)$.</td>
</tr>
</tbody>
</table>

```python
Link.deriv(p)

Derivative of the link function $g'(p)$. Just a placeholder.

**Parameters**  
`p` : array-like

**Returns**  
The value of the derivative of the link function $g'(p)$:

```python
Link.inverse(z)

Inverse of the link function. Just a placeholder.

**Parameters**  
`z` : array-like

`z` is usually the linear predictor of the transformed variable in the IRLS algorithm for GLM.
Returns The value of the inverse of the link function $g^{-1}(z) = p$:

```
statsmodels.genmod.families.links.Link.inverse_deriv
```

Derivative of the inverse link function $g^{-1}(z)$.

**Parameters**

`z` : array-like

`z` is usually the linear predictor for a GLM or GEE model.

**Returns**

The value of the derivative of the inverse of the link function:

**Notes**

This reference implementation gives the correct result but it inefficient, so it can be overridden in subclasses.

```
statsmodels.genmod.families.links.CDFLink
class statsmodels.genmod.families.links.CDFLink (dbn=<scipy.stats.distributions.norm_gen
object at 0x0534E8D0>)
```

The use the CDF of a scipy.stats distribution

CDFLink is a subclass of logit in order to use its `_clean` method for the link and its derivative.

**Parameters**

`dbn` : scipy.stats distribution

Default is `dbn=scipy.stats.norm`

**Notes**

The CDF link is untested.

**Methods**

```
derv(p) Derivative of CDF link
inverse(z) The inverse of the CDF link
inverse_deriv(z) Derivative of the inverse of the CDF transformation link function
```

```
statsmodels.genmod.families.links.CDFLink.deriv
CDFLink.deriv(p)
```

Derivative of CDF link

**Parameters**

`p` : array-like

mean parameters

**Returns**

`g'(p)` : array

The derivative of CDF transform at `p`

**Notes**

$g'(p) = 1 / dbn.pdf(dbn.ppf(p))$
**statsmodels.genmod.families.links.CDFLink.inverse**

CDFLink.inverse(z)

The inverse of the CDF link

**Parameters**

z : array-like

The value of the inverse of the link function at p

**Returns**

p : array

Mean probabilities. The value of the inverse of CDF link of z

**Notes**

\[ g^{-1}(z) = dbn.cdf(z) \]

**statsmodels.genmod.families.links.CDFLink.inverse_deriv**

CDFLink.inverse_deriv(z)

Derivative of the inverse of the CDF transformation link function

**Parameters**

z : array

The inverse of the link function at p

**Returns**

The value of the derivative of the inverse of the logit function:

\[ g'(p) \]

**class statsmodels.genmod.families.links.CLogLog**

The complementary log-log transform

CLogLog inherits from Logit in order to have access to its _clean method for the link and its derivative.

**Notes**

CLogLog is untested.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of C-Log-Log transform link function</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of C-Log-Log transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the C-Log-Log transform link function</td>
</tr>
</tbody>
</table>

**statsmodels.genmod.families.links.CLogLog.deriv**

CLogLog.deriv(p)

Derivative of C-Log-Log transform link function

**Parameters**

p : array-like

Mean parameters

**Returns**

g'(p) : array

The derivative of the CLogLog transform link function
Notes

\[ g'(p) = - \frac{1}{\log(p) * p} \]

\[ \text{statsmodels.genmod.families.links.CLogLog.inverse} \]
\[ \text{CLogLog.}^{-}\text{inverse}(z) \]
Inverse of C-Log-Log transform link function

Parameters:
\[ z : \text{array-like} \]

The value of the inverse of the CLogLog link function at \( p \)

Returns:
\[ p : \text{array} \]

Mean parameters

Notes

\[ g^\Gamma(-1)(z) = 1 - \exp(-\exp(z)) \]

\[ \text{statsmodels.genmod.families.links.CLogLog.inverse_deriv} \]
\[ \text{CLogLog.}^{-}\text{inverse}_\text{deriv}(z) \]
Derivative of the inverse of the C-Log-Log transform link function

Parameters:
\[ z : \text{array-like} \]

The value of the inverse of the CLogLog link function at \( p \)

Returns:
The derivative of the inverse of the CLogLog link function

\[ \text{statsmodels.genmod.families.links.Log} \]
\[ \text{class statsmodels.genmod.families.links.Log} \]
The log transform

Notes

call and derivative call a private method \_clean to trim the data by machine epsilon so that \( p \) is in \((0,1)\). log is an alias of Log.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of log transform link function</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of log transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the log transform link function</td>
</tr>
</tbody>
</table>

\[ \text{statsmodels.genmod.families.links.Log_deriv} \]
\[ \text{Log.}^{-}\text{deriv}(p) \]
Derivative of log transform link function

Parameters:
\[ p : \text{array-like} \]
Mean parameters

Returns $g'(p)$: array

derivative of log transform of $x$

Notes

$g(x) = 1/x$

`statsmodels.genmod.families.links.Log.inverse`

_Log.inverse$(z)

Inverse of log transform link function

Parameters $z$: array

The inverse of the link function at $p$

Returns $p$: array

The mean probabilities given the value of the inverse $z$

Notes

$g^{-1}(z) = \exp(z)$

`statsmodels.genmod.families.links.Log.inverse_deriv`

_Log.inverse_deriv$(z)$

Derivative of the inverse of the log transform link function

Parameters $z$: array

The inverse of the link function at $p$

Returns The value of the derivative of the inverse of the logit function:

`statsmodels.genmod.families.links.Logit`

class `statsmodels.genmod.families.links.Logit`

The logit transform

Notes

call and derivative use a private method _clean to make trim $p$ by machine epsilon so that $p$ is in (0,1)

Alias of Logit: logit = Logit()

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deriv(p)</code></td>
<td>Derivative of the logit transform</td>
</tr>
<tr>
<td><code>inverse(z)</code></td>
<td>Inverse of the logit transform</td>
</tr>
<tr>
<td><code>inverse_deriv(z)</code></td>
<td>Derivative of the inverse of the logit transform</td>
</tr>
</tbody>
</table>
statsmodels.genmod.families.links.Logit.deriv

Logit.deriv(p)

Derivative of the logit transform

**Parameters** p: array-like:
- Probabilities

**Returns** g'(p): array
- Value of the derivative of logit transform at p

**Notes**

\[ g'(p) = \frac{1}{p \cdot (1 - p)} \]

Alias for Logit: logit = Logit()

statsmodels.genmod.families.links.Logit.inverse

Logit.inverse(z)

Inverse of the logit transform

**Parameters** z: array-like
- The value of the logit transform at p

**Returns** p: array
- Probabilities

**Notes**

\[ g^{-1}(z) = \frac{\exp(z)}{1 + \exp(z)} \]

statsmodels.genmod.families.links.Logit.inverse_deriv

Logit.inverse_deriv(z)

Derivative of the inverse of the logit transform

**Parameters** z: array-like
- z is usually the linear predictor for a GLM or GEE model.

**Returns** The value of the derivative of the inverse of the logit function:

statsmodels.genmod.families.links.NegativeBinomial

class statsmodels.genmod.families.links.NegativeBinomial(alpha=1.0)

The negative binomial link function

**Parameters** alpha: float, optional
- Alpha is the ancillary parameter of the Negative Binomial link function. It is assumed to be nonstochastic. The default value is 1. Permissible values are usually assumed to be in (.01,2).
Methods
### statsmodels.genmod.families.links.NegativeBinomial.deriv

NegativeBinomial.deriv(p)

Derivative of the negative binomial transform

**Parameters**

- **p**: array-like
  Mean parameters

**Returns**

- g'(p): array
  The derivative of the negative binomial transform link function

**Notes**

g'(x) = 1/(x+alpha*x^2)

### statsmodels.genmod.families.links.NegativeBinomial.inverse

NegativeBinomial.inverse(z)

Inverse of the negative binomial transform

**Parameters**

- **z**: array-like
  The value of the inverse of the negative binomial link at p.

**Returns**

- **p**: array
  Mean parameters

**Notes**

g^(-1)(z) = exp(z)/(alpha*(1-exp(z)))

### statsmodels.genmod.families.links.NegativeBinomial.inverse_deriv

NegativeBinomial.inverse_deriv(z)

Derivative of the inverse of the negative binomial transform

**Parameters**

- **z**: array-like
  Usually the linear predictor for a GLM or GEE model

**Returns**

The value of the inverse of the derivative of the negative binomial link:

### statsmodels.genmod.families.links.Power

class statsmodels.genmod.families.links.Power(power=1.0)

The power transform
Parameters `power` : float

The exponent of the power transform

Notes

Aliases of Power: inverse = Power(power=-1) sqrt = Power(power=0.5) inverse_squared = Power(power=-2.) identity = Power(power=1.)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of the power transform</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of the power transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the power transform</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.links.Power.deriv

`Power.deriv(p)`

Derivative of the power transform

Parameters `p` : array-like

Mean parameters

Returns `g'(p)` : array

Derivative of power transform of `p`

Notes

\[ g'(p) = power \times p^{power - 1} \]

statsmodels.genmod.families.links.Power.inverse

`Power.inverse(z)`

Inverse of the power transform link function

Parameters `z` : array-like

Value of the transformed mean parameters at `p`

Returns `p` : array

Mean parameters

Notes

\[ g^{-1}(z) = z^{1/power} \]

statsmodels.genmod.families.links.Power.inverse_deriv

`Power.inverse_deriv(z)`

Derivative of the inverse of the power transform

Parameters `z` : array-like
\( z \) is usually the linear predictor for a GLM or GEE model.

**Returns**  The value of the derivative of the inverse of the power transform function:

\[
\text{statsmodels.genmod.families.links.cauchy}
\]

**class** `statsmodels.genmod.families.links.cauchy`  
The Cauchy (standard Cauchy CDF) transform

**Notes**

\[ g(p) = \text{scipy.stats.cauchy.ppf}(p) \]

cauchy is an alias of CDFFlink with dbn=scipy.stats.cauchy

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv((p))</td>
<td>Derivative of CDF link</td>
</tr>
<tr>
<td>inverse((z))</td>
<td>The inverse of the CDF link</td>
</tr>
<tr>
<td>inverse_deriv((z))</td>
<td>Derivative of the inverse of the CDF transformation link function</td>
</tr>
</tbody>
</table>

**statsmodels.genmod.families.links.cauchy.deriv**  
\text{cauchy}.deriv\((p)\)

**Derivative of CDF link**

**Parameters**  \( p \) : array-like  
mean parameters

**Returns**  \( g'(p) \) : array  
The derivative of CDF transform at \( p \)

**Notes**

\[ g'(p) = 1./\text{dbn.pdf}(\text{dbn.ppf}(p)) \]

**statsmodels.genmod.families.links.cauchy.inverse**  
\text{cauchy}.inverse\((z)\)

**The inverse of the CDF link**

**Parameters**  \( z \) : array-like
The value of the inverse of the link function at \( p \)

**Returns**  \( p \) : array
Mean probabilities. The value of the inverse of CDF link of \( z \)

**Notes**

\[ g^\wedge(-1)(z) = \text{dbn.cdf}(z) \]
statsmodels.genmod.families.links.cauchy.inverse_deriv

\texttt{cauchy.inverse\_deriv}(z)

Derivative of the inverse of the CDF transformation link function

\textbf{Parameters} \texttt{z} : array

The inverse of the link function at \( p \)

\textbf{Returns} The value of the derivative of the inverse of the logit function:

statsmodels.genmod.families.links.CLogLog

class \texttt{statsmodels.genmod.families.links.CLogLog}

The complementary log-log transform

CLogLog inherits from Logit in order to have access to its \_clean method for the link and its derivative.

\textbf{Notes}

CLogLog is untested.

\textbf{Methods}

\begin{center}
\begin{tabular}{ll}
\texttt{deriv}(p) & Derivative of C-Log-Log transform link function \\
\texttt{inverse}(z) & Inverse of C-Log-Log transform link function \\
\texttt{inverse\_deriv}(z) & Derivative of the inverse of the C-Log-Log transform link function \\
\end{tabular}
\end{center}

statsmodels.genmod.families.links.CLogLog.deriv

\texttt{CLogLog.deriv}(p)

Derivative of C-Log-Log transform link function

\textbf{Parameters} \texttt{p} : array-like

Mean parameters

\textbf{Returns} \texttt{g'(p)} : array

The derivative of the CLogLog transform link function

\textbf{Notes}

\[ g'(p) = -\frac{1}{\log(p) \cdot p} \]

statsmodels.genmod.families.links.CLogLog.inverse

\texttt{CLogLog.inverse}(z)

Inverse of C-Log-Log transform link function

\textbf{Parameters} \texttt{z} : array-like

The value of the inverse of the CLogLog link function at \( p \)

\textbf{Returns} \texttt{p} : array

Mean parameters
Notes

g^(-1)(z) = 1-exp(-exp(z))

statsmodels.genmod.families.links.CLogLog.inverse_deriv
CLogLog_.inverse_deriv(z)
Derivative of the inverse of the C-Log-Log transform link function

Parameters  z : array-like

The value of the inverse of the CLogLog link function at p

Returns  The derivative of the inverse of the CLogLog link function :

statsmodels.genmod.families.links.identity

class  statsmodels.genmod.families.links.identity

The identity transform

Notes

g(p) = p

Alias of statsmodels.family.links.Power(power=1.)

Methods

| deriv(p) | Derivative of the power transform |
| inverse(z) | Inverse of the power transform link function |
| inverse_deriv(z) | Derivative of the inverse of the power transform |

statsmodels.genmod.families.links.identity.deriv

identity.deriv(p)
Derivative of the power transform

Parameters  p : array-like

Mean parameters

Returns  g'(p) : array

Derivative of power transform of p

Notes

g'(p) = power * p**(power - 1)

statsmodels.genmod.families.links.identity.inverse

identity.inverse(z)
Inverse of the power transform link function

Parameters  'z' : array-like
Value of the transformed mean parameters at $p$

Returns  
'p' : array
Mean parameters

Notes

g^(-1)(z') = z'^{(1/power)}

statsmodels.genmod.families.links.identity.inverse_deriv
identity.inverse_deriv(z)
Derivative of the inverse of the power transform

Parameters  
z : array-like

z is usually the linear predictor for a GLM or GEE model.

Returns  
The value of the derivative of the inverse of the power transform function :

statsmodels.genmod.families.links.inverse_power

class statsmodels.genmod.families.links.inverse_power
The inverse transform

Notes

g(p) = 1/p
Alias of statsmodels.family.links.Power(power=-1.)

Methods

derv(p)  Derivative of the power transform
inverse(z)  Inverse of the power transform link function
inverse_deriv(z)  Derivative of the inverse of the power transform

statsmodels.genmod.families.links.inverse_power.deriv

inverse_power.deriv(p)
Derivative of the power transform

Parameters  
p : array-like
Mean parameters

Returns  
g'(p) : array
Derivative of power transform of $p$

Notes

g'(p) = power * p**('power - 1)
statsmodels.genmod.families.links.inverse_power.inverse

inverse_power.inverse(z)
Inverse of the power transform link function

Parameters  `z` : array-like
Value of the transformed mean parameters at `p`

Returns  `p` : array
Mean parameters

Notes

\[ g^{-1}(z') = z'^{(1/power)} \]

statsmodels.genmod.families.links.inverse_power.inverse_deriv

inverse_power.inverse_deriv(z)
Derivative of the inverse of the power transform

Parameters  `z` : array-like
`z` is usually the linear predictor for a GLM or GEE model.

Returns  The value of the derivative of the inverse of the power transform function :

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of the power transform</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of the power transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the power transform</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.links.inverse_squared

class statsmodels.genmod.families.links.inverse_squared
The inverse squared transform

Notes

\[ g(p) = 1/(p^{**2}) \]
Alias of statsmodels.family.links.Power(power=2.)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of the power transform</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of the power transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the power transform</td>
</tr>
</tbody>
</table>

statsmodels.genmod.families.links.inverse_squared.deriv

inverse_squared.deriv(p)
Derivative of the power transform

Parameters  `p` : array-like
Mean parameters

Returns  `g'(p)` : array
Derivative of power transform of `p`
Notes

\[ g'(p) = power \times p^{\ast\ast\ast\ast(\text{power - 1})} \]

```
statsmodels.genmod.families.links.inverse_squared.inverse
inverse_squared.inverse(z)
```
Inverse of the power transform link function

- **Parameters**
  - `z` : array-like
    - Value of the transformed mean parameters at `p`
  - **Returns**
    - `p` : array
      - Mean parameters

Notes

\[ g^{-1}(z') = z'^{\ast\ast(1/power)} \]

```
statsmodels.genmod.families.links.inverse_squared.inverse_deriv
inverse_squared.inverse_deriv(z)
```
Derivative of the inverse of the power transform

- **Parameters**
  - `z` : array-like
    - `z` is usually the linear predictor for a GLM or GEE model.
  - **Returns**
    - The value of the derivative of the inverse of the power transform function:

```
statsmodels.genmod.families.links.Log
```

**class** `statsmodels.genmod.families.links.Log`

The log transform

Notes

call and derivative call a private method _clean to trim the data by machine epsilon so that `p` is in (0,1). log is an alias of Log.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriv(p)</td>
<td>Derivative of log transform link function</td>
</tr>
<tr>
<td>inverse(z)</td>
<td>Inverse of log transform link function</td>
</tr>
<tr>
<td>inverse_deriv(z)</td>
<td>Derivative of the inverse of the log transform link function</td>
</tr>
</tbody>
</table>

```
statsmodels.genmod.families.links.Log.deriv
Log.deriv(p)
```
Derivative of log transform link function

- **Parameters**
  - `p` : array-like
Mean parameters

**Returns** $g'(p):$ array

derivative of log transform of $x$

**Notes**

\[ g(x) = \frac{1}{x} \]

`statsmodels.genmod.families.links.Log.inverse`

`Log.inverse(z)`

Inverse of log transform link function

**Parameters** $z$: array

The inverse of the link function at $p$

**Returns** $p$: array

The mean probabilities given the value of the inverse $z$

**Notes**

\[ g^{-1}(z) = \exp(z) \]

`statsmodels.genmod.families.links.Log.inverse_deriv`

`Log.inverse_deriv(z)`

Derivative of the inverse of the log transform link function

**Parameters** $z$: array

The inverse of the link function at $p$

**Returns** The value of the derivative of the inverse of the logit function:

`statsmodels.genmod.families.links.Logit`

**class** `statsmodels.genmod.families.links.Logit`

The logit transform

**Notes**

call and derivative use a private method _clean to make trim $p$ by machine epsilon so that $p$ is in $(0,1)$

Alias of Logit: logit = Logit()

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deriv(p)</code></td>
<td>Derivative of the logit transform</td>
</tr>
<tr>
<td><code>inverse(z)</code></td>
<td>Inverse of the logit transform</td>
</tr>
<tr>
<td><code>inverse_deriv(z)</code></td>
<td>Derivative of the inverse of the logit transform</td>
</tr>
</tbody>
</table>
statsmodels.genmod.families.links.Logit.deriv

Logit.deriv(p)
Derivative of the logit transform

Parameters p: array-like:
Probabilities

Returns g'(p): array
Value of the derivative of logit transform at p

Notes

\[ g'(p) = \frac{1}{p \times (1 - p)} \]

Alias for Logit: logit = Logit()

statsmodels.genmod.families.links.Logit.inverse

Logit.inverse(z)
Inverse of the logit transform

Parameters z: array-like
The value of the logit transform at p

Returns p: array
Probabilities

Notes

\[ g^{-1}(z) = \frac{\exp(z)}{1+\exp(z)} \]

statsmodels.genmod.families.links.Logit.inverse_deriv

Logit.inverse_deriv(z)
Derivative of the inverse of the logit transform

Parameters z: array-like
\( z \) is usually the linear predictor for a GLM or GEE model.

Returns The value of the derivative of the inverse of the logit function:

statsmodels.genmod.families-links.nbinom

class statsmodels.genmod.families.links.nbinom(alpha=1.0)
The negative binomial link function.

Notes

\[ g(p) = \log(p/(p + 1/\alpha)) \]
nbinom is an alias of NegativeBinomial. nbinom = NegativeBinomial(alpha=1.)
Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>deriv(p)</strong></td>
<td>Derivative of the negative binomial transform</td>
</tr>
<tr>
<td><strong>inverse(z)</strong></td>
<td>Inverse of the negative binomial transform</td>
</tr>
<tr>
<td><strong>inverse_deriv(z)</strong></td>
<td>Derivative of the inverse of the negative binomial transform</td>
</tr>
</tbody>
</table>

**statsmodels.genmod.families.links.nbinom.deriv**

nbinom.deriv(p)

Derivative of the negative binomial transform

- **Parameters** `p`: array-like
  - Mean parameters
- **Returns** `g'(p)`: array
  - The derivative of the negative binomial transform link function

**Notes**

\[ g'(x) = \frac{1}{x+\alpha x^2} \]

**statsmodels.genmod.families.links.nbinom.inverse**

nbinom.inverse(z)

Inverse of the negative binomial transform

- **Parameters** `z`: array-like
  - The value of the inverse of the negative binomial link at `p`.
- **Returns** :
  - `p`: array
  - Mean parameters

**Notes**

\[ g^{-1}(z) = \frac{\exp(z)}{\alpha(1-\exp(z))} \]

**statsmodels.genmod.families.links.nbinom.inverse_deriv**

nbinom.inverse_deriv(z)

Derivative of the inverse of the negative binomial transform

- **Parameters** `z`: array-like
  - Usually the linear predictor for a GLM or GEE model
- **Returns** The value of the inverse of the derivative of the negative binomial link :
statsmodels.genmod.families.links.probit

class statsmodels.genmod.families.links.probit(dbn=<scipy.stats.distributions.norm_gen object at 0x0534E8D0>)

The probit (standard normal CDF) transform

Notes

g(p) = scipy.stats.norm.ppf(p)
probit is an alias of CDFLink.

Methods

| deriv(p) | Derivative of CDF link
| inverse(z) | The inverse of the CDF link
| inverse_deriv(z) | Derivative of the inverse of the CDF transformation link function

statsmodels.genmod.families.links.probit.deriv

probit.deriv(p)
Derivative of CDF link

Parameters p : array-like
mean parameters

Returns g'(p) : array
The derivative of CDF transform at p

Notes

g'(p) = 1./dbn.pdf(dbn.ppf(p))

statsmodels.genmod.families.links.probit.inverse

probit.inverse(z)
The inverse of the CDF link

Parameters z : array-like
The value of the inverse of the link function at p

Returns p : array
Mean probabilities. The value of the inverse of CDF link of z

Notes

g^-1(z) = dbn.cdf(z)
probit\_inverse\_deriv(z)

Derivative of the inverse of the CDF transformation link function

**Parameters**

- \( z \) : array
  
  The inverse of the link function at \( p \)

**Returns**

The value of the derivative of the inverse of the logit function:

### 3.4 Robust Linear Models

Robust linear models with support for the M-estimators listed under **Norms**.

See Module Reference for commands and arguments.

#### 3.4.1 Examples

```python
# Load modules and data
import statsmodels.api as sm
data = sm.datasets.stackloss.load()
data.exog = sm.add_constant(data.exog)

# Fit model and print summary
rlm_model = sm.RLM(data.endog, data.exog, M=sm.robust.norms.HuberT())
rlm_results = rlm_model.fit()
print rlm_results.params
```

Detailed examples can be found here:

#### 3.4.2 Technical Documentation

**Weight Functions**

Andrew’s Wave
Huber’s t
Least Squares
Ramsay's Ea
Trimmed Mean
Tukey’s Biweight
References


3.4.3 Module Reference

Model Classes

| RLM(endog, exog[, M, missing]) | Robust Linear Models |
Robust Linear Models

Estimate a robust linear model via iteratively reweighted least squares given a robust criterion estimator.

Parameters

- **endog**: array-like
  
  1-d endogenous response variable. The dependent variable.

- **exog**: array-like
  
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant`.

- **M**: statsmodels.robust.norms.RobustNorm, optional
  
  The robust criterion function for downweighting outliers. The current options are LeastSquares, HuberT, RamsayE, AndrewWave, TrimmedMean, Hampel, and Tukey-Biweight. The default is HuberT(). See `statsmodels.robust.norms` for more information.

- **missing**: str
  
  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none’.

Notes

Attributes

- **df_model** [float] The degrees of freedom of the model. The number of regressors p less one for the intercept. Note that the reported model degrees of freedom does not count the intercept as a regressor, though the model is assumed to have an intercept.

- **df_resid** [float] The residual degrees of freedom. The number of observations n less the number of regressors p. Note that here p does include the intercept as using a degree of freedom.

- **endog** [array] See above. Note that endog is a reference to the data so that if data is already an array and it is changed, then endog changes as well.

- **exog** [array] See above. Note that endog is a reference to the data so that if data is already an array and it is changed, then endog changes as well.

- **M** [statsmodels.robust.norms.RobustNorm] See above. Robust estimator instance instantiated.

- **nobs** [float] The number of observations n

- **pinv_wexog** [array] The pseudoinverse of the design / exogenous data array. Note that RLM has no whiten method, so this is just the pseudo inverse of the design.

- **normalized_cov_params** [array] The p x p normalized covariance of the design / exogenous data. This is approximately equal to (X.T * X)^(-1)
Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.stackloss.load()
>>> data.exog = sm.add_constant(data.exog)
>>> rlm_model = sm.RLM(data.endog, data.exog,
                      M=sm.robust.norms.HuberT())
>>> rlm_results = rlm_model.fit()
>>> rlm_results.params
array([ 0.82938433, 0.92606597, -0.12784672, -41.02649835])
>>> rlm_results.bse
array([ 0.11100521, 0.30293016, 0.12864961, 9.79189854])
>>> rlm_results_HC2 = rlm_model.fit(cov="H2")
>>> rlm_results_HC2.params
array([ 0.82938433, 0.92606597, -0.12784672, -41.02649835])
>>> rlm_results_HC2.bse
array([ 0.11945975, 0.32235497, 0.11796313, 9.08950419])
>>> rlm_hamp_hub = sm.RLM(data.endog, data.exog,
                         M=sm.robust.norms.Hampel()).fit(
                         sm.robust.scale.HuberScale())
>>> rlm_hamp_hub.params
array([ 0.73175452, 1.25082038, -0.14794399, -40.27122257])
```

Methods

- `deviance(tmp_results)`
  Returns the (unnormalized) log-likelihood from the M estimator.
- `fit([maxiter, tol, scale_est, init, cov, ...])`
  Fits the model using iteratively reweighted least squares.
- `from_formula(formula, data[, subset])`
  Create a Model from a formula and dataframe.
- `hessian(params)`
  The Hessian matrix of the model
- `information(params)`
- `initialize()`
  Initialize (possibly re-initialize) a Model instance. For
- `loglike(params)`
- `predict(params[, exog])`
  Return linear predicted values from a design matrix.
- `score(params)`

```
statsmodels.robust.robust_linear_model.RLM.deviance
RLM.deviance(tmp_results)
Returns the (unnormalized) log-likelihood from the M estimator.

statsmodels.robust.robust_linear_model.RLM.fit
RLM.fit(maxiter=50, tol=1e-08, scale_est='mad', init=None, cov='H1', update_scale=True, conv='dev')
Fits the model using iteratively reweighted least squares.

The IRLS routine runs until the specified objective converges to tol or maxiter has been reached.
```

Parameters **conv**: string
Indicates the convergence criteria. Available options are “coeffs” (the coefficients), “weights” (the weights in the iteration), “sresid” (the standardized residuals), and “dev” (the un-normalized log-likelihood for the M estimator). The default is “dev”.

330 Chapter 3. Table of Contents
cov : string, optional

‘H1’, ‘H2’, or ‘H3’ Indicates how the covariance matrix is estimated. Default is ‘H1’. See rlm.RLMResults for more information.

init : string

Specifies method for the initial estimates of the parameters. Default is None, which means that the least squares estimate is used. Currently it is the only available choice.

maxiter : int

The maximum number of iterations to try. Default is 50.

scale_est : string or HuberScale()

‘mad’ or HuberScale() Indicates the estimate to use for scaling the weights in the IRLS. The default is ‘mad’ (median absolute deviation. Other options are ‘HuberScale’ for Huber’s proposal 2. Huber’s proposal 2 has optional keyword arguments d, tol, and maxiter for specifying the tuning constant, the convergence tolerance, and the maximum number of iterations. See statsmodels.robust.scale for more information.

tol : float

The convergence tolerance of the estimate. Default is 1e-8.

update_scale : Bool

If update_scale is False then the scale estimate for the weights is held constant over the iteration. Otherwise, it is updated for each fit in the iteration. Default is True.

Returns results : object

statsmodels.rlm.RLMresults

---

3.4. Robust Linear Models 331
Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

`statsmodels.robust.robust_linear_model.RLM.hessian`

RLM.hessian (params)
The Hessian matrix of the model

`statsmodels.robust.robust_linear_model.RLM.information`

RLM.information (params)

`statsmodels.robust.robust_linear_model.RLM.initialize`

RLM.initialize()
Initialize (possibly re-initialize) a Model instance. For instance, the design matrix of a linear model may change and some things must be recomputed.

`statsmodels.robust.robust_linear_model.RLM.loglike`

RLM.loglike (params)

`statsmodels.robust.robust_linear_model.RLM.predict`

RLM.predict (params, exog=None)
Return linear predicted values from a design matrix.

Parameters

- params : array-like, optional after fit has been called
  Parameters of a linear model
- exog : array-like, optional.
  Design / exogenous data. Model exog is used if None.

Returns

An array of fitted values:

Notes

If the model as not yet been fit, params is not optional.

`statsmodels.robust.robust_linear_model.RLM.score`

RLM.score (params)

Attributes

- endog_names
- exog_names

Model Results
class *statsmodels.robust.robust_linear_model.RLMResults*(model, params, ...)  
Class to contain RLM results

Returns **Attributes**:

*bcov_scaled* : array  
\[ p \times p \] scaled covariance matrix specified in the model fit method. The default is H1. H1 is defined as 
\[ k^2 \times \left( \frac{1}{df_{\text{resid}}} \times \sum (M.\psi(sresid)^2) \times \text{scale}^2 \right) \] 
\[ \div \left( \frac{1}{nobs} \times \sum (M.\psi(deriv)(sresid))\right)^2 \times (X^T X)^{-1} \]  
where  
\[ k = 1 + \frac{(df_{\text{model}} + 1)}{nobs} \times \frac{\text{var}_psiprime}{m^2} \]  
where  
\[ m = \text{mean}(M.\psi(deriv)(sresid)) \]  
and  
\[ \text{var}_psiprime = \text{var}(M.\psi(deriv)(sresid)) \]  
H2 is defined as  
\[ k \times \left( \frac{1}{df_{\text{resid}}} \times \sum (M.\psi(sresid)^2) \times \text{scale}^2 \right) \] 
\[ \div \left( \frac{1}{nobs} \times \sum (M.\psi(deriv)(sresid))\right) \times W^{-1} \]  
H3 is defined as  
\[ \frac{1}{k} \times \left( \frac{1}{df_{\text{resid}}} \times \sum (M.\psi(sresid)^2) \times \text{scale}^2 \right) \] 
\[ \div \left( W^{-1} \times X^T X \times W^{-1} \right) \]  
where  
\[ k \] is defined as above and  
\[ W^{-1} = (M.\psi(deriv)(sresid) \times \text{exog.T exog})^{-1} \]  
See the technical documentation for cleaner formulae.

*bcov_unscaled* : array  
The usual \[ p \times p \] covariance matrix with scale set equal to 1. It is then just equivalent to normalized_cov_params.

*bse* : array  
An array of the standard errors of the parameters. The standard errors are taken from the robust covariance matrix specified in the argument to fit.

*chisq* : array  
An array of the chi-squared values of the parameter estimates.

*df_model* :  
See RLM.df_model

*df_resid* :  
See RLM.df_resid

*fit_history* : dict  
Contains information about the iterations. Its keys are deviance, params, iteration and the convergence criteria specified in RLM.fit, if different from deviance or params.

*fit_options* : dict  
Contains the options given to fit.

*fittedvalues* : array
The linear predicted values. dot(exog, params)

**model**: statsmodels.rlm.RLM

A reference to the model instance

**nobs**: float

The number of observations n

**normalized_cov_params**: array

See RLM.normalized_cov_params

**params**: array

The coefficients of the fitted model

**pinv_wexog**: array

See RLM.pinv_wexog

**pvalues**: array

The p values associated with *tvalues*. Note that *tvalues* are assumed to be distributed standard normal rather than Student’s t.

**resid**: array

The residuals of the fitted model. endog - fittedvalues

**scale**: float

The type of scale is determined in the arguments to the fit method in RLM. The reported scale is taken from the residuals of the weighted least squares in the last IRLS iteration if update_scale is True. If update_scale is False, then it is the scale given by the first OLS fit before the IRLS iterations.

**sresid**: array

The scaled residuals.

**tvalues**: array

The “t-statistics” of params. These are defined as params/bse where bse are taken from the robust covariance matrix specified in the argument to fit.

**weights**: array

The reported weights are determined by passing the scaled residuals from the last weighted least squares fit in the IRLS algorithm.

See also:

statsmodels.model.LikelihoodModelResults

---

**Methods**

- **bcov_scaled()**
- **bcov_unscaled()**
- **bse()**
- **chisq()**
- **conf_int([alpha, cols, method])**

Returns the confidence interval of the fitted parameters.
Table 3.96 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cov_params([r_matrix, column, scale, cov_p,...])</code></td>
<td>Returns the variance/covariance matrix.</td>
</tr>
<tr>
<td><code>f_test(r_matrix[, q_matrix, cov_p, scale,...])</code></td>
<td>Compute the F-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td><code>fittedvalues()</code></td>
<td></td>
</tr>
<tr>
<td><code>initialize(model, params, **kwd)</code></td>
<td></td>
</tr>
<tr>
<td><code>llf()</code></td>
<td></td>
</tr>
<tr>
<td><code>load(fname)</code></td>
<td>Load a pickle, (class method)</td>
</tr>
<tr>
<td><code>normalized_cov_params()</code></td>
<td></td>
</tr>
<tr>
<td><code>predict([exog, transform])</code></td>
<td>Call self.model.predict with self.params as the first argument.</td>
</tr>
<tr>
<td><code>pvalues()</code></td>
<td></td>
</tr>
<tr>
<td><code>remove_data()</code></td>
<td>Remove data arrays, all nobs arrays from result and model</td>
</tr>
<tr>
<td><code>resid()</code></td>
<td></td>
</tr>
<tr>
<td><code>save(fname[, remove_data])</code></td>
<td>Save a pickle of this instance</td>
</tr>
<tr>
<td><code>sresid()</code></td>
<td></td>
</tr>
<tr>
<td><code>summary([yname, xname, title, alpha, return_fmt])</code></td>
<td>This is for testing the new summary setup</td>
</tr>
<tr>
<td><code>summary2([xname, yname, title, alpha,...])</code></td>
<td>Experimental summary function for regression results</td>
</tr>
<tr>
<td><code>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</code></td>
<td>Compute a t-test for a joint linear hypothesis of the form Rb = q</td>
</tr>
<tr>
<td><code>tvalues()</code></td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td><code>wald_test(r_matrix[, q_matrix, cov_p,...])</code></td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td><code>weights()</code></td>
<td></td>
</tr>
</tbody>
</table>

#### statsmodels.robust.robust_linear_model.RLMResults.bcov_scaled

```
static RLMResults.bcov_scaled()
```

#### statsmodels.robust.robust_linear_model.RLMResults.bcov_unscaled

```
static RLMResults.bcov_unscaled()
```

#### statsmodels.robust.robust_linear_model.RLMResults.bse

```
static RLMResults.bse()
```

#### statsmodels.robust.robust_linear_model.RLMResults.chisq

```
static RLMResults.chisq()
```

#### statsmodels.robust.robust_linear_model.RLMResults.conf_int

```
RLMResults.conf_int(alpha=0.05, cols=None, method='default')
```

Returns the confidence interval of the fitted parameters.

**Parameters**

- `alpha` : float, optional
  
  The `alpha` level for the confidence interval. i.e., The default `alpha` = .05 returns a 95% confidence interval.

- `cols` : array-like, optional
  
  `cols` specifies which confidence intervals to return

- `method` : string
  
  Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

**Returns**

- `conf_int` : array
  
  Each row contains [lower, upper] confidence interval

### 3.4. Robust Linear Models

335
Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[-5496529.48322745, -1467987.78596704],
      [-177.02903529, 207.15277984],
      [-0.1115811 , 0.03994274],
      [-3.12506664, -0.91539297],
      [-1.5179487 , -0.54850503],
      [-0.56251721, 0.460309  ],
      [ 798.7875153 , 2859.51541392]])
>>> results.conf_int(cols=(2,3))
array([[ -0.1115811 ,  0.03994274],
      [ -3.12506664, -0.91539297]])
```

**statsmodels.robust.robust_linear_model.RLMResults.cov_params**

RLMResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters

- **r_matrix** : array-like
  
  Can be 1d, or 2d. Can be used alone or with other.

- **column** : array-like, optional
  
  Must be used on its own. Can be 0d or 1d see below.

- **scale** : float, optional
  
  Can be specified or not. Default is None, which means that the scale argument is taken from the model.

- **other** : array-like, optional
  
  Can be used when r_matrix is specified.

Returns (The below are assumed to be in matrix notation.) :

- **cov** : ndarray
  
  If no argument is specified returns the covariance matrix of a model :

  (scale)*(X.T X)^(-1) :

  If contrast is specified it pre and post-multiplies as follows :

  (scale) * r_matrix (X.T X)^(-1) r_matrix.T :
If contrast and other are specified returns:

(scale) * r_matrix (X.T X)^(-1) other.T:

If column is specified returns:

(scale) * (X.T X)^(-1)[column,column] if column is 0d:

OR:

(scale) * (X.T X)^(-1)[column][:,column] if column is 1d:

```
statsmodels.robust.robust_linear_model.RLMResults.f_test
RLMResults.f_test (r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
Compute the F-test for a joint linear hypothesis.
```

**Parameters**

- **r_matrix**: array-like, str, or tuple
  - array: An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like
  This is deprecated. See r_matrix and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p**: array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale**: float, optional
  Default is 1.0 for no scaling.

- **invcov**: array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**See also:**

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

**Notes**

The matrix r_matrix is assumed to be non-singular. More precisely,

r_matrix (pX pX.T) r_matrix.T

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**Examples**
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.

>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
 df_denom=9, df_num=6>

Compare this to

>>> results.F
330.2853392346658

>>> results.F_p
4.98403096572e-10

>>> B = np.array(((0, 0, 1, -1, 0, 0, 0), (0, 0, 0, 0, 0, 1, -1)))

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the
5th and 6th regressors are equal.

>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
 df_num=2>

Alternatively, you can specify the hypothesis tests using a string

>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)

statsmodels.robust.robust_linear_model.RLMResults.fittedvalues
static RLMResults.fittedvalues()

statsmodels.robust.robust_linear_model.RLMResults.initialize
RLMResults.initialize(model, params, **kwds)

statsmodels.robust.robust_linear_model.RLMResults.llf
static RLMResults.llf()

statsmodels.robust.robust_linear_model.RLMResults.load
classmethod RLMResults.load(fname)
load a pickle, (class method)

Parameters
fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

**Returns**  unpickled instance :

```python
statsmodels.robust.robust_linear_model.RLMResults.normalized_cov_params
```

```python
RLMResults.normalized_cov_params()
```

```python
statsmodels.robust.robust_linear_model.RLMResults.predict
```

```python
RLMResults.predict(exog=None, transform=True, *args, **kwargs)
```

Call self.model.predict with self.params as the first argument.

**Parameters**  exog : array-like, optional

The values for which you want to predict.

transform : bool, optional

If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model \( y \sim \log(x1) + \log(x2) \), and transform is True, then you can pass a data structure that contains \( x1 \) and \( x2 \) in their original form. Otherwise, you’d need to log the data first.

**Returns**  See self.model.predict :

```python
statsmodels.robust.robust_linear_model.RLMResults.pvalues
```

```python
static RLMResults.pvalues()
```

```python
statsmodels.robust.robust_linear_model.RLMResults.remove_data
```

```python
RLMResults.remove_data()
```

remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

```python
statsmodels.robust.robust_linear_model.RLMResults.resid
```

```python
static RLMResults.resid()
```

```python
statsmodels.robust.robust_linear_model.RLMResults.save
```

```python
RLMResults.save(fname, remove_data=False)
```

save a pickle of this instance

**Parameters**  fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.
remove_data : bool

If False (default), then the instance is pickled without changes. If True, then all arrays
with length nobs are set to None before pickling. See the remove_data method. In some
cases not all arrays will be set to None.

Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise
an exception.

statsmodels.robust.robust_linear_model.RLMResults.sresid

static RLMResults.sresid()

statsmodels.robust.robust_linear_model.RLMResults.summary

RLMResults.summary(yname=None, xname=None, title=0, alpha=0.05, return_fmt=’text’)

This is for testing the new summary setup

statsmodels.robust.robust_linear_model.RLMResults.summary2

RLMResults.summary2(xname=None, yname=None, title=None, alpha=0.05,
float_format=’%.4f’)

Experimental summary function for regression results

Parameters

xname : List of strings of length equal to the number of parameters
Names of the independent variables (optional)

yname : string
Name of the dependent variable (optional)

title : string, optional
Title for the top table. If not None, then this replaces the default title

alpha : float
significance level for the confidence intervals

float_format: string :
print format for floats in parameters summary

Returns

smry : Summary instance
this holds the summary tables and text, which can be printed or converted to various
output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.robust.robust_linear_model.RLMResults.t_test

RLMResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)
Compute a t-test for a joint linear hypothesis of the form Rb = q

Parameters

r_matrix : array-like, str, tuple
• array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
• str : The full hypotheses to test can be given as a string. See the examples.
• tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics

f_test for F tests

patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]

r tests that the coefficients on the 5th and 6th independent variable are the same.
```

```python
>>> T_Test = results.t_test(r) >>>print(T_Test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = ‘TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR’
>>> results = ols(formula, dta).fit()
>>> hypotheses = ‘GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1’
```
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)

statsmodels.robust.robust_linear_model.RLMResults.tvalues
static RLMResults.tvalues()
Return the t-statistic for a given parameter estimate.

statsmodels.robust.robust_linear_model.RLMResults.wald_test
RLMResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, use_f=None)
Compute a Wald-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
    • array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.
q_matrix : array-like
    This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.
cov_p : array-like, optional
    An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
scale : float, optional
    Default is 1.0 for no scaling.
invcov : array-like, optional
    A q x q array to specify an inverse covariance matrix based on a restrictions matrix.
use_f : bool
    If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,

r_matrix (pX pX.T) r_matrix.T

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
statsmodels.robust.robust_linear_model.RLMResults.weights

static RLMResults.weights()

Norms

<table>
<thead>
<tr>
<th>Norm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndrewWave([a])</td>
<td>Andrew’s wave for M estimation.</td>
</tr>
<tr>
<td>Hampel([a, b, c])</td>
<td>Hampel function for M-estimation.</td>
</tr>
<tr>
<td>HuberT([t])</td>
<td>Huber’s T for M estimation.</td>
</tr>
<tr>
<td>LeastSquares</td>
<td>Least squares rho for M-estimation and its derived functions.</td>
</tr>
<tr>
<td>RamsayE([a])</td>
<td>Ramsay’s Ea for M estimation.</td>
</tr>
<tr>
<td>RobustNorm</td>
<td>The parent class for the norms used for robust regression.</td>
</tr>
<tr>
<td>TrimmedMean([c])</td>
<td>Trimmed mean function for M-estimation.</td>
</tr>
<tr>
<td>TukeyBiweight([c])</td>
<td>Tukey’s biweight function for M-estimation.</td>
</tr>
</tbody>
</table>

**estimate_location**(a, scale[, norm, axis, ...])  
M-estimator of location using self.norm and a current estimator of scale.

**statsmodels.robust.norms.AndrewWave**

class statsmodels.robust.norms.AndrewWave(a=1.339)

Andrew’s wave for M estimation.

Parameters a : float, optional

The tuning constant for Andrew’s Wave function. The default value is 1.339.

See also:

statsmodels.robust.norms.RobustNorm

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>psi(z)</td>
<td>The psi function for Andrew’s wave</td>
</tr>
<tr>
<td>psi_deriv(z)</td>
<td>The derivative of Andrew’s wave psi function</td>
</tr>
<tr>
<td>rho(z)</td>
<td>The robust criterion function for Andrew’s wave.</td>
</tr>
<tr>
<td>weights(z)</td>
<td>Andrew’s wave weighting function for the IRLS algorithm</td>
</tr>
</tbody>
</table>

**statsmodels.robust.norms.AndrewWave.psi**

AndrewWave.psi(z)

The psi function for Andrew’s wave

The analytic derivative of rho

Parameters z : array-like

1d array

Returns psi : array

psi(z) = sin(z/a) for |z| <= a*pi
psi(z) = 0 for |z| > a*pi

**statsmodels.robust.norms.AndrewWave.psi_deriv**

AndrewWave.psi_deriv(z)

The derivative of Andrew’s wave psi function

3.4. Robust Linear Models
Notes

Used to estimate the robust covariance matrix.

```python
statsmodels.robust.norms.AndrewWave.rho
AndrewWave.rho(z)
The robust criterion function for Andrew’s wave.

Parameters z : array-like
    1d array

Returns rho : array
    rho(z) = a*(1-cos(z/a)) for |z| <= a*pi
    rho(z) = 2*a for |z| > a*pi
```

```python
statsmodels.robust.norms.AndrewWave.weights
AndrewWave.weights(z)
Andrew’s wave weighting function for the IRLS algorithm

The psi function scaled by z

Parameters z : array-like
    1d array

Returns weights : array
    weights(z) = sin(z/a)/(z/a) for |z| <= a*pi
    weights(z) = 0 for |z| > a*pi
```

```python
statsmodels.robust.norms.Hampel
class statsmodels.robust.norms.Hampel(a=2.0, b=4.0, c=8.0)
Hampel function for M-estimation.

Parameters a : float, optional
    b : float, optional
    c : float, optional

The tuning constants for Hampel’s function. The default values are a,b,c = 2, 4, 8.

See also:
statsmodels.robust.norms.RobustNorm

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>psi(z)</td>
<td>The psi function for Hampel’s estimator</td>
</tr>
<tr>
<td>psi_deriv(z)</td>
<td>The robust criterion function for Hampel’s estimator</td>
</tr>
<tr>
<td>rho(z)</td>
<td>The robust criterion function for Hampel’s estimator</td>
</tr>
<tr>
<td>weights(z)</td>
<td>Hampel weighting function for the IRLS algorithm</td>
</tr>
</tbody>
</table>
**statsmodels.robust.norms.Hampel.psi**

Hampel psi(z)
The psi function for Hampel’s estimator

The analytic derivative of rho

**Parameters** z : array-like

1d array

**Returns** psi : array

psi(z) = z for |z| <= a
psi(z) = a*sign(z) for a < |z| <= b
psi(z) = a*sign(z)*(c - |z|)/(c-b) for b < |z| <= c
psi(z) = 0 for |z| > c

**statsmodels.robust.norms.Hampel.psi_deriv**

Hampel psi_deriv(z)

**statsmodels.robust.norms.Hampel.rho**

Hampel rho(z)
The robust criterion function for Hampel’s estimator

**Parameters** z : array-like

1d array

**Returns** rho : array

rho(z) = (1/2.)*z**2 for |z| <= a
rho(z) = a*|z| - 1/2.*a**2 for a < |z| <= b
rho(z) = a*(c*|z|-(1/2.)*z**2)/(c-b) for b < |z| <= c
rho(z) = a*(b + c - a) for |z| > c

**statsmodels.robust.norms.Hampel.weights**

Hampel weights(z)
The Hampel weighting function for the IRLS algorithm

The psi function scaled by z

**Parameters** z : array-like

1d array

**Returns** weights : array

weights(z) = 1 for |z| <= a
weights(z) = a/|z| for a < |z| <= b
weights(z) = a*(c - |z|)/(|z|*(c-b)) for b < |z| <= c
weights(z) = 0 for |z| > c

3.4. Robust Linear Models
class statsmodels.robust.norms.HuberT

Huber’s T for M estimation.

Parameters
- t : float, optional
  The tuning constant for Huber’s t function. The default value is 1.345.

See also:
- statsmodels.robust.norms.RobustNorm

Methods

- psi(z)
  The psi function for Huber’s t estimator

- psi_deriv(z)
  The derivative of Huber’s t psi function

- rho(z)
  The robust criterion function for Huber’s t.

- weights(z)
  Huber’s t weighting function for the IRLS algorithm

statsmodels.robust.norms.HuberT.psi

HuberT.psi(z)

The psi function for Huber’s t estimator

Parameters
- z : array-like
  1d array

Returns
- psi : array
  psi(z) = z for |z| <= t
  psi(z) = sign(z)*t for |z| > t

statsmodels.robust.norms.HuberT.psi_deriv

HuberT.psi_deriv(z)

The derivative of Huber’s t psi function

Notes

Used to estimate the robust covariance matrix.

statsmodels.robust.norms.HuberT.rho

HuberT.rho(z)

The robust criterion function for Huber’s t.

Parameters
- z : array-like
  1d array

Returns
- rho : array
  rho(z) = .5*z**2 for |z| <= t
  rho(z) = |z|*t - .5*t**2 for |z| > t
Huber's t weighting function for the IRLS algorithm

The psi function scaled by $z$

**Parameters**
- $z$: array-like
  - 1d array

**Returns**
- weights: array
  - weights$(z) = 1$ for $|z| \leq t$
  - weights$(z) = t/|z|$ for $|z| > t$

---

**Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>psi$(z)$</td>
<td>The psi function for the least squares estimator</td>
</tr>
<tr>
<td>psi_deriv$(z)$</td>
<td>The derivative of the least squares psi function.</td>
</tr>
<tr>
<td>rho$(z)$</td>
<td>The least squares estimator rho function</td>
</tr>
<tr>
<td>weights$(z)$</td>
<td>The least squares estimator weighting function for the IRLS algorithm.</td>
</tr>
</tbody>
</table>

---

**Notes**

Used to estimate the robust covariance matrix.
statsmodels.robust.norms.LeastSquares.rho

LeastSquares.rho(z)

The least squares estimator rho function

Parameters  

z : array

1d array

Returns  

rho : array

\[ \rho(z) = \frac{1}{2} z^2 \]

statsmodels.robust.norms.LeastSquares.weights

LeastSquares.weights(z)

The least squares estimator weighting function for the IRLS algorithm.

The psi function scaled by the input z

Parameters  

z : array-like

1d array

Returns  

weights : array

\[ \text{weights}(z) = \text{np.ones}(z\text{.shape}) \]

statsmodels.robust.norms.RamsayE

class statsmodels.robust.norms.RamsayE(a=0.3)

Ramsay’s Ea for M estimation.

Parameters  

a : float, optional

The tuning constant for Ramsay’s Ea function. The default value is 0.3.

See also:  

statsmodels.robust.norms.RobustNorm

Methods

<table>
<thead>
<tr>
<th>psi(z)</th>
<th>The psi function for Ramsay’s Ea estimator</th>
</tr>
</thead>
<tbody>
<tr>
<td>psi_deriv(z)</td>
<td>The derivative of Ramsay’s Ea psi function.</td>
</tr>
<tr>
<td>rho(z)</td>
<td>The robust criterion function for Ramsay’s Ea.</td>
</tr>
<tr>
<td>weights(z)</td>
<td>Ramsay’s Ea weighting function for the IRLS algorithm</td>
</tr>
</tbody>
</table>

statsmodels.robust.norms.RamsayE.psi

RamsayE.psi(z)

The psi function for Ramsay’s Ea estimator

The analytic derivative of rho

Parameters  

z : array-like

1d array

Returns  

psi : array

\[ \psi(z) = z \exp(-a|z|) \]
statsmodels.robust.norms.RamsayE.psi_deriv

RamsayE.psi_deriv(z)

The derivative of Ramsay’s Ea psi function.

**Notes**

Used to estimate the robust covariance matrix.

statsmodels.robust.norms.RamsayE.rho

RamsayE.rho(z)

The robust criterion function for Ramsay’s Ea.

**Parameters**

- **z**: array-like
  - 1d array

**Returns**

- **rho**: array
  - rho(z) = a**-2 * (1 - exp(-a*|z|))*(1 + a*|z|))

statsmodels.robust.norms.RamsayE.weights

RamsayE.weights(z)

Ramsay’s Ea weighting function for the IRLS algorithm

The psi function scaled by z

**Parameters**

- **z**: array-like
  - 1d array

**Returns**

- **weights**: array
  - weights(z) = exp(-a*|z|)

statsmodels.robust.norms.RobustNorm

**class**

- **statsmodels.robust.norms.RobustNorm**

The parent class for the norms used for robust regression.

Lays out the methods expected of the robust norms to be used by statsmodels.RLM.

**Parameters**

- **None**:

  Some subclasses have optional tuning constants.

**See also**

- statsmodels.rlm, and

**Notes**

Currently only M-estimators are available.
References


Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>psi(z)</td>
<td>Derivative of rho.</td>
</tr>
<tr>
<td>psi_deriv(z)</td>
<td>Derivative of psi.</td>
</tr>
<tr>
<td>rho(z)</td>
<td>The robust criterion estimator function.</td>
</tr>
<tr>
<td>weights(z)</td>
<td>Returns the value of psi(z) / z</td>
</tr>
</tbody>
</table>

statsmodels.robust.norms.RobustNorm.psi

RobustNorm.psi(z)

Derivative of rho. Sometimes referred to as the influence function.

Abstract method:

psi = rho'

statsmodels.robust.norms.RobustNorm.psi_deriv

RobustNorm.psi_deriv(z)

Derivative of psi. Used to obtain robust covariance matrix.

See statsmodels.rlm for more information.

Abstract method:

psi_derive = psi'

statsmodels.robust.norms.RobustNorm.rho

RobustNorm.rho(z)

The robust criterion estimator function.

Abstract method:

-2 loglike used in M-estimator

statsmodels.robust.norms.RobustNorm.weights

RobustNorm.weights(z)

Returns the value of psi(z) / z

Abstract method:

psi(z) / z

statsmodels.robust.norms.TrimmedMean

class statsmodels.robust.norms.TrimmedMean(c=2.0)

Trimmed mean function for M-estimation.
Parameters  $c$: float, optional

The tuning constant for Ramsay’s Ea function. The default value is 2.0.

See also:

statsmodels.robust.norms.RobustNorm

Methods

<table>
<thead>
<tr>
<th>psi(z)</th>
<th>The psi function for least trimmed mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>psi_deriv(z)</td>
<td>The derivative of least trimmed mean psi function</td>
</tr>
<tr>
<td>rho(z)</td>
<td>The robust criterion function for least trimmed mean.</td>
</tr>
<tr>
<td>weights(z)</td>
<td>Least trimmed mean weighting function for the IRLS algorithm</td>
</tr>
</tbody>
</table>

statsmodels.robust.norms.TrimmedMean.psi

TrimmedMean.psi(z)

The psi function for least trimmed mean

The analytic derivative of rho

Parameters  $z$: array-like

1d array

Returns  psi: array

psi(z) = $z$ for $|z| \leq c$

psi(z) = 0 for $|z| > c$

statsmodels.robust.norms.TrimmedMean.psi_deriv

TrimmedMean.psi_deriv(z)

The derivative of least trimmed mean psi function

Notes

Used to estimate the robust covariance matrix.

statsmodels.robust.norms.TrimmedMean.rho

TrimmedMean.rho(z)

The robust criterion function for least trimmed mean.

Parameters  $z$: array-like

1d array

Returns  rho: array

rho(z) = $(1/2.)z^2$ for $|z| \leq c$

rho(z) = 0 for $|z| > c$
statsmodels.robust.norms.TrimmedMean.weights

`TrimmedMean.weights(z)`
Least trimmed mean weighting function for the IRLS algorithm

The psi function scaled by z

**Parameters**
- `z`: array-like 1d array

**Returns**
- `weights`: array
  - weights(z) = 1 for |z| <= c
  - weights(z) = 0 for |z| > c

---

statsmodels.robust.norms.TukeyBiweight

**class** statsmodels.robust.norms.TukeyBiweight (c=4.685)
Tukey’s biweight function for M-estimation.

**Parameters**
- `c`: float, optional
  The tuning constant for Tukey’s Biweight. The default value is c = 4.685.

**Notes**
Tukey’s biweight is sometime’s called bisquare.

**Methods**

| psi(z) | The psi function for Tukey’s biweight estimator |
| psi_deriv(z) | The derivative of Tukey’s biweight psi function |
| rho(z) | The robust criterion function for Tukey’s biweight estimator |
| weights(z) | Tukey’s biweight weighting function for the IRLS algorithm |

---

statsmodels.robust.norms.TukeyBiweight.psi

`TukeyBiweight.psi(z)`
The psi function for Tukey’s biweight estimator

The analytic derivative of rho

**Parameters**
- `z`: array-like 1d array

**Returns**
- `psi`: array
  - psi(z) = z*(1 - (z/c)**2)**2 for |z| <= R
  - psi(z) = 0 for |z| > R

---

statsmodels.robust.norms.TukeyBiweight.psi_deriv

`TukeyBiweight.psi_deriv(z)`
The derivative of Tukey’s biweight psi function
**Notes**

Used to estimate the robust covariance matrix.

```python
statsmodels.robust.norms.TukeyBiweight.rho
TukeyBiweight.rho(z)
The robust criterion function for Tukey’s biweight estimator

**Parameters**  
z : array-like
    1d array

**Returns**  
rho : array

\[
\rho(z) = -(1 - (z/c)^2)^3 * c^2/6 \text{ for } |z| \leq R
\]
\[
\rho(z) = 0 \text{ for } |z| > R
\]
```

```python
statsmodels.robust.norms.TukeyBiweight.weights
TukeyBiweight.weights(z)
Tukey's biweight weighting function for the IRLS algorithm

The psi function scaled by z

**Parameters**  
z : array-like
    1d array

**Returns**  
weights : array

\[
\psi(z) = (1 - (z/c)^2)^2 \text{ for } |z| \leq R
\]
\[
\psi(z) = 0 \text{ for } |z| > R
\]
```

```python
statsmodels.robust.norms.estimate_location
statsmodels.robust.norms.estimate_location(a, scale, norm=None, axis=0, initial=None, maxiter=30, tol=1e-06)
M-estimator of location using self.norm and a current estimator of scale.

This iteratively finds a solution to

\[
\text{norm.psi((a-mu)/scale).sum()} = 0
\]

**Parameters**  
a : array
    Array over which the location parameter is to be estimated

scale : array
    Scale parameter to be used in M-estimator

norm : RobustNorm, optional
    Robust norm used in the M-estimator. The default is HuberT().

axis : int, optional
    Axis along which to estimate the location parameter. The default is 0.

initial : array, optional
```
Initial condition for the location parameter. Default is None, which uses the median of a.

**niter**: int, optional

Maximum number of iterations. The default is 30.

**tol**: float, optional

Tolerance for convergence. The default is 1e-06.

**Returns** **mu**: array

Estimate of location

---

### Scale

#### statsmodels.robust.scale.Huber

**class** `statsmodels.robust.scale.Huber(c=1.5, tol=1e-08, maxiter=30, norm=None)`

Huber’s proposal 2 for estimating location and scale jointly.

**Parameters**

- **c**: float, optional

  Threshold used in threshold for $\chi=psi^2$. Default value is 1.5.

- **tol**: float, optional

  Tolerance for convergence. Default value is 1e-08.

- **maxiter**: int, optional

  Maximum number of iterations. Default value is 30.

- **norm**: statsmodels.robust.norms.RobustNorm, optional

  A robust norm used in M estimator of location. If None, the location estimator defaults to a one-step fixed point version of the M-estimator using Huber’s T.

**call**:

Return joint estimates of Huber’s scale and location.

#### Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> chem_data = np.array([2.20, 2.20, 2.4, 2.4, 2.5, 2.7, 2.8, 2.9, 3.03,
... 3.03, 3.10, 3.37, 3.4, 3.4, 3.5, 3.6, 3.7, 3.7, 3.7, 3.7, 3.77, 5.28, 28.95])
>>> sm.robust.scale.huber(chem_data)
(array(3.2054980819923693), array(0.67365260010478967))
```
Methods

statsmodels.robust.scale.HuberScale

class statsmodels.robust.scale.HuberScale(d=2.5, tol=1e-08, maxiter=30)
Huber’s scaling for fitting robust linear models.

Huber’s scale is intended to be used as the scale estimate in the IRLS algorithm and is slightly different than the Huber class.

Parameters

- **d**: float, optional
  - d is the tuning constant for Huber’s scale. Default is 2.5
- **tol**: float, optional
  - The convergence tolerance
- **maxiter**: int, optional
  - The maximum number of iterations. The default is 30.

Notes

Huber’s scale is the iterative solution to

\[
\text{scale}_{(i+1)}^2 = \frac{1}{(n-h)\sum(\chi(r/sigma_i)\times sigma_i^2)}
\]

where the Huber function is

\[
\chi(x) = \begin{cases} 
\frac{x^2}{2} & \text{for } |x| < d \\
\frac{d^2}{2} & \text{for } |x| \geq d
\end{cases}
\]

and the Huber constant 

\[
h = \frac{(n-p)h*(d**2 + (1-d**2)*\text{scipy.stats.norm.cdf}(d) - .5 - d*sqrt(2*pi)*exp(-0.5*d**2)}
\]

Methods

- **call**
  - Return’s Huber’s scale computed as below

Methods

statsmodels.robust.scale.mad

statsmodels.robust.scale.mad(a, c=0.6744897501960817, axis=0, center=<function median at 0x02ECA670>)
The Median Absolute Deviation along given axis of an array

Parameters

- **a**: array-like
  - Input array.
- **c**: float, optional
  - The normalization constant. Defined as scipy.stats.norm.ppf(3/4.), which is approxi-
mately .6745.

**axis**: int, optional

The default is 0. Can also be None.

**center**: callable or float

If a callable is provided, such as the default `np.median` then it is expected to be called `center(a)`. The axis argument will be applied via np.apply_over_axes. Otherwise, provide a float.

**Returns mad**: float

\[
mad = \frac{\text{median}(a - \text{center})}{c}
\]

---

### statsmodels.robust.scale.Huber

**class** `statsmodels.robust.scale.Huber` *(c=1.5, tol=1e-08, maxiter=30, norm=None)*

Huber’s proposal 2 for estimating location and scale jointly.

**Parameters**

- **c**: float, optional
  
  Threshold used in threshold for chi=\psi^2. Default value is 1.5.

- **tol**: float, optional
  
  Tolerance for convergence. Default value is 1e-08.

- **maxiter**: int, optional
  
  Maximum number of iterations. Default value is 30.

- **norm**: `statsmodels.robust.norms.RobustNorm`, optional
  
  A robust norm used in M estimator of location. If None, the location estimator defaults to a one-step fixed point version of the M-estimator using Huber’s T.

**call**: Return joint estimates of Huber’s scale and location.

---

### Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> chem_data = np.array([2.20, 2.20, 2.4, 2.4, 2.5, 2.7, 2.8, 2.9, 3.03, ...
... 3.03, 3.10, 3.37, 3.4, 3.4, 3.4, 3.5, 3.6, 3.7, 3.7, 3.7, 3.7, ...
... 3.77, 5.28, 28.95])
>>> sm.robust.scale.huber(chem_data)
(array(3.2054980819923693), array(0.67365260010478967))
```

---

### Methods
Huber's scaling for fitting robust linear models.

Huber’s scale is intended to be used as the scale estimate in the IRLS algorithm and is slightly different than the `Huber` class.

**Parameters**
- `d`: float, optional
  - `d` is the tuning constant for Huber’s scale. Default is 2.5
- `tol`: float, optional
  - The convergence tolerance
- `maxiter`: int, optional
  - The maximum number of iterations. The default is 30.

**Notes**

Huber’s scale is the iterative solution to

\[
scale_{(i+1)}**2 = 1/(n*h)*\sum(chi(r/sigma_i)*sigma_i**2)
\]

where the Huber function is

\[
chi(x) = \begin{cases} 
  (x**2)/2 & \text{for } |x| < d \\
  (d**2)/2 & \text{for } |x| \geq d 
\end{cases}
\]

and the Huber constant

\[
h = (n-p)/n*(d**2 + (1-d**2)* \text{scipy.stats.norm.cdf}(d) - .5 - d*sqrt(2*pi)*exp(-0.5*d**2))
\]

**Methods**

- `call`: Return’s Huber’s scale computed as below

3.5. Regression with Discrete Dependent Variable

Regression models for limited and qualitative dependent variables. The module currently allows the estimation of models with binary (Logit, Probit), nominal (MNLogit), or count (Poisson) data.

See Module Reference for commands and arguments.

3.5.1 Examples

```python
# Load the data from Spector and Mazzeo (1980)
spector_data = sm.datasets.spector.load()
spector_data.exog = sm.add_constant(spector_data.exog)

# Logit Model
```
logit_mod = sm.Logit(spector_data.endog, spector_data.exog)
logit_res = logit_mod.fit()
print logit_res.summary()

Detailed examples can be found here:

### 3.5.2 Technical Documentation

Currently all models are estimated by Maximum Likelihood and assume independently and identically distributed errors.

All discrete regression models define the same methods and follow the same structure, which is similar to the regression results but with some methods specific to discrete models. Additionally some of them contain additional model specific methods and attributes.

**References**

General references for this class of models are:


### 3.5.3 Module Reference

The specific model classes are:

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logit</td>
<td>Binary choice logit model</td>
</tr>
<tr>
<td>Probit</td>
<td>Binary choice Probit model</td>
</tr>
<tr>
<td>MNLogit</td>
<td>Multinomial logit model</td>
</tr>
<tr>
<td>Poisson</td>
<td>Poisson model for count data</td>
</tr>
<tr>
<td>NegativeBinomial</td>
<td>Negative Binomial Model for count data</td>
</tr>
</tbody>
</table>

**statsmodels.discrete.discrete_model.Logit**

**class** statsmodels.discrete.discrete_model.Logit( **endog, exog, **kwargs**)

Binary choice logit model

**Parameters**

- **endog** : array-like  
  1-d endogenous response variable. The dependent variable.

- **exog** : array-like  
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See statsmodels.tools.add_constant.

- **missing** : str
Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’

Attributes

<table>
<thead>
<tr>
<th>endog</th>
<th>array</th>
<th>A reference to the endogenous response variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>exog</td>
<td>array</td>
<td>A reference to the exogenous design.</td>
</tr>
</tbody>
</table>

Methods

- **cdf(X)**: The logistic cumulative distribution function
- **cov_params_func_l1(likelihood_model, xopt, ...)**: Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
- **fit([start_params, method, maxiter, ...])**: Fit the model using maximum likelihood.
- **fit_regularized([start_params, method, ...])**: Fit the model using a regularized maximum likelihood.
- **from_formula(formula, data[, subset])**: Create a Model from a formula and dataframe.
- **hessian(params)**: Logit model Hessian matrix of the log-likelihood
- **information(params)**: Fisher information matrix of model
- **initialize()**
- **jac(params)**: Logit model Jacobian of the log-likelihood for each observation
- **loglike(params)**: Log-likelihood of logit model.
- **loglikeobs(params)**: Log-likelihood of logit model for each observation.
- **pdf(X)**: The logistic probability density function
- **predict(params[, exog, linear])**: Predict response variable of a model given exogenous variables.
- **score(params)**: Logit model score (gradient) vector of the log-likelihood

**statsmodels.discrete.discrete_model.Logit.cdf**

Logit.cdf(X)
The logistic cumulative distribution function

**Parameters**

- **X**: array-like

  X is the linear predictor of the logit model. See notes.

**Returns**

1/(1 + exp(-X)) :

**Notes**

In the logit model,

\[
\Lambda(x'\beta) = \text{Prob}(Y = 1|x) = \frac{e^{x'\beta}}{1 + e^{x'\beta}}
\]

**statsmodels.discrete.discrete_model.Logit.cov_params_func_l1**

Logit.cov_params_func_l1(likelihood_model, xopt, retvals)
Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
Returns a full cov_params matrix, with entries corresponding to zero’d values set to np.nan.

**statsmodels.discrete.discrete_model.Logit.fit**

Logit.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

Fit method for likelihood based models

**Parameters**

**start_params** : array-like, optional

Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

**method** : str, optional

The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:

- ‘newton’ for Newton-Raphson, ‘nm’ for Nelder-Mead
- ‘bfgs’ for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
- ‘lbfgs’ for limited-memory BFGS with optional box constraints
- ‘powell’ for modified Powell’s method
- ‘cg’ for conjugate gradient
- ‘ncg’ for Newton-conjugate gradient
- ‘basinhopping’ for global basin-hopping solver

The explicit arguments in fit are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

**maxiter** : int, optional

The maximum number of iterations to perform.

**full_output** : bool, optional

Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

**disp** : bool, optional

Set to True to print convergence messages.

**fargs** : tuple, optional

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

**callback** : callable callback(xk), optional

Called after each iteration, as callback(xk), where xk is the current parameter vector.

**retall** : bool, optional
Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

Notes

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

'newton'
  tol : float
    Relative error in params acceptable for convergence.

'nm' -- Nelder Mead
  xtol : float
    Relative error in params acceptable for convergence
  ftol : float
    Relative error in loglike(params) acceptable for convergence
  maxfun : int
    Maximum number of function evaluations to make.

'bfgs'
  gtol : float
    Stop when norm of gradient is less than gtol.
  norm : float
    Order of norm (np.Inf is max, -np.Inf is min)
  epsilon
    If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

'lbfgs'
  m : int
    This many terms are used for the Hessian approximation.
  factr : float
    A stop condition that is a variant of relative error.
  pgtol : float
    A stop condition that uses the projected gradient.
  epsilon
    If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
  maxfun : int
    Maximum number of function evaluations to make.
  bounds : sequence
    (min, max) pairs for each element in x, defining the bounds on that parameter.
    Use None for one of min or max when there is no bound in that direction.

'cg'
  gtol : float
    Stop when norm of gradient is less than gtol.
  norm : float
    Order of norm (np.Inf is max, -np.Inf is min)
  epsilon
    If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if Likelihoodmodel.score is None.

'ncg'
  fhess_p : callable f’(x,•*args)
    Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if
avextol : float
Stop when the average relative error in the minimizer
falls below this amount.
epsilon : float or ndarray
If hess is approximated, use this value for the step size.
Only relevant if Likelihoodmodel.hessian is None.

'powell'
xtol : float
Line-search error tolerance
ftol : float
Relative error in loglike(params) for acceptable for
convergence.
maxfun : int
Maximum number of function evaluations to make.
start_direc : ndarray
Initial direction set.

'basinhopping'
niter : integer
The number of basin hopping iterations.
niter_success : integer
Stop the run if the global minimum candidate remains the
same for this number of iterations.
T : float
The "temperature" parameter for the accept or reject
criterion. Higher "temperatures" mean that larger jumps
in function value will be accepted. For best results
'T' should be comparable to the separation (in function
value) between local minima.
stepsize : float
Initial step size for use in the random displacement.
interval : integer
The interval for how often to update the 'stepsize'.

minimizer : dict
Extra keyword arguments to be passed to the minimizer
'scipy.optimize.minimize()', for example 'method' - the
minimization method (e.g. 'L-BFGS-B'), or 'tol' - the
tolerance for termination. Other arguments are mapped from
explicit argument of 'fit':
- 'args' <- 'fargs'
- 'jac' <- 'score'
- 'hess' <- 'hess'

statsmodels.discrete.discrete_model.Logit.fit_regularized

Logit.fit_regularized(start_params=None, method='l1', maxiter='defined_by_method',
full_output=1, disp=1, callback=None, alpha=0, trim_mode='auto',
auto_trim_tol=0.01, size_trim_tol=0.0001, qc_tol=0.03, **kwargs)
Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is
determined by the argument method.

Parameters start_params : array-like, optional
Initial guess of the solution for the loglikelihood maximization. The default is an array
of zeros.
method : ‘l1’ or ‘l1_cvxopt_cp’
See notes for details.

**maxiter** : Integer or ‘defined_by_method’

Maximum number of iterations to perform. If ‘defined_by_method’, then use method defaults (see notes).

**full_output** : bool

Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

**disp** : bool

Set to True to print convergence messages.

**fargs** : tuple

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

**callback** : callable callback(xk)

Called after each iteration, as callback(xk), where xk is the current parameter vector.

**retall** : bool

Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

**alpha** : non-negative scalar or numpy array (same size as parameters)

The weight multiplying the l1 penalty term

**trim_mode** : ‘auto’, ‘size’, or ‘off’

If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim params if they have very small absolute value

**size_trim_tol** : float or ‘auto’ (default = ‘auto’)

For use when trim_mode == ‘size’

**auto_trim_tol** : float

For sue when trim_mode == ‘auto’. Use

**qc_tol** : float

Print warning and don’t allow auto trim when (ii) (above) is violated by this much.

**qc_verbose** : Boolean

If true, print out a full QC report upon failure

**Notes**

Optional arguments for the solvers (available in Results.mle_settings):

‘l1’

**acc** : float (default: 1e-6)

Requested accuracy as used by slsqp

‘l1_cvxopt_cp’

**abstol** : float

absolute accuracy (default: 1e-7).
\[ \min_{\beta} L(\beta) + \sum_k \alpha_k |\beta_k| \]

via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the “added variables” \( u_k \))

\[ \min_{\beta, u} L(\beta) + \sum_k \alpha_k u_k, \]

subject to

\[ -u_k \leq \beta_k \leq u_k. \]

With \( \partial_k L \) the derivative of \( L \) in the \( k^{th} \) parameter direction, theory dictates that, at the minimum, exactly one of two conditions holds:

1. \( |\partial_k L| = \alpha_k \) and \( \beta_k \neq 0 \)
2. \( |\partial_k L| \leq \alpha_k \) and \( \beta_k = 0 \)

classmethod Logit.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

Parameters

- **formula**: str or generic Formula object
  The formula specifying the model
- **data**: array-like
  The data for the model. See Notes.
- **subset**: array-like
  An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
- **args**: extra arguments
  These are passed to the model
- **kwargs**: extra keyword arguments
  These are passed to the model.

Returns

- **model**: Model instance
Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

Statsmodels.discrete.discrete_model.Logit.hessian

Logit.hessian(params)
Logit model Hessian matrix of the log-likelihood

Parameters params : array-like
The parameters of the model

Returns hess : ndarray, (k_vars, k_vars)
The Hessian, second derivative of loglikelihood function, evaluated at params

Notes

\[ \frac{\partial^2 \ln L}{\partial \beta \partial \beta'} = - \sum_i \Lambda_i (1 - \Lambda_i) x_i x_i' \]

Statsmodels.discrete.discrete_model.Logit.information

Logit.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

Statsmodels.discrete.discrete_model.Logit.initialize

Logit.initialize()
Initialize is called by statsmodels.model.LikelihoodModel.__init__ and should contain any preprocessing that needs to be done for a model.

Statsmodels.discrete.discrete_model.Logit.jac

Logit.jac(params)
Logit model Jacobian of the log-likelihood for each observation

Parameters params : array-like :
The parameters of the model

Returns jac : ndarray, (nobs, k_vars)
The derivative of the loglikelihood for each observation evaluated at params.
Notes

\[
\frac{\partial \ln L_i}{\partial \beta} = (y_i - \Lambda_i) x_i
\]

for observations \(i = 1, \ldots, n\)

**statsmodels.discrete.discrete_model.Logit.loglike**

**Logit.loglike** *(params)*

Log-likelihood of logit model.

**Parameters** *params* : array-like

The parameters of the logit model.

**Returns** *loglike* : float

The log-likelihood function of the model evaluated at *params*. See notes.

**Notes**

\[
\ln L = \sum_i \ln \Lambda (q_i x'_i \beta)
\]

Where \(q = 2y - 1\). This simplification comes from the fact that the logistic distribution is symmetric.

**statsmodels.discrete.discrete_model.Logit.loglikeobs**

**Logit.loglikeobs** *(params)*

Log-likelihood of logit model for each observation.

**Parameters** *params* : array-like

The parameters of the logit model.

**Returns** *loglike* : ndarray (nobs,)

The log likelihood for each observation of the model evaluated at *params*. See Notes

**Notes**

\[
\ln L = \sum_i \ln \Lambda (q_i x'_i \beta)
\]

for observations \(i = 1, \ldots, n\)

where \(q = 2y - 1\). This simplification comes from the fact that the logistic distribution is symmetric.
Logit.pdf \( (X) \)
The logistic probability density function

- **Parameters** `X` : array-like
  
  \( X \) is the linear predictor of the logit model. See notes.

- **Returns** `pdf` : ndarray
  
  The value of the Logit probability mass function, PMF, for each point of \( X \).
  
  \[
  \text{np.exp}(-x)/(1+\text{np.exp}(-X))**2
  \]

**Notes**

In the logit model,

\[
\lambda(x' \beta) = \frac{e^{-x' \beta}}{(1 + e^{-x' \beta})^2}
\]

Logit.predict \( (\text{params, exog=None, linear=False}) \)

Predict response variable of a model given exogenous variables.

- **Parameters** `params` : array-like
  
  Fitted parameters of the model.

- `exog` : array-like
  
  1d or 2d array of exogenous values. If not supplied, the whole exog attribute of the model is used.

- `linear` : bool, optional
  
  If True, returns the linear predictor \( \text{dot}(\text{exog}, \text{params}) \). Else, returns the value of the cdf at the linear predictor.

- **Returns** `array` :
  
  Fitted values at exog.

Logit.score \( (\text{params}) \)

Logit model score (gradient) vector of the log-likelihood

- **Parameters** `params` : array-like
  
  The parameters of the model

- **Returns** `score` : ndarray, 1-D
  
  The score vector of the model, i.e. the first derivative of the loglikelihood function, evaluated at \( params \)
Notes

\[
\frac{\partial \ln L}{\partial \beta} = \sum_{i=1}^{n} (y_i - \Lambda_i) x_i
\]

Attributes

<table>
<thead>
<tr>
<th>endog_names</th>
<th>array</th>
<th>A reference to the endogenous response variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>exog_names</td>
<td>array</td>
<td>A reference to the exogenous design.</td>
</tr>
</tbody>
</table>

`statsmodels.discrete.discrete_model.Probit`

```python
class statsmodels.discrete.discrete_model.Probit(endog, exog, **kwargs)
```

Binary choice Probit model

**Parameters**
- `endog`: array-like
  - 1-d endogenous response variable. The dependent variable.
- `exog`: array-like
  - A `nobs` x `k` array where `nobs` is the number of observations and `k` is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant`.
- `missing`: str
  - Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’

**Attributes**

<table>
<thead>
<tr>
<th>endog</th>
<th>array</th>
<th>A reference to the endogenous response variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>exog</td>
<td>array</td>
<td>A reference to the exogenous design.</td>
</tr>
</tbody>
</table>

**Methods**

- `cdf(X)`
- `cov_params_func_l1(likelihood_model, xopt, ...)`
- `fit([start_params, method, maxiter, ...])`
- `fit_regularized([start_params, method, ...])`
- `from_formula(formula, data[, subset])`
- `hessian(params)`
- `information(params)`
- `initialize(params)`
- `jac(params)`

Probit (Normal) cumulative distribution function
Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
Fit the model using maximum likelihood.
Fit the model using a regularized maximum likelihood.
Create a Model from a formula and data frame.
Probit model Hessian matrix of the log-likelihood
Fisher information matrix of model
Initialize is called by
Probit model Jacobian for each observation
Table 3.113 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loglike</td>
<td>Log-likelihood of probit model (i.e., the normal distribution).</td>
</tr>
<tr>
<td>loglikeobs</td>
<td>Log-likelihood of probit model for each observation</td>
</tr>
<tr>
<td>pdf</td>
<td>Probit (Normal) probability density function</td>
</tr>
<tr>
<td>predict</td>
<td>Predict response variable of a model given exogenous variables.</td>
</tr>
<tr>
<td>score</td>
<td>Probit model score (gradient) vector</td>
</tr>
</tbody>
</table>

```
statsmodels.discrete.discrete_model.Probit.cdf

Probit.cdf(X)
Probit (Normal) cumulative distribution function

Parameters
X : array-like
The linear predictor of the model (XB).

Returns
cdf : ndarray
The cdf evaluated at X.

Notes
This function is just an alias for scipy.stats.norm.cdf
```

```
statsmodels.discrete.discrete_model.Probit.cov_params_func_l1

Probit.cov_params_func_l1(likelihood_model, xopt, retvals)
Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.

Returns
a full cov_params matrix, with entries corresponding to zero’d values set to np.nan.
```

```
statsmodels.discrete.discrete_model.Probit.fit

Probit.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)
Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

Fit method for likelihood based models

Parameters
start_params : array-like, optional
Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

method : str, optional
The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:
- ‘newton’ for Newton-Raphson, ‘nm’ for Nelder-Mead
- ‘bfgs’ for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
- ‘lbfgs’ for limited-memory BFGS with optional box constraints

3.5. Regression with Discrete Dependent Variable

369
• ‘powell’ for modified Powell’s method
• ‘cg’ for conjugate gradient
• ‘ncg’ for Newton-conjugate gradient
• ‘basinhopping’ for global basin-hopping solver

The explicit arguments in `fit` are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

maxiter : int, optional
The maximum number of iterations to perform.

full_output : bool, optional
Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

disp : bool, optional
Set to True to print convergence messages.

fargs : tuple, optional
Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

callback : callable callback(xk), optional
Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool, optional
Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

Notes

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

‘newton’
tol : float
Relative error in params acceptable for convergence.

‘nm’ -- Nelder Mead
xtol : float
Relative error in params acceptable for convergence
ftol : float
Relative error in loglike(params) acceptable for convergence
maxfun : int
Maximum number of function evaluations to make.

‘bfgs’
gtol : float
Stop when norm of gradient is less than gtol.
norm : float
Order of norm (np.Inf is max, -np.Inf is min)
epsilon

370 Chapter 3. Table of Contents
If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

'lbfgs'
 m : int
    This many terms are used for the Hessian approximation.
 factr : float
    A stop condition that is a variant of relative error.
 pgtol : float
    A stop condition that uses the projected gradient.
 epsilon
    If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
 maxfun : int
    Maximum number of function evaluations to make.
 bounds : sequence
    (min, max) pairs for each element in x,
    defining the bounds on that parameter.
    Use None for one of min or max when there is no bound in that direction.
 'cg'
 gtol : float
    Stop when norm of gradient is less than gtol.
 norm : float
    Order of norm (np.Inf is max, -np.Inf is min)
 epsilon : float
    If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if LikelihoodModel.score is None.
 'ncg'
 fhess_p : callable f'(x,*args)
    Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if LikelihoodModel.hessian is None.
 avextol : float
    Stop when the average relative error in the minimizer falls below this amount.
 epsilon : float or ndarray
    If fhess is approximated, use this value for the step size. Only relevant if LikelihoodModel.hessian is None.
 'powell'
 xtol : float
    Line-search error tolerance
 ftol : float
    Relative error in loglike(params) for acceptable for convergence.
 maxfun : int
    Maximum number of function evaluations to make.
 start_direc : ndarray
    Initial direction set.
 'basinhopping'
 niter : integer
    The number of basin hopping iterations.
 niter_success : integer
    Stop the run if the global minimum candidate remains the same for this number of iterations.
 T : float
    The "temperature" parameter for the accept or reject criterion. Higher "temperatures" mean that larger jumps
in function value will be accepted. For best results
'T' should be comparable to the separation (in function
value) between local minima.

stepsize : float
    Initial step size for use in the random displacement.
interval : integer
    The interval for how often to update the 'stepsize'.
minimizer : dict
    Extra keyword arguments to be passed to the minimizer
'scipy.optimize.minimize()', for example 'method' - the
minimization method (e.g. 'L-BFGS-B'), or 'tol' - the
tolerance for termination. Other arguments are mapped from
explicit argument of 'fit':
    - 'args' <= 'fargs'
    - 'jac' <= 'score'
    - 'hess' <= 'hess'

statsmodels.discrete.discrete_model.Probit.fit_regularized

Probit.fit_regularized(start_params=None, method='l1', maxiter='defined_by_method',
full_output=1, disp=1, callback=None, alpha=0, trim_mode='auto',
auto_trim_tol=0.01, size_trim_tol=0.0001, qc_tol=0.03, **kwargs)
Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is
determined by the argument method.

Parameters start_params : array-like, optional
    Initial guess of the solution for the loglikelihood maximization. The default is an array
    of zeros.

method : 'l1' or 'l1_cvxopt_cp'
    See notes for details.

maxiter : Integer or 'defined_by_method'
    Maximum number of iterations to perform. If 'defined_by_method', then use method
defaults (see notes).

full_output : bool
    Set to True to have all available output in the Results object’s mle_retvals attribute. The
    output is dependent on the solver. See LikelihoodModelResults notes section for more
    information.

disp : bool
    Set to True to print convergence messages.

fargs : tuple
    Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

callback : callable callback(xk)
    Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool
    Set to True to return list of solutions at each iteration. Available in Results object’s
    mle_retvals attribute.
alpha : non-negative scalar or numpy array (same size as parameters)  
The weight multiplying the l1 penalty term

trim_mode : ‘auto’, ‘size’, or ‘off’  
If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached  
the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim  
params if they have very small absolute value

size_trim_tol : float or ‘auto’ (default = ‘auto’)  
For use when trim_mode == ‘size’

auto_trim_tol : float  
For sue when trim_mode == ‘auto’. Use

qc_tol : float  
Print warning and don’t allow auto trim when (ii) (above) is violated by this much.

qc_verbose : Boolean  
If true, print out a full QC report upon failure

Notes

Optional arguments for the solvers (available in Results.mle_settings):

'11'
acc : float (default 1e-6)  
Requested accuracy as used by slsqp

'11_cvxopt_cp'
abstol : float  
absolute accuracy (default: 1e-7).
reltol : float  
relative accuracy (default: 1e-6).
feastol : float  
tolerance for feasibility conditions (default: 1e-7).
refinement : int  
number of iterative refinement steps when solving KKT  
equations (default: 1).

Optimization methodology

With $L$ the negative log likelihood, we solve the convex but non-smooth problem

$$
\min_{\beta} L(\beta) + \sum_k \alpha_k |\beta_k|
$$

via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the  
“added variables” $u_k$)

$$
\min_{\beta, u} L(\beta) + \sum_k \alpha_k u_k,
$$
subject to

\[-u_k \leq \beta_k \leq u_k.\]

With \(\partial_k L\) the derivative of \(L\) in the \(k^{th}\) parameter direction, theory dictates that, at the minimum, exactly one of two conditions holds:

1. \(|\partial_k L| = \alpha_k\) and \(\beta_k \neq 0\)
2. \(|\partial_k L| \leq \alpha_k\) and \(\beta_k = 0\)

**statsmodels.discrete.discrete_model.Probit.from_formula**

classmethod Probit.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

**Parameters**

- **formula**: str or generic Formula object
  - The formula specifying the model
- **data**: array-like
  - The data for the model. See Notes.
- **subset**: array-like
  - An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
- **args**: extra arguments
  - These are passed to the model
- **kwargs**: extra keyword arguments
  - These are passed to the model.

**Returns**

- **model**: Model instance

**Notes**

data must define \_getitem\_ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**statsmodels.discrete.discrete_model.Probit.hessian**

Probit.hessian(params)

Probit model Hessian matrix of the log-likelihood

**Parameters**

- **params**: array-like
  - The parameters of the model

**Returns**

- **hess**: ndarray, \((k_{\text{vars}}, k_{\text{vars}})\)
  - The Hessian, second derivative of loglikelihood function, evaluated at \(params\)
\[ \frac{\partial^2 \ln L}{\partial \beta \partial \beta'} = -\lambda_i (\lambda_i + x_i' \beta) x_i' \]

where

\[ \lambda_i = \frac{q_i \phi (q_i x_i' \beta)}{\Phi (q_i x_i' \beta)} \]

and \( q = 2y - 1 \)

**Notes**

\[ \frac{\partial \ln L_i}{\partial \beta} = \left[ \frac{q_i \phi (q_i x_i' \beta)}{\Phi (q_i x_i' \beta)} \right] x_i \]

for observations \( i = 1, \ldots, n \)

Where \( q = 2y - 1 \). This simplification comes from the fact that the normal distribution is symmetric.
statsmodels.discrete.discrete_model.Probit.loglike

Probit.loglike(params)
Log-likelihood of probit model (i.e., the normal distribution).

Parameters params : array-like
The parameters of the model.

Returns loglike : float
The log-likelihood function of the model evaluated at params. See notes.

Notes

\[ \ln L = \sum_i \ln \Phi (q_i x_i' \beta) \]

Where \( q = 2y - 1 \). This simplification comes from the fact that the normal distribution is symmetric.

statsmodels.discrete.discrete_model.Probit.loglikeobs

Probit.loglikeobs(params)
Log-likelihood of probit model for each observation

Parameters params : array-like
The parameters of the model.

Returns loglike : ndarray (nobs,)
The log likelihood for each observation of the model evaluated at params. See Notes

Notes

\[ \ln L_i = \ln \Phi (q_i x_i' \beta) \]

for observations \( i = 1, ..., n \)

where \( q = 2y - 1 \). This simplification comes from the fact that the normal distribution is symmetric.

statsmodels.discrete.discrete_model.Probit.pdf

Probit.pdf(X)
Probit (Normal) probability density function

Parameters X : array-like
The linear predictor of the model (XB).

Returns pdf : ndarray
The value of the normal density function for each point of X.
Notes

This function is just an alias for scipy.stats.norm.pdf

**statsmodels.discrete.discrete_model.Probit.predict**

`Probit.predict(params, exog=None, linear=False)`

Predict response variable of a model given exogenous variables.

**Parameters**

- `params` : array-like
  Fitted parameters of the model.

- `exog` : array-like
  1d or 2d array of exogenous values. If not supplied, the whole exog attribute of the model is used.

- `linear` : bool, optional
  If True, returns the linear predictor dot(exog,params). Else, returns the value of the cdf at the linear predictor.

**Returns**

- `array` : Fitted values at exog.

**statsmodels.discrete.discrete_model.Probit.score**

`Probit.score(params)`

Probit model score (gradient) vector

**Parameters**

- `params` : array-like
  The parameters of the model

**Returns**

- `score` : ndarray, 1-D
  The score vector of the model, i.e. the first derivative of the loglikelihood function, evaluated at `params`

Notes

\[
\frac{\partial \ln L}{\partial \beta} = \sum_{i=1}^{n} \left[ \frac{q_i \phi (q_i x_i' \beta)}{\Phi (q_i x_i' \beta)} \right] x_i
\]

Where \( q = 2y - 1 \). This simplification comes from the fact that the normal distribution is symmetric.

Attributes

- `endog_names`
- `exog_names`
**statsmodels.discrete.discrete_model.MNLogit**

**class** statsmodels.discrete.discrete_model.MNLogit(endo, exog, **kwargs)

Multinomial logit model

**Parameters**

endog : array-like

  endog is an 1-d vector of the endogenous response. endog can contain strings, ints, or floats. Note that if it contains strings, every distinct string will be a category. No stripping of whitespace is done.

exog : array-like

  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See statsmodels.tools.add_constant.

missing : str

  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none’.

**Notes**

See developer notes for further information on MNLogit internals.

**Attributes**

<table>
<thead>
<tr>
<th>endog</th>
<th>array</th>
<th>A reference to the endogenous response variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>exog</td>
<td>array</td>
<td>A reference to the exogenous design.</td>
</tr>
<tr>
<td>J</td>
<td>float</td>
<td>The number of choices for the endogenous variable. Note that this is zero-indexed.</td>
</tr>
<tr>
<td>K</td>
<td>float</td>
<td>The actual number of parameters for the exogenous design. Includes the constant if the design has one.</td>
</tr>
<tr>
<td>names</td>
<td>dict</td>
<td>A dictionary mapping the column number in endog to the variables in endog.</td>
</tr>
<tr>
<td>wendog</td>
<td>array</td>
<td>An n x j array where j is the number of unique categories in endog. Each column of j is a dummy variable indicating the category of each observation. See names for a dictionary mapping each column to its category.</td>
</tr>
</tbody>
</table>

**Methods**

cdf(X) Multinomial logit cumulative distribution function.
cov_params_func_l1(likelihood_model, xopt, ...) Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
fit([start_params, method, maxiter, ...]) Fit the model using maximum likelihood.
fit_regularized([start_params, method, ...]) Fit the model using a regularized maximum likelihood.
from_formula(formula, data[, subset]) Create a Model from a formula and dataframe.
hessian(params) Multinomial logit Hessian matrix of the log-likelihood.
information(params) Fisher information matrix of model.
initialize() Preprocesses the data for MNLogit.
jac(params) Jacobian matrix for multinomial logit model log-likelihood.
Table 3.115 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loglike(params)</td>
<td>Log-likelihood of the multinomial logit model.</td>
</tr>
<tr>
<td>loglike_and_score(params)</td>
<td>Returns log likelihood and score, efficiently reusing calculations.</td>
</tr>
<tr>
<td>loglikeobs(params)</td>
<td>Log-likelihood of the multinomial logit model for each observation.</td>
</tr>
<tr>
<td>pdf(eXB)</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>predict(params[, exog, linear])</td>
<td>Predict response variable of a model given exogenous variables.</td>
</tr>
<tr>
<td>score(params)</td>
<td>Score matrix for multinomial logit model log-likelihood</td>
</tr>
</tbody>
</table>

**statsmodels.discrete.discrete_model.MNLogit.cdf**

MNLogit.cdf(X)

Multinomial logit cumulative distribution function.

**Parameters**

- X : array
  - The linear predictor of the model XB.

**Returns**

- cdf : ndarray
  - The cdf evaluated at X.

**Notes**

In the multinomial logit model.

\[
\frac{\text{exp}(\beta_j^T x_i)}{\sum_{k=0}^{J} \text{exp}(\beta_k^T x_i)}
\]

**statsmodels.discrete.discrete_model.MNLogit.cov_params_func_l1**

MNLogit.cov_params_func_l1(likelihood_model, xopt, retvals)

Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.

Returns a full cov_params matrix, with entries corresponding to zero’d values set to np.nan.

**statsmodels.discrete.discrete_model.MNLogit.fit**

MNLogit.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

Fit method for likelihood based models

**Parameters**

- start_params : array-like, optional
  - Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

- method : str, optional
  - The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:
    - ‘newton’ for Newton-Raphson, ‘nm’ for Nelder-Mead
    - ‘bfgs’ for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
• ‘lbfgs’ for limited-memory BFGS with optional box constraints
• ‘powell’ for modified Powell’s method
• ‘cg’ for conjugate gradient
• ‘ncg’ for Newton-conjugate gradient
• ‘basinhopping’ for global basin-hopping solver

The explicit arguments in fit are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

maxiter : int, optional
The maximum number of iterations to perform.

full_output : bool, optional
Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

disp : bool, optional
Set to True to print convergence messages.

fargs : tuple, optional
Extra arguments passed to the likelihood function, i.e., loglike(x,*args)
callback : callable callback(xk), optional
Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool, optional
Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

Notes

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

‘newton’
  tol : float
    Relative error in params acceptable for convergence.
‘nm’ -- Nelder Mead
  xtol : float
    Relative error in params acceptable for convergence
  ftol : float
    Relative error in loglike(params) acceptable for convergence
  maxfun : int
    Maximum number of function evaluations to make.
‘bfgs’
  gtol : float
    Stop when norm of gradient is less than gtol.
  norm : float
Order of norm (np.Inf is max, -np.Inf is min)
epsilon
If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

'lbfgs'
m : int
This many terms are used for the Hessian approximation.
factr : float
A stop condition that is a variant of relative error.
pgtol : float
A stop condition that uses the projected gradient.
epsilon
If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
maxfun : int
Maximum number of function evaluations to make.
bounds : sequence
(min, max) pairs for each element in x, defining the bounds on that parameter. Use None for one of min or max when there is no bound in that direction.

'cg'
gtol : float
Stop when norm of gradient is less than gtol.
norm : float
Order of norm (np.Inf is max, -np.Inf is min)
epsilon : float
If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if LikelihoodModel.score is None.

'ncg'
fhess_p : callable f'(x,*args)
Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if LikelihoodModel.hessian is None.
avextol : float
Stop when the average relative error in the minimizer falls below this amount.
epsilon : float or ndarray
If fhess is approximated, use this value for the step size. Only relevant if LikelihoodModel.hessian is None.

'powell'
xtol : float
Line-search error tolerance
ftol : float
Relative error in loglike(params) for acceptable for convergence.
maxfun : int
Maximum number of function evaluations to make.
start_direc : ndarray
Initial direction set.

'basinhopping'
niter : integer
The number of basin hopping iterations.
niter_success : integer
Stop the run if the global minimum candidate remains the same for this number of iterations.
T : float
Statsmodels Documentation, Release 0.6.0

The "temperature" parameter for the accept or reject criterion. Higher "temperatures" mean that larger jumps in function value will be accepted. For best results 'T' should be comparable to the separation (in function value) between local minima.

stepsize : float
Initial step size for use in the random displacement.

interval : integer
The interval for how often to update the 'stepsize'.

minimizer : dict
Extra keyword arguments to be passed to the minimizer 'scipy.optimize.minimize()', for example 'method' - the minimization method (e.g. 'L-BFGS-B'), or 'tol' - the tolerance for termination. Other arguments are mapped from explicit argument of 'fit':
- 'args' <- 'fargs'
- 'jac' <- 'score'
- 'hess' <- 'hess'

statsmodels.discrete.discrete_model.MNLogit.fit_regularized

MNLogit.fit_regularized(start_params=None, method='l1', maxiter='defined_by_method',
full_output=1, disp=1, callback=None, alpha=0, trim_mode='auto',
auto_trim_tol=0.01, size_trim_tol=0.0001, qc_tol=0.03, **kwargs)

Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is determined by the argument method.

Parameters start_params : array-like, optional
Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

method : 'l1' or 'l1_cvxopt_cp'
See notes for details.

maxiter : Integer or 'defined_by_method'
Maximum number of iterations to perform. If 'defined_by_method', then use method defaults (see notes).

full_output : bool
Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

disp : bool
Set to True to print convergence messages.

fargs : tuple
Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

callback : callable callback(xk)
Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool
Set to True to return list of solutions at each iteration. Available in Results object’s
mle_retvals attribute.

**alpha** : non-negative scalar or numpy array (same size as parameters)

The weight multiplying the l1 penalty term

**trim_mode** : ‘auto’, ‘size’, or ‘off’

If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached
the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim
params if they have very small absolute value

**size_trim_tol** : float or ‘auto’ (default = ‘auto’)

For use when trim_mode == ‘size’

**auto_trim_tol** : float

For sue when trim_mode == ‘auto’. Use

**qc_tol** : float

Print warning and don’t allow auto trim when (ii) (above) is violated by this much.

**qc_verbose** : Boolean

If true, print out a full QC report upon failure

**Notes**

Optional arguments for the solvers (available in Results.mle_settings):

- ‘l1’
  - acc : float (default 1e-6)
    Requested accuracy as used by slsqp
- ‘l1_cvxopt_cp’
  - abstol : float
    absolute accuracy (default: 1e-7).
  - reltol : float
    relative accuracy (default: 1e-6).
  - feastol : float
    tolerance for feasibility conditions (default: 1e-7).
  - refinement : int
    number of iterative refinement steps when solving KKT
equations (default: 1).

**Optimization methodology**

With \( L \) the negative log likelihood, we solve the convex but non-smooth problem

\[
\min_{\beta} L(\beta) + \sum_{k} \alpha_{k}|\beta_{k}|
\]

via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the
“added variables” \( u_{k} \))

\[
\min_{\beta, u} L(\beta) + \sum_{k} \alpha_{k}u_{k},
\]
subject to

\[-u_k \leq \beta_k \leq u_k.\]

With $\partial_k L$ the derivative of $L$ in the $k^{th}$ parameter direction, theory dictates that, at the minimum, exactly one of two conditions holds:

1. $|\partial_k L| = \alpha_k$ and $\beta_k \neq 0$
2. $|\partial_k L| \leq \alpha_k$ and $\beta_k = 0$

**statsmodels.discrete.discrete_model.MNLogit.from_formula**

**classmethod MNLogit.from_formula**(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

**Parameters**

- **formula**: str or generic Formula object
  - The formula specifying the model
- **data**: array-like
  - The data for the model. See Notes.
- **subset**: array-like
  - An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
- **args**: extra arguments
  - These are passed to the model
- **kwargs**: extra keyword arguments
  - These are passed to the model.

**Returns**

- **model**: Model instance

**Notes**

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**statsmodels.discrete.discrete_model.MNLogit.hessian**

**MNLogit.hessian**(params)

Multinomial logit Hessian matrix of the log-likelihood

**Parameters**

- **params**: array-like
  - The parameters of the model

**Returns**

- **hess**: ndarray, (J*K, J*K)
  - The Hessian, second derivative of loglikelihood function with respect to the flattened parameters, evaluated at params
Notes

\[ \frac{\partial^2 \ln L}{\partial \beta_j \partial \beta_l} = - \sum_{i=1}^{n} \frac{\exp (\beta_j' x_i)}{ \sum_{k=0}^{J} \exp (\beta_k' x_i)} \left[ 1 \left( j = l \right) - \frac{\exp (\beta_l' x_i)}{ \sum_{k=0}^{J} \exp (\beta_k' x_i)} \right] x_i x_i' \]

where \( 1 \left( j = l \right) \) equals 1 if \( j = l \) and 0 otherwise.

The actual Hessian matrix has \( J^2 \times K \times K \) elements. Our Hessian is reshaped to be square \( (J \times K) \times (J \times K) \) so that the solvers can use it.

This implementation does not take advantage of the symmetry of the Hessian and could probably be refactored for speed.

```
statsmodels.discrete.discrete_model.MNLogit.information
```

```
MNLogit.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.
```

```
statsmodels.discrete.discrete_model.MNLogit.initialize
```

```
MNLogit.initialize()
Preprocesses the data for MNLogit.

Turns the endogenous variable into an array of dummies and assigns J and K.
```

```
statsmodels.discrete.discrete_model.MNLogit.jac
```

```
MNLogit.jac(params)
Jacobian matrix for multinomial logit model log-likelihood

Parameters params : array
The parameters of the multinomial logit model.

Returns jac : ndarray, (nobs, K*(J-1))
The derivative of the loglikelihood for each observation evaluated at params
```

Notes

\[ \frac{\partial \ln L_i}{\partial \beta_j} = \left( d_{ij} - \frac{\exp (\beta_j' x_i)}{ \sum_{k=0}^{J} \exp (\beta_k' x_i)} \right) x_i \]

for \( j = 1, \ldots, J \), for observations \( i = 1, \ldots, n \)

In the multinomial model the score vector is \( K \times (J-1) \) but is returned as a flattened array. The Jacobian has the observations in rows and the flattened array of derivatives in columns.
\texttt{statsmodels.discrete.discrete_model.MNLogit.loglike}

\texttt{MNLogit.loglike} \texttt{(params)}

Log-likelihood of the multinomial logit model.

\textbf{Parameters} \texttt{params} : array-like

The parameters of the multinomial logit model.

\textbf{Returns} \texttt{loglike} : float

The log-likelihood function of the model evaluated at \texttt{params}. See notes.

\textbf{Notes}

\[
\ln L = \sum_{i=1}^{n} \sum_{j=0}^{J} d_{ij} \ln \left( \frac{\exp (\beta_j' x_i)}{\sum_{k=0}^{J} \exp (\beta_k' x_i)} \right)
\]

where \(d_{ij} = 1\) if individual \(i\) chose alternative \(j\) and 0 if not.

\texttt{statsmodels.discrete.discrete_model.MNLogit.loglike_and_score}

\texttt{MNLogit.loglike_and_score} \texttt{(params)}

Returns log likelihood and score, efficiently reusing calculations.

Note that both of these returned quantities will need to be negated before being minimized by the maximum likelihood fitting machinery.

\texttt{statsmodels.discrete.discrete_model.MNLogit.loglikeobs}

\texttt{MNLogit.loglikeobs} \texttt{(params)}

Log-likelihood of the multinomial logit model for each observation.

\textbf{Parameters} \texttt{params} : array-like

The parameters of the multinomial logit model.

\textbf{Returns} \texttt{loglike} : ndarray (nobs,)

The log likelihood for each observation of the model evaluated at \texttt{params}. See Notes

\textbf{Notes}

\[
\ln L_i = \sum_{j=0}^{J} d_{ij} \ln \left( \frac{\exp (\beta_j' x_i)}{\sum_{k=0}^{J} \exp (\beta_k' x_i)} \right)
\]

for observations \(i = 1, \ldots, n\)

where \(d_{ij} = 1\) if individual \(i\) chose alternative \(j\) and 0 if not.
statsmodels.discrete.discrete_model.MNLogit

MNLogit.pdf (eXB)
NotImplemented

statsmodels.discrete.discrete_model.MNLogit.predict

MNLogit.predict (params, exog=None, linear=False)
Predict response variable of a model given exogenous variables.

Parameters params : array-like
2d array of fitted parameters of the model. Should be in the order returned from the model.

exog : array-like
1d or 2d array of exogenous values. If not supplied, the whole exog attribute of the model is used. If a 1d array is given it assumed to be 1 row of exogenous variables. If you only have one regressor and would like to do prediction, you must provide a 2d array with shape[1] == 1.

linear : bool, optional
If True, returns the linear predictor dot(exog,params). Else, returns the value of the cdf at the linear predictor.

Notes

Column 0 is the base case, the rest conform to the rows of params shifted up one for the base case.

statsmodels.discrete.discrete_model.MNLogit.score

MNLogit.score (params)
Score matrix for multinomial logit model log-likelihood

Parameters params : array
The parameters of the multinomial logit model.

Returns score : ndarray, (K * (J-1),)
The 2-d score vector, i.e. the first derivative of the loglikelihood function, of the multinomial logit model evaluated at params.

Notes

\[
\frac{\partial \ln L}{\partial \beta_j} = \sum_i \left( d_{ij} - \frac{\exp (\beta_j^\prime x_i)}{\sum_{k=0}^J \exp (\beta_k^\prime x_i)} \right) x_i
\]

for \( j = 1, ..., J \)
In the multinomial model the score matrix is K x J-1 but is returned as a flattened array to work with the solvers.

**Attributes**

```python
endog_names
exog_names
```

**statsmodels.discrete.discrete_model.Poisson**

**class** statsmodels.discrete.discrete_model.Poisson(endog, exog, offset=None, exposure=None, missing='none')

Poisson model for count data

**Parameters**

endog : array-like
1-d endogenous response variable. The dependent variable.

exog : array-like
A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See statsmodels.tools.add_constant.

missing : str
Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none’.

**Attributes**

<table>
<thead>
<tr>
<th>endog</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>exog</td>
<td>array</td>
</tr>
</tbody>
</table>

A reference to the endogenous response variable

A reference to the exogenous design.

**Methods**

- cdf(X) : Poisson model cumulative distribution function
- cov_params_func_l1(likelihood_model, xopt, ...) : Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
- fit([start_params, method, maxiter, ...]) : Fit the model using maximum likelihood.
- fit_regularized([start_params, method, ...]) : Fit the model using maximum likelihood.
- from_formula(formula, data[, subset]) : Create a Model from a formula and dataframe.
- hessian(params) : Poisson model Hessian matrix of the loglikelihood
- information(params) : Fisher information matrix of model
- initialize() : Initialize is called by
- jac(params) : Poisson model Jacobian of the log-likelihood
- loglike(params) : Loglikelihood of Poisson model
- loglikeobs(params) : Loglikelihood for observations of Poisson model
- pdf(X) : Poisson model probability mass function
- predict(params[, exog, exposure, offset, linear]) : Predict response variable of a count model given exogenous variables.
- score(params) : Poisson model score (gradient) vector of the log-likelihood
**statsmodels.discrete.discrete_model.Poisson.cdf**

Poisson.cdf(X)

Poisson model cumulative distribution function

**Parameters**

- **X**: array-like

  *X* is the linear predictor of the model. See notes.

**Returns**

The value of the Poisson CDF at each point.

**Notes**

The CDF is defined as

\[
\exp(-\lambda) \sum_{i=0}^{y} \frac{\lambda^i}{i!}
\]

where \(\lambda\) assumes the loglinear model. I.e.,

\[
\ln(\lambda_i) = X\beta
\]

The parameter \(X\) is \(X\beta\) in the above formula.

**statsmodels.discrete.discrete_model.Poisson.cov_params_func_l1**

Poisson.cov_params_func_l1(likelihood_model, xopt, retvals)

Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.

Returns a full cov_params matrix, with entries corresponding to zero’d values set to np.nan.

**statsmodels.discrete.discrete_model.Poisson.fit**

Poisson.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

Fit method for likelihood based models

**Parameters**

- **start_params**: array-like, optional

  Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

- **method**: str, optional

  The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:
  - 'newton' for Newton-Raphson, ‘nm’ for Nelder-Mead
  - 'bfgs' for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
• ‘lbfgs’ for limited-memory BFGS with optional box constraints
• ‘powell’ for modified Powell’s method
• ‘cg’ for conjugate gradient
• ‘ncg’ for Newton-conjugate gradient
• ‘basinhopping’ for global basin-hopping solver

The explicit arguments in `fit` are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

`maxiter` : int, optional

The maximum number of iterations to perform.

`full_output` : bool, optional

Set to True to have all available output in the Results object’s `mle_retvals` attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

`disp` : bool, optional

Set to True to print convergence messages.

`fargs` : tuple, optional

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

`callback` : callable callback(xk), optional

Called after each iteration, as callback(xk), where xk is the current parameter vector.

`retall` : bool, optional

Set to True to return list of solutions at each iteration. Available in Results object’s `mle_retvals` attribute.

Notes

The ‘basinhopping’ solver ignores `maxiter`, `retall`, `full_output` explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

`newton`

tol : float

Relative error in params acceptable for convergence.

`nm` -- Nelder Mead

xtol : float

Relative error in params acceptable for convergence.

ftol : float

Relative error in loglike(params) acceptable for convergence.

maxfun : int

Maximum number of function evaluations to make.

`bfgs`

gtol : float

Stop when norm of gradient is less than gtol.

norm : float
Order of norm (np.Inf is max, -np.Inf is min)

epsilon
If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

'lbfgs'
m : int
This many terms are used for the Hessian approximation.
factr : float
A stop condition that is a variant of relative error.
pgtol : float
A stop condition that uses the projected gradient.
epsilon
If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
maxfun : int
Maximum number of function evaluations to make.
bounds : sequence
(min, max) pairs for each element in x,
defining the bounds on that parameter.
Use None for one of min or max when there is no bound in that direction.

'cg'
gtol : float
Stop when norm of gradient is less than gtol.
norm : float
Order of norm (np.Inf is max, -np.Inf is min)
epsilon : float
If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if LikelihoodModel.score is None.

'ncg'
fhess_p : callable f'(x,*args)
Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if LikelihoodModel.hessian is None.
avextol : float
Stop when the average relative error in the minimizer falls below this amount.
epsilon : float or ndarray
If fhess is approximated, use this value for the step size. Only relevant if LikelihoodModel.hessian is None.

'powell'
xtol : float
Line-search error tolerance
ftol : float
Relative error in loglike(params) for acceptable for convergence.
maxfun : int
Maximum number of function evaluations to make.
start_direc : ndarray
Initial direction set.

'basinhopping'
niter : integer
The number of basin hopping iterations.
niter_success : integer
Stop the run if the global minimum candidate remains the same for this number of iterations.
T : float
The "temperature" parameter for the accept or reject criterion. Higher "temperatures" mean that larger jumps in function value will be accepted. For best results, 'T' should be comparable to the separation (in function value) between local minima.

**stepsize**: float
   Initial step size for use in the random displacement.

**interval**: integer
   The interval for how often to update the 'stepsize'.

**minimizer**: dict
   Extra keyword arguments to be passed to the minimizer 'scipy.optimize.minimize()', for example 'method' - the minimization method (e.g. 'L-BFGS-B'), or 'tol' - the tolerance for termination. Other arguments are mapped from explicit argument of 'fit':
   - 'args' <- 'fargs'
   - 'jac' <- 'score'
   - 'hess' <- 'hess'

```python
statsmodels.discrete.discrete_model.Poisson.fit_regularized
```

```python
Poisson.fit_regularized(start_params=None, method='l1', maxiter='defined_by_method', full_output=1, disp=1, callback=None, alpha=0, trim_mode='auto', auto_trim_tol=0.01, size_trim_tol=0.0001, qc_tol=0.03, **kwargs)
```

```python
statsmodels.discrete.discrete_model.Poisson.from_formula
```

```python
classmethod Poisson.from_formula(formula, data, subset=None, *args, **kwargs)
```

Create a Model from a formula and dataframe.

**Parameters**

- **formula**: str or generic Formula object
  The formula specifying the model

- **data**: array-like
  The data for the model. See Notes.

- **subset**: array-like
  An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

- **args**: extra arguments
  These are passed to the model

- **kwargs**: extra keyword arguments
  These are passed to the model.

**Returns**

- **model**: Model instance

**Notes**

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.
Poisson.hessian(params)
   Poisson model Hessian matrix of the loglikelihood
   Parameters params : array-like
      The parameters of the model
   Returns hess : ndarray, (k_vars, k_vars)
      The Hessian, second derivative of loglikelihood function, evaluated at params

Notes

\[
\frac{\partial^2 \ln L}{\partial \beta \partial \beta'} = -\sum_{i=1}^{n} \lambda_i x_i x_i'
\]

where the loglinear model is assumed

\[
\ln \lambda_i = x_i \beta
\]
Notes

\[
\frac{\partial \ln L_i}{\partial \beta} = (y_i - \lambda_i) x_i
\]

for observations \( i = 1, \ldots, n \)

where the loglinear model is assumed

\[
\ln \lambda_i = x_i \beta
\]

\textit{statsmodels.discrete.discrete_model.Poisson.loglike}

\texttt{Poisson.loglike(params)}

Loglikelihood of Poisson model

\textbf{Parameters} \texttt{params} : array-like

The parameters of the model.

\textbf{Returns} \texttt{loglike} : float

The log-likelihood function of the model evaluated at \texttt{params}. See notes.

Notes

\[
\ln L = \sum_{i=1}^{n} \left[ -\lambda_i + y_i x_i' \beta - \ln y_i! \right]
\]

\textit{statsmodels.discrete.discrete_model.Poisson.loglikeobs}

\texttt{Poisson.loglikeobs(params)}

Loglikelihood for observations of Poisson model

\textbf{Parameters} \texttt{params} : array-like

The parameters of the model.

\textbf{Returns} \texttt{loglike} : ndarray (nobs,)

The log likelihood for each observation of the model evaluated at \texttt{params}. See Notes

Notes

\[
\ln L_i = \left[ -\lambda_i + y_i x_i' \beta - \ln y_i! \right]
\]

for observations \( i = 1, \ldots, n \)
Poisson.\texttt{pdf}(X)

Poisson model probability mass function

**Parameters** X : array-like

X is the linear predictor of the model. See notes.

**Returns** pdf : ndarray

The value of the Poisson probability mass function, PMF, for each point of X.

**Notes**

The PMF is defined as

$$ e^{-\lambda_i} \frac{\lambda_i^{y_i}}{y_i!} $$

where $\lambda$ assumes the loglinear model. I.e.,

$$ \ln \lambda_i = x_i \beta $$

The parameter $X$ is $x_i \beta$ in the above formula.

Poisson.\texttt{predict}(params, exog=None, exposure=None, offset=None, linear=False)

Predict response variable of a count model given exogenous variables.

**Notes**

If exposure is specified, then it will be logged by the method. The user does not need to log it first.

Poisson.\texttt{score}(params)

Poisson model score (gradient) vector of the log-likelihood

**Parameters** params : array-like

The parameters of the model

**Returns** score : ndarray, 1-D

The score vector of the model, i.e. the first derivative of the loglikelihood function, evaluated at params
Notes

\[
\frac{\partial \ln L}{\partial \beta} = \sum_{i=1}^{n} (y_i - \lambda_i) x_i
\]

where the loglinear model is assumed

\[
\ln \lambda_i = x_i \beta
\]

Attributes

```python
endog_names
exog_names
```

**statsmodels.discrete.discrete_model.NegativeBinomial**

```python
class statsmodels.discrete.discrete_model.NegativeBinomial(endog, exog, loglike_method='nb2', offset=None, exposure=None, missing='none')
```

Negative Binomial Model for count data

**Parameters**

- **endog** : array-like
  
  1-d endogenous response variable. The dependent variable.

- **exog** : array-like
  
  A nobs x k array where nobs is the number of observations and k is the number of regressors. An intercept is not included by default and should be added by the user. See `statsmodels.tools.add_constant`.

- **loglike_method** : string
  
  Log-likelihood type. ‘nb2’, ‘nb1’, or ‘geometric’. Fitted value \( \mu \) Heterogeneity parameter \( \alpha \) nb2: Variance equal to \( \mu + \alpha \mu^2 \) (most common) nb1: Variance equal to \( \mu + \alpha \mu \) geometric: Variance equal to \( \mu + \mu^2 \)

- **missing** : str
  
  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’

**References**

References:

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>endog</td>
<td>array</td>
</tr>
<tr>
<td>exog</td>
<td>array</td>
</tr>
</tbody>
</table>

Methods

- **cdf**
  - The cumulative distribution function of the model.
- **cov_params_func_l1**
  - Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
- **fit**
  - Fit the model using a regularized maximum likelihood.
- **fit_regularized**
  - Fit the model using a regularized maximum likelihood.
- **from_formula**
  - Create a Model from a formula and dataframe.
- **hessian**
  - The Hessian matrix of the model.
- **information**
  - Fisher information matrix of model.
- **initialize**
  - Initialize is called by.
- **jac**
  - Score vector of model.
- **loglike**
  - Loglikelihood for negative binomial model.
- **pdf**
  - The probability density (mass) function of the model.
- **predict**
  - Predict response variable of a count model given exogenous variables.
- **score**
  - Score vector of model.

---

**statsmodels.discrete.discrete_model.NegativeBinomial.cdf**

NegativeBinomial.cdf(X)

The cumulative distribution function of the model.

**statsmodels.discrete.discrete_model.NegativeBinomial.cov_params_func_l1**

NegativeBinomial.cov_params_func_l1(likelihood_model, xopt, ...)

Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.

Returns a full cov_params matrix, with entries corresponding to zero’d values set to np.nan.

**statsmodels.discrete.discrete_model.NegativeBinomial.fit**

NegativeBinomial.fit(start_params=None, method=’bfgs’, maxiter=35, full_output=1, disp=1, callback=None, **kwargs)

**statsmodels.discrete.discrete_model.NegativeBinomial.fit_regularized**

NegativeBinomial.fit_regularized(start_params=None, method=’l1’, maxiter=’defined_by_method’, full_output=1, disp=1, callback=None, alpha=0, trim_mode=’auto’, auto_trim_tol=0.01, size_trim_tol=0.0001, qc_tol=0.03, **kwargs)

Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is

---

3.5. Regression with Discrete Dependent Variable
determined by the argument method.

**Parameters**

**start_params** : array-like, optional

Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

**method** : ‘l1’ or ‘l1_cvxopt_cp’

See notes for details.

**maxiter** : Integer or ‘defined_by_method’

Maximum number of iterations to perform. If ‘defined_by_method’, then use method defaults (see notes).

**full_output** : bool

Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

**disp** : bool

Set to True to print convergence messages.

**fargs** : tuple

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

**callback** : callable callback(xk)

Called after each iteration, as callback(xk), where xk is the current parameter vector.

**retall** : bool

Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

**alpha** : non-negative scalar or numpy array (same size as parameters)

The weight multiplying the l1 penalty term

**trim_mode** : ‘auto’, ‘size’, or ‘off’

If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim params if they have very small absolute value

**size_trim_tol** : float or ‘auto’ (default = ‘auto’)

For use when trim_mode == ’size’

**auto_trim_tol** : float

For sue when trim_mode == ’auto’. Use

**qc_tol** : float

Print warning and don’t allow auto trim when (ii) (above) is violated by this much.

**qc_verbose** : Boolean

If true, print out a full QC report upon failure
Optional arguments for the solvers (available in Results.mle_settings):

'\texttt{l1}'

\begin{verbatim}
    acc : float (default 1e-6)
    Requested accuracy as used by slsqp

    '\texttt{l1_cvxopt_cp}'
    abstol : float
    absolute accuracy (default: 1e-7).
    reltol : float
    relative accuracy (default: 1e-6).
    feastol : float
    tolerance for feasibility conditions (default: 1e-7).
    refinement : int
    number of iterative refinement steps when solving KKT equations (default: 1).
\end{verbatim}

Optimization methodology

With $L$ the negative log likelihood, we solve the convex but non-smooth problem

$$
\min_{\beta} L(\beta) + \sum_k \alpha_k |\beta_k|
$$

via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the “added variables” $u_k$)

$$
\min_{\beta,u} L(\beta) + \sum_k \alpha_k u_k,
$$

subject to

$$
-u_k \leq \beta_k \leq u_k.
$$

With $\partial_k L$ the derivative of $L$ in the $k^{th}$ parameter direction, theory dictates that, at the minimum, exactly one of two conditions holds:

1. $|\partial_k L| = \alpha_k$ and $\beta_k \neq 0$
2. $|\partial_k L| \leq \alpha_k$ and $\beta_k = 0$

\begin{verbatim}
statsmodels.discrete.discrete_model.NegativeBinomial.from_formula
\end{verbatim}

\texttt{classmethod NegativeBinomial.from_formula(formula, data, subset=None, \*args, **kwargs)}

Create a Model from a formula and dataframe.

**Parameters**

\begin{verbatim}
formula : str or generic Formula object
    The formula specifying the model

data : array-like
    The data for the model. See Notes.

subset : array-like
\end{verbatim}
An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

**args** : extra arguments
These are passed to the model

**kwargs** : extra keyword arguments
These are passed to the model.

**Returns** model : Model instance

**Notes**

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**statsmodels.discrete.discrete_model.NegativeBinomial.hessian**

NegativeBinomial.hessian(params)
The Hessian matrix of the model

**statsmodels.discrete.discrete_model.NegativeBinomial.information**

NegativeBinomial.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

**statsmodels.discrete.discrete_model.NegativeBinomial.initialize**

NegativeBinomial.initialize()
Initialize is called by statsmodels.model.LikelihoodModel.__init__ and should contain any preprocessing that needs to be done for a model.

**statsmodels.discrete.discrete_model.NegativeBinomial.jac**

NegativeBinomial.jac(params)

**statsmodels.discrete.discrete_model.NegativeBinomial.loglike**

NegativeBinomial.loglike(params)
Loglikelihood for negative binomial model

**Parameters** params : array-like
The parameters of the model. If loglike_method is nb1 or nb2, then the ancillary parameter is expected to be the last element.

**Returns** llf : float
The loglikelihood value at params
Notes

Following notation in Greene (2008), with negative binomial heterogeneity parameter $\alpha$:

$$\lambda_i = \exp(X\beta)$$
$$\theta = 1/\alpha$$
$$g_i = \theta \lambda_i^Q$$
$$w_i = g_i/(g_i + \lambda_i)$$
$$r_i = \theta/(\theta + \lambda_i)$$

$$\ln L_i = \ln \Gamma(y_i + g_i) - \ln \Gamma(1 + y_i) + g_i \ln(r_i) + y_i \ln(1 - r_i)$$

where $Q=0$ for NB2 and geometric and $Q = 1$ for NB1. For the geometric, $\alpha = 0$ as well.

statsmodels.discrete.discrete_model.NegativeBinomial.pdf

NegativeBinomial.pdf(X)
The probability density (mass) function of the model.

statsmodels.discrete.discrete_model.NegativeBinomial.predict

NegativeBinomial.predict(params, exog=None, exposure=None, offset=None, linear=False)
Predict response variable of a count model given exogenous variables.

Notes

If exposure is specified, then it will be logged by the method. The user does not need to log it first.

statsmodels.discrete.discrete_model.NegativeBinomial.score

NegativeBinomial.score(params)
Score vector of model.
The gradient of logL with respect to each parameter.

Attributes

<table>
<thead>
<tr>
<th>endog_names</th>
<th>exog_names</th>
</tr>
</thead>
</table>

The specific result classes are:

- LogitResults(model, mlefit) A results class for Logit Model
- ProbitResults(model, mlefit) A results class for Probit Model
- CountResults(model, mlefit) A results class for count data
- MultinomialResults(model, mlefit) A results class for multinomial data
- NegativeBinomialAncillaryResults(model, mlefit) A results class for NegativeBinomial 1 and 2

3.5. Regression with Discrete Dependent Variable 401
statsmodels.discrete.discrete_model.LogitResults

class statsmodels.discrete.discrete_model.LogitResults(model, mlefit)
A results class for Logit Model

Parameters model : A DiscreteModel instance

    params : array-like
              The parameters of a fitted model.

    hessian : array-like
              The hessian of the fitted model.

    scale : float
              A scale parameter for the covariance matrix.

Returns *Attributes* :

    aic : float
          Akaike information criterion. -2*(llf - p) where p is the number of regressors including
          the intercept.

    bic : float
          Bayesian information criterion. -2*llf + ln(nobs)*p where p is the number of regressors
          including the intercept.

    bse : array
          The standard errors of the coefficients.

    df_resid : float
               See model definition.

    df_model : float
               See model definition.

    fitted_values : array
                    Linear predictor XB.

    llf : float
          Value of the loglikelihood

    llnull : float
              Value of the constant-only loglikelihood

    llr : float
          Likelihood ratio chi-squared statistic; -2*(llnull - llf)

    llr_pvalue : float
                  The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr
                  has a chi-squared distribution with degrees of freedom df_model.

    prsquared : float
                  McFadden’s pseudo-R-squared. 1 - (llf/llnull)
Methods

\begin{itemize}
  \item \texttt{aic()} \quad \text{Returns the Akaike Information Criterion.}
  \item \texttt{bic()} \quad \text{Returns the Bayesian Information Criterion.}
  \item \texttt{bse()} \quad \text{Returns the standard error of the fitted parameters.}
  \item \texttt{conf_int([alpha, cols, method])} \quad \text{Returns the confidence interval of the fitted parameters.}
  \item \texttt{cov_params([r_matrix, column, scale, cov_p, ...])} \quad \text{Returns the variance/covariance matrix.}
  \item \texttt{f_test(r_matrix[, q_matrix, cov_p, scale, ...])} \quad \text{Compute the F-test for a joint linear hypothesis.}
  \item \texttt{fittedvalues()} \quad \text{Get marginal effects of the fitted model.}
  \item \texttt{get_margeff([at, method, atexog, dummy, count])} \quad \text{Get marginal effects of the fitted model.}
  \item \texttt{initialize(model, params, **kwds)} \quad \text{Load model instance.}
  \item \texttt{llf()} \quad \text{Calculates the log-likelihood.}
  \item \texttt{lnull()} \quad \text{Calculates the log-likelihood of the null model.}
  \item \texttt{llr()} \quad \text{Calculates the likelihood ratio statistic.}
  \item \texttt{llr_pvalue()} \quad \text{Calculates the p-value of the likelihood ratio test.}
  \item \texttt{load(fname)} \quad \text{Load a pickle, (class method).}
  \item \texttt{margeff([at, method, atexog, dummy, count])} \quad \text{Get marginal effects of the fitted model.}
  \item \texttt{normalized_cov_params()} \quad \text{Returns the normalized covariance matrix.}
  \item \texttt{pred_table([threshold])} \quad \text{Prediction table}
  \item \texttt{predict([exog, transform])} \quad \text{Call self.model.predict with self.params as the first argument.}
  \item \texttt{prsquared()} \quad \text{Calculates the proportion of variance explained.}
  \item \texttt{pvalues()} \quad \text{Calculates the p-values of the fitted parameters.}
  \item \texttt{remove_data()} \quad \text{Remove data arrays, all nobs arrays from result and model.}
  \item \texttt{resid()} \quad \text{Deviance residuals.}
  \item \texttt{resid_dev()} \quad \text{Generalized residuals.}
  \item \texttt{resid_generalized()} \quad \text{Pearson residuals.}
  \item \texttt{resid_pearson()} \quad \text{The response residuals.}
  \item \texttt{resid_response()} \quad \text{save(fname[, remove_data])} \quad \text{Save a pickle of this instance.}
  \item \texttt{summary([yname, xname, title, alpha, yname_list])} \quad \text{Summarize the Regression Results.}
  \item \texttt{summary2([yname, xname, title, alpha, ...])} \quad \text{Experimental function to summarize regression results.}
  \item \texttt{t_test(r_matrix[, q_matrix, cov_p, scale, use_t])} \quad \text{Compute a t-test for a joint linear hypothesis of the form Rb = q.}
  \item \texttt{tvalues()} \quad \text{Return the t-statistic for a given parameter estimate.}
  \item \texttt{wald_test(r_matrix[, q_matrix, cov_p, ...])} \quad \text{Compute a Wald-test for a joint linear hypothesis.}
\end{itemize}
statsmodels.discrete.discrete_model.LogitResults.conf_int

LogitResults.conf_int(alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

Parameters alpha : float, optional

The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

cols : array-like, optional

cols specifies which confidence intervals to return

method : string

Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

Returns conf_int : array

Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[5496529.48322745, -1467987.78596704],
[ -177.02903529, 207.15277984],
[ -0.1115811 , 0.03994274],
[ -3.12506664, -0.91539297],
[ -1.5179487 , -0.54850503],
[ -0.56251721, 0.460309 ],
[ 798.7875153 , 2859.51541392]])
```

```python
>>> results.conf_int(cols=(2,3))
array([[-0.1115811 , 0.03994274],
[-3.12506664, -0.91539297]])
```

statsmodels.discrete.discrete_model.LogitResults.cov_params

LogitResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.
Parameters  

- **r_matrix**: array-like
  
  Can be 1d, or 2d. Can be used alone or with other.

- **column**: array-like, optional
  
  Must be used on its own. Can be 0d or 1d see below.

- **scale**: float, optional
  
  Can be specified or not. Default is None, which means that the scale argument is taken from the model.

- **other**: array-like, optional
  
  Can be used when r_matrix is specified.

Returns  
*(The below are assumed to be in matrix notation.)*  

- **cov**: ndarray
  
  If no argument is specified returns the covariance matrix of a model:
  
  $(\text{scale}) * (X.T \ X)^{(-1)}$

  If contrast is specified it pre and post-multiplies as follows:
  
  $(\text{scale}) * r\_matrix \ (X.T \ X)^{(-1)} \ r\_matrix.T$

  If contrast and other are specified returns:
  
  $(\text{scale}) * r\_matrix \ (X.T \ X)^{(-1)} \ \text{other.T}$

  If column is specified returns:
  
  $(\text{scale}) * (X.T \ X)^{(-1)}[\text{column},\text{column}]$ if column is 0d

  OR:

  $(\text{scale}) * (X.T \ X)^{(-1)}[;\text{column}]$ if column is 1d

**statsmodels.discrete.discrete_model.LogitResults.f_test**

LogitResults.f_test  (*r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None*)

Compute the F-test for a joint linear hypothesis.

Parameters  

- **r_matrix**: array-like, str, or tuple
  
  - array: An r x k array where r is the number of restrictions to test and k is the number of regressors.
  
  - str: The full hypotheses to test can be given as a string. See the examples.
  
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like
  
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p**: array-like, optional
  
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale**: float, optional

3.5. Regression with Discrete Dependent Variable  405
Default is 1.0 for no scaling.

**invcov** : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts.statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test,patsy.DesignInfo.linear_constraint

Notes

The matrix $r_{matrix}$ is assumed to be non-singular. More precisely,

$r_{matrix} (pX pX.T) r_{matrix}.T$

is assumed invertible. Here, $pX$ is the generalized inverse of the design matrix of the model. There can be
problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.

```print(results.f_test(A))```

F contrast: F=330.2853392346658, p=4.98403096572e-10,
df_denom=9, df_num=6>

Compare this to

```results.F
330.2853392346658
```results.F_p
4.98403096572e-10

```B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the
5th and 6th regressors are equal.

```print(results.f_test(B)) ```

F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
df_num=2>

Alternatively, you can specify the hypothesis tests using a string

```from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> data = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, data).fit()```
```python
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```
static LogitResults.fittedvalues()

LogitResults.get_margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)
```

Get marginal effects of the fitted model.

**Parameters**

- **at**: str, optional
  - Options are:
    - 'overall', The average of the marginal effects at each observation.
    - 'mean', The marginal effects at the mean of each regressor.
    - 'median', The marginal effects at the median of each regressor.
    - 'zero', The marginal effects at zero for each regressor.
    - 'all', The marginal effects at each observation. If at is all only margeff will be available from the returned object.
  
  Note that if exog is specified, then marginal effects for all variables not specified by exog are calculated using the at option.

- **method**: str, optional
  - Options are:
    - 'dydx' - dy/dx - No transformation is made and marginal effects are returned. This is the default.
    - 'eyex' - estimate elasticities of variables in exog \( \frac{d(\ln y)}{d(\ln x)} \)
    - 'dyex' - estimate semielasticity \( \frac{dy}{d(\ln x)} \)
    - 'eydx' - estimate semeilasticity \( \frac{d(\ln y)}{dx} \)
  
  Note that transformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. 'dyex' and 'eyex' do not make sense for discrete variables.

- **atexog**: array-like, optional
  - Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

- **dummy**: bool, optional
  - If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.

### 3.5. Regression with Discrete Dependent Variable

407
count : bool, optional

If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

Returns DiscreteMargins : marginal effects instance

Returns an object that holds the marginal effects, standard errors, confidence intervals, etc. See statsmodels.discrete.discrete_margins.DiscreteMargins for more information.

Notes

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

statsmodels.discrete.discrete_model.LogitResults.initialize

LogitResults.initialize(model, params, **kwd)

statsmodels.discrete.discrete_model.LogitResults.llf

static LogitResults.llf()

statsmodels.discrete.discrete_model.LogitResults.llnull

static LogitResults.llnull()

statsmodels.discrete.discrete_model.LogitResults.llr

static LogitResults.llr()

statsmodels.discrete.discrete_model.LogitResults.llr_pvalue

static LogitResults.llr_pvalue()

statsmodels.discrete.discrete_model.LogitResults.load

classmethod LogitResults.load(fname)

load a pickle, (class method)

Parameters fname: string or filehandle

fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance:

statsmodels.discrete.discrete_model.LogitResults.margeff

LogitResults.margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)
LogitResults.normalized_cov_params

LogitResults.normalized_cov_params()

LogitResults.pred_table

LogitResults.pred_table(threshold=0.5)
Prediction table

Parameters threshold : scalar
Number between 0 and 1. Threshold above which a prediction is considered 1 and below which a prediction is considered 0.

Notes
	pred_table[i,j] refers to the number of times “i” was observed and the model predicted “j”. Correct predictions are along the diagonal.

LogitResults.predict

LogitResults.predict(exog=None, transform=True, *args, **kwargs)
Call self.model.predict with self.params as the first argument.

Parameters exog : array-like, optional
The values for which you want to predict.
transform : bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

Returns See self.model.predict :

LogitResults.prsquared

static LogitResults.prsquared()

LogitResults.pvalues

static LogitResults.pvalues()

LogitResults.remove_data

LogitResults.remove_data()
remove data arrays, all nobs arrays from result and model
This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.
Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

\texttt{statsmodels.discrete.discrete_model.LogitResults.resid}

\texttt{static LogitResults.resid()}

\texttt{statsmodels.discrete.discrete_model.LogitResults.resid_dev}

\texttt{static LogitResults.resid_dev()}

Deviance residuals

Notes

Deviance residuals are defined

\[ d_j = \pm \left( 2 \left[ Y_j \ln \left( \frac{Y_j}{M_j p_j} \right) + (M_j - Y_j) \ln \left( \frac{M_j - Y_j}{M_j (1 - p_j)} \right) \right] \right)^{1/2} \]

where

\[ p_j = \text{cdf}(X \beta) \]

\[ M_j \]

is the total number of observations sharing the covariate pattern \( j \).

For now \( M_j \) is always set to 1.

\texttt{statsmodels.discrete.discrete_model.LogitResults.resid_generalized}

\texttt{static LogitResults.resid_generalized()}

Generalized residuals

Notes

The generalized residuals for the Logit model are defined

\[ y - p \]

where \( p = \text{cdf}(X \beta) \). This is the same as the resid_response for the Logit model.

\texttt{statsmodels.discrete.discrete_model.LogitResults.resid_pearson}

\texttt{static LogitResults.resid_pearson()}

Pearson residuals
Notes

Pearson residuals are defined to be

\[ r_j = \frac{(y - M_j p_j)}{\sqrt{M_j p_j (1 - p_j)}} \]

where \( p_j = \text{cdf}(X \beta) \) and \( M_j \) is the total number of observations sharing the covariate pattern \( j \).

For now \( M_j \) is always set to 1.

\texttt{statsmodels.discrete.discrete_model.LogitResults.resid_response}

\texttt{static LogitResults\_resid\_response()}

The response residuals

Notes

Response residuals are defined to be

\[ y - p \]

where \( p = \text{cdf}(X \beta) \).

\texttt{statsmodels.discrete.discrete_model.LogitResults.save}

\texttt{LogitResults\_save(fname, remove\_data=False)}

save a pickle of this instance

Parameters \texttt{fname} : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

\texttt{remove\_data} : bool

If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove\_data method. In some cases not all arrays will be set to None.

Notes

If remove\_data is true and the model result does not implement a remove\_data method then this will raise an exception.

\texttt{statsmodels.discrete.discrete_model.LogitResults.summary}

\texttt{LogitResults\_summary(yname=None, xname=None, title=None, alpha=0.05, yname\_list=None)}

Summarize the Regression Results

Parameters \texttt{yname} : string, optional
Default is $y$

**xname**: list of strings, optional

Default is $var_{##}$ for ## in p the number of regressors

**title**: string, optional

Title for the top table. If not None, then this replaces the default title

**alpha**: float

significance level for the confidence intervals

**Returns** 

**smry**: Summary instance

this holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

`statsmodels.iolib.summary.Summary` class to hold summary results

`statsmodels.discrete.discrete_model.LogitResults.summary2`

LogitResults.summary2 (**yname=None**, **xname=None**, **title=None**, **alpha=0.05**, **float_format='%.4f'**)  

Experimental function to summarize regression results

**Parameters**

**xname**: List of strings of length equal to the number of parameters  
Names of the independent variables (optional)

**yname**: string  
Name of the dependent variable (optional)

**title**: string, optional

Title for the top table. If not None, then this replaces the default title

**alpha**: float

significance level for the confidence intervals

**float_format**: string  
print format for floats in parameters summary

**Returns** 

**smry**: Summary instance

this holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

`statsmodels.iolib.summary.Summary` class to hold summary results

`statsmodels.discrete.discrete_model.LogitResults.t_test`

LogitResults.t_test (**r_matrix**, **q_matrix=None**, **cov_p=None**, **scale=None**, **use_t=None**)  

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

**Parameters**

**r_matrix**: array-like, str, tuple
• array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
• str : The full hypotheses to test can be given as a string. See the examples.
• tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics
f_test for F tests
patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]
```
r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_test) "<T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```
Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
```
```python
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

```
statsmodels.discrete.discrete_model.LogitResults.tvalues

static LogitResults.tvalues()
   Return the t-statistic for a given parameter estimate.

statsmodels.discrete.discrete_model.LogitResults.wald_test

LogitResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
   Compute a Wald-test for a joint linear hypothesis.

   Parameters r_matrix : array-like, str, or tuple
       array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
       str : The full hypotheses to test can be given as a string. See the examples.
       tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

   q_matrix : array-like
       This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

   cov_p : array-like, optional
       An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

   scale : float, optional
       Default is 1.0 for no scaling.

   invcov : array-like, optional
       A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

   use_f : bool
       If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,
r_matrix (pX pX.T) r_matrix.T
```
is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**statsmodels.discrete.discrete_model.ProbitResults**

class statsmodels.discrete.discrete_model.ProbitResults(model, mlefit)

A results class for Probit Model

- **Parameters**
  - model: A DiscreteModel instance
  - params: array-like
    - The parameters of a fitted model.
  - hessian: array-like
    - The hessian of the fitted model.
  - scale: float
    - A scale parameter for the covariance matrix.

- **Returns**
  - *Attributes*
    - aic: float
      - Akaike information criterion. \(-2^{*}(llf - p)\) where \( p \) is the number of regressors including the intercept.
    - bic: float
      - Bayesian information criterion. \(-2^{*}ln(llf) + ln(nobs)*p\) where \( p \) is the number of regressors including the intercept.
    - bse: array
      - The standard errors of the coefficients.
    - df_resid: float
      - See model definition.
    - df_model: float
      - See model definition.
    - fitted_values: array
      - Linear predictor XB.
    - llf: float
      - Value of the loglikelihood
    - llnull: float
      - Value of the constant-only loglikelihood
    - llr: float
      - Likelihood ratio chi-squared statistic; \(-2^{*}(llnull - llf)\)
    - llr_pvalue: float
      - The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr has a chi-squared distribution with degrees of freedom df_model.
    - prsquared: float
McFadden’s pseudo-R-squared. 1 - (llf/llnull)

Methods

aic()  # Returns the confidence interval of the fitted parameters.
bic()  # Returns the variance/covariance matrix.
bse()  # Compute the F-test for a joint linear hypothesis.
conf_int([alpha, cols, method])  # Get marginal effects of the fitted model.
cov_params([r_matrix, column, scale, cov_p, ...])
f_test(r_matrix[, q_matrix, cov_p, scale, ...])
fittedvalues()
get_margeff([at, method, atexog, dummy, count])
initialize(model, params, **kwd)
llf()
llnull()
llr()
llr_pvalue()  # load a pickle, (class method)
load(fname)
margeff([at, method, atexog, dummy, count])
normalized_cov_params()
pred_table([threshold])  # Prediction table
predict([exog, transform])
prsquared()
pvalues()  # Call self.model.predict with self.params as the first argument.
remove_data()  # remove data arrays, all nobs arrays from result and model
resid()  # Deviance residuals..
resid_dev()  # Generalized residuals..
resid_generalized()  # Pearson residuals..
resid_pearson()  # The response residuals..
resid_response()  # save a pickle of this instance
save(fname[, remove_data])
summary([yname, xname, title, alpha, yname_list])
summary2([yname, xname, title, alpha, ...])
t_test(r_matrix[, q_matrix, cov_p, scale, use_t])
tvalues()  # Compute a t-test for a joint linear hypothesis of the form Rb = q
wald_test(r_matrix[, q_matrix, cov_p, ...])  # Return the t-statistic for a given parameter estimate.
Compute a Wald-test for a joint linear hypothesis.

statsmodels.discrete.discrete_model.ProbitResults.aic

static ProbitResults.aic()

statsmodels.discrete.discrete_model.ProbitResults.bic

static ProbitResults.bic()

statsmodels.discrete.discrete_model.ProbitResults.bse

static ProbitResults.bse()
ProbitResults.\texttt{conf\_int}(alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

**Parameters**

- **alpha**: float, optional
  
The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

- **cols**: array-like, optional
  
  cols specifies which confidence intervals to return

- **method**: string
  
  Not Implemented Yet Method to estimate the confidence interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jnj” : “jac” : “boot-bse” “boot\_quant” “profile”

**Returns**

- **conf\_int**: array
  
  Each row contains [lower, upper] confidence interval

**Notes**

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

**Examples**

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[-5496529.48322745, -1467987.78596704],
       [-177.02903529, 207.15277984],
       [-0.1115811 , 0.03994274],
       [-3.12506664, -0.91539297],
       [-1.5179487 , -0.54850503],
       [-0.56251721, 0.460309   ],
       [ 798.7875153 , 2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[-0.1115811 , 0.03994274],
       [-3.12506664, -0.91539297]])
```

ProbitResults.\texttt{cov\_params}(r\_matrix=None, column=None, scale=None, cov\_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.
Parameters  r_matrix : array-like
    Can be 1d, or 2d. Can be used alone or with other.

column : array-like, optional
    Must be used on its own. Can be 0d or 1d see below.

scale : float, optional
    Can be specified or not. Default is None, which means that the scale argument is taken
    from the model.

other : array-like, optional
    Can be used when r_matrix is specified.

Returns  (The below are assumed to be in matrix notation.) :

cov : ndarray
    If no argument is specified returns the covariance matrix of a model :
    (scale)*(X.T X)^(-1) :
    If contrast is specified it pre and post-multiplies as follows :
    (scale) * r_matrix (X.T X)^(-1) r_matrix.T :
    If contrast and other are specified returns :
    (scale) * r_matrix (X.T X)^(-1) other.T :
    If column is specified returns :
    (scale) * (X.T X)^(-1)[column,column] if column is 0d :
    OR :
    (scale) * (X.T X)^(-1)[column][:,column] if column is 1d :

statsmodels.discrete.discrete_model.ProbitResults.f_test

ProbitResults.f_test (r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
    Compute the F-test for a joint linear hypothesis.

Parameters  r_matrix : array-like, str, or tuple
    • array : An r x k array where r is the number of restrictions to test and k is the number of
      regressors.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
    This is deprecated. See r_matrix and the examples for more information on new usage.
    Can be either a scalar or a length p row vector. If omitted and r_matrix is an array,
    q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
    An alternative estimate for the parameter covariance matrix. If None is given,
    self.normalized_cov_params is used.

scale : float, optional
Default is 1.0 for no scaling.

invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix $r_{matrix}$ is assumed to be non-singular. More precisely,

$r_{matrix} (pX pX^T) r_{matrix}^T$

is assumed invertible. Here, $pX$ is the generalized inverse of the design matrix of the model. There can be
problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]

This tests that each coefficient is jointly statistically significantly different from zero.

```print(results.f_test(A))```  
F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6

Compare this to

```results.F 330.2853392346658
>>> results.F_p 4.98403096572e-10
>>> B = np.array(((0, 0, 1, -1, 0, 0, 0), (0, 0, 0, 0, 0, 1, -1)))

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the
5th and 6th regressors are equal.

```print(results.f_test(B))```  
F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2

Alternatively, you can specify the hypothesis tests using a string

```from statsmodels.datasets import longley
from statsmodels.formula.api import ols
data = longley.load_pandas().data
formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
results = ols(formula, data).fit()```
```python
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```
statsmodels.discrete.discrete_model.ProbitResults.fittedvalues

statsmodels.discrete.discrete_model.ProbitResults.fittedvalues()

statsmodels.discrete.discrete_model.ProbitResults.get_margeff

ProbitResults.get_margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)

Get marginal effects of the fitted model.

Parameters at : str, optional

Options are:

- 'overall', The average of the marginal effects at each observation.
- 'mean', The marginal effects at the mean of each regressor.
- 'median', The marginal effects at the median of each regressor.
- 'zero', The marginal effects at zero for each regressor.
- 'all', The marginal effects at each observation. If at is all only margeff will be available from the returned object.

Note that if exog is specified, then marginal effects for all variables not specified by exog are calculated using the at option.

method : str, optional

Options are:

- 'dydx' - dy/dx - No transformation is made and marginal effects are returned. This is the default.
- 'eyex' - estimate elasticities of variables in exog – d(lny)/d(lnx)
- 'dyex' - estimate semielasticity – dy/d(lnx)
- 'eydx' - estimate semeilasticity – d(lny)/dx

Note that tranformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. ‘dyex’ and ‘eyex’ do not make sense for discrete variables.

atexog : array-like, optional

Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

dummy : bool, optional

If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.
count : bool, optional

If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

Returns DiscreteMargins : marginal effects instance

Returns an object that holds the marginal effects, standard errors, confidence intervals, etc. See statsmodels.discrete.discrete_margins.DiscreteMargins for more information.

Notes

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

statsmodels.discrete.discrete_model.ProbitResults.initialize

ProbitResults.initialize(model, params, **kwds)

statsmodels.discrete.discrete_model.ProbitResults.llf

static ProbitResults.llf()

statsmodels.discrete.discrete_model.ProbitResults.llnull

static ProbitResults.llnull()

statsmodels.discrete.discrete_model.ProbitResults.llr

static ProbitResults.llr()

statsmodels.discrete.discrete_model.ProbitResults.llr_pvalue

static ProbitResults.llr_pvalue()

statsmodels.discrete.discrete_model.ProbitResults.load

classmethod ProbitResults.load(fname)

load a pickle, (class method)

Parameters fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance :
ProbitResults.margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)

ProbitResults.normalized_cov_params()

ProbitResults.pred_table(threshold=0.5)

**Prediction table**

**Parameters**

**threshold**: scalar

Number between 0 and 1. Threshold above which a prediction is considered 1 and below which a prediction is considered 0.

**Notes**

pred_table[i,j] refers to the number of times “i” was observed and the model predicted “j”. Correct predictions are along the diagonal.

ProbitResults.predict(exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

**Parameters**

**exog**: array-like, optional

The values for which you want to predict.

**transform**: bool, optional

If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

**Returns**

See self.model.predict:

ProbitResults.prsquared()

ProbitResults.pvalues()
**statsmodels.discrete.discrete_model.ProbitResults.remove_data**

```python
ProbitResults.remove_data()
```

remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

**statsmodels.discrete.discrete_model.ProbitResults.resid**

```python
static ProbitResults.resid()
```

**statsmodels.discrete.discrete_model.ProbitResults.resid_dev**

```python
static ProbitResults.resid_dev()
```

Deviance residuals

**Notes**

Deviance residuals are defined

\[ d_j = \pm \left( 2 \left[ Y_j \ln \left( \frac{Y_j}{M_j p_j} \right) + (M_j - Y_j) \ln \left( \frac{M_j - Y_j}{M_j (1 - p_j)} \right) \right] \right)^{1/2} \]

where

\( p_j = \text{cdf}(X \beta) \) and \( M_j \) is the total number of observations sharing the covariate pattern \( j \).

For now \( M_j \) is always set to 1.

**statsmodels.discrete.discrete_model.ProbitResults.resid_generalized**

```python
static ProbitResults.resid_generalized()
```

Generalized residuals

3.5. Regression with Discrete Dependent Variable 423
The generalized residuals for the Probit model are defined

\[
y \frac{\phi(X\beta)}{\Phi(X\beta)} - (1 - y) \frac{\phi(X\beta)}{1 - \Phi(X\beta)}
\]

**statsmodels.discrete.discrete_model.ProbitResults.resid_pearson**

```python
static ProbitResults.resid_pearson()
```

*Pearson residuals*

**Notes**

Pearson residuals are defined to be

\[
r_j = \frac{(y - M_j p_j)}{\sqrt{M_j p_j (1 - p_j)}}
\]

where \( p_j = \text{cdf}(X\beta) \) and \( M_j \) is the total number of observations sharing the covariate pattern \( j \).

For now \( M_j \) is always set to 1.

**statsmodels.discrete.discrete_model.ProbitResults.resid_response**

```python
static ProbitResults.resid_response()
```

*The response residuals*

**Notes**

Response residuals are defined to be

\[
y - p
\]

where \( p = \text{cdf}(X\beta) \).

**statsmodels.discrete.discrete_model.ProbitResults.save**

```python
ProbitResults.save(fname, remove_data=False)
```

*save a pickle of this instance*

**Parameters**

- **fname** : string or filehandle

  fname can be a string to a file path or filename, or a filehandle.

- **remove_data** : bool

  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.
Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

**statsmodels.discrete.discrete_model.ProbitResults.summary**

```
ProbitResults.summary(yname=None, xname=None, title=None, alpha=0.05, yname_list=None)
```

Summarize the Regression Results

**Parameters**

- **yname** : string, optional
  - Default is `y`
- **xname** : list of strings, optional
  - Default is `var_##` for ## in p the number of regressors
- **title** : string, optional
  - Title for the top table. If not None, then this replaces the default title
- **alpha** : float
  - Significance level for the confidence intervals

**Returns**

- **smry** : Summary instance
  - This holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

- **statsmodels.iolib.summary.Summary** class to hold summary results

**statsmodels.discrete.discrete_model.ProbitResults.summary2**

```
ProbitResults.summary2(yname=None, xname=None, title=None, alpha=0.05, float_format='%.4f')
```

Experimental function to summarize regression results

**Parameters**

- **xname** : List of strings of length equal to the number of parameters
  - Names of the independent variables (optional)
- **yname** : string
  - Name of the dependent variable (optional)
- **title** : string, optional
  - Title for the top table. If not None, then this replaces the default title
- **alpha** : float
  - Significance level for the confidence intervals
- **float_format** : string
  - Print format for floats in parameters summary

**Returns**

- **smry** : Summary instance
this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.discrete.discrete_model.ProbitResults.t_test

ProbitResults.t_test (r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)
Compute a t-test for a joint linear hypothesis of the form Rb = q

Parameters r_matrix : array-like, str, tuple
  • array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  • str : The full hypotheses to test can be given as a string. See the examples.
  • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
  An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional
  If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics
f_test for F tests

patsy.DesignInfo.linear_constraint

Examples

>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r[5:] = [1,-1]
r tests that the coefficients on the 5th and 6th independent variable are the same.

Alternatively, you can specify the hypothesis tests using a string:

```python
dta = sm.datasets.longley.load_pandas().data
formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
results = ols(formula, dta).fit()
hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
t_test = results.t_test(hypotheses)
```

```python
print(t_test)
```

This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.
If True, then the F-distribution is used. If False, then the asymptotic distribution, chi-square is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes
The matrix r_matrix is assumed to be non-singular. More precisely,
\[ r_{\text{matrix}} (pX pX^T) r_{\text{matrix}}^T \]
is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**statsmodels.discrete.discrete_model.CountResults**

class statsmodels.discrete.discrete_model.CountResults(model, mlefit)
A results class for count data

Parameters
- **model**: A DiscreteModel instance
- **params**: array-like
  The parameters of a fitted model.
- **hessian**: array-like
  The hessian of the fitted model.
- **scale**: float
  A scale parameter for the covariance matrix.

Returns
- **Attributes**:
  - **aic**: float
    Akaike information criterion. \(-2*(llf - p)\) where p is the number of regressors including the intercept.
  - **bic**: float
    Bayesian information criterion. \(-2*llf + \ln(nobs)*p\) where p is the number of regressors including the intercept.
  - **bse**: array
    The standard errors of the coefficients.
  - **df_resid**: float
    See model definition.
  - **df_model**: float
    See model definition.
  - **fitted_values**: array
    Linear predictor XB.
**llf**: float
Value of the loglikelihood

**llnull**: float
Value of the constant-only loglikelihood

**llr**: float
Likelihood ratio chi-squared statistic; \(-2*\text{llnull} - \text{llf}\)

**llr_pvalue**: float
The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr has a chi-squared distribution with degrees of freedom \(df_{model}\).

**prsquared**: float
McFadden’s pseudo-R-squared. \(1 - (\text{llf/llnull})\)

**Methods**

```python
aic()
bic()
bse()
conf_int([alpha, cols, method])
cov_params([r_matrix, column, scale, cov_p, ...])
f_test(r_matrix[, q_matrix, cov_p, scale, ...])
fittedvalues()
get_margeff([at, method, atexog, dummy, count])
initialize(model, params, **kwds)
llf()
llnull()
llr()
llr_pvalue()
load(fname)
margeff([at, method, atexog, dummy, count])
normalized_cov_params()
predict([exog, transform])
prsquared()
pvalues()
remove_data()
resid()
save(fname[, remove_data])
summary([yname, xname, title, alpha, yname_list])
summary2([yname, xname, title, alpha, ...])
t_test(r_matrix[, q_matrix, cov_p, scale, use_t])
tvalues()
wald_test(r_matrix[, q_matrix, cov_p, ...])
```

Returns the confidence interval of the fitted parameters.

Returns the variance/covariance matrix.

Compute the F-test for a joint linear hypothesis.

Get marginal effects of the fitted model.

load a pickle, (class method)

Call self.model.predict with self.params as the first argument.

remove data arrays, all nobs arrays from result and model

Residuals ..

save a pickle of this instance

Summarize the Regression Results

Experimental function to summarize regression results

Compute a t-test for a joint linear hypothesis of the form \(Rb = q\)

Return the t-statistic for a given parameter estimate.

Compute a Wald-test for a joint linear hypothesis.

**statsmodels.discrete.discrete_model.CountResults.aic**

```python
static CountResults.aic()
```

3.5. Regression with Discrete Dependent Variable
static CountResults.bic()

statsmodels.discrete.discrete_model.CountResults.bse

static CountResults.bse()

statsmodels.discrete.discrete_model.CountResults.conf_int

CountResults.conf_int(alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

Parameters:
- alpha : float, optional
  The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.
- cols : array-like, optional
  cols specifies which confidence intervals to return
- method : string
  Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

Returns:
- conf_int : array
  Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[-5496529.48322745, -1467987.78596704],
       [-177.02903529, 207.15277984],
       [-0.1115811 ,  0.03994274],
       [-3.12506664,  -0.91539297],
       [-1.5179487 ,  -0.54850503],
       [-0.56251721,   0.460309  ],
       [  798.7875153 ,  2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[ -0.1115811 ,  0.03994274],
       [-3.12506664,  -0.91539297]])
```
CountResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters r_matrix : array-like
    Can be 1d, or 2d. Can be used alone or with other.

column : array-like, optional
    Must be used on its own. Can be 0d or 1d see below.

scale : float, optional
    Can be specified or not. Default is None, which means that the scale argument is taken from the model.

other : array-like, optional
    Can be used when r_matrix is specified.

Returns (The below are assumed to be in matrix notation.)

cov : ndarray
    If no argument is specified returns the covariance matrix of a model:
    (scale)*(X.T X)^(-1):

    If contrast is specified it pre and post-multiplies as follows:
    (scale) * r_matrix (X.T X)^(-1) r_matrix.T:

    If contrast and other are specified returns:
    (scale) * r_matrix (X.T X)^(-1) other.T:

    If column is specified returns:
    (scale) * (X.T X)^(-1)[column,column] if column is 0d:
    OR:
    (scale) * (X.T X)^(-1)[column][:,column] if column is 1d:

CountResults.f_test(r_matrix, q_matrix=None, scale=1.0, invcov=None)

Compute the F-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
    • array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
This is deprecated. See \textit{r\_matrix} and the examples for more information on new usage. Can be either a scalar or a length $p$ row vector. If omitted and \textit{r\_matrix} is an array, \textit{q\_matrix} is assumed to be a conformable array of zeros.

\textbf{cov\_p} : array-like, optional
An alternative estimate for the parameter covariance matrix. If None is given, self.normalized\_cov\_params is used.

\textbf{scale} : float, optional
Default is 1.0 for no scaling.

\textbf{invcov} : array-like, optional
A $q \times q$ array to specify an inverse covariance matrix based on a restrictions matrix.

\textbf{See also:}
\texttt{statsmodels.contrasts}, \texttt{statsmodels.model.LikelihoodModelResults.wald\_test}, \texttt{statsmodels.model.LikelihoodModelResults.t\_test}, \texttt{patsy.DesignInfo.linear\_constraint}

\textbf{Notes}
The matrix \textit{r\_matrix} is assumed to be non-singular. More precisely, \textit{r\_matrix} (pX pX.T) \textit{r\_matrix}.T is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

\textbf{Examples}

\begin{verbatim}
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.

>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
 df_denom=9, df_num=6>

Compare this to

>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10

>>> B = np.array([[0,0,1,-1,0,0,0], [0,0,0,0,0,1,-1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.
\end{verbatim}
```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```
statsmodels.discrete.discrete_model.CountResults.fittedvalues

static CountResults.fittedvalues()

statsmodels.discrete.discrete_model.CountResults.get_margeff

CountResults.get_margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)

Get marginal effects of the fitted model.

Parameters at : str, optional

Options are:

* 'overall', The average of the marginal effects at each observation.
* 'mean', The marginal effects at the mean of each regressor.
* 'median', The marginal effects at the median of each regressor.
* 'zero', The marginal effects at zero for each regressor.
* 'all', The marginal effects at each observation. If at is all only margeff will be available from the returned object.

Note that if exog is specified, then marginal effects for all variables not specified by exog are calculated using the at option.

method : str, optional

Options are:

* 'dydx' - dy/dx - No transformation is made and marginal effects are returned. This is the default.
* 'eyex' - estimate elasticities of variables in exog – d(lny)/d(lnx)
* 'dyex' - estimate semielasticity – dy/d(lnx)
* 'eydx' - estimate semielasticity – d(lny)/dx

Note that tranformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. 'dyex' and 'eyex' do not make sense for discrete variables.

atexog : array-like, optional

3.5. Regression with Discrete Dependent Variable 433
Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

dummy : bool, optional

If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.

count : bool, optional

If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

Returns DiscreteMargins : marginal effects instance

Returns an object that holds the marginal effects, standard errors, confidence intervals, etc. See statsmodels.discrete.discrete_margins.DiscreteMargins for more information.

Notes

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

statsmodels.discrete.discrete_model.CountResults.initialize

CountResults.initialize(model, params, **kwd)

statsmodels.discrete.discrete_model.CountResults.llf

static CountResults.llf()

statsmodels.discrete.discrete_model.CountResults.llnull

static CountResults.llnull()

statsmodels.discrete.discrete_model.CountResults.llr

static CountResults.llr()

statsmodels.discrete.discrete_model.CountResults.llr_pvalue

static CountResults.llr_pvalue()
**statsmodels.discrete.discrete_model.CountResults.load**

**classmethod CountResults.load(fname)**

Load a pickle, (class method)

**Parameters**

fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

**Returns**

unpickled instance

**statsmodels.discrete.discrete_model.CountResults.margeff**

CountResults.margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)

**statsmodels.discrete.discrete_model.CountResults.normalized_cov_params**

CountResults.normalized_cov_params()

**statsmodels.discrete.discrete_model.CountResults.predict**

CountResults.predict(exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

**Parameters**

exog : array-like, optional

The values for which you want to predict.

transform : bool, optional

If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

**Returns**

See self.model.predict

**statsmodels.discrete.discrete_model.CountResults.prsquared**

static CountResults.prsquared()

**statsmodels.discrete.discrete_model.CountResults.pvalues**

static CountResults.pvalues()

**statsmodels.discrete.discrete_model.CountResults.remove_data**

CountResults.remove_data()

Remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.
Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

\texttt{statsmodels.discrete.discrete\_model.CountResults.resid}

\texttt{static CountResults.resid()}

Residuals

Notes

The residuals for Count models are defined as

\[ y - p \]

where \( p = \exp(X\beta) \). Any exposure and offset variables are also handled.

\texttt{statsmodels.discrete.discrete\_model.CountResults.save}

\texttt{CountResults.save(fname, remove\_data=False)}

save a pickle of this instance

Parameters

- \texttt{fname : string or filehandle}
  
  fname can be a string to a file path or filename, or a filehandle.

- \texttt{remove\_data : bool}
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

\texttt{statsmodels.discrete.discrete\_model.CountResults.summary}

\texttt{CountResults.summary(yname=None, xname=None, title=None, alpha=0.05, yname\_list=None)}

Summarize the Regression Results

Parameters

- \texttt{yname : string, optional}
  
  Default is \( y \)
xname : list of strings, optional
    Default is var_## for ## in p the number of regressors

title : string, optional
    Title for the top table. If not None, then this replaces the default title

alpha : float
    significance level for the confidence intervals

Returns smry : Summary instance
    this holds the summary tables and text, which can be printed or converted to various
    output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.discrete.discrete_model.CountResults.summary2

CountResults.summary2(yname=None, xname=None, title=None, alpha=0.05,
            float_format='%.4f')

Experimental function to summarize regression results

Parameters xname : List of strings of length equal to the number of parameters
    Names of the independent variables (optional)

yname : string
    Name of the dependent variable (optional)

title : string, optional
    Title for the top table. If not None, then this replaces the default title

alpha : float
    significance level for the confidence intervals

float_format: string:
    print format for floats in parameters summary

Returns smry : Summary instance
    this holds the summary tables and text, which can be printed or converted to various
    output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.discrete.discrete_model.CountResults.t_test

CountResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)

Compute a t-test for a joint linear hypothesis of the form Rb = q

Parameters r_matrix : array-like, str, tuple
• array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
• str : The full hypotheses to test can be given as a string. See the examples.
• tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics
f_test for F tests

patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]
```
r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_Test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```
Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = ‘TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR’
>>> results = ols(formula, dta).fit()
>>> hypotheses = ‘GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1’
```
```python
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)

statsmodels.discrete.discrete_model.CountResults.tvalues

static CountResults.tvalues()
Return the t-statistic for a given parameter estimate.

statsmodels.discrete.discrete_model.CountResults.wald_test

CountResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
    • array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
    This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
    An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
    Default is 1.0 for no scaling.

invcov : array-like, optional
    A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool
    If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes
The matrix r_matrix is assumed to be non-singular. More precisely, r_matrix (pX pX.T) r_matrix.T
```
is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**statsmodels.discrete.discrete_model.MultinomialResults**

class statsmodels.discrete.discrete_model.MultinomialResults(model, mlefit)

A results class for multinomial data

**Parameters**

- model : A DiscreteModel instance
- params : array-like
  The parameters of a fitted model.
- hessian : array-like
  The hessian of the fitted model.
- scale : float
  A scale parameter for the covariance matrix.

**Returns**

**Attributes**

- aic : float
  Akaike information criterion. $-2(\text{llf} - p)$ where $p$ is the number of regressors including the intercept.
- bic : float
  Bayesian information criterion. $-2\cdot \text{llf} + \ln(\text{nobs})\cdot p$ where $p$ is the number of regressors including the intercept.
- bse : array
  The standard errors of the coefficients.
- df_resid : float
  See model definition.
- df_model : float
  See model definition.
- fitted_values : array
  Linear predictor XB.
- llf : float
  Value of the loglikelihood
- llnull : float
  Value of the constant-only loglikelihood
- llr : float
  Likelihood ratio chi-squared statistic; $-2(\text{llnull} - \text{llf})$
- llr_pvalue : float
  The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr has a chi-squared distribution with degrees of freedom df_model.
- prsquared : float
McFadden’s pseudo-R-squared. $1 - \frac{llf}{llnull}$

**Methods**

- `aic()`
- `bic()`
- `bse()`
- `conf_int(alpha, cols)`
- `cov_params([r_matrix, column, scale, cov_p, ...])`
- `f_test(r_matrix[, q_matrix, cov_p, scale, ...])`
- `fittedvalues()`
- `get_margeff([at, method, atexog, dummy, count])`
- `initialize(model, params, **kwd)`
- `llf()`
- `llnull()`
- `llr()`
- `llr_pvalue()`
- `load(fname)`
- `margeff()`
- `normalized_cov_params()`
- `pred_table()`
- `predict([lexog, transform])`
- `prsquared()`
- `pvalues()`
- `remove_data()`
- `resid_misclassified()`
- `save(fname[, remove_data])`
- `summary([yname, xname, title, alpha, yname_list])`
- `summary2([alpha, float_format])`
- `t_test(r_matrix[, q_matrix, cov_p, scale, use_t])`
- `tvalues()`
- `wald_test(r_matrix[, q_matrix, cov_p, ...])`

**static** `MultinomialResults.aic()`

**static** `MultinomialResults.bic()`

**static** `MultinomialResults.bse()`

**static** `MultinomialResults.conf_int(alpha=0.05, cols=None)`

---

3.5. Regression with Discrete Dependent Variable 441
MultinomialResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of $\sigma^2$. Scale is assumed to be a scalar.

Parameters

r_matrix : array-like
Can be 1d, or 2d. Can be used alone or with other.

column : array-like, optional
Must be used on its own. Can be 0d or 1d see below.

case : float, optional
Can be specified or not. Default is None, which means that the scale argument is taken from the model.

other : array-like, optional
Can be used when r_matrix is specified.

Returns

(The below are assumed to be in matrix notation.)

cov : ndarray
If no argument is specified returns the covariance matrix of a model:

(scale)*(X.T X)^(-1):

If contrast is specified it pre and post-multiplies as follows:

(scale) * r_matrix (X.T X)^(-1) r_matrix.T:

If contrast and other are specified returns:

(scale) * r_matrix (X.T X)^(-1) other.T:

If column is specified returns:

(scale) * (X.T X)^(-1)[column,column] if column is 0d:

OR:

(scales) * (X.T X)^(-1)[column][:,column] if column is 1d:

MultinomialResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)

Compute the F-test for a joint linear hypothesis.

Parameters

r_matrix : array-like, str, or tuple

- array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

**cov_p**: array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

**scale**: float, optional

Default is 1.0 for no scaling.

**invcov**: array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**See also:**

statsmodels_contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

**Notes**

The matrix `r_matrix` is assumed to be non-singular. More precisely, 

```
r_matrix (pX pX.T) r_matrix.T
```

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:,:]

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
df_denom=9, df_num=6>
```

Compare this to

```python
>>> results.F
330.28533923463488
>>> results.F_p
4.98403052872e-10
```

```python
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 1, -1]])

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.
```
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
df_num=2>

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

---

```
statsmodels.discrete.discrete_model.MultinomialResults.fittedvalues

static MultinomialResults.fittedvalues()

statsmodels.discrete.discrete_model.MultinomialResults.get_margeff

MultinomialResults.get_margeff(at='overall', method='dydx', atexog=None,
dummy=False, count=False)

Get marginal effects of the fitted model.

Parameters
at : str, optional

Options are:

• ‘overall’, The average of the marginal effects at each observation.
• ‘mean’, The marginal effects at the mean of each regressor.
• ‘median’, The marginal effects at the median of each regressor.
• ‘zero’, The marginal effects at zero for each regressor.
• ‘all’, The marginal effects at each observation. If at is all only margeff will be available from the returned object.

Note that if exog is specified, then marginal effects for all variables not specified by exog are calculated using the at option.

method : str, optional

Options are:

• ‘dydx’ - dy/dx - No transformation is made and marginal effects are returned. This is the default.
• ‘eyex’ - estimate elasticities of variables in exog – d(lny)/d(lnx)
• ‘dyex’ - estimate semielasticity – dy/d(lnx)
• ‘eydx’ - estimate semeilasticity – d(lny)/dx

Note that tranformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. ‘dyex’ and ‘eyex’ do not make sense for discrete variables.

atexog : array-like, optional
```
Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

**dummy**: bool, optional

If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.

**count**: bool, optional

If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

Returns **DiscreteMargins**: marginal effects instance

Returns an object that holds the marginal effects, standard errors, confidence intervals, etc. See `statsmodels.discrete.discrete_margins.DiscreteMargins` for more information.

**Notes**

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

```python
statsmodels.discrete.discrete_model.MultinomialResults.initialize

MultinomialResults.initialize(model, params, **kwd)
```

```python
statsmodels.discrete.discrete_model.MultinomialResults.llf

static MultinomialResults.llf()
```

```python
statsmodels.discrete.discrete_model.MultinomialResults.llnull

static MultinomialResults.llnull()
```

```python
statsmodels.discrete.discrete_model.MultinomialResults.llr

static MultinomialResults.llr()
```

```python
statsmodels.discrete.discrete_model.MultinomialResults.llr_pvalue

static MultinomialResults.llr_pvalue()
```
class method MultinomialResults.load fname

Parameters fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance :

MultinomialResults.margeff()

MultinomialResults.normalized_cov_params()

MultinomialResults.pred_table()

Returns the J x J prediction table.

Notes

pred_table[i,j] refers to the number of times “i” was observed and the model predicted “j”. Correct predictions are along the diagonal.

MultinomialResults.predict (exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

Parameters exog : array-like, optional

The values for which you want to predict.

transform : bool, optional

If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

Returns See self.model.predict :

MultinomialResults.prsquared()
statsmodels.discrete.discrete_model.MultinomialResults.pvalues

\texttt{static MultinomialResults.pvalues()}\

statsmodels.discrete.discrete_model.MultinomialResults.remove_data

MultinomialResults.remove_data()
remove data arrays, all nobs arrays from result and model
This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

\textbf{Warning:} Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.
The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

statsmodels.discrete.discrete_model.MultinomialResults.resid_misclassified

\texttt{static MultinomialResults.resid_misclassified()}\
Residuals indicating which observations are misclassified.

\textbf{Notes}
The residuals for the multinomial model are defined as

$$argmax(y_i) \neq argmax(p_i)$$

where \(argmax(y_i)\) is the index of the category for the endogenous variable and \(argmax(p_i)\) is the index of the predicted probabilities for each category. That is, the residual is a binary indicator that is 0 if the category with the highest predicted probability is the same as that of the observed variable and 1 otherwise.

statsmodels.discrete.discrete_model.MultinomialResults.save

MultinomialResults.save (\texttt{fname, remove_data=False})
save a pickle of this instance

\textbf{Parameters} \texttt{fname} : string or filehandle
fname can be a string to a file path or filename, or a filehandle.
\texttt{remove_data} : bool
If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.
Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

statsmodels.discrete.discrete_model.MultinomialResults.summary

MultinomialResults.summary (yname=None, xname=None, title=None, alpha=0.05, yname_list=None)
Summarize the Regression Results

Parameters

- **yname**: string, optional
  - Default is y

- **xname**: list of strings, optional
  - Default is \texttt{var_##} for ## in \texttt{p} the number of regressors

- **title**: string, optional
  - Title for the top table. If not None, then this replaces the default title

- **alpha**: float
  - significance level for the confidence intervals

Returns

smry: Summary instance

this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

- statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.discrete.discrete_model.MultinomialResults.summary2

MultinomialResults.summary2 (alpha=0.0.5, float_format='%.4f')
Experimental function to summarize regression results

Parameters

- **alpha**: float
  - significance level for the confidence intervals

- **float_format**: string
  - print format for floats in parameters summary

Returns

smry: Summary instance

this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

- statsmodels.iolib.summary2.Summary class to hold summary results
**statsmodels.discrete.discrete_model.MultinomialResults.t_test**

MultinomialResults.t_test(r_matrix=None, q_matrix=None, cov_p=None, scale=None, use_t=None)

Compute a t-test for a joint linear hypothesis of the form \( Rb = q \)

**Parameters**

- **r_matrix**: array-like, str, tuple
  - array: If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like or scalar, optional
  - An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **cov_p**: array-like, optional
  - An optional scale to use. Default is the scale specified by the model fit.

- **scale**: float, optional
  - An optional scale to use. Default is the scale specified by the model fit.

- **use_t**: bool, optional
  - If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

**See also:**

- tvalues: individual t statistics
- f_test: for F tests

**patsy.DesignInfo.linear_constraint**

**Examples**

```python
def main():
    import numpy as np
    import statsmodels.api as sm
    data = sm.datasets.longley.load()
    data.exog = sm.add_constant(data.exog)
    results = sm.OLS(data.endog, data.exog).fit()
    r = np.zeros_like(results.params)
    r[5:] = [1,-1]
    print(r)  
    [ 0. 0. 0. 0. 0. 1. -1.]
    r tests that the coefficients on the 5th and 6th independent variable are the same.
```

```python
>>> T_Test = results.t_test(r) >> print(T_Test) 
<T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```
Alternatively, you can specify the hypothesis tests using a string.

```python
dta = sm.datasets.longley.load_pandas().data
formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
results = ols(formula, dta).fit()
hyotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
t_test = results.t_test(hypotheses)
print(t_test)
```

```python
statsmodels.discrete.discrete_model.MultinomialResults.tvalues

```

```python
statsmodels.discrete.discrete_model.MultinomialResults.tvalues()

Return the t-statistic for a given parameter estimate.
```

```python
statsmodels.discrete.discrete_model.MultinomialResults.wald_test

MultinomialResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)

Compute a Wald-test for a joint linear hypothesis.

Parameters

- **r_matrix**: array-like, str, or tuple
  - array: An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like
  This is deprecated. See **r_matrix** and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array
  q_matrix is assumed to be a conformable array of zeros.

- **cov_p**: array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given,
  self.normalized_cov_params is used.

- **scale**: float, optional
  Default is 1.0 for no scaling.

- **invcov**: array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

- **use_f**: bool
  If True, then the F-distribution is used. If False, then the asymptotic distribution,
  chisquare is used. The test statistic is proportionally adjusted for the distribution by
  the number of constraints in the hypothesis.

See also:

- statsmodels.contrasts
- statsmodels.model.LikelihoodModelResults.f_test
- statsmodels.model.LikelihoodModelResults.t_test
- patsy.DesignInfo.linear_constraint
Notes

The matrix \( r_{\text{matrix}} \) is assumed to be non-singular. More precisely,
\[
r_{\text{matrix}} (pX \ pX^T) r_{\text{matrix}}^T
\]
is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

```python
class statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults:

A results class for NegativeBinomial 1 and 2

**Parameters**
- **model**: A DiscreteModel instance
  - **params**: array-like
    The parameters of a fitted model.
  - **hessian**: array-like
    The hessian of the fitted model.
  - **scale**: float
    A scale parameter for the covariance matrix.

**Returns**

- **Attributes**: 
  - **aic**: float
    Akaike information criterion. \(-2^{*}(llf - p)\) where \( p \) is the number of regressors including the intercept.
  - **bic**: float
    Bayesian information criterion. \(-2^{*}llf + \ln(nobs)^{*}p\) where \( p \) is the number of regressors including the intercept.
  - **bse**: array
    The standard errors of the coefficients.
  - **df_resid**: float
    See model definition.
  - **df_model**: float
    See model definition.
  - **fitted_values**: array
    Linear predictor \( XB \).
  - **llf**: float
    Value of the loglikelihood
  - **llnull**: float
    Value of the constant-only loglikelihood
```
**llr**: float

Likelihood ratio chi-squared statistic; \(-2*(\text{llnull} - \text{llf})\)

**llr_pvalue**: float

The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr has a chi-squared distribution with degrees of freedom \(df\_model\).

**prsquared**: float

McFadden’s pseudo-R-squared. 1 - (llf/llnull)

**Methods**

- `aic()`
- `bic()`
- `bse()`
- `conf_int([alpha, cols, method])`
- `cov_params([r_matrix, column, scale, cov_p, ...])`
- `f_test(r_matrix[, q_matrix, cov_p, scale, ...])`
- `fittedvalues()`
- `get_margeff([at, method, atexog, dummy, count])`
- `initialize(model, params, **kwds)`
- `llf()`
- `llnull()`
- `llr()`
- `llr_pvalue()`
- `lnalpha()`
- `lnalpha_std_err()`
- `load(fname)`
- `margeff([at, method, atexog, dummy, count])`
- `normalized_cov_params()`
- `predict([exog, transform])`
- `prsquared()`
- `pvalues()`
- `remove_data()`
- `resid()`
- `save(fname[, remove_data])`
- `summary([yname, xname, title, alpha, yname_list])`
- `summary2([yname, xname, title, alpha, ...])`
- `t_test(r_matrix[, q_matrix, cov_p, scale, use_t])`
- `tvalues()`
- `wald_test(r_matrix[, q_matrix, cov_p, ...])`

**static** `NegativeBinomialAncillaryResults.aic()`

**static** `NegativeBinomialAncillaryResults.bic()`
NegativeBinomialAncillaryResults.conf_int(alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

Parameters

alpha : float, optional
    The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

cols : array-like, optional
    cols specifies which confidence intervals to return

method : string
    Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

Returns

conf_int : array
    Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[-5496529.48322745, -1467987.78596704],
      [-177.02903529, 207.15277984],
      [-0.1115811, 0.03994274],
      [-3.12506664, -0.91539297],
      [-1.5179487, -0.54850503],
      [-0.56251721, 0.460309],
      [ 798.7875153, 2859.51541392]])
```

```python
>>> results.conf_int(cols=(2,3))
array([[ -0.1115811 , 0.03994274],
       [-3.12506664, -0.91539297]])
```
negativebinomialancillaryresults.cov_params

Parameters

- **r_matrix**: array-like
  - Can be 1d, or 2d. Can be used alone or with other.
- **column**: array-like, optional
  - Must be used on its own. Can be 0d or 1d see below.
- **scale**: float, optional
  - Can be specified or not. Default is None, which means that the scale argument is taken from the model.
- **other**: array-like, optional
  - Can be used when r_matrix is specified.

Returns

(The below are assumed to be in matrix notation.)

- **cov**: ndarray
  - If no argument is specified returns the covariance matrix of a model:
    (scale)*(X.T X)^(-1):
  - If contrast is specified it pre and post-multiplies as follows:
    (scale) * r_matrix (X.T X)^(-1) r_matrix.T:
  - If contrast and other are specified returns:
    (scale) * r_matrix (X.T X)^(-1) other.T:
  - If column is specified returns:
    (scale) * (X.T X)^(-1)[column,column] if column is 0d:
    OR:
    (scale) * (X.T X)^(-1)[column][:,column] if column is 1d:

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.f_test

Parameters

- **r_matrix**: array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.
**q_matrix**: array-like

This is deprecated. See *r_matrix* and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and *r_matrix* is an array, *q_matrix* is assumed to be a conformable array of zeros.

**cov_p**: array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, *self.normalized_cov_params* is used.

**scale**: float, optional

Default is 1.0 for no scaling.

**invcov**: array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**Notes**

The matrix *r_matrix* is assumed to be non-singular. More precisely, 

\[ (pX pX^T) r_matrix.T \]

is assumed invertible. Here, *pX* is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]

This tests that each coefficient is jointly statistically significantly different from zero.

```print(results.f_test(A))```  

<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>

Compare this to

```results.F 330.2853392346658
>>> results.F_p 4.98403096572e-10
```

```B = np.array(([0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]))

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols

>>> dta = longley.load_pandas().data

>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'

>>> results = ols(formula, dta).fit()

>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'

>>> f_test = results.f_test(hypotheses)

>>> print(f_test)
```

### statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.fittedvalues

```python
static NegativeBinomialAncillaryResults.fittedvalues()
```

### statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.get_margeff

```python
NegativeBinomialAncillaryResults.get_margeff(at='overall', method='dydx',
                                          atexog=None, dummy=False,
                                          count=False)
```

Get marginal effects of the fitted model.

**Parameters**

- **at**: str, optional

  Options are:
  
  - 'overall', The average of the marginal effects at each observation.
  - 'mean', The marginal effects at the mean of each regressor.
  - 'median', The marginal effects at the median of each regressor.
  - 'zero', The marginal effects at zero for each regressor.
  - 'all', The marginal effects at each observation. If `at` is all only margeff will be available from the returned object.

  Note that if `exog` is specified, then marginal effects for all variables not specified by `exog` are calculated using the `at` option.

- **method**: str, optional

  Options are:
  
  - 'dydx' - dy/dx - No transformation is made and marginal effects are returned. This is the default.
  - 'eyex' - estimate elasticities of variables in `exog` – d(lny)/d(lnx)
  - 'dyex' - estimate semielasticity – dy/d(lnx)
  - 'eydx' - estimate semeilasticity – d(lny)/dx

  Note that tranformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. 'dyex' and 'eyex' do not make sense for discrete variables.
atexog : array-like, optional

Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

dummy : bool, optional

If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.

count : bool, optional

If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

Returns DiscreteMargins : marginal effects instance

Returns an object that holds the marginal effects, standard errors, confidence intervals, etc. See statsmodels.discrete.discreteMargins.DiscreteMargins for more information.

Notes

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

```python
statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.initialize

NegativeBinomialAncillaryResults.initialize(model, params, **kwd)

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.llf

static NegativeBinomialAncillaryResults.llf()

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.llnull

static NegativeBinomialAncillaryResults.llnull()

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.llr

static NegativeBinomialAncillaryResults.llr()

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.llr_pvalue

static NegativeBinomialAncillaryResults.llr_pvalue()
```
statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.lnalpha

static NegativeBinomialAncillaryResults.lnalpha()

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.lnalpha_std_err

static NegativeBinomialAncillaryResults.lnalpha_std_err()

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.load
classmethod NegativeBinomialAncillaryResults.load(fname)
    load a pickle, (class method)
    Parameters fname : string or filehandle
                        fname can be a string to a file path or filename, or a filehandle.
    Returns unpickled instance :

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.margeff

NegativeBinomialAncillaryResults.margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.normalized_cov_params

NegativeBinomialAncillaryResults.normalized_cov_params()

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.predict

NegativeBinomialAncillaryResults.predict(exog=None, transform=True, **kwargs)
    Call self.model.predict with self.params as the first argument.
    Parameters exog : array-like, optional
                      The values for which you want to predict.
    transform : bool, optional
                If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.
    Returns See self.model.predict :

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.prsquared

static NegativeBinomialAncillaryResults.prsquared()
static NegativeBinomialAncillaryResults.pvalues()

NegativeBinomialAncillaryResults.remove_data()
remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

static NegativeBinomialAncillaryResults.resid()

Residuals

**Notes**

The residuals for Count models are defined as

\[ y - p \]

where \( p = \exp(X\beta) \). Any exposure and offset variables are also handled.

NegativeBinomialAncillaryResults.save (fname, remove_data=False)
save a pickle of this instance

**Parameters**

- **fname** : string or filehandle
  fname can be a string to a file path or filename, or a filehandle.
- **remove_data** : bool
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.
Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.summary

NegativeBinomialAncillaryResults.summary(yname=None, xname=None, title=None, alpha=0.05, yname_list=None)

Summarize the Regression Results

Parameters

- **yname**: string, optional
  - Default is y
- **xname**: list of strings, optional
  - Default is var_## for ## in p the number of regressors
- **title**: string, optional
  - Title for the top table. If not None, then this replaces the default title
- **alpha**: float
  - significance level for the confidence intervals

Returns

- **smry**: Summary instance
  - this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

- **statsmodels.iolib.summary.Summary**: class to hold summary results

statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.summary2

NegativeBinomialAncillaryResults.summary2(yname=None, xname=None, title=None, alpha=0.05, float_format='%.4f')

Experimental function to summarize regression results

Parameters

- **xname**: List of strings of length equal to the number of parameters
  - Names of the independent variables (optional)
- **yname**: string
  - Name of the dependent variable (optional)
- **title**: string, optional
  - Title for the top table. If not None, then this replaces the default title
- **alpha**: float
  - significance level for the confidence intervals
- **float_format**: string
  - print format for floats in parameters summary

Returns

- **smry**: Summary instance
this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

- `statsmodels.iolib.summary.Summary` class to hold summary results

```python
statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.t_test
```

NegativeBinomialAncillaryResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

**Parameters**

- **r_matrix** : array-like, str, tuple
  - array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like or scalar, optional
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p** : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale** : float, optional
  An optional scale to use. Default is the scale specified by the model fit.

- **use_t** : bool, optional
  If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

- `tvalues` individual t statistics
- `f_test` for F tests
- `patsy.DesignInfo.linear_constraint`

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
```
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]

r tests that the coefficients on the 5th and 6th independent variable are the same.

>>> T_Test = results.t_test(r) >>>print(T_test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9> >>> T_test.effect -1829.2025687192481 >>> T_test.sd 455.39079425193762 >>> T_test.tvalue -4.0167754636411717 >>> T_test.pvalue 0.0015163772380899498

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
```

```python
>>> print(t_test)
```

### statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.tvalues

```python
static NegativeBinomialAncillaryResults.tvalues()
Return the t-statistic for a given parameter estimate.
```

### statsmodels.discrete.discrete_model.NegativeBinomialAncillaryResults.wald_test

```python
NegativeBinomialAncillaryResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.
```

**Parameters**

- **r_matrix**: array-like, str, or tuple
  - array: An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p**: array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale**: float, optional
  Default is 1.0 for no scaling.

- **invcov**: array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

- **use_f**: bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:

- statsmodels.contrasts
- statsmodels.model.LikelihoodModelResults.f_test
- statsmodels.model.LikelihoodModelResults.t_test
- patsy.DesignInfo.linear_constraint

Notes

The matrix \( r\_matrix \) is assumed to be non-singular. More precisely,

\[
r\_matrix (pX \ pX.T) r\_matrix.T
\]

is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

DiscreteModel is a superclass of all discrete regression models. The estimation results are returned as an instance of one of the subclasses of DiscreteResults. Each category of models, binary, count and multinomial, have their own intermediate level of model and results classes. This intermediate classes are mostly to facilitate the implementation of the methods and attributes defined by DiscreteModel and DiscreteResults.

---

```python
DiscreteModel(endog, exog, **kwargs) Abstract class for discrete choice models.
DiscreteResults(model, mlefit) A results class for the discrete dependent variable models.
BinaryModel(endog, exog, **kwargs) A results class for binary data
CountModel(endog, exog[, offset, exposure, ...])
MultinomialModel(endog, exog, **kwargs)
```

---

### statsmodels.discrete.discrete_model.DiscreteModel

class statsmodels.discrete.discrete_model.DiscreteModel

Abstract class for discrete choice models.

This class does not do anything itself but lays out the methods and call signature expected of child classes in addition to those of statsmodels.model.LikelihoodModel.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdf(X)</td>
<td>The cumulative distribution function of the model.</td>
</tr>
<tr>
<td>cov(params_func_11)</td>
<td>Computes cov_params on a reduced parameter space corresponding to the nonzero parameters.</td>
</tr>
<tr>
<td>fit([start_params, method, maxiter, ...])</td>
<td>Fit the model using maximum likelihood.</td>
</tr>
<tr>
<td>fit_regularized([start_params, method, maxiter, ...])</td>
<td>Fit the model using a regularized maximum likelihood.</td>
</tr>
<tr>
<td>from_formula(formula, data[, subset])</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>hessian(params)</td>
<td>The Hessian matrix of the model</td>
</tr>
<tr>
<td>information(params)</td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td>initialize()</td>
<td>Initialize is called by</td>
</tr>
<tr>
<td>loglike(params)</td>
<td>Log-likelihood of model.</td>
</tr>
<tr>
<td>pdf(X)</td>
<td>The probability density (mass) function of the model.</td>
</tr>
<tr>
<td>predict([params, exog, linear])</td>
<td>Predict response variable of a model given exogenous variables.</td>
</tr>
<tr>
<td>score(params)</td>
<td>Score vector of model.</td>
</tr>
</tbody>
</table>
DiscreteModel.cdf(X)
The cumulative distribution function of the model.

DiscreteModel.cov_params_func_l1(likelihood_model, xopt, retvals)
Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.

Returns a full cov_params matrix, with entries corresponding to zero'd values set to np.nan.

DiscreteModel.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)
Fit the model using maximum likelihood.

Parameters start_params : array-like, optional
    Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

method : str, optional
    The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:
    • ‘newton’ for Newton-Raphson, ‘nm’ for Nelder-Mead
    • ‘bfgs’ for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
    • ‘lbfgs’ for limited-memory BFGS with optional box constraints
    • ‘powell’ for modified Powell’s method
    • ‘cg’ for conjugate gradient
    • ‘ncg’ for Newton-conjugate gradient
    • ‘basinhopping’ for global basin-hopping solver

    The explicit arguments in fit are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

maxiter : int, optional
    The maximum number of iterations to perform.

full_output : bool, optional
    Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.
disp : bool, optional
    Set to True to print convergence messages.

fargs : tuple, optional
    Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

callback : callable callback(xk), optional
    Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool, optional
    Set to True to return list of solutions at each iteration. Available in Results object’s
    mle_retvals attribute.

Notes

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

'newton'
    tol : float
        Relative error in params acceptable for convergence.
    'nm' -- Nelder Mead
        xtol : float
            Relative error in params acceptable for convergence
        ftol : float
            Relative error in loglike(params) acceptable for convergence
        maxfun : int
            Maximum number of function evaluations to make.

'bfgs'
    gtol : float
        Stop when norm of gradient is less than gtol.
    norm : float
        Order of norm (np.Inf is max, -np.Inf is min)
    epsilon
        If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

'lbfgs'
    m : int
        This many terms are used for the Hessian approximation.
    factr : float
        A stop condition that is a variant of relative error.
    pgtol : float
        A stop condition that uses the projected gradient.
    epsilon
        If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
    maxfun : int
        Maximum number of function evaluations to make.
    bounds : sequence
        (min, max) pairs for each element in x,
        defining the bounds on that parameter.
        Use None for one of min or max when there is no bound in that direction.

'cg'
gtol : float
Stop when norm of gradient is less than gtol.
norm : float
Order of norm (np.Inf is max, -np.Inf is min)
epsilon : float
If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if LikelihoodModel.score is None.

'ncg'
fhess_p : callable f’(x,*args)
Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if LikelihoodModel.hessian is None.
avextol : float
Stop when the average relative error in the minimizer falls below this amount.
epsilon : float or ndarray
If fhess is approximated, use this value for the step size. Only relevant if Likelihoodmodel.hessian is None.

'powell'
xtol : float
Line-search error tolerance
ftol : float
Relative error in loglike(params) for acceptable for convergence.
maxfun : int
Maximum number of function evaluations to make.
start_direc : ndarray
Initial direction set.

'basinhopping'
niter : integer
The number of basin hopping iterations.
niter_success : integer
Stop the run if the global minimum candidate remains the same for this number of iterations.
T : float
The "temperature" parameter for the accept or reject criterion. Higher "temperatures" mean that larger jumps in function value will be accepted. For best results 'T' should be comparable to the separation (in function value) between local minima.
stepsize : float
Initial step size for use in the random displacement.
interval : integer
The interval for how often to update the 'stepsize'.
minimizer : dict
Extra keyword arguments to be passed to the minimizer 'scipy.optimize.minimize()’, for example ‘method’ - the minimization method (e.g. 'L-BFGS-B'), or ‘tol’ - the tolerance for termination. Other arguments are mapped from explicit argument of ‘fit’:
- ‘args’ <- ‘fargs’
- ‘jac’ <- ‘score’
- ‘hess’ <- ‘hess’
Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is determined by the argument method.

**Parameters**

`start_params` : array-like, optional

Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

`method` : ‘l1’ or ‘l1_cvxopt_cp’

See notes for details.

`maxiter` : Integer or ‘defined_by_method’

Maximum number of iterations to perform. If ‘defined_by_method’, then use method defaults (see notes).

`full_output` : bool

Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

`disp` : bool

Set to True to print convergence messages.

`fargs` : tuple

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

`callback` : callable callback(xk)

Called after each iteration, as callback(xk), where xk is the current parameter vector.

`retall` : bool

Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

`alpha` : non-negative scalar or numpy array (same size as parameters)

The weight multiplying the l1 penalty term

`trim_mode` : ‘auto’, ‘size’, or ‘off’

If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim params if they have very small absolute value

`size_trim_tol` : float or ‘auto’ (default = ‘auto’)

For use when trim_mode == ‘size’

`auto_trim_tol` : float

For sue when trim_mode == ‘auto’. Use

`qc_tol` : float
Print warning and don’t allow auto trim when (ii) (above) is violated by this much.

**qc_verbose**: Boolean

If true, print out a full QC report upon failure

**Notes**

Optional arguments for the solvers (available in Results.mle_settings):

- `'l1'`
  - acc : float (default 1e-6)
    - Requested accuracy as used by slsqp
- `'l1_cvxopt_cp'`
  - abstol : float
    - absolute accuracy (default: `1e-7`).
  - reltol : float
    - relative accuracy (default: `1e-6`).
  - feastol : float
    - tolerance for feasibility conditions (default: `1e-7`).
  - refinement : int
    - number of iterative refinement steps when solving KKT equations (default: `1`).

**Optimization methodology**

With $L$ the negative log likelihood, we solve the convex but non-smooth problem

$$
\min_{\beta} L(\beta) + \sum_k \alpha_k |\beta_k|
$$

via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the “added variables” $u_k$)

$$
\min_{\beta, u} L(\beta) + \sum_k \alpha_k u_k,
$$

subject to

$$
-u_k \leq \beta_k \leq u_k.
$$

With $\partial_k L$ the derivative of $L$ in the $k^{th}$ parameter direction, theory dictates that, at the minimum, exactly one of two conditions holds:

1. $|\partial_k L| = \alpha_k$ and $\beta_k \neq 0$
2. $|\partial_k L| \leq \alpha_k$ and $\beta_k = 0$

**statsmodels.discrete.discrete_model.DiscreteModel.from_formula**

**classmethod** `DiscreteModel.from_formula(formula, data, subset=None, *args, **kwargs)`

Create a Model from a formula and dataframe.

**Parameters**

- **formula**: str or generic Formula object
The formula specifying the model

data : array-like
   The data for the model. See Notes.
subset : array-like
   An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

args : extra arguments
   These are passed to the model
kwags : extra keyword arguments
   These are passed to the model.

Returns model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

statsmodels.discrete.discrete_model.DiscreteModel.hessian

DiscreteModel.hessian (params)
   The Hessian matrix of the model

statsmodels.discrete.discrete_model.DiscreteModel.information

DiscreteModel.information (params)
   Fisher information matrix of model
   Returns -Hessian of loglike evaluated at params.

statsmodels.discrete.discrete_model.DiscreteModel.initialize

DiscreteModel.initialize ()
   Initialize is called by statsmodels.model.LikelihoodModel.__init__ and should contain any preprocessing that needs to be done for a model.

statsmodels.discrete.discrete_model.DiscreteModel.loglike

DiscreteModel.loglike (params)
   Log-likelihood of model.

statsmodels.discrete.discrete_model.DiscreteModel.pdf

DiscreteModel.pdf (X)
   The probability density (mass) function of the model.
**statsmodels.discrete.discrete_model.DiscreteModel.predict**

DiscreteModel.predict (params, exog=None, linear=False)

Predict response variable of a model given exogenous variables.

**statsmodels.discrete.discrete_model.DiscreteModel.score**

DiscreteModel.score (params)

Score vector of model.

The gradient of logL with respect to each parameter.

**Attributes**

- endog_names
- exog_names

**statsmodels.discrete.discrete_model.DiscreteResults**

class statsmodels.discrete.discrete_model.DiscreteResults (model, mlefit)

A results class for the discrete dependent variable models.

**Parameters**

- **model**: A DiscreteModel instance
- **params**: array-like
  The parameters of a fitted model.
- **hessian**: array-like
  The hessian of the fitted model.
- **scale**: float
  A scale parameter for the covariance matrix.

**Returns**

- **Attributes**
  - **aic**: float
    Akaike information criterion. \(-2*(llf - p)\) where p is the number of regressors including the intercept.
  - **bic**: float
    Bayesian information criterion. \(-2*'llf' + ln(nobs)*p\) where p is the number of regressors including the intercept.
  - **bse**: array
    The standard errors of the coefficients.
  - **df_resid**: float
    See model definition.
  - **df_model**: float
    See model definition.
**fitted_values** : array
Linear predictor $XB$.

**llf** : float
Value of the loglikelihood

**llnull** : float
Value of the constant-only loglikelihood

**llr** : float
Likelihood ratio chi-squared statistic; $-2*(\text{llnull} - \text{llf})$

**llr_pvalue** : float
The chi-squared probability of getting a log-likelihood ratio statistic greater than llr. llr has a chi-squared distribution with degrees of freedom $df_{\text{model}}$.

**prsquared** : float
McFadden’s pseudo-R-squared. $1 - (\text{llf/llnull})$

### Methods

- `aic()`: Returns the confidence interval of the fitted parameters.
- `bic()`: Returns the variance/covariance matrix.
- `bse()`: Compute the F-test for a joint linear hypothesis.
- `conf_int([alpha, cols, method])`: Get marginal effects of the fitted model.
- `cov_params([r_matrix, column, scale, cov_p, ...])`: Returns the variance/covariance matrix.
- `f_test(r_matrix[, q_matrix, cov_p, scale, ...])`: Call self.model.predict with self.params as the first argument.
- `fittedvalues()`: remove data arrays, all nobs arrays from result and model save a pickle of this instance
- `get_margeff([at, method, atexog, dummy, count])`: Summarize the Regression Results
- `initialize(model, params, **kwd)`: Experimental function to summarize regression results
- `llf()`: Call self.model.predict with self.params as the first argument.
- `llnull()`: Compute a t-test for a joint linear hypothesis of the form $Rb = q$
- `llr()`: Return the t-statistic for a given parameter estimate.
- `llr_pvalue()`: Compute a Wald-test for a joint linear hypothesis.
- `load(fname)`: remove data arrays, all nobs arrays from result and model save a pickle of this instance
- `margeff([at, method, atexog, dummy, count])`: Experimental function to summarize regression results
- `normalized_cov_params()`: Compute a t-test for a joint linear hypothesis of the form $Rb = q$
- `predict([exog, transform])`: Return the t-statistic for a given parameter estimate.
- `prsquared()`: Compute a Wald-test for a joint linear hypothesis.
- `pvalues()`: remove data arrays, all nobs arrays from result and model save a pickle of this instance
- `remove_data()`: Summarize the Regression Results
- `save(fname[, remove_data])`: Experimental function to summarize regression results
- `summary([yname, xname, title, alpha, yname_list])`: Summarize the Regression Results
- `summary2([yname, xname, title, alpha, ...])`: Experimental function to summarize regression results
- `t_test(r_matrix[, q_matrix, cov_p, scale, use_t])`: Compute a t-test for a joint linear hypothesis of the form $Rb = q$
- `tvalues()`: Return the t-statistic for a given parameter estimate.
- `wald_test(r_matrix[, q_matrix, cov_p, ...])`: Compute a Wald-test for a joint linear hypothesis.

```python
statsmodels.discrete.discrete_model.DiscreteResults.aic

static DiscreteResults.aic()```

### 3.5. Regression with Discrete Dependent Variable
DiscreteResults.conf_int (alpha=0.05, cols=None, method='default')
Returns the confidence interval of the fitted parameters.

Parameters

- **alpha**: float, optional
  The alpha level for the confidence interval. IE., The default alpha = .05 returns a 95% confidence interval.
- **cols**: array-like, optional
  cols specifies which confidence intervals to return
- **method**: string
  Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

Returns

- **conf_int**: array
  Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[5496529.48322745, -1467987.78596704],
      [177.02903529, 207.15277984],
      [-0.1115811, 0.03994274],
      [-3.12506664, -0.91539297],
      [-1.5179487, -0.54850503],
      [-0.56251721, 0.460309],
      [798.7875153, 2859.51541392]])
```
DiscreteResults.cov_params

DiscreteResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters

- **r_matrix**: array-like
  - Can be 1d, or 2d. Can be used alone or with other.

- **column**: array-like, optional
  - Must be used on its own. Can be 0d or 1d see below.

- **scale**: float, optional
  - Can be specified or not. Default is None, which means that the scale argument is taken from the model.

- **other**: array-like, optional
  - Can be used when r_matrix is specified.

Returns (The below are assumed to be in matrix notation.):

- **cov**: ndarray
  - If no argument is specified returns the covariance matrix of a model:
    - (scale)*(X.T X)^(-1):
  - If contrast is specified it pre and post-multiplies as follows:
    - (scale) * r_matrix (X.T X)^(-1) r_matrix.T:
  - If contrast and other are specified returns:
    - (scale) * r_matrix (X.T X)^(-1) other.T:
  - If column is specified returns:
    - (scale) * (X.T X)^(-1)[column,column] if column is 0d:
    - OR:
    - (scale) * (X.T X)^(-1)[column][:,column] if column is 1d:

DiscreteResults.f_test

DiscreteResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)

Compute the F-test for a joint linear hypothesis.

Parameters

- **r_matrix**: array-like, str, or tuple
  - array: An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like
This is deprecated. See \texttt{r\_matrix} and the examples for more information on new usage. Can be either a scalar or a length \( p \) row vector. If omitted and \( r\_matrix \) is an array, \( q\_matrix \) is assumed to be a conformable array of zeros.

**cov\_p** : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized\_cov\_params is used.

**scale** : float, optional

Default is 1.0 for no scaling.

**invcov** : array-like, optional

A \( q \times q \) array to specify an inverse covariance matrix based on a restrictions matrix.

See also:

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix \( r\_matrix \) is assumed to be non-singular. More precisely, \( r\_matrix (pX pX.T) r\_matrix.T \) is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
```

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
df_denom=9, df_num=6>
```

Compare this to

```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10
```

```python
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
```

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.
Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

**statsmodels.discrete.discrete_model.DiscreteResults.fittedvalues**

```python
static DiscreteResults.fittedvalues()
```

**statsmodels.discrete.discrete_model.DiscreteResults.get_margeff**

```python
DiscreteResults.get_margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)
```

Get marginal effects of the fitted model.

**Parameters**

- **at** : str, optional
  Options are:
  - 'overall', The average of the marginal effects at each observation.
  - 'mean', The marginal effects at the mean of each regressor.
  - 'median', The marginal effects at the median of each regressor.
  - 'zero', The marginal effects at zero for each regressor.
  - 'all', The marginal effects at each observation. If `at` is all only `margeff` will be available from the returned object.

  Note that if `exog` is specified, then marginal effects for all variables not specified by `exog` are calculated using the `at` option.

- **method** : str, optional
  Options are:
  - 'dydx' - dy/dx - No transformation is made and marginal effects are returned. This is the default.
  - 'eyex' - estimate elasticities of variables in `exog` – d(lny)/d(lnx)
  - 'dyex' - estimate semielasticity – dy/d(lnx)
  - 'eydx' - estimate semielasticity – d(lny)/dx

  Note that transformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. ‘dyex’ and ‘eyex’ do not make sense for discrete variables.

- **atexog** : array-like, optional
Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

**dummy**: bool, optional

If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.

**count**: bool, optional

If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

**Returns** DiscreteMargins : marginal effects instance

Returns an object that holds the marginal effects, standard errors, confidence intervals, etc. See `statsmodels.discrete.discrete_margins.DiscreteMargins` for more information.

**Notes**

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

```python
statsmodels.discrete.discrete_model.DiscreteResults.initialize

DiscreteResults.initialize(model, params, **kwd)
```

```python
statsmodels.discrete.discrete_model.DiscreteResults.llf

static DiscreteResults.llf()
```

```python
statsmodels.discrete.discrete_model.DiscreteResults.llnull

static DiscreteResults.llnull()
```

```python
statsmodels.discrete.discrete_model.DiscreteResults.llr

static DiscreteResults.llr()
```

```python
statsmodels.discrete.discrete_model.DiscreteResults.llr_pvalue

static DiscreteResults.llr_pvalue()
```
**statsmodels.discrete.discrete_model.DiscreteResults.load**

**classmethod** `DiscreteResults.load(fname)`  
load a pickle, (class method)

**Parameters**  
fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

**Returns**  
unpickled instance :

**statsmodels.discrete.discrete_model.DiscreteResults.margeff**

`DiscreteResults.margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)`  

**statsmodels.discrete.discrete_model.DiscreteResults.normalized_cov_params**

`DiscreteResults.normalized_cov_params()`  

**statsmodels.discrete.discrete_model.DiscreteResults.predict**

`DiscreteResults.predict(exog=None, transform=True, *args, **kwargs)`  
Call self.model.predict with self.params as the first argument.

**Parameters**  
exog : array-like, optional  
The values for which you want to predict.
transform : bool, optional  
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

**Returns**  
See self.model.predict :

**statsmodels.discrete.discrete_model.DiscreteResults.prsquared**

**static** `DiscreteResults.prsquared()`  

**statsmodels.discrete.discrete_model.DiscreteResults.pvalues**

**static** `DiscreteResults.pvalues()`  

**statsmodels.discrete.discrete_model.DiscreteResults.remove_data**

`DiscreteResults.remove_data()`  
remove data arrays, all nobs arrays from result and model
This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**3.5. Regression with Discrete Dependent Variable**
Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

statsmodels.discrete.discrete_model.DiscreteResults.save

DiscreteResults.save(fname, remove_data=False)
save a pickle of this instance

Parameters

fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

remove_data : bool
If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

statsmodels.discrete.discrete_model.DiscreteResults.summary

DiscreteResults.summary(yname=None, xname=None, title=None, alpha=0.05, yname_list=None)
Summarize the Regression Results

Parameters

yname : string, optional
Default is y

xname : list of strings, optional
Default is var_## for ## in p the number of regressors

title : string, optional
Title for the top table. If not None, then this replaces the default title

alpha : float
significance level for the confidence intervals

Returns

smry : Summary instance
this holds the summary tables and text, which can be printed or converted to various output formats.

See also:
**statsmodels.iolib.summary.Summary** class to hold summary results

**statsmodels.discrete.discrete_model.DiscreteResults.summary2**

DiscreteResults.summary2 (yname=None, xname=None, title=None, alpha=0.05, float_format='%.4f')

Experimental function to summarize regression results

**Parameters**

- **xname**: List of strings of length equal to the number of parameters
  - Names of the independent variables (optional)
- **yname**: string
  - Name of the dependent variable (optional)
- **title**: string, optional
  - Title for the top table. If not None, then this replaces the default title
- **alpha**: float
  - significance level for the confidence intervals
- **float_format**: string:
  - print format for floats in parameters summary

**Returns**

- **smry**: Summary instance
  - this holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

**statsmodels.iolib.summary.Summary** class to hold summary results

**statsmodels.discrete.discrete_model.DiscreteResults.t_test**

DiscreteResults.t_test (r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)

Compute a t-test for a joint linear hypothesis of the form Rb = q

**Parameters**

- **r_matrix**: array-like, str, tuple
  - array: If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix**: array-like or scalar, optional
  - This is deprecated. See r_matrix and the examples for more information on new usage.
  - Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p**: array-like, optional
  - An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale**: float, optional

---

**3.5. Regression with Discrete Dependent Variable**
An optional *scale* to use. Default is the scale specified by the model fit.

**use_t**: bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

- `tvalues`  individual t statistics
- `f_test`  for F tests

*patsy.DesignInfo.linear_constraint*

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm

>>> data = sm.datasets.longley.load()

>>> data.exog = sm.add_constant(data.exog)

>>> results = sm.OLS(data.endog, data.exog).fit()

>>> r = np.zeros_like(results.params)

>>> r[5:] = [1,-1]

>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_test)  
<T contrast: effect=-1829.2025687192481, 
sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```Python

```python
>>> T_test.effect -1829.2025687192481 >>> T_test.sd 455.39079425193762 >>> T_test.tvalue -4.0167754636411717 >>> T_test.pvalue 0.0015163772380899498
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data

>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'

>>> results = ols(formula, dta).fit()

>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'

>>> t_test = results.t_test(hypotheses)

>>> print(t_test)
```

```python
statsmodels.discrete.discrete_model.DiscreteResults.tvalues

**static** DiscreteResults.tvalues()

Return the t-statistic for a given parameter estimate.

```python
statsmodels.discrete.discrete_model.DiscreteResults.wald_test

DiscreteResults.wald_test( r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)

Compute a Wald-test for a joint linear hypothesis.

**Parameters**

- **r_matrix**: array-like, str, or tuple
array: An r x k array where r is the number of restrictions to test and k is the number of regressors.

str: The full hypotheses to test can be given as a string. See the examples.

tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix: array-like

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p: array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale: float, optional

Default is 1.0 for no scaling.

invcov: array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f: bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely, r_matrix (pX pX.T) r_matrix.T is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
Table 3.131 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>information(params)</td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td>initialize()</td>
<td>Initialize is called by</td>
</tr>
<tr>
<td>loglike(params)</td>
<td>Log-likelihood of model.</td>
</tr>
<tr>
<td>pdf(X)</td>
<td>The probability density (mass) function of the model.</td>
</tr>
<tr>
<td>predict(params[, exog, linear])</td>
<td>Predict response variable of a model given exogenous variables.</td>
</tr>
<tr>
<td>score(params)</td>
<td>Score vector of model.</td>
</tr>
</tbody>
</table>

**statsmodels.discrete.discrete_model.BinaryModel.cdf**

```python
BinaryModel.cdf(X)
```

The cumulative distribution function of the model.

**statsmodels.discrete.discrete_model.BinaryModel.cov_params_func_l1**

```python
BinaryModel.cov_params_func_l1(likelihood_model, xopt, retvals)
```

Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.

Returns a full cov_params matrix, with entries corresponding to zero’d values set to np.nan.

**statsmodels.discrete.discrete_model.BinaryModel.fit**

```python
BinaryModel.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)
```

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

Fit method for likelihood based models

**Parameters**

- **start_params**: array-like, optional
  Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

- **method**: str, optional
  The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:
  - ‘newton’ for Newton-Raphson, ‘nm’ for Nelder-Mead
  - ‘bfgs’ for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
  - ‘lbfgs’ for limited-memory BFGS with optional box constraints
  - ‘powell’ for modified Powell’s method
  - ‘cg’ for conjugate gradient
  - ‘ncg’ for Newton-conjugate gradient
  - ‘basinhopping’ for global basin-hopping solver

The explicit arguments in fit are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.
maxiter : int, optional

The maximum number of iterations to perform.

full_output : bool, optional

Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

disp : bool, optional

Set to True to print convergence messages.

fargs : tuple, optional

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

callback : callable callback(xk), optional

Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool, optional

Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

Notes

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

'newton'

tol : float

Relative error in params acceptable for convergence.

'nm' -- Nelder Mead

xtol : float

Relative error in params acceptable for convergence

ftol : float

Relative error in loglike(params) acceptable for convergence

maxfun : int

Maximum number of function evaluations to make.

'bfgs'

gtol : float

Stop when norm of gradient is less than gtol.

norm : float

Order of norm (np.Inf is max, -np.Inf is min)

epsilon

If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

'lbfgs'

m : int

This many terms are used for the Hessian approximation.

factr : float

A stop condition that is a variant of relative error.

pgtol : float

A stop condition that uses the projected gradient.

epsilon

If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
maxfun : int
   Maximum number of function evaluations to make.
 bounds : sequence
   (min, max) pairs for each element in x,
   defining the bounds on that parameter.
   Use None for one of min or max when there is no bound
   in that direction.

'cg'
   gtol : float
      Stop when norm of gradient is less than gtol.
   norm : float
      Order of norm (np.Inf is max, -np.Inf is min)
   epsilon : float
      If fprime is approximated, use this value for the step
      size. Can be scalar or vector. Only relevant if
      Likelihoodmodel.score is None.

'nag'
   fhess_p : callable f'(x,*args)
      Function which computes the Hessian of f times an arbitrary
      vector, p. Should only be supplied if
      LikelihoodModel.hessian is None.
   avextol : float
      Stop when the average relative error in the minimizer
      falls below this amount.
   epsilon : float or ndarray
      If fhess is approximated, use this value for the step size.
      Only relevant if Likelihoodmodel.hessian is None.

'powell'
   xtol : float
      Line-search error tolerance
   ftol : float
      Relative error in loglike(params) for acceptable for
      convergence.
   maxfun : int
      Maximum number of function evaluations to make.
   start_direc : ndarray
      Initial direction set.

'basinhopping'
   niter : integer
      The number of basin hopping iterations.
   niter_success : integer
      Stop the run if the global minimum candidate remains the
      same for this number of iterations.
   T : float
      The "temperature" parameter for the accept or reject
      criterion. Higher "temperatures" mean that larger jumps
      in function value will be accepted. For best results
      'T' should be comparable to the separation (in function
      value) between local minima.
   stepsize : float
      Initial step size for use in the random displacement.
   interval : integer
      The interval for how often to update the 'stepsize'.
   minimizer : dict
      Extra keyword arguments to be passed to the minimizer
      'scipy.optimize.minimize()', for example 'method' - the
      minimization method (e.g. 'L-BFGS-B'), or 'tol' - the
      tolerance for termination. Other arguments are mapped from
explicit argument of `fit`:
- `args` <- `fargs`
- `jac` <- `score`
- `hess` <- `hess`

```python
class BinaryModel:
    def fit_regularized(self, start_params=None, method='l1', maxiter='defined_by_method', full_output=1, disp=1, callback=None, retall=False, alpha=0, trim_mode='auto', auto_trim_tol=0.01, size_trim_tol=0.0001, qc_tol=0.03, **kwargs):
```

Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is determined by the argument method.

**Parameters**

- `start_params`: array-like, optional
  Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

- `method`: `l1` or `l1_cvxopt_cp`
  See notes for details.

- `maxiter`: Integer or `defined_by_method`
  Maximum number of iterations to perform. If `defined_by_method`, then use method defaults (see notes).

- `full_output`: bool
  Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

- `disp`: bool
  Set to True to print convergence messages.

- `fargs`: tuple
  Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

- `callback`: callable callback(xk)
  Called after each iteration, as callback(xk), where xk is the current parameter vector.

- `retall`: bool
  Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

- `alpha`: non-negative scalar or numpy array (same size as parameters)
  The weight multiplying the l1 penalty term

- `trim_mode`: ‘auto’, ‘size’, or ‘off’
  If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim params if they have very small absolute value

- `size_trim_tol`: float or ‘auto’ (default = ‘auto’)
  For use when trim_mode == ‘size’
auto_trim_tol : float
    For sue when trim_mode == 'auto'. Use
qc_tol : float
    Print warning and don’t allow auto trim when (ii) (above) is violated by this much.
qc_verbose : Boolean
    If true, print out a full QC report upon failure

Notes

Optional arguments for the solvers (available in Results.mle_settings):

'lsq:
    acc : float (default 1e-6)
        Requested accuracy as used by slsqp
'l1_cvxopt_cp'
    abstol : float
        absolute accuracy (default: 1e-7).
    reltol : float
        relative accuracy (default: 1e-6).
    feastol : float
        tolerance for feasibility conditions (default: 1e-7).
    refinement : int
        number of iterative refinement steps when solving KKT
equations (default: 1).

Optimization methodology

With $L$ the negative log likelihood, we solve the convex but non-smooth problem

$$
\min_{\beta} L(\beta) + \sum_k \alpha_k |\beta_k|
$$

via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the
“added variables” $u_k$)

$$
\min_{\beta, u} L(\beta) + \sum_k \alpha_k u_k,
$$

subject to

$$
-u_k \leq \beta_k \leq u_k.
$$

With $\partial_k L$ the derivative of $L$ in the $k^{th}$ parameter direction, theory dictates that, at the minimum, exactly
one of two conditions holds:

1. $|\partial_k L| = \alpha_k$ and $\beta_k \neq 0$
2. $|\partial_k L| \leq \alpha_k$ and $\beta_k = 0$
classmethod BinaryModel.from_formula(formula, data, subset=None, *args, **kwargs)
    Create a Model from a formula and dataframe.

    Parameters:
    formula : str or generic Formula object
        The formula specifying the model
    data : array-like
        The data for the model. See Notes.
    subset : array-like
        An array-like object of booleans, integers, or index values that indicate the subset of df
to use in the model. Assumes df is a pandas.DataFrame
    args : extra arguments
        These are passed to the model
    kwargs : extra keyword arguments
        These are passed to the model.

    Returns:
    model : Model instance

    Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the
model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

BinaryModel.hessian(params)
    The Hessian matrix of the model

BinaryModel.information(params)
    Fisher information matrix of model

    Returns -Hessian of loglike evaluated at params.

BinaryModel.initialize()
    Initialize is called by statsmodels.model.LikelihoodModel.__init__ and should contain any preprocessing
that needs to be done for a model.

BinaryModel.loglike(params)
    Log-likelihood of model.
BinaryModel.pdf \( (X) \)

The probability density (mass) function of the model.

**BinaryModel.predict**

BinaryModel.predict \( (\text{params}, \text{exog} = \text{None}, \text{linear} = \text{False}) \)

Predict response variable of a model given exogenous variables.

**Parameters**

- **params**: array-like
  - Fitted parameters of the model.
- **exog**: array-like
  - 1d or 2d array of exogenous values. If not supplied, the whole exog attribute of the model is used.
- **linear**: bool, optional
  - If True, returns the linear predictor dot(exog, params). Else, returns the value of the cdf at the linear predictor.

**Returns**

- **array**: Fitted values at exog.

**BinaryModel.score**

BinaryModel.score \( (\text{params}) \)

Score vector of model.

The gradient of logL with respect to each parameter.

**Attributes**

- **endog_names**
- **exog_names**

**BinaryResults**

class statsmodels.discrete.discrete_model.BinaryResults \( (\text{model}, \text{mlefit}) \)

A results class for binary data

**Parameters**

- **model**: A DiscreteModel instance
- **params**: array-like
  - The parameters of a fitted model.
- **hessian**: array-like
  - The hessian of the fitted model.
- **scale**: float
A scale parameter for the covariance matrix.

**Returns** *Attributes*:

- **aic**: float
  
  Akaike information criterion. \(-2^{*}(llf - p)\) where \(p\) is the number of regressors including the intercept.

- **bic**: float
  
  Bayesian information criterion. \(-2^{*}llf + \ln(nobs)*p\) where \(p\) is the number of regressors including the intercept.

- **bse**: array
  
  The standard errors of the coefficients.

- **df_resid**: float
  
  See model definition.

- **df_model**: float
  
  See model definition.

- **fitted_values**: array
  
  Linear predictor \(XB\).

- **llf**: float
  
  Value of the loglikelihood

- **llnull**: float
  
  Value of the constant-only loglikelihood

- **llr**: float
  
  Likelihood ratio chi-squared statistic; \(-2^{*}(llnull - llf)\)

- **llr_pvalue**: float
  
  The chi-squared probability of getting a log-likelihood ratio statistic greater than \(llr\). \(llr\) has a chi-squared distribution with degrees of freedom \(df_{model}\).

- **prssquared**: float
  
  McFadden's pseudo-R-squared. \(1 - (llf/llnull)\)

**Methods**

- **aic()**
- **bic()**
- **bse()**
- **conf_int([alpha, cols, method])**
  
  Returns the confidence interval of the fitted parameters.

- **cov_params([r_matrix, column, scale, cov_p, ...])**
  
  Returns the variance/covariance matrix.

- **f_test(r_matrix[, q_matrix, cov_p, scale, ...])**
  
  Compute the F-test for a joint linear hypothesis.

- **fittedvalues()**

- **get_margeff([at, method, atexog, dummy, count])**
  
  Get marginal effects of the fitted model.

- **initialize(model, params, **kwd)**

- **llf()**

Continued on next page
Table 3.133 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>llnull()</td>
<td>load a pickle, (class method)</td>
</tr>
<tr>
<td>llr()</td>
<td></td>
</tr>
<tr>
<td>llr_pvalue()</td>
<td></td>
</tr>
<tr>
<td>load(fname)</td>
<td></td>
</tr>
<tr>
<td>margeff([at, method, atexog, dummy, count])</td>
<td></td>
</tr>
<tr>
<td>normalized_cov_params()</td>
<td></td>
</tr>
<tr>
<td>pred_table([threshold])</td>
<td>Prediction table</td>
</tr>
<tr>
<td>predict([exog, transform])</td>
<td>Call self.model.predict with self.params as the first argument.</td>
</tr>
<tr>
<td>prsquared()</td>
<td></td>
</tr>
<tr>
<td>pvalues()</td>
<td></td>
</tr>
<tr>
<td>remove_data()</td>
<td>remove data arrays, all nobs arrays from result and model</td>
</tr>
<tr>
<td>resid()</td>
<td></td>
</tr>
<tr>
<td>resid_dev()</td>
<td>Deviance residuals ..</td>
</tr>
<tr>
<td>resid_pearson()</td>
<td>Pearson residuals ..</td>
</tr>
<tr>
<td>resid_response()</td>
<td>The response residuals ..</td>
</tr>
<tr>
<td>save(fname[, remove_data])</td>
<td>save a pickle of this instance</td>
</tr>
<tr>
<td>summary([yname, xname, title, alpha, yname_list])</td>
<td>Summarize the Regression Results</td>
</tr>
<tr>
<td>summary2([yname, xname, title, alpha, ...])</td>
<td>Experimental function to summarize regression results</td>
</tr>
<tr>
<td>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</td>
<td>Compute a t-test for a joint linear hypothesis of the form $Rb = q$</td>
</tr>
<tr>
<td>tvalues()</td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td>wald_test(r_matrix[, q_matrix, cov_p, ...])</td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
</tbody>
</table>

```python
statsmodels.discrete.discrete_model.BinaryResults.aic

static BinaryResults.aic()

statsmodels.discrete.discrete_model.BinaryResults.bic

static BinaryResults.bic()

statsmodels.discrete.discrete_model.BinaryResults.bse

static BinaryResults.bse()

statsmodels.discrete.discrete_model.BinaryResults.conf_int

BinaryResults.conf_int(alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

**Parameters**

- **alpha** : float, optional
  
The alpha level for the confidence interval. ie., The default $\alpha = 0.05$ returns a 95% confidence interval.

- **cols** : array-like, optional
  
  $cols$ specifies which confidence intervals to return

- **method** : string
  
  Not Implemented Yet Method to estimate the confidence interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jij” : “jac” : “boot-bse” “boot_quant” “profile”
```
Returns `conf_int` : array

Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different
distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[ -5496529.48322745,  -1467987.78596704],
       [-177.02903529,    207.15277984],
       [  -0.1115811 ,     0.03994274],
       [ -3.12506664,    -0.91539297],
       [-1.5179487 ,     -0.54850503],
       [-0.56251721,     0.460309    ],
       [ 798.7875153 ,   2859.51541392]])
```

```python
>>> results.conf_int(cols=(2,3))
array([[-0.1115811 ,  0.03994274],
       [-3.12506664,  207.15277984]])
```

`statsmodels.discrete.discrete_model.BinaryResults.cov_params`

BinaryResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multi-
plied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

**Parameters**

- **r_matrix** : array-like
  Can be 1d, or 2d. Can be used alone or with other.

- **column** : array-like, optional
  Must be used on its own. Can be 0d or 1d see below.

- **scale** : float, optional
  Can be specified or not. Default is None, which means that the scale argument is taken from the model.

- **other** : array-like, optional
  Can be used when r_matrix is specified.

**Returns**

(The below are assumed to be in matrix notation.)

- **cov** : ndarray
  If no argument is specified returns the covariance matrix of a model :
(scale)*(X.T X)^(-1) :

If contrast is specified it pre and post-multiplies as follows :

(scale) * r_matrix (X.T X)^(-1) r_matrix.T :

If contrast and other are specified returns :

(scale) * r_matrix (X.T X)^(-1) other.T :

If column is specified returns :

(scalenew) * (X.T X)^(-1)(column,column) if column is 0d :

OR :

(scalenew) * (X.T X)^(-1)(column)[;column] if column is 1d :

```
statsmodels.discrete.discrete_model.BinaryResults.f_test
```

BinaryResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)

Compute the F-test for a joint linear hypothesis.

**Parameters**

- **r_matrix** : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like
  
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p** : array-like, optional
  
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale** : float, optional
  
  Default is 1.0 for no scaling.

- **invcov** : array-like, optional
  
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**See also:**

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

**Notes**

The matrix r_matrix is assumed to be non-singular. More precisely, r_matrix (pX pX.T) r_matrix.T is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.

```print(results.f_test(A))
< F contrast: F=330.28533923463488, p=4.98403052872e-10,
df_denom=9, df_num=6>
``` Compare this to

```python
results.F
330.2853392346658
results.F_p
4.98403096572e-10
``` >>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
print(results.f_test(B))
< F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
df_num=2>
``` Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```
statsmodels.discrete.discrete_model.BinaryResults.fittedvalues

static BinaryResults.fittedvalues()

statsmodels.discrete.discrete_model.BinaryResults.get_margeff

BinaryResults.get_margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)

Get marginal effects of the fitted model.

Parameters at : str, optional

Options are:

- ‘overall’, The average of the marginal effects at each observation.
• ‘mean’, The marginal effects at the mean of each regressor.
• ‘median’, The marginal effects at the median of each regressor.
• ‘zero’, The marginal effects at zero for each regressor.
• ‘all’, The marginal effects at each observation. If at is all only margeff will be available from the returned object.

Note that if exog is specified, then marginal effects for all variables not specified by exog are calculated using the at option.

**method** : str, optional
Options are:
• ‘dydx’ - dy/dx - No transformation is made and marginal effects are returned. This is the default.
• ‘eyex’ - estimate elasticities of variables in exog – d(lny)/d(lnx)
• ‘dyex’ - estimate semielasticity – dy/d(lnx)
• ‘eydx’ - estimate semeilasticity – d(lny)/dx

Note that tranformations are done after each observation is calculated. Semi-elasticities for binary variables are computed using the midpoint method. ‘dyex’ and ‘eyex’ do not make sense for discrete variables.

**atexog** : array-like, optional
Optionally, you can provide the exogenous variables over which to get the marginal effects. This should be a dictionary with the key as the zero-indexed column number and the value of the dictionary. Default is None for all independent variables less the constant.

**dummy** : bool, optional
If False, treats binary variables (if present) as continuous. This is the default. Else if True, treats binary variables as changing from 0 to 1. Note that any variable that is either 0 or 1 is treated as binary. Each binary variable is treated separately for now.

**count** : bool, optional
If False, treats count variables (if present) as continuous. This is the default. Else if True, the marginal effect is the change in probabilities when each observation is increased by one.

**Returns** **DiscreteMargins** : marginal effects instance

Returns an object that holds the marginal effects, standard errors, confidence intervals, etc. See *statsmodels.discrete.discrete_model.DiscreteMargins* for more information.

**Notes**

When using after Poisson, returns the expected number of events per period, assuming that the model is loglinear.

*statsmodels.discrete.discrete_model.BinaryResults.initialize*

**BinaryResults.initialize** *(model, params, **kwds)*
statsmodels.discrete.discrete_model.BinaryResults.llf

    static BinaryResults.llf()

statsmodels.discrete.discrete_model.BinaryResults.llnull

    static BinaryResults.llnull()

statsmodels.discrete.discrete_model.BinaryResults.llr

    static BinaryResults.llr()

statsmodels.discrete.discrete_model.BinaryResults.llr_pvalue

    static BinaryResults.llr_pvalue()

statsmodels.discrete.discrete_model.BinaryResults.load

    classmethod BinaryResults.load(fname)
    load a pickle, (class method)

        Parameters  fname : string or filehandle
                        fname can be a string to a file path or filename, or a filehandle.

        Returns  unpickled instance :

statsmodels.discrete.discrete_model.BinaryResults.margeff

    BinaryResults.margeff(at='overall', method='dydx', atexog=None, dummy=False, count=False)

statsmodels.discrete.discrete_model.BinaryResults.normalized_cov_params

    BinaryResults.normalized_cov_params()

statsmodels.discrete.discrete_model.BinaryResults.pred_table

    BinaryResults.pred_table(threshold=0.5)
    Prediction table

        Parameters  threshold : scalar
                            Number between 0 and 1. Threshold above which a prediction is considered 1 and
                            below which a prediction is considered 0.
Notes

pred_table[i,j] refers to the number of times “i” was observed and the model predicted “j”. Correct predictions are along the diagonal.

statsmodels.discrete.discrete_model.BinaryResults.predict

BinaryResults.predict(exog=None, transform=True, *args, **kwargs)

Call self.model.predict with self.params as the first argument.

Parameters

exog : array-like, optional
  The values for which you want to predict.

transform : bool, optional
  If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

Returns

See self.model.predict :

statsmodels.discrete.discrete_model.BinaryResults.prsquared

static BinaryResults.prsquared()

statsmodels.discrete.discrete_model.BinaryResults.pvalues

static BinaryResults.pvalues()

statsmodels.discrete.discrete_model.BinaryResults.remove_data

BinaryResults.remove_data()

remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

statsmodels.discrete.discrete_model.BinaryResults.resid

static BinaryResults.resid()
Deviance residuals are defined

\[ d_j = \pm \left( 2 \left[ Y_j \ln \left( \frac{Y_j}{M_j p_j} \right) + (M_j - Y_j) \ln \left( \frac{M_j - Y_j}{M_j(1 - p_j)} \right) \right] \right)^{1/2} \]

where

\[ p_j = cdf(X\beta) \] and \( M_j \) is the total number of observations sharing the covariate pattern \( j \).

For now \( M_j \) is always set to 1.

Pearson residuals are defined to be

\[ r_j = \frac{(y - M_j p_j)}{\sqrt{M_j p_j(1 - p_j)}} \]

where \( p_j = cdf(X\beta) \) and \( M_j \) is the total number of observations sharing the covariate pattern \( j \).

For now \( M_j \) is always set to 1.

Response residuals are defined to be

\[ y - p \]

where \( p = cdf(X\beta) \).
statsmodels.discrete.discrete_model.BinaryResults.save

```
BinaryResults.save(fname, remove_data=False)
```

save a pickle of this instance

**Parameters**

- **fname** : string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- **remove_data** : bool
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

statsmodels.discrete.discrete_model.BinaryResults.summary

```
BinaryResults.summary(yname=None, xname=None, title=None, alpha=0.05, yname_list=None)
```

Summarize the Regression Results

**Parameters**

- **yname** : string, optional
  
  Default is y

- **xname** : list of strings, optional
  
  Default is var_## for ## in p the number of regressors

- **title** : string, optional
  
  Title for the top table. If not None, then this replaces the default title

- **alpha** : float
  
  significance level for the confidence intervals

**Returns**

- **smry** : Summary instance
  
  this holds the summary tables and text, which can be printed or converted to various output formats.

**See also:**

- `statsmodels.iolib.summary.Summary` class to hold summary results

statsmodels.discrete.discrete_model.BinaryResults.summary2

```
BinaryResults.summary2(yname=None, xname=None, title=None, alpha=0.05, float_format='%.4f')
```

Experimental function to summarize regression results

**Parameters**

- **xname** : List of strings of length equal to the number of parameters
  
  Names of the independent variables (optional)
yname : string
    Name of the dependent variable (optional)

title : string, optional
    Title for the top table. If not None, then this replaces the default title

alpha : float
    significance level for the confidence intervals

float_format: string :
    print format for floats in parameters summary

Returns smry : Summary instance
    this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.discrete.discrete_model.BinaryResults.t_test

BinaryResults.t_test (r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)
    Compute a t-test for a joint linear hypothesis of the form Rb = q

Parameters r_matrix : array-like, str, tuple
    • array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional
    This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
    An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
    An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional
    If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics
f_test for F tests
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```

This tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>> print(T_test)
<T contrast: effect=-1829.2025687192481, sd=455.39079463411717, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string.

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

```
statsmodels.discrete.discrete_model.BinaryResults.tvalues

static BinaryResults.tvalues()
Return the t-statistic for a given parameter estimate.
```

```
statsmodels.discrete.discrete_model.BinaryResults.wald_test

BinaryResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
   • array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
   • str : The full hypotheses to test can be given as a string. See the examples.
   • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
   This is deprecated. See r_matrix and the examples for more information on new usage.
   Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
```
An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
Default is 1.0 for no scaling.
invcov : array-like, optional
A q x q array to specify an inverse covariance matrix based on a restrictions matrix.
use_f : bool
If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes
The matrix r_matrix is assumed to be non-singular. More precisely, r_matrix (pX pX.T) r_matrix.T is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
CountModel.cov_params_func_l1(likelihood_model, xopt, retvals)
Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
Returns a full cov_params matrix, with entries corresponding to zero'd values set to np.nan.

CountModel.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)
Fit the model using maximum likelihood.
The rest of the docstring is from statsmodels.LikelihoodModel.fit
Fit method for likelihood based models

Parameters start_params : array-like, optional
Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

method : str, optional
The method determines which solver from scipy.optimize is used, and it can be chosen from among the following strings:
• 'newton' for Newton-Raphson, 'nm' for Nelder-Mead
• 'bfgs' for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
• 'lbfgs' for limited-memory BFGS with optional box constraints
• 'powell' for modified Powell’s method
• 'cg' for conjugate gradient
• 'ncg' for Newton-conjugate gradient
• 'basinhopping' for global basin-hopping solver
The explicit arguments in fit are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

maxiter : int, optional
The maximum number of iterations to perform.

full_output : bool, optional
Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

disp : bool, optional
Set to True to print convergence messages.

fargs : tuple, optional
Extra arguments passed to the likelihood function, i.e., loglike(x,*args)
callback : callable callback(xk), optional

Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool, optional

Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

Notes

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

'newton'
tol : float
    Relative error in params acceptable for convergence.

'nm' -- Nelder Mead
xtol : float
    Relative error in params acceptable for convergence
ftol : float
    Relative error in loglike(params) acceptable for convergence
maxfun : int
    Maximum number of function evaluations to make.

'bfgs'
gtol : float
    Stop when norm of gradient is less than gtol.
norm : float
    Order of norm (np.Inf is max, -np.Inf is min)
epsilon
    If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

'lbfgs'
m : int
    This many terms are used for the Hessian approximation.
factr : float
    A stop condition that is a variant of relative error.
pgtol : float
    A stop condition that uses the projected gradient.
epsilon
    If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
maxfun : int
    Maximum number of function evaluations to make.
bounds : sequence
    (min, max) pairs for each element in x,
    defining the bounds on that parameter.
    Use None for one of min or max when there is no bound in that direction.

'cg'
gtol : float
    Stop when norm of gradient is less than gtol.
norm : float
    Order of norm (np.Inf is max, -np.Inf is min)
epsilon : float
    If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if
Likelihoodmodel.score is None.

'ncg'
  fhess_p : callable f'(x,*args)
      Function which computes the Hessian of f times an arbitrary
      vector, p. Should only be supplied if
      LikelihoodModel.hessian is None.
  avextol : float
      Stop when the average relative error in the minimizer
      falls below this amount.
  epsilon : float or ndarray
      If fhess is approximated, use this value for the step size.
      Only relevant if Likelihoodmodel.hessian is None.

'powell'
  xtol : float
      Line-search error tolerance
  ftol : float
      Relative error in loglike(params) for acceptable for
      convergence.
  maxfun : int
      Maximum number of function evaluations to make.
  start_dir electromagnetic
      Initial direction set.

'basinhopping'
  niter : integer
      The number of basin hopping iterations.
  niter_success : integer
      Stop the run if the global minimum candidate remains the
      same for this number of iterations.
  T : float
      The "temperature" parameter for the accept or reject
      criterion. Higher "temperatures" mean that larger jumps
      in function value will be accepted. For best results
      'T' should be comparable to the separation (in function
      value) between local minima.
  stepsize : float
      Initial step size for use in the random displacement.
  interval : integer
      The interval for how often to update the 'stepsize'.
  minimizer : dict
      Extra keyword arguments to be passed to the minimizer
      'scipy.optimize.minimize()', for example 'method' - the
      minimization method (e.g. 'L-BFGS-B'), or 'tol' - the
      tolerance for termination. Other arguments are mapped from
      explicit argument of 'fit':
      - 'args' <- 'fargs'
      - 'jac' <- 'score'
      - 'hess' <- 'hess'

statsmodels.discrete.discrete_model.CountModel.fit_regularized

CountModel.fit_regularized(start_params=None, method='l1', max-
 iter='defined_by_method', full_output=1, disp=1, call-
 back=None, alpha=0, trim_mode='auto', auto_trim_tol=0.01,
 size_trim_tol=0.0001, qc_tol=0.03, **kwargs)

Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is

determined by the argument method.
Parameters start_params : array-like, optional

Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

method : ‘l1’ or ‘l1_cvxopt_cp’

See notes for details.

maxiter : Integer or ‘defined_by_method’

Maximum number of iterations to perform. If ‘defined_by_method’, then use method defaults (see notes).

full_output : bool

Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

disp : bool

Set to True to print convergence messages.

fargs : tuple

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

callback : callable callback(xk)

Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool

Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.

alpha : non-negative scalar or numpy array (same size as parameters)

The weight multiplying the l1 penalty term

trim_mode : ‘auto’, ‘size’, or ‘off’

If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim params if they have very small absolute value

size_trim_tol : float or ‘auto’ (default = ‘auto’)

For use when trim_mode == ‘size’

auto_trim_tol : float

For sue when trim_mode == ‘auto’. Use

qc_tol : float

Print warning and don’t allow auto trim when (ii) (above) is violated by this much.

qc_verbose : Boolean

If true, print out a full QC report upon failure
Optional arguments for the solvers (available in Results.mle_settings):

- `'l1'`
  
  acc : float (default 1e-6)
  Requested accuracy as used by slsqp

- `'l1_cvxopt_cp'`
  
  abstol : float
  absolute accuracy (default: 1e-7).
  reltol : float
  relative accuracy (default: 1e-6).
  feastol : float
  tolerance for feasibility conditions (default: 1e-7).
  refinement : int
  number of iterative refinement steps when solving KKT equations (default: 1).

Optimization methodology

With $L$ the negative log likelihood, we solve the convex but non-smooth problem

$$
\min_{\beta} L(\beta) + \sum_k \alpha_k |\beta_k|
$$

via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the “added variables” $u_k$)

$$
\min_{\beta, u} L(\beta) + \sum_k \alpha_k u_k,
$$

subject to

$$
-u_k \leq \beta_k \leq u_k.
$$

With $\partial_k L$ the derivative of $L$ in the $k^{th}$ parameter direction, theory dictates that, at the minimum, exactly one of two conditions holds:

1. $|\partial_k L| = \alpha_k$ and $\beta_k \neq 0$
2. $|\partial_k L| \leq \alpha_k$ and $\beta_k = 0$

**statsmodels.discrete.discrete_model.CountModel.from_formula**

**classmethod** CountModel.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

**Parameters**

- **formula** : str or generic Formula object
  
  The formula specifying the model

- **data** : array-like
  
  The data for the model. See Notes.

- **subset** : array-like
An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

```python
args : extra arguments
These are passed to the model
```

```python
kwargs : extra keyword arguments
These are passed to the model.
```

**Returns**

```python
model : Model instance
```

**Notes**

data must define `__getitem__` with the keys in the formula terms `args` and `kwargs` are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

```python
statsmodels.discrete.discrete_model.CountModel.hessian
```

```python
CountModel.hessian(params)
The Hessian matrix of the model
```

```python
statsmodels.discrete.discrete_model.CountModel.information
```

```python
CountModel.information(params)
Fisher information matrix of model
```

```python
Retruns -Hessian of loglike evaluated at params.
```

```python
statsmodels.discrete.discrete_model.CountModel.initialize
```

```python
CountModel.initialize()
Initialize is called by statsmodels.model.LikelihoodModel.__init__ and should contain any preprocessing that needs to be done for a model.
```

```python
statsmodels.discrete.discrete_model.CountModel.loglike
```

```python
CountModel.loglike(params)
Log-likelihood of model.
```

```python
statsmodels.discrete.discrete_model.CountModel.pdf
```

```python
CountModel.pdf(X)
The probability density (mass) function of the model.
```

```python
statsmodels.discrete.discrete_model.CountModel.predict
```

```python
CountModel.predict(params, exog=None, exposure=None, offset=None, linear=False)
Predict response variable of a count model given exogenous variables.
```
Notes

If exposure is specified, then it will be logged by the method. The user does not need to log it first.

\texttt{statsmodels.discrete.discrete_model.CountModel.score}

\texttt{CountModel.score(params)}

Score vector of model.

The gradient of logL with respect to each parameter.

Attributes

\begin{verbatim}
endog_names
exog_names
\end{verbatim}

\texttt{statsmodels.discrete.discrete_model.MultinomialModel}

class \texttt{statsmodels.discrete.discrete_model.MultinomialModel(endog, exog, **kwargs)}

Methods

\begin{verbatim}
cdf(X)
cov_params_func_l1(likelihood_model, xopt, ...)
fitted_start_params, method, maxiter, ...)
fitted_regularized([start_params, method, ...])
from_formula(formula, data[, subset])
hessian(params)
information(params)
initialize()
loglike(params)
pdf(X)
predict(params[, exog, linear])
score(params)
\end{verbatim}

The cumulative distribution function of the model.
Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
Fit the model using maximum likelihood.
Fit the model using a regularized maximum likelihood.
Create a Model from a formula and dataframe.
The Hessian matrix of the model
Fisher information matrix of model
Preprocesses the data for MNLogit.
Log-likelihood of model.
The probability density (mass) function of the model.
Predict response variable of a model given exogenous variables.
Score vector of model.

\texttt{statsmodels.discrete.discrete_model.MultinomialModel.cdf}

\texttt{MultinomialModel.cdf(X)}

The cumulative distribution function of the model.

\texttt{statsmodels.discrete.discrete_model.MultinomialModel.cov_params_func_l1}

\texttt{MultinomialModel.cov_params_func_l1(likelihood_model, xopt, retvals)}

Computes cov_params on a reduced parameter space corresponding to the nonzero parameters resulting from the l1 regularized fit.
Returns a full cov_params matrix, with entries corresponding to zero’d values set to np.nan.
MultinomialModel.fit

MultinomialModel.fit(start_params=None, method='newton', maxiter=35, full_output=1, disp=1, callback=None, **kwargs)

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

Fit method for likelihood based models

**Parameters** start_params : array-like, optional

Initial guess of the solution for the loglikelihood maximization. The default is an array of zeros.

method : str, optional

The `method` determines which solver from `scipy.optimize` is used, and it can be chosen from among the following strings:

- `newton` for Newton-Raphson, `nm` for Nelder-Mead
- `bfgs` for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
- `lbfgs` for limited-memory BFGS with optional box constraints
- `powell` for modified Powell’s method
- `cg` for conjugate gradient
- `ncg` for Newton-conjugate gradient
- `basinhopping` for global basin-hopping solver

The explicit arguments in `fit` are passed to the solver, with the exception of the basin-hopping solver. Each solver has several optional arguments that are not the same across solvers. See the notes section below (or scipy.optimize) for the available arguments and for the list of explicit arguments that the basin-hopping solver supports.

maxiter : int, optional

The maximum number of iterations to perform.

full_output : bool, optional

Set to True to have all available output in the Results object’s mle_retvals attribute. The output is dependent on the solver. See LikelihoodModelResults notes section for more information.

disp : bool, optional

Set to True to print convergence messages.

fargs : tuple, optional

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)

callback : callable callback(xk), optional

Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool, optional

Set to True to return list of solutions at each iteration. Available in Results object’s mle_retvals attribute.
Notes

The ‘basinhopping’ solver ignores maxiter, retall, full_output explicit arguments.

Optional arguments for solvers (see returned Results.mle_settings):

‘newton’
  tol : float
    Relative error in params acceptable for convergence.

‘nm’ -- Nelder Mead
  xtol : float
    Relative error in params acceptable for convergence
  ftol : float
    Relative error in loglike(params) acceptable for convergence
  maxfun : int
    Maximum number of function evaluations to make.

‘bfgs’
  gtol : float
    Stop when norm of gradient is less than gtol.
  norm : float
    Order of norm (np.Inf is max, -np.Inf is min)
  epsilon
    If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.

‘lbfgs’
  m : int
    This many terms are used for the Hessian approximation.
  factr : float
    A stop condition that is a variant of relative error.
  pgtol : float
    A stop condition that uses the projected gradient.
  epsilon
    If fprime is approximated, use this value for the step size. Only relevant if LikelihoodModel.score is None.
  maxfun : int
    Maximum number of function evaluations to make.
  bounds : sequence
    (min, max) pairs for each element in x, defining the bounds on that parameter.
    Use None for one of min or max when there is no bound in that direction.

‘cg’
  gtol : float
    Stop when norm of gradient is less than gtol.
  norm : float
    Order of norm (np.Inf is max, -np.Inf is min)
  epsilon : float
    If fprime is approximated, use this value for the step size. Can be scalar or vector. Only relevant if Likelihoodmodel.score is None.

‘ncg’
  fhess_p : callable f’(x,*args)
    Function which computes the Hessian of f times an arbitrary vector, p. Should only be supplied if LikelihoodModel.hessian is None.
  avextol : float
    Stop when the average relative error in the minimizer falls below this amount.
epsilon : float or ndarray
    If fhess is approximated, use this value for the step size.
    Only relevant if Likelihoodmodel.hessian is None.

'powell'
xtol : float
    Line-search error tolerance
ftol : float
    Relative error in loglike(params) for acceptable for
    convergence.
maxfun : int
    Maximum number of function evaluations to make.
start_diric : ndarray
    Initial direction set.

'basinhopping'
niter : integer
    The number of basin hopping iterations.
niter_success : integer
    Stop the run if the global minimum candidate remains the
    same for this number of iterations.
T : float
    The "temperature" parameter for the accept or reject
    criterion. Higher "temperatures" mean that larger jumps
    in function value will be accepted. For best results
    'T' should be comparable to the separation (in function
    value) between local minima.
stepsize : float
    Initial step size for use in the random displacement.
interval : integer
    The interval for how often to update the 'stepsize'.
minimizer : dict
    Extra keyword arguments to be passed to the minimizer
    'scipy.optimize.minimize()', for example 'method' - the
    minimization method (e.g. 'L-BFGS-B'), or 'tol' - the
    tolerance for termination. Other arguments are mapped from
    explicit argument of 'fit':
    - 'args' <- 'fargs'
    - 'jac' <- 'score'
    - 'hess' <- 'hess'

statsmodels.discrete.discrete_model.MultinomialModel.fit_regularized

MultinomialModel.fit_regularized(start_params=None, method='l1', max-
iter='defined_by_method', full_output=1, disp=1, callback=None, alpha=0, trim_mode='auto',
auto_trim_tol=0.01, size_trim_tol=0.0001, qc_tol=0.03, **kwargs)

Fit the model using a regularized maximum likelihood. The regularization method AND the solver used is
determined by the argument method.

Parameters start_params : array-like, optional
    Initial guess of the solution for the loglikelihood maximization. The default is an array
    of zeros.

method : 'l1' or 'l1_cvxopt_cp'
    See notes for details.
maxiter : Integer or ‘defined_by_method’

Maximum number of iterations to perform. If ‘defined_by_method’, then use method
defaults (see notes).

full_output : bool

Set to True to have all available output in the Results object’s mle_retvals attribute. The
output is dependent on the solver. See LikelihoodModelResults notes section for more
information.

disp : bool

Set to True to print convergence messages.

fargs : tuple

Extra arguments passed to the likelihood function, i.e., loglike(x,*args)
callback : callable callback(xk)

Called after each iteration, as callback(xk), where xk is the current parameter vector.

retall : bool

Set to True to return list of solutions at each iteration. Available in Results object’s
mle_retvals attribute.

alpha : non-negative scalar or numpy array (same size as parameters)

The weight multiplying the l1 penalty term

trim_mode : ‘auto’, ‘size’, or ‘off’

If not ‘off’, trim (set to zero) parameters that would have been zero if the solver reached
the theoretical minimum. If ‘auto’, trim params using the Theory above. If ‘size’, trim
params if they have very small absolute value

size_trim_tol : float or ‘auto’ (default = ‘auto’)

For use when trim_mode == ‘size’

auto_trim_tol : float

For sue when trim_mode == ‘auto’. Use

qc_tol : float

Print warning and don’t allow auto trim when (ii) (above) is violated by this much.

qc_verbose : Boolean

If true, print out a full QC report upon failure

Notes

Optional arguments for the solvers (available in Results.mle_settings):

‘l1’
    acc : float (default 1e-6)
    Requested accuracy as used by slsqp
‘l1_cvxopt_cp’
    abstol : float
    absolute accuracy (default: 1e-7).
    reltol : float
relative accuracy (default: 1e-6).
feastol : float
tolerance for feasibility conditions (default: 1e-7).
refinement : int
number of iterative refinement steps when solving KKT
equations (default: 1).

Optimization methodology
With $L$ the negative log likelihood, we solve the convex but non-smooth problem
\[
\min_{\beta} L(\beta) + \sum_k \alpha_k |\beta_k|
\]
via the transformation to the smooth, convex, constrained problem in twice as many variables (adding the “added variables” $u_k$)
\[
\min_{\beta, u} L(\beta) + \sum_k \alpha_k u_k,
\]
subject to
\[-u_k \leq \beta_k \leq u_k.\]

With $\partial_k L$ the derivative of $L$ in the $k^{th}$ parameter direction, theory dictates that, at the minimum, exactly one of two conditions holds:
1. $|\partial_k L| = \alpha_k$ and $\beta_k \neq 0$
2. $|\partial_k L| \leq \alpha_k$ and $\beta_k = 0$

`statsmodels.discrete.discrete_model.MultinomialModel.from_formula`

classmethod `MultinomialModel.from_formula` *(formula, data, subset=None, *args, **kwargs)*
Create a Model from a formula and dataframe.

Parameters

- **formula**: str or generic Formula object
  The formula specifying the model
- **data**: array-like
  The data for the model. See Notes.
- **subset**: array-like
  An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
- **args**: extra arguments
  These are passed to the model
- **kwargs**: extra keyword arguments
  These are passed to the model.

Returns

- **model**: Model instance

3.5. Regression with Discrete Dependent Variable 513
Notes

data must define \_\_getitem\_\_ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**statsmodels.discrete.discrete_model.MultinomialModel.hessian**

MultinomialModel.hessian(params)
The Hessian matrix of the model

**statsmodels.discrete.discrete_model.MultinomialModel.information**

MultinomialModel.information(params)
Fisher information matrix of model
Returns -Hessian of loglike evaluated at params.

**statsmodels.discrete.discrete_model.MultinomialModel.initialize**

MultinomialModel.initialize()
Preprocesses the data for MNLogit.

Turns the endogenous variable into an array of dummies and assigns J and K.

**statsmodels.discrete.discrete_model.MultinomialModel.loglike**

MultinomialModel.loglike(params)
Log-likelihood of model.

**statsmodels.discrete.discrete_model.MultinomialModel.pdf**

MultinomialModel.pdf(X)
The probability density (mass) function of the model.

**statsmodels.discrete.discrete_model.MultinomialModel.predict**

MultinomialModel.predict(params, exog=None, linear=False)
Predict response variable of a model given exogenous variables.

Parameters

- **params**: array-like
  2d array of fitted parameters of the model. Should be in the order returned from the model.
- **exog**: array-like
  1d or 2d array of exogenous values. If not supplied, the whole exog attribute of the model is used. If a 1d array is given it assumed to be 1 row of exogenous variables. If you only have one regressor and would like to do prediction, you must provide a 2d array with shape[1] == 1.
- **linear**: bool, optional
If True, returns the linear predictor dot(exog.params). Else, returns the value of the cdf at the linear predictor.

Notes

Column 0 is the base case, the rest conform to the rows of params shifted up one for the base case.

```python
statsmodels.discrete.discrete_model.MultinomialModel.score
```

**MultinomialModel.score** *(params)*

Score vector of model.

The gradient of logL with respect to each parameter.

Attributes

- `endog_names`
- `exog_names`

### 3.6 ANOVA

Analysis of Variance models

#### 3.6.1 Examples

```python
In [1]: import statsmodels.api as sm
In [2]: from statsmodels.formula.api import ols
In [3]: moore = sm.datasets.get_rdataset("Moore", "car",
...: cache=True) # load data
...:
In [4]: data = moore.data
In [5]: data = data.rename(columns={"partner.status" :
...: "partner_status"}) # make name pythonic
...:
In [6]: moore_lm = ols(‘conformity ~ C(fcategory, Sum)*C(partner_status, Sum)’,
...: data=data).fit()
...:
In [7]: table = sm.stats.anova_lm(moore_lm, typ=2) # Type 2 ANOVA DataFrame
In [8]: print table
```

```
sum_sq df    F     PR(>F)  
C(fcategory, Sum)   11.614700 2 0.276958 0.759564
C(partner_status, Sum)   212.213778 1 10.120692 0.002874
C(fcategory, Sum):C(partner_status, Sum)   175.488928 2 4.184623 0.022572
Residual            817.763961 39 NaN     NaN
```
A more detailed example can be found here:

# 3.6.2 Module Reference

```
anova_lm(*args, **kwargs)  ANOVA table for one or more fitted linear models.
```

```
statsmodels.stats.anova.anova_lm

statsmodels.stats.anova.anova_lm(*args, **kwargs)
  ANOVA table for one or more fitted linear models.

  Parameters
  args: fitted linear model results instance
    One or more fitted linear models
  scale: float
    Estimate of variance, If None, will be estimated from the largest model. Default is None.
  test: str {"F", "Chisq", "Cp"} or None
    Test statistics to provide. Default is “F”.
  typ: str or int {"I","II","III"} or {1,2,3}
    The type of ANOVA test to perform. See notes.
  robust: {None, "hc0", "hc1", "hc2", "hc3"}
    Use heteroscedasticity-corrected coefficient covariance matrix. If robust covariance is desired, it is recommended to use hc3.

  Returns
  anova: DataFrame
    A DataFrame containing:

See also:

model_results.compare_f_test, model_results.compare_lm_test

Notes

Model statistics are given in the order of args. Models must have been fit using the formula api.

Examples

```python
>>> import statsmodels.api as sm
>>> from statsmodels.formula.api import ols

>>> moore = sm.datasets.get_rdataset("Moore", "car", ... cache=True)  # load data
>>> data = moore.data
>>> data = data.rename(columns={"partner.status" : ...  "partner_status")}  # make name pythonic
>>> moore_lm = ols(‘conformity ~ C(fcategory, Sum)*C(partner_status, Sum)’, ...  data=data).fit()
```
3.7 Time Series analysis tsa

statsmodels.tsa contains model classes and functions that are useful for time series analysis. This currently includes univariate autoregressive models (AR), vector autoregressive models (VAR) and univariate autoregressive moving average models (ARMA). It also includes descriptive statistics for time series, for example autocorrelation, partial autocorrelation function and periodogram, as well as the corresponding theoretical properties of ARMA or related processes. It also includes methods to work with autoregressive and moving average lag-polynomials. Additionally, related statistical tests and some useful helper functions are available.

Estimation is either done by exact or conditional Maximum Likelihood or conditional least-squares, either using Kalman Filter or direct filters.

Currently, functions and classes have to be imported from the corresponding module, but the main classes will be made available in the statsmodels.tsa namespace. The module structure is within statsmodels.tsa is

- stattools : empirical properties and tests, acf, pacf, granger-causality, adf unit root test, ljung-box test and others.
- ar_model : univariate autoregressive process, estimation with conditional and exact maximum likelihood and conditional least-squares
- arima_model : univariate ARMA process, estimation with conditional and exact maximum likelihood and conditional least-squares
- vector_ar, var : vector autoregressive process (VAR) estimation models, impulse response analysis, forecast error variance decompositions, and data visualization tools
- kalmanf : estimation classes for ARMA and other models with exact MLE using Kalman Filter
- arma_process : properties of arma processes with given parameters, this includes tools to convert between ARMA, MA and AR representation as well as acf, pacf, spectral density, impulse response function and similar
- sandbox.tsa.fftarma : similar to arma_process but working in frequency domain
- tsatools : additional helper functions, to create arrays of lagged variables, construct regressors for trend, detrend and similar.
- filters : helper function for filtering time series

Some additional functions that are also useful for time series analysis are in other parts of statsmodels, for example additional statistical tests.

Some related functions are also available in matplotlib, nitime, and scikits.talkbox. Those functions are designed more for the use in signal processing where longer time series are available and work more often in the frequency domain.

3.7.1 Descriptive Statistics and Tests

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stattools.acovf(x[, unbiased, demean, fft])</code></td>
<td>Autocovariance for 1D</td>
</tr>
<tr>
<td><code>stattools.acf(x[, unbiased, nlags, confint, ...])</code></td>
<td>Autocorrelation function for 1d arrays.</td>
</tr>
<tr>
<td><code>stattools.pacf(x[, nlags, method, alpha])</code></td>
<td>Partial autocorrelation estimated</td>
</tr>
<tr>
<td><code>stattools.pacf_yw(x[, nlags, method])</code></td>
<td>Partial autocorrelation estimated with non-recursive yule_walker</td>
</tr>
<tr>
<td><code>stattools.pacf_ols(x[, nlags])</code></td>
<td>Calculate partial autocorrelations</td>
</tr>
<tr>
<td><code>stattools.cocovf(x, y[, unbiased, demean])</code></td>
<td>Crosscovariance for 1D</td>
</tr>
<tr>
<td><code>stattools.ccf(x, y[, unbiased])</code></td>
<td>Cross-correlation function for 1d</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.139 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stattools.periodogram(X)</code></td>
<td>Returns the periodogram for the natural frequency of X</td>
</tr>
<tr>
<td><code>stattools.adfuller(x[, maxlag, regression, ...])</code></td>
<td>Augmented Dickey-Fuller unit root test</td>
</tr>
<tr>
<td><code>stattools.q_stat(x, nobs[, type])</code></td>
<td>Return’s Ljung-Box Q Statistic</td>
</tr>
<tr>
<td><code>stattools.grangercausalitytests(x, maxlag[,...])</code></td>
<td>four tests for granger non causality of 2 timeseries</td>
</tr>
<tr>
<td><code>stattools.levinson_durbin(s[, nlags, isacov])</code></td>
<td>Levinson-Durbin recursion for autoregressive processes</td>
</tr>
<tr>
<td><code>stattools.arma_order_select_ic(y[, max_ar,...])</code></td>
<td>Returns information criteria for many ARMA models</td>
</tr>
</tbody>
</table>

**statsmodels.tsa.stattools.acovf**

`s = acovf(x, unbiased=False, demean=True, fft=False)`

Autocovariance for 1D

**Parameters**

- **x**: array
  - Time series data. Must be 1d.
- **unbiased**: bool
  - If True, then denominators is n-k, otherwise n
- **demean**: bool
  - If True, then subtract the mean x from each element of x
- **fft**: bool
  - If True, use FFT convolution. This method should be preferred for long time series.

**Returns**

- **acovf**: array
  - autocovariance function

**statsmodels.tsa.stattools.acf**

`s = acf(x, unbiased=False, nlags=40, confint=None, qstat=False, fft=False, alpha=None)`

Autocorrelation function for 1d arrays.

**Parameters**

- **x**: array
  - Time series data
- **unbiased**: bool
  - If True, then denominators for autocovariance are n-k, otherwise n
- **nlags**: int, optional
  - Number of lags to return autocorrelation for.
- **confint**: scalar, optional
  - The use of confint is deprecated. See alpha. If a number is given, the confidence intervals for the given level are returned. For instance if confint=95, 95 % confidence intervals are returned where the standard deviation is computed according to Bartlett’s formula.
- **qstat**: bool, optional
  - If True, returns the Ljung-Box q statistic for each autocorrelation coefficient. See q_stat for more information.
fft : bool, optional
    If True, computes the ACF via FFT.

alpha : scalar, optional
    If a number is given, the confidence intervals for the given level are returned. For instance if alpha=.05, 95 \% confidence intervals are returned where the standard deviation is computed according to Bartlett’s formula.

Returns aec : array
    autocorrelation function

confint : array, optional
    Confidence intervals for the ACF. Returned if confint is not None.

qstat : array, optional
    The Ljung-Box Q-Statistic. Returned if q_stat is True.

pvalues : array, optional
    The p-values associated with the Q-statistics. Returned if q_stat is True.

Notes

The aec at lag 0 (ie., 1) is returned.

This is based np.correlate which does full convolution. For very long time series it is recommended to use fft convolution instead.

If unbiased is true, the denominator for the autocovariance is adjusted but the autocorrelation is not an unbiased estimator.

**statsmodels.tsa.stattools.pacf**

**statsmodels.tsa.stattools.pacf**(x, nlags=40, method='ywunbiased', alpha=None)

Partial autocorrelation estimated

Parameters x : 1d array
    observations of time series for which pacf is calculated

nlags : int
    largest lag for which pacf is returned

method : ‘ywunbiased’ (default) or ‘ywml’ or ‘ols’
    specifies which method for the calculations to use:
    • yw or ywunbiased : yule walker with bias correction in denominator for acovf
    • ywm or ywmle : yule walker without bias correction
    • ols - regression of time series on lags of it and on constant
    • ld or ldunbiased : Levinson-Durbin recursion with bias correction
    • ldb or ldbiased : Levinson-Durbin recursion without bias correction

alpha : scalar, optional
If a number is given, the confidence intervals for the given level are returned. For instance if alpha=.05, 95\% confidence intervals are returned where the standard deviation is computed according to $1/\sqrt{\text{len}(x)}$.

**Returns**
- **pacf**: 1d array
  - partial autocorrelations, nlags elements, including lag zero
- **confint**: array, optional
  - Confidence intervals for the PACF. Returned if confint is not None.

**Notes**

This solves yule_walker equations or ols for each desired lag and contains currently duplicate calculations.

**statsmodels.tsa.stattools.pacf_yw**

**statsmodels.tsa.stattools.pacf_yw**($x$, **nlags**=40, **method**='unbiased')

Partial autocorrelation estimated with non-recursive yule_walker

**Parameters**
- **x**: 1d array
  - observations of time series for which pacf is calculated
- **nlags**: int
  - largest lag for which pacf is returned
- **method**: ‘unbiased’ (default) or ‘mle’
  - method for the autocovariance calculations in yule walker

**Returns**
- **pacf**: 1d array
  - partial autocorrelations, maxlag+1 elements

**Notes**

This solves yule_walker for each desired lag and contains currently duplicate calculations.

**statsmodels.tsa.stattools.pacf_ols**

**statsmodels.tsa.stattools.pacf_ols**($x$, **nlags**=40)

Calculate partial autocorrelations

**Parameters**
- **x**: 1d array
  - observations of time series for which pacf is calculated
- **nlags**: int
  - Number of lags for which pacf is returned. Lag 0 is not returned.

**Returns**
- **pacf**: 1d array
  - partial autocorrelations, maxlag+1 elements
Notes

This solves a separate OLS estimation for each desired lag.

\textbf{statsmodels.tsa.stattools.ccovf}

\texttt{statsmodels.tsa.stattools.ccovf}(x, y, unbiased=True, demean=True)

crosscovariance for 1D

\textbf{Parameters} x, y : arrays
time series data

\textbf{unbiased} : boolean
if True, then denominators is n-k, otherwise n

\textbf{Returns} ccovf : array
autocovariance function

Notes

This uses \texttt{np.correlate} which does full convolution. For very long time series it is recommended to use \texttt{fft} convolution instead.

\textbf{statsmodels.tsa.stattools.ccf}

\texttt{statsmodels.tsa.stattools.ccf}(x, y, unbiased=True)

cross-correlation function for 1d

\textbf{Parameters} x, y : arrays
time series data

\textbf{unbiased} : boolean
if True, then denominators for autocovariance is n-k, otherwise n

\textbf{Returns} ccf : array
cross-correlation function of x and y

Notes

This is based \texttt{np.correlate} which does full convolution. For very long time series it is recommended to use \texttt{fft} convolution instead.

If unbiased is true, the denominator for the autocovariance is adjusted but the autocorrelation is not an unbiased estimator.

\textbf{statsmodels.tsa.stattools.periodogram}

\texttt{statsmodels.tsa.stattools.periodogram}(X)

Returns the periodogram for the natural frequency of X

\textbf{Parameters} X : array-like
Array for which the periodogram is desired.

Returns  

pgram : array

\[
\text{1./len(X) * np.abs(np.fft.fft(X))**2}
\]

References

Brockwell and Davis.

statsmodels.tsa.stattools.adfuller

statsmodels.tsa.stattools.adfuller(x, maxlag=None, regression='c', autolag='AIC', store=False, regresults=False)

Augmented Dickey-Fuller unit root test

The Augmented Dickey-Fuller test can be used to test for a unit root in a univariate process in the presence of serial correlation.

Parameters  
x : array_like, 1d

data series

maxlag : int

Maximum lag which is included in test, default \(12 \times (\text{nobs}/100)^{1/4}\)

regression : str

\{'c', 'ct', 'ctt', 'nc'\}

Constant and trend order to include in regression

- 'c' : constant only (default)
- 'ct' : constant and trend
- 'ctt' : constant, and linear and quadratic trend
- 'nc' : no constant, no trend

autolag : \{‘AIC’, ‘BIC’, ‘t-stat’, None\}

- if None, then maxlag lags are used
- if ‘AIC’ (default) or ‘BIC’, then the number of lags is chosen to minimize the corresponding information criterium
- ‘t-stat’ based choice of maxlag. Starts with maxlag and drops a lag until the t-statistic on the last lag length is significant at the 95 % level.

store : bool

If True, then a result instance is returned additionally to the adf statistic (default is False)

regresults : bool

If True, the full regression results are returned (default is False)

Returns  

adf : float

Test statistic

pvalue : float

MacKinnon’s approximate p-value based on MacKinnon (1994)

usedlag : int

Number of lags used.

nobs : int
Number of observations used for the ADF regression and calculation of the critical values.

**critical values** : dict

Critical values for the test statistic at the 1 %, 5 %, and 10 % levels. Based on MacKinnon (2010)

**icbest** : float

The maximized information criterion if autolag is not None.

**regresults** : RegressionResults instance

The

**resstore** : (optional) instance of ResultStore

an instance of a dummy class with results attached as attributes

**Notes**

The null hypothesis of the Augmented Dickey-Fuller is that there is a unit root, with the alternative that there is no unit root. If the pvalue is above a critical size, then we cannot reject that there is a unit root.

The p-values are obtained through regression surface approximation from MacKinnon 1994, but using the updated 2010 tables. If the p-value is close to significant, then the critical values should be used to judge whether to accept or reject the null.

The autolag option and maxlag for it are described in Greene.

**References**

Greene Hamilton


**Examples**

see example script

```
statsmodels.tsa.stattools.q_stat

statsmodels.tsa.stattools.q_stat(x, nobs, type='ljungbox')
Return's Ljung-Box Q Statistic

x [array-like] Array of autocorrelation coefficients. Can be obtained from acf.

nobs [int] Number of observations in the entire sample (ie., not just the length of the autocorrelation function results.

Returns q-stat : array

Ljung-Box Q-statistic for autocorrelation parameters
```
**p-value**: array

P-value of the Q statistic

**Notes**

Written to be used with acf.

```python
statsmodels.tsa.stattools.grangercausalitytests
```

```python
statsmodels.tsa.stattools.grangercausalitytests(x, maxlag, addconst=True, verbose=True)
```

four tests for granger non causality of 2 timeseries

all four tests give similar results `params_ftest` and `ssr_ftest` are equivalent based on F test which is identical to lmtest:grangertest in R

**Parameters**

- `x`: array, 2d, (nobs, 2)
  
data for test whether the time series in the second column Granger causes the time series in the first column

- `maxlag`: integer
  
  the Granger causality test results are calculated for all lags up to maxlag

- `verbose`: bool
  
  print results if true

**Returns**

- `results`: dictionary
  
  all test results, dictionary keys are the number of lags. For each lag the values are a tuple, with the first element a dictionary with teststatistic, pvalues, degrees of freedom, the second element are the OLS estimation results for the restricted model, the unrestricted model and the restriction (contrast) matrix for the parameter f_test.

**Notes**

TODO: convert to class and attach results properly

The Null hypothesis for grangercausalitytests is that the time series in the second column, x2, does NOT Granger cause the time series in the first column, x1. Grange causality means that past values of x2 have a statistically significant effect on the current value of x1, taking past values of x1 into account as regressors. We reject the null hypothesis that x2 does not Granger cause x1 if the pvalues are below a desired size of the test.

The null hypothesis for all four test is that the coefficients corresponding to past values of the second time series are zero.

'params_ftest', 'ssr_ftest' are based on F distribution

'ssr_ch2test', 'lrtest' are based on chi-square distribution

**References**

http://en.wikipedia.org/wiki/Granger_causality
Greene: Econometric Analysis
Levinson-Durbin recursion for autoregressive processes

**Parameters**

- **s**: array_like
  - If isacov is False, then this is the time series. If isacov is true then this is interpreted as autocovariance starting with lag 0
- **nlags**: integer
  - largest lag to include in recursion or order of the autoregressive process
- **isacov**: boolean
  - flag to indicate whether the first argument, s, contains the autocovariances or the data series.

**Returns**

- **sigma_v**: float
  - estimate of the error variance
- **arcoefs**: ndarray
  - estimate of the autoregressive coefficients
- **pacf**: ndarray
  - partial autocorrelation function
- **sigma**: ndarray
  - entire sigma array from intermediate result, last value is sigma_v
- **phi**: ndarray
  - entire phi array from intermediate result, last column contains autoregressive coefficients for AR(nlags) with a leading 1

**Notes**

This function returns currently all results, but maybe we drop sigma and phi from the returns.

If this function is called with the time series (isacov=False), then the sample autocovariance function is calculated with the default options (biased, no fft).

**statsmodels.tsa.stattools arma_order_select_ic**

Returns information criteria for many ARMA models

**Parameters**

- **y**: array-like
  - Time-series data
- **max_ar**: int
  - Maximum number of AR lags to use. Default 4.
- **max_ma**: int
  - Maximum number of MA lags to use. Default 2.
ic : str, list

Information criteria to report. Either a single string or a list of different criteria is possible.

trend : str

The trend to use when fitting the ARMA models.

model_kw : dict

Keyword arguments to be passed to the ARMA model

fit_kw : dict

Keyword arguments to be passed to ARMA.fit.

Returns obj : Results object

Each ic is an attribute with a DataFrame for the results. The AR order used is the row index. The ma order used is the column index. The minimum orders are available as ic_min_order.

Notes

This method can be used to tentatively identify the order of an ARMA process, provided that the time series is stationary and invertible. This function computes the full exact MLE estimate of each model and can be, therefore a little slow. An implementation using approximate estimates will be provided in the future. In the meantime, consider passing {method : 'css'} to fit_kw.

Examples

>>> from statsmodels.tsa.arima_process import arma_generate_sample
>>> import statsmodels.api as sm
>>> import numpy as np

>>> arparams = np.array([.75, -.25])
>>> maparams = np.array([.65, .35])
>>> arparams = np.r_[1, -arparams]
>>> maparam = np.r_[1, maparams]
>>> nos = 250
>>> np.random.seed(2014)
>>> y = arma_generate_sample(arparams, maparams, nos)
>>> res = sm.tsa.arma_order_select_ic(y, ic=['aic', 'bic'], trend='nc')
>>> res.aic_min_order
>>> res.bic_min_order

3.7.2 Estimation

The following are the main estimation classes, which can be accessed through statsmodels.tsa.api and their result classes

Univariate Autogressive Processes (AR)
Table 3.140 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ar_model.AR</code></td>
<td>Autoregressive AR(p) model</td>
</tr>
<tr>
<td><code>ar_model.ARResults</code></td>
<td>Class to hold results from fitting an AR model.</td>
</tr>
</tbody>
</table>

**statsmodels.tsa.ar_model.AR**

**class** `statsmodels.tsa.ar_model.AR(endog[, dates, freq, missing])`

Autoregressive AR(p) model

**Parameters**

- **endog**: array-like
  
  1-d endogenous response variable. The independent variable.

- **dates**: array-like of datetime, optional
  
  An array-like object of datetime objects. If a pandas object is given for endog or exog, it is assumed to have a DateIndex.

- **freq**: str, optional
  
  The frequency of the time-series. A Pandas offset or ‘B’, ‘D’, ‘W’, ‘M’, ‘A’, or ‘Q’. This is optional if dates are given.

- **missing**: str
  
  Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘none.’

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit([maxlag, method, ic, trend, ...])</code></td>
<td>Fit the unconditional maximum likelihood of an AR(p) process.</td>
</tr>
<tr>
<td><code>from_formula(formula, data[, subset])</code></td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td><code>hessian(params)</code></td>
<td>Returns numerical hessian for now.</td>
</tr>
<tr>
<td><code>information(params)</code></td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td><code>initialize()</code></td>
<td></td>
</tr>
<tr>
<td><code>loglike(params)</code></td>
<td>The loglikelihood of an AR(p) process</td>
</tr>
<tr>
<td><code>predict(params[, start, end, dynamic])</code></td>
<td>Returns in-sample and out-of-sample prediction.</td>
</tr>
<tr>
<td><code>score(params)</code></td>
<td>Return the gradient of the loglikelihood at params.</td>
</tr>
<tr>
<td><code>select_order(maxlag, ic[, trend, method])</code></td>
<td>Select the lag order according to the information criterion.</td>
</tr>
</tbody>
</table>

**statsmodels.tsa.ar_model.AR.fit**

**AR.fit** are:

- `maxlag=None, method=’cmle’, ic=None, trend=’c’, transparams=True, start_params=None, solver=’lbfgs’, maxiter=35, full_output=1, disp=1, callback=None, **kwprops)`

Fit the unconditional maximum likelihood of an AR(p) process.

**Parameters**

- **maxlag**: int
  
  If `ic` is None, then maxlag is the lag length used in fit. If `ic` is specified then maxlag is the highest lag order used to select the correct lag order. If maxlag is None, the default is round(12*(nobs/100.)**(1/4.))

- **method**: str (‘cmle’, ‘mle’), optional
  
  cmle - Conditional maximum likelihood using OLS mle - Unconditional (exact) maximum likelihood. See `solver` and the Notes.

---

3.7. Time Series analysis tsa 527

Criterion used for selecting the optimal lag length. aic - Akaike Information Criterion
bic - Bayes Information Criterion t-stat - Based on last lag hqic - Hannan-Quinn In-
formation Criterion If any of the information criteria are selected, the lag length which
results in the lowest value is selected. If t-stat, the model starts with maxlag and drops
a lag until the highest lag has a t-stat that is significant at the 95 % level.

trend : str {‘c’, ‘nc’}

Whether to include a constant or not. ‘c’ - include constant. ‘nc’ - no constant.

The below can be specified if method is ‘mle’:

transparams : bool, optional

Whether or not to transform the parameters to ensure stationarity. Uses the transforma-
tion suggested in Jones (1980).

start_params : array-like, optional

A first guess on the parameters. Default is cmle estimates.

solver : str or None, optional

Solver to be used if method is ‘mle’. The default is ‘lbfgs’ (limited memory Broyden-
Fletcher-Goldfarb-Shanno). Other choices are ‘bfgs’, ‘newton’ (Newton-Raphson),
‘nm’ (Nelder-Mead), ‘cg’ - (conjugate gradient), ‘ncg’ (non-conjugate gradient), and
‘powell’.

maxiter : int, optional

The maximum number of function evaluations. Default is 35.

tol : float

The convergence tolerance. Default is 1e-08.

full_output : bool, optional

If True, all output from solver will be available in the Results object’s mle_retvals at-
tribute. Output is dependent on the solver. See Notes for more information.

disp : bool, optional

If True, convergence information is output.

callback : function, optional

Called after each iteration as callback(xk) where xk is the current parameter vector.

kwags : 

See Notes for keyword arguments that can be passed to fit.

See also:

statsmodels.base.model.LikelihoodModel.fit

References

Jones, R.H. 1980 “Maximum likelihood fitting of ARMA models to time series with missing observ-
ations.” Technometrics. 22.3. 389-95.
Create a Model from a formula and dataframe.

Parameters

- **formula**: str or generic Formula object
  - The formula specifying the model
- **data**: array-like
  - The data for the model. See Notes.
- **subset**: array-like
  - An array-like object of boolean, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
- **args**: extra arguments
  - These are passed to the model
- **kwargs**: extra keyword arguments
  - These are passed to the model.

Returns

- **model**: Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

Returns numerical hessian for now.

Not Implemented Yet

The loglikelihood of an AR(p) process

Parameters

- **params**: array
  - The fitted parameters of the AR model

Returns

- **llf**: float
  - The loglikelihood evaluated at params
Notes

Contains constant term. If the model is fit by OLS then this returns the conditional maximum likelihood.

\[
\frac{(n - p)}{2} \left( \log(2\pi) + \log(\sigma^2) \right) - \frac{1}{\sigma^2} \sum_i^2 e_i^2
\]

If it is fit by MLE then the (exact) unconditional maximum likelihood is returned.

\[
-\frac{n}{2} \log(2\pi) - \frac{n}{2} \log(\sigma^2) + \frac{1}{2} |V_p^{-1}| - \frac{1}{2\sigma^2} (y_p - \mu_p)' V_p^{-1} (y_p - \mu_p) - \frac{1}{2\sigma^2} \sum_{t=p+1}^n e_i^2
\]

where

\( \mu_p \) is a \((p \times 1)\) vector with each element equal to the mean of the AR process and \( \sigma^2 V_p \) is the \((p \times p)\) variance-covariance matrix of the first \( p \) observations.

statsmodels.tsa.ar_model.AR.predict

AR.predict(params, start=None, end=None, dynamic=False)

Returns in-sample and out-of-sample prediction.

Parameters params : array

The fitted model parameters.

start : int, str, or datetime

Zero-indexed observation number at which to start forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type.

end : int, str, or datetime

Zero-indexed observation number at which to end forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type.

dynamic : bool

The dynamic keyword affects in-sample prediction. If dynamic is False, then the in-sample lagged values are used for prediction. If dynamic is True, then in-sample forecasts are used in place of lagged dependent variables. The first forecasted value is start.

Returns predicted values : array

Notes

The linear Gaussian Kalman filter is used to return pre-sample fitted values. The exact initial Kalman Filter is used. See Durbin and Koopman in the references for more information.

statsmodels.tsa.ar_model.AR.score

AR.score(params)

Return the gradient of the loglikelihood at params.

Parameters params : array-like

The parameter values at which to evaluate the score function.
Notes

Returns numerical gradient.

```
statsmodels.tsa.ar_model.AR.select_order
AR.select_order(maxlag, ic, trend='c', method='mle')
```

Select the lag order according to the information criterion.

**Parameters**

- **maxlag** : int
  The highest lag length tried. See `AR.fit`.
- **ic** : str {'aic', 'bic', 'hqic', 't-stat'}
  Criterion used for selecting the optimal lag length. See `AR.fit`.
- **trend** : str {'c', 'nc'}
  Whether to include a constant or not. 'c' - include constant. 'nc' - no constant.

**Returns**

- **bestlag** : int
  Best lag according to IC.

**Attributes**

- `endog_names`
- `exog_names`

```
statsmodels.tsa.ar_model.ARResults
```

class statsmodels.tsa.ar_model.ARResults(model, params, normalized_cov_params=None, scale=1.0)

Class to hold results from fitting an AR model.

**Parameters**

- **model** : AR Model instance
  Reference to the model that is fit.
- **params** : array
  The fitted parameters from the AR Model.
- **normalized_cov_params** : array
  `inv(dot(X.T,X))` where X is the lagged values.
- **scale** : float, optional
  An estimate of the scale of the model.

**Returns**

**Attributes**

- **aic** : float
  Akaike Information Criterion using Lutkephol’s definition. \( \log(\sigma) + 2*(1+k_ar + k_trend)/nobs \)
- **bic** : float
  Bayes Information Criterion \( \log(\sigma) + (1+k_ar + k_trend) \ast \log(nobs)/nobs \)
bse : array

The standard errors of the estimated parameters. If method is ‘cmle’, then the standard
errors that are returned are the OLS standard errors of the coefficients. If the method is
‘mle’ then they are computed using the numerical Hessian.

fittedvalues : array

The in-sample predicted values of the fitted AR model. The k_ar initial values are
computed via the Kalman Filter if the model is fit by mle.

fpe : float

Final prediction error using Lutkepohl’s definition \(((n_{totobs}+k_{trend})/(n_{totobs}-k_{ar}-
k_{trend}))\)*sigma

hqic : float

Hannan-Quinn Information Criterion.

k_ar : float

Lag length. Sometimes used as p in the docs.

k_trend : float

The number of trend terms included. ‘nc’=0, ‘c’=1.

llf : float

The loglikelihood of the model evaluated at params. See AR.loglike

model : AR model instance

A reference to the fitted AR model.

nobs : float

The number of available observations nobs - k_ar

n_totobs : float

The number of total observations in endog. Sometimes n in the docs.

params : array

The fitted parameters of the model.

pvalues : array

The p values associated with the standard errors.

resid : array

The residuals of the model. If the model is fit by ‘mle’ then the pre-sample residuals are
calculated using fittedvalues from the Kalman Filter.

roots : array

The roots of the AR process are the solution to (1 - arparams[0]*z - arparams[1]*z**2
-...- arparams[p-1]*z**k_ar) = 0 Stability requires that the roots in modulus lie outside
the unit circle.

scale : float

Same as sigma2

sigma2 : float
The variance of the innovations (residuals).

**trendorder** : int

The polynomial order of the trend. ‘nc’ = None, ‘c’ or ‘t’ = 0, ‘ct’ = 1, etc.

**tvalues** : array

The t-values associated with *params*.

**Methods**

- **aic()**
- **bic()**
- **bse()**
- **conf_int([alpha, cols, method])**
  Returns the confidence interval of the fitted parameters.
- **cov_params([r_matrix, column, scale, cov_p, ...])**
  Returns the variance/covariance matrix.
- **f_test(r_matrix[, q_matrix, cov_p, scale, ...])**
  Compute the F-test for a joint linear hypothesis.
- **fittedvalues()**
- **fpe()**
- **hqic()**
- **initialize(model, params, **kwd)**
- **llf()**
- **load(fname)**
  load a pickle, (class method)
- **normalized_cov_params()**
- **predict([start, end, dynamic])**
  Returns in-sample and out-of-sample prediction.
- **pvalues()**
- **remove_data()**
- **resid()**
- **roots()**
- **save(fname[, remove_data])**
  save a pickle of this instance
- **scale()**
- **sigma2()**
- **t_test(r_matrix[, q_matrix, cov_p, scale, use_t])**
  Compute a t-test for a joint linear hypothesis of the form Rb = q
  Return the t-statistic for a given parameter estimate.
- **tvalues()**
- **wald_test(r_matrix[, q_matrix, cov_p, ...])**
  Compute a Wald-test for a joint linear hypothesis.

**`statsmodels.tsa.ar_model.ARResults.aic`**

**`statsmodels.tsa.ar_model.ARResults.bic`**

**`statsmodels.tsa.ar_model.ARResults.bse`**

**`statsmodels.tsa.ar_model.ARResults.conf_int`**

ARResults.conf_int (alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

**Parameters**

- **alpha** : float, optional
The \textit{alpha} level for the confidence interval. i.e., The default $\alpha = .05$ returns a 95% confidence interval.

\textbf{cols} : array-like, optional

\textit{cols} specifies which confidence intervals to return

\textbf{method} : string

Not Implemented Yet Method to estimate the confidence interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

\textbf{Returns} \texttt{conf_int} : array

Each row contains [lower, upper] confidence interval

\textbf{Notes}

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

\textbf{Examples}

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[ -5496529.48322745,  -1467987.78596704],
       [ -177.02903529,   207.15277984],
       [  -0.1115811 ,   0.03994274],
       [ -3.12506664,  -0.91539297],
       [ -1.5179487 ,  -0.54850503],
       [ -0.56251721,   0.460309 ],
       [  798.7875153 ,  2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[ -0.1115811 ,   0.03994274],
       [ -3.12506664,  -0.91539297]])
```

\textbf{statsmodels.tsa.ar_model.ARResults.cov_params} ARResults.\texttt{cov_params} (r\_matrix=None, column=None, scale=None, cov\_p=None, other=\texttt{None})

Returns the variance/covariance matrix.

The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of $\sigma^2$. Scale is assumed to be a scalar.

\textbf{Parameters} \texttt{r\_matrix} : array-like

Can be 1d, or 2d. Can be used alone or with other.

\textbf{column} : array-like, optional

Must be used on its own. Can be 0d or 1d see below.

\textbf{scale} : float, optional
Can be specified or not. Default is None, which means that the scale argument is taken from the model.

other : array-like, optional

Can be used when r_matrix is specified.

Returns (The below are assumed to be in matrix notation.):

cov : ndarray
If no argument is specified returns the covariance matrix of a model:
\[(\text{scale})*(X.T \times X)^{(-1)}:\]
If contrast is specified it pre and post-multiplies as follows:
\[(\text{scale}) * r\_matrix (X.T \times X)^{(-1)} r\_matrix.T:\]
If contrast and other are specified returns:
\[(\text{scale}) * r\_matrix (X.T \times X)^{(-1)} other.T:\]
If column is specified returns:
\[(\text{scale}) * (X.T \times X)^{(-1)}[\text{column},\text{column}]\] if column is 0d:
OR:
\[(\text{scale}) * (X.T \times X)^{(-1)}[\text{column}][:,\text{column}]\] if column is 1d:

```
statsmodels.tsa.ar_model.ARResults.f_test
ARResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
```
Compute the F-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
- array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
Default is 1.0 for no scaling.

invcov : array-like, optional
A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:

statsmodels.contrasts.statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint
Notes

The matrix $r_{matrix}$ is assumed to be non-singular. More precisely, 

$r_{matrix} (pX pX.T) r_{matrix}.T$

is assumed invertible. Here, $pX$ is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm

>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]

This tests that each coefficient is jointly statistically significantly different from zero.

```{.highlight}
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>
```{.highlight}

Compare this to

```{.highlight}
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10
```

```{.highlight}
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
```

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```{.highlight}
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>
```

Alternatively, you can specify the hypothesis tests using a string

```{.highlight}
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols

>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, data).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

statsmodels.tsa.ar_model.ARResults.hqic
static ARResults.hqic()

statsmodels.tsa.ar_model.ARResults.initialize
ARResults.initialize(model, params, **kwds)

statsmodels.tsa.ar_model.ARResults.llf
static ARResults.llf()

statsmodels.tsa.ar_model.ARResults.load
classmethod ARResults.load(fname)
load a pickle, (class method)

Parameters
name : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance:

statsmodels.tsa.ar_model.ARResults.normalized_cov_params
ARResults.normalized_cov_params()

statsmodels.tsa.ar_model.ARResults.predict
ARResults.predict(start=None, end=None, dynamic=False)
Returns in-sample and out-of-sample prediction.

Parameters
start : int, str, or datetime
0-indexed observation number at which to start forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type.

end : int, str, or datetime
0-indexed observation number at which to end forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type.

dynamic : bool
The dynamic keyword affects in-sample prediction. If dynamic is False, then the in-sample lagged values are used for prediction. If dynamic is True, then in-sample forecasts are used in place of lagged dependent variables. The first forecasted

confint : bool, float
Whether to return confidence intervals. If confint == True, 95% confidence intervals are returned. Else if confint is a float, then it is assumed to be the alpha value of the confidence interval. That is confint == .05 returns a 95% confidence interval, and .10 would return a 90% confidence interval. value is start.

Returns predicted values: array

Notes
The linear Gaussian Kalman filter is used to return pre-sample fitted values. The exact initial Kalman Filter is used. See Durbin and Koopman in the references for more information.
ARResults.pvalues

ARResults.remove_data

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

Warning: Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

ARResults.resid

ARResults.roots

ARResults.save

save a pickle of this instance

Parameters

fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

remove_data : bool
If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

ARResults.scale

ARResults.sigma2
statsmodels.tsa_ar_model.ARResults.t_test

```
ARResults.t_test(r_matrix=None, q_matrix=None, cov_p=None, scale=None, use_t=None)
```

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

**Parameters**

`r_matrix` : array-like, str, tuple

- array : If an array is given, a $p \times k$ 2d array or length $k$ 1d array specifying the linear restrictions.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form $(R, q)$, since $q_matrix$ is deprecated.

`q_matrix` : array-like or scalar, optional

This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length $p$ row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.

`cov_p` : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

`scale` : float, optional

An optional `scale` to use. Default is the scale specified by the model fit.

`use_t` : bool, optional

If `use_t` is None, then the default of the model is used. If `use_t` is True, then the p-values are based on the t distribution. If `use_t` is False, then the p-values are based on the normal distribution.

**See also:**

`tvalues` individual t statistics

`f_test` for F tests

`patsy.DesignInfo.linear_constraint`

**Examples**

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]
```

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_Test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string
```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

**statsmodels.tsa.ar_model.ARResults.tvalues**

`ARResults.tvalues()`

Return the t-statistic for a given parameter estimate.

**statsmodels.tsa.ar_model.ARResults.wald_test**

`ARResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)`

Compute a Wald-test for a joint linear hypothesis.

**Parameters**

- **r_matrix** : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p** : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale** : float, optional
  Default is 1.0 for no scaling.

- **invcov** : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

- **use_f** : bool
  If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

**See also:**

- `statsmodels.contrasts`, `statsmodels.model.LikelihoodModelResults.f_test`, `statsmodels.model.LikelihoodModelResults.t_test`, `patsy.DesignInfo.linear_constraint`

**Notes**

The matrix r_matrix is assumed to be non-singular. More precisely,

r_matrix (pX pX.T) r_matrix.T
is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

### Attributes

- **extra_doc**
  - `list() -> new empty list`

- **preddoc**
  - `list() -> new empty list`

### Autoregressive Moving-Average Processes (ARMA) and Kalman Filter

- **arima_model.ARMA**
  - `endog[, order, exog, ...])`
  - Autoregressive Moving Average ARMA(p,q) Model

- **arima_model.ARMAResults**
  - `(model, params[, ...])`
  - Class to hold results from fitting an ARMA model.

- **arima_model.ARIMA**
  - `endog[, order[, exog, ...]])`
  - Autoregressive Integrated Moving Average ARIMA(p,d,q) Model

- **kalmanf.kalmanfilter.KalmanFilter**
  - Kalman Filter code intended for use with the ARMA model.

### `statsmodels.tsa.arima_model.ARMA`

#### Class `statsmodels.tsa.arima_model.ARMA` *(endog, order=None, exog=None, dates=None, freq=None, missing='none')*

**Autoregressive Moving Average ARMA(p,q) Model**

**Parameters**

- **endog** : array-like
  - The endogenous variable.

- **order** : iterable
  - The (p,q) order of the model for the number of AR parameters, differences, and MA parameters to use. Though optional, the order keyword in fit is deprecated and it is recommended to give order here.

- **exog** : array-like, optional
  - An optional arry of exogenous variables. This should not include a constant or trend. You can specify this in the `fit` method.

- **dates** : array-like of datetime, optional
  - An array-like object of datetime objects. If a pandas object is given for endog or exog, it is assumed to have a DateIndex.

- **freq** : str, optional
  - The frequency of the time-series. A Pandas offset or ‘B’, ‘D’, ‘W’, ‘M’, ‘A’, or ‘Q’. This is optional if dates are given.

**Notes**

If exogenous variables are given, then the model that is fit is

\[
\phi(L)(y_t - X_t\beta) = \theta(L)\varepsilon_t
\]

where \( \phi \) and \( \theta \) are polynomials in the lag operator, \( L \). This is the regression model with ARMA errors, or ARMAX model. This specification is used, whether or not the model is fit using conditional sum of square
or maximum-likelihood, using the method argument in `statsmodels.tsa.arima_model.ARMA.fit`. Therefore, for now, css and mle refer to estimation methods only. This may change for the case of the css model in future versions.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit(order, start_params, trend, method, ...)</code></td>
<td>Fits ARMA(p,q) model using exact maximum likelihood via Kalman filter.</td>
</tr>
<tr>
<td><code>from_formula(formula, data[, subset])</code></td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td><code>geterrors(params)</code></td>
<td>Get the errors of the ARMA process.</td>
</tr>
<tr>
<td><code>hessian(params)</code></td>
<td>Compute the Hessian at params,</td>
</tr>
<tr>
<td><code>information(params)</code></td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td><code>initialize()</code></td>
<td>Initialize (possibly re-initialize) a Model instance. For</td>
</tr>
<tr>
<td><code>loglike(params[, set_sigma2])</code></td>
<td>Compute the log-likelihood for ARMA(p,q) model</td>
</tr>
<tr>
<td><code>loglike_css(params[, set_sigma2])</code></td>
<td>Conditional Sum of Squares likelihood function.</td>
</tr>
<tr>
<td><code>loglike_kalman(params[, set_sigma2])</code></td>
<td>Compute exact loglikelihood for ARMA(p,q) model by the Kalman Filter.</td>
</tr>
<tr>
<td><code>predict(params[, start, end, exog, dynamic])</code></td>
<td>ARMA model in-sample and out-of-sample prediction</td>
</tr>
<tr>
<td><code>score(params)</code></td>
<td>Compute the score function at params.</td>
</tr>
</tbody>
</table>

```python
statsmodels.tsa.arima_model.ARMA.fit
ARMA.fit(order=None, start_params=None, trend='c', method='css-mle', transparams=True, solver='lbfgs', maxiter=50, full_output=1, disp=5, callback=None, **kwargs)
```

Fits ARMA(p,q) model using exact maximum likelihood via Kalman filter.

**Parameters**

- **start_params**: array-like, optional
  Starting parameters for ARMA(p,q). If None, the default is given by ARMA.fit_start_params. See there for more information.

- **transparams**: bool, optional
  Whether or not to transform the parameters to ensure stationarity. Uses the transformation suggested in Jones (1980). If False, no checking for stationarity or invertibility is done.

- **method** : str (`'css-mle', 'mle', 'css'`)
  This is the loglikelihood to maximize. If “css-mle”, the conditional sum of squares likelihood is maximized and its values are used as starting values for the computation of the exact likelihood via the Kalman filter. If “mle”, the exact likelihood is maximized via the Kalman Filter. If “css” the conditional sum of squares likelihood is maximized. All three methods use `start_params` as starting parameters. See above for more information.

- **trend** : str (`'c', 'nc'`)
  Whether to include a constant or not. ‘c’ includes constant, ‘nc’ no constant.

- **solver** : str or None, optional
  Solver to be used. The default is ‘lbfgs’ (limited memory Broyden-Fletcher-Goldfarb-Shanno). Other choices are ‘bfgs’, ‘newton’ (Newton-Raphson), ‘nm’ (Nelder-Mead), ‘cg’ - (conjugate gradient), ‘ncg’ (non-conjugate gradient), and ‘powell’. By default, the limited memory BFGS uses m=12 to approximate the Hessian, projected gradient tolerance of 1e-8 and factr = 1e2. You can change these by using kwargs.

- **maxiter** : int, optional
  The maximum number of function evaluations. Default is 50.
tol : float
    The convergence tolerance. Default is 1e-08.

full_output : bool, optional
    If True, all output from solver will be available in the Results object’s mle_retvals attribute. Output is dependent on the solver. See Notes for more information.

disp : bool, optional
    If True, convergence information is printed. For the default l_bfgs_b solver, disp controls the frequency of the output during the iterations. disp < 0 means no output in this case.

callback : function, optional
    Called after each iteration as callback(xk) where xk is the current parameter vector.

kw_args : 
    See Notes for keyword arguments that can be passed to fit.

Returns statsmodels.tsa.arima_model.ARMAResults class :

See also:

statsmodels.base.model.LikelihoodModel.fit for more information on using the solvers.

ARMAResults results class returned by fit

Notes

If fit by ‘mle’, it is assumed for the Kalman Filter that the initial unkown state is zero, and that the initial variance is
P = dot(inv(identity(m**2)-kron(T,T)),dot(R,R.T).ravel(‘F’)).reshape(r, r, order = ‘F’)

statsmodels.tsa.arima_model.ARMA.from_formula
classmethod ARMA.from_formula(formula, data, subset=None, *args, **kwargs)
    Create a Model from a formula and dataframe.

Parameters formula : str or generic Formula object
    The formula specifying the model

data : array-like
    The data for the model. See Notes.

subset : array-like
    An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

args : extra arguments
    These are passed to the model

kw_args : extra keyword arguments
    These are passed to the model.

Returns model : Model instance
Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**ARMA.geterrors**

Get the errors of the ARMA process.

**Parameters**

```
params : array-like
The fitted ARMA parameters
```

**order : array-like**

```
3 item iterable, with the number of AR, MA, and exogenous parameters, including the trend
```

**ARMA.hessian**

Compute the Hessian at params.

**Notes**

This is a numerical approximation.

**ARMA.information**

Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

**ARMA.initialize**

Initialize (possibly re-initialize) a Model instance. For instance, the design matrix of a linear model may change and some things must be recomputed.

**ARMA.loglike**

Compute the log-likelihood for ARMA(p,q) model

**Notes**

Likelihood used depends on the method set in fit

**ARMA.loglike_css**

Conditional Sum of Squares likelihood function.
ARMA.loglike_kalman (params, set_sigma2=True)
Compute exact loglikelihood for ARMA(p,q) model by the Kalman Filter.

ARMA.predict (params, start=None, end=None, exog=None, dynamic=False)
ARMA model in-sample and out-of-sample prediction

Parameters
params : array-like
The fitted parameters of the model.

start : int, str, or datetime
Zero-indexed observation number at which to start forecasting, ie., the first forecast is start. Can also be a date string to parse or a datetime type.

end : int, str, or datetime
Zero-indexed observation number at which to end forecasting, ie., the first forecast is start. Can also be a date string to parse or a datetime type. However, if the dates index does not have a fixed frequency, end must be an integer index if you want out of sample prediction.

exog : array-like, optional
If the model is an ARMAX and out-of-sample forecasting is requested, exog must be given. Note that you’ll need to pass k_ar additional lags for any exogenous variables. E.g., if you fit an ARMAX(2, q) model and want to predict 5 steps, you need 7 observations to do this.

dynamic : bool, optional
The dynamic keyword affects in-sample prediction. If dynamic is False, then the in-sample lagged values are used for prediction. If dynamic is True, then in-sample forecasts are used in place of lagged dependent variables. The first forecasted value is start.

Returns
predict : array
The predicted values.

Notes
Use the results predict method instead.

ARMA.score (params)
Compute the score function at params.

Notes
This is a numerical approximation.
Class `statsmodels.tsa.arima_model.ARMAResults(model, params, normalized_cov_params=None, scale=1.0)`

Class to hold results from fitting an ARMA model.

**Parameters**

- **model**: ARMA instance
  - The fitted model instance
- **params**: array
  - Fitted parameters
- **normalized_cov_params**: array, optional
  - The normalized variance covariance matrix
- **scale**: float, optional
  - Optional argument to scale the variance covariance matrix.

**Returns**

**Attributes**:

- **aic**: float
  - Akaike Information Criterion $-2 \times llf + 2 \times df_model$ where $df_model$ includes all AR parameters, MA parameters, constant terms parameters on constant terms and the variance.
- **arparams**: array
  - The parameters associated with the AR coefficients in the model.
- **arroots**: array
  - The roots of the AR coefficients are the solution to $(1 - arparams[0]*z - arparams[1]*z^2 - ... - arparams[p-1]*z^k_ar) = 0$ Stability requires that the roots in modulus lie outside the unit circle.
- **bic**: float
  - Bayes Information Criterion $-2*llf + \log(nobs)*df_model$ Where if the model is fit using conditional sum of squares, the number of observations $nobs$ does not include the $p$ pre-sample observations.
- **bse**: array
  - The standard errors of the parameters. These are computed using the numerical Hessian.
- **df_model**: array
  - The model degrees of freedom $= k_exog + k_trend + k_ar + k_ma$
- **df_resid**: array
  - The residual degrees of freedom $= nobs - df_model$
- **fittedvalues**: array
  - The predicted values of the model.
hqic : float
    Hannan-Quinn Information Criterion -2*llf + 2*(df_model)*log(log(nobs)) Like bic if
    the model is fit using conditional sum of squares then the k_ar pre-sample observations
    are not counted in nobs.

k_ar : int
    The number of AR coefficients in the model.

k_exog : int
    The number of exogenous variables included in the model. Does not include the con-
    stant.

k_ma : int
    The number of MA coefficients.

k_trend : int
    This is 0 for no constant or 1 if a constant is included.

llf : float
    The value of the log-likelihood function evaluated at params.

maparams : array
    The value of the moving average coefficients.

maroots : array
    The roots of the MA coefficients are the solution to (1 + maparams[0]*z + ma-
    params[1]*z**2 + ... + maparams[q-1]*z**q) = 0 Stability requires that the roots in
    modules lie outside the unit circle.

model : ARMA instance
    A reference to the model that was fit.

nobs : float
    The number of observations used to fit the model. If the model is fit using exact maxi-
    mum likelihood this is equal to the total number of observations, n_totobs. If the model
    is fit using conditional maximum likelihood this is equal to n_totobs - k_ar.

n_totobs : float
    The total number of observations for endog. This includes all observations, even pre-
    sample values if the model is fit using css.

params : array
    The parameters of the model. The order of variables is the trend coefficients and the
    k_exog exognous coefficients, then the k_ar AR coefficients, and finally the k_ma MA
    coefficients.

pvalues : array
    The p-values associated with the t-values of the coefficients. Note that the coefficients
    are assumed to have a Student’s T distribution.

resid : array

3.7. Time Series analysis ts

statsmodels Documentation, Release 0.6.0
The model residuals. If the model is fit using ‘mle’ then the residuals are created via the Kalman Filter. If the model is fit using ‘css’ then the residuals are obtained via scipy.signal.lfilter adjusted such that the first $k_{\text{ma}}$ residuals are zero. These zero residuals are not returned.

scale : float

This is currently set to 1.0 and not used by the model or its results.

sigma2 : float

The variance of the residuals. If the model is fit by ‘css’, $\sigma^2 = \text{ssr}/nobs$, where ssr is the sum of squared residuals. If the model is fit by ‘mle’, then $\sigma^2 = 1/nobs \times \sum(v^2/F)$ where $v$ is the one-step forecast error and $F$ is the forecast error variance. See nobs for the difference in definitions depending on the fit.

**Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aic()</td>
<td>Returns the frequency of the AR roots.</td>
</tr>
<tr>
<td>arfreq()</td>
<td>Returns the frequency of the MA roots.</td>
</tr>
<tr>
<td>arparams()</td>
<td></td>
</tr>
<tr>
<td>arroots()</td>
<td></td>
</tr>
<tr>
<td>bic()</td>
<td>Returns the confidence interval of the fitted parameters.</td>
</tr>
<tr>
<td>bse()</td>
<td></td>
</tr>
<tr>
<td>conf_int([alpha, cols, method])</td>
<td>Compute a t-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td>cov_params()</td>
<td></td>
</tr>
<tr>
<td>f_test(r_matrix[, q_matrix, cov_p, scale, ...])</td>
<td>Out-of-sample forecasts</td>
</tr>
<tr>
<td>fittedvalues()</td>
<td></td>
</tr>
<tr>
<td>forecast([steps, exog, alpha])</td>
<td>ARMA model in-sample and out-of-sample prediction</td>
</tr>
<tr>
<td>hqic()</td>
<td>Load a pickle, (class method)</td>
</tr>
<tr>
<td>initialize(model, params, **kwds)</td>
<td>Remove data arrays, all nobs arrays from result and model</td>
</tr>
<tr>
<td>llf()</td>
<td></td>
</tr>
<tr>
<td>load(fname)</td>
<td></td>
</tr>
<tr>
<td>mafreq()</td>
<td>Returns the frequency of the MA roots.</td>
</tr>
<tr>
<td>maparams()</td>
<td></td>
</tr>
<tr>
<td>maroots()</td>
<td></td>
</tr>
<tr>
<td>normalized_cov_params()</td>
<td>Summarize the Model</td>
</tr>
<tr>
<td>predict([start, end, exog, dynamic])</td>
<td>Experimental summary function for ARIMA Results</td>
</tr>
<tr>
<td>pvalues()</td>
<td></td>
</tr>
<tr>
<td>remove_data()</td>
<td></td>
</tr>
<tr>
<td>resid()</td>
<td></td>
</tr>
<tr>
<td>save(fname[, remove_data])</td>
<td>Compute a t-test for a joint linear hypothesis of the form $Rb = q$.</td>
</tr>
<tr>
<td>summary([alpha])</td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td>summary2([title, alpha, float_format])</td>
<td>Return the t-statistic for a given parameter estimate.</td>
</tr>
<tr>
<td>t_test(r_matrix[, q_matrix, cov_p, scale, use_t])</td>
<td>Compute a Wald-test for a joint linear hypothesis.</td>
</tr>
<tr>
<td>tvalues()</td>
<td></td>
</tr>
<tr>
<td>wald_test(r_matrix[, q_matrix, cov_p, ...])</td>
<td></td>
</tr>
</tbody>
</table>

statsmodels.tsa.arima_model.ARMAResults.aic

statsmodels.tsa.arima_model.ARMAResults.arfreq
statsmodels Documentation, Release 0.6.0

static ARMAResults.arfreq()
    Returns the frequency of the AR roots.
    This is the solution, x, to \( z = \text{abs}(z) \cdot \exp(2j\pi x) \) where z are the roots.

statsmodels.tsa.arima_model.ARMAResults.arparams
static ARMAResults.arparams()

statsmodels.tsa.arima_model.ARMAResults.arroots
static ARMAResults.arroots()

statsmodels.tsa.arima_model.ARMAResults.bic
static ARMAResults.bic()

statsmodels.tsa.arima_model.ARMAResults.bse
static ARMAResults.bse()

statsmodels.tsa.arima_model.ARMAResults.conf_int
ARMAResults.conf_int(alpha=0.05, cols=None, method='default')
    Returns the confidence interval of the fitted parameters.

    Parameters
    -------
    alpha : float, optional
        The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95%
        confidence interval.
    cols : array-like, optional
        cols specifies which confidence intervals to return

    method : string
        Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses
        self.bse which is based on inverse Hessian for MLE “jh” : “jac” : “boot-bse”
        “boot_quant” “profile”

    Returns
    -------
    conf_int : array
        Each row contains [lower, upper] confidence interval

Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different
distribution should overwrite this method.

Examples

>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[ 5.49652948, 43.227454],
       [-1.48798778, 7.85967041],
       [ 0.43798778, 2.85967041]])
```python
>>> results.conf_int(cols=(2,3))
array([[ 0.11158111, 0.03994274],
       [-3.12506664, -0.91539297]])
```

**statsmodels.tsa.arima_model.ARMAResults.cov_params**

`ARMAResults.cov_params()`

**statsmodels.tsa.arima_model.ARMAResults.f_test**

`ARMAResults.f_test(r_matrix=q_matrix=None, cov_p=None, scale=1.0, invcov=None)`

Compute the F-test for a joint linear hypothesis.

**Parameters**

- **r_matrix** : array-like, str, or tuple
  
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  
  - str : The full hypotheses to test can be given as a string. See the examples.
  
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like
  
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformance array of zeros.

- **cov_p** : array-like, optional
  
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale** : float, optional
  
  Default is 1.0 for no scaling.

- **invcov** : array-like, optional
  
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:

- `statsmodels.contrasts.statsmodels.model.LikelihoodModelResults.wald_test`
- `statsmodels.model.LikelihoodModelResults.t_test`, `patsy.DesignInfo.linear_constraint`

**Notes**

The matrix r_matrix is assumed to be non-singular. More precisely, r_matrix (pX pX.T) r_matrix.T is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
```

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
 df_denom=9, df_num=6>
```

Compare this to

```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10
```

```python
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
```

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
 df_num=2>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
additional lags for any exogenous variables. E.g., if you fit an ARMAX(2, q) model and
want to predict 5 steps, you need 7 observations to do this.

alpha : float
 The confidence intervals for the forecasts are (1 - alpha) %

Returns forecast : array
 Array of out of sample forecasts

stderr : array
 Array of the standard error of the forecasts.

conf_int : array
 2d array of the confidence interval for the forecast

statsmodels.tsa.arima_model.ARMAResults.hqic
static ARMAResults.hqic()

statsmodels.tsa.arima_model.ARMAResults.initialize
ARMAResults.initialize(model, params, **kwd)

statsmodels.tsa.arima_model.ARMAResults.llf
static ARMAResults.llf()

statsmodels.tsa.arima_model.ARMAResults.load
classmethod ARMAResults.load(fname)
       load a pickle, (class method)

       Parameters fname : string or filehandle
       fname can be a string to a file path or filename, or a filehandle.

       Returns unpickled instance :

statsmodels.tsa.arima_model.ARMAResults.mafreq
static ARMAResults.mafreq()
       Returns the frequency of the MA roots.

       This is the solution, x, to z = abs(z)*exp(2j*np.pi*x) where z are the roots.

statsmodels.tsa.arima_model.ARMAResults.maparams
static ARMAResults.maparams()

statsmodels.tsa.arima_model.ARMAResults.maroots
static ARMAResults.maroots()

statsmodels.tsa.arima_model.ARMAResults.normalized_cov_params
ARMAResults.normalized_cov_params()
ARMA model in-sample and out-of-sample prediction

Parameters

- **start**: int, str, or datetime
  
  Zero-indexed observation number at which to start forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type.

- **end**: int, str, or datetime
  
  Zero-indexed observation number at which to end forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type. However, if the dates index does not have a fixed frequency, end must be an integer index if you want out of sample prediction.

- **exog**: array-like, optional
  
  If the model is an ARMAX and out-of-sample forecasting is requested, exog must be given. Note that you’ll need to pass \( k_{ar} \) additional lags for any exogenous variables. E.g., if you fit an ARMAX(2, q) model and want to predict 5 steps, you need 7 observations to do this.

- **dynamic**: bool, optional
  
  The `dynamic` keyword affects in-sample prediction. If dynamic is False, then the in-sample lagged values are used for prediction. If `dynamic` is True, then in-sample forecasts are used in place of lagged dependent variables. The first forecasted value is `start`.

Returns

- **predict**: array
  
  The predicted values.

Notes

It is recommended to use dates with the time-series models, as the below will probably make clear. However, if ARIMA is used without dates and/or `start` and `end` are given as indices, then these indices are in terms of the original, undifferenced series. I.e., given some undifferenced observations:

1970Q1, 1  
1970Q2, 1.5  
1970Q3, 1.25  
1970Q4, 2.25  
1971Q1, 1.2  
1971Q2, 4.1

1970Q1 is observation 0 in the original series. However, if we fit an ARIMA(p,1,q) model then we lose this first observation through differencing. Therefore, the first observation we can forecast (if using exact MLE) is index 1. In the differenced series this is index 0, but we refer to it as 1 from the original series.

statsmodels.tsa.arima_model.ARMAResults.pvalues

static ARMAResults.pvalues()

statsmodels.tsa.arima_model.ARMAResults.remove_data

ARMAResults.remove_data()

remove data arrays, all nobs arrays from result and model
This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

```python
def resid():
    ...  

def save(fname, remove_data=False):
    ...  
```

**Parameters**

- **fname**  : string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- **remove_data**  : bool

  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

```python
def summary(alpha=0.05):
    ...  
```

**Parameters**

- **alpha**  : float, optional

  Significance level for the confidence intervals.

**Returns**

- **smry**  : Summary instance

  This holds the summary table and text, which can be printed or converted to various output formats.

**See also:**

- `statsmodels.iolib.summary.Summary`
Experimental summary function for ARIMA Results

Parameters:

- **title**: string, optional
  Title for the top table. If not None, then this replaces the default title
- **alpha**: float
  Significance level for the confidence intervals
- **float_format**: string
  Print format for floats in parameters summary

Returns:

- **smry**: Summary instance
  This holds the summary table and text, which can be printed or converted to various output formats.

See also:

- [Summary](https://www.statsmodels.org/stable/generated/statsmodels.iolib.summary2.Summary.html) class to hold summary results

Statsmodels Documentation, Release 0.6.0

3.7. Time Series analysis `tsa`

3.7.1. Time Series Analysis in Python

---

**Summary**

- **title**: string, optional
  Title for the top table. If not None, then this replaces the default title
- **alpha**: float
  Significance level for the confidence intervals
- **float_format**: string
  Print format for floats in parameters summary

**Parameters**

- **r_matrix**: array-like, str, tuple
  - array: If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.
- **q_matrix**: array-like or scalar, optional
  This is deprecated. See `r_matrix` and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.
- **cov_p**: array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
- **scale**: float, optional
  An optional scale to use. Default is the scale specified by the model fit.
- **use_t**: bool, optional
  If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_test)
<T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

The `t_test` function in `statsmodels` is used to calculate the t-statistic for a given parameter estimate. It can also be used to specify joint linear hypotheses using a string or matrix. The `wald_test` function is used to compute a Wald-test for a joint linear hypothesis.
scale : float, optional
    Default is 1.0 for no scaling.

invcov : array-like, optional
    A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool
    If True, then the F-distribution is used. If False, then the asymptotic distribution,
    chisquare is used. The test statistic is proportionally adjusted for the distribution by
    the number of constraints in the hypothesis.

See also:
    statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
    statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes
    The matrix \( r\_matrix \) is assumed to be non-singular. More precisely,
    \( r\_matrix (pX pX.T) r\_matrix.T \)
    is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be
    problems in non-OLS models where the rank of the covariance of the noise is not full.

statsmodels.tsa.arima_model.ARIMA

class statsmodels.tsa.arima_model.ARIMA(endog, order, exog=None, dates=None, freq=None, missing='none')
    Autoregressive Integrated Moving Average ARIMA(p,d,q) Model

Parameters
    endog : array-like
        The endogenous variable.

    order : iterable
        The (p,d,q) order of the model for the number of AR parameters, differences, and MA
        parameters to use.

    exog : array-like, optional
        An optional array of exogenous variables. This should not include a constant or trend.
        You can specify this in the fit method.

    dates : array-like of datetime, optional
        An array-like object of datetime objects. If a pandas object is given for endog or exog,
        it is assumed to have a DateIndex.

    freq : str, optional
        This is optional if dates are given.
Notes

If exogenous variables are given, then the model that is fit is

$$\phi(L)(y_t - X_t\beta) = \theta(L)\epsilon_t$$

where $\phi$ and $\theta$ are polynomials in the lag operator, $L$. This is the regression model with ARMA errors, or ARMAX model. This specification is used, whether or not the model is fit using conditional sum of square or maximum-likelihood, using the method argument in `statsmodels.tsa.arima_model.ARIMA.fit`. Therefore, for now, `css` and `mle` refer to estimation methods only. This may change for the case of the `css` model in future versions.

Methods

```
fit([start_params, trend, method, ...])
from_formula(formula, data[, subset])
geterrors(params)
hessian(params)
information=params)
initialize()
loglike(params[, set_sigma2])
loglike_css(params[, set_sigma2])
loglike_kalman(params[, set_sigma2])
predict(params[, start, end, exog, typ, dynamic])
score(params)
```

- `fit([start_params, trend, method, ...])`: Fits ARIMA(p,d,q) model by exact maximum likelihood via Kalman filter.
- `from_formula(formula, data[, subset])`: Create a Model from a formula and dataframe.
- `geterrors(params)`: Get the errors of the ARMA process.
- `hessian(params)`: Compute the Hessian at params.
- `information(params)`: Fisher information matrix of model.
- `initialize()`: Initialize (possibly re-initialize) a Model instance. For
- `loglike(params[, set_sigma2])`: Compute the log-likelihood for ARMA(p,q) model.
- `loglike_css(params[, set_sigma2])`: Conditional Sum of Squares likelihood function.
- `loglike_kalman(params[, set_sigma2])`: Compute exact loglikelihood for ARMA(p,q) model by the Kalman Filter.
- `predict(params[, start, end, exog, typ, dynamic])`: ARIMA model in-sample and out-of-sample prediction.
- `score(params)`: Compute the score function at params.

```
statsmodels.tsa.arima_model.ARIMA.fit
ARIMA.fit (start_params=None, trend='c', method='css-mle', transparams=True, solver='lbfgs', maxiter=50, full_output=1, disp=5, callback=None, **kwargs)
```

Fits ARIMA(p,d,q) model by exact maximum likelihood via Kalman filter.

**Parameters**

- `start_params`: array-like, optional
  - Starting parameters for ARMA(p,q). If None, the default is given by `ARIMA.fit` start_params. See there for more information.
- `transparams`: bool, optional
  - Whether or not to transform the parameters to ensure stationarity. Uses the transformation suggested in Jones (1980). If False, no checking for stationarity or invertibility is done.
- `method`: str {'css-mle','mle','css'}
  - This is the loglikelihood to maximize. If “css-mle”, the conditional sum of squares likelihood is maximized and its values are used as starting values for the computation of the exact likelihood via the Kalman filter. If “mle”, the exact likelihood is maximized via the Kalman Filter. If “css” the conditional sum of squares likelihood is maximized. All three methods use `start_params` as starting parameters. See above for more information.
- `trend`: str {'c','nc'}
  - Whether to include a constant or not. ‘c’ includes constant, ‘nc’ no constant.
- `solver`: str or None, optional
Solver to be used. The default is ‘lbfgs’ (limited memory Broyden-Fletcher-Goldfarb-Shanno). Other choices are ‘bfgs’, ‘newton’ (Newton-Raphson), ‘nm’ (Nelder-Mead), ‘cg’ - (conjugate gradient), ‘ncg’ (non-conjugate gradient), and ‘powell’. By default, the limited memory BFGS uses m=12 to approximate the Hessian, projected gradient tolerance of 1e-8 and factr = 1e2. You can change these by using kwargs.

maxiter : int, optional

The maximum number of function evaluations. Default is 50.

tol : float

The convergence tolerance. Default is 1e-08.

full_output : bool, optional

If True, all output from solver will be available in the Results object’s mle_retvals attribute. Output is dependent on the solver. See Notes for more information.

disp : bool, optional

If True, convergence information is printed. For the default l_bfgs_b solver, disp controls the frequency of the output during the iterations. disp < 0 means no output in this case.

callback : function, optional

Called after each iteration as callback(xk) where xk is the current parameter vector.

kwargs :

See Notes for keyword arguments that can be passed to fit.

Returns ‘statsmodels.tsa.arima_ARIMAResults‘ class :

See also:

statsmodels.base.model.LikelihoodModel.fit for more information on using the solvers.
ARIMAResults results class returned by fit

Notes

If fit by ‘mle’, it is assumed for the Kalman Filter that the initial unkown state is zero, and that the initial variance is P = dot(inv(identity(m**2)-kron(T,T)),dot(R,R.T).ravel('F')).reshape(r, r, order = ‘F’)

statsmodels.tsa.arima_model.ARIMA.from_formula

classmethod ARIMA.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

Parameters formula : str or generic Formula object

The formula specifying the model

data : array-like

The data for the model. See Notes.

subset : array-like

An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
**args**: extra arguments
These are passed to the model

**kwargs**: extra keyword arguments
These are passed to the model.

**Returns model**: Model instance

**Notes**

Data must define `__getitem__` with the keys in the formula terms `args` and `kwargs` are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

```python
statsmodels.tsa.arima_model.ARIMA.geterrors
ARIMA.geterrors(params)
Get the errors of the ARMA process.

Parameters params : array-like
The fitted ARMA parameters

order : array-like
3 item iterable, with the number of AR, MA, and exogenous parameters, including the trend
```

```python
statsmodels.tsa.arima_model.ARIMA.hessian
ARIMA.hessian(params)
Compute the Hessian at params,

**Notes**

This is a numerical approximation.

```python
statsmodels.tsa.arima_model.ARIMA.information
ARIMA.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.
```

```python
statsmodels.tsa.arima_model.ARIMA.initialize
ARIMA.initialize()
Initialize (possibly re-initialize) a Model instance. For instance, the design matrix of a linear model may change and some things must be recomputed.
```

```python
statsmodels.tsa.arima_model.ARIMA.loglike
ARIMA.loglike(params, set_sigma2=True)
Compute the log-likelihood for ARMA(p,q) model
```
Notes

Likelihood used depends on the method set in fit

statsmodels.tsa.arima_model.ARIMA.loglike_css

ARIMA.loglike_css(params, set_sigma2=True)
Conditional Sum of Squares likelihood function.

statsmodels.tsa.arima_model.ARIMA.loglike_kalman

ARIMA.loglike_kalman(params, set_sigma2=True)
Compute exact loglikelihood for ARMA(p,q) model by the Kalman Filter.

statsmodels.tsa.arima_model.ARIMA.predict

ARIMA.predict(params, start=None, end=None, exog=None, typ='linear', dynamic=False)
ARIMA model in-sample and out-of-sample prediction

Parameters

params : array-like
The fitted parameters of the model.

start : int, str, or datetime
Zero-indexed observation number at which to start forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type.

end : int, str, or datetime
Zero-indexed observation number at which to end forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type. However, if the dates index does not have a fixed frequency, end must be an integer index if you want out of sample prediction.

exog : array-like, optional
If the model is an ARMAX and out-of-sample forecasting is requested, exog must be given. Note that you’ll need to pass k_ar additional lags for any exogenous variables. E.g., if you fit an ARMAX(2, q) model and want to predict 5 steps, you need 7 observations to do this.

dynamic : bool, optional
The dynamic keyword affects in-sample prediction. If dynamic is False, then the in-sample lagged values are used for prediction. If dynamic is True, then in-sample forecasts are used in place of lagged dependent variables. The first forecasted value is start.

typ : str {'linear', 'levels'}
- ‘linear’ : Linear prediction in terms of the differenced endogenous variables.
- ‘levels’ : Predict the levels of the original endogenous variables.

Returns

predict : array
The predicted values.

Notes

Use the results predict method instead.
ARIMA.score(params)
Compute the score function at params.

Notes
This is a numerical approximation.

Attributes

| endog_names |
| exog_names |

ARIMAResults

Attributes

endog_names
exog_names

Methods

aic()
arfreq()
arparams()
arroots()
bic()
bse()
conf_int([alpha, cols, method])
cov_params()
f_test(r_matrix[, q_matrix, cov_p, scale, ...])
fittedvalues()
forecast([steps, exog, alpha])
hqic()
initialize(model, params, **kwd)
llf()
load(fname)
mafreq()
maparams()
maroots()
normalized_cov_params()
predict([start, end, exog, typ, dynamic])
pvalues()
remove_data()
resid()
save(fname[, remove_data])
summary([alpha])
summary2([title, alpha, float_format])
t_test(r_matrix[, q_matrix, cov_p, scale, use_t])
tvalues()

Continued on next page
wald_test(r_matrix[, q_matrix, cov_p, ...])  Compute a Wald-test for a joint linear hypothesis.

statsmodels.tsa.arima_model.ARIMAResults.aic
static ARIMAResults.aic()

statsmodels.tsa.arima_model.ARIMAResults.arfreq
static ARIMAResults.arfreq()
    Returns the frequency of the AR roots.
    This is the solution, x, to z = abs(z)*exp(2j*π*x) where z are the roots.

statsmodels.tsa.arima_model.ARIMAResults.arparams
static ARIMAResults.arparams()

statsmodels.tsa.arima_model.ARIMAResults.arroots
static ARIMAResults.arroots()

statsmodels.tsa.arima_model.ARIMAResults.bic
static ARIMAResults.bic()

statsmodels.tsa.arima_model.ARIMAResults.bse
static ARIMAResults.bse()

statsmodels.tsa.arima_model.ARIMAResults.conf_int
ARIMAResults.conf_int(alpha=0.05, cols=None, method='default')
    Returns the confidence interval of the fitted parameters.
    
    Parameters
    
    alpha : float, optional
        The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

    cols : array-like, optional
        cols specifies which confidence intervals to return

    method : string
        Not Implemented Yet Method to estimate the confidence_interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “jfh” : “jac” : “boot-bse” “boot_quant” “profile”

    Returns
    
    conf_int : array
        Each row contains [lower, upper] confidence interval

    Notes

        The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.
Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[ 5496529.48322745, 1467987.78596704],
       [-177.02903529, 207.15277984],
       [-3.12506664, -0.91539297],
       [-1.5179487, 0.460309],
       [ 798.7875153, 2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[ 0.1115811, 0.03994274],
       [-3.12506664, -0.91539297]])
```

```python
statsmodels.tsa.arima_model.ARIMAResults.cov_params
ARIMAResults.cov_params()
```

```python
statsmodels.tsa.arima_model.ARIMAResults.f_test
ARIMAResults.f_test(r_matrix=None, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
```

Compute the F-test for a joint linear hypothesis.

**Parameters**

- **r_matrix** : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like
  - This is deprecated. See r_matrix and the examples for more information on new usage.
  - Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p** : array-like, optional
  - An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale** : float, optional
  - Default is 1.0 for no scaling.

- **invcov** : array-like, optional
  - A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

**See also:**

- statsmodels.contrasts
- statsmodels.model.LikelihoodModelResults.wald_test
- statsmodels.model.LikelihoodModelResults.t_test
- patsy.DesignInfo.linear_constraint
Notes

The matrix \( r_{matrix} \) is assumed to be non-singular. More precisely, \( r_{matrix} (pX pX.T) r_{matrix}.T \) is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm

>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)

>>> results = sm.OLS(data.endog, data.exog).fit()

>>> A = np.identity(len(results.params))
>>> A = A[1:, :]

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>
```

Compare this to

```python
>>> results.F
330.2853392346658

>>> results.F_p
4.98403096572e-10
```

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.

```python
>>> B = np.array(([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'

>>> from statsmodels.formula.api import ols

>>> dta = longley.load_pandas().data

>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'

>>> results = ols(formula, dta).fit()

>>> f_test = results.f_test(hypotheses)

>>> print(f_test)
```

```python
< F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>
```

```python
>>> statsmodels.tsa.arima_model.ARIMAResults.fittedvalues

statsmodels.tsa.arima_model.ARIMAResults.fittedvalues()

static ARIMAResults.fittedvalues()
```
ARIMAResults.forecast(steps=1, exog=None, alpha=0.05)

Out-of-sample forecasts

**Parameters**

- **steps**: int
  The number of out-of-sample forecasts from the end of the sample.

- **exog**: array
  If the model is an ARIMAX, you must provide out-of-sample values for the exogenous variables. This should not include the constant.

- **alpha**: float
  The confidence intervals for the forecasts are (1 - alpha) %

**Returns**

- **forecast**: array
  Array of out of sample forecasts

- **stderr**: array
  Array of the standard error of the forecasts.

- **conf_int**: array
  2d array of the confidence interval for the forecast

**Notes**

Prediction is done in the levels of the original endogenous variable. If you would like prediction of differences in levels use `predict`.
ARIMA model in-sample and out-of-sample prediction

Parameters

**start**: int, str, or datetime

Zero-indexed observation number at which to start forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type.

**end**: int, str, or datetime

Zero-indexed observation number at which to end forecasting, i.e., the first forecast is start. Can also be a date string to parse or a datetime type. However, if the dates index does not have a fixed frequency, end must be an integer index if you want out of sample prediction.

**exog**: array-like, optional

If the model is an ARMAX and out-of-sample forecasting is requested, exog must be given. Note that you’ll need to pass $k_{ar}$ additional lags for any exogenous variables. E.g., if you fit an ARMAX(2, q) model and want to predict 5 steps, you need 7 observations to do this.

**dynamic**: bool, optional

The *dynamic* keyword affects in-sample prediction. If dynamic is False, then the in-sample lagged values are used for prediction. If dynamic is True, then in-sample forecasts are used in place of lagged dependent variables. The first forecasted value is *start*.

**typ**: str (‘linear’, ‘levels’)

- *linear*: Linear prediction in terms of the differenced endogenous variables.
- *levels*: Predict the levels of the original endogenous variables.

Returns **predict**: array

The predicted values.

Notes

It is recommended to use dates with the time-series models, as the below will probably make clear. However, if ARIMA is used without dates and/or *start* and *end* are given as indices, then these indices are in terms of the *original*, undifferenced series. I.e., given some undifferenced observations:

```
1970Q1, 1
1970Q2, 1.5
1970Q3, 1.25
1970Q4, 2.25
```
1971Q1, 1.2
1971Q2, 4.1

1970Q1 is observation 0 in the original series. However, if we fit an ARIMA(p,1,q) model then we lose this first observation through differencing. Therefore, the first observation we can forecast (if using exact MLE) is index 1. In the differenced series this is index 0, but we refer to it as 1 from the original series.

```python
statsmodels.tsa.arima_model.ARIMAResults.pvalues
static ARIMAResults.pvalues()
```

```python
statsmodels.tsa.arima_model.ARIMAResults.remove_data
ARIMAResults.remove_data()
```

remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

```python
statsmodels.tsa.arima_model.ARIMAResults.resid
static ARIMAResults.resid()
```

```python
statsmodels.tsa.arima_model.ARIMAResults.save
ARIMAResults.save(fname, remove_data=False)
```

save a pickle of this instance

**Parameters**

- **fname**: string or filehandle
  
  fname can be a string to a file path or filename, or a filehandle.

- **remove_data**: bool
  
  If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.

**Notes**

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.
statsmodels.tsa.arima_model.ARIMAResults.summary

ARIMAResults.summary(alpha=0.05)
Summarize the Model

Parameters alpha : float, optional
Significance level for the confidence intervals.

Returns smry : Summary instance
This holds the summary table and text, which can be printed or converted to various output formats.

See also:
statsmodels.iolib.summary.Summary

statsmodels.tsa.arima_model.ARIMAResults.summary2

ARIMAResults.summary2(title=None, alpha=0.05, float_format='%.4f')
Experimental summary function for ARIMA Results

Parameters title : string, optional
Title for the top table. If not None, then this replaces the default title

alpha : float
significance level for the confidence intervals

float_format : string :
print format for floats in parameters summary

Returns smry : Summary instance
This holds the summary table and text, which can be printed or converted to various output formats.

See also:
statsmodels.iolib.summary2.Summary class to hold summary results

statsmodels.tsa.arima_model.ARIMAResults.t_test

ARIMAResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)
Compute a t-test for a joint linear hypothesis of the form Rb = q

Parameters r_matrix : array-like, str, tuple
- array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional
This is deprecated. See r_matrix and the examples for more information on new usage.
Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional
An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional
If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics

f_test for F tests

patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1, -1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```
r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_Test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
>>> T_test.effect -1829.2025687192481 >>> T_test.sd 455.39079425193762 >>> T_test.tvalue -4.0167754636411717 >>> T_test.pvalue 0.0015163772380899498
```
Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = ‘TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR’
>>> results = ols(formula, dta).fit()
>>> hypotheses = ‘GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1’
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

```
```

```
```
Parameters  

- `r_matrix` : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since `q_matrix` is deprecated.

- `q_matrix` : array-like
  This is deprecated. See `r_matrix` and the examples for more information on new usage.
  Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array,
  `q_matrix` is assumed to be a conformable array of zeros.

- `cov_p` : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given,
  self.normalized_cov_params is used.

- `scale` : float, optional
  Default is 1.0 for no scaling.

- `invcov` : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

- `use_f` : bool
  If True, then the F-distribution is used. If False, then the asymptotic distribution,
  chi-square is used. The test statistic is proportionally adjusted for the distribution by
  the number of constraints in the hypothesis.

See also:

- statsmodels.contrasts
- statsmodels.model.LikelihoodModelResults.f_test
- statsmodels.model.LikelihoodModelResults.t_test
- patsy.DesignInfo.linear_constraint

Notes

The matrix `r_matrix` is assumed to be non-singular. More precisely,

\[
\text{r_matrix} \cdot (pX \cdot pX^T) \cdot \text{r_matrix}^T
\]

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

```
statsmodels.tsa.kalmanf.kalmanfilter.KalmanFilter
```

- **class** `statsmodels.tsa.kalmanf.kalmanfilter.KalmanFilter`
  Kalman Filter code intended for use with the ARMA model.

Notes

The notation for the state-space form follows Durbin and Koopman (2001).
The observation equations is

\[ y_t = Z_t \alpha_t + \epsilon_t \]

The state equation is

\[ \alpha_{t+1} = T_t \alpha_t + R_t \eta_t \]

For the present purposed \[ \epsilon_t \] is assumed to always be zero.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>R</code></td>
<td>The coefficient matrix for the state vector in the observation equation.</td>
</tr>
<tr>
<td><code>T</code></td>
<td>The coefficient matrix for the state vector in the state equation.</td>
</tr>
<tr>
<td><code>Z</code></td>
<td>Returns the Z selector matrix in the observation equation.</td>
</tr>
<tr>
<td><code>geterrors</code></td>
<td>Returns just the errors of the Kalman Filter</td>
</tr>
<tr>
<td><code>loglike</code></td>
<td>The loglikelihood for an ARMA model using the Kalman Filter recursions.</td>
</tr>
</tbody>
</table>

**classmethod** `KalmanFilter.R`  
`R(params, r, k, q, p)`  
The coefficient matrix for the state vector in the observation equation.  
Its dimension is \( r+k \times 1 \).  

**Parameters**  
\( r : \text{int} \)  
In the context of the ARMA model \( r \) is \( \max(p,q+1) \) where \( p \) is the AR order and \( q \) is the MA order.  
\( k : \text{int} \)  
The number of exogenous variables in the ARMA model, including the constant if appropriate.  
\( q : \text{int} \)  
The MA order in an ARMA model.  
\( p : \text{int} \)  
The AR order in an ARMA model.

**References**

Durbin and Koopman Section 3.7.

**classmethod** `KalmanFilter.T`  
`T(params, r, k, p)`  
The coefficient matrix for the state vector in the state equation.  
Its dimension is \( r+k \times r+k \).  

**Parameters**  
\( r : \text{int} \)
In the context of the ARMA model $r$ is $\max(p, q+1)$ where $p$ is the AR order and $q$ is the MA order.

$k$ : int
The number of exogenous variables in the ARMA model, including the constant if appropriate.

$p$ : int
The AR coefficient in an ARMA model.

**References**

Durbin and Koopman Section 3.7.

```python
statsmodels.tsa.kalmanf.kalmanfilter.KalmanFilter.Z
```

**class method** KalmanFilter.Z($r$)

Returns the $Z$ selector matrix in the observation equation.

**Parameters** $r$ : int

In the context of the ARMA model $r$ is $\max(p, q+1)$ where $p$ is the AR order and $q$ is the MA order.

**Notes**

Currently only returns a $1 \times r$ vector $[1, 0, 0, \ldots, 0]$. Will need to be generalized when the Kalman Filter becomes more flexible.

```python
statsmodels.tsa.kalmanf.kalmanfilter.KalmanFilter.geterrors
```

**class method** KalmanFilter.geterrors($y, k, k_ar, k_ma, k_lags, nobs, Z_mat, m, R_mat, T_mat, paramsdtype$)

Returns just the errors of the Kalman Filter

```python
statsmodels.tsa.kalmanf.kalmanfilter.KalmanFilter.loglike
```

**class method** KalmanFilter.loglike($params, arma_model, set_sigma2=True$)

The loglikelihood for an ARMA model using the Kalman Filter recursions.

**Parameters** params : array
The coefficients of the ARMA model, assumed to be in the order of trend variables and $k$ exogenous coefficients, the $p$ AR coefficients, then the $q$ MA coefficients.

arma_model : statsmodels.tsa.arima.ARMA instance
A reference to the ARMA model instance.

set_sigma2 : bool, optional
True if arma_model.sigma2 should be set. Note that sigma2 will be computed in any case, but it will be discarded if set_sigma2 is False.
Notes

This works for both real valued and complex valued parameters. The complex values being used to compute the numerical derivative. If available will use a Cython version of the Kalman Filter.

Vector Autogressive Processes (VAR)

vector_ar.var_model.VAR(endog[, dates, ...]) Fit VAR(p) process and do lag order selection
vector_ar.var_model.VARResults(endog, ...[, ...]) Estimate VAR(p) process with fixed number of lags
vector_ar.dynamic.DynamicVAR(data[, ...]) Estimates time-varying vector autoregression (VAR(p)) using

statsmodels.tsa.vector_ar.var_model.VAR

class statsmodels.tsa.vector_ar.var_model.VAR (endog, dates=None, names=None, freq=None, missing='none')

Fit VAR(p) process and do lag order selection

\[ y_t = A_1 y_{t-1} + \ldots + A_p y_{t-p} + u_t \]

Parameters

endog : array-like

2-d endogenous response variable. The independent variable.

names : array-like

must match number of columns of endog

dates : array-like

must match number of rows of endog

References

Lutkepohl (2005) New Introduction to Multiple Time Series Analysis

Methods

fit([maxlags, method, ic, trend, verbose]) Fit the VAR model
from_formula(formula, data[, subset]) Create a Model from a formula and dataframe.
hessian(params) The Hessian matrix of the model
information(params) Fisher information matrix of model
initialize() Initialize (possibly re-initialize) a Model instance. For
loglike(params) Log-likelihood of model.
predict(params[, start, end, lags, trend]) Returns in-sample predictions or forecasts
score(params) Score vector of model.
select_order([maxlags, verbose]) Compute lag order selections based on each of the available information

statsmodels.tsa.vector_ar.var_model.VAR.fit

VAR.fit (maxlags=None, method='ols', ic=None, trend='c', verbose=False) Fit the VAR model
Parameters `maxlags` : int

Maximum number of lags to check for order selection, defaults to 12 * (nobs/100.)**(1./4), see select_order function

`method` : {'ols'}

Estimation method to use

`ic` : {'aic', 'fpe', 'hqic', 'bic', None}

Information criterion to use for VAR order selection. aic : Akaike fpe : Final prediction error hqic : Hannan-Quinn bic : Bayesian a.k.a. Schwarz

`verbose` : bool, default False

Print order selection output to the screen

`trend`, str ("c", "ct", "ctt", "nc") :

"c" - add constant "ct" - constant and trend "ctt" - constant, linear and quadratic trend "nc" - no constant, no trend Note that these are prepended to the columns of the dataset.

Returns `est` : VARResults

Notes

Lutkepohl pp. 146-153

statsmodels.tsa.vector_ar.var_model.VAR.from_formula
classmethod VAR.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

Parameters `formula` : str or generic Formula object

The formula specifying the model

`data` : array-like

The data for the model. See Notes.

`subset` : array-like

An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

`args` : extra arguments

These are passed to the model

`kwargs` : extra keyword arguments

These are passed to the model.

Returns `model` : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.
statsmodels.tsa.vector_ar.var_model.VAR.hessian

VAR.hessian(params)
The Hessian matrix of the model

statsmodels.tsa.vector_ar.var_model.VAR.information

VAR.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

statsmodels.tsa.vector_ar.var_model.VAR.initialize

VAR.initialize()
Initialize (possibly re-initialize) a Model instance. For instance, the design matrix of a linear model may change and some things must be recomputed.

statsmodels.tsa.vector_ar.var_model.VAR.loglike

VAR.loglike(params)
Log-likelihood of model.

statsmodels.tsa.vector_ar.var_model.VAR.predict

VAR.predict(params, start=None, end=None, lags=1, trend='c')
Returns in-sample predictions or forecasts

statsmodels.tsa.vector_ar.var_model.VAR.score

VAR.score(params)
Score vector of model.

The gradient of logL with respect to each parameter.

statsmodels.tsa.vector_ar.var_model.VAR.select_order

VAR.select_order(maxlags=None, verbose=True)
Compute lag order selections based on each of the available information criteria

Parameters maxlags : int
    if None, defaults to 12 * (nobs/100.)**(1/4)

verbose : bool, default True
    If True, print table of info criteria and selected orders

Returns selections : dict {info_crit -> selected_order}

Attributes

endog_names
exog_names
class statsmodels.tsa.vector_ar.var_model.VARResults(endog, endog_lagged, params, sigma_u, lag_order, model=None, trend='c', names=None, dates=None)

Estimate VAR(p) process with fixed number of lags

Parameters

- **endog**: array
- **endog_lagged**: array
- **params**: array
- **sigma_u**: array
- **lag_order**: int
- **model**: VAR model instance
- **trend**: str {'nc', 'c', 'ct'}
- **names**: array-like

List of names of the endogenous variables in order of appearance in endog.

Returns

**Attributes**:

- **aic**
- **bic**
- **bse**
- **coefs**: ndarray (p x K x K)
  
  Estimated A_i matrices, A_i = coefs[i-1]
- **cov_params**
- **dates**
- **detomega**
- **df_model**: int
- **df_resid**: int
- **endog**
- **endog_lagged**
- **fittedvalues**
- **fpe**
- **intercept**
- **info_criteria**
- **k_ar**: int
- **k_trend**: int
- **llf**
- **model**

3.7. Time Series analysis tsa
names:
define neqs : int
    Number of variables (equations)
define nobs : int
define n_totobs : int
define params :
define k_ar : int
    Order of VAR process
define params : ndarray (Kp + 1) x K
    A_i matrices and intercept in stacked form [int A_1 ... A_p]
define pvalues :
define names : list
    variables names
define resid :
define roots : array
    The roots of the VAR process are the solution to (I - coefs[0]*z - coefs[1]*z**2 ... -
    coefs[p-1]*z**k_ar) = 0. Note that the inverse roots are returned, and stability requires
    that the roots lie outside the unit circle.
define sigma_u : ndarray (K x K)
    Estimate of white noise process variance Var[u_t]
define sigma_u_mle :
define stderr :
define trenorder :
define tvalues :
define y :
define ys_lagged :

Methods

acf([nlags]) Compute theoretical autocovariance function
acorr([nlags]) Compute theoretical autocorrelation function
bse() Standard errors of coefficients, reshaped to match in size
coef_names() Coefficient names (deprecated)
cov_params() Estimated variance-covariance of model coefficients..
cov_ybar() Asymptotically consistent estimate of covariance of the sample mean..
detomega() Return determinant of white noise covariance with degrees of freedom correction
fevd([periods, var_decomp]) Compute forecast error variance decomposition (“fevd”)
fittedvalues() The predicted insample values of the response variables of the model.
forecast(y, steps) Produce linear minimum MSE forecasts for desired number of steps
forecast_cov([steps]) Compute forecast covariance matrices for desired number of steps
Table 3.156 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>forecast_interval(y, steps[, alpha])</code></td>
<td>Construct forecast interval estimates assuming the y are Gaussian</td>
</tr>
<tr>
<td><code>get_eq_index(name)</code></td>
<td>Return integer position of requested equation name</td>
</tr>
<tr>
<td><code>info_criteria()</code></td>
<td>Information criteria for lagorder selection</td>
</tr>
<tr>
<td><code>irf([periods, var_decomp, var_order])</code></td>
<td>Analyze impulse responses to shocks in system</td>
</tr>
<tr>
<td><code>irf_errband_mc([orth, repl, T, signif, ...])</code></td>
<td>Compute Monte Carlo integrated error bands assuming normally</td>
</tr>
<tr>
<td><code>irf_resim([orth, repl, T, seed, burn, cum])</code></td>
<td>Simulates impulse response function, returning an array of simulations.</td>
</tr>
<tr>
<td><code>is_stable([verbose])</code></td>
<td>Determine stability based on model coefficients</td>
</tr>
<tr>
<td><code>llf()</code></td>
<td>Compute VAR(p) loglikelihood</td>
</tr>
<tr>
<td><code>long_run_effects()</code></td>
<td>Compute long-run effect of unit impulse..</td>
</tr>
<tr>
<td><code>ma_rep([maxn])</code></td>
<td>Compute MA(∞) coefficient matrices</td>
</tr>
<tr>
<td><code>mean()</code></td>
<td>Mean of stable process</td>
</tr>
<tr>
<td><code>mse(steps)</code></td>
<td>Compute theoretical forecast error variance matrices</td>
</tr>
<tr>
<td><code>orth_ma_rep([maxn, P])</code></td>
<td>Compute orthogonalized MA coefficient matrices using P matrix such</td>
</tr>
<tr>
<td><code>plot()</code></td>
<td>Plot input time series</td>
</tr>
<tr>
<td><code>plot_acorr([nlags, linewidth])</code></td>
<td>Plot theoretical autocorrelation function</td>
</tr>
<tr>
<td><code>plot_forecast(steps[, alpha, plot_stderr])</code></td>
<td>Plot forecast</td>
</tr>
<tr>
<td><code>plot_sample_acorr([nlags, linewidth])</code></td>
<td>Plot theoretical autocorrelation function</td>
</tr>
<tr>
<td><code>plotsim([steps])</code></td>
<td>Plot a simulation from the VAR(p) process for the desired number of</td>
</tr>
<tr>
<td><code>pvalues()</code></td>
<td>Two-sided p-values for model coefficients from Student t-distribution</td>
</tr>
<tr>
<td><code>reorder(order)</code></td>
<td>Reorder variables for structural specification</td>
</tr>
<tr>
<td><code>resid()</code></td>
<td>Residuals of response variable resulting from estimated coefficients</td>
</tr>
<tr>
<td><code>resid_acorr([nlags])</code></td>
<td>Compute sample autocorrelation (including lag 0)</td>
</tr>
<tr>
<td><code>resid_acov([nlags])</code></td>
<td>Compute centered sample autocovariance (including lag 0)</td>
</tr>
<tr>
<td><code>resid_corr()</code></td>
<td>Centered residual correlation matrix</td>
</tr>
<tr>
<td><code>roots()</code></td>
<td>(Biased) maximum likelihood estimate of noise process covariance</td>
</tr>
<tr>
<td><code>sample_acorr([nlags])</code></td>
<td>Standard errors of coefficients, reshaped to match in size</td>
</tr>
<tr>
<td><code>sample_acov([nlags])</code></td>
<td>Compute console output summary of estimates</td>
</tr>
<tr>
<td><code>sigma_u_mle()</code></td>
<td>Compute test statistic for null hypothesis of Granger-noncausality,</td>
</tr>
<tr>
<td><code>stderr()</code></td>
<td>Test assumption of normal-distributed errors using Jarque-Bera-style</td>
</tr>
<tr>
<td><code>summary()</code></td>
<td>Compute white noise assumption.</td>
</tr>
<tr>
<td><code>test_causality(equation, variables[, kind, ...])</code></td>
<td>Compute t-statistics.</td>
</tr>
<tr>
<td><code>test_normality([signif, verbose])</code></td>
<td>Test white noise assumption.</td>
</tr>
<tr>
<td><code>test_whiteness([nlags, plot, linewidth])</code></td>
<td>Compute t-statistics.</td>
</tr>
</tbody>
</table>

### statsmodels.tsa.vector_ar.var_model.VARResults.acf

```python
VARResults.acf(nlags=None)
```

Compute theoretical autocovariance function

**Returns** `acf` : ndarray (p x k x k)

### statsmodels.tsa.vector_ar.var_model.VARResults.acorr

```python
VARResults.acorr(nlags=None)
```

Compute theoretical autocorrelation function

**Returns** `acorr` : ndarray (p x k x k)

### statsmodels.tsa.vector_ar.var_model.VARResults.bse

```python
static VARResults.bse()
```

Standard errors of coefficients, reshaped to match in size

### 3.7. Time Series analysis tsa 579
statsmodels.tsa.vector_ar.var_model.VARResults.coef_names

```python
static VARResults.coef_names()
```

Coefficient names (deprecated)

statsmodels.tsa.vector_ar.var_model.VARResults.cov_params

```python
static VARResults.cov_params()
```

Estimated variance-covariance of model coefficients

Notes

Covariance of $\text{vec}(B)$, where $B$ is the matrix $[\text{intercept}, A_1, \ldots, A_p]$ ($K \times (K p + 1)$) Adjusted to be an unbiased estimator Ref: Lutkepohl p.74-75

statsmodels.tsa.vector_ar.var_model.VARResults.cov_ybar

```python
VARResults.cov_ybar()
```

Asymptotically consistent estimate of covariance of the sample mean

$$\sqrt{T}(\bar{y} - \mu) \rightarrow N(0, \Sigma_{\bar{y}})$$

$$\Sigma_{\bar{y}} = B \Sigma_u B'$$, where $B = (I_K - A_1 - \cdots - A_p)^{-1}$

Notes

Lutkepohl Proposition 3.3

statsmodels.tsa.vector_ar.var_model.VARResults.detomega

```python
static VARResults.detomega()
```

Return determinant of white noise covariance with degrees of freedom correction:

$$\hat{\Omega} = \frac{T}{T - Kp - 1} \hat{\Omega}_{MLE}$$

statsmodels.tsa.vector_ar.var_model.VARResults.fevd

```python
VARResults.fevd(periods=10, var_decomp=None)
```

Compute forecast error variance decomposition (“fevd”)

Returns fevd : FEVD instance

statsmodels.tsa.vector_ar.var_model.VARResults.fittedvalues

```python
static VARResults.fittedvalues()
```

The predicted in-sample values of the response variables of the model.

statsmodels.tsa.vector_ar.var_model.VARResults.forecast

```python
VARResults.forecast(y, steps)
```

Produce linear minimum MSE forecasts for desired number of steps ahead, using prior values $y$

Parameters

- y : ndarray ($p \times k$)
- steps : int

Returns forecasts : ndarray ($steps \times neqs$)
Notes

Lutkepohl pp 37-38

statsmodels.tsa.vector_ar.var_model.VARResults.forecast_cov

VARResults.forecast_cov(steps=1)

Compute forecast covariance matrices for desired number of steps

Parameters steps : int

Returns cova : ndarray (steps x k x k)

Notes

$$\Sigma_y(h) = \Sigma_y(h) + \Omega(h)/T$$

Ref: Lutkepohl pp. 96-97

statsmodels.tsa.vector_ar.var_model.VARResults.forecast_interval

VARResults.forecast_interval(y, steps, alpha=0.05)

Construct forecast interval estimates assuming the y are Gaussian

Returns (lower, mid, upper) : (ndarray, ndarray, ndarray)

Notes

Lutkepohl pp. 39-40

statsmodels.tsa.vector_ar.var_model.VARResults.get_eq_index

VARResults.get_eq_index(name)

Return integer position of requested equation name

statsmodels.tsa.vector_ar.var_model.VARResults.info_criteria

static VARResults.info_criteria()

information criteria for lagorder selection

statsmodels.tsa.vector_ar.var_model.VARResults.irf

VARResults.irf(periods=10, var_decomp=None, var_order=None)

Analyze impulse responses to shocks in system

Parameters periods : int

var_decomp : ndarray (k x k), lower triangular

Must satisfy Omega = P P', where P is the passed matrix. Defaults to Cholesky decomposition of Omega

var_order : sequence

Alternate variable order for Cholesky decomposition

3.7. Time Series analysis tsa
Returns irf : IRAnalysis

statsmodels.tsa.vector_ar.var_model.VARResults.irf_errband_mc

VARResults.irf_errband_mc(orth=False, repl=1000, T=10, signif=0.05, seed=None, burn=100, cum=False)

Compute Monte Carlo integrated error bands assuming normally distributed for impulse response functions

Parameters orth: bool, default False :

Compute orthogonalized impulse response error bands

repl: int :

number of Monte Carlo replications to perform

T: int, default 10 :

number of impulse response periods

signif: float (0 < signif <1) :

Significance level for error bars, defaults to 95% CI

seed: int :

np.random.seed for replications

burn: int :

number of initial observations to discard for simulation

cum: bool, default False :

produce cumulative irf error bands

Returns Tuple of lower and upper arrays of ma_rep monte carlo standard errors :

Notes

Lutkepohl (2005) Appendix D

statsmodels.tsa.vector_ar.var_model.VARResults.irf_resim

VARResults.irf_resim(orth=False, repl=1000, T=10, signif=0.05, seed=None, burn=100, cum=False)

Simulates impulse response function, returning an array of simulations. Used for Sims-Zha error band calculation.

Parameters orth: bool, default False :

Compute orthogonalized impulse response error bands

repl: int :

number of Monte Carlo replications to perform

T: int, default 10 :

number of impulse response periods

signif: float (0 < signif <1) :

Significance level for error bars, defaults to 95% CI

seed: int :
np.random.seed for replications

**burn**: int :
number of initial observations to discard for simulation

**cum**: bool, default False :
produce cumulative irf error bands

**Returns** Array of simulated impulse response functions :

**Notes**


```python
statsmodels.tsa.vector_ar.var_model.VARResults.is_stable
VARResults.is_stable(verbos=True)
```

Determine stability based on model coefficients

**Parameters** verbose : bool

Print eigenvalues of the VAR(1) companion

**Notes**

Checks if det(I - Az) = 0 for any mod(z) <= 1, so all the eigenvalues of the companion matrix must lie outside the unit circle

```python
statsmodels.tsa.vector_ar.var_model.VARResults.llf
static VARResults.llf()
```

Compute VAR(p) loglikelihood

```python
statsmodels.tsa.vector_ar.var_model.VARResults.long_run_effects
VARResults.long_run_effects()
```

Compute long-run effect of unit impulse

\[
\Psi_{\infty} = \sum_{i=0}^{\infty} \Phi_i
\]

```python
statsmodels.tsa.vector_ar.var_model.VARResults.ma_rep
VARResults.ma_rep(maxn=10)
```

Compute MA(\(\infty\)) coefficient matrices

**Parameters** maxn : int

Number of coefficient matrices to compute

**Returns** coefs : ndarray (maxn x k x k)
\texttt{statsmodels.tsa.vector_ar.var_model.VARResults.mean}

\texttt{VARResults.mean()}

\begin{itemize}
  \item Mean of stable process
  \item Lutkepohl eq. 2.1.23
  \end{itemize}

\[
\mu = (I - A_1 - \cdots - A_p)^{-1} \alpha
\]

\texttt{statsmodels.tsa.vector_ar.var_model.VARResults.mse}

\texttt{VARResults.mse(steps)}

\begin{itemize}
  \item Compute theoretical forecast error variance matrices
  \item Parameters \texttt{steps}: int
  \item Number of steps ahead
  \item \textbf{Returns} \texttt{force_covs}: ndarray (steps x neqs x neqs)
  \end{itemize}

\textbf{Notes}

\[
\text{MSE}(h) = \sum_{i=0}^{h-1} \Phi \Sigma_u \Phi^T
\]

\texttt{statsmodels.tsa.vector_ar.var_model.VARResults.orth_ma_rep}

\texttt{VARResults.orth_ma_rep(maxn=10, P=None)}

\begin{itemize}
  \item Compute Orthogonalized MA coefficient matrices using P matrix such that \( \Sigma_u = PP' \). P defaults to the Cholesky decomposition of \( \Sigma_u \)
  \item Parameters \texttt{maxn}: int
  \item Number of coefficient matrices to compute
  \item \texttt{P}: ndarray (k x k), optional
  \item Matrix such that Sigma_u = PP', defaults to Cholesky descomp
  \item \textbf{Returns} \texttt{coefs}: ndarray (maxn x k x k)
  \end{itemize}

\texttt{statsmodels.tsa.vector_ar.var_model.VARResults.plot}

\texttt{VARResults.plot()}

\begin{itemize}
  \item Plot input time series
  \end{itemize}

\texttt{statsmodels.tsa.vector_ar.var_model.VARResults.plot_acorr}

\texttt{VARResults.plot_acorr(nlags=10, linewidth=8)}

\begin{itemize}
  \item Plot theoretical autocorrelation function
  \end{itemize}

\texttt{statsmodels.tsa.vector_ar.var_model.VARResults.plot_forecast}

\texttt{VARResults.plot_forecast(steps, alpha=0.05, plot_stderr=True)}

\begin{itemize}
  \item Plot forecast
  \end{itemize}
statsmodels.tsa.vector_ar.var_model.VARResults.plot_sample_acorr
VARResults.plot_sample_acorr(nlags=10, linewidth=8)
  Plot theoretical autocorrelation function

statsmodels.tsa.vector_ar.var_model.VARResults.plotsim
VARResults.plotsim(steps=1000)
  Plot a simulation from the VAR(p) process for the desired number of steps

statsmodels.tsa.vector_ar.var_model.VARResults.pvalues
static VARResults.pvalues()
  Two-sided p-values for model coefficients from Student t-distribution

statsmodels.tsa.vector_ar.var_model.VARResults.reorder
VARResults.reorder(order)
  Reorder variables for structural specification

statsmodels.tsa.vector_ar.var_model.VARResults.resid
static VARResults.resid()
  Residuals of response variable resulting from estimated coefficients

statsmodels.tsa.vector_ar.var_model.VARResults.resid_acorr
VARResults.resid_acorr(nlags=1)
  Compute sample autocorrelation (including lag 0)

  Parameters
    nlags : int

statsmodels.tsa.vector_ar.var_model.VARResults.resid_acov
VARResults.resid_acov(nlags=1)
  Compute centered sample autocovariance (including lag 0)

  Parameters
    nlags : int

statsmodels.tsa.vector_ar.var_model.VARResults.resid_corr
static VARResults.resid_corr()
  Centered residual correlation matrix

statsmodels.tsa.vector_ar.var_model.VARResults.roots
static VARResults.roots()

statsmodels.tsa.vector_ar.var_model.VARResults.sample_acorr
VARResults.sample_acorr(nlags=1)

statsmodels.tsa.vector_ar.var_model.VARResults.sample_acov
VARResults.sample_acov(nlags=1)

statsmodels.tsa.vector_ar.var_model.VARResults.sigma_u_mle
static VARResults.sigma_u_mle()
  (Biased) maximum likelihood estimate of noise process covariance
statsmodels.tsa.vector_ar.var_model.VARResults.stderr

```
static VARResults.stderr()
```

Standard errors of coefficients, reshaped to match in size

statsmodels.tsa.vector_ar.var_model.VARResults.summary

```
VARResults.summary()
```

Compute console output summary of estimates

**Returns** summary : VARSummary

statsmodels.tsa.vector_ar.var_model.VARResults.test_causality

```
VARResults.test_causality(equation, variables, kind='f', signif=0.05, verbose=True)
```

Compute test statistic for null hypothesis of Granger-noncausality, general function to test joint Granger-causality of multiple variables

**Parameters**

- **equation** : string or int
  
  Equation to test for causality

- **variables** : sequence (of strings or ints)
  
  List, tuple, etc. of variables to test for Granger-causality

- **kind** : {'f', 'wald'}
  
  Perform F-test or Wald (chi-sq) test

- **signif** : float, default 5%
  
  Significance level for computing critical values for test, defaulting to standard 0.95 level

**Returns** results : dict

**Notes**

Null hypothesis is that there is no Granger-causality for the indicated variables. The degrees of freedom in the F-test are based on the number of variables in the VAR system, that is, degrees of freedom are equal to the number of equations in the VAR times degree of freedom of a single equation.

statsmodels.tsa.vector_ar.var_model.VARResults.test_normality

```
VARResults.test_normality(signif=0.05, verbose=True)
```

Test assumption of normal-distributed errors using Jarque-Bera-style omnibus Chi^2 test

**Parameters** signif : float

Test significance threshold

**Notes**

H0 (null) : data are generated by a Gaussian-distributed process
```
statsmodels.tsa.vector_ar.var_model.VARResults.test_whiteness

VARResults.test_whiteness(nlags=10, plot=True, linewidth=8)
Test white noise assumption. Sample (Y) autocorrelations are compared with the standard $2/\sqrt{T}$ bounds.

Parameters
plot : boolean, default True
Plot autocorrelations with $2 / \sqrt{T}$ bounds

statsmodels.tsa.vector_ar.var_model.VARResults.tvalues

static VARResults.tvalues()
Compute t-statistics. Use Student-t($T - Kp - 1$) = $t(df_{resid})$ to test significance.

Attributes

aic         Akaike information criterion
bic         Bayesian a.k.a.
df_model    Number of estimated parameters, including the intercept / trends
df_resid    Number of observations minus number of estimated parameters
fpe         Final Prediction Error (FPE)
hqic        Hannan-Quinn criterion
```

```
statsmodels.tsa.vector_ar.dynamic.DynamicVAR

class statsmodels.tsa.vector_ar.dynamic.DynamicVAR(data, lag_order=1, window=None,
    window_type='expanding',
    trend='c', min_periods=None)
Estimates time-varying vector autoregression (VAR(p)) using equation-by-equation least squares

Parameters

data : pandas.DataFrame
lag_order : int, default 1
window : int
window_type : {'expanding', 'rolling'}
min_periods : int or None
    Minimum number of observations to require in window, defaults to window size if None specified

Returns

Attributes:

coefs : WidePanel
    items : coefficient names major_axis : dates minor_axis : VAR equation names

Methods

T() Number of time periods in results
```
Table 3.158 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>coefs()</code></td>
<td>Return dynamic regression coefficients as WidePanel</td>
</tr>
<tr>
<td><code>equations()</code></td>
<td></td>
</tr>
<tr>
<td><code>forecast()</code></td>
<td>Produce dynamic forecast</td>
</tr>
<tr>
<td><code>plot_forecast([steps])</code></td>
<td>Plot h-step ahead forecasts against actual realizations of time series. Returns the r-squared values.</td>
</tr>
<tr>
<td><code>r2()</code></td>
<td></td>
</tr>
<tr>
<td><code>resid()</code></td>
<td></td>
</tr>
</tbody>
</table>

statsmodels.tsa.vector_ar.dynamic.DynamiCVar.T

**static** DynamicVAR.T()
Number of time periods in results

statsmodels.tsa.vector_ar.dynamic.DynamiCVar.coefs

**static** DynamicVAR.coefs()
Return dynamic regression coefficients as WidePanel

statsmodels.tsa.vector_ar.dynamic.DynamiCVar.equations

**static** DynamicVAR.equations()

statsmodels.tsa.vector_ar.dynamic.DynamiCVar.forecast

DynamicVAR.forecast(steps=1)
Produce dynamic forecast

**Parameters** steps :

**Returns** forecasts : pandas.DataFrame

statsmodels.tsa.vector_ar.dynamic.DynamiCVar.plot_forecast

DynamicVAR.plot_forecast(steps=1, figsize=(10, 10))
Plot h-step ahead forecasts against actual realizations of time series. Note that forecasts are lined up with their respective realizations.

**Parameters** steps :

statsmodels.tsa.vector_ar.dynamic.DynamiCVar.r2

**static** DynamicVAR.r2()
Returns the r-squared values.

statsmodels.tsa.vector_ar.dynamic.DynamiCVar.resid

**static** DynamicVAR.resid()

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>nob</td>
</tr>
<tr>
<td>result_index</td>
</tr>
</tbody>
</table>

See also:
3.7.3 Vector Autogressive Processes (VAR)

Besides estimation, several process properties and additional results after estimation are available for vector autoregressive processes.

$$y_t = A_1 y_{t-1} + \ldots + A_p y_{t-p} + u_t$$

**vector_ar.var_model.VAR(endog[, dates, ...])**
Fit VAR(p) process and do lag order selection

**vector_ar.var_model.VARProcess(coefs, ...[, ...])**
Class represents a known VAR(p) process

**vector_ar.var_model.VARResults(endog, ...[, ...])**
Estimate VAR(p) process with fixed number of lags

**vector_ar.irf.IRAnalysis(model[, P, ...])**
Impulse response analysis class.

**vector_ar.var_model.FEVD(model[, P, periods])**
Compute and plot Forecast error variance decomposition and asymptotic

**vector_ar.dynamic.DynamicVAR(data[, ...])**
Estimates time-varying vector autoregression (VAR(p)) using

**References**
Lutkepohl (2005) New Introduction to Multiple Time Series Analysis

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit(maxlags, method, ic, trend, verbose)</td>
<td>Fit the VAR model</td>
</tr>
<tr>
<td>from_formula(formula, data[, subset])</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>hessian(params)</td>
<td>The Hessian matrix of the model</td>
</tr>
<tr>
<td>information(params)</td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td>initialize()</td>
<td>Initialize (possibly re-initialize) a Model instance. For</td>
</tr>
<tr>
<td>loglike(params)</td>
<td>Log-likelihood of model.</td>
</tr>
<tr>
<td>predict(params[, start, end, lags, trend])</td>
<td>Returns in-sample predictions or forecasts</td>
</tr>
<tr>
<td>score(params)</td>
<td>Score vector of model.</td>
</tr>
<tr>
<td>select_order(maxlags, verbose)</td>
<td>Compute lag order selections based on each of the available information</td>
</tr>
</tbody>
</table>

3.7. Time Series analysis tsa
VAR.fit (maxlags=None, method='ols', ic=None, trend='c', verbose=False)
Fit the VAR model

Parameters:
- **maxlags**: int
  - Maximum number of lags to check for order selection, defaults to 12 * (nobs/100.)**(1./4), see select_order function
- **method**: {'ols'}
  - Estimation method to use
- **ic**: {'aic', 'fpe', 'hqic', 'bic', None}
  - Information criterion to use for VAR order selection. aic : Akaike fpe : Final prediction error hqic : Hannan-Quinn bic : Bayesian a.k.a. Schwarz
- **verbose**: bool, default False
  - Print order selection output to the screen
- **trend**, str (“c”, “ct”, “ctt”, “nc”):
  - “c” - add constant “ct” - constant and trend “ctt” - constant, linear and quadratic trend “nc” - co constant, no trend Note that these are prepended to the columns of the dataset.

Returns:
- **est**: VARResults

Notes
Lutkepohl pp. 146-153

VAR.from_formula (formula, data, subset=None, *args, **kwargs)
Create a Model from a formula and dataframe.

Parameters:
- **formula**: str or generic Formula object
  - The formula specifying the model
- **data**: array-like
  - The data for the model. See Notes.
- **subset**: array-like
  - An array-like object of bools, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
- **args**: extra arguments
  - These are passed to the model
- **kwargs**: extra keyword arguments
  - These are passed to the model.

Returns:
- **model**: Model instance
Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

```
statsmodels.tsa.vector_ar.var_model.VAR.hessian

VAR.hessian(params)
   The Hessian matrix of the model
```

```
statsmodels.tsa.vector_ar.var_model.VAR.information

VAR.information(params)
   Fisher information matrix of model
   Returns -Hessian of loglike evaluated at params.
```

```
statsmodels.tsa.vector_ar.var_model.VAR.initialize

VAR.initialize()
   Initialize (possibly re-initialize) a Model instance. For instance, the design matrix of a linear model may change and some things must be recomputed.
```

```
statsmodels.tsa.vector_ar.var_model.VAR.loglike

VAR.loglike(params)
   Log-likelihood of model.
```

```
statsmodels.tsa.vector_ar.var_model.VAR.predict

VAR.predict(params, start=None, end=None, lags=1, trend='c')
   Returns in-sample predictions or forecasts
```

```
statsmodels.tsa.vector_ar.var_model.VAR.score

VAR.score(params)
   Score vector of model.
   The gradient of logL with respect to each parameter.
```

```
statsmodels.tsa.vector_ar.var_model.VAR.select_order

VAR.select_order(maxlags=None, verbose=True)
   Compute lag order selections based on each of the available information criteria
   Parameters maxlags : int
      if None, defaults to 12 * (nobs/100.)**(1./4)
   verbose : bool, default True
```
If True, print table of info criteria and selected orders

**Returns**

**selections**: dict [info_crit -> selected_order]

**Attributes**

- `endog_names`
- `exog_names`

**statsmodels.tsa.vector_ar.var_model.VARProcess**

### class `statsmodels.tsa.vector_ar.var_model.VARProcess(coefs, intercept, sigma_u, names=None)`

Class represents a known VAR(p) process

**Parameters**

- `coefs` : ndarray (p x k x k)
- `intercept` : ndarray (length k)
- `sigma_u` : ndarray (k x k)
- `names` : sequence (length k)

**Returns**

**Attributes**:

### Methods

- `acf([nlags])` Compute theoretical autocovariance function
- `acorr([nlags])` Compute theoretical autocorrelation function
- `forecast(y, steps)` Produce linear minimum MSE forecasts for desired number of steps
- `forecast_cov(steps)` Compute theoretical forecast error variance matrices
- `forecast_interval(y, steps[, alpha])` Construct forecast interval estimates assuming the y are Gaussian
- `get_eq_index(name)` Return integer position of requested equation name
- `is_stable([verbose])` Determine stability based on model coefficients
- `long_run_effects()` Compute long-run effect of unit impulse ..
- `ma_rep([maxn])` Compute MA(∞) coefficient matrices
- `mean()` Mean of stable process
- `mse(steps)` Compute theoretical forecast error variance matrices
- `orth_ma_rep([maxn, P])` Compute Orthogonalized MA coefficient matrices using P matrix such
- `plot_acorr([nlags, linewidth])` Plot theoretical autocorrelation function
- `plotsim([steps])` Plot a simulation from the VAR(p) process for the desired number of

**statsmodels.tsa.vector_ar.var_model.VARProcess.acf**

`VARProcess.acf(nlags=None)`

Compute theoretical autocovariance function

**Returns**

`acf` : ndarray (p x k x k)
statsmodels.tsa.vector_ar.var_model.VARProcess.acorr

VARProcess.acorr(nlags=None)
Compute theoretical autocorrelation function

Returns acorr : ndarray (p x k x k)

statsmodels.tsa.vector_ar.var_model.VARProcess.forecast

VARProcess.forecast(y, steps)
Produce linear minimum MSE forecasts for desired number of steps ahead, using prior values y

Parameters y : ndarray (p x k)
steps : int

Returns forecasts : ndarray (steps x neqs)

Notes
Lutkepohl pp 37-38

statsmodels.tsa.vector_ar.var_model.VARProcess.forecast_cov

VARProcess.forecast_cov(steps)
Compute theoretical forecast error variance matrices

Parameters steps : int
Number of steps ahead

Returns forc_covs : ndarray (steps x neqs x neqs)

Notes

\[ \text{MSE}(h) = \sum_{i=0}^{h-1} \Phi \Sigma \theta \Phi^T \]

statsmodels.tsa.vector_ar.var_model.VARProcess.forecast_interval

VARProcess.forecast_interval(y, steps, alpha=0.05)
Construct forecast interval estimates assuming the y are Gaussian

Returns (lower, mid, upper) : (ndarray, ndarray, ndarray)

Notes
Lutkepohl pp. 39-40
statsmodels.tsa.vector_ar.var_model.VARProcess.get_eq_index

VARProcess.get_eq_index(name)
Return integer position of requested equation name

statsmodels.tsa.vector_ar.var_model.VARProcess.is_stable

VARProcess.is_stable(verbos=False)
Determine stability based on model coefficients

Parameters verbose : bool
Print eigenvalues of the VAR(1) companion

Notes
Checks if det(I - Az) = 0 for any mod(z) <= 1, so all the eigenvalues of the companion matrix must lie outside the unit circle

statsmodels.tsa.vector_ar.var_model.VARProcess.long_run_effects

VARProcess.long_run_effects()
Compute long-run effect of unit impulse

\[ \Psi_\infty = \sum_{i=0}^{\infty} \Phi_i \]

statsmodels.tsa.vector_ar.var_model.VARProcess.ma_rep

VARProcess.ma_rep(maxn=10)
Compute MA(\infty) coefficient matrices

Parameters maxn : int
Number of coefficient matrices to compute

Returns coeffs : ndarray (maxn x k x k)

statsmodels.tsa.vector_ar.var_model.VARProcess.mean

VARProcess.mean()
Mean of stable process

Lutkepohl eq. 2.1.23

\[ \mu = (I - A_1 - \cdots - A_p)^{-1} \alpha \]
**`statsmodels.tsa.vector_ar.var_model.VARProcess.mse`**

`VARProcess.mse(steps)`  
Compute theoretical forecast error variance matrices  

**Parameters**  
`steps` : int  
Number of steps ahead  

**Returns**  
`forc_covs` : ndarray (steps x neqs x neqs)

**Notes**

\[ \text{MSE}(h) = \sum_{i=0}^{h-1} \Phi \Sigma \Phi^T \]

**`statsmodels.tsa.vector_ar.var_model.VARProcess.orth_ma_rep`**

`VARProcess.orth_ma_rep(maxn=10, P=None)`  
Compute Orthogonalized MA coefficient matrices using P matrix such that \( \Sigma_u = PP' \). P defaults to the Cholesky decomposition of \( \Sigma_u \)

**Parameters**  
`maxn` : int  
Number of coefficient matrices to compute  

`P` : ndarray (k x k), optional  
Matrix such that Sigma_u = PP', defaults to Cholesky descomp  

**Returns**  
`coefs` : ndarray (maxn x k x k)

**`statsmodels.tsa.vector_ar.var_model.VARProcess.plot_acorr`**

`VARProcess.plot_acorr(nlags=10, linewidth=8)`  
Plot theoretical autocorrelation function

**`statsmodels.tsa.vector_ar.var_model.VARProcess.plotsim`**

`VARProcess.plotsim(steps=1000)`  
Plot a simulation from the VAR(p) process for the desired number of steps

**`statsmodels.tsa.vector_ar.var_model.VARResults`**

**class** `statsmodels.tsa.vector_ar.var_model.VARResults(  
endog,  
endog_lagged,  
params,  
sigma_u,  
lag_order,  
model=None,  
trend='c',  
names=None,  
dates=None)`

Estimate VAR(p) process with fixed number of lags
Parameters endog : array
    endog_lagged : array
    params : array
    sigma_u : array
    lag_order : int
    model : VAR model instance
    trend : str {'nc', 'c', 'ct'}
    names : array-like
        List of names of the endogenous variables in order of appearance in endog.

dates :

Returns **Attributes** :
    aic :
    bic :
    bse :
    coefs : ndarray (p x K x K)
        Estimated A_i matrices, A_i = coefs[i-1]
    cov_params :
    dates :
    detomega :
    df_model : int
    df_resid : int
    endog :
    endog_lagged :
    fittedvalues :
    fpe :
    intercept :
    info_criteria :
    k_ar : int
    k_trend : int
    llf :
    model :
    names :
    neqs : int
        Number of variables (equations)
nobs : int
n_totobs : int

params :
k_ar : int
   Order of VAR process

params : ndarray (Kp + 1) x K
   A_i matrices and intercept in stacked form [int A_1 ... A_p]

pvalues :

names : list
   variables names

resid :

roots : array
   The roots of the VAR process are the solution to (I - coefs[0]*z - coefs[1]*z**2 ... - coefs[p-1]*z**k_ar) = 0. Note that the inverse roots are returned, and stability requires that the roots lie outside the unit circle.

sigma_u : ndarray (K x K)
   Estimate of white noise process variance Var[u_t]

sigma_u_mle :

stderr :

trenorder :

tvalues :

y :

ys_lagged :

Methods

acf([nlags]) Compute theoretical autocovariance function
acorr([nlags]) Compute theoretical autocorrelation function
bse() Standard errors of coefficients, reshaped to match in size
coef_names() Coefficient names (deprecated)
cov_params() Estimated variance-covariance of model coefficients ..
cov_ybar() Asymptotically consistent estimate of covariance of the sample mean ..
detomega() Return determinant of white noise covariance with degrees of freedom correction
fevd([periods, var_decomp]) Compute forecast error variance decomposition (“fevd”)
fittedvalues() The predicted insample values of the response variables of the model.
forecast(y, steps) Produce linear minimum MSE forecasts for desired number of steps
forecast_cov([steps]) Compute forecast covariance matrices for desired number of steps
forecast_interval(y, steps[, alpha]) Construct forecast interval estimates assuming the y are Gaussian
get_eq_index(name) Return integer position of requested equation name
info_criteria() information criteria for lagorder selection
irf([periods, var_decomp, var_order]) Analyze impulse responses to shocks in system

Continued on next page

3.7. Time Series analysis tsa
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>irf_errband_mc([orth, repl, T, signif, ...])</code></td>
<td>Compute Monte Carlo integrated error bands assuming normally</td>
</tr>
<tr>
<td><code>irf_resim([orth, repl, T, seed, burn, cum])</code></td>
<td>Simulates impulse response function, returning an array of simulations.</td>
</tr>
<tr>
<td><code>is_stable([verbose])</code></td>
<td>Determine stability based on model coefficients</td>
</tr>
<tr>
<td><code>llf()</code></td>
<td>Compute VAR(p) loglikelihood</td>
</tr>
<tr>
<td><code>long_run_effects()</code></td>
<td>Compute long-run effect of unit impulse ..</td>
</tr>
<tr>
<td><code>ma_rep([maxn])</code></td>
<td>Compute MA((\infty)) coefficient matrices</td>
</tr>
<tr>
<td><code>mean()</code></td>
<td>Mean of stable process</td>
</tr>
<tr>
<td><code>mse(steps)</code></td>
<td>Compute theoretical forecast error variance matrices</td>
</tr>
<tr>
<td><code>orth_ma_rep([maxn, P])</code></td>
<td>Compute Orthogonalized MA coefficient matrices using P matrix such</td>
</tr>
<tr>
<td><code>plot()</code></td>
<td>Plot input time series</td>
</tr>
<tr>
<td><code>plot_acorr([nlags, linewidth])</code></td>
<td>Plot theoretical autocorrelation function</td>
</tr>
<tr>
<td><code>plot_forecast(steps[, alpha, plot_stderr])</code></td>
<td>Plot forecast</td>
</tr>
<tr>
<td><code>plot_sample_acorr([nlags, linewidth])</code></td>
<td>Plot theoretical autocorrelation function</td>
</tr>
<tr>
<td><code>plotsim([steps])</code></td>
<td>Plot a simulation from the VAR(p) process for the desired number of</td>
</tr>
<tr>
<td><code>pvalues()</code></td>
<td>Two-sided p-values for model coefficients from Student t-distribution</td>
</tr>
<tr>
<td><code>reorder(order)</code></td>
<td>Reorder variables for structural specification</td>
</tr>
<tr>
<td><code>resid()</code></td>
<td>Residuals of response variable resulting from estimated coefficients</td>
</tr>
<tr>
<td><code>resid_acorr([nlags])</code></td>
<td>Compute sample autocorrelation (including lag 0)</td>
</tr>
<tr>
<td><code>resid_acov([nlags])</code></td>
<td>Compute centered sample autocovariance (including lag 0)</td>
</tr>
<tr>
<td><code>resid_corr()</code></td>
<td>Centered residual correlation matrix</td>
</tr>
<tr>
<td><code>roots()</code></td>
<td>(Biased) maximum likelihood estimate of noise process covariance</td>
</tr>
<tr>
<td><code>sample_acorr([nlags])</code></td>
<td>Standard errors of coefficients, reshaped to match in size</td>
</tr>
<tr>
<td><code>sample_acov([nlags])</code></td>
<td>Compute console output summary of estimates</td>
</tr>
<tr>
<td><code>sigma_u_mle()</code></td>
<td>Compute test statistic for null hypothesis of Granger-noncausality,</td>
</tr>
<tr>
<td><code>stderr()</code></td>
<td>Test assumption of normal-distributed errors using Jarque-Bera-style</td>
</tr>
<tr>
<td><code>summary()</code></td>
<td>Test white noise assumption.</td>
</tr>
<tr>
<td><code>tvalues()</code></td>
<td>Compute t-statistics.</td>
</tr>
</tbody>
</table>

```python
statsmodels.tsa.vector_ar.var_model.VARResults.acf

VARResults.acf(nlags=None)
    Compute theoretical autocovariance function

    Returns acf : ndarray (p x k x k)
```

```python
statsmodels.tsa.vector_ar.var_model.VARResults.acorr

VARResults.acorr(nlags=None)
    Compute theoretical autocorrelation function

    Returns acorr : ndarray (p x k x k)
```

```python
statsmodels.tsa.vector_ar.var_model.VARResults.bse

static VARResults.bse()
    Standard errors of coefficients, reshaped to match in size
```
statsmodels.tsa.vector_ar.var_model.VARResults.coef_names

```python
static VARResults.coef_names()
    Coefficient names (deprecated)
```

statsmodels.tsa.vector_ar.var_model.VARResults.cov_params

```python
static VARResults.cov_params()
    Estimated variance-covariance of model coefficients
```

**Notes**

Covariance of vec(B), where B is the matrix [intercept, A_1, ..., A_p] (K x (Kp + 1)) Adjusted to be an unbiased estimator Ref: Lutkepohl p.74-75

statsmodels.tsa.vector_ar.var_model.VARResults.cov_ybar

```python
VARResults.cov_ybar()
    Asymptotically consistent estimate of covariance of the sample mean
    √(T)(ȳ - μ) → N(0, Σ_ȳ)
    Σ_ȳ = BΣ_uB', where B = (I_K - A_1 - ... - A_p)^{-1}
```

**Notes**

Lutkepohl Proposition 3.3

statsmodels.tsa.vector_ar.var_model.VARResults.detomega

```python
static VARResults.detomega()
    Return determinant of white noise covariance with degrees of freedom correction:
    ~Ω = \frac{T}{T - kp - 1} ~Ω_{MLE}
```

statsmodels.tsa.vector_ar.var_model.VARResults.fevd

```python
VARResults.fevd(periods=10, var_decomp=None)
    Compute forecast error variance decomposition (“fevd”)
    Returns fevd : FEVD instance
```

statsmodels.tsa.vector_ar.var_model.VARResults.fittedvalues

```python
static VARResults.fittedvalues()
    The predicted insample values of the response variables of the model.
```
VARResults.forecast(y, steps)
    Produce linear minimum MSE forecasts for desired number of steps ahead, using prior values y
    Parameters
        y : ndarray (p x k)
        steps : int
    Returns
        forecasts : ndarray (steps x neqs)

Notes
Lutkepohl pp 37-38

VARResults.forecast_cov(steps=1)
    Compute forecast covariance matrices for desired number of steps
    Parameters
        steps : int
    Returns
        covs : ndarray (steps x k x k)

Notes
\[ \Sigma_y(h) = \Sigma_y(h) + \Omega(h)/T \]
Ref: Lutkepohl pp. 96-97

VARResults.forecast_interval(y, steps, alpha=0.05)
    Construct forecast interval estimates assuming the y are Gaussian
    Returns
        (lower, mid, upper) : (ndarray, ndarray, ndarray)

Notes
Lutkepohl pp. 39-40

VARResults.get_eq_index(name)
    Return integer position of requested equation name
```python
statsmodels.tsa.vector_ar.var_model.VARResults.info_criteria

static VARResults.info_criteria()
    information criteria for lagorder selection

statsmodels.tsa.vector_ar.var_model.VARResults.irf

VARResults.irf(periods=10, var_decomp=None, var_order=None)
    Analyze impulse responses to shocks in system

    Parameters
    __________
    periods : int
    var_decomp : ndarray (k x k), lower triangular
    Must satisfy Omega = P P', where P is the passed matrix. Defaults to Cholesky decomp-
    osition of Omega
    var_order : sequence
    Alternate variable order for Cholesky decomposition

    Returns
    -------
    irf : IRAnalysis

statsmodels.tsa.vector_ar.var_model.VARResults.irf_errband_mc

VARResults.irf_errband_mc(orth=False, repl=1000, T=10, signif=0.05, seed=None, burn=100, cum=False)
    Compute Monte Carlo integrated error bands assuming normally distributed for impulse response functions

    Parameters
    __________
    orth : bool, default False
    Compute orthogonalized impulse response error bands
    repl : int
    number of Monte Carlo replications to perform
    T : int, default 10
    number of impulse response periods
    signif : float (0 < signif <1)
    Significance level for error bars, defaults to 95% CI
    seed : int
    np.random.seed for replications
    burn : int
    number of initial observations to discard for simulation
    cum : bool, default False
    produce cumulative irf error bands

    Returns
    -------
    Tuple of lower and upper arrays of ma_rep monte carlo standard errors

Notes

Lutkepohl (2005) Appendix D
```
statsmodels.tsa.vector_ar.var_model.VARResults.irf_resim

VARResults.irf_resim(orth=False, repl=1000, T=10, seed=None, burn=100, cum=False)

Simulates impulse response function, returning an array of simulations. Used for Sims-Zha error band calculation.

Parameters

- **orth**: bool, default False
  Compute orthogonalized impulse response error bands

- **repl**: int
  number of Monte Carlo replications to perform

- **T**: int, default 10
  number of impulse response periods

- **signif**: float (0 < signif <1)
  Significance level for error bars, defaults to 95% CI

- **seed**: int
  np.random.seed for replications

- **burn**: int
  number of initial observations to discard for simulation

- **cum**: bool, default False
  produce cumulative irf error bands

Returns

Array of simulated impulse response functions

Notes


statsmodels.tsa.vector_ar.var_model.VARResults.is_stable

VARResults.is_stable(verbose=False)

Determine stability based on model coefficients

Parameters

- **verbose**: bool
  Print eigenvalues of the VAR(1) companion

Notes

Checks if det(I - Az) = 0 for any mod(z) <= 1, so all the eigenvalues of the companion matrix must lie outside the unit circle
statsmodels.tsa.vector_ar.var_model.VARResults.llf

static VARResults.llf()  
Compute VAR(p) loglikelihood

statsmodels.tsa.vector_ar.var_model.VARResults.long_run_effects

VARResults.long_run_effects()  
Compute long-run effect of unit impulse

\[ \Psi_\infty = \sum_{i=0}^{\infty} \Phi_i \]

statsmodels.tsa.vector_ar.var_model.VARResults.ma_rep

VARResults.ma_rep(maxn=10)  
Compute MA(\infty) coefficient matrices

Parameters maxn : int  
Number of coefficient matrices to compute

Returns coefs : ndarray (maxn x k x k)

statsmodels.tsa.vector_ar.var_model.VARResults.mean

VARResults.mean()  
Mean of stable process

Lutkepohl eq. 2.1.23

\[ \mu = (I - A_1 - \cdots - A_p)^{-1} \alpha \]

statsmodels.tsa.vector_ar.var_model.VARResults.mse

VARResults.mse(steps)  
Compute theoretical forecast error variance matrices

Parameters steps : int  
Number of steps ahead

Returns forc_covs : ndarray (steps x neqs x neqs)

Notes

\[ \text{MSE}(h) = \sum_{i=0}^{h-1} \Phi \Sigma_u \Phi^T \]
statsmodels.tsa.vector_ar.var_model.VARResults.orth_ma_rep

VARResults.orth_ma_rep(maxn=10, P=None)
Compute Orthogonalized MA coefficient matrices using P matrix such that \( \Sigma_u = PP' \). P defaults to the Cholesky decomposition of \( \Sigma_u \)

Parameters maxn : int
Number of coefficient matrices to compute
P : ndarray (k x k), optional
Matrix such that Sigma_u = PP', defaults to Cholesky descomp

Returns coefs : ndarray (maxn x k x k)

statsmodels.tsa.vector_ar.var_model.VARResults.plot

VARResults.plot()
Plot input time series

statsmodels.tsa.vector_ar.var_model.VARResults.plot_acorr

VARResults.plot_acorr(nlags=10, linewidth=8)
Plot theoretical autocorrelation function

statsmodels.tsa.vector_ar.var_model.VARResults.plot_forecast

VARResults.plot_forecast(steps, alpha=0.05, plot_stderr=True)
Plot forecast

statsmodels.tsa.vector_ar.var_model.VARResults.plot_sample_acorr

VARResults.plot_sample_acorr(nlags=10, linewidth=8)
Plot theoretical autocorrelation function

statsmodels.tsa.vector_ar.var_model.VARResults.plotsim

VARResults.plotsim(steps=1000)
Plot a simulation from the VAR(p) process for the desired number of steps

statsmodels.tsa.vector_ar.var_model.VARResults.pvalues

static VARResults.pvalues()
Two-sided p-values for model coefficients from Student t-distribution

statsmodels.tsa.vector_ar.var_model.VARResults.reorder

VARResults.reorder(order)
Reorder variables for structural specification
statsmodels.tsa.vector_ar.var_model.VARResults.resid

```python
static VARResults.resid()
   Residuals of response variable resulting from estimated coefficients
```

statsmodels.tsa.vector_ar.var_model.VARResults.resid_acorr

```python
VARResults.resid_acorr(nlags=1)
   Compute sample autocorrelation (including lag 0)

Parameters nlags : int
```

statsmodels.tsa.vector_ar.var_model.VARResults.resid_acov

```python
VARResults.resid_acov(nlags=1)
   Compute centered sample autocovariance (including lag 0)

Parameters nlags : int
```

statsmodels.tsa.vector_ar.var_model.VARResults.resid_corr

```python
static VARResults.resid_corr()
   Centered residual correlation matrix
```

statsmodels.tsa.vector_ar.var_model.VARResults.roots

```python
static VARResults.roots()
```

statsmodels.tsa.vector_ar.var_model.VARResults.sample_acorr

```python
VARResults.sample_acorr(nlags=1)
```

statsmodels.tsa.vector_ar.var_model.VARResults.sample_acov

```python
VARResults.sample_acov(nlags=1)
```

statsmodels.tsa.vector_ar.var_model.VARResults.sigma_u_mle

```python
static VARResults.sigma_u_mle()
   (Biased) maximum likelihood estimate of noise process covariance
```

statsmodels.tsa.vector_ar.var_model.VARResults.stderr

```python
static VARResults.stderr()
   Standard errors of coefficients, reshaped to match in size
```
VARResults.summary()  
Compute console output summary of estimates  

Returns summary : VARSummary

VARResults.test_causality(equation, variables, kind='f', signif=0.05, verbose=True)  
Compute test statistic for null hypothesis of Granger-noncausality, general function to test joint Granger-causality of multiple variables  

Parameters equation : string or int  
Equation to test for causality  
variables : sequence (of strings or ints)  
List, tuple, etc. of variables to test for Granger-causality  
kind : {'f', 'wald'}  
Perform F-test or Wald (chi-sq) test  
signif : float, default 5%  
Significance level for computing critical values for test, defaulting to standard 0.95 level  

Returns results : dict

Notes  
Null hypothesis is that there is no Granger-causality for the indicated variables. The degrees of freedom in the F-test are based on the number of variables in the VAR system, that is, degrees of freedom are equal to the number of equations in the VAR times degree of freedom of a single equation.

VARResults.test_normality(signif=0.05, verbose=True)  
Test assumption of normal-distributed errors using Jarque-Bera-style omnibus Chi^2 test  

Parameters signif : float  
Test significance threshold  

Notes  
H0 (null) : data are generated by a Gaussian-distributed process

VARResults.test_whiteness(nlags=10, plot=True, linewidth=8)  
Test white noise assumption. Sample (Y) autocorrelations are compared with the standard $2/\sqrt{T}$ bounds.
**Parameters plot**: boolean, default True

Plot autocorrelations with $2 / \sqrt{T}$ bounds

```python
statsmodels.tsa.vector_ar.var_model.VARResults.tvalues
```

```python
static VARResults.tvalues()
```

Compute t-statistics. Use Student-t($T - Kp - 1$) = $t(df_{resid})$ to test significance.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aic</td>
<td>Akaike information criterion</td>
</tr>
<tr>
<td>bic</td>
<td>Bayesian a.k.a.</td>
</tr>
<tr>
<td>df_model</td>
<td>Number of estimated parameters, including the intercept / trends</td>
</tr>
<tr>
<td>df_resid</td>
<td>Number of observations minus number of estimated parameters</td>
</tr>
<tr>
<td>fpe</td>
<td>Final Prediction Error (FPE)</td>
</tr>
<tr>
<td>hqic</td>
<td>Hannan-Quinn criterion</td>
</tr>
</tbody>
</table>

```python
statsmodels.tsa.vector_ar.irf.IRAnalysis
```

```python
class statsmodels.tsa.vector_ar.irf.IRAnalysis(model, P=None, periods=10, order=None, svar=False)
```

Impulse response analysis class. Computes impulse responses, asymptotic standard errors, and produces relevant plots

**Parameters model**: VAR instance

**Notes**

Using Lutkepohl (2005) notation

**Methods**

- `G()`: Compute asymptotic standard errors for impulse response coefficients.
- `H()`: Compute asymptotic standard errors for cumulative impulse response.
- `cov([orth])`: IRF Monte Carlo integrated error bands of cumulative effect.
- `cum_effect_cov([orth])`: IRF Sims-Zha error band method 1. Assumes symmetric error bands around mean.
- `cum_effect_stderr([orth])`: IRF Sims-Zha error band method 2.
- `cum_errband_mc([orth, repl, signif, seed, burn])`: IRF Sims-Zha error band method 3. Does not assume symmetric error bands around mean.
- `err_band_sz1([orth, svar, repl, signif, seed, burn])`: IRF Monte Carlo integrated error bands.
- `err_band_sz2([orth, repl, signif, seed, ...])`: IRF Monte Carlo integrated error bands.
- `err_band_sz3([orth, repl, signif, seed, ...])`: IRF Monte Carlo integrated error bands.
- `fevd_table()`: Plot impulse responses.
- `lr_effect_cov([orth])`: Plot cumulative impulse response functions.
- `lr_effect_stderr([orth])`: Plot cumulative impulse response functions.
- `plot([orth, impulse, response, signif, ...])`: Plot impulse responses.
- `plot_cum_effects([orth, impulse, response, ...])`: Plot cumulative impulse response functions.
- `stderr([orth])`: Plot cumulative impulse response functions.
statsmodels.tsa.vector_ar.irf.IRAnalysis.G

static IRAnalysis.G()

statsmodels.tsa.vector_ar.irf.IRAnalysis.H

static IRAnalysis.H()

statsmodels.tsa.vector_ar.irf.IRAnalysis.cov

IRAnalysis.cov(orth=False)
Compute asymptotic standard errors for impulse response coefficients

Notes
Lutkepohl eq 3.7.5

statsmodels.tsa.vector_ar.irf.IRAnalysis.cum_effect_cov

IRAnalysis.cum_effect_cov(orth=False)
Compute asymptotic standard errors for cumulative impulse response coefficients

Parameters orth : boolean

Notes
eq. 3.7.7 (non-orth), 3.7.10 (orth)

statsmodels.tsa.vector_ar.irf.IRAnalysis.cum_effect_stderr

IRAnalysis.cum_effect_stderr(orth=False)

statsmodels.tsa.vector_ar.irf.IRAnalysis.cum_errband_mc

IRAnalysis.cum_errband_mc(orth=False, repl=1000, signif=0.05, seed=None, burn=100)
IRF Monte Carlo integrated error bands of cumulative effect

statsmodels.tsa.vector_ar.irf.IRAnalysis.err_band_sz1

IRAnalysis.err_band_sz1(orth=False, svar=False, repl=1000, signif=0.05, seed=None, burn=100, component=None)
IRF Sims-Zha error band method 1. Assumes symmetric error bands around mean.

Parameters orth : bool, default False
Compute orthogonalized impulse responses
repl : int, default 1000
Number of MC replications

**signif**: float \(0 < \text{signif} < 1\)
Significance level for error bars, defaults to 95% CI

**seed**: int, default None
np.random seed

**burn**: int, default 100
Number of initial simulated obs to discard

**component**: \(\text{neqs} \times \text{neqs}\) array, default to largest for each
Index of column of eigenvector/value to use for each error band Note: period of impulse \((t=0)\) is not included when computing principle component

**References**


```python
statsmodels.tsa.vector_ar.irf.IRAnalysis.err_band_sz2

IRAnalysis.err_band_sz2(orth=False, repl=1000, signif=0.05, seed=None, burn=100, component=None)

IRF Sims-Zha error band method 2.
This method Does not assume symmetric error bands around mean.

**Parameters**
**orth**: bool, default False
Compute orthogonalized impulse responses

**repl**: int, default 1000
Number of MC replications

**signif**: float \(0 < \text{signif} < 1\)
Significance level for error bars, defaults to 95% CI

**seed**: int, default None
np.random seed

**burn**: int, default 100
Number of initial simulated obs to discard

**component**: \(\text{neqs} \times \text{neqs}\) array, default to largest for each
Index of column of eigenvector/value to use for each error band Note: period of impulse \((t=0)\) is not included when computing principle component

**References**


3.7. Time Series analysis tsa
IRAnalysis.err_band_sz3(orth=False, repl=1000, signif=0.05, seed=None, burn=100, component=None)

IRF Sims-Zha error band method 3. Does not assume symmetric error bands around mean.

**Parameters**

- **orth**: bool, default False
  - Compute orthogonalized impulse responses
- **repl**: int, default 1000
  - Number of MC replications
- **signif**: float (0 < signif < 1)
  - Significance level for error bars, defaults to 95% CI
- **seed**: int, default None
  - np.random seed
- **burn**: int, default 100
  - Number of initial simulated obs to discard
- **component**: vector length neqs, default to largest for each
  - Index of column of eigenvector/value to use for each error band Note: period of impulse (t=0) is not included when computing principle component

**References**

IRAnalysis.plot(orth=False, impulse=None, response=None, signif=0.05, plot_params=None, subplot_params=None, plot_stderr=True, stderr_type='asym', repl=1000, seed=None, component=None)

Plot impulse responses

**Parameters**

- **orth**: bool, default False
  
  Compute orthogonalized impulse responses

- **impulse**: string or int
  
  variable providing the impulse

- **response**: string or int
  
  variable affected by the impulse

- **signif**: float ($0 < \text{signif} < 1$)
  
  Significance level for error bars, defaults to 95% CI

- **subplot_params**: dict
  
  To pass to subplot plotting functions. Example: if fonts are too big, pass `{'fontsize' : 8}` or some number to your taste.

- **plot_params**: dict

- **plot_stderr**: bool, default True
  
  Plot standard impulse response error bands

- **stderr_type**: string
  
  ‘asym’: default, computes asymptotic standard errors
  ‘mc’: monte carlo standard errors (use repl)

- **repl**: int, default 1000
  
  Number of replications for Monte Carlo and Sims-Zha standard errors

- **seed**: int
  
  np.random.seed for Monte Carlo replications

- **component**: array or vector of principal component indices

IRAnalysis.plot_cum_effects(orth=False, impulse=None, response=None, signif=0.05, plot_params=None, subplot_params=None, plot_stderr=True, stderr_type='asym', repl=1000, seed=None)

Plot cumulative impulse response functions

**Parameters**

- **orth**: bool, default False
  
  Compute orthogonalized impulse responses

- **impulse**: string or int
  
  variable providing the impulse

- **response**: string or int
variable affected by the impulse

**signif**: float \((0 < \text{signif} < 1)\)

Significance level for error bars, defaults to 95% CI

**subplot_params**: dict

To pass to subplot plotting functions. Example: if fonts are too big, pass \{'fontsize': 8\} or some number to your taste.

**plot_params**: dict

**plot_stderr**: bool, default True:

Plot standard impulse response error bands

**stderr_type**: string:

‘asym’: default, computes asymptotic standard errors ‘mc’: monte carlo standard errors (use rpl)

**repl**: int, default 1000:

Number of replications for monte carlo standard errors

**seed**: int:

np.random.seed for Monte Carlo replications

```python
statsmodels.tsa.vector_ar.irf.IRAnalysis.stderr

IRAnalysis.stderr(orth=False)
```

```python
statsmodels.tsa.vector_ar.var_model.FEVD

class statsmodels.tsa.vector_ar.var_model.FEVD(model, P=None, periods=None)

Compute and plot Forecast error variance decomposition and asymptotic standard errors

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cov()</td>
<td>Compute asymptotic standard errors</td>
</tr>
<tr>
<td>plot([periods, figsize])</td>
<td>Plot graphical display of FEVD</td>
</tr>
<tr>
<td>summary()</td>
<td></td>
</tr>
</tbody>
</table>

```python
statsmodels.tsa.vector_ar.var_model.FEVD.cov

FEVD.cov()

Compute asymptotic standard errors

```python
statsmodels.tsa.vector_ar.var_model.FEVD.plot

FEVD.plot(periods=None, figsize=(10, 10), **plot_kwds)

Plot graphical display of FEVD

**Parameters**

**periods**: int, default None
Defaults to number originally specified. Can be at most that number

```
statsmodels.tsa.vector_ar.var_model.FEVD.summary
```

```
FEVD.summary()
```

### statsmodels.tsa.vector_ar.dynamic.DynamicVAR

**class** `statsmodels.tsa.vector_ar.dynamic.DynamicVAR(data, lag_order=1, window=None, window_type='expanding', trend='c', min_periods=None)`

Estimates time-varying vector autoregression (VAR(p)) using equation-by-equation least squares

**Parameters**
- `data` : pandas.DataFrame
- `lag_order` : int, default 1
- `window` : int
- `window_type` : {'expanding', 'rolling'}
- `min_periods` : int or None
  - Minimum number of observations to require in window, defaults to window size if None specified
- `trend` : {'c', 'nc', 'ct', 'ctt'}

**TODO**

**Returns**
- **Attributes**:
  - `coefs` : WidePanel
    - `items` : coefficient names
    - `major_axis` : dates
    - `minor_axis` : VAR equation names

**Methods**

```
T()
coefs()
equations()
forecast(steps)
plot_forecast([steps, figsize])
r2()
resid()
```

**Number of time periods in results**

**Return dynamic regression coefficients as WidePanel**

**Produce dynamic forecast**

**Plot h-step ahead forecasts against actual realizations of time series.**

**Returns the r-squared values.**

**Statsmodels.tsa.vector_ar.dynamic.DynamicVAR.T**

**static** `DynamicVAR.T()`

Number of time periods in results

**Statsmodels.tsa.vector_ar.dynamic.DynamicVAR.coefs**

**static** `DynamicVAR.coefs()`

Return dynamic regression coefficients as WidePanel
statsmodels Documentation, Release 0.6.0

```
statsmodels.tsa.vector_ar.dynamic.DynamicVAR.equations

static DynamicVAR.equations()
```

```
statsmodels.tsa.vector_ar.dynamic.DynamicVAR.forecast

DynamicVAR.forecast(steps=1)
    Produce dynamic forecast
    Parameters steps:
    Returns forecasts: pandas.DataFrame
```

```
statsmodels.tsa.vector_ar.dynamic.DynamicVAR.plot_forecast

DynamicVAR.plot_forecast(steps=1, figsize=(10, 10))
    Plot h-step ahead forecasts against actual realizations of time series. Note that forecasts are lined up with their respective realizations.
    Parameters steps:
```

```
statsmodels.tsa.vector_ar.dynamic.DynamicVAR.r2

static DynamicVAR.r2()
    Returns the r-squared values.
```

```
statsmodels.tsa.vector_ar.dynamic.DynamicVAR.resid

static DynamicVAR.resid()
```

**Attributes**

<table>
<thead>
<tr>
<th>nobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>result_index</td>
</tr>
</tbody>
</table>

See also:
tutorial VAR documentation

### 3.7.4 ARMA Process

The following are tools to work with the theoretical properties of an ARMA process for given lag-polynomials.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arima_process.ArmaProcess(ar, ma[, nobs])</td>
<td>represents an ARMA process for given lag-polynomials</td>
</tr>
<tr>
<td>arima_process.ar2arma(ar_des, p, q[, n, ...])</td>
<td>find arma approximation to ar process</td>
</tr>
<tr>
<td>arima_process.arma2ar(ar, ma[, nobs])</td>
<td>get the AR representation of an ARMA process</td>
</tr>
<tr>
<td>arima_process.arma2ma(ar, ma[, nobs])</td>
<td>get the impulse response function (MA representation) for ARMA process</td>
</tr>
<tr>
<td>arima_process arma_acf(ar, ma[, nobs])</td>
<td>theoretical autocorrelation function of an ARMA process</td>
</tr>
<tr>
<td>arima_process arma_acovf(ar, ma[, nobs])</td>
<td>theoretical autocovariance function of ARMA process</td>
</tr>
</tbody>
</table>
Table 3.170 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arima_process.arma_generate_sample(ar, ma, ...)</code></td>
<td>Generate a random sample of an ARMA process</td>
</tr>
<tr>
<td><code>arima_process.arma_impulse_response(ar, ma)</code></td>
<td>get the impulse response function (MA representation) for ARMA process</td>
</tr>
<tr>
<td><code>arima_process.arma_pacf(ar, ma[, nobs])</code></td>
<td>partial autocorrelation function of an ARMA process</td>
</tr>
<tr>
<td><code>arima_process.arma_periodogram(ar, ma[, ...])</code></td>
<td>periodogram for ARMA process given by lag-polynomials ar and ma</td>
</tr>
<tr>
<td><code>arima_process.deconvolve(num, den[, n])</code></td>
<td>Deconvolves divisor out of signal, division of polynomials for n terms</td>
</tr>
<tr>
<td><code>arima_process.index2lpol(coeffs, index)</code></td>
<td>expand coefficients to lag poly</td>
</tr>
<tr>
<td><code>arima_process.lpol2index(ar)</code></td>
<td>remove zeros from lagpolynomial, squeezed representation with index</td>
</tr>
<tr>
<td><code>arima_process.lpol_fiar(d[, n])</code></td>
<td>AR representation of fractional integration</td>
</tr>
<tr>
<td><code>arima_process.lpol_fima(d[, n])</code></td>
<td>MA representation of fractional integration</td>
</tr>
<tr>
<td><code>arima_process.lpol_sdiff(s)</code></td>
<td>return coefficients for seasonal difference (1-L^s)</td>
</tr>
</tbody>
</table>

**statsmodels.tsa.arima_process.ArmaProcess**

class `statsmodels.tsa.arima_process.ArmaProcess(ar, ma, nobs=None)`

represents an ARMA process for given lag-polynomials

This is a class to bring together properties of the process. It does not do any estimation or statistical analysis.

maybe needs special handling for unit roots

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acf(nobs=None)</code></td>
<td>theoretical autocorrelation function of an ARMA process</td>
</tr>
<tr>
<td><code>acovf(nobs)</code></td>
<td>theoretical autocovariance function of ARMA process</td>
</tr>
<tr>
<td><code>ar_roots()</code></td>
<td>roots of autoregressive lag-polynomial</td>
</tr>
<tr>
<td><code>arma2ar([nobs])</code></td>
<td>create ArmaProcess instance from coefficients of the lag-polynomials</td>
</tr>
<tr>
<td><code>arma2ma([nobs])</code></td>
<td>create ArmaProcess instance from estimation results</td>
</tr>
<tr>
<td><code>from_coeffs(arcoefs, macoefs[, nobs])</code></td>
<td>get the impulse response function (MA representation) for ARMA process</td>
</tr>
<tr>
<td><code>from_estimation(model_results[, nobs])</code></td>
<td>make MA polynomial invertible by inverting roots inside unit circle</td>
</tr>
<tr>
<td><code>generate_sample([size, scale, distrvs, ...])</code></td>
<td>Arma process is invertible if MA roots are outside unit circle</td>
</tr>
<tr>
<td><code>impulse_response([nobs])</code></td>
<td>Arma process is stationary if AR roots are outside unit circle</td>
</tr>
<tr>
<td><code>invertroots([retnew])</code></td>
<td>roots of moving average lag-polynomial</td>
</tr>
<tr>
<td><code>isinvertible()</code></td>
<td>partial autocorrelation function of an ARMA process</td>
</tr>
<tr>
<td><code>isstationary()</code></td>
<td>periodogram for ARMA process given by lag-polynomials ar and ma</td>
</tr>
<tr>
<td><code>ma_roots()</code></td>
<td></td>
</tr>
<tr>
<td><code>pacf([nobs])</code></td>
<td></td>
</tr>
<tr>
<td><code>periodogram([nobs])</code></td>
<td></td>
</tr>
</tbody>
</table>

**statsmodels.tsa.arima_process.ArmaProcess.acf**

ArmaProcess.acf(nobs=None)

theoretical autocorrelation function of an ARMA process

**Parameters**

- `ar` : array_like, 1d
  - coefficient for autoregressive lag polynomial, including zero lag
- `ma` : array_like, 1d
  - coefficient for moving-average lag polynomial, including zero lag
- `nobs` : int
  - number of terms (lags plus zero lag) to include in returned acf

3.7. Time Series analysis ts a
Returns acf : array
autocorrelation of ARMA process given by ar, ma

See also:
arma_acovf, acf, acovf

statsmodels.tsa.arima_process.ArmaProcess.acovf

ArmaProcess.acovf(nobs=None)
theoretical autocovariance function of ARMA process

Parameters ar : array_like, 1d
coefficient for autoregressive lag polynomial, including zero lag
ma : array_like, 1d
coefficient for moving-average lag polynomial, including zero lag
nobs : int
number of terms (lags plus zero lag) to include in returned acovf

Returns acovf : array
autocovariance of ARMA process given by ar, ma

See also:
arma_acf, acovf

Notes
Tries to do some crude numerical speed improvements for cases with high persistence. However, this algorithm is slow if the process is highly persistent and only a few autocovariances are desired.

statsmodels.tsa.arima_process.ArmaProcess.ar_roots

ArmaProcess.ar_roots()
roots of autoregressive lag-polynomial

statsmodels.tsa.arima_process.ArmaProcess.arma2ar

ArmaProcess.arma2ar(nobs=None)

statsmodels.tsa.arima_process.ArmaProcess.arma2ma

ArmaProcess.arma2ma(nobs=None)

statsmodels.tsa.arima_process.ArmaProcess.from_coeffs

classmethod ArmaProcess.from_coeffs(arcoefs, macoefs, nobs=None)
create ArmaProcess instance from coefficients of the lag-polynomials
`statsmodels.tsa.arima_process.ArmaProcess.from_estimation`

**classmethod ArmaProcess.from_estimation(model_results, nobs=None)**

create ArmaProcess instance from estimation results

`statsmodels.tsa.arima_process.ArmaProcess.generate_sample`

**ArmaProcess.generate_sample(size=100, scale=1, distrvs=None, axis=0, burnin=0)**

generate ARMA samples

**Parameters**

- **size**: int or tuple of ints
  - If `size` is an integer, then this creates a 1d timeseries of length `size`. If `size` is a tuple, then the timeseries is along axis. All other axis have independent arma samples.

- **returns rvs**: ndarray
  - random sample(s) of arma process

**Notes**

Should work for n-dimensional with time series along axis, but not tested yet. Processes are sampled independently.

`statsmodels.tsa.arima_process.ArmaProcess.impulse_response`

**ArmaProcess.impulse_response(nobs=None)**

get the impulse response function (MA representation) for ARMA process

**Parameters**

- **ma**: array_like, 1d
  - moving average lag polynomial
- **ar**: array_like, 1d
  - auto regressive lag polynomial
- **nobs**: int
  - number of observations to calculate

**Returns**

- **ir**: array, 1d
  - impulse response function with nobs elements

**Notes**

This is the same as finding the MA representation of an ARMA(p,q). By reversing the role of ar and ma in the function arguments, the returned result is the AR representation of an ARMA(p,q), i.e

```python
ma_representation = arma_impulse_response(ar, ma, nobs=100) ar_representation = arma_impulse_response(ma, ar, nobs=100)
```

fully tested against matlab
Examples

AR(1)

```python
gt  arma_impulse_response([1.0, -0.8], [1.], nobs=10)
array([ 1. , 0.8 , 0.64 , 0.512 , 0.4096 , 0.32768 , 0.262144 , 0.2097152 , 0.16777216, 0.13421773])
```

this is the same as

```python
gt  0.8*np.arange(10)
array([ 1. , 0.8 , 0.64 , 0.512 , 0.4096 , 0.32768 , 0.262144 , 0.2097152 , 0.16777216, 0.13421773])
```

MA(2)

```python
gt  arma_impulse_response([1.0], [1., 0.5, 0.2], nobs=10)
array([ 1. , 0.5, 0.2, 0. , 0. , 0. , 0. , 0. , 0. , 0. ])
```

ARMA(1,2)

```python
gt  arma_impulse_response([1.0, -0.8], [1., 0.5, 0.2], nobs=10)
array([ 1. , 1.3 , 1.24 , 0.992 , 0.7936 , 0.63488 , 0.507904 , 0.4063232 , 0.32505856, 0.26004685])
```

`statsmodels.tsa.arima_process.ArmaProcess.invertroots`

`ArmaProcess.invertroots`(retnew=False)

make MA polynomial invertible by inverting roots inside unit circle

Parameters retnew : boolean

If False (default), then return the lag-polynomial as array. If True, then return a new instance with invertible MA-polynomial

Returns manew : array

new invertible MA lag-polynomial, returned if retnew is false.

wasinvertible : boolean

True if the MA lag-polynomial was already invertible, returned if retnew is false.

armaprocess : new instance of class

If retnew is true, then return a new instance with invertible MA-polynomial

`statsmodels.tsa.arima_process.ArmaProcess.isinvertible`

`ArmaProcess.isinvertible()`

Arma process is invertible if MA roots are outside unit circle

Returns isninvertible : boolean

True if moving average roots are outside unit circle
```
statsmodels.tsa.arima_process.ArmaProcess.isstationary

ArmaProcess.isstationary()
   Arma process is stationary if AR roots are outside unit circle

    Returns isstationary : boolean
   True if autoregressive roots are outside unit circle

statsmodels.tsa.arima_process.ArmaProcess.ma_roots

ArmaProcess.ma_roots()
   roots of moving average lag-polynomial

statsmodels.tsa.arima_process.ArmaProcess.pacf

ArmaProcess.pacf(nobs=None)
   partial autocorrelation function of an ARMA process

    Parameters ar : array_like, 1d
       coefficient for autoregressive lag polynomial, including zero lag
    ma : array_like, 1d
       coefficient for moving-average lag polynomial, including zero lag
    nobs : int
       number of terms (lags plus zero lag) to include in returned pacf

    Returns pacf : array
       partial autocorrelation of ARMA process given by ar, ma

Notes

   solves yule-walker equation for each lag order up to nobs lags
   not tested/checked yet

statsmodels.tsa.arima_process.ArmaProcess.periodogram

ArmaProcess.periodogram(nobs=None)
   periodogram for ARMA process given by lag-polynomials ar and ma

    Parameters ar : array_like
       autoregressive lag-polynomial with leading 1 and lhs sign
    ma : array_like
       moving average lag-polynomial with leading 1
    worN : {None, int}, optional
       option for scipy.signal.freqz (read “w or N”) If None, then compute at 512 frequencies around the unit circle. If a single integer, the compute at that many frequencies. Otherwise, compute the response at frequencies given in worN
```
**whole**: \{0,1\}, optional

options for scipy.signal.freqz Normally, frequencies are computed from 0 to pi (upper-
half of unit-circle. If whole is non-zero compute frequencies from 0 to 2*pi.

**Returns**

- **w**: array
  - frequencies
- **sd**: array
  - periodogram, spectral density

**Notes**

Normalization ?

This uses signal.freqz, which does not use fft. There is a fft version somewhere.

---

**statsmodels.tsa.arima_process.ar2arma**

statsmodels.tsa.arima_process.ar2arma\( (ar\_des, p, q, n=20, mse='ar', \text{start}=\text{None}) \)

find arma approximation to ar process

This finds the ARMA(p,q) coefficients that minimize the integrated squared difference between the im-
pulse_response functions (MA representation) of the AR and the ARMA process. This does currently not
check whether the MA lagpolynomial of the ARMA process is invertible, neither does it check the roots of the
AR lagpolynomial.

**Parameters**

- **ar_des**: array_like
  - coefficients of original AR lag polynomial, including lag zero
- **p, q**: int
  - length of desired ARMA lag polynomials
- **n**: int
  - number of terms of the impuls_response function to include in the objective function
    for the approximation
- **mse**: string, ‘ar’
  - not used yet,

**Returns**

- **ar_app, ma_app**: arrays
  - coefficients of the AR and MA lag polynomials of the approximation
- **res**: tuple
  - result of optimize.leastsq

**Notes**

Extension is possible if we want to match autocovariance instead of impulse response function.

TODO: convert MA lag polynomial, ma_app, to be invertible, by mirroring roots outside the unit intervall to
ones that are inside. How do we do this?
statsmodels.tsa.arima_process arma2ar

statsmodels.tsa.arima_process arma2ar(ar, ma, nobs=100)
get the AR representation of an ARMA process

Parameters:
- ar : array_like, 1d
  auto regressive lag polynomial
- ma : array_like, 1d
  moving average lag polynomial
- nobs : int
  number of observations to calculate

Returns:
- ar : array, 1d
  coefficients of AR lag polynomial with nobs elements

Notes
This is just an alias for
ar_representation = arma_impulse_response(ma, ar, nobs=100)
fully tested against matlab

statsmodels.tsa.arima_process arma2ma

statsmodels.tsa.arima_process arma2ma(ar, ma, nobs=100)
get the impulse response function (MA representation) for ARMA process

Parameters:
- ma : array_like, 1d
  moving average lag polynomial
- ar : array_like, 1d
  auto regressive lag polynomial
- nobs : int
  number of observations to calculate

Returns:
- ir : array, 1d
  impulse response function with nobs elements

Notes
This is the same as finding the MA representation of an ARMA(p,q). By reversing the role of ar and ma in the
function arguments, the returned result is the AR representation of an ARMA(p,q), i.e
ma_representation = arma_impulse_response(ar, ma, nobs=100)
ar_representation = arma_impulse_response(ma, ar, nobs=100)
fully tested against matlab
Examples

AR(1)

```python
>>> arma_impulse_response([1.0, -0.8], [1.], nobs=10)
array([ 1. , 0.8 , 0.64 , 0.512 , 0.4096 ,
       0.32768, 0.262144, 0.2097152, 0.16777216, 0.13421773])
```

this is the same as

```python
>>> 0.8**np.arange(10)
array([ 1. , 0.8 , 0.64 , 0.512 , 0.4096 ,
       0.32768, 0.262144, 0.2097152, 0.16777216, 0.13421773])
```

MA(2)

```python
>>> arma_impulse_response([1.0], [1., 0.5, 0.2], nobs=10)
array([ 1. , 0.5, 0.2, 0., 0., 0., 0., 0., 0., 0. ])
```

ARMA(1,2)

```python
>>> arma_impulse_response([1.0, -0.8], [1., 0.5, 0.2], nobs=10)
array([ 1. , 1.3 , 1.24 , 0.992 , 0.7936 ,
       0.63488, 0.507904, 0.4063232, 0.32505856, 0.26004685])
```

---

**statsmodels.tsa.arima_process arma_acf**

*statsmodels.tsa.arima_process.arma_acf$(ar, ma, nobs=10)$*

theoretical autocorrelation function of an ARMA process

Parameters

- **ar**: array_like, 1d
  coefficient for autoregressive lag polynomial, including zero lag
- **ma**: array_like, 1d
  coefficient for moving-average lag polynomial, including zero lag
- **nobs**: int
  number of terms (lags plus zero lag) to include in returned acf

Returns **acf**: array
autocorrelation of ARMA process given by ar, ma

See also:

- `arma_acovf`, `acf`, `acovf`

---

**statsmodels.tsa.arima_process arma_acovf**

*statsmodels.tsa.arima_process.arma_acovf$(ar, ma, nobs=10)$*

theoretical autocovariance function of ARMA process

Parameters

- **ar**: array_like, 1d
  coefficient for autoregressive lag polynomial, including zero lag
- **ma**: array_like, 1d
  coefficient for moving-average lag polynomial, including zero lag
nobs : int
   number of terms (lags plus zero lag) to include in returned acovf

Returns acovf : array
   autocovariance of ARMA process given by ar, ma

See also:
   arma_acf, acovf

Notes
   Tries to do some crude numerical speed improvements for cases with high persistence. However, this algorithm
   is slow if the process is highly persistent and only a few autocovariances are desired.

statsmodels.tsa.arima_process arma_generate_sample

statsmodels.tsa.arima_process arma_generate_sample(ar, ma, nsample, sigma=1,
   distrvs=<built-in method randn of mtrand.RandomState object at
   0x005899E0>, burnin=0)

   Generate a random sample of an ARMA process

Parameters ar : array_like, 1d
   coefficient for autoregressive lag polynomial, including zero lag
ma : array_like, 1d
   coefficient for moving-average lag polynomial, including zero lag
nsample : int
   length of simulated time series
sigma : float
   standard deviation of noise
distrvs : function, random number generator
   function that generates the random numbers, and takes sample size as argument default:
   np.random.randn TODO: change to size argument
burnin : integer (default: 0)
   to reduce the effect of initial conditions, burnin observations at the beginning of the
   sample are dropped

Returns sample : array
   sample of ARMA process given by ar, ma of length nsample

Notes
   As mentioned above, both the AR and MA components should include the coefficient on the zero-lag. This is
typically 1. Further, due to the conventions used in signal processing used in signal.lfilter vs. conventions in
statistics for ARMA processes, the AR parameters should have the opposite sign of what you might expect. See
the examples below.
Examples

```python
>>> import numpy as np
>>> np.random.seed(12345)
>>> ar = np.array([.75, -.25])
>>> ma = np.array([.65, .35])
>>> arparams = np.r_[1, -ar] # add zero-lag and negate
>>> maparams = np.r_[1, ma] # add zero-lag
>>> y = sm.tsa.arma_generate_sample(arparams, maparams, 250)
>>> model = sm.tsa.ARMA(y, (2, 2)).fit(trend='nc', disp=0)
>>> model.params
array([ 0.79044189, -0.23140636, 0.70072904, 0.40608028])
```

**statsmodels tsa.arima_process arma_impulse_response**

This function gets the impulse response function (MA representation) for ARMA process.

**Parameters**
- **ma**: array_like, 1d
  - Moving average lag polynomial
- **ar**: array_like, 1d
  - Auto regressive lag polynomial
- **nobs**: int
  - Number of observations to calculate

**Returns**
- **ir**: array, 1d
  - Impulse response function with nobs elements

**Notes**

This is the same as finding the MA representation of an ARMA(p,q). By reversing the role of ar and ma in the function arguments, the returned result is the AR representation of an ARMA(p,q), i.e.

```
ma_representation = arma_impulse_response(ar, ma, nobs=100)
ar_representation = arma_impulse_response(ma, ar, nobs=100)
```

Fully tested against matlab.

**Examples**

**AR(1)**

```python
>>> arma_impulse_response([1.0, -0.8], [1.], nobs=10)
array([[ 1. , 0.8 , 0.64 , 0.512 , 0.4096 , 0.32768 , 0.262144 ,
        0.2097152 , 0.16777216, 0.13421773]])
```

This is the same as

```python
>>> 0.8**np.arange(10)
array([[ 1. , 0.8 , 0.64 , 0.512 , 0.4096 , 0.32768 , 0.262144 ,
        0.2097152 , 0.16777216, 0.13421773]])
```
MA(2)

```python
>>> arma_impulse_response([1.0], [1., 0.5, 0.2], nobs=10)
array([ 1. , 0.5, 0.2, 0. , 0. , 0. , 0. , 0. , 0. , 0. ])
```

ARMA(1,2)

```python
>>> arma_impulse_response([1.0, -0.8], [1., 0.5, 0.2], nobs=10)
array([ 1. , 1.3 , 1.24 , 0.992 , 0.7936 , 0.63488 , 0.507904 , 0.4063232
, 0.32505856, 0.26004685])
```

**statsmodels.tsa.arima_process.arma_pacf**

Partial autocorrelation function of an ARMA process

```python
statsmodels.tsa.arima_process.arma_pacf(ar, ma, nobs=10)
```

- **Parameters**
  - `ar`: array_like, 1d
    - Coefficient for autoregressive lag polynomial, including zero lag
  - `ma`: array_like, 1d
    - Coefficient for moving-average lag polynomial, including zero lag
  - `nobs`: int
    - Number of terms (lags plus zero lag) to include in returned pacf

- **Returns**
  - `pacf`: array
    - Partial autocorrelation of ARMA process given by ar, ma

**Notes**

Solves Yule-Walker equation for each lag order up to nobs lags

Not tested/checked yet

**statsmodels.tsa.arima_process.arma_periodogram**

Periodogram for ARMA process given by lag-polynomials ar and ma

```python
statsmodels.tsa.arima_process.arma_periodogram(ar, ma, worN=None, whole=0)
```

- **Parameters**
  - `ar`: array_like
    - Autoregressive lag-polynomial with leading 1 and lhs sign
  - `ma`: array_like
    - Moving average lag-polynomial with leading 1
  - `worN`: {None, int}, optional
    - Option for scipy.signal.freqz (read “w or N”) If None, then compute at 512 frequencies around the unit circle. If a single integer, the compute at that many frequencies. Otherwise, compute the response at frequencies given in worN
  - `whole`: {0, 1}, optional
    - Options for scipy.signal.freqz Normally, frequencies are computed from 0 to pi (upper-half of unit-circle. If whole is non-zero compute frequencies from 0 to 2*pi.
Returns  

w : array

frequencies

sd : array

periodogram, spectral density

Notes

Normalization ?

This uses signal.freqz, which does not use fft. There is a fft version somewhere.

statsmodels.tsa.arima_process.deconvolve

statsmodels.tsa.arima_process.deconvolve(num, den, n=None)

Deconvolves divisor out of signal, division of polynomials for n terms

calculates den^{-1} * num

Parameters  

num : array_like

signal or lag polynomial

denom : array_like

coefficients of lag polynomial (linear filter)

n : None or int

number of terms of quotient

Returns  

quot : array

quotient or filtered series

rem : array

remainder

Notes

If num is a time series, then this applies the linear filter den^{-1}. If both num and den are both lagpolynomials, then this calculates the quotient polynomial for n terms and also returns the remainder.

This is copied from scipy.signal.signaltools and added n as optional parameter.

statsmodels.tsa.arima_process.index2pol

statsmodels.tsa.arima_process.index2pol(coeffs, index)

expand coefficients to lag poly

Parameters  

coeffs : array

non-zero coefficients of lag polynomial

index : array

index (lags) of lagpolynomial with non-zero elements
\textbf{ar}: array_like
coefficients of lag polynomial

**Returns ar**: array_like
coefficients of lag polynomial

\texttt{statsmodels.tsa.arima_process.lpol2index}

\texttt{statsmodels.tsa.arima_process.lpol2index\(\text{ar}\)}
remove zeros from lagpolynomial, squeezed representation with index

**Parameters ar**: array_like
coefficients of lag polynomial

**Returns coeffs**: array
non-zero coefficients of lag polynomial

**index**: array
index (lags) of lagpolynomial with non-zero elements

\texttt{statsmodels.tsa.arima_process.lpol_fiar}

\texttt{statsmodels.tsa.arima_process.lpol_fiar\(d, n=20\)}
AR representation of fractional integration

\[(1 - L)^d f or |d| < 0.5or|d| < 1(?)\]

**Parameters d**: float
fractional power

**n**: int
number of terms to calculate, including lag zero

**Returns ar**: array
coefficients of lag polynomial

**Notes**: first coefficient is 1, negative signs except for first term,

\texttt{ar(L)*x_t}:

\texttt{statsmodels.tsa.arima_process.lpol_fima}

\texttt{statsmodels.tsa.arima_process.lpol_fima\(d, n=20\)}
MA representation of fractional integration

\[(1 - L)^{-d} f or |d| < 0.5or|d| < 1(?)\]

**Parameters d**: float
fractional power

\[ n : \text{int} \]

number of terms to calculate, including lag zero

**Returns**

\[ \text{ma} : \text{array} \]

coefficients of lag polynomial

### statsmodels.tsa.arima_process.lpol_sdiff

stattmodels.tsa.arima_process.lpol_sdiff\( (s) \)

return coefficients for seasonal difference \((1-L^s)\)

just a trivial convenience function

**Parameters**

\[ s : \text{int} \]

number of periods in season

**Returns**

\[ \text{sdiff} : \text{list, length } s+1 \]

### sandbox.tsa.fftarma.ArmaFft

class sandbox.tsa.fftarma.ArmaFft\( (ar, ma, n) \)

fft tools for arma processes

This class contains several methods that are providing the same or similar returns to try out and test different implementations.

**Notes**

TODO: check whether we don’t want to fix maxlags, and create new instance if maxlag changes. usage for different lengths of timeseries ? or fix frequency and length for fft

check default frequencies w, terminology norw n_or_w

some ffts are currently done without padding with zeros

returns for spectral density methods needs checking, is it always the power spectrum \(hw*hw.conj()\)

normalization of the power spectrum, spectral density: not checked yet, for example no variance of underlying process is used

**Methods**

- \( \text{acf}([\text{nobs}]) \)
  theoretical autocorrelation function of an ARMA process

- \( \text{acf2spdfreq}([\text{acovf}, \text{nfreq}, \text{w}]) \)
  not really a method

- \( \text{acovf}([\text{nobs}]) \)
  theoretical autocovariance function of ARMA process

- \( \text{ar_roots}() \)
  roots of autoregressive lag-polynomial

- \( \text{arma2ar}([\text{nobs}]) \)

- \( \text{arma2ma}([\text{nobs}]) \)
  Continued on next page
### Table 3.17 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fftar([n])</code></td>
<td>Fourier transform of AR polynomial, zero-padded at end to n</td>
</tr>
<tr>
<td><code>fftarma([n])</code></td>
<td>Fourier transform of ARMA polynomial, zero-padded at end to n</td>
</tr>
<tr>
<td><code>fftma(n)</code></td>
<td>Fourier transform of MA polynomial, zero-padded at end to n</td>
</tr>
<tr>
<td><code>filter(x)</code></td>
<td>Filter a timeseries with the ARMA filter</td>
</tr>
<tr>
<td><code>filter2(x[, pad])</code></td>
<td>Filter a time series using <code>fftcconvolve3</code> with ARMA filter</td>
</tr>
<tr>
<td><code>from_coeffs(arcoefs, macoefs[, nobs])</code></td>
<td>Create ArmaProcess instance from coefficients of the lag-polynomials</td>
</tr>
<tr>
<td><code>from_estimation(model_results[, nobs])</code></td>
<td>Create ArmaProcess instance from estimation results</td>
</tr>
<tr>
<td><code>generate_sample([size, scale, distrvs, ...])</code></td>
<td>Generate ARMA samples</td>
</tr>
<tr>
<td><code>impulse_response([nobs])</code></td>
<td>Get the impulse response function (MA representation) for ARMA process</td>
</tr>
<tr>
<td><code>invroots([retnew])</code></td>
<td>Make MA polynomial invertible by inverting roots inside unit circle</td>
</tr>
<tr>
<td><code>invpowerspd(n)</code></td>
<td>Autocovariance from spectral density</td>
</tr>
<tr>
<td><code>isinvertible()</code></td>
<td>Arma process is invertible if MA roots are outside unit circle</td>
</tr>
<tr>
<td><code>isstationary()</code></td>
<td>Arma process is stationary if AR roots are outside unit circle</td>
</tr>
<tr>
<td><code>ma_roots()</code></td>
<td>Roots of moving average lag-polynomial</td>
</tr>
<tr>
<td><code>pacf([nobs])</code></td>
<td>Partial autocorrelation function of an ARMA process</td>
</tr>
<tr>
<td><code>pad(maxlag)</code></td>
<td>Partial autocorrelation function of an ARMA process</td>
</tr>
<tr>
<td><code>padarz(arr, maxlag[, atend])</code></td>
<td>Construct AR and MA polynomials that are zero-padded to a common length</td>
</tr>
<tr>
<td><code>periodogram([nobs])</code></td>
<td>Power spectral density using fftshift</td>
</tr>
<tr>
<td><code>plot4([fig, nobs, nacf, nfreq])</code></td>
<td>Raw spectral density, returns Fourier transform</td>
</tr>
<tr>
<td><code>spd(npos)</code></td>
<td>Power spectral density using padding to length n done by <code>fft</code></td>
</tr>
<tr>
<td><code>spddirect(n)</code></td>
<td>Power spectral density using padding to length n done by <code>fft</code></td>
</tr>
<tr>
<td><code>spdmopoly(w[, twosided])</code></td>
<td>Spectral density from MA polynomial representation for ARMA process</td>
</tr>
<tr>
<td><code>spdpoly(w[, nma])</code></td>
<td>Spectral density for frequency using polynomial roots</td>
</tr>
<tr>
<td><code>spdroots(w)</code></td>
<td>Spectral density for frequency using polynomial roots</td>
</tr>
<tr>
<td><code>spdroots_(arroots, maroots, w)</code></td>
<td>Power spectral density using <code>fftshift</code></td>
</tr>
</tbody>
</table>

#### statsmodels.sandbox.tsa.fftarma.ArmaFft.acf

```python
ArmaFft.acf(nobs=None)
```

Theoretical autocorrelation function of an ARMA process.

**Parameters**
- `ar` : array_like, 1d
  - Coefficient for autoregressive lag polynomial, including zero lag
- `ma` : array_like, 1d
  - Coefficient for moving-average lag polynomial, including zero lag
- `nobs` : int
  - Number of terms (lags plus zero lag) to include in returned acf

**Returns**
- `acf` : array
  - Autocorrelation of ARMA process given by `ar`, `ma`

**See also:**
- `arma_acovf`, `acf`, `acovf`

#### statsmodels.sandbox.tsa.fftarma.ArmaFft.acf2spdfreq

```python
ArmaFft.acf2spdfreq(acovf, nfreq=100, w=None)
```

Not really a method just for comparison, not efficient for large `n` or long `acf`

### 3.7. Time Series analysis tsa

[Page 629]
this is also similarly use in tsa.stattools.periodogram with window

```python
statsmodels.sandbox.tsa.fftarma.ArmaFft.acovf
```

ArmaFft.acovf(nobs=None)

theoretical autocovariance function of ARMA process

**Parameters**

- `ar`: array_like, 1d
  coefficient for autoregressive lag polynomial, including zero lag
- `ma`: array_like, 1d
  coefficient for moving-average lag polynomial, including zero lag
- `nobs`: int
  number of terms (lags plus zero lag) to include in returned acovf

**Returns**

- `acovf`: array
  autocovariance of ARMA process given by ar, ma

**See also:**

arma_acf, acovf

**Notes**

Tries to do some crude numerical speed improvements for cases with high persistence. However, this algorithm is slow if the process is highly persistent and only a few autocovariances are desired.

```python
statsmodels.sandbox.tsa.fftarma.ArmaFft.ar_roots
```

ArmaFft.ar_roots()

roots of autoregressive lag-polynomial

```python
statsmodels.sandbox.tsa.fftarma.ArmaFft.arma2ar
```

ArmaFft.arma2ar(nobs=None)

```python
statsmodels.sandbox.tsa.fftarma.ArmaFft.arma2ma
```

ArmaFft.arma2ma(nobs=None)

```python
statsmodels.sandbox.tsa.fftarma.ArmaFft.fftar
```

ArmaFft.fftar(n=None)

Fourier transform of AR polynomial, zero-padded at end to n

**Parameters**

- `n`: int
  length of array after zero-padding

**Returns**

- `fftar`: ndarray
fft of zero-padded ar polynomial

**statsmodels.sandbox.tsa.fftarma.ArmaFft.fftarma**

ArmaFft.fftarma \((n=None)\)

Fourier transform of ARMA polynomial, zero-padded at end to \(n\)

The Fourier transform of the ARMA process is calculated as the ratio of the fft of the MA polynomial divided by the fft of the AR polynomial.

**Parameters**

- \(n\) : int
  
  length of array after zero-padding

**Returns**

- fftarma : ndarray
  
  fft of zero-padded arma polynomial

**statsmodels.sandbox.tsa.fftarma.ArmaFft.fftma**

ArmaFft.fftma \((n)\)

Fourier transform of MA polynomial, zero-padded at end to \(n\)

**Parameters**

- \(n\) : int
  
  length of array after zero-padding

**Returns**

- fftar : ndarray
  
  fft of zero-padded ar polynomial

**statsmodels.sandbox.tsa.fftarma.ArmaFft.filter**

ArmaFft.filter \((x)\)

filter a timeseries with the ARMA filter

padding with zero is missing, in example I needed the padding to get initial conditions identical to direct filter

Initial filtered observations differ from filter2 and signal.lfilter, but at end they are the same.

**See also:**

tsa.filters.fftconvolve

**statsmodels.sandbox.tsa.fftarma.ArmaFft.filter2**

ArmaFft.filter2 \((x, \text{pad}=0)\)

filter a time series using fftconvolve3 with ARMA filter

padding of x currently works only if x is 1d in example it produces same observations at beginning as lfilter even without padding.

TODO: this returns 1 additional observation at the end
ArmaFft.from_coeffs

```python
classmethod ArmaFft.from_coeffs(arcoefs, macoefs, nobs=None)
create ArmaProcess instance from coefficients of the lag-polynomials
```

ArmaFft.from_estimation

```python
classmethod ArmaFft.from_estimation(model_results, nobs=None)
create ArmaProcess instance from estimation results
```

ArmaFft.generate_sample

```python
ArmaFft.generate_sample(size=100, scale=1, distrvs=None, axis=0, burnin=0)
generate ARMA samples
```

**Parameters**

- `size`: int or tuple of ints
  - If size is an integer, then this creates a 1d timeseries of length size. If size is a tuple, then the timeseries is along axis. All other axis have independent arma samples.

**Returns**

- `rvs`: ndarray
  - random sample(s) of arma process

**Notes**

Should work for n-dimensional with time series along axis, but not tested yet. Processes are sampled independently.

ArmaFft.impulse_response

```python
ArmaFft.impulse_response(nobs=None)
get the impulse response function (MA representation) for ARMA process
```

**Parameters**

- `ma`: array_like, 1d
  - moving average lag polynomial
- `ar`: array_like, 1d
  - auto regressive lag polynomial
- `nobs`: int
  - number of observations to calculate

**Returns**

- `ir`: array, 1d
  - impulse response function with nobs elements

**Notes**

This is the same as finding the MA representation of an ARMA(p,q). By reversing the role of ar and ma in the function arguments, the returned result is the AR representation of an ARMA(p,q), i.e
ma_representation = arma_impulse_response(ar, ma, nobs=100) ar_representation = arma_impulse_response(ma, ar, nobs=100)

fully tested against matlab

Examples

AR(1)

```python
>>> arma_impulse_response([1.0, -0.8], [1.], nobs=10)
array([ 1. , 0.8 , 0.64 , 0.512 , 0.4096 ,
       0.32768, 0.262144, 0.2097152, 0.16777216, 0.13421773])
```

this is the same as

```python
>>> 0.8**np.arange(10)
array([ 1. , 0.8 , 0.64 , 0.512 , 0.4096 ,
       0.32768, 0.262144, 0.2097152, 0.16777216, 0.13421773])
```

MA(2)

```python
>>> arma_impulse_response([1.0], [1., 0.5, 0.2], nobs=10)
array([ 1. , 0.5, 0.2, 0. , 0. , 0. , 0. , 0. , 0. , 0. ]) 
```

ARMA(1,2)

```python
>>> arma_impulse_response([1.0, -0.8], [1., 0.5, 0.2], nobs=10)
array([ 1. , 1.3 , 1.24 , 0.992 , 0.7936 ,
       0.63488, 0.507904, 0.4063232, 0.32505856, 0.26004685])
```

---

**statsmodels.sandbox.tsa.fftarma.ArmaFft.invertroots**

ArmaFft.invertroots(retnew=False)

make MA polynomial invertible by inverting roots inside unit circle

**Parameters**

- retnew : boolean
  - If False (default), then return the lag-polynomial as array. If True, then return a new instance with invertible MA-polynomial

**Returns**

- manew : array
  - new invertible MA lag-polynomial, returned if retnew is false.

- wasinvertible : boolean
  - True if the MA lag-polynomial was already invertible, returned if retnew is false.

- armaprocess : new instance of class
  - If retnew is true, then return a new instance with invertible MA-polynomial

---

**statsmodels.sandbox.tsa.fftarma.ArmaFft.invpowerspd**

ArmaFft.invpowerspd(n)

autocovariance from spectral density

scaling is correct, but n needs to be large for numerical accuracy maybe padding with zero in fft would be faster without slicing it returns 2-sided autocovariance with fftshift
>>> ArmaFft([1, -0.5], [1., 0.4], 40).invpowerspd(2**8)[:10]
array([ 2.08 , 1.44 , 0.72 , 0.36 , 0.18 , 0.09 ,
      0.045 , 0.0225 , 0.01125 , 0.005625])

>>> ArmaFft([1, -0.5], [1., 0.4], 40).acovf(10)
array([ 2.08 , 1.44 , 0.72 , 0.36 , 0.18 , 0.09 ,
      0.045 , 0.0225 , 0.01125 , 0.005625])

**statsmodels.sandbox.tsa.fftarma.ArmaFft.isinvertible**

ArmaFft.isinvertible()  
Arma process is invertible if MA roots are outside unit circle  
**Returns** isinvertible : boolean  
True if moving average roots are outside unit circle

**statsmodels.sandbox.tsa.fftarma.ArmaFft.isstationary**

ArmaFft.isstationary()  
Arma process is stationary if AR roots are outside unit circle  
**Returns** isstationary : boolean  
True if autoregressive roots are outside unit circle

**statsmodels.sandbox.tsa.fftarma.ArmaFft.ma_roots**

ArmaFft.ma_roots()  
roots of moving average lag-polynomial

**statsmodels.sandbox.tsa.fftarma.ArmaFft.pacf**

ArmaFft.pacf(nobs=None)  
partial autocorrelation function of an ARMA process  
**Parameters**  
ar : array_like, 1d  
coefficient for autoregressive lag polynomial, including zero lag  
ma : array_like, 1d  
coefficient for moving-average lag polynomial, including zero lag  
nobs : int  
number of terms (lags plus zero lag) to include in returned pacf  
**Returns** pacf : array  
partial autocorrelation of ARMA process given by ar, ma  

**Notes**

solves yule-walker equation for each lag order up to nobs lags  
not tested/checked yet
ArmaFft.pad(maxlag)

construct AR and MA polynomials that are zero-padded to a common length

Parameters maxlag : int
    new length of lag-polynomials

Returns ar : ndarray
    extended AR polynomial coefficients

ma : ndarray
    extended AR polynomial coefficients

ArmaFft.padarr(arr, maxlag, atend=True)

pad 1d array with zeros at end to have length maxlag function that is a method, no self used

Parameters arr : array_like, 1d
    array that will be padded with zeros

maxlag : int
    length of array after padding

atend : boolean
    If True (default), then the zeros are added to the end, otherwise to the front of the array

Returns arp : ndarray
    zero-padded array

Notes

This is mainly written to extend coefficient arrays for the lag-polynomials. It returns a copy.

ArmaFft.periodogram(nobs=None)

periodogram for ARMA process given by lag-polynomials ar and ma

Parameters ar : array_like
    autoregressive lag-polynomial with leading 1 and lhs sign

ma : array_like
    moving average lag-polynomial with leading 1

worN : {None, int}, optional
    option for scipy.signal.freqz (read “w or N”) If None, then compute at 512 frequencies around the unit circle. If a single integer, the compute at that many frequencies. Otherwise, compute the response at frequencies given in worN
whole : {0,1}, optional

options for scipy.signal.freqz Normally, frequencies are computed from 0 to pi (upper-
half of unit circle). If whole is non-zero compute frequencies from 0 to 2*pi.

Returns w : array
    frequencies
sd : array
    periodogram, spectral density

Notes

Normalization ?
This uses signal.freqz, which does not use fft. There is a fft version somewhere.

```
statsmodels.sandbox.tsa.fftarma.ArmaFft.plot4
```

```
ArmaFft.plot4(fig=None, nobs=100, nacf=20, nfreq=100)
```

```
statsmodels.sandbox.tsa.fftarma.ArmaFft.spd
```

```
ArmaFft.spd(npos)
```

raw spectral density, returns Fourier transform
n is number of points in positive spectrum, the actual number of points is twice as large. different from
other spd methods with fft

```
statsmodels.sandbox.tsa.fftarma.ArmaFft.spddirect
```

```
ArmaFft.spddirect(n)
```

two-sided according to fft frequencies, use first half

```
statsmodels.sandbox.tsa.fftarma.ArmaFft.spdmapoly
```

```
ArmaFft.spdmapoly(w, twosided=False)
```

ma only, need division for ar, use LagPolynomial

```
statsmodels.sandbox.tsa.fftarma.ArmaFft.spdpoly
```

```
ArmaFft.spdpoly(w, nma=50)
```

spectral density from MA polynomial representation for ARMA process

```
References
```

Cochrane, section 8.3.3
statsmodels.sandbox.tsa.fftarma.ArmaFft.spdroots

ArmaFft.spdroots(w)
spectral density for frequency using polynomial roots
builds two arrays (number of roots, number of frequencies)

statsmodels.sandbox.tsa.fftarma.ArmaFft.spdroots

ArmaFft.spdroots_(arroots, maroots, w)
spectral density for frequency using polynomial roots
builds two arrays (number of roots, number of frequencies)

   Parameters arroots : ndarray
                        roots of ar (denominator) lag-polynomial
                        maroots : ndarray
                                  roots of ma (numerator) lag-polynomial
                           w : array_like
                           frequencies for which spd is calculated

Notes
this should go into a function

statsmodels.sandbox.tsa.fftarma.ArmaFft.spdshift

ArmaFft.spdshift(n)
power spectral density using fftshift
currently returns two-sided according to fft frequencies, use first half

3.7.5 Time Series Filters

filters.bk_filter.bkfilter(X[, low, high, K]) Baxter-King bandpass filter
filters.hp_filter.hpfilter(X[, lamb]) Hodrick-Prescott filter
filters.cf_filter.cffilter(X[, low, high, drift])
                      Christiano Fitzgerald asymmetric, random walk filter
filters.filtertools.convolution_filter(x, filt) Linear filtering via convolution.
filters.filtertools.recursive_filter(x, ar_coeff) Autoregressive, or recursive, filtering.
filters.filtertools.miso_lfilter(ar, ma, x) use nd convolution to merge inputs,
filters.filtertools.fftconvolve3(in1[, in2, ...]) Convolves two N-dimensional arrays using FFT. See convolve.
filters.filtertools.fftconvolveinv(in1, in2) Convolves two N-dimensional arrays using FFT. See convolve.

statsmodels.tsa.filters.bk_filter.bkfilter

statsmodels.tsa.filters.bk_filter.bkfilter(X, low=6, high=32, K=12)
Baxter-King bandpass filter

   Parameters X : array-like

3.7. Time Series analysis tsa
A 1 or 2d ndarray. If 2d, variables are assumed to be in columns.

**low** : float

Minimum period for oscillations, i.e., Baxter and King suggest that the Burns-Mitchell U.S. business cycle has 6 for quarterly data and 1.5 for annual data.

**high** : float

Maximum period for oscillations BK suggest that the U.S. business cycle has 32 for quarterly data and 8 for annual data.

**K** : int

Lead-lag length of the filter. Baxter and King propose a truncation length of 12 for quarterly data and 3 for annual data.

**Returns**

**Y** : array

Cyclical component of X

**Notes**

Returns a centered weighted moving average of the original series. Where the weights a[j] are computed

\[ a[j] = b[j] + \theta, \text{ for } j = 0, +/-1, +/-2, \ldots +/- K \]

\[ b[0] = \frac{(\omega_2 - \omega_1)}{\pi} \]

\[ b[j] = \frac{1}{\pi j} \left( \sin(\omega_2 j) - \sin(\omega_1 j) \right), \text{ for } j = +/-1, +/-2, \ldots \]

and \( \theta \) is a normalizing constant

\[ \theta = -\text{sum}(b)/(2K+1) \]

**References**


**Examples**

```python
>>> import statsmodels.api as sm
>>> import pandas as pd

>>> dta = sm.datasets.macrodata.load_pandas().data
>>> dates = sm.tsa.datetools.dates_from_range('1959Q1', '2009Q3')
>>> index = pd.DatetimeIndex(dates)
>>> dta.set_index(index, inplace=True)

>>> cycles = sm.tsa.filters.bkfilter(dta[['realinv']], 6, 24, 12)

>>> import matplotlib.pyplot as plt

>>> fig, ax = plt.subplots()
>>> cycles.plot(ax=ax, style=['r--', 'b-'])
>>> plt.show()
```
Hodrick-Prescott filter

**Parameters**

- **X**: array-like
  
The 1d ndarray timeseries to filter of length (nobs,) or (nobs,1)

- **lamb**: float
  
The Hodrick-Prescott smoothing parameter. A value of 1600 is suggested for quarterly data. Ravn and Uhlig suggest using a value of 6.25 \((1600/4^{**4})\) for annual data and \(129600 (1600*3^{**4})\) for monthly data.

**Returns**

- **cycle**: array
  
The estimated cycle in the data given lamb.

- **trend**: array
  
The estimated trend in the data given lamb.

**Notes**

The HP filter removes a smooth trend, \(T\), from the data \(X\) by solving

\[
\min \sum (X[t] - T[t])^{**2} + \text{lamb} \times ((T[t+1] - T[t]) - (T[t] - T[t-1]))^{**2} \quad T[t]
\]
Here we implemented the HP filter as a ridge-regression rule using scipy.sparse. In this sense, the solution can be written as

$$T = \text{inv}(I - \lambda \mathbf{K}'\mathbf{K})\mathbf{X}$$

where $I$ is a $n_{\text{obs}} \times n_{\text{obs}}$ identity matrix, and $\mathbf{K}$ is a $(n_{\text{obs}}-2) \times n_{\text{obs}}$ matrix such that

$$K[i,j] = 1 \text{ if } i == j \text{ or } i == j + 2 \quad \quad K[i,j] = -2 \text{ if } i == j + 1 \quad \quad K[i,j] = 0 \text{ otherwise}$$

**References**


**Examples**

```python
>>> import statsmodels.api as sm
>>> import pandas as pd
>>> dta = sm.datasets.macrodata.load_pandas().data
>>> dates = sm.tsa.datetools.dates_from_range('1959Q1', '2009Q3')
>>> index = pd.DatetimeIndex(dates)
>>> dta.set_index(index, inplace=True)
>>> cycle, trend = sm.tsa.filters.hpfilter(dta.realgdp, 1600)
>>> gdp_decomp = dta[['realgdp']]
>>> gdp_decomp['cycle'] = cycle
>>> gdp_decomp['trend'] = trend

>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> gdp_decomp[['realgdp', 'trend']]['2000-03-31':].plot(ax=ax,...  fontsize=16);
>>> plt.show()
```
statsmodels.tsa.filters.cf_filter.cffilter

statsmodels.tsa.filters.cf_filter.cffilter \(X, \text{low}=6, \text{high}=32, \text{drift}=True\)

Christiano Fitzgerald asymmetric, random walk filter

**Parameters**

- **X**: array-like
  1 or 2d array to filter. If 2d, variables are assumed to be in columns.

- **low**: float
  Minimum period of oscillations. Features below low periodicity are filtered out. Default is 6 for quarterly data, giving a 1.5 year periodicity.

- **high**: float
  Maximum period of oscillations. Features above high periodicity are filtered out. Default is 32 for quarterly data, giving an 8 year periodicity.

- **drift**: bool
  Whether or not to remove a trend from the data. The trend is estimated as \(\text{np.arange(nobs)}*(X[-1]-X[0])/(\text{len}(X)-1)\)

**Returns**

- **cycle**: array
  The features of \(X\) between periodicities given by low and high

- **trend**: array
  The features of \(X\) after removing the trend.
The trend in the data with the cycles removed.

Examples

```python
>>> import statsmodels.api as sm
>>> import pandas as pd
>>> dta = sm.datasets.macrodata.load_pandas().data
>>> dates = sm.tsa.datetools.dates_from_range('1959Q1', '2009Q3')
>>> index = pd.DatetimeIndex(dates)
>>> dta.set_index(index, inplace=True)
>>> cf_cycles, cf_trend = sm.tsa.filters.cffilter(dta[['infl', 'unemp']])

>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> cf_cycles.plot(ax=ax, style=['r--', 'b-'])
>>> plt.show()
```

![Graph showing trend and cycles removed from data](image)

`statsmodels.tsa.filters.filtertools.convolution_filter`

`statsmodels.tsa.filters.filtertools.convolution_filter(x, filt, nsides=2)`

Linear filtering via convolution. Centered and backward displaced moving weighted average.

**Parameters**

- `x`: array_like
  
  data array, 1d or 2d, if 2d then observations in rows
**filt** : array_like

Linear filter coefficients in reverse time-order. Should have the same number of dimensions as x though if 1d and x is 2d will be coerced to 2d.

**nsides** : int, optional

If 2, a centered moving average is computed using the filter coefficients. If 1, the filter coefficients are for past values only. Both methods use scipy.signal.convolve.

**Returns**

**y** : ndarray, 2d

Filtered array, number of columns determined by x and filt. If a pandas object is given, a pandas object is returned. The index of the return is the exact same as the time period in x.

**Notes**

In nsides == 1, x is filtered

\[ y[n] = filt[0] \times x[n-1] + \ldots + filt[n_{filt}-1] \times x[n-n_{filt}] \]

where \( n_{filt} \) is \( \text{len(filt)} \).

If nsides == 2, x is filtered around lag 0

\[ y[n] = filt[0] \times [n - n_{filt}/2] + \ldots + filt[n_{filt}/2] \times x[n] + \ldots + x[n + n_{filt}/2] \]

where \( n_{filt} \) is \( \text{len(filt)} \). If \( n_{filt} \) is even, then more of the filter is forward in time than backward.

If filt is 1d or (nlags,1) one lag polynomial is applied to all variables (columns of x). If filt is 2d, (nlags, nvars) each series is independently filtered with its own lag polynomial, uses loop over nvar. This is different than the usual 2d vs 2d convolution.

Filtering is done with scipy.signal.convolve, so it will be reasonably fast for medium sized data. For large data fft convolution would be faster.

**statsmodels.tsa.filters.filtertools.recursive_filter**

**statsmodels.tsa.filters.filtertools.recursive_filter** (x, ar_coeff, init=None)

Autoregressive, or recursive, filtering.

**Parameters**

**x** : array-like

Time-series data. Should be 1d or n x 1.

**ar_coeff** : array-like

AR coefficients in reverse time order. See Notes

**init** : array-like

Initial values of the time-series prior to the first value of y. The default is zero.

**Returns**

**y** : array

Filtered array, number of columns determined by x and ar_coeff. If a pandas object is given, a pandas object is returned.
**Notes**

Computes the recursive filter

\[ y[n] = \ar_coeff[0] \times y[n-1] + \ldots \\
+ \ar_coeff[n\_coeff - 1] \times y[n - n\_coeff] + x[n] \]

where \( n\_coeff = \text{len}(\arcoeff) \).

**statsmodels.tsa.filters.filtertools.miso_lfilter**

Use nd convolution to merge inputs, then use lfilter to produce output

Arguments for column variables return currently 1d

**Parameters**

- \( \ar \): array_like, 1d, float
  - autoregressive lag polynomial including lag zero, \( \ar(L) y_t \)
- \( \ma \): array_like, same ndim as \( x \), currently 2d
  - moving average lag polynomial \( \ma(L) x_t \)
- \( x \): array_like, 2d
  - input data series, time in rows, variables in columns

**Returns**

- \( y \): array, 1d
  - filtered output series
- \( \text{inp} \): array, 1d
  - combined input series

**Notes**

Currently for 2d inputs only, no choice of axis. Use of signal.lfilter requires that \( \ar \) lag polynomial contains floating point numbers does not cut off invalid starting and final values.

\( \text{miso}_\text{lfilter} \) find array \( y \) such that:

\[ \ar(L) y_t = \ma(L) x_t \]

with shapes \( y \) (nobs,), \( x \) (nobs,nvars), \( \ar \) (narlags,), \( \ma \) (narlags,nvars)

**statsmodels.tsa.filters.filtertools.fftconvolve3**

Convolve two N-dimensional arrays using FFT. See `convolve`.

For use with `arma` (old version: `in1=num` `in2=den` `in3=data`)

- Better for consistency with other functions `in1=data` `in2=num` `in3=den`
- Note `in2` and `in3` need to have consistent dimension/shape since I’m using max of `in2`, `in3` shapes and not the sum
copied from scipy.signal.signaltools, but here used to try out inverse filter doesn’t work or I can’t get it to work
2010-10-23 looks ok to me for 1d, from results below with padded data array (fftp) but it doesn’t work for
multidimensional inverse filter (fftn) original signal.fftconvolve also uses fftn

```
statsmodels.tsa.filters.filtertools.fftconvolveinv
```

```
statsmodels.tsa.filters.filtertools.fftconvolveinv(in1, in2, mode='full')
    Convolve two N-dimensional arrays using FFT. See convolve.
```

copied from scipy.signal.signaltools, but here used to try out inverse filter doesn’t work or I can’t get it to work
2010-10-23: looks ok to me for 1d, from results below with padded data array (fftp) but it doesn’t work for
multidimensional inverse filter (fftn) original signal.fftconvolve also uses fftn

### 3.7.6 TSA Tools

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tsatools.add_constant(data[, prepend])</code></td>
<td>This appends a column of ones to an array if prepend==False.</td>
</tr>
<tr>
<td><code>tsatools.add_trend(X[, trend, prepend])</code></td>
<td>Adds a trend and/or constant to an array.</td>
</tr>
<tr>
<td><code>tsatools.detrend(x[, order, axis])</code></td>
<td>detrend an array with a trend of given order along axis 0 or 1</td>
</tr>
<tr>
<td><code>tsatools.lagmat(x, maxlag[, trim, original])</code></td>
<td>create 2d array of lags</td>
</tr>
<tr>
<td><code>tsatools.lagmat2ds(x, maxlag0[, maxlagex, ...])</code></td>
<td>generate lagmatrix for 2d array, columns arranged by variables</td>
</tr>
</tbody>
</table>

### 3.7.6.1 Statsmodels.tsa.tsatools.add_constant

```
statsmodels.tsa.tsatools.add_constant(data[, prepend])
```

This appends a column of ones to an array if prepend==False.

For ndarrays and pandas.DataFrames, checks to make sure a constant is not already included. If there is at least
one column of ones then the original object is returned. Does not check for a constant if a structured or recarray
is given.

**Parameters**

- **data** : array-like
  - `data` is the column-ordered design matrix
- **prepend** : bool
  - True and the constant is prepended rather than appended.

**Returns**

- **data** : array
  - The original array with a constant (column of ones) as the first or last column.

### 3.7.6.2 Statsmodels.tsa.tsatools.add_trend

```
statsmodels.tsa.tsatools.add_trend(X[, trend, prepend])
```

Adds a trend and/or constant to an array.

**Parameters**

- **X** : array-like
  - Original array of data.
- **trend** : str ("c","t","ct","ctt")
  - “c” add constant only “t” add trend only “ct” add constant and linear trend “ctt” add constant and linear and quadratic trend.
prepend : bool

If True, prepends the new data to the columns of X.

See also:
statsmodels.add_constant

Notes

Returns columns as ["ctt","ct","c"] whenever applicable. There is currently no checking for an existing constant or trend.

statsmodels.tsa.tsatools.detrend

statsmodels.tsa.tsatools.detrend(x, order=1, axis=0)
detrend an array with a trend of given order along axis 0 or 1

Parameters x : array_like, 1d or 2d
data, if 2d, then each row or column is independently detrended with the same trend-order, but independent trend estimates

order : int
specifies the polynomial order of the trend, zero is constant, one is linear trend, two is quadratic trend

axis : int
for detrending with order > 0, axis can be either 0 observations by rows, or 1, observations by columns

Returns detrended data series : ndarray
The detrended series is the residual of the linear regression of the data on the trend of given order.

statsmodels.tsa.tsatools.lagmat

statsmodels.tsa.tsatools.lagmat(x, maxlag, trim='forward', original='ex')
create 2d array of lags

Parameters x : array_like, 1d or 2d
data; if 2d, observation in rows and variables in columns

maxlag : int or sequence of ints
all lags from zero to maxlag are included

trim : str {'forward', 'backward', 'both', 'none'} or None
- ‘forward’ : trim invalid observations in front
- ‘backward’ : trim invalid initial observations
- ‘both’ : trim invalid observations on both sides
- ‘none’, None : no trimming of observations

original : str {'ex','sep','in'}
• ‘ex’ : drops the original array returning only the lagged values.
• ‘in’ : returns the original array and the lagged values as a single array.
• ‘sep’ [returns a tuple (original array, lagged values). The original] array is truncated to have
  the same number of rows as the returned lagmat.

Returns  lagmat : 2d array
  array with lagged observations
y : 2d array, optional
  Only returned if original == ‘sep’

Notes

TODO: * allow list of lags additional to maxlag * create varnames for columns

Examples

```python
>>> from statsmodels.tsa.tsatools import lagmat
>>> import numpy as np
>>> X = np.arange(1,7).reshape(-1,2)
```

```python
>>> lagmat(X, maxlag=2, trim="forward", original='in')
array([[ 1.,  2.,  0.,  0.,  0.,  0.],
       [ 3.,  4.,  1.,  2.,  0.,  0.],
       [ 5.,  6.,  3.,  4.,  1.,  2.]])
```

```python
>>> lagmat(X, maxlag=2, trim="backward", original='in')
array([[ 5.,  6.,  3.,  4.,  1.,  2.],
       [ 0.,  0.,  5.,  6.,  3.,  4.],
       [ 0.,  0.,  0.,  0.,  5.,  6.]])
```

```python
>>> lagmat(X, maxlag=2, trim="both", original='in')
array([[ 5.,  6.,  3.,  4.,  1.,  2.]])
```

```python
>>> lagmat(X, maxlag=2, trim="none", original='in')
array([[ 1.,  2.,  0.,  0.,  0.,  0.],
       [ 3.,  4.,  1.,  2.,  0.,  0.],
       [ 5.,  6.,  3.,  4.,  1.,  2.],
       [ 0.,  0.,  5.,  6.,  3.,  4.],
       [ 0.,  0.,  0.,  0.,  5.,  6.]])
```

statsmodels.tsa.tsatools.lagmat2ds

statsmodels.tsa.tsatools.lagmat2ds(x, maxlag0, maxlagex=None, dropex=0, trim='forward')
generate lagmatrix for 2d array, columns arranged by variables

Parameters x : array_like, 2d
  2d data, observation in rows and variables in columns
maxlag0 : int
  for first variable all lags from zero to maxlag are included
maxlagex : None or int

3.7. Time Series analysis tsa
max lag for all other variables all lags from zero to maxlag are included

**dropex** : int (default is 0)

exclude first dropex lags from other variables for all variables, except the first, lags from dropex to maxlagex are included

**trim** : string

- ‘forward’ : trim invalid observations in front
- ‘backward’ : trim invalid initial observations
- ‘both’ : trim invalid observations on both sides
- ‘none’ : no trimming of observations

**Returns**

**lagmat** : 2d array

array with lagged observations, columns ordered by variable

**Notes**

very inefficient for unequal lags, just done for convenience

### 3.7.7 VARMA Process

```python
varma_process.VarmaPoly(ar[, ma]) class to keep track of Varma polynomial format
```

#### class statsmodels.tsa.varma_process.VarmaPoly

class to keep track of Varma polynomial format

**Examples**

```python
ar23 = np.array([[1., 0.],
[0., 1.]],
[[0.6, 0.], [0.2, 0.6]],
[[1, 0.], [0.1, 0.1]])

ma22 = np.array([[1., 0.],
[0., 1.]],
[[0.4, 0.], [0.2, 0.3]])
```

**Methods**

```python
getisinvertible(a) check whether the auto-regressive lag-polynomial is stationary
```

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getisstationary(a)</code></td>
<td>check whether the auto-regressive lag-polynomial is stationary</td>
</tr>
<tr>
<td><code>hstack([a, name])</code></td>
<td>stack lagpolynomial horizontally in 2d array</td>
</tr>
<tr>
<td><code>hstackarma_minus1()</code></td>
<td>stack ar and lagpolynomial vertically in 2d array</td>
</tr>
<tr>
<td><code>reduceform(apoly)</code></td>
<td>this assumes no exog. todo</td>
</tr>
<tr>
<td><code>stacksquare([a, name, orientation])</code></td>
<td>stack lagpolynomial vertically in 2d square array with eye</td>
</tr>
<tr>
<td><code>vstack([a, name])</code></td>
<td>stack lagpolynomial vertically in 2d array</td>
</tr>
<tr>
<td><code>vstackarma_minus1()</code></td>
<td>stack ar and lagpolynomial vertically in 2d array</td>
</tr>
</tbody>
</table>

**statsmodels.tsa.varma_process.VarmaPoly.getisinvertible**

`VarmaPoly.getisinvertible(a=None)`
check whether the auto-regressive lag-polynomial is stationary

- **Returns** `isinvertible` : boolean
- *attaches* :
  - `maeigenvalues` : complex array
eigenvalues sorted by absolute value

**References**
formula taken from NAG manual

**statsmodels.tsa.varma_process.VarmaPoly.getisstationary**

`VarmaPoly.getisstationary(a=None)`
check whether the auto-regressive lag-polynomial is stationary

- **Returns** `isstationary` : boolean
- *attaches* :
  - `areigenvalues` : complex array
eigenvalues sorted by absolute value

**References**
formula taken from NAG manual

**statsmodels.tsa.varma_process.VarmaPoly.hstack**

`VarmaPoly.hstack(a=None, name='ar')`
stack lagpolynomial horizontally in 2d array
3.7.8 Interpolation

**interp.denton.dentonm**(indicator, benchmark)  Modified Denton’s method to convert low-frequency to high-frequency data.

**Parameters**

- **indicator**: A low-frequency indicator series. It is assumed that there are no pre-sample indicators. I.e., the first indicators line up with the first benchmark.
- **benchmark**: array-like
  The higher frequency benchmark. A 1d or 2d data series in columns. If 2d, then M series are assumed.
- **freq**: str ("aq", "qm", "other")
  - “aq” - Benchmarking an annual series to quarterly.
  - “qm” - Benchmarking a quarterly series to monthly.
  - “other” - Custom stride. A kwarg, k, must be supplied.
kwarggs :

    k [int] The number of high-frequency observations that sum to make an aggregate low-frequency observation. k is used with freq == "other".

Returns :

    ——— :

    benchmarked series : array

Notes

Denton’s method minimizes the distance given by the penalty function, in a least squares sense, between the unknown benchmarked series and the indicator series subject to the condition that the sum of the benchmarked series is equal to the benchmark. The modification allows that the first value not be pre-determined as is the case with Denton’s original method. If there is no benchmark provided for the last few indicator observations, then extrapolation is performed using the last benchmark-indicator ratio of the previous period.

Minimizes \( \sum((X[t]/I[t] - X[t-1]/I[t-1])**2) \)

s.t.

\( \sum(X) = A, \) for each period. Where \( X \) is the benchmarked series, \( I \) is the indicator, and \( A \) is the benchmark.

References


Examples

```python
>>> indicator = [50,100,150,100] * 5
>>> benchmark = [500,400,300,400,500]
>>> benchmarked = dentonm(indicator, benchmark, freq="aq")
```

3.8 Statistics

This section collects various statistical tests and tools. Some can be used independently of any models, some are intended as extension to the models and model results.

API Warning: The functions and objects in this category are spread out in various modules and might still be moved around. We expect that in future the statistical tests will return class instances with more informative reporting instead of only the raw numbers.

3.8.1 Residual Diagnostics and Specification Tests
**statsmodels Documentation, Release 0.6.0**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>durbin_watson(resids[, axis])</code></td>
<td>Calculates the Durbin-Watson statistic</td>
</tr>
<tr>
<td><code>jarque_bera(resids[, axis])</code></td>
<td>Calculate residual skewness, kurtosis, and do the JB test for normality</td>
</tr>
<tr>
<td><code>omni_normtest(resids[, axis])</code></td>
<td>Omnibus test for normality</td>
</tr>
</tbody>
</table>

### statsmodels.stats.stattools.durbin_watson

**Calculates the Durbin-Watson statistic**

**Parameters**
- `resids`: array-like

**Returns**
- `dw`: float, array-like
  - The Durbin-Watson statistic.

**Notes**

The null hypothesis of the test is that there is no serial correlation. The Durbin-Watson test statistics is defined as:

\[
T \sum_{t=2}^{T} \frac{(e_t - e_{t-1})^2}{\sum_{t=1}^{T} e_t^2}
\]

The test statistic is approximately equal to \(2(1-r)\) where \(r\) is the sample autocorrelation of the residuals. Thus, for \(r = 0\), indicating no serial correlation, the test statistic equals 2. This statistic will always be between 0 and 4. The closer to 0 the statistic, the more evidence for positive serial correlation. The closer to 4, the more evidence for negative serial correlation.

### statsmodels.stats.stattools.jarque_bera

**Calculate residual skewness, kurtosis, and do the JB test for normality**

**Parameters**
- `resids`: array-like
- `axis`: int, optional
  - Default is 0

**Returns**
- `JB`, `JBpv`, `skew`, `kurtosis`:
  - `JB` = \(n/6^{*}(S^2 + (K-3)^2/4)\)
  - `JBpv` is the Chi^2 two-tail probability value
  - `skew` is the measure of skewness
  - `kurtosis` is the measure of kurtosis

### statsmodels.stats.stattools.omni_normtest

**Omnibus test for normality**

**Parameters**
- `resid`: array-like
- `axis`: int, optional
Default is 0

Returns Chi^2 score, two-tail probability:

```python
acorr_ljungbox(x[, lags, boxpierce])
acorr_breush_godfrey(results[, nlags, store])
hetGoldfeldQuandt
het_goldfeldquandt
het_breushpagan(resid, exog_het)
het_white(resid, exog[, retres])
het_arch(resid[, maxlag, autolag, store, ...])
linear_harvey_collier(res)
linear_rainbow(res[, frac])
linear_lm(resid, exog[, func])
breaks_cusumolsresid(olsresidual[, ddof])
breaks_hansen(olsresults)
recursive_olsresiduals(olsresults[, skip, ...])
CompareCox
compare_cox
CompareJ
compare_j
unitroot_adf(x[, maxlag, trendorder, ...])
normal_ad(x[, axis])
kstest_normal(x[, pvalmethod])
lillifors(x[, pvalmethod])
```

Ljung-Box test for no autocorrelation
Breush Godfrey Lagrange Multiplier tests for residual autocorrelation
test whether variance is the same in 2 subsamples
see class docstring
Breush-Pagan Lagrange Multiplier test for heteroscedasticity
White’s Lagrange Multiplier Test for Heteroscedasticity
Enlge’s Test for Autoregressive Conditional Heteroscedasticity (ARCH)
Harvey Collier test for linearity
Rainbow test for linearity
Lagrange multiplier test for linearity against functional alternative
cum test for parameter stability based on ols residuals
test for model stability, breaks in parameters for ols, Hansen 1992
calculate recursive ols with residuals and cum test statistic
Cox Test for non-nested models
Cox Test for non-nested models
J-Test for comparing non-nested models
J-Test for comparing non-nested models
Anderson-Dalring test for normal distribution unknown mean and variance
Lillifors test for normality,
Lillifors test for normality,
p-value based for Box-Pierce test on chi-square distribution

Notes

Ljung-Box and Box-Pierce statistic differ in their scaling of the autocorrelation function. Ljung-Box test is reported to have better small sample properties.

TODO: could be extended to work with more than one series 1d or nd ? axis ? ravel ? needs more testing

‘’Verification‘’

Looks correctly sized in Monte Carlo studies. not yet compared to verified values

References

Greene Wikipedia

Examples

see example script

statsmodels.stats.diagnostic.acorr_breush_godfrey

```python
statsmodels.stats.diagnostic.acorr_breush_godfrey(results, nlags=None, store=False)
```

Breush Godfrey Lagrange Multiplier tests for residual autocorrelation

Parameters

- **results**: Result instance
  - Estimation results for which the residuals are tested for serial correlation
- **nlags**: int
  - Number of lags to include in the auxiliary regression. (nlags is highest lag)
- **store**: bool
  - If store is true, then an additional class instance that contains intermediate results is returned.

Returns

- **lm**: float
  - Lagrange multiplier test statistic
- **impval**: float
  - p-value for Lagrange multiplier test
- **fval**: float
  - fstatistic for F test, alternative version of the same test based on F test for the parameter restriction
- **fpval**: float
  - pvalue for F test
- **resstore**: instance (optional)
  - a class instance that holds intermediate results. Only returned if store=True
Notes

BG adds lags of residual to exog in the design matrix for the auxiliary regression with residuals as endog, see Greene 12.7.1.

References

Greene Econometrics, 5th edition

**statsmodels.stats.diagnostic.HetGoldfeldQuandt**

class statsmodels.stats.diagnostic.HetGoldfeldQuandt

test whether variance is the same in 2 subsamples

**Parameters**  
  
y : array_like  
  endogenous variable  
  
x : array_like  
  exogenous variable, regressors  
  
idx : integer  
  column index of variable according to which observations are sorted for the split  
  
split : None or integer or float in intervall (0,1)  
  index at which sample is split. If 0<split<1 then split is interpreted as fraction of the observations in the first sample  
  
drop : None, float or int  
  If this is not None, then observation are dropped from the middle part of the sorted series. If 0<split<1 then split is interpreted as fraction of the number of observations to be dropped. Note: Currently, observations are dropped between split and split+drop, where split and drop are the indices (given by rounding if specified as fraction). The first sample is [0:split], the second sample is [split+drop:]  
  
alternative : string, ‘increasing’, ‘decreasing’ or ‘two-sided’  
  default is increasing. This specifies the alternative for the p-value calculation.

**Returns**  
  
(fval, pval) or res :  
  
fval : float  
  value of the F-statistic  
  
pval : float  
  p-value of the hypothesis that the variance in one subsample is larger than in the other subsample  
  
res : instance of result class  
  The class instance is just a storage for the intermediate and final results that are calculated
Notes

The Null hypothesis is that the variance in the two sub-samples are the same. The alternative hypothesis, can be increasing, i.e. the variance in the second sample is larger than in the first, or decreasing or two-sided.

Results are identical R, but the drop option is defined differently. (sorting by idx not tested yet)

Methods

```python
run(y, x[, idx, split, drop, alternative, ...]) see class docstring
```

```python
statsmodels.stats.diagnostic.HetGoldfeldQuandt.run
```

```python
HetGoldfeldQuandt.run(y, x, idx=None, split=None, drop=None, alternative='increasing', attach=True)
```

```python
see class docstring
```

```python
statsmodels.stats.diagnostic.het_goldfeldquandt
```

```python
statsmodels.stats.diagnostic.het_goldfeldquandt = <statsmodels.sandbox.stats.diagnostic.HetGoldfeldQuandt object at 0x07FF4270>
```

```python
see class docstring
```

```python
statsmodels.stats.diagnostic.het_breushpagan
```

```python
statsmodels.stats.diagnostic.het_breushpagan(resid, exog_het)
```

Breush-Pagan Lagrange Multiplier test for heteroscedasticity

The tests the hypothesis that the residual variance does not depend on the variables in x in the form

Math

\[ \sigma_i = \sigma * f(\alpha_0 + \alpha z_i) \]

Homoscedasticity implies that \( \alpha=0 \)

Parameters

- **resid**: arraylike, (nobs,)
  For the Breush-Pagan test, this should be the residual of a regression. If an array is given in `exog`, then the residuals are calculated by the an OLS regression or resid on \( \text{exog} \). In this case resid should contain the dependent variable. Exog can be the same as x. TODO: I dropped the exog option, should I add it back?

- **exog_het**: array_like, (nobs, nvars)
  This contains variables that might create data dependent heteroscedasticity.

Returns

- **lm**: float
  lagrange multiplier statistic

- **lm_pvalue**: float
  p-value of lagrange multiplier test

- **fvalue**: float
  f-statistic of the hypothesis that the error variance does not depend on x

- **f_pvalue**: float
p-value for the f-statistic

Notes

Assumes x contains constant (for counting dof and calculation of $R^2$). In the general description of LM test, Greene mentions that this test exaggerates the significance of results in small or moderately large samples. In this case the F-statistic is preferable.

Verification

Chisquare test statistic is exactly ($<1\times10^{-13}$) the same result as bptest in R-stats with defaults (studentize=True).

Implementation This is calculated using the generic formula for LM test using $R^2$ (Greene, section 17.6) and not with the explicit formula (Greene, section 11.4.3). The degrees of freedom for the p-value assume x is full rank.

References


\texttt{statsmodels.stats.diagnostic.het_white}

\texttt{statsmodels.stats.diagnostic.het_white}(\texttt{resid, exog, retres=False}) White’s Lagrange Multiplier Test for Heteroscedasticity

- **Parameters**
  - \texttt{resid} : array_like
    - residuals, square of it is used as endogenous variable
  - \texttt{exog} : array_like
    - possible explanatory variables for variance, squares and interaction terms are included in the auxiliary regression.
  - \texttt{resstore} : instance (optional)
    - a class instance that holds intermediate results. Only returned if store=True

- **Returns**
  - \texttt{lm} : float
    - lagrange multiplier statistic
  - \texttt{lm_pvalue} : float
    - p-value of lagrange multiplier test
  - \texttt{fvalue} : float
    - f-statistic of the hypothesis that the error variance does not depend on x. This is an alternative test variant not the original LM test.
  - \texttt{f_pvalue} : float
    - p-value for the f-statistic

Notes

- assumes x contains constant (for counting dof)
- question: does f-statistic make sense? constant?
References

Greene section 11.4.1 5th edition p. 222 now test statistic reproduces Greene 5th, example 11.3

```python
statsmodels.stats.diagnostic.het_arch
```

Enlge’s Test for Autoregressive Conditional Heteroscedasticity (ARCH)

**Parameters**
- `resid`: ndarray, (nobs,)
  - residuals from an estimation, or time series
- `maxlag`: int
  - highest lag to use
- `autolag`: None or string
  - If None, then a fixed number of lags given by maxlag is used.
- `store`: bool
  - If true then the intermediate results are also returned
- `ddof`: int
  - Not Implemented Yet
  - If the residuals are from a regression, or ARMA estimation, then there are recommendations to correct the degrees of freedom by the number of parameters that have been estimated, for example ddof=p+a for an ARMA(p,q) (need reference, based on discussion on R finance mailinglist)

**Returns**
- `lm`: float
  - Lagrange multiplier test statistic
- `impval`: float
  - p-value for Lagrange multiplier test
- `fval`: float
  - fstatistic for F test, alternative version of the same test based on F test for the parameter restriction
- `fpval`: float
  - pvalue for F test
- `resstore`: instance (optional)
  - a class instance that holds intermediate results. Only returned if store=True

**Notes**

verified against R:FinTS::ArchTest
statsmodels.stats.diagnostic.linear_harvey_collier

statsmodels.stats.diagnostic.linear_harvey_collier(res)

Harvey Collier test for linearity

The Null hypothesis is that the regression is correctly modeled as linear.

Parameters:
res : Result instance

Returns:

tvalue : float

test statistic, based on ttest_1sample

pvalue : float

pvalue of the test

Notes

TODO: add sort_by option

This test is a t-test that the mean of the recursive ols residuals is zero. Calculating the recursive residuals might take some time for large samples.

statsmodels.stats.diagnostic.linear_rainbow

statsmodels.stats.diagnostic.linear_rainbow(res, frac=0.5)

Rainbow test for linearity

The Null hypothesis is that the regression is correctly modelled as linear. The alternative for which the power might be large are convex, check

Parameters:
res : Result instance

Returns:

fstat : float

test statistic based of F test

pvalue : float

pvalue of the test

statsmodels.stats.diagnostic.linear_lm

statsmodels.stats.diagnostic.linear_lm(resid, exog, func=None)

Lagrange multiplier test for linearity against functional alternative

limitations: Assumes currently that the first column is integer. Currently it doesn’t check whether the transformed variables contain NaNs, for example log of negative number.

Parameters:
resid : ndarray
residuals of a regression

exog : ndarray
exogenous variables for which linearity is tested

func : callable
If func is None, then squares are used. func needs to take an array of exog and return an array of transformed variables.
Returns `lm`: float
Lagrange multiplier test statistic

`lm_pval`: float
p-value of Lagrange multiplier test

`ftest`: ContrastResult instance
the results from the F test variant of this test

Notes
written to match Gretl’s linearity test. The test runs an auxiliary regression of the residuals on the combined original and transformed regressors. The Null hypothesis is that the linear specification is correct.

`statsmodels.stats.diagnostic.breaks_cusumolsresid`

`statsmodels.stats.diagnostic.breaks_cusumolsresid(olsresidual, ddof=0)`
cusum test for parameter stability based on ols residuals

Parameters `olsresiduals`: ndarray
array of residuals from an OLS estimation

`ddof`: int
number of parameters in the OLS estimation, used as degrees of freedom correction for error variance.

Returns `sup_b`: float
test statistic, maximum of absolute value of scaled cumulative OLS residuals

`pval`: float
Probability of observing the data under the null hypothesis of no structural change, based on asymptotic distribution which is a Brownian Bridge

`crit`: list:
tabulated critical values, for alpha = 1%, 5% and 10%

Notes
tested against R: strucchange
Not clear: Assumption 2 in Ploberger, Kramer assumes that exog x have asymptotically zero mean, x.mean(0) = [1, 0, 0, ..., 0] Is this really necessary? I don’t see how it can affect the test statistic under the null. It does make a difference under the alternative. Also, the asymptotic distribution of test statistic depends on this.

From examples it looks like there is little power for standard cusum if exog (other than constant) have mean zero.

References
**statsmodels.stats.diagnostic.breaks_hansen**

statsmodels.stats.diagnostic.breaks_hansen(olsresults)

test for model stability, breaks in parameters for ols, Hansen 1992

**Parameters**
olsresults : instance of RegressionResults

uses only endog and exog

**Returns**
teststat : float

Hansen’s test statistic
crit : structured array

critical values at alpha=0.95 for different nvars

pvalue Not yet :

ft, s : arrays

temporary return for debugging, will be removed

**Notes**

looks good in example, maybe not very powerful for small changes in parameters

According to Greene, distribution of test statistics depends on nvar but not on nobs.
Test statistic is verified against R:strucchange

**References**

Greene section 7.5.1, notation follows Greene

**statsmodels.stats.diagnostic.recursive_olsresiduals**

statsmodels.stats.diagnostic.recursive_olsresiduals(olsresults, skip=None, lamda=0.0, alpha=0.95)

calculate recursive ols with residuals and cusum test statistic

**Parameters**
olsresults : instance of RegressionResults

uses only endog and exog

skip : int or None

number of observations to use for initial OLS, if None then skip is set equal to the number of regressors (columns in exog)

lamda : float

weight for Ridge correction to initial (X’X)^{-1}

alpha : {0.95, 0.99}

confidence level of test, currently only two values supported, used for confidence interval in cusum graph

**Returns**
rresid : array

recursive ols residuals
rparams : array
    recursive ols parameter estimates
rypred : array
    recursive prediction of endogenous variable
rresid_standardized : array
    recursive residuals standardized so that N(0,sigma2) distributed, where sigma2 is the
    error variance
rresid_scaled : array
    recursive residuals normalize so that N(0,1) distributed
rcusum : array
    cumulative residuals for cusum test
rcusumci : array
    confidence interval for cusum test, currently hard coded for alpha=0.95

Notes

It produces same recursive residuals as other version. This version updates the inverse of the X’X matrix and
does not require matrix inversion during updating. looks efficient but no timing

Confidence interval in Greene and Brown, Durbin and Evans is the same as in Ploberger after a little bit of
algebra.

References

jplv to check formulas, follows Harvey BigJudge 5.5.2b for formula for inverse(X’X) updating Greene section
7.5.2


statsmodels.stats.diagnostic.CompareCox

class statsmodels.stats.diagnostic.CompareCox
    Cox Test for non-nested models

    Parameters results_x : Result instance
        result instance of first model
    results_z : Result instance
        result instance of second model
    attach : bool

    Formulas from Greene, section 8.3.4 translated to code :
    produces correct results for Example 8.3, Greene :
Methods

```python
run(results_x, results_z[, attach]) run Cox test for non-nested models
```

```python
statsmodels.stats.diagnostic.CompareCox.run
```

```python
CompareCox.run(results_x, results_z, attach=True)
run Cox test for non-nested models
```

**Parameters**

- `results_x`: Result instance
  - result instance of first model
- `results_z`: Result instance
  - result instance of second model
- `attach`: bool
  - If true, then the intermediate results are attached to the instance.

**Returns**

- `tstat`: float
  - t statistic for the test that including the fitted values of the first model in the second model has no effect.
- `pvalue`: float
  - two-sided pvalue for the t statistic

**Notes**

Tests of non-nested hypothesis might not provide unambiguous answers. The test should be performed in both directions and it is possible that both or neither test rejects. see ??? for more information.

**References**

???

```python
statsmodels.stats.diagnostic.compare_cox
```

```python
statsmodels.stats.diagnostic.compare_cox = <statsmodels.sandbox.stats.diagnostic.CompareCox object at 0x07FF41B0>
Cox Test for non-nested models
```

**Parameters**

- `results_x`: Result instance
  - result instance of first model
- `results_z`: Result instance
  - result instance of second model
- `attach`: bool

**Formulas from Greene, section 8.3.4 translated to code**:

- produces correct results for Example 8.3, Greene:
statsmodels.stats.diagnostic.CompareJ

class statsmodels.stats.diagnostic.CompareJ
    J-Test for comparing non-nested models

    Parameters
    results_x : Result instance
        result instance of first model
    results_z : Result instance
        result instance of second model
    attach : bool
        From description in Greene, section 8.3.3 :
        produces correct results for Example 8.3, Greene - not checked yet :
        #currently an exception, but I don’t have clean reload in python session :
        check what results should be attached :

    Methods

    run(results_x, results_z[, attach]) run J-test for non-nested models

statsmodels.stats.diagnostic.CompareJ.run

CompareJ.run(results_x, results_z, attach=True) run J-test for non-nested models

    Parameters
    results_x : Result instance
        result instance of first model
    results_z : Result instance
        result instance of second model
    attach : bool
        If true, then the intermediate results are attached to the instance.

    Returns
    tstat : float
        t statistic for the test that including the fitted values of the first model in the second model has no effect.
    pvalue : float
        two-sided pvalue for the t statistic

    Notes

    Tests of non-nested hypothesis might not provide unambiguous answers. The test should be performed in both directions and it is possible that both or neither test rejects. see ??? for more information.
References

statsmodels.stats.diagnostic.compare_j

statsmodels.stats.diagnostic.compare_j = <statsmodels.sandbox.stats.diagnostic.CompareJ object at 0x07FF4210>

J-Test for comparing non-nested models

Parameters

- results_x: Result instance
  result instance of first model
- results_z: Result instance
  result instance of second model
- attach: bool

From description in Greene, section 8.3.3:
produces correct results for Example 8.3, Greene - not checked yet:
#currently an exception, but I don’t have clean reload in python session:
check what results should be attached:

statsmodels.stats.diagnostic.unitroot_adf

statsmodels.stats.diagnostic.unitroot_adf(x, maxlag=None, trendorder=0, autolag='AIC', store=False)

statsmodels.stats.diagnostic.normal_ad

statsmodels.stats.diagnostic.normal_ad(x, axis=0)

Anderson-Darling test for normal distribution unknown mean and variance

Parameters

- x: array_like
  data array, currently only 1d

Returns

- ad2: float
  Anderson Darling test statistic
- pval: float
  pvalue for hypothesis that the data comes from a normal distribution with unknown mean and variance

statsmodels.stats.diagnostic.kstest_normal

statsmodels.stats.diagnostic.kstest_normal(x, pvalmethod='approx')

Lillifors test for normality, Kolmogorov Smirnov test with estimated mean and variance

Parameters

- x: array_like, 1d
  data series, sample
pvalmethod : ‘approx’, ‘table’

‘approx’ uses the approximation formula of Dalal and Wilkinson, valid for pvalues < 0.1. If the pvalue is larger than 0.1, then the result of ‘table’ is returned ‘table’ uses the table from Dalal and Wilkinson, which is available for pvalues between 0.001 and 0.2, and the formula of Lilliefors for large n (n>900). Values in the table are linearly interpolated. Values outside the range will be returned as bounds, 0.2 for large and 0.001 for small pvalues.

Returns ksstat : float

Kolmogorov-Smirnov test statistic with estimated mean and variance.

pvalue : float

If the pvalue is lower than some threshold, e.g. 0.05, then we can reject the Null hypothesis that the sample comes from a normal distribution

Notes

Reported power to distinguish normal from some other distributions is lower than with the Anderson-Darling test.

could be vectorized

statsmodels.stats.diagnostic.lillifors

statsmodels.stats.diagnostic.lillifors(x, pvalmethod='approx')

Lillifors test for normality.

Kolmogorov Smirnov test with estimated mean and variance

Parameters x : array_like, 1d

data series, sample

pvalmethod : ‘approx’, ‘table’

‘approx’ uses the approximation formula of Dalal and Wilkinson, valid for pvalues < 0.1. If the pvalue is larger than 0.1, then the result of ‘table’ is returned ‘table’ uses the table from Dalal and Wilkinson, which is available for pvalues between 0.001 and 0.2, and the formula of Lilliefors for large n (n>900). Values in the table are linearly interpolated. Values outside the range will be returned as bounds, 0.2 for large and 0.001 for small pvalues.

Returns ksstat : float

Kolmogorov-Smirnov test statistic with estimated mean and variance.

pvalue : float

If the pvalue is lower than some threshold, e.g. 0.05, then we can reject the Null hypothesis that the sample comes from a normal distribution

Notes

Reported power to distinguish normal from some other distributions is lower than with the Anderson-Darling test.

could be vectorized
Outliers and influence measures

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLSInfluence(results)</td>
<td>class to calculate outlier and influence measures for OLS result</td>
</tr>
<tr>
<td>variance_inflation_factor(exog, exog_idx)</td>
<td>variance inflation factor, VIF, for one exogenous variable</td>
</tr>
</tbody>
</table>

```python
statsmodels.stats.outliers_influence.OLSInfluence

class statsmodels.stats.outliers_influence.OLSInfluence(results)
    class to calculate outlier and influence measures for OLS result

Parameters

results : Regression Results instance
    currently assumes the results are from an OLS regression

Notes

One part of the results can be calculated without any auxiliary regression (some of which have the _internal postfix in the name. Other statistics require leave-one-observation-out (LOOO) auxiliary regression, and will be slower (mainly results with _external postfix in the name). The auxiliary LOOO regression only the required results are stored.

Using the LOO measures is currently only recommended if the data set is not too large. One possible approach for LOO measures would be to identify possible problem observations with the _internal measures, and then run the leave-one-observation-out only with observations that are possible outliers. (However, this is not yet available in an automated way.)

This should be extended to general least squares.

The leave-one-variable-out (LOVO) auxiliary regression are currently not used.

Methods

cooks_distance() (cached attribute) Cooks distance
cov_ratio() (cached attribute) covariance ratio between LOOO and original
det_cov_params_not_obsi() (cached attribute) determinant of cov_params of all LOOO regressions
dfbetas() (cached attribute) dfbetas
dffits() (cached attribute) dffits measure for influence of an observation
dffits_internal() (cached attribute) dffits measure for influence of an observation
ess_press() (cached attribute) error sum of squares of PRESS residuals
get_resid_studentized_external([sigma]) calculate studentized residuals
hat_diag_factor() (cached attribute) factor of diagonal of hat_matrix used in influence
hat_matrix_diag() (cached attribute) diagonal of the hat_matrix for OLS
influence() (cached attribute) influence measure
params_not_obsi() (cached attribute) parameter estimates for all LOOO regressions
resid_press() (cached attribute) PRESS residuals
resid_std() (cached attribute) estimate of standard deviation of the residuals
resid_studentized_external() (cached attribute) studentized residuals using LOOO variance
resid_studentized_internal() (cached attribute) studentized residuals using variance from OLS
resid_var() (cached attribute) estimate of variance of the residuals
sigma2_not_obsi() (cached attribute) error variance for all LOOO regressions
summary_frame() creates a DataFrame with all available influence results.
summary_table([float_fmt]) create a summary table with all influence and outlier measures
statsmodels.stats.outliers_influence.OLSInfluence.cooks_distance
static OLSInfluence.cooks_distance()
  (cached attribute) Cooks distance
  uses original results, no nobs loop

statsmodels.stats.outliers_influence.OLSInfluence.cov_ratio
static OLSInfluence.cov_ratio()
  (cached attribute) covariance ratio between LOOO and original
  This uses determinant of the estimate of the parameter covariance from leave-one-out estimates. requires
  leave one out loop for observations

statsmodels.stats.outliers_influence.OLSInfluence.det_cov_params_not_obsi
static OLSInfluence.det_cov_params_not_obsi()
  (cached attribute) determinant of cov_params of all LOOO regressions
  uses results from leave-one-observation-out loop

statsmodels.stats.outliers_influence.OLSInfluence.dfbetas
static OLSInfluence.dfbetas()
  (cached attribute) dfbetas
  uses results from leave-one-observation-out loop

statsmodels.stats.outliers_influence.OLSInfluence.dffits
static OLSInfluence.dffits()
  (cached attribute) dffits measure for influence of an observation
  based on resid_studentized_external, uses results from leave-one-observation-out loop
  It is recommended that observations with dffits large than a threshold of $2 \sqrt{k / n}$ where k is the number
  of parameters, should be investigated.

  Returns dffits : float :
    dffits_threshold : float

References

Wikipedia

statsmodels.stats.outliers_influence.OLSInfluence.dffits_internal
static OLSInfluence.dffits_internal()
  (cached attribute) dffits measure for influence of an observation
  based on resid_studentized_internal uses original results, no nobs loop

statsmodels.stats.outliers_influence.OLSInfluence.ess_press
static OLSInfluence.ess_press()
  (cached attribute) error sum of squares of PRESS residuals
statsmodels.stats.outliers_influence.OLSInfluence.get_resid_studentized_external

`get_resid_studentized_external(sigma=None)`
calculate studentized residuals

**Parameters**
- `sigma`: None or float
  estimate of the standard deviation of the residuals. If None, then the estimate from the regression results is used.

**Returns**
- `stzd_resid`: ndarray
  studentized residuals

**Notes**

Studentized residuals are defined as

\[ \text{resid} / \sigma / \sqrt{1 - h_{ii}} \]

where resid are the residuals from the regression, \( \sigma \) is an estimate of the standard deviation of the residuals, and \( h_{ii} \) is the diagonal of the hat_matrix.

---

`hat_diag_factor` static

`hat_diag_factor()`
(cached attribute) factor of diagonal of hat_matrix used in influence
this might be useful for internal reuse \( h / (1 - h) \)

`hat_matrix_diag` static

`hat_matrix_diag()`
(cached attribute) diagonal of the hat_matrix for OLS

**Notes**

temporarily calculated here, this should go to model class

`influence` static

`influence()`
(cached attribute) influence measure
matches the influence measure that gretl reports \( u * h / (1 - h) \) where \( u \) are the residuals and \( h \) is the diagonal of the hat_matrix

`params_not_obsi` static

`params_not_obsi()`
(cached attribute) parameter estimates for all LOOO regressions
uses results from leave-one-observation-out loop

`resid_press` static

`resid_press()`
(cached attribute) PRESS residuals

---

3.8. Statistics stats

---
statsmodels.stats.outliers_influence.OLSInfluence.resid_std

static OLSInfluence.resid_std()

(cached attribute) estimate of standard deviation of the residuals

See also:

resid_var

statsmodels.stats.outliers_influence.OLSInfluence.resid_studentized_external

static OLSInfluence.resid_studentized_external()

(cached attribute) studentized residuals using LOOO variance

this uses sigma from leave-one-out estimates

requires leave one out loop for observations

statsmodels.stats.outliers_influence.OLSInfluence.resid_studentized_internal

static OLSInfluence.resid_studentized_internal()

(cached attribute) studentized residuals using variance from OLS

this uses sigma from original estimate does not require leave one out loop

statsmodels.stats.outliers_influence.OLSInfluence.resid_var

static OLSInfluence.resid_var()

(cached attribute) estimate of variance of the residuals

\[ \sigma^2 = \sigma^2_{\text{OLS}} \times (1 - \text{hii}) \]

where hii is the diagonal of the hat matrix

statsmodels.stats.outliers_influence.OLSInfluence.sigma2_not_obsi

static OLSInfluence.sigma2_not_obsi()

(cached attribute) error variance for all LOOO regressions

This is ‘mse_resid’ from each auxiliary regression.

uses results from leave-one-observation-out loop

statsmodels.stats.outliers_influence.OLSInfluence.summary_frame

OLSInfluence.summary_frame()

Creates a DataFrame with all available influence results.

Returns frame : DataFrame

A DataFrame with all results.

Notes

The resultant DataFrame contains six variables in addition to the DFBETAS. These are:

- **cooks_d**: Cook’s Distance defined in `Influence.cooks_distance`
- **standard_resid**: Standardized residuals defined in `Influence.resid_studentized_internal`
- **hat_diag**: The diagonal of the projection, or hat, matrix defined in `Influence.hat_matrix_diag`
- `dffits_internal`: DFFITS statistics using internally Studentized residuals defined in `Influence.dffits_internal`
- `dffits`: DFFITS statistics using externally Studentized residuals defined in `Influence.dffits`
- `student_resid`: Externally Studentized residuals defined in `Influence.resid_studentized_external`

```python
statsmodels.stats.outliers_influence.OLSInfluence.summary_table
```

Create a summary table with all influence and outlier measures.

This does currently not distinguish between statistics that can be calculated from the original regression results and for which a leave-one-observation-out loop is needed.

**Returns**

- `res`: SimpleTable instance
  
  SimpleTable instance with the results, can be printed

**Notes**

This also attaches `table_data` to the instance.

```python
statsmodels.stats.outliers_influence.variance_inflation_factor
```

Variance inflation factor, VIF, for one exogenous variable.

The variance inflation factor is a measure for the increase of the variance of the parameter estimates if an additional variable, given by `exog_idx` is added to the linear regression. It is a measure for multicollinearity of the design matrix, `exog`.

One recommendation is that if VIF is greater than 5, then the explanatory variable given by `exog_idx` is highly collinear with the other explanatory variables, and the parameter estimates will have large standard errors because of this.

**Parameters**

- `exog`: ndarray, (nobs, k_vars)
  
  Design matrix with all explanatory variables, as for example used in regression

- `exog_idx`: int
  
  Index of the exogenous variable in the columns of `exog`

**Returns**

- `vif`: float
  
  Variance inflation factor

**See also:**

- `xxx` class for regression diagnostics TODO: doesn’t exist yet

**Notes**

This function does not save the auxiliary regression.
3.8.2 Sandwich Robust Covariances

The following functions calculate covariance matrices and standard errors for the parameter estimates that are robust to heteroscedasticity and autocorrelation in the errors. Similar to the methods that are available for the LinearModel-Results, these methods are designed for use with OLS.

- `sandwich_covariance.cov_hac(results[, ...])` heteroscedasticity and autocorrelation robust covariance matrix (Newey-West)
- `sandwich_covariance.cov_nw_panel(results, ...)` Panel HAC robust covariance matrix
- `sandwich_covariance.cov_nw_groupsum(results, ...)` Driscoll and Kraay Panel robust covariance matrix
- `sandwich_covariance.cov_cluster(results, group)` cluster robust covariance matrix
- `sandwich_covariance.cov_cluster_2groups(...)` cluster robust covariance matrix for two groups/clusters
- `sandwich_covariance.cov_white_simple(results)` heteroscedasticity robust covariance matrix (White)

### `sandwich_covariance.cov_hac` Function

Assumes we have a single time series with zero axis consecutive, equal spaced time periods

**Parameters**

- `results` : result instance
  - result of a regression, uses results.model.exog and results.resid TODO: this should use wexog instead
- `nlags` : int or None
  - highest lag to include in kernel window. If None, then nlags = floor[4(T/100)^(2/9)] is used.
- `weights_func` : callable
  - `weights_func` is called with nlags as argument to get the kernel weights. default are Bartlett weights

**Returns**

- `cov` : ndarray, (k_vars, k_vars)
  - HAC robust covariance matrix for parameter estimates

**Notes**

verified only for nlags=0, which is just White just guessing on correction factor, need reference options might change when other kernels besides Bartlett are available.
Panel HAC robust covariance matrix

Assumes we have a panel of time series with consecutive, equal spaced time periods. Data is assumed to be in long format with time series of each individual stacked into one array. Panel can be unbalanced.

**Parameters**

- **results**: result instance
  - result of a regression, uses results.model.exog and results.resid TODO: this should use wexog instead
- **nlags**: int or None
  - Highest lag to include in kernel window. Currently, no default because the optimal length will depend on the number of observations per cross-sectional unit.
- **groupidx**: list of tuple
  - each tuple should contain the start and end index for an individual. (groupidx might change in future).
- **weights_func**: callable
  - weights_func is called with nlags as argument to get the kernel weights. default are Bartlett weights
- **use_correction**: ‘cluster’ or ‘hac’ or False
  - If False, then no small sample correction is used. If ‘cluster’ (default), then the same correction as in cov_cluster is used. If ‘hac’, then the same correction as in single time series, cov_hac is used.

**Returns**

- **cov**: ndarray, (k_vars, k_vars)
  - HAC robust covariance matrix for parameter estimates

**Notes**

For nlags=0, this is just White covariance, cov_white. If kernel is uniform, weights_uniform, with nlags equal to the number of observations per unit in a balance panel, then cov_cluster and cov_hac_panel are identical.

Tested against STATA newey command with same defaults.

Options might change when other kernels besides Bartlett and uniform are available.
Assumes we have a panel of time series where the time index is available. The time index is assumed to represent equal spaced periods. At least one observation per period is required.

**Parameters results**: result instance

result of a regression, uses results.model.exog and results.resid TODO: this should use wexog instead

**nlags**: int or None

Highest lag to include in kernel window. Currently, no default because the optimal length will depend on the number of observations per cross-sectional unit.

**time**: ndarray of int

This should contain the coding for the time period of each observation. Time periods should be integers in range(maxT) where maxT is obs of i

**weights_func**: callable

weights_func is called with nlags as argument to get the kernel weights. Default are Bartlett weights

**use_correction**: ‘cluster’ or ‘hac’ or False

If False, then no small sample correction is used. If ‘hac’ (default), then the same correction as in single time series, cov_hac is used. If ‘cluster’, then the same correction as in cov_cluster is used.

**Returns cov**: ndarray, (k_vars, k_vars)

HAC robust covariance matrix for parameter estimates

**Notes**

Tested against STATA xtscc package, which uses no small sample correction.

This first averages relevant variables for each time period over all individuals/groups, and then applies the same kernel weighted averaging over time as in HAC.

Warning: In the example with a short panel (few time periods and many individuals) with mainly across individual variation this estimator did not produce reasonable results.

Options might change when other kernels besides Bartlett and uniform are available.

**References**

Daniel Hoechle, xtscc paper Driscoll and Kraay

```python
statsmodels.stats.sandwich_covariance.cov_cluster
```

Calculates sandwich covariance matrix for a single cluster, i.e. grouped variables.

**Parameters results**: result instance

result of a regression, uses results.model.exog and results.resid TODO: this should use wexog instead
use_correction : bool
If true (default), then the small sample correction factor is used.

Returns cov : ndarray, (k_vars, k_vars)
cluster robust covariance matrix for parameter estimates

Notes
same result as Stata in UCLA example and same as Peterson

statsmodels.stats.sandwich_covariance.cov_cluster_2groups

statsmodels.stats.sandwich_covariance.cov_cluster_2groups(results, group, group2=None, use_correction=True)
cluster robust covariance matrix for two groups/clusters

Parameters results : result instance
result of a regression, uses results.model.exog and results.resid TODO: this should use wexog instead

use_correction : bool
If true (default), then the small sample correction factor is used.

Returns cov_both : ndarray, (k_vars, k_vars)
cluster robust covariance matrix for parameter estimates, for both clusters
cov_0 : ndarray, (k_vars, k_vars)
cluster robust covariance matrix for parameter estimates for first cluster
cov_1 : ndarray, (k_vars, k_vars)
cluster robust covariance matrix for parameter estimates for second cluster

Notes
verified against Peterson’s table, (4 decimal print precision)

statsmodels.stats.sandwich_covariance.cov_white_simple

statsmodels.stats.sandwich_covariance.cov_white_simple(results, use_correction=True)
heteroscedasticity robust covariance matrix (White)

Parameters results : result instance
result of a regression, uses results.model.exog and results.resid TODO: this should use wexog instead

Returns cov : ndarray, (k_vars, k_vars)
heteroscedasticity robust covariance matrix for parameter estimates
See also:

cov_hc1, cov_hc2, cov_hc3

Notes

This produces the same result as cov_hc0, and does not include any small sample correction. verified (against LinearRegressionResults and Peterson)
The following are standalone versions of the heteroscedasticity robust standard errors attached to LinearModelResults

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sandwich_covariance.cov_hc0(results)</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>sandwich_covariance.cov_hc1(results)</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>sandwich_covariance.cov_hc2(results)</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>sandwich_covariance.cov_hc3(results)</td>
<td>See statsmodels.RegressionResults</td>
</tr>
<tr>
<td>sandwich_covariance.se_cov(cov)</td>
<td>get standard deviation from covariance matrix</td>
</tr>
</tbody>
</table>

statsmodels.stats.sandwich_covariance.cov_hc0

statsmodels.stats.sandwich_covariance.cov_hc0(results)
See statsmodels.RegressionResults

statsmodels.stats.sandwich_covariance.cov_hc1

statsmodels.stats.sandwich_covariance.cov_hc1(results)
See statsmodels.RegressionResults

statsmodels.stats.sandwich_covariance.cov_hc2

statsmodels.stats.sandwich_covariance.cov_hc2(results)
See statsmodels.RegressionResults

statsmodels.stats.sandwich_covariance.cov_hc3

statsmodels.stats.sandwich_covariance.cov_hc3(results)
See statsmodels.RegressionResults

statsmodels.stats.sandwich_covariance.se_cov

statsmodels.stats.sandwich_covariance.se_cov(cov)
get standard deviation from covariance matrix

Parameters:
cov : array_like, square
covariance matrix

Returns:
std : ndarray
standard deviation from diagonal of cov
3.8.3 Goodness of Fit Tests and Measures

some tests for goodness of fit for univariate distributions

- **powerdiscrepancy**(observed, expected[, ...])
  Calculates power discrepancy, a class of goodness-of-fit tests as a measure of discrepancy between observed and expected data. This contains several goodness-of-fit tests as special cases, see the description of lambda, the exponent of the power discrepancy. The pvalue is based on the asymptotic chi-square distribution of the test statistic.

  freeman_tukey: \( D(x|\theta) = \sum_j (\sqrt{x_j} - \sqrt{e_j})^2 \)

  **Parameters**
  - **o**: Iterable
    Observed values
  - **e**: Iterable
    Expected values
  - **lambda**: float or string
    - float: exponent \( a \) for power discrepancy
    - 'loglikeratio': \( a = 0 \)
    - 'freeman_tukey': \( a = -0.5 \)
    - 'pearson': \( a = 1 \) (standard chisquare test statistic)
    - 'modified_loglikeratio': \( a = -1 \)
    - 'cressie_read': \( a = 2/3 \)
    - 'neyman': \( a = -2 \) (Neyman-modified chisquare, reference from a book?)
  - **axis**: int
    axis for observations of one series
  - **ddof**: int
    degrees of freedom correction,

  **Returns**
  - **D_obs**: Discrepancy of observed values
  - **pvalue**: pvalue

  **References**


Steele, M. 1,2, C. Hurst 3 and J. Chaseling, Simulated Power of Discrete Goodness-of-Fit Tests for Likert Type Data

Examples

```python
>>> observed = np.array([2., 4., 2., 1., 1.])
>>> expected = np.array([0.2, 0.2, 0.2, 0.2, 0.2])

for checking correct dimension with multiple series

```python
>>> powerdiscrepancy(np.column_stack((observed,observed)).T, 10*expected, lambd='freeman_tukey', axis=1)
darray([[ 2.745166, 2.745166]])
>>> powerdiscrepancy(np.column_stack((observed,observed)).T, 10*expected, axis=1)
darray([[ 0.6013346, 0.6013346]])
>>> powerdiscrepancy(np.column_stack((observed,observed)).T, 10*expected, axis=1)
darray([[ 0.59657359, 0.59657359]])
>>> powerdiscrepancy(np.column_stack((observed,observed)).T, 10*expected, lambd=0, axis=1)
darray([[ 0.59657359, 0.59657359]])
>>> powerdiscrepancy(np.column_stack((observed,observed)).T, 10*expected, lambd=1, axis=1)
darray([[ 0.59657359, 0.59657359]])
>>> powerdiscrepancy(np.column_stack((observed,observed)).T, 10*expected, lambd=2/3.0, axis=1)
darray([[ 0.57518277, 0.57518277]])
>>> powerdiscrepancy(np.column_stack((observed,observed)).T, expected, lambd=2/3.0, axis=1)
darray([[ 0.57518277, 0.57518277]])
>>> powerdiscrepancy(np.column_stack((observed,observed)), expected, lambd=2/3.0, axis=0)
darray([[ 2.89714546, 2.89714546]])
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), expected, lambd=2/3.0, axis=0)
darray([[ 5.79429093, 5.79429093]])
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), 20*expected, axis=0)
darray([[ 2.89714546, 5.79429093]])
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), 20*expected, axis=0)
darray([[ 5.79429093, 5.79429093]])
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), expected, axis=0)
darray([[ 5.79429093, 5.79429093]])
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), 20*expected, axis=0)
darray([[ 5.79429093, 5.79429093]])
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), expected, axis=0)
darray([[ 2.89714546, 5.79429093]])
```

each random variable can have different total count/sum

```python
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), expected, lambd=2/3.0, axis=0)
darray([[ 2.89714546, 2.89714546]])
>>> powerdiscrepancy(np.column_stack((observed,2*observed)), expected, lambd=2/3.0, axis=0)
darray([[ 2.89714546, 2.89714546]])
>>> powerdiscrepancy(np.column_stack((2*observed,2*observed)), expected, lambd=2/3.0, axis=0)
darray([[ 5.79429093, 5.79429093]])
>>> powerdiscrepancy(np.column_stack((2*observed,2*observed)), expected, lambd=2/3.0, axis=0)
darray([[ 5.79429093, 5.79429093]])
>>> powerdiscrepancy(np.column_stack((2*observed,2*observed)), 20*expected, axis=0)
darray([[ 2.89714546, 2.89714546]])
>>> powerdiscrepancy(np.column_stack((2*observed,2*observed)), 20*expected, axis=0)
darray([[ 2.89714546, 2.89714546]])
```
parameters of distribution

**alpha** : float

significance level, threshold for p-value

**Returns result** : bool

0 if test passes, 1 if test fails

**Notes**

originally written for scipy.stats test suite, still needs to be checked for standalone usage, insufficient input checking may not run yet (after copy/paste)

refactor: **maybe a class, check returns, or separate binning from** test results

### statsmodels.stats.gof.gof_binning_discrete

gets bins for chisquare type gof tests for a discrete distribution

**Parameters**

**rvs** : array

sample data

**distname** : string

name of distribution function

**arg** : sequence

parameters of distribution

**nsupp** : integer

number of bins. The algorithm tries to find bins with equal weights. depending on the distribution, the actual number of bins can be smaller.

**Returns**

**freq** : array

empirical frequencies for sample; not normalized, adds up to sample size

**expfreq** : array

theoretical frequencies according to distribution

**histsupp** : array

bin boundaries for histogram, (added 1e-8 for numerical robustness)

**Notes**

The results can be used for a chisquare test

(chis,pval) = stats.chisquare(freq, expfreq)

originally written for scipy.stats test suite, still needs to be checked for standalone usage, insufficient input checking may not run yet (after copy/paste)

refactor: **maybe a class, check returns, or separate binning from** test results
**todo:** optimal number of bins? (check easyfit), recommendation in literature at least 5 expected observations in each bin

```python
statsmodels.stats.gof.chisquare_effectsize
```

Effect size for a chisquare goodness-of-fit test

**Parameters**

- `probs0` : array_like
  probabilities or cell frequencies under the Null hypothesis

- `probs1` : array_like
  probabilities or cell frequencies under the Alternative hypothesis

- `correction` : None or tuple (nobs, df)
  If None, then the effect size is the chisquare statistic divide by the number of observations. If the correction is a tuple (nobs, df), then the effectsize is corrected to have less bias and a smaller variance. However, the correction can make the effectsize negative. In that case, the effectsize is set to zero. Pederson and Johnson (1990) as referenced in McLaren et all. (1994)

- `cohen` : bool
  If True, then the square root is returned as in the definition of the effect size by Cohen (1977), If False, then the original effect size is returned.

- `axis` : int
  If the probability arrays broadcast to more than 1 dimension, then this is the axis over which the sums are taken.

**Returns**

- `effectsize` : float
  effect size of chisquare test

```python
normal_ad(x[, axis])
kstest_normal(x[, pvalmethod])
lillifors(x[, pvalmethod])
```

Anderson-Darling test for normal distribution unknown mean and variance
Lillifors test for normality,
Lillifors test for normality.

```python
statsmodels.stats.diagnostic.normal_ad
```

Anderson-Darling test for normal distribution unknown mean and variance

**Parameters**

- `x` : array_like
  data array, currently only 1d

**Returns**

- `ad2` : float
  Anderson Darling test statistic

- `pval` : float
  pvalue for hypothesis that the data comes from a normal distribution with unknown mean and variance
statsmodels.stats.diagnostic.kstest_normal

statsmodels.stats.diagnostic.kstest_normal(x, pvalmethod='approx')

Lillifors test for normality,

Kolmogorov Smirnov test with estimated mean and variance

Parameters x : array_like, 1d
data series, sample

pvalmethod : ‘approx’, ‘table’

‘approx’ uses the approximation formula of Dalal and Wilkinson, valid for pvalues < 0.1. If the pvalue is larger than 0.1, then the result of ‘table’ uses the table from Dalal and Wilkinson, which is available for pvalues between 0.001 and 0.2, and the formula of Lilliefors for large n (n>900). Values in the table are linearly interpolated. Values outside the range will be returned as bounds, 0.2 for large and 0.001 for small pvalues.

Returns ksstat : float
Kolmogorov-Smirnov test statistic with estimated mean and variance.

pvalue : float
If the pvalue is lower than some threshold, e.g. 0.05, then we can reject the Null hypothesis that the sample comes from a normal distribution

Notes

Reported power to distinguish normal from some other distributions is lower than with the Anderson-Darling test.

could be vectorized

statsmodels.stats.diagnostic.lillifors

statsmodels.stats.diagnostic.lillifors(x, pvalmethod='approx')

Lillifors test for normality,

Kolmogorov Smirnov test with estimated mean and variance

Parameters x : array_like, 1d
data series, sample

pvalmethod : ‘approx’, ‘table’

‘approx’ uses the approximation formula of Dalal and Wilkinson, valid for pvalues < 0.1. If the pvalue is larger than 0.1, then the result of ‘table’ uses the table from Dalal and Wilkinson, which is available for pvalues between 0.001 and 0.2, and the formula of Lilliefors for large n (n>900). Values in the table are linearly interpolated. Values outside the range will be returned as bounds, 0.2 for large and 0.001 for small pvalues.

Returns ksstat : float
Kolmogorov-Smirnov test statistic with estimated mean and variance.

pvalue : float
If the pvalue is lower than some threshold, e.g. 0.05, then we can reject the Null hypothesis that the sample comes from a normal distribution.

**Notes**

Reported power to distinguish normal from some other distributions is lower than with the Anderson-Darling test.

could be vectorized

### 3.8.4 Non-Parametric Tests

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mcnemar(x[, y, exact, correction])</code></td>
<td>McNemar test</td>
</tr>
<tr>
<td><code>symmetry_bowker(table)</code></td>
<td>Test for symmetry of a (k, k) square contingency table</td>
</tr>
<tr>
<td><code>median_test_ksample(x, groups)</code></td>
<td>Chisquare test for equality of median/location</td>
</tr>
<tr>
<td><code>runstest_1samp(x[, cutoff, correction])</code></td>
<td>Use runs test on binary discretized data above/below cutoff</td>
</tr>
<tr>
<td><code>runstest_2samp(x[, y, groups, correction])</code></td>
<td>Wald-Wolfowitz runtest for two samples</td>
</tr>
<tr>
<td><code>cochrans_q(x)</code></td>
<td>Cochran’s Q test for identical effect of k treatments</td>
</tr>
<tr>
<td><code>Runs(x)</code></td>
<td>Class for runs in a binary sequence</td>
</tr>
</tbody>
</table>

**statsmodels.sandbox.stats.runs.mcnemar**

**statsmodels.sandbox.stats.runs.mcnemar (x, y=None, exact=True, correction=True)**

McNemar test

**Parameters**  
`x, y` : array_like  
Two paired data samples. If `y` is None, then `x` can be a 2 by 2 contingency table. `x` and `y` can have more than one dimension, then the results are calculated under the assumption that axis zero contains the observation for the samples.

`exact` : bool  
If `exact` is true, then the binomial distribution will be used. If `exact` is false, then the chisquare distribution will be used, which is the approximation to the distribution of the test statistic for large sample sizes.

`correction` : bool  
If true, then a continuity correction is used for the chisquare distribution (if `exact` is false.)

**Returns**  
`stat` : float or int, array  
The test statistic is the chisquare statistic if `exact` is false. If the exact binomial distribution is used, then this contains the min(n1, n2), where n1, n2 are cases that are zero in one sample but one in the other sample.

`pvalue` : float or array  
The p-value of the null hypothesis of equal effects.
Notes

This is a special case of Cochran’s Q test. The results when the chisquare distribution is used are identical, except for continuity correction.

**statsmodels.sandbox.stats.runs.symmetry_bowker**

`statsmodels.sandbox.stats.runs.symmetry_bowker(table)`

Test for symmetry of a (k, k) square contingency table

This is an extension of the McNemar test to test the Null hypothesis that the contingency table is symmetric around the main diagonal, that is

\[ n_{i, j} = n_{j, i} \] for all \( i, j \)

**Parameters**

- **table**: array_like, 2d, (k, k)
  - a square contingency table that contains the count for k categories in rows and columns.

**Returns**

- **statistic**: float
  - chisquare test statistic
- **p-value**: float
  - p-value of the test statistic based on chisquare distribution
- **df**: int
  - degrees of freedom of the chisquare distribution

**See also:**

- mcnemar

**Notes**

Implementation is based on the SAS documentation, R includes it in `mcnemar.test` if the table is not 2 by 2.

The pvalue is based on the chisquare distribution which requires that the sample size is not very small to be a good approximation of the true distribution. For 2x2 contingency tables exact distribution can be obtained with `mcnemar`

**statsmodels.sandbox.stats.runs.median_test_ksample**

`statsmodels.sandbox.stats.runs.median_test_ksample(x, groups)`

chisquare test for equality of median/location

This tests whether all groups have the same fraction of observations above the median.

**Parameters**

- **x**: array_like
  - data values stacked for all groups
- **groups**: array_like
  - group labels or indicator

**Returns**

- **stat**: float
  - test statistic
**pvalue** : float
pvalue from the chisquare distribution

**others ????:**
currently some test output, table and expected

**statsmodels.sandbox.stats.runs.runstest_1samp**

Use runs test on binary discretized data above/below cutoff

**Parameters**
- **x** : array_like
data, numeric
- **cutoff** : {'mean', 'median'} or number
  This specifies the cutoff to split the data into large and small values.
- **correction** : bool:
  Following the SAS manual, for samplesize below 50, the test statistic is corrected by 0.5. This can be turned off with correction=False, and was included to match R, tseries, which does not use any correction.

**Returns**
- **z_stat** : float
test statistic, asymptotically normally distributed
- **p-value** : float
  p-value, reject the null hypothesis if it is below an type 1 error level, alpha.

**statsmodels.sandbox.stats.runs.runstest_2samp**

Wald-Wolfowitz runstest for two samples
This tests whether two samples come from the same distribution.

**Parameters**
- **x** : array_like
data, numeric, contains either one group, if y is also given, or both groups, if additionally a group indicator is provided
- **y** : array_like (optional)
data, numeric
- **groups** : array_like
group labels or indicator the data for both groups is given in a single 1-dimensional array, x. If group labels are not [0,1], then
- **correction** : bool:
  Following the SAS manual, for samplesize below 50, the test statistic is corrected by 0.5. This can be turned off with correction=False, and was included to match R, tseries, which does not use any correction.

**Returns**
- **z_stat** : float
test statistic, asymptotically normally distributed

**p-value** : float

p-value, reject the null hypothesis if it is below an type 1 error level, alpha.

**See also:**

`runs_test_1samp`, `Runs`, `RunsProb`

**Notes**

Wald-Wolfowitz runs test.

If there are ties, then the test statistic and p-value that is reported, is based on the higher p-value between sorting all tied observations of the same group.

This test is intended for continuous distributions SAS has treatment for ties, but not clear, and sounds more complicated (minimum and maximum possible runs prevent use of argsort) (maybe it’s not so difficult, idea: add small positive noise to first one, run test, then to the other, run test, take max(?) p-value - DONE This gives not the minimum and maximum of the number of runs, but should be close. Not true, this is close to minimum but far away from maximum. maximum number of runs would use alternating groups in the ties.) Maybe adding random noise would be the better approach.

SAS has exact distribution for sample size <=30, doesn’t look standard but should be easy to add.

Currently two-sided test only

This has not been verified against a reference implementation. In a short Monte Carlo simulation where both samples are normally distribute, the test seems to be correctly sized for larger number of observations (30 or larger), but conservative (i.e. reject less often than nominal) with a sample size of 10 in each group.

### `statsmodels.sandbox.stats.runs.cochrans_q`

`statsmodels.sandbox.stats.runs.cochrans_q(x)`

Cochran’s Q test for identical effect of k treatments

Cochran’s Q is a k-sample extension of the McNemar test. If there are only two treatments, then Cochran’s Q test and McNemar test are equivalent.

Test that the probability of success is the same for each treatment. The alternative is that at least two treatments have a different probability of success.

**Parameters**

- `x` : array_like, 2d (N,k)
  
data with N cases and k variables

**Returns**

- `q_stat` : float
  
test statistic

- `pvalue` : float
  
pvalue from the chisquare distribution

**Notes**

In Wikipedia terminology, rows are blocks and columns are treatments. The number of rows N, should be large for the chisquare distribution to be a good approximation. The Null hypothesis of the test is that all treatments have the same effect.
statsmodels Documentation, Release 0.6.0

References


statsmodels.sandbox.stats.runs.Runs
class statsmodels.sandbox.stats.runs.Runs(x)
class for runs in a binary sequence

Parameters x : array_like, 1d
data array,

Notes

This was written as a more general class for runs. This has some redundant calculations when only the runs_test
is used.

TODO: make it lazy

The runs test could be generalized to more than 1d if there is a use case for it.
This should be extended once I figure out what the distribution of runs of any length k is.
The exact distribution for the runs test is also available but not yet verified.

Methods

runs_test([correction]) basic version of runs test

statsmodels.sandbox.stats.runs.Runs.runs_test

Runs.runs_test(correction=True)
basic version of runs test

Parameters correction: bool :

Following the SAS manual, for samplesize below 50, the test statistic is corrected by
0.5. This can be turned off with correction=False, and was included to match R, tseries,
which does not use any correction.

pvalue based on normal distribution, with integer correction :

sign_test(samp[, mu0]) Signs test.

statsmodels.stats.descriptivestats.sign_test

statsmodels.stats.descriptivestats.sign_test(samp, mu0=0)
Signs test.

Parameters samp : array-like

1d array. The sample for which you want to perform the signs test.

mu0 : float
See Notes for the definition of the sign test. mu0 is 0 by default, but it is common to set it to the median.

Returns M, p-value:

See also:
scipy.stats.wilcoxon

Notes

The signs test returns

\[ M = \frac{N(+) - N(-)}{2} \]

where \( N(+) \) is the number of values above \( \mu_0 \), \( N(-) \) is the number of values below. Values equal to \( \mu_0 \) are discarded.

The p-value for \( M \) is calculated using the binomial distribution and can be interpreted the same as for a t-test. The test-statistic is distributed Binom(min(N(+), N(-)), n_trials, .5) where n_trials equals N(+) + N(-).

3.8.5 Interrater Reliability and Agreement

The main function that statsmodels has currently available for interrater agreement measures and tests is Cohen’s Kappa. Fleiss’ Kappa is currently only implemented as a measures but without associated results statistics.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cohens_kappa(table[, weights, ...])</td>
<td>Compute Cohen’s kappa with variance and equal-zero test</td>
</tr>
<tr>
<td>fleiss_kappa(table)</td>
<td>Fleiss’ kappa multi-rater agreement measure</td>
</tr>
<tr>
<td>to_table(data[, bins])</td>
<td>convert raw data with shape (subject, rater) to (rater1, rater2)</td>
</tr>
<tr>
<td>aggregate_raters(data[, n_cat])</td>
<td>convert raw data with shape (subject, rater) to (subject, cat_counts)</td>
</tr>
</tbody>
</table>

statsmodels.stats.inter_rater.cohens_kappa

statsmodels.stats.inter_rater.cohens_kappa (table, weights=None, return_results=True, wt=None)

Compute Cohen’s kappa with variance and equal-zero test

Parameters table : array_like, 2-Dim

square array with results of two raters, one rater in rows, second rater in columns

weights : array_like

The interpretation of weights depends on the wt argument. If both are None, then the simple kappa is computed. see wt for the case when wt is not None If weights is two dimensional, then it is directly used as a weight matrix. For computing the variance of kappa, the maximum of the weights is assumed to be smaller or equal to one. TODO: fix conflicting definitions in the 2-Dim case for

wt : None or string

If wt and weights are None, then the simple kappa is computed. If wt is given, but weights is None, then the weights are set to be [0, 1, 2, ..., k]. If weights is a one-dimensional array, then it is used to construct the weight matrix given the following options.

wt in ['linear', 'ca' or None] [use linear weights, Cicchetti-Allison] actual weights are linear in the score “weights” difference
wt in ['quadratic', 'fc'] [use linear weights, Fleiss-Cohen] actual weights are squared in the score “weights” difference
wt = ‘toeplitz’ [weight matrix is constructed as a toeplitz matrix] from the one dimensional weights.

return_results : bool
If True (default), then an instance of KappaResults is returned. If False, then only kappa is computed and returned.

Returns results or kappa :
If return_results is True (default), then a results instance with all statistics is returned If return_results is False, then only kappa is calculated and returned.

Notes

There are two conflicting definitions of the weight matrix, Wikipedia versus SAS manual. However, the computation are invariant to rescaling of the weights matrix, so there is no difference in the results.
Weights for ‘linear’ and ‘quadratic’ are interpreted as scores for the categories, the weights in the computation are based on the pairwise difference between the scores. Weights for ‘toeplitz’ are a interpreted as weighted distance. The distance only depends on how many levels apart two entries in the table are but not on the levels themselves.
example:
weights = ‘0, 1, 2, 3’ and wt is either linear or toeplitz means that the weighting only depends on the simple distance of levels.
weights = ‘0, 0, 1, 1’ and wt = ‘linear’ means that the first two levels are zero distance apart and the same for the last two levels. This is the same as forming two aggregated levels by merging the first two and the last two levels, respectively.
weights = [0, 1, 2, 3] and wt = ‘quadratic’ is the same as squaring these weights and using wt = ‘toeplitz’.

References

Wikipedia SAS Manual

statsmodels.stats.inter_rater.fleiss_kappa

statsmodels.stats.inter_rater.fleiss_kappa(table)
Fleiss’ kappa multi-rater agreement measure

Parameters table : array_like, 2-D
assumes subjects in rows, and categories in columns

Returns kappa : float
Fleiss’s kappa statistic for inter rater agreement
Notes

coded from Wikipedia page http://en.wikipedia.org/wiki/Fleiss%27_kappa
no variance or tests yet

statsmodels.stats.inter_rater.to_table

statsmodels.stats.inter_rater.to_table(data, bins=None)
convert raw data with shape (subject, rater) to (rater1, rater2)
brings data into correct format for cohens_kappa

Parameters data : array_like, 2-Dim
data containing category assignment with subjects in rows and raters in columns.
bins : None, int or tuple of array_like
If None, then the data is converted to integer categories, 0,1,2,...,n_cat-1. Because of the
relabeling only category levels with non-zero counts are included. If this is an integer,
then the category levels in the data are already assumed to be in integers, 0,1,2,...,n_cat-
1. In this case, the returned array may contain columns with zero count, if no subject
has been categorized with this level. If bins are a tuple of two array_like, then the
bins are directly used by numpy.histogramdd. This is useful if we want to merge
categories.

Returns arr : nd_array, (n_cat, n_cat)
Contingency table that contains counts of category level with rater1 in rows and rater2
in columns.

Notes

no NaN handling, delete rows with missing values
This works also for more than two raters. In that case the dimension of the resulting contingency table is the
same as the number of raters instead of 2-dimensional.

statsmodels.stats.inter_rater.aggregate_raters

statsmodels.stats.inter_rater.aggregate_raters(data, n_cat=None)
convert raw data with shape (subject, rater) to (subject, cat_counts)
brings data into correct format for fleiss_kappa
bincount will raise exception if data cannot be converted to integer.

Parameters data : array_like, 2-Dim
data containing category assignment with subjects in rows and raters in columns.
n_cat : None or int
If None, then the data is converted to integer categories, 0,1,2,...,n_cat-1. Because of the
relabeling only category levels with non-zero counts are included. If this is an integer,
then the category levels in the data are already assumed to be in integers, 0,1,2,...,n_cat-
1. In this case, the returned array may contain columns with zero count, if no subject
has been categorized with this level.

3.8. Statistics stats

689
Returns \texttt{arr}: nd\_array, (n\_rows, n\_cat)

Contains counts of raters that assigned a category level to individuals. Subjects are in rows, category levels in columns.

3.8.6 Multiple Tests and Multiple Comparison Procedures

\texttt{multipletests} is a function for p-value correction, which also includes p-value correction based on fdr in \texttt{fdrcorrection}. \texttt{tukeyhsd} performs simultaneous testing for the comparison of (independent) means. These three functions are verified. GroupsStats and MultiComparison are convenience classes to multiple comparisons similar to one way ANOVA, but still in development

\begin{verbatim}
multipletests\(pvals[, \alpha, \text{method}, ...]\) test results and p-value correction for multiple tests
fdrcorrection\(pvals[, \alpha, \text{method}, \text{is_sorted}]\) pvalue correction for false discovery rate
GroupsStats\(x[, \text{useranks}, \text{uni}, \text{intlab}]\) statistics by groups (another version)
MultiComparison\(\text{data, groups[, group_order]}\) Tests for multiple comparisons
TukeyHSDResults\(\text{mc\_object, results\_table, q\_crit}\) Results from Tukey HSD test, with additional plot methods
\end{verbatim}

\texttt{statsmodels.sandbox.stats.multicomp.multipletests}

\begin{verbatim}
statsmodels.sandbox.stats.multicomp.multipletests\(pvals[, \alpha=0.05, \text{method}='hs', \text{is\_sorted}=False, \text{return\_sorted}=False]\)
\end{verbatim}

\texttt{test results and p-value correction for multiple tests}

\textbf{Parameters} \texttt{pvals}: array\_like

uncorrected p-values

\texttt{alpha}: float

FWER, family-wise error rate, e.g. 0.1

\texttt{method}: string

Method used for testing and adjustment of pvalues. Can be either the full name or initial letters. Available methods are

\begin{verbatim}
'bonferroni': one-step correction
'sidak': one-step correction
'holm-sidak': step down method using Sidak adjustments
'holm': step-down method using Bonferroni adjustments
'simes-hochberg': step-up method (independent)
'hommel': closed method based on Simes tests (non-negative)
'fdr_bh': Benjamini/Hochberg (non-negative)
'fdr_by': Benjamini/Yekutieli (negative)
'fdr_tsbh': two stage fdr correction (non-negative)
'fdr_tsbky': two stage fdr correction (non-negative)
\end{verbatim}

\texttt{is\_sorted}: bool

If False (default), the p\_values will be sorted, but the corrected pvalues are in the original order. If True, then it assumed that the pvalues are already sorted in ascending order.

\texttt{return\_sorted}: bool

not tested, return sorted p-values instead of original sequence
Returns reject: array, boolean

- true for hypothesis that can be rejected for given alpha

pvals_corrected: array

- p-values corrected for multiple tests

alphacSidak: float

- corrected alpha for Sidak method

alphacBonf: float

- corrected alpha for Bonferroni method

Notes

Except for ‘fdr_twostage’, the p-value correction is independent of the alpha specified as argument. In these cases the corrected p-values can also be compared with a different alpha. In the case of ‘fdr_twostage’, the corrected p-values are specific to the given alpha, see fdr_correction_twostage.

all corrected p-values now tested against R. insufficient “cosmetic” tests yet The ‘fdr_gbs’ procedure is not verified against another package, p-values are derived from scratch and are not derived in the reference. In Monte Carlo experiments the method worked correctly and maintained the false discovery rate.

All procedures that are included, control FWER or FDR in the independent case, and most are robust in the positively correlated case.

fdr_gbs: high power, fdr control for independent case and only small violation in positively correlated case

Timing:

Most of the time with large arrays is spent in argsort. When we want to calculate the p-value for several methods, then it is more efficient to presort the p-values, and put the results back into the original order outside of the function.

Method=’hommel’ is very slow for large arrays, since it requires the evaluation of n partitions, where n is the number of p-values.

there will be API changes.

**statsmodels.sandbox.stats.multicomp.fdr_correction0**

    statsmodels.sandbox.stats.multicomp.fdr_correction0(pvals, alpha=0.05, method=‘indep’, is_sorted=False)

pvalue correction for false discovery rate

This covers Benjamini/Hochberg for independent or positively correlated and Benjamini/Yekutieli for general or negatively correlated tests. Both are available in the function multipletests, as method=‘fdr_bh’, resp. fdr_by.

Parameters pvals: array_like

- set of p-values of the individual tests.

alpha: float

- error rate

method: (‘indep’, ‘negcor’)

Returns rejected: array, bool
True if a hypothesis is rejected, False if not

$pvalue$-corrected : array

pvalues adjusted for multiple hypothesis testing to limit FDR

Notes

If there is prior information on the fraction of true hypothesis, then alpha should be set to alpha * m/m_0 where m is the number of tests, given by the p-values, and m_0 is an estimate of the true hypothesis. (see Benjamini, Krieger and Yekuteli)

The two-step method of Benjamini, Krieger and Yekutiel that estimates the number of false hypotheses will be available (soon).

Method names can be abbreviated to first letter, ‘i’ or ‘p’ for fdr_bh and ‘n’ for fdr_by.

**statsmodels.sandbox.stats.multicomp.GroupsStats**

class statsmodels.sandbox.stats.multicomp.GroupsStats(x, useranks=False, uni=None, intlab=None)

statistics by groups (another version)
groupstats as a class with lazy evaluation (not yet - decorators are still missing)
written this time as equivalent of scipy.stats.rankdata gs = GroupsStats(X, useranks=True) assert_almost_equal(gs.groupmeanfilter, stats.rankdata(X[:,0]), 15)
TODO: incomplete doc strings

Methods

- groupdemean()
- groupsswithin()
- groupvarwithin()
- runbasic([useranks])
- runbasic_old([useranks])

**statsmodels.sandbox.stats.multicomp.GroupsStats.groupdemean**

GroupsStats.groupdemean()

**statsmodels.sandbox.stats.multicomp.GroupsStats.groupsswithin**

GroupsStats.groupsswithin()

**statsmodels.sandbox.stats.multicomp.GroupsStats.groupvarwithin**

GroupsStats.groupvarwithin()
statsmodels.sandbox.stats.multicomp.GroupsStats.runbasic

GroupsStats.runbasic(useranks=False)

statsmodels.sandbox.stats.multicomp.GroupsStats.runbasic_old

GroupsStats.runbasic_old(useranks=False)

statsmodels.sandbox.stats.multicomp.MultiComparison

class statsmodels.sandbox.stats.multicomp.MultiComparison(data, groups, group_order=None)

Tests for multiple comparisons

Parameters
data: array
    independent data samples
groups: array
    group labels corresponding to each data point
group_order: list of strings, optional
    the desired order for the group mean results to be reported in. If not specified, results
    are reported in increasing order

Methods

allpairtest(testfunc[, alpha, method, pvalidx])
    run a pairwise test on all pairs with multiple test correction
getranks()
    convert data to rankdata and attach
kruskal([pairs, multimethod])
    pairwise comparison for kruskal-wallis test
tukeyhsd([alpha])
    Tukey’s range test to compare means of all pairs of groups

statsmodels.sandbox.stats.multicomp.MultiComparison.allpairtest

MultiComparison.allpairtest(testfunc, alpha=0.05, method=‘bonf’, pvalidx=1)
    run a pairwise test on all pairs with multiple test correction

    The statistical test given in testfunc is calculated for all pairs and the p-values are adjusted by methods
    in multipletests. The p-value correction is generic and based only on the p-values, and does not take any
    special structure of the hypotheses into account.

    Parameters
testfunc: function
        A test function for two (independent) samples. It is assumed that the return value on
        position pvalidx is the p-value.
alpha: float
        familywise error rate
method: string
        This specifies the method for the p-value correction. Any method of multipletests is
        possible.
\texttt{pvalidx} : int (default: 1)
position of the p-value in the return of testfunc

Returns \texttt{sumtab} : SimpleTable instance
summary table for printing

\texttt{errors:} TODO: check if this is still wrong, I think it’s fixed.
results from multipletests are in different order:
pval\_corrected can be larger than 1 ??? :

\texttt{statsmodels.sandbox.stats.multicomp.MultiComparison.getranks}

\texttt{MultiComparison.getranks()}  
convert data to rankdata and attach
This creates rankdata as it is used for non-parametric tests, where in the case of ties the average rank is assigned.

\texttt{statsmodels.sandbox.stats.multicomp.MultiComparison.kruskal}

\texttt{MultiComparison.kruskal(pairs=None, multimethod='T')}  
pairwise comparison for kruskal-wallis test
This is just a reimplementation of scipy.stats.kruskal and does not yet use a multiple comparison correction.

\texttt{statsmodels.sandbox.stats.multicomp.MultiComparison.tukeyhsd}

\texttt{MultiComparison.tukeyhsd(alpha=0.05)}  
Tukey’s range test to compare means of all pairs of groups

Parameters \texttt{alpha} : float, optional
Value of FWER at which to calculate HSD.

Returns \texttt{results} : TukeyHSDResults instance
A results class containing relevant data and some post-hoc calculations

\texttt{statsmodels.sandbox.stats.multicomp.TukeyHSDResults}

class \texttt{statsmodels.sandbox.stats.multicomp.TukeyHSDResults}(mc\_object, \texttt{results\_table}, \texttt{q\_crit}, \texttt{reject=None}, \texttt{meandiffs=None}, \texttt{std\_pairs=None}, \texttt{conf\_int=None}, \texttt{df\_total=None}, \texttt{reject2=None}, \texttt{variance=None})

Results from Tukey HSD test, with additional plot methods
Can also compute and plot additional post-hoc evaluations using this results class.
Notes

halfwidths is only available after call to plot_simultaneous.

Other attributes contain information about the data from the MultiComparison instance: data, df_total, groups, groupsunique, variance.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reject</td>
<td>array of boolean, True if we reject Null for group pair</td>
</tr>
<tr>
<td>meandiffs</td>
<td>pairwise mean differences</td>
</tr>
<tr>
<td>confint</td>
<td>confidence interval for pairwise mean differences</td>
</tr>
<tr>
<td>std_pairs</td>
<td>standard deviation of pairwise mean differences</td>
</tr>
<tr>
<td>q_crit</td>
<td>critical value of studentized range statistic at given alpha</td>
</tr>
<tr>
<td>halfwidths</td>
<td>half widths of simultaneous confidence interval</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_simultaneous</td>
<td>Plot a universal confidence interval of each group mean</td>
</tr>
<tr>
<td>summary</td>
<td>Summary table that can be printed</td>
</tr>
</tbody>
</table>

statsmodels.sandbox.stats.multicomp.TukeyHSDResults.plot_simultaneous

TukeyHSDResults.plot_simultaneous(comparison_name=None, ax=None, figsize=(10, 6), xlabel=None, ylabel=None)

Plot a universal confidence interval of each group mean

Visualize significant differences in a plot with one confidence interval per group instead of all pairwise confidence intervals.

Parameters

- **comparison_name**: string, optional
  - if provided, plot_intervals will color code all groups that are significantly different from the comparison_name red, and will color code insignificant groups gray. Otherwise, all intervals will just be plotted in black.

- **ax**: matplotlib axis, optional
  - An axis handle on which to attach the plot.

- **figsize**: tuple, optional
  - tuple for the size of the figure generated

- **xlabel**: string, optional
  - Name to be displayed on x axis

- **ylabel**: string, optional
  - Name to be displayed on y axis

Returns

- **fig**: Matplotlib Figure object
  - handle to figure object containing interval plots
Notes

Multiple comparison tests are nice, but lack a good way to be visualized. If you have, say, 6 groups, showing a graph of the means between each group will require 15 confidence intervals. Instead, we can visualize inter-group differences with a single interval for each group mean. Hochberg et al. [1] first proposed this idea and used Tukey’s Q critical value to compute the interval widths. Unlike plotting the differences in the means and their respective confidence intervals, any two pairs can be compared for significance by looking for overlap.

References

[R6]

Examples

```python
>>> from statsmodels.examples.try_tukey_hsd import cylinders, cyl_labels
>>> from statsmodels.stats.multicomp import MultiComparison
>>> cardata = MultiComparison(cylinders, cyl_labels)
>>> results = cardata.tukeyhsd()
>>> results.plot_simultaneous()
<matplotlib.figure.Figure at 0x...>
```

This example shows an example plot comparing significant differences in group means. Significant differences at the alpha=0.05 level can be identified by intervals that do not overlap (i.e. USA vs Japan, USA vs Germany).

```python
>>> results.plot_simultaneous(comparison_name="USA")
<matplotlib.figure.Figure at 0x...>
```

Optionally provide one of the group names to color code the plot to highlight group means different from comparison_name.

```python
def TukeyHSDResults.summary()
Summary table that can be printed
```
**alpha** : float

significance level for the test

**Returns results** : TukeyHSDResults instance

A results class containing relevant data and some post-hoc calculations

**See also:**

MultiComparison, tukeyhsd, statsmodels.sandbox.stats.multicomp.TukeyHSDResults

The following functions are not (yet) public

```python
varcorrection_pairs_unbalanced(nobs_all[, ...])
varcorrection_pairs_unequal(var_all, ...)
varcorrection_unbalanced(nobs_all[, range])
StepDown(var_all, nobs_all, df_all)
```

**correction factor for variance with unequal sample sizes for all pairs**

**return joint variance from samples with unequal variances and unequal**

**correction factor for variance with unequal sample sizes**

**return joint variance from samples with unequal variances and unequal**

**a class for step down methods**

```python
catstack(args)
ccoils
compare_ordered(vals, alpha)
distance_st_range(mean_all, nobs_all, var_all)
ecdf(x)
get_tukeyQcrit(k, df[, alpha])
homogeneous_subsets(vals, dcrit)
line
maxzero(x)
maxzerodown(x)
mcfdr([nrepl, nobs, ntests, ntrue, mu, ...])
qcrit
randmvn(rho[, size, standardize])
rankdata(x)
rejectionline(n[, alpha])
set_partition(ssli)
set_remove_subs(ssli)
tiecorrect(xranks)
```

**simple ordered sequential comparison of means**

**pairwise distance matrix, outsourced from tukeyhsd**

**no frills empirical cdf used in fdrcorrection**

**return critical values for Tukey’s HSD (Q)**

**recursively check all pairs of vals for minimum distance**

**str(object) -> string**

**find all up zero crossings and return the index of the highest**

**find all up zero crossings and return the index of the highest**

**MonteCarlo to test fdrcorrection**

**str(object) -> string**

**create random draws from equi-correlated multivariate normal distribution**

**rankdata, equivalent to scipy.stats.rankdata**

**reference line for rejection in multiple tests**

**extract a partition from a list of tuples**

**remove sets that are subsets of another set from a list of tuples**

**should be equivalent of scipy.stats.tiecorrect**

**statsmodels.sandbox.stats.multicomp.varcorrection_pairs_unbalanced**

**correction factor for variance with unequal sample sizes for all pairs**

**this is just a harmonic mean**

**Parameters nobs_all** : array_like

The number of observations for each sample

**srangr** : bool

if true, then the correction is divided by 2 for the variance of the studentized range statistic

**Returns correction** : array

Correction factor for variance.
Notes

variance correction factor is
\[ \frac{1}{k} \sum_{i=0}^{k-1} \frac{1}{n_i} \]
where k is the number of samples and summation is over i=0,...,k-1. If all n_i are the same, then the correction factor is 1.

This needs to be multiplied by the joint variance estimate, means square error, MSE. To obtain the correction factor for the standard deviation, square root needs to be taken.

For the studentized range statistic, the resulting factor has to be divided by 2.

\texttt{statsmodels.sandbox.stats.multicomp.varcorrection_pairs_unequal}

\texttt{statsmodels.sandbox.stats.multicomp.varcorrection_pairs_unequal(var\_all, nobs\_all, df\_all)}

return joint variance from samples with unequal variances and unequal sample sizes for all pairs something is wrong

\textbf{Parameters}
- \texttt{var\_all} : array_like
  The variance for each sample
- \texttt{nobs\_all} : array_like
  The number of observations for each sample
- \texttt{df\_all} : array_like
  degrees of freedom for each sample

\textbf{Returns}
- \texttt{varjoint} : array
  joint variance.
- \texttt{dfjoint} : array
  joint Satterthwait's degrees of freedom

Notes

(copy, paste not correct) variance is
\[ \frac{1}{k} \sum_{i=0}^{k-1} \frac{1}{n_i} \]
where k is the number of samples and summation is over i=0,...,k-1. If all n_i are the same, then the correction factor is 1.

This needs to be multiplied by the joint variance estimate, means square error, MSE. To obtain the correction factor for the standard deviation, square root needs to be taken.

TODO: something looks wrong with dfjoint, is formula from SPSS
correction factor for variance with unequal sample sizes

this is just a harmonic mean

**Parameters**

nobs_all : array_like

The number of observations for each sample

srange : bool

if true, then the correction is divided by the number of samples for the variance of the studentized range statistic

**Returns**

correction : float

Correction factor for variance.

**Notes**

variance correction factor is

\[
\frac{1}{k} \sum_{i=0}^{k-1} \frac{1}{n_i}
\]

where k is the number of samples and summation is over i=0,...,k-1. If all n_i are the same, then the correction factor is 1.

This needs to be multiplied by the joint variance estimate, means square error, MSE. To obtain the correction factor for the standard deviation, square root needs to be taken.

\[\text{return joint variance from samples with unequal variances and unequal sample sizes}\]

\[\text{something is wrong}\]

**Parameters**

var_all : array_like

The variance for each sample

nobs_all : array_like

The number of observations for each sample

df_all : array_like

degrees of freedom for each sample

**Returns**

varjoint : float

joint variance.

dfjoint : float

joint Satterthwait’s degrees of freedom
Notes

(copy, paste not correct) variance is

\[
\frac{1}{k} \sum_{i} \frac{1}{n_i}
\]

where \( k \) is the number of samples and summation is over \( i=0,...,k-1 \). If all \( n_i \) are the same, then the correction factor is \( \frac{1}{n} \).

This needs to be multiplies by the joint variance estimate, means square error, MSE. To obtain the correction factor for the standard deviation, square root needs to be taken.

This is for variance of mean difference not of studentized range.

```
statsmodels.sandbox.stats.multicomp.StepDown
class statsmodels.sandbox.stats.multicomp.StepDown(vals, nobs_all, var_all, df=None):
    a class for step down methods

    This is currently for simple tree subset descend, similar to homogeneous_subsets, but checks all leave-one-out subsets instead of assuming an ordered set. Comment in SAS manual: SAS only uses interval subsets of the sorted list, which is sufficient for range tests (maybe also equal variance and balanced sample sizes are required). For F-test based critical distances, the restriction to intervals is not sufficient.

    This version uses a single critical value of the studentized range distribution for all comparisons, and is therefore a step-down version of Tukey HSD. The class is written so it can be subclassed, where the get_distance_matrix and get_crit are overwritten to obtain other step-down procedures such as REGW.

    iter_subsets can be overwritten, to get a recursion as in the many to one comparison with a control such as in Dunnet’s test.

    A one-sided right tail test is not covered because the direction of the inequality is hard coded in check_set. Also Peritz’s check of partitions is not possible, but I have not seen it mentioned in any more recent references. I have only partially read the step-down procedure for closed tests by Westfall.

    One change to make it more flexible, is to separate out the decision on a subset, also because the F-based tests, FREGW in SPSS, take information from all elements of a set and not just pairwise comparisons. I haven’t looked at the details of the F-based tests such as Sheffe yet. It looks like running an F-test on equality of means in each subset. This would also outsource how pairwise conditions are combined, any larger or max. This would also imply that the distance matrix cannot be calculated in advance for tests like the F-based ones.

Methods

```
check_set(indices)
check whether pairwise distances of indices satisfy condition
get_crit(alpha)
studentized range statistic
get_distance_matrix()
iter_subsets(indices)
run(alpha)
stepdown(indices)main function to run the test,
```

```
statsmodels.sandbox.stats.multicomp.StepDown.check_set

StepDown.check_set(indices)
check whether pairwise distances of indices satisfy condition
```
statsmodels.sandbox.stats.multicomp.StepDown.get_crit

StepDown.get_crit(alpha)

statsmodels.sandbox.stats.multicomp.StepDown.get_distance_matrix

StepDown.get_distance_matrix()
studentized range statistic

statsmodels.sandbox.stats.multicomp.StepDown.iter_subsets

StepDown.iter_subsets(indices)

statsmodels.sandbox.stats.multicomp.StepDown.run

StepDown.run(alpha)
main function to run the test,
could be done in __call__ instead this could have all the initialization code

statsmodels.sandbox.stats.multicomp.StepDown.stepdown

StepDown.stepdown(indices)

statsmodels.sandbox.stats.multicomp.catstack

statsmodels.sandbox.stats.multicomp.catstack(args)

statsmodels.sandbox.stats.multicomp.ccols

statsmodels.sandbox.stats.multicomp.ccols = array([2, 3, 4, 5, 6, 7, 8, 9, 10])

statsmodels.sandbox.stats.multicomp.compare_ordered

statsmodels.sandbox.stats.multicomp.compare_ordered(vals, alpha)
simple ordered sequential comparison of means
vals [array_like] means or rankmeans for independent groups
incomplete, no return, not used yet

statsmodels.sandbox.stats.multicomp.distance_st_range

statsmodels.sandbox.stats.multicomp.distance_st_range(mean_all, nobs_all, var_all,
df=None, triu=False)
pairwise distance matrix, outsourced from tukeyhsd
CHANGED: meandiffs are with sign, studentized range uses abs
q_crit added for testing

3.8. Statistics stats
TODO: error in variance calculation when nobs_all is scalar, missing 1/n

**statsmodels.sandbox.stats.multicomp.ecdf**

```
statsmodels.sandbox.stats.multicomp.ecdf(x)
```

no frills empirical cdf used in fdr correction

**statsmodels.sandbox.stats.multicomp.get_tukeyQcrit**

```
statsmodels.sandbox.stats.multicomp.get_tukeyQcrit(k, df, alpha=0.05)
```

return critical values for Tukey’s HSD (Q)

- **Parameters**
  - `k`: int in `{2, ..., 10}`
    - number of tests
  - `df`: int
    - degrees of freedom of error term
  - `alpha`: `{0.05, 0.01}`
    - type 1 error, 1-confidence level
  - **not enough error checking for limitations**:

**statsmodels.sandbox.stats.multicomp.homogeneous_subsets**

```
statsmodels.sandbox.stats.multicomp.homogeneous_subsets(vals, dcrit)
```

recursively check all pairs of vals for minimum distance

- step down method as in Newman-Keuls and Ryan procedures. This is not a closed procedure since not all partitions are checked.

- **Parameters**
  - `vals`: array_like
    - values that are pairwise compared
  - `dcrit`: array_like or float
    - critical distance for rejecting, either float, or 2-dimensional array with distances on the upper triangle.

- **Returns**
  - `rejs`: list of pairs
    - list of pair-indices with (strictly) larger than critical difference
  - `nrejs`: list of pairs
    - list of pair-indices with smaller than critical difference
  - `lli`: list of tuples
    - list of subsets with smaller than critical difference
  - `res`: tree
    - result of all comparisons (for checking)
this follows description in SPSS notes on Post-Hoc Tests:

Because of the recursive structure, some comparisons are made several times, but only unique pairs or sets are returned.

Examples

```python
>>> m = [0, 2, 2.5, 3, 6, 8, 9, 9.5, 10]
>>> rej, nrej, ssli, res = homogeneous_subsets(m, 2)
>>> set_partition(ssli)
([(5, 6, 7, 8), (1, 2, 3), (4,)], [0])
>>> [np.array(m)[list(pp)] for pp in set_partition(ssli)[0]]
[array([ 8. , 9. , 9.5, 10. ]), array([ 2. , 2.5, 3. ]), array([ 6. ])]
```

**statsmodels.sandbox.stats.multicomp.line**

*statsmodels.sandbox.stats.multicomp.line = 'str(object) -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

**statsmodels.sandbox.stats.multicomp.maxzero**

*statsmodels.sandbox.stats.multicomp.maxzero(x)*

find all up zero crossings and return the index of the highest

Not used anymore

```python
>>> np.random.seed(12345)
>>> x = np.random.randn(8)
>>> x
array([-0.20470766, 0.47894334, -0.51943872, -0.5557303, 1.96578057,
       1.39340583, 0.09290788, 0.28174615])
>>> maxzero(x)
(4, array([1, 4]))
```

no up-zero-crossing at end

```python
>>> np.random.seed(0)
>>> x = np.random.randn(8)
>>> x
array([ 1.76405235, 0.40015721, 0.97873798, 2.2408932, 1.86755799,
       -0.97727788, 0.95008842, -0.15135721])
>>> maxzero(x)
(None, array([6]))
```

**statsmodels.sandbox.stats.multicomp.maxzerodown**

*statsmodels.sandbox.stats.multicomp.maxzerodown(x)*

find all up zero crossings and return the index of the highest

Not used anymore
```python
>>> np.random.seed(12345)
>>> x = np.random.randn(8)
>>> x
array([-0.20470766, 0.47894334, -0.51943872, -0.5557303 , 1.96578057,
        1.39340583, 0.09290788, 0.28174615])
>>> maxzero(x)
(4, array([1, 4]))
no up-zero-crossing at end

>>> np.random.seed(0)
>>> x = np.random.randn(8)
>>> x
array([ 1.76405235, 0.40015721, 0.97873798, 2.2408932 , 1.86755799,
         -0.97727788, 0.95008842, -0.15135721])
>>> maxzero(x)
(None, array([6]))
```

**statsmodels.sandbox.stats.multicomp.mcfdr**

```python
statsmodels.sandbox.stats.multicomp.mcfdr(nrepl=100, nobsl=50, ntests=10, ntrue=6, mu=0.5, alpha=0.05, rho=0.0)
```
MonteCarlo to test fdrcorrection

**statsmodels.sandbox.stats.multicomp.qcrit**

```python
statsmodels.sandbox.stats.multicomp.qcrit = ['
 2 3 4 5 6 7 8 9 10
6 ... 5.21 4.56 5.30
infinity 2.77 3.64 3.31 4.12 3.63 4.40 3.86 4.60 4.03 4.76 4.17 4.88 4.29 4.99 4.39 5.08 4.47 5.16
''
```

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

**statsmodels.sandbox.stats.multicomp.randmvn**

```python
statsmodels.sandbox.stats.multicomp.randmvn(rho, size=(1, 2), standardize=False)
create random draws from equi-correlated multivariate normal distribution
```

**Parameters**

- `rho`: float
  correlation coefficient
- `size`: tuple of int
  size is interpreted (nobs, nvars) where each row

**Returns**

- `rvs`: ndarray, (nobs, nvars)
  where each row is a independent random draw of nvars-dimensional correlated rvs

**statsmodels.sandbox.stats.multicomp.rankdata**

```python
statsmodels.sandbox.stats.multicomp.rankdata(x)
rankdata, equivalent to scipy.stats.rankdata
just a different implementation, I have not yet compared speed
```
statsmodels.sandbox.stats.multicomp.rejectionline

statsmodels.sandbox.stats.multicomp.rejectionline\(n, \alpha=0.5\)
reference line for rejection in multiple tests
Not used anymore
from: section 3.2, page 60

statsmodels.sandbox.stats.multicomp.set_partition

statsmodels.sandbox.stats.multicomp.set_partition\(ssli\)
extact a partition from a list of tuples

this should be correctly called select largest disjoint sets. Begun and Gabriel 1981 don’t seem to be bothered by
sets of accepted hypothesis with joint elements, e.g. maximal_accepted_sets = \{ \{1,2,3\}, \{2,3,4\} \}

This creates a set partition from a list of sets given as tuples. It tries to find the partition with the largest sets.
That is, sets are included after being sorted by length.

If the list doesn’t include the singletons, then it will be only a partial partition. Missing items are singletons (I
think).

Examples

```python
>>> li
[(5, 6, 7, 8), (1, 2, 3), (4, 5), (0, 1)]
>>> set_partition(li)
([(5, 6, 7, 8), (1, 2, 3)], [0, 4])
```

statsmodels.sandbox.stats.multicomp.set_remove_subs

statsmodels.sandbox.stats.multicomp.set_remove_subs\(ssli\)
remove sets that are subsets of another set from a list of tuples

Parameters  ssli : list of tuples
each tuple is considered as a set

Returns  part : list of tuples
new list with subset tuples removed, it is sorted by set-length of tuples. The list contains
original tuples, duplicate elements are not removed.

Examples

```python
>>> set_remove_subs([(0, 1), (1, 2), (1, 2, 3), (0,)])
[(1, 2, 3), (0, 1)]
>>> set_remove_subs([(0, 1), (1, 2), (1, 1, 1, 2, 3), (0,)])
[(1, 1, 1, 2, 3), (0, 1)]
```
3.8.7 Basic Statistics and t-Tests with frequency weights

Besides basic statistics, like mean, variance, covariance and correlation for data with case weights, the classes here provide one and two sample tests for means. The t-tests have more options than those in scipy.stats, but are more restrictive in the shape of the arrays. Confidence intervals for means are provided based on the same assumptions as the t-tests.

Additionally, tests for equivalence of means are available for one sample and for two, either paired or independent, samples. These tests are based on TOST, two one-sided tests, which have as null hypothesis that the means are not “close” to each other.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DescrStatsW</td>
<td>descriptive statistics and tests with weights for case weights</td>
</tr>
<tr>
<td>CompareMeans</td>
<td>class for two sample comparison</td>
</tr>
<tr>
<td>ttest_ind</td>
<td>test independent sample</td>
</tr>
<tr>
<td>ttost_ind</td>
<td>test of (non-)equivalence for two independent samples</td>
</tr>
<tr>
<td>ttost_paired</td>
<td>test of (non-)equivalence for two dependent, paired sample</td>
</tr>
<tr>
<td>ztest</td>
<td>test for mean based on normal distribution, one or two samples</td>
</tr>
<tr>
<td>ztost</td>
<td>Equivalence test based on normal distribution</td>
</tr>
<tr>
<td>zconfint</td>
<td>confidence interval based on normal distribution z-test</td>
</tr>
</tbody>
</table>

**statsmodels.stats.weightstats.DescrStatsW**

**class** statsmodels.stats.weightstats.DescrStatsW(data[, weights=\(None\), ddof=\(0\)]

descriptive statistics and tests with weights for case weights

Assumes that the data is 1d or 2d with (nobs, nvars) observations in rows, variables in columns, and that the same weight applies to each column.

If degrees of freedom correction is used, then weights should add up to the number of observations. ttest also assumes that the sum of weights corresponds to the sample size.

This is essentially the same as replicating each observation by its weight, if the weights are integers, often called case or frequency weights.

**Parameters**

- **data**: array_like, 1-D or 2-D
  dataset

- **weights**: None or 1-D ndarray
  weights for each observation, with same length as zero axis of data

- **ddof**: int
  default ddof=0, degrees of freedom correction used for second moments, var, std, cov, corrcoef. However, statistical tests are independent of ddof, based on the standard formulas.

**Examples**

Note: I don’t know the seed for the following, so the numbers will differ
>>> x1_2d = 1.0 + np.random.randn(20, 3)
>>> w1 = np.random.randint(1,4, 20)
>>> d1 = DescrStatsW(x1_2d, weights=w1)

>>> d1.mean
array([ 1.42739844, 1.23174284, 1.083753  ])

>>> d1.var
array([ 0.94855633, 0.52074626, 1.12309325])

>>> d1.std_mean
array([ 0.14682676, 0.10878944, 0.15976497])

>>> tstat, pval, df = d1.ttest_mean(0)
>>> tstat; pval; df
array([ 9.72165021, 11.32226471, 6.78342055])
array([ 1.58414212e-12, 1.26536887e-14, 2.37623126e-08])
44.0

>>> tstat, pval, df = d1.ttest_mean([0, 1, 1])
>>> tstat; pval; df
array([ 9.72165021, 2.13019609, 0.52422632])
array([ 1.58414212e-12, 3.87842808e-02, 6.02752170e-01])
44.0

# if weights are integers, then asrepeats can be used

>>> xlr = d1.asrepeats()
>>> xlr.shape
...
>>> stats.ttest_1samp(xlr, [0, 1, 1])
...

Methods

- `asrepeats()`: get array that has repeats given by floor(weights)
- `corrcoef()`: weighted correlation with default ddof
- `cov()`: weighted covariance of data if data is 2 dimensional
- `demeaned()`: data with weighted mean subtracted
- `get_compare(other[, weights])`: return an instance of CompareMeans with self and other
- `mean()`: weighted mean of data
- `nobs()`: alias for number of observations/cases, equal to sum of weights
- `std()`: standard deviation with default degrees of freedom correction
- `std_ddof([ddof])`: standard deviation of data given ddof
- `std_mean()`: standard deviation of weighted mean
- `sum()`: weighted sum of data
- `sum_weights()`: weighted sum of squares of demeaned data
- `sumsquares()`: two-sided confidence interval for weighted mean of data
- `tconfint_mean([alpha, alternative])`: two-sided confidence interval for weighted mean of data
- `ttest_mean([value, alternative])`: test of Null hypothesis that mean is equal to value.
- `ttost_mean(low, upp)`: test of (non-)equivalence of one sample
- `var()`: variance with default degrees of freedom correction
- `var_ddof([ddof])`: variance of data given ddof
- `zconfint_mean([alpha, alternative])`: two-sided confidence interval for weighted mean of data
- `ztest_mean([value, alternative])`: z-test of Null hypothesis that mean is equal to value.
- `ztost_mean(low, upp)`: test of (non-)equivalence of one sample, based on z-test
DescrStatsW.asrepeats()  
get array that has repeats given by floor(weights)  
observations with weight=0 are dropped

DescrStatsW.corrcoef()  
weighted correlation with default ddof  
assumes variables in columns and observations in rows

DescrStatsW.cov()  
weighted covariance of data if data is 2 dimensional  
assumes variables in columns and observations in rows uses default ddof

DescrStatsW.demeaned()  
data with weighted mean subtracted

DescrStatsW.get_compare(other, weights=None)  
return an instance of CompareMeans with self and other  
    Parameters other : array_like or instance of DescrStatsW  
        If array_like then this creates an instance of DescrStatsW with the given weights.  
    weights : None or array  
        weights are only used if other is not an instance of DescrStatsW  
    Returns cm : instance of CompareMeans  
        the instance has self attached as d1 and other as d2.

See also:  
    CompareMeans

DescrStatsW.mean()  
weighted mean of data
statsmodels.stats.weightstats.DescrStatsW.nobs

```python
static DescrStatsW.nobs()  
alias for number of observations/cases, equal to sum of weights
```

statsmodels.stats.weightstats.DescrStatsW.std

```python
static DescrStatsW.std()  
standard deviation with default degrees of freedom correction
```

statsmodels.stats.weightstats.DescrStatsW.std_ddof

```python
DescrStatsW.std_ddof(ddof=0)  
standard deviation of data with given ddof

Parameters
- ddof : int, float
  degrees of freedom correction, independent of attribute ddof

Returns
- std : float, ndarray
  standard deviation with denominator sum_weights - ddof
```

statsmodels.stats.weightstats.DescrStatsW.std_mean

```python
static DescrStatsW.std_mean()  
standard deviation of weighted mean
```

statsmodels.stats.weightstats.DescrStatsW.sum

```python
static DescrStatsW.sum()  
weighted sum of data
```

statsmodels.stats.weightstats.DescrStatsW.sum_weights

```python
static DescrStatsW.sum_weights()  
```

statsmodels.stats.weightstats.DescrStatsW.sumssquares

```python
static DescrStatsW.sumssquares()  
weighted sum of squares of demeaned data
```

statsmodels.stats.weightstats.DescrStatsW.tconfint_mean

```python
DescrStatsW.tconfint_mean(alpha=0.05, alternative='two-sided')  
two-sided confidence interval for weighted mean of data

If the data is 2d, then these are separate confidence intervals for each column.

Parameters
- alpha : float
```

3.8. Statistics stats
significance level for the confidence interval, coverage is $1 - \alpha$

**alternative** : string

This specifies the alternative hypothesis for the test that corresponds to the confidence interval. The alternative hypothesis, $H_1$, has to be one of the following:

- 'two-sided': $H_1$: mean not equal to value (default)
- 'larger': $H_1$: mean larger than value
- 'smaller': $H_1$: mean smaller than value

**Returns** lower, upper : floats or ndarrays

lower and upper bound of confidence interval

**Notes**

In a previous version, statsmodels 0.4, alpha was the confidence level, e.g. 0.95

**statsmodels.stats.weightstats.DescrStatsW.ttest_mean**

DescrStatsW.ttest_mean(value=0, alternative='two-sided')

test of Null hypothesis that mean is equal to value.

The alternative hypothesis $H_1$ is defined by the following:

- 'two-sided': $H_1$: mean not equal to value (default)
- 'larger': $H_1$: mean larger than value
- 'smaller': $H_1$: mean smaller than value

**Parameters** value : float or array

the hypothesized value for the mean

alternative : string

The alternative hypothesis, $H_1$, has to be one of the following:

- 'two-sided': $H_1$: mean not equal to value (default)
- 'larger': $H_1$: mean larger than value
- 'smaller': $H_1$: mean smaller than value

**Returns** tstat : float

test statistic

pvalue : float

pvalue of the t-test

df : int or float

**statsmodels.stats.weightstats.DescrStatsW.ttost_mean**

DescrStatsW.ttost_mean(low, upp)

test of (non-)equivalence of one sample

TOST: two one-sided t tests

null hypothesis: $m < \text{low}$ or $m > \text{upp}$ alternative hypothesis: $\text{low} < m < \text{upp}$

where $m$ is the expected value of the sample (mean of the population).

If the pvalue is smaller than a threshold, say 0.05, then we reject the hypothesis that the expected value of the sample (mean of the population) is outside of the interval given by thresholds low and upp.

**Parameters** low, upp : float
equivalence interval low < mean < upp

**Returns**

- **pvalue**: float
  
  pvalue of the non-equivalence test

- **t1, pv1, df1**: tuple
  
  test statistic, pvalue and degrees of freedom for lower threshold test

- **t2, pv2, df2**: tuple
  
  test statistic, pvalue and degrees of freedom for upper threshold test

---

**statsmodels.stats.weightstats.DescrStatsW.var**

*static* `DescrStatsW.var()`

variance with default degrees of freedom correction

**statsmodels.stats.weightstats.DescrStatsW.var_ddof**

`DescrStatsW.var_ddof(ddof=0)`

variance of data given ddof

**Parameters**

- **ddof**: int, float
  
  degrees of freedom correction, independent of attribute ddof

**Returns**

- **var**: float, ndarray
  
  variance with denominator `sum_weights - ddof`

**statsmodels.stats.weightstats.DescrStatsW.zconfint_mean**

`DescrStatsW.zconfint_mean(alpha=0.05, alternative='two-sided')`

two-sided confidence interval for weighted mean of data

Confidence interval is based on normal distribution. If the data is 2d, then these are separate confidence intervals for each column.

**Parameters**

- **alpha**: float
  
  significance level for the confidence interval, coverage is 1-alpha

- **alternative**: string
  
  This specifies the alternative hypothesis for the test that corresponds to the confidence interval. The alternative hypothesis, H1, has to be one of the following

  - ‘two-sided’: H1: mean not equal to value (default) ‘larger’ : H1: mean larger than value ‘smaller’ : H1: mean smaller than value

**Returns**

- **lower, upper**: floats or ndarrays
  
  lower and upper bound of confidence interval

**Notes**

In a previous version, statsmodels 0.4, alpha was the confidence level, e.g. 0.95
**statsmodels.stats.weightstats.DescrStatsW.ztest_mean**

DescrStatsW.ztest_mean(value=0, alternative='two-sided')

Z-test of Null hypothesis that mean is equal to value.

The alternative hypothesis H1 is defined by the following: H1: mean not equal to value ‘larger’ 
H1: mean larger than value ‘smaller’ H1: mean smaller than value

**Parameters**

- **value** : float or array
  - the hypothesized value for the mean
- **alternative** : string
  - The alternative hypothesis, H1, has to be one of the following
    - ‘two-sided’: H1: mean not equal to value (default) ‘larger’: H1: mean larger than value ‘smaller’: H1: mean smaller than value

**Returns**

- **tstat** : float
  - test statistic
- **pvalue** : float
  - pvalue of the t-test

**Notes**

This uses the same degrees of freedom correction as the t-test in the calculation of the standard error of the mean, i.e it uses \((\text{sum_weights} - 1)\) instead of \(\text{sum_weights}\) in the denominator. See Examples below for the difference.

**Examples**

Z-test on a proportion, with 20 observations, 15 of those are our event

```python
>>> x1 = [0, 1]
>>> w1 = [5, 15]
>>> d1 = smws.DescrStatsW(x1, w1)
>>> d1.ztest_mean(0.5)
(2.5166114784235836, 0.011848940928347452)
```

This differs from the proportions_ztest because of the degrees of freedom correction:

```python
>>> smprop.proportions_ztest(15, 20., value=0.5)
(2.5819888974716112, 0.009823274507519247).
```

We can replicate the results from proportions_ztest if we increase the weights to have artificially one more observation:

```python
>>> smws.DescrStatsW(x1, np.array(w1)*21./20).ztest_mean(0.5)
(2.5819888974716116, 0.0098232745075192366)
```
TOST: two one-sided z-tests

null hypothesis: \( m < \text{low} \) or \( m > \text{upp} \)
alternative hypothesis: \( \text{low} < m < \text{upp} \)

where \( m \) is the expected value of the sample (mean of the population).

If the p-value is smaller than a threshold, say 0.05, then we reject the hypothesis that the expected value of the sample (mean of the population) is outside of the interval given by thresholds low and upp.

**Parameters**
- **low, upp**: float
  - equivalence interval \( \text{low} < \text{mean} < \text{upp} \)

**Returns**
- **pvalue**: float
  - p-value of the non-equivalence test
- **t1, pv1**: tuple
  - test statistic and p-value for lower threshold test
- **t2, pv2**: tuple
  - test statistic and p-value for upper threshold test

---

**statsmodels.stats.weightstats.CompareMeans**

**class**

```python
statsmodels.stats.weightstats.CompareMeans(d1, d2)
```

class for two sample comparison

The tests and the confidence interval work for multi-endpoint comparison: If \( d1 \) and \( d2 \) have the same number of rows, then each column of the data in \( d1 \) is compared with the corresponding column in \( d2 \).

**Parameters**
- **d1, d2**: instances of DescrStatsW

**Notes**

The result for the statistical tests and the confidence interval are independent of the user specified ddof.

**TODO**: Extend to any number of groups or write a version that works in that case, like in SAS and SPSS.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dof_satt()</td>
<td>degrees of freedom of Satterthwaite for unequal variance</td>
</tr>
<tr>
<td>std_meandiff_pooledvar()</td>
<td>variance assuming equal variance in both data sets</td>
</tr>
<tr>
<td>std_meandiff_separatevar()</td>
<td></td>
</tr>
<tr>
<td>tconfint_diff([alpha, alternative, usevar])</td>
<td>confidence interval for the difference in means</td>
</tr>
<tr>
<td>ttest_ind([alternative, usevar, value])</td>
<td>test for the null hypothesis of identical means</td>
</tr>
<tr>
<td>ttest_ind([low, upp[, usevar]])</td>
<td>test of equivalence for two independent samples, based on t-test</td>
</tr>
<tr>
<td>zconfint_diff([alpha, alternative, usevar])</td>
<td>confidence interval for the difference in means</td>
</tr>
<tr>
<td>ztest_ind([alternative, usevar, value])</td>
<td>z-test for the null hypothesis of identical means</td>
</tr>
<tr>
<td>ztost_ind([low, upp[, usevar]])</td>
<td>test of equivalence for two independent samples, based on z-test</td>
</tr>
</tbody>
</table>

---

**statsmodels.stats.weightstats.CompareMeans.dof_satt**

```python
CompareMeans.dof_satt()
```

degrees of freedom of Satterthwaite for unequal variance
\texttt{statsmodels.stats.weightstats.CompareMeans.std_meandiff_pooledvar}

\texttt{static CompareMeans.std_meandiff_pooledvar()}

variance assuming equal variance in both data sets

\texttt{statsmodels.stats.weightstats.CompareMeans.std_meandiff_separatevar}

\texttt{static CompareMeans.std_meandiff_separatevar()}

\texttt{statsmodels.stats.weightstats.CompareMeans.tconfint_diff}

\texttt{CompareMeans.tconfint_diff(alpha=0.05, alternative='two-sided', usevar='pooled')}

confidence interval for the difference in means

**Parameters**

- **alpha** : float
  - significance level for the confidence interval, coverage is $1 - \alpha$

- **alternative** : string
  - This specifies the alternative hypothesis for the test that corresponds to the confidence interval. The alternative hypothesis, $H_1$, has to be one of the following:
    - 'two-sided': $H_1$: difference in means not equal to value (default)
    - 'larger': $H_1$: difference in means larger than value
    - 'smaller': $H_1$: difference in means smaller than value

- **usevar** : string, 'pooled' or 'unequal'
  - If 'pooled', then the standard deviation of the samples is assumed to be the same. If 'unequal', then Welsh ttest with Satterthwait degrees of freedom is used

**Returns**

- **lower, upper** : floats
  - lower and upper limits of the confidence interval

**Notes**

The result is independent of the user specified ddof.

\texttt{statsmodels.stats.weightstats.CompareMeans.ttest_ind}

\texttt{CompareMeans.ttest_ind(alternative='two-sided', usevar='pooled', value=0)}

test for the null hypothesis of identical means

this should also be the same as onewaygls, except for ddof differences

**Parameters**

- **x1, x2** : array_like, 1-D or 2-D
  - two independent samples, see notes for 2-D case

- **alternative** : string
  - The alternative hypothesis, $H_1$, has to be one of the following:
    - 'two-sided': $H_1$: difference in means not equal to value (default)
    - 'larger': $H_1$: difference in means larger than value
    - 'smaller': $H_1$: difference in means smaller than value
usevar : string, ‘pooled’ or ‘unequal’
If pooled, then the standard deviation of the samples is assumed to be the same. If unequal, then Welsh ttest with Satterthwait degrees of freedom is used
value : float
difference between the means under the Null hypothesis.

Returns
tstat : float
test statistic
pvalue : float
pvalue of the t-test
df : int or float
degrees of freedom used in the t-test

Notes
The result is independent of the user specified ddof.

statsmodels.stats.weightstats.CompareMeans.ttost_ind

CompareMeans.ttost_ind(low, upp, usevar='pooled')
test of equivalence for two independent samples, base on t-test

Parameters
low, upp : float
equivalence interval low < m1 - m2 < upp
usevar : string, ‘pooled’ or ‘unequal’
If pooled, then the standard deviation of the samples is assumed to be the same. If unequal, then Welsh ttest with Satterthwait degrees of freedom is used

Returns
pvalue : float
pvalue of the non-equivalence test
t1, pv1 : tuple of floats
test statistic and pvalue for lower threshold test
t2, pv2 : tuple of floats
test statistic and pvalue for upper threshold test

statsmodels.stats.weightstats.CompareMeans.zconfint_diff

CompareMeans.zconfint_diff(alpha=0.05, alternative='two-sided', usevar='pooled')
confidence interval for the difference in means

Parameters
alpha : float
significance level for the confidence interval, coverage is 1-alpha
alternative : string
This specifies the alternative hypothesis for the test that corresponds to the confidence interval. The alternative hypothesis, H1, has to be one of the following:

- 'two-sided': H1: difference in means not equal to value (default)
- 'larger': H1: difference in means larger than value
- 'smaller': H1: difference in means smaller than value

**usevar**: string, 'pooled' or 'unequal'

If 'pooled', then the standard deviation of the samples is assumed to be the same. If 'unequal', then Welsh ttest with Satterthwait degrees of freedom is used.

**Returns** lower, upper: floats

lower and upper limits of the confidence interval

**Notes**

The result is independent of the user specified ddof.

```python
statsmodels.stats.weightstats.CompareMeans.ztest_ind
```

CompareMeans.ztest_ind(alternative='two-sided', usevar='pooled', value=0)

test for the null hypothesis of identical means

**Parameters** x1, x2: array_like, 1-D or 2-D

two independent samples, see notes for 2-D case

**alternative**: string

The alternative hypothesis, H1, has to be one of the following 'two-sided': H1: difference in means not equal to value (default) 'larger': H1: difference in means larger than value 'smaller': H1: difference in means smaller than value

**usevar**: string, 'pooled' or 'unequal'

If 'pooled', then the standard deviation of the samples is assumed to be the same. If 'unequal', then Welsh ttest with Satterthwait degrees of freedom is used

**value**: float

difference between the means under the Null hypothesis.

**Returns** tstat: float

test statistic

pvalue: float

pvalue of the t-test

df: int or float

degrees of freedom used in the t-test

```python
statsmodels.stats.weightstats.CompareMeans.ztost_ind
```

CompareMeans.ztost_ind(low, upp, usevar='pooled')

test of equivalence for two independent samples, based on z-test
Parameters `low, upp`: float

equivalence interval \( m1 - m2 < \text{upp} \)

`usevar`: string, ‘pooled’ or ‘unequal’

If pooled, then the standard deviation of the samples is assumed to be the same. If unequal, then Welsh test with Satterthwait degrees of freedom is used

Returns `pvalue`: float

pvalue of the non-equivalence test

`t1, pv1`: tuple of floats

test statistic and pvalue for lower threshold test

`t2, pv2`: tuple of floats

test statistic and pvalue for upper threshold test

`statsmodels.stats.weightstats.ttest_ind`

`statsmodels.stats.weightstats.ttest_ind(x1, x2, alternative='two-sided', usevar='pooled', weights=(None, None), value=0)`

test independent sample

convenience function that uses the classes and throws away the intermediate results, compared to scipy stats: drops axis option, adds alternative, usevar, and weights option

Parameters `x1, x2`: array_like, 1-D or 2-D

two independent samples, see notes for 2-D case

`alternative`: string

The alternative hypothesis, H1, has to be one of the following

‘two-sided’: H1: difference in means not equal to value (default) ‘larger’ : H1: difference in means larger than value ‘smaller’ : H1: difference in means smaller than value

`usevar`: string, ‘pooled’ or ‘unequal’

If pooled, then the standard deviation of the samples is assumed to be the same. If unequal, then Welsh test with Satterthwait degrees of freedom is used

`weights`: tuple of None or ndarrays

Case weights for the two samples. For details on weights see `DescrStatsW`

`value`: float

difference between the means under the Null hypothesis.

Returns `tstat`: float

test statistic

`pvalue`: float

pvalue of the t-test

`df`: int or float

degrees of freedom used in the t-test
test of (non-)equivalence for two independent samples

TOST: two one-sided t tests

null hypothesis: m1 - m2 < low or m1 - m2 > upp alternative hypothesis: low < m1 - m2 < upp

where m1, m2 are the means, expected values of the two samples.

If the pvalue is smaller than a threshold, say 0.05, then we reject the hypothesis that the difference between the two samples is larger than the the thresholds given by low and upp.

**Parameters**

- **x1, x2**: array_like, 1-D or 2-D
  two independent samples, see notes for 2-D case
- **low, upp**: float
  equivalence interval low < m1 - m2 < upp
- **usevar**: string, ‘pooled’ or ‘unequal’
  If pooled, then the standard deviation of the samples is assumed to be the same. If unequal, then Welsh ttest with Satterthwait degrees of freedom is used
- **weights**: tuple of None or ndarrays
  Case weights for the two samples. For details on weights see `DescrStatsW`
- **transform**: None or function
  If None (default), then the data is not transformed. Given a function, sample data and thresholds are transformed. If transform is log, then the equivalence interval is in ratio: low < m1 / m2 < upp

**Returns**

- **pvalue**: float
  pvalue of the non-equivalence test
- **t1, pv1**: tuple of floats
  test statistic and pvalue for lower threshold test
- **t2, pv2**: tuple of floats
  test statistic and pvalue for upper threshold test

**Notes**

The test rejects if the 2*alpha confidence interval for the difference is contained in the (low, upp) interval.

This test works also for multi-endpoint comparisons: If d1 and d2 have the same number of columns, then each column of the data in d1 is compared with the corresponding column in d2. This is the same as comparing each of the corresponding columns separately. Currently no multi-comparison correction is used. The raw p-values reported here can be correction with the functions in `multitest`. 
statsmodels.stats.weightstats.ttost_paired

statsmodels.stats.weightstats.ttost_paired(x1, x2, low, upp, transform=None, weights=None)

test of (non-)equivalence for two dependent, paired sample
TOST: two one-sided t tests
null hypothesis: md < low or md > upp alternative hypothesis: low < md < upp
where md is the mean, expected value of the difference x1 - x2
If the pvalue is smaller than a threshold, say 0.05, then we reject the hypothesis that the difference between the two samples is larger than the the thresholds given by low and upp.

Parameters  
x1, x2 : array_like
    two dependent samples
low, upp : float
    equivalence interval low < mean of difference < upp
weights : None or ndarray
    case weights for the two samples. For details on weights see DescrStatsW
transform : None or function
    If None (default), then the data is not transformed. Given a function sample data and thresholds are transformed. If transform is log the the equivalence interval is in ratio: low < x1 / x2 < upp

Returns  
pvalue : float
    pvalue of the non-equivalence test
t1, pv1, df1 : tuple
    test statistic, pvalue and degrees of freedom for lower threshold test
t2, pv2, df2 : tuple
    test statistic, pvalue and degrees of freedom for upper threshold test

statsmodels.stats.weightstats.ztest

statsmodels.stats.weightstats.ztest(x1, x2=None, value=0, alternative='two-sided', usevar='pooled', ddof=1.0)

test for mean based on normal distribution, one or two samples
In the case of two samples, the samples are assumed to be independent.

Parameters  
x1, x2 : array_like, 1-D or 2-D
    two independent samples
value : float
    In the one sample case, value is the mean of x1 under the Null hypothesis. In the two sample case, value is the difference between mean of x1 and mean of x2 under the Null hypothesis. The test statistic is x1_mean - x2_mean - value.
alternative : string
    The alternative hypothesis, H1, has to be one of the following
'two-sided': H1: difference in means not equal to value (default) ‘larger’: H1: difference in means larger than value ‘smaller’: H1: difference in means smaller than value

**usevar** : string, ‘pooled’

Currently, only ‘pooled’ is implemented. If **pooled**, then the standard deviation of the samples is assumed to be the same. See `CompareMeans.ztest_ind` for different options.

**ddof** : int

Degrees of freedom use in the calculation of the variance of the mean estimate. In the case of comparing means this is one, however it can be adjusted for testing other statistics (proportion, correlation)

**Returns**

**tstat** : float

test statistic

**pvalue** : float

pvalue of the t-test

**Notes**

usevar not implemented, is always pooled in two sample case use `CompareMeans` instead.

### statsmodels.stats.weightstats.ztost

#### statsmodels.stats.weightstats.ztost(x1, low, upp, x2=None, usevar='pooled', ddof=1.0)

Equivalence test based on normal distribution

**Parameters**

**x1** : array_like

one sample or first sample for 2 independent samples

**low, upp** : float

equivalence interval low < m1 - m2 < upp

**x1** : array_like or None

second sample for 2 independent samples test. If None, then a one-sample test is performed.

**usevar** : string, ‘pooled’

If **pooled**, then the standard deviation of the samples is assumed to be the same. Only **pooled** is currently implemented.

**Returns**

**pvalue** : float

pvalue of the non-equivalence test

**t1, pv1** : tuple of floats

test statistic and pvalue for lower threshold test

**t2, pv2** : tuple of floats

test statistic and pvalue for upper threshold test
Notes

cHECKED ONLY FOR 1 SAMPLE CASE

statsmodels.stats.weightstats.zconfint

**statsmodels.stats.weightstats.zconfint**(*x1*, *x2=None*, *value=0*, *alpha=0.05*, *alternative='two-sided'*, *usevar='pooled'*, *ddof=1.0*)

confidence interval based on normal distribution z-test

**Parameters**

\[
x1, x2 : \text{array-like, 1-D or 2-D}
\]

two independent samples, see notes for 2-D case

\[
\text{value} : \text{float}
\]

In the one sample case, value is the mean of x1 under the Null hypothesis. In the two sample case, value is the difference between mean of x1 and mean of x2 under the Null hypothesis. The test statistic is \(x1\_\text{mean} - x2\_\text{mean} - \text{value}\).

\[
\text{usevar} : \text{string, 'pooled'}
\]

Currently, only 'pooled' is implemented. If pooled, then the standard deviation of the samples is assumed to be the same. see CompareMeans.ztest_ind for different options.

\[
\text{ddof} : \text{int}
\]

Degrees of freedom use in the calculation of the variance of the mean estimate. In the case of comparing means this is one, however it can be adjusted for testing other statistics (proportion, correlation)

**See also:**

ztest, CompareMeans

Notes

cHECKED ONLY FOR 1 SAMPLE CASE

usevar not implemented, is always pooled in two sample case

value shifts the confidence interval so it is centered at \(x1\_\text{mean} - x2\_\text{mean} - \text{value}\)

weightstats also contains tests and confidence intervals based on summary data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_tconfint_generic</td>
<td>generic t-confint to save typing</td>
</tr>
<tr>
<td>_tstat_generic</td>
<td>generic t-test to save typing</td>
</tr>
<tr>
<td>_zconfint_generic</td>
<td>generic normal-confint to save typing</td>
</tr>
<tr>
<td>_zstat_generic</td>
<td>generic (normal) z-test to save typing</td>
</tr>
<tr>
<td>_zstat_generic2</td>
<td>generic (normal) z-test to save typing</td>
</tr>
</tbody>
</table>

statsmodels.stats.weightstats._tconfint_generic

**statsmodels.stats.weightstats._tconfint_generic**(*mean, std_mean, dof, ...)**

generic t-confint to save typing
The `power` module currently implements power and sample size calculations for the t-tests, normal based test, F-tests and Chisquare goodness of fit test. The implementation is class based, but the module also provides three shortcut functions, `tt_solve_power`, `tt_ind_solve_power` and `zt_ind_solve_power` to solve for any one of the parameters of the power equations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTestIndPower(<strong>kwds</strong>)</td>
<td>Statistical Power calculations for t-test for two independent sample</td>
</tr>
<tr>
<td>TTtestPower(<strong>kwds</strong>)</td>
<td>Statistical Power calculations for one sample or paired sample t-test</td>
</tr>
<tr>
<td>GofChisquarePower(<strong>kwds</strong>)</td>
<td>Statistical Power calculations for one sample chisquare test</td>
</tr>
<tr>
<td>NormalIndPower([ddof])</td>
<td>Statistical Power calculations for z-test for two independent samples.</td>
</tr>
<tr>
<td>FTestAnovaPower(<strong>kwds</strong>)</td>
<td>Statistical Power calculations F-test for one factor balanced ANOVA</td>
</tr>
<tr>
<td>FTestPower(<strong>kwds</strong>)</td>
<td>Statistical Power calculations for generic F-test</td>
</tr>
<tr>
<td>tt_solve_power</td>
<td>solve for any one parameter of the power of a one sample t-test</td>
</tr>
<tr>
<td>tt_ind_solve_power</td>
<td>solve for any one parameter of the power of a two sample t-test</td>
</tr>
<tr>
<td>zt_ind_solve_power</td>
<td>solve for any one parameter of the power of a two sample z-test</td>
</tr>
</tbody>
</table>

```python
statsmodels.stats.power.TTestIndPower

class statsmodels.stats.power.TTestIndPower(**kwds)
    Statistical Power calculations for t-test for two independent sample
    currently only uses pooled variance
```
**Methods**

- **plot_power**([dep_var, nobs, effect_size, ...])  
  plot power with number of observations or effect size on x-axis

- **power**(effect_size, nobs1, alpha[, ratio, ...])  
  Calculate the power of a t-test for two independent samples

- **solve_power**([effect_size, nobs1, alpha, ...])  
  solve for any one parameter of the power of a two sample t-test

---

**statsmodels.stats.power.TTestIndPower.plot_power**

TTestIndPower.plot_power *(dep_var=’nobs’, nobs=None, effect_size=None, alpha=0.05, ax=None, title=None, plt_kwds=None, **kwds)*  
plot power with number of observations or effect size on x-axis

**Parameters**

- **dep_var** : string in [’nobs’, ’effect_size’, ’alpha’]
  This specifies which variable is used for the horizontal axis. If dep_var=’nobs’ (default), then one curve is created for each value of effect_size. If dep_var=’effect_size’ or alpha, then one curve is created for each value of nobs.

  - **nobs** : scalar or array_like
    specifies the values of the number of observations in the plot

  - **effect_size** : scalar or array_like
    specifies the values of the effect_size in the plot

  - **alpha** : float or array_like
    The significance level (type I error) used in the power calculation. Can only be more than a scalar, if dep_var=’alpha’

  - **ax** : None or axis instance
    If ax is None, than a matplotlib figure is created. If ax is a matplotlib axis instance, then it is reused, and the plot elements are created with it.

  - **title** : string
    title for the axis. Use an empty string, '', to avoid a title.

  - **plt_kwds** : None or dict
    not used yet

  - **kwds** : optional keywords for power function
    These remaining keyword arguments are used as arguments to the power function. Many power function support `alternative` as a keyword argument, two-sample test
    support `ratio`.

**Returns**

- **fig** : matplotlib figure instance

**Notes**

This works only for classes where the power method has effect_size, nobs and alpha as the first three arguments. If the second argument is nobs1, then the number of observations in the plot are those for the first sample. TODO: fix this for FTestPower and GofChisquarePower

TODO: maybe add line variable, if we want more than nobs and effectsize
\texttt{statsmodels.stats.power.TTestIndPower.power}

\texttt{TTestIndPower.power (effect\_size, nobs1, alpha, ratio=1, df=None, alternative='two-sided')}  
Calculate the power of a t-test for two independent samples

**Parameters**  
effect\_size : float  
standardized effect size, difference between the two means divided by the standard deviation. effect\_size has to be positive.

nobs1 : int or float  
number of observations of sample 1. The number of observations of sample two is ratio times the size of sample 1, i.e. nobs2 = nobs1 * ratio

alpha : float in interval (0,1)  
significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

ratio : float  
ratio of the number of observations in sample 2 relative to sample 1. see description of nobs1 The default for ratio is 1; to solve for ratio given the other arguments, it has to be explicitly set to None.

df : int or float  
degrees of freedom. By default this is None, and the df from the ttest with pooled variance is used, df = (nobs1 - 1 + nobs2 - 1)

alternative : string, ‘two-sided’ (default), ‘larger’, ‘smaller’  
extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. The one-sided test can be either ‘larger’, ‘smaller’.

**Returns**  
power : float  
Power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

\texttt{statsmodels.stats.power.TTestIndPower.solve\_power}

\texttt{TTestIndPower.solve\_power (effect\_size=None, nobs1=None, alpha=None, power=None, ratio=1.0, alternative='two-sided')}  
solve for any one parameter of the power of a two sample t-test

**for t-test the keywords are:** effect\_size, nobs1, alpha, power, ratio

exactly one needs to be None, all others need numeric values

**Parameters**  
effect\_size : float  
standardized effect size, difference between the two means divided by the standard deviation. effect\_size has to be positive.

nobs1 : int or float  
number of observations of sample 1. The number of observations of sample two is ratio times the size of sample 1, i.e. nobs2 = nobs1 * ratio

alpha : float in interval (0,1)
significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

**power**: float in interval (0,1)

power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

**ratio**: float

ratio of the number of observations in sample 2 relative to sample 1. see description of nobs! The default for ratio is 1; to solve for ratio given the other arguments it has to be explicitly set to None.

**alternative**: string, ‘two-sided’ (default), ‘larger’, ‘smaller’

extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. The one-sided test can be either ‘larger’, ‘smaller’.

**Returns**

**value**: float

The value of the parameter that was set to None in the call. The value solves the power equation given the remaining parameters.

**Notes**

The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses brentq with a prior search for bounds. If this fails to find a root, fsolve is used. If fsolve also fails, then, for alpha, power and effect_size, brentq with fixed bounds is used. However, there can still be cases where this fails.

**statsmodels.stats.power.TTestPower**

**class** statsmodels.stats.power.TTestPower(**kwds**)  
Statistical Power calculations for one sample or paired sample t-test

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_power</td>
<td>plot power with number of observations or effect size on x-axis</td>
</tr>
<tr>
<td>power</td>
<td>Calculate the power of a t-test for one sample or paired samples.</td>
</tr>
<tr>
<td>solve_power</td>
<td>solve for any one parameter of the power of a one sample t-test</td>
</tr>
</tbody>
</table>

**statsmodels.stats.power.TTestPower.plot_power**

TTestPower.plot_power(**kwds**)  
plot power with number of observations or effect size on x-axis

**Parameters**

**dep_var**: string in [‘nobs’, ‘effect_size’, ‘alpha’]

This specifies which variable is used for the horizontal axis. If dep_var=’nobs’ (default), then one curve is created for each value of effect_size. If dep_var=’effect_size’ or alpha, then one curve is created for each value of nobs.

**nobs**: scalar or array_like
specifies the values of the number of observations in the plot

effect_size : scalar or array_like
    specifies the values of the effect_size in the plot
alpha : float or array_like
    The significance level (type I error) used in the power calculation. Can only be more
    than a scalar, if dep_var='alpha'
ax : None or axis instance
    If ax is None, than a matplotlib figure is created. If ax is a matplotlib axis instance, then
    it is reused, and the plot elements are created with it.
title : string
    title for the axis. Use an empty string, '', to avoid a title.
plt_kwds : None or dict
    not used yet
kwds : optional keywords for power function
    These remaining keyword arguments are used as arguments to the power function.
    Many power function support alternative as a keyword argument, two-sample test
    support ratio.

Returns fig : matplotlib figure instance

Notes

This works only for classes where the power method has effect_size, nobs and alpha as the first
three arguments. If the second argument is nobs1, then the number of observations in the plot are those
for the first sample. TODO: fix this for FTestPower and GofChisquarePower

TODO: maybe add line variable, if we want more than nobs and effectsize

```
statsmodels.stats.power.TTestPower.power
test
```
The `TTestPower.solve_power` function is used to solve for any one parameter of the power of a one sample t-test. For the one sample t-test, the keywords are: `effect_size`, `nobs`, `alpha`, `power`. Exactly one needs to be `None`, all others need numeric values.

This test can also be used for a paired t-test, where `effect_size` is defined in terms of the mean difference, and `nobs` is the number of pairs.

**Parameters**

- `effect_size`: float
  - Standardized effect size, mean divided by the standard deviation. `effect_size` has to be positive.

- `nobs`: int or float
  - Sample size, number of observations.

- `alpha`: float in interval (0,1)
  - Significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

- `power`: float in interval (0,1)
  - Power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

- `alternative`: string, ‘two-sided’ (default) or ‘one-sided’
  - Extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. ‘one-sided’ assumes we are in the relevant tail.

**Returns**

- `value`: float
  - The value of the parameter that was set to `None` in the call. The value solves the power equation given the remaining parameters.

*attaches*:

- `cache_fit_res`: list
  - Cache of the result of the root finding procedure for the latest call to `solve_power`, mainly for debugging purposes. The first element is the success indicator, one if successful. The remaining elements contain the return information of the up to three solvers that have been tried.
Notes

The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses 'brentq' with a prior search for bounds. If this fails to find a root, 'fsolve' is used. If 'fsolve' also fails, then, for 'alpha', 'power' and 'effect_size', 'brentq' with fixed bounds is used. However, there can still be cases where this fails.

statsmodels.stats.power.GofChisquarePower

class statsmodels.stats.power.GofChisquarePower(**kwds)

Statistical Power calculations for one sample chisquare test

Methods

plot_power([dep_var, nobs, effect_size, ...]) plot power with number of observations or effect size on x-axis

gofchisquarepower.power(effect_size, nobs, alpha, n_bins[, ddof]) Calculate the power of a chisquare test for one sample

gofchisquarepower.solve_power([effect_size, nobs, alpha, ...]) solve for any one parameter of the power of a one sample chisquare-test

statsmodels.stats.power.GofChisquarePower.plot_power

GofChisquarePower.plot_power (dep_var='nobs', nobs=None, effect_size=None, alpha=0.05, ax=None, title=None, plt_kwds=None, **kwds)

plot power with number of observations or effect size on x-axis

Parameters

dep_var : string in ['nobs', 'effect_size', 'alpha']

This specifies which variable is used for the horizontal axis. If dep_var='nobs' (default), then one curve is created for each value of effect_size. If dep_var='effect_size' or alpha, then one curve is created for each value of nobs.

nobs : scalar or array_like

specifies the values of the number of observations in the plot

effect_size : scalar or array_like

specifies the values of the effect_size in the plot

alpha : float or array_like

The significance level (type I error) used in the power calculation. Can only be more than a scalar, if dep_var='alpha'

ax : None or axis instance

If ax is None, than a matplotlib figure is created. If ax is a matplotlib axis instance, then it is reused, and the plot elements are created with it.

title : string

title for the axis. Use an empty string, '', to avoid a title.

plt_kwds : None or dict

not used yet

kwds : optional keywords for power function
These remaining keyword arguments are used as arguments to the power function. Many power function support alternative as a keyword argument, two-sample test support ratio.

Returns fig : matplotlib figure instance

Notes

This works only for classes where the power method has effect_size, nobs and alpha as the first three arguments. If the second argument is nobs1, then the number of observations in the plot are those for the first sample. TODO: fix this for FTestPower and GofChisquarePower

TODO: maybe add line variable, if we want more than nobs and effectsize

statsmodels.stats.power.GofChisquarePower.power

GofChisquarePower.power (effect_size, nobs, alpha, n_bins, ddof=0)
Calculate the power of a chisquare test for one sample

Only two-sided alternative is implemented

Parameters effect_size : float
standardized effect size, according to Cohen's definition. see
statsmodels.stats.gof.chisquare_effectsize

nobs : int or float
sample size, number of observations.

alpha : float in interval (0,1)
significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

n_bins : int
number of bins or cells in the distribution.

Returns power : float
Power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

statsmodels.stats.power.GofChisquarePower.solve_power

GofChisquarePower.solve_power (effect_size=None, nobs=None, alpha=None, power=None, n_bins=2)
solve for any one parameter of the power of a one sample chisquare-test

for the one sample chisquare-test the keywords are: effect_size, nobs, alpha, power

Exactly one needs to be None, all others need numeric values.
n_bins needs to be defined, a default=2 is used.

Parameters effect_size : float
standardized effect size, according to Cohen’s definition. see
statsmodels.stats.gof.chisquare_effectsize

nobs : int or float
    sample size, number of observations.

alpha : float in interval (0,1)
    significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections
    if the Null Hypothesis is true.

power : float in interval (0,1)
    power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the
    probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis
    is true.

n_bins : int
    number of bins or cells in the distribution

Returns value : float
    The value of the parameter that was set to None in the call. The value solves the power
    equation given the remaining parameters.

Notes

The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses
brentq with a prior search for bounds. If this fails to find a root, fsolve is used. If fsolve also
fails, then, for alpha, power and effect_size, brentq with fixed bounds is used. However, there
can still be cases where this fails.

statsmodels.stats.power.NormalIndPower

class statsmodels.stats.power.NormalIndPower (ddof=0, **kwds)
    Statistical Power calculations for z-test for two independent samples.
    currently only uses pooled variance

Methods

plot_power([dep_var, nobs, effect_size, ...]) plot power with number of observations or effect size on x-axis
power(effect_size, nobs1, alpha[, ratio, ...])    Calculate the power of a t-test for two independent sample
solve_power([effect_size, nobs1, alpha, ...])    solve for any one parameter of the power of a two sample z-test

statsmodels.stats.power.NormalIndPower.plot_power

NormalIndPower.plot_power (dep_var='nobs', nobs=None, effect_size=None, alpha=0.05,
ax=None, title=None, plt_kws=None, **kws)
    plot power with number of observations or effect size on x-axis

Parameters dep_var : string in [‘nobs’, ‘effect_size’, ‘alpha’]
    This specifies which variable is used for the horizontal axis. If dep_var=’nobs’ (default),
then one curve is created for each value of effect_size. If dep_var='effect_size' or alpha, then one curve is created for each value of nobs.

nobs : scalar or array_like
    specifies the values of the number of observations in the plot
effect_size : scalar or array_like
    specifies the values of the effect_size in the plot
alpha : float or array_like
    The significance level (type I error) used in the power calculation. Can only be more than a scalar, if dep_var='alpha'
ax : None or axis instance
    If ax is None, than a matplotlib figure is created. If ax is a matplotlib axis instance, then it is reused, and the plot elements are created with it.
title : string
    title for the axis. Use an empty string, '', to avoid a title.
plt_kwds : None or dict
    not used yet
kwds : optional keywords for power function
    These remaining keyword arguments are used as arguments to the power function. Many power function support alternative as a keyword argument, two-sample test support ratio.

Returns fig : matplotlib figure instance

Notes

This works only for classes where the power method has effect_size, nobs and alpha as the first three arguments. If the second argument is nobs1, then the number of observations in the plot are those for the first sample. TODO: fix this for FTestPower and GofChisquarePower

TODO: maybe add line variable, if we want more than nobs and effectsize

statsmodels.stats.power.NormalIndPower.power

NormalIndPower.power(effect_size, nobs1, alpha, ratio=1, alternative='two-sided')
Calculate the power of a t-test for two independent sample

Parameters effect_size : float
    standardized effect size, difference between the two means divided by the standard deviation. effect size has to be positive.
nobs1 : int or float
    number of observations of sample 1. The number of observations of sample two is ratio times the size of sample 1, i.e. nobs2 = nobs1 * ratio ratio can be set to zero in order to get the power for a one sample test.
alpha : float in interval (0,1)
significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

**ratio** : float

ratio of the number of observations in sample 2 relative to sample 1. see description of `nobs1` The default for ratio is 1; to solve for ratio given the other arguments it has to be explicitly set to None.

**alternative** : string, ‘two-sided’ (default), ‘larger’, ‘smaller’

extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. The one-sided test can be either ‘larger’, ‘smaller’.

**Returns power** : float

Power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

```python
statsmodels.stats.power.NormalIndPower.solve_power
```

`NormalIndPower.solve_power(effect_size=None, nobs1=None, alpha=None, power=None, ratio=1.0, alternative='two-sided')`

solve for any one parameter of the power of a two sample z-test

**for z-test the keywords are:** effect_size, nobs1, alpha, power, ratio

exactly one needs to be `None`, all others need numeric values

**Parameters effect_size** : float

standardized effect size, difference between the two means divided by the standard deviation. If ratio=0, then this is the standardized mean in the one sample test.

`nobs1` : int or float

number of observations of sample 1. The number of observations of sample two is ratio times the size of sample 1, i.e. `nobs2 = nobs1 * ratio` ratio can be set to zero in order to get the power for a one sample test.

`alpha` : float in interval (0,1)

significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

`power` : float in interval (0,1)

power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

`ratio` : float

ratio of the number of observations in sample 2 relative to sample 1. see description of `nobs1` The default for ratio is 1; to solve for ration given the other arguments it has to be explicitly set to None.

`alternative` : string, ‘two-sided’ (default), ‘larger’, ‘smaller’

extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. The one-sided test can be either ‘larger’, ‘smaller’.

**Returns value** : float
The value of the parameter that was set to None in the call. The value solves the power equation given the remaining parameters.

Notes

The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses brentq with a prior search for bounds. If this fails to find a root, fsolve is used. If fsolve also fails, then, for alpha, power and effect_size, brentq with fixed bounds is used. However, there can still be cases where this fails.

statsmodels.stats.power.FTestAnovaPower

class statsmodels.stats.power.FTestAnovaPower(**kwds)

Statistical Power calculations F-test for one factor balanced ANOVA

Methods

plot_power([dep_var, nobs, effect_size, ...])  
plot power with number of observations or effect size on x-axis

power(effect_size, nobs, alpha[, k_groups])  
Calculate the power of a F-test for one factor ANOVA.

solve_power([effect_size, nobs, alpha, ...])  
solve for any one parameter of the power of a F-test

statsmodels.stats.power.FTestAnovaPower.plot_power

FTestAnovaPower.plot_power (dep_var='nobs', nobs=None, effect_size=None, alpha=0.05, ax=None, title=None, plt_kwds=None, **kwds)

plot power with number of observations or effect size on x-axis

Parameters dep_var : string in ['nobs', 'effect_size', 'alpha']

This specifies which variable is used for the horizontal axis. If dep_var='nobs' (default), then one curve is created for each value of effect_size. If dep_var='effect_size' or alpha, then one curve is created for each value of nobs.

nobs : scalar or array_like

specifies the values of the number of observations in the plot

effect_size : scalar or array_like

specifies the values of the effect_size in the plot

alpha : float or array_like

The significance level (type I error) used in the power calculation. Can only be more than a scalar, if dep_var='alpha'

ax : None or axis instance

If ax is None, than a matplotlib figure is created. If ax is a matplotlib axis instance, then it is reused, and the plot elements are created with it.

title : string

title for the axis. Use an empty string, '', to avoid a title.

plt_kwds : None or dict
not used yet

**kwds**: optional keywords for power function

These remaining keyword arguments are used as arguments to the power function. Many power function support `alternative` as a keyword argument, two-sample test support `ratio`.

**Returns**

**fig**: matplotlib figure instance

**Notes**

This works only for classes where the `power` method has `effect_size`, `nobs` and `alpha` as the first three arguments. If the second argument is `nobs1`, then the number of observations in the plot are those for the first sample. TODO: fix this for `FTestPower` and `GofChisquarePower`

TODO: maybe add line variable, if we want more than nobs and effectsize

```python
def power(effect_size, nobs, alpha, k_groups=2):
    Calculate the power of a F-test for one factor ANOVA.

    Parameters effect_size : float
        standardized effect size, mean divided by the standard deviation. effect size has to be positive.

    nobs : int or float
        sample size, number of observations.

    alpha : float in interval (0,1)
        significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

    k_groups : int or float
        number of groups in the ANOVA or k-sample comparison. Default is 2.

    Returns power : float
        Power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.
```

```python
def solve_power(effect_size=None, nobs=None, alpha=None, power=None, k_groups=2):
    solve for any one parameter of the power of a F-test

    for the one sample F-test the keywords are: effect_size, nobs, alpha, power

    Exactly one needs to be None, all others need numeric values.

    Parameters effect_size : float
```
standardized effect size, mean divided by the standard deviation. effect size has to be positive.

**nobs** : int or float

sample size, number of observations.

**alpha** : float in interval (0,1)

significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

**power** : float in interval (0,1)

power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

**Returns**

**value** : float

The value of the parameter that was set to None in the call. The value solves the power equation given the remaining parameters.

**Notes**

The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses *brentq* with a prior search for bounds. If this fails to find a root, *fsolve* is used. If *fsolve* also fails, then, for **alpha**, **power** and **effect_size**, *brentq* with fixed bounds is used. However, there can still be cases where this fails.

**statsmodels.stats.power.FTestPower**

class `statsmodels.stats.power.FTestPower(**kwds)`

Statistical Power calculations for generic F-test

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plot_power</code></td>
<td>plot power with number of observations or effect size on x-axis</td>
</tr>
<tr>
<td><code>power</code></td>
<td>Calculate the power of a F-test.</td>
</tr>
<tr>
<td><code>solve_power</code></td>
<td>solve for any one parameter of the power of a F-test</td>
</tr>
</tbody>
</table>

**statsmodels.stats.power.FTestPower.plot_power**

`FTestPower.plot_power` *(dep_var=’nobs’, nobs=None, effect_size=None, alpha=0.05, ax=None, title=None, plt_kwds=None, **kwds]*)

plot power with number of observations or effect size on x-axis

**Parameters**

**dep_var** : string in [’nobs’, ‘effect_size’, ‘alpha’]

This specifies which variable is used for the horizontal axis. If dep_var=’nobs’ (default), then one curve is created for each value of effect_size. If dep_var=’effect_size’ or alpha, then one curve is created for each value of nobs.

**nobs** : scalar or array_like

specifies the values of the number of observations in the plot
**effect_size**: scalar or array_like

specifies the values of the effect_size in the plot

**alpha**: float or array_like

The significance level (type I error) used in the power calculation. Can only be more than a scalar, if `dep_var='alpha'`

**ax**: None or axis instance

If ax is None, than a matplotlib figure is created. If ax is a matplotlib axis instance, then it is reused, and the plot elements are created with it.

**title**: string

title for the axis. Use an empty string, '', to avoid a title.

**plt_kwds**: None or dict

not used yet

**kwds**: optional keywords for power function

These remaining keyword arguments are used as arguments to the power function. Many power function support `alternative` as a keyword argument, two-sample test support `ratio`.

**Returns** **fig**: matplotlib figure instance

**Notes**

This works only for classes where the `power` method has `effect_size`, `nobs` and `alpha` as the first three arguments. If the second argument is `nobs1`, then the number of observations in the plot are those for the first sample. TODO: fix this for FTestPower and GofChisquarePower

TODO: maybe add line variable, if we want more than nobs and effectsize

```python
statsmodels.stats.power.FTestPower.power
```

FTestPower. `power` *(effect_size, df_num, df_denom, alpha, ncc=1)*

Calculate the power of a F-test.

**Parameters** **effect_size**: float

standardized effect size, mean divided by the standard deviation. effect size has to be positive.

**df_num**: int or float

numerator degrees of freedom.

**df_denom**: int or float

denominator degrees of freedom.

**alpha**: float in interval (0,1)

significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

**ncc**: int

degrees of freedom correction for non-centrality parameter. see Notes
Returns power : float

Power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

Notes

Sample size is given implicitly by df_num

Set ncc=0 to match t-test, or f-test in LikelihoodModelResults. ncc=1 matches the non-centrality parameter in R::pwr::pwr.f2.test

Ftest_power with ncc=0 should also be correct for f_test in regression models, with df_num and d_denom as defined there. (not verified yet)

statsmodels.stats.power.FTestPower.solve_power

FTestPower.solve_power (effect_size=None, df_num=None, df_denom=None, nobs=None, alpha=None, power=None, ncc=1)
solve for any one parameter of the power of a F-test

For the one sample F-test the keywords are: effect_size, df_num, df_denom, alpha, power

Exactly one needs to be None, all others need numeric values.

Parameters effect_size : float

Standardized effect size, mean divided by the standard deviation. Effect size has to be positive.

nobs : int or float

Sample size, number of observations.

alpha : float in interval (0, 1)

Significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

power : float in interval (0, 1)

Power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

alternative : string, ‘two-sided’ (default) or ‘one-sided’

Extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. ‘one-sided’ assumes we are in the relevant tail.

Returns value : float

The value of the parameter that was set to None in the call. The value solves the power equation given the remaining parameters.
Notes

The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses brentq with a prior search for bounds. If this fails to find a root, fsolve is used. If fsolve also fails, then, for alpha, power and effect_size, brentq with fixed bounds is used. However, there can still be cases where this fails.

statsmodels.stats.power.tt_solve_power

statsmodels.stats.power.tt_solve_power = <bound method TTestPower.solve_power of <statsmodels.stats.power.TTestPower object at 0x081164B0>>

solve for any one parameter of the power of a one sample t-test

for the one sample t-test the keywords are: effect_size, nobs, alpha, power

Exactly one needs to be None, all others need numeric values.

This test can also be used for a paired t-test, where effect size is defined in terms of the mean difference, and nobs is the number of pairs.

Parameters
effect_size : float

standardized effect size, mean divided by the standard deviation. effect size has to be positive.

nobs : int or float

sample size, number of observations.

alpha : float in interval (0,1)

significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

power : float in interval (0,1)

power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

alternative : string, ‘two-sided’ (default) or ‘one-sided’

extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. ‘one-sided’ assumes we are in the relevant tail.

Returns

value : float

The value of the parameter that was set to None in the call. The value solves the power equation given the remaining parameters.

*attaches*

*cache_fit_res* : list

Cache of the result of the root finding procedure for the latest call to solve_power, mainly for debugging purposes. The first element is the success indicator, one if successful. The remaining elements contain the return information of the up to three solvers that have been tried.
The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses `brentq` with a prior search for bounds. If this fails to find a root, `fsolve` is used. If `fsolve` also fails, then, for `alpha`, `power` and `effect_size`, `brentq` with fixed bounds is used. However, there can still be cases where this fails.

**statsmodels.stats.power.tt_ind_solve_power**

```python
statsmodels.stats.power.tt_ind_solve_power = <bound method TTestIndPower.solve_power of <statsmodels.stats.power.TTestIndPower object at 0x08116510>>
```

solve for any one parameter of the power of a two sample t-test

**for t-test the keywords are:** `effect_size`, `nobs1`, `alpha`, `power`, `ratio`

exactly one needs to be `None`, all others need numeric values

**Parameters**

- `effect_size`: float
  standardized effect size, difference between the two means divided by the standard deviation. `effect_size` has to be positive.

- `nobs1`: int or float
  number of observations of sample 1. The number of observations of sample two is ratio times the size of sample 1, i.e. `nobs2 = nobs1 * ratio`

- `alpha`: float in interval (0,1)
  significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

- `power`: float in interval (0,1)
  power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

- `ratio`: float
  ratio of the number of observations in sample 2 relative to sample 1. see description of `nobs1` The default for ratio is 1; to solve for ratio given the other arguments it has to be explicitly set to None.

- `alternative`: string, ‘two-sided’ (default), ‘larger’, ‘smaller’
  extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. The one-sided test can be either ‘larger’, ‘smaller’.

**Returns**

- `value`: float
  The value of the parameter that was set to None in the call. The value solves the power equation given the remaining parameters.

**Notes**

The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses `brentq` with a prior search for bounds. If this fails to find a root, `fsolve` is used. If `fsolve` also fails, then, for `alpha`, `power` and `effect_size`, `brentq` with fixed bounds is used. However, there can still be cases where this fails.

3.8. Statistics **stats** 739
for z-test the keywords are: effect_size, nobs1, alpha, power, ratio

exactly one needs to be None, all others need numeric values

Parameters
effect_size : float
standardized effect size, difference between the two means divided by the standard deviation. If ratio=0, then this is the standardized mean in the one sample test.

nobs1 : int or float
number of observations of sample 1. The number of observations of sample two is ratio times the size of sample 1, i.e. nobs2 = nobs1 * ratio ratio can be set to zero in order to get the power for a one sample test.

alpha : float in interval (0,1)
significance level, e.g. 0.05, is the probability of a type I error, that is wrong rejections if the Null Hypothesis is true.

power : float in interval (0,1)
power of the test, e.g. 0.8, is one minus the probability of a type II error. Power is the probability that the test correctly rejects the Null Hypothesis if the Alternative Hypothesis is true.

ratio : float
ratio of the number of observations in sample 2 relative to sample 1. see description of nobs1 The default for ratio is 1; to solve for ratio given the other arguments it has to be explicitly set to None.

alternative : string, ‘two-sided’ (default), ‘larger’, ‘smaller’
extra argument to choose whether the power is calculated for a two-sided (default) or one sided test. The one-sided test can be either ‘larger’, ‘smaller’.

Returns
value : float
The value of the parameter that was set to None in the call. The value solves the power equation given the remaining parameters.

Notes
The function uses scipy.optimize for finding the value that satisfies the power equation. It first uses brentq with a prior search for bounds. If this fails to find a root, fsolve is used. If fsolve also fails, then, for alpha, power and effect_size, brentq with fixed bounds is used. However, there can still be cases where this fails.

3.8.9 Proportion

Also available are hypothesis test, confidence intervals and effect size for proportions that can be used with NormalIndPower.

proportion_confint(count, nobs[, alpha, method]) confidence interval for a binomial proportion

Continued on next page
### Table 3.212 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>proportion_effectsize(prop1, prop2[, method])</code></td>
<td>effect size for a test comparing two proportions</td>
</tr>
<tr>
<td><code>binom_test(count, nobs[, prop, alternative])</code></td>
<td>Perform a test that the probability of success is ( p ).</td>
</tr>
<tr>
<td><code>binom_test_reject_interval(value, nobs[, ...])</code></td>
<td>rejection region for binomial test for one sample proportion</td>
</tr>
<tr>
<td><code>binom_tost(count, nobs, low, upp)</code></td>
<td>exact TOST test for one proportion using binomial distribution</td>
</tr>
<tr>
<td><code>binom_tost_reject_interval(low, upp, nobs[, ...])</code></td>
<td>rejection region for binomial TOST</td>
</tr>
<tr>
<td><code>proportions_ztest(count, nobs[, value])</code></td>
<td>test for proportions based on normal (z) test</td>
</tr>
<tr>
<td><code>proportions_tost(count, nobs, low, upp[, ...])</code></td>
<td>Equivalence test based on normal distribution</td>
</tr>
<tr>
<td><code>proportions_chisquare(count, nobs[, value])</code></td>
<td>test for proportions based on chi-square test</td>
</tr>
<tr>
<td><code>proportions_chisquare_allpairs(count, nobs)</code></td>
<td>chi-square test of proportions for all pairs of ( k ) samples compared to control</td>
</tr>
<tr>
<td><code>proportions_chisquare_pairscontrol(count, nobs)</code></td>
<td>chi-square test of proportions for pairs of ( k ) samples compared to control</td>
</tr>
<tr>
<td><code>proportion_effectsize(prop1, prop2[, method])</code></td>
<td>effect size for a test comparing two proportions</td>
</tr>
<tr>
<td><code>power_binom_tost(low, upp, nobs[, p_alt, alpha])</code></td>
<td>Power of proportions equivalence test based on normal distribution</td>
</tr>
<tr>
<td><code>power_ztost_prop(low, upp, nobs, p_alt[, ...])</code></td>
<td>find sample size to get desired confidence interval length</td>
</tr>
<tr>
<td><code>samplesize_confint_proportion(proportion, ...)</code></td>
<td></td>
</tr>
</tbody>
</table>

---

**statsmodels.stats.proportion.proportion_confint**

The function `proportion_confint` in the `statsmodels` library computes the confidence interval for a binomial proportion.

**Parameters**

- `count` : int or array
  - number of successes
- `nobs` : int
  - total number of trials
- `alpha` : float in (0, 1)
  - significance level, default 0.05
- `method` : string in ['normal']
  - method to use for confidence interval, currently available methods :
    - `normal` : asymptotic normal approximation
    - `agresti_coull` : Agresti-Coull interval
    - `beta` : Clopper-Pearson interval based on Beta distribution
    - `wilson` : Wilson Score interval
    - `jeffrey` : Jeffrey’s Bayesian Interval
    - `binom_test` : experimental, inversion of binom_test

**Returns**

- `ci_low`, `ci_upp` : float
  - lower and upper confidence level with coverage (approximately) \( 1 - \alpha \). Note: Beta has coverage coverage is only \( 1 - \alpha \) on average for some other methods.)

**Notes**

- Beta, the Clopper-Pearson interval has coverage at least \( 1 - \alpha \), but is in general conservative. Most of the other methods have average coverage equal to \( 1 - \alpha \), but will have smaller coverage in some cases.
- Method “binom_test” directly inverts the binomial test in scipy.stats. which has discrete steps.

---

### 3.8. Statistics stats

741
TODO: binom_test intervals raise an exception in small samples if one interval bound is close to zero or one.

References


statsmodels.stats.proportion.proportion_effectsize

statsmodels.stats.proportion.proportion_effectsize(prop1, prop2, method='normal')
effect size for a test comparing two proportions
for use in power function

Parameters  prop1, prop2: float or array_like :

Returns  es : float or ndarray
    effect size for (transformed) prop1 - prop2

Notes

only method='normal' is implemented to match pwr.p2.test see http://www.statmethods.net/stats/power.html

Effect size for normal is defined as

2 * (arcsin(sqrt(prop1)) - arcsin(sqrt(prop2)))

I think other conversions to normality can be used, but I need to check.

Examples

>>> smpr.proportion_effectsize(0.5, 0.4)
0.20135792079033088
>>> smpr.proportion_effectsize([0.3, 0.4, 0.5], 0.4)
array([-0.21015893, 0. , 0.20135792])

statsmodels.stats.proportion.binom_test

statsmodels.stats.proportion.binom_test(count, nobs, prop=0.5, alternative='two-sided')
Perform a test that the probability of success is p.

This is an exact, two-sided test of the null hypothesis that the probability of success in a Bernoulli experiment is p.

Parameters  count : integer or array_like
    the number of successes in nobs trials.

nobs : integer
    the number of trials or observations.
prop : float, optional
The probability of success under the null hypothesis, \(0 \leq prop \leq 1\). The default value is \(prop = 0.5\)

alternative : string in ['two-sided', 'smaller', 'larger']
alternative hypothesis, which can be two-sided or either one of the one-sided tests.

Returns p-value : float
The p-value of the hypothesis test

Notes
This uses scipy.stats.binom_test for the two-sided alternative.

statsmodels.stats.proportion.binom_test_reject_interval
rejection region for binomial test for one sample proportion
The interval includes the end points of the rejection region.

Parameters value : float
proportion under the Null hypothesis
nobs : integer
the number of trials or observations.

Returns x_low, x_upp : float
lower and upper bound of rejection region

statsmodels.stats.proportion.binom_tost
exact TOST test for one proportion using binomial distribution

Parameters count : integer or array_like
the number of successes in nobs trials.
nobs : integer
the number of trials or observations.
low, upp : floats
lower and upper limit of equivalence region

Returns pvalue : float
p-value of equivalence test
pval_low, pval_upp : floats
p-values of lower and upper one-sided tests
statsmodels.stats.proportion.binom_tost_reject_interval

statsmodels.stats.proportion.binom_tost_reject_interval(low, upp, nobs, alpha=0.05)

rejection region for binomial TOST
The interval includes the end points, reject if and only if \( r_{low} \leq x \leq r_{upp} \).
The interval might be empty with \( r_{upp} < r_{low} \).

Parameters

- low, upp : floats
  lower and upper limit of equivalence region
- nobs : integer
  the number of trials or observations.

Returns

- x_low, x_upp : float
  lower and upper bound of rejection region

statsmodels.stats.proportion.proportions_ztest

statsmodels.stats.proportion.proportions_ztest(count, nobs, value=None, alternative='two-sided', prop_var=False)

test for proportions based on normal (z) test

Parameters

- count : integer or array_like
  the number of successes in nobs trials. If this is array_like, then the assumption is that
  this represents the number of successes for each independent sample
- nobs : integer
  the number of trials or observations, with the same length as count.
- value : None or float or array_like
  This is the value of the null hypothesis equal to the proportion in the case of a one
  sample test. In the case of a two-sample test, the null hypothesis is that \( \text{prop}[0] - \text{prop}[1] = \text{value} \), where prop is the proportion in the two samples
- alternative : string in ['two-sided', 'smaller', 'larger']
  The alternative hypothesis can be either two-sided or one of the one-sided tests, smaller
  means that the alternative hypothesis is \( \text{prop} < \text{value} \) and larger means
  \( \text{prop} > \text{value} \), or the corresponding inequality for the two sample test.
- prop_var : False or float in (0, 1)
  If prop_var is false, then the variance of the proportion estimate is calculated based on
  the sample proportion. Alternatively, a proportion can be specified to calculate this var-
  iance. Common use case is to use the proportion under the Null hypothesis to specify the
  variance of the proportion estimate. TODO: change options similar to propotion_ztost

Returns

- zstat : float
  test statistic for the z-test
- p-value : float
  p-value for the z-test
Notes

This uses a simple normal test for proportions. It should be the same as running the mean z-test on the data encoded 1 for event and 0 for no event, so that the sum corresponds to count.

In the one and two sample cases with two-sided alternative, this test produces the same p-value as `proportions_chisquare`, since the chisquare is the distribution of the square of a standard normal distribution. (TODO: verify that this really holds)

TODO: add continuity correction or other improvements for small samples.

**statsmodels.stats.proportion.proportions_ztost**

```
statsmodels.stats.proportion.proportions_ztost(count, nobs, low, upp, prop_var='sample')
```

Equivalence test based on normal distribution

**Parameters**

- `count` : integer or array_like
  - the number of successes in nobs trials. If this is array_like, then the assumption is that this represents the number of successes for each independent sample

- `nobs` : integer
  - the number of trials or observations, with the same length as count.

- `low, upp` : float
  - equivalence interval low < prop1 - prop2 < upp

- `prop_var` : string or float in (0, 1)
  - prop_var determines which proportion is used for the calculation of the standard deviation of the proportion estimate. The available options for string are ‘sample’ (default), ‘null’ and ‘limits’. If prop_var is a float, then it is used directly.

**Returns**

- `pvalue` : float
  - pvalue of the non-equivalence test

- `t1, pv1` : tuple of floats
  - test statistic and pvalue for lower threshold test

- `t2, pv2` : tuple of floats
  - test statistic and pvalue for upper threshold test

**Notes**

checked only for 1 sample case

**statsmodels.stats.proportion.proportions_chisquare**

```
statsmodels.stats.proportion.proportions_chisquare(count, nobs, value=None)
```

test for proportions based on chisquare test

**Parameters**

- `count` : integer or array_like
the number of successes in nobs trials. If this is array_like, then the assumption is that
this represents the number of successes for each independent sample

**nobs** : integer

the number of trials or observations, with the same length as count.

**value** : None or float or array_like

**Returns**

- **chi2stat** : float
test statistic for the chisquare test

- **p-value** : float
p-value for the chisquare test

- **(table, expected)** :
table is a (k, 2) contingency table, expected is the corresponding table of counts that
are expected under independence with given margins

**Notes**

Recent version of scipy.stats have a chisquare test for independence in contingency tables.
This function provides a similar interface to chisquare tests as `prop.test` in R, however without the option
for Yates continuity correction.

count can be the count for the number of events for a single proportion, or the counts for several independent
proportions. If value is given, then all proportions are jointly tested against this value. If value is not given and
count and nobs are not scalar, then the null hypothesis is that all samples have the same proportion.
Notes

Yates continuity correction is not available.

**statsmodels.stats.proportion.proportions_chisquare_pairscontrol**

```python
statsmodels.stats.proportion.proportions_chisquare_pairscontrol(count, nobs, value=None, multi-test_method='hs', alternative='two-sided')
```

chisquare test of proportions for pairs of k samples compared to control

Performs a chisquare test for proportions for pairwise comparisons with a control (Dunnet’s test). The control is assumed to be the first element of `count` and `nobs`. The alternative is two-sided, larger or smaller.

**Parameters**
- `count`: integer or array_like  
  the number of successes in `nobs` trials.
- `nobs`: integer  
  the number of trials or observations.
- `prop`: float, optional  
  The probability of success under the null hypothesis, 0 <= prop <= 1. The default value is `prop = 0.5`.
- `multitest_method`: string  
  This chooses the method for the multiple testing p-value correction, that is used as default in the results. It can be any method that is available in `multipletesting`. The default is Holm-Sidak ‘hs’.
- `alternative`: string in ['two-sided', 'smaller', 'larger']  
  alternative hypothesis, which can be two-sided or either one of the one-sided tests.

**Returns**
- `result`: AllPairsResults instance  
  The returned results instance has several statistics, such as p-values, attached, and additional methods for using a non-default `multitest_method`.

Notes

Yates continuity correction is not available.

`value` and `alternative` options are not yet implemented.

**statsmodels.stats.proportion.proportion_effectsize**

```python
statsmodels.stats.proportion.proportion_effectsize(prop1, prop2, method='normal')
```

effect size for a test comparing two proportions

for use in power function

**Parameters**
- `prop1, prop2`: float or array_like :
Returns \( es \): float or ndarray

\[
effect \text{ size for } (\text{transformed}) \text{ prop1} - \text{prop2}
\]

Notes

Only method='normal' is implemented to match pwr.p2.test see http://www.statmethods.net/stats/power.html

Effect size for normal is defined as

\[
2 \times (\arcsin(\sqrt{\text{prop1}}) - \arcsin(\sqrt{\text{prop2}}))
\]

I think other conversions to normality can be used, but I need to check.

Examples

```python
>>> smpr.proportion_effectsize(0.5, 0.4)
0.20135792079033088
>>> smpr.proportion_effectsize([0.3, 0.4, 0.5], 0.4)
array([-0.21015893, 0., 0.20135792])
```

`statsmodels.stats.proportion.power_binom_tost`

`statsmodels.stats.proportion.power_binom_tost` (low, upp, nobs, p_alt=None, alpha=0.05)

`statsmodels.stats.proportion.power_ztost_prop`

`statsmodels.stats.proportion.power_ztost_prop` (low, upp, nobs, p_alt, alpha=0.05, dist='norm', variance_prop=None, discrete=True, continuity=0, critical_continuity=0)

Power of proportions equivalence test based on normal distribution

Parameters low, upp: floats

lower and upper limit of equivalence region

nobs: int

number of observations

p_alt: float in (0,1)

proportion under the alternative

alpha: float in (0,1)

significance level of the test

dist: string in ['norm', 'binom']

This defines the distribution to evaluate the power of the test. The critical values of the TOST test are always based on the normal approximation, but the distribution for the power can be either the normal (default) or the binomial (exact) distribution.

variance_prop: None or float in (0,1)
If this is None, then the variances for the two one sided tests are based on the proportions equal to the equivalence limits. If variance_prop is given, then it is used to calculate the variance for the TOST statistics. If this is based on an sample, then the estimated proportion can be used.

**discrete**: `bool`

If true, then the critical values of the rejection region are converted to integers. If dist is “binom”, this is automatically assumed. If discrete is false, then the TOST critical values are used as floating point numbers, and the power is calculated based on the rejection region that is not discretized.

**continuity**: `bool or float`

Adjust the rejection region for the normal power probability. This has an effect only if `dist='norm'`

**critval_continuity**: `bool or float`

If this is non-zero, then the critical values of the TOST rejection region are adjusted before converting to integers. This affects both distributions, `dist='norm'` and `dist='binom'`.

Returns **power**: `float`

Statistical power of the equivalence test.

**(k_low, k_upp, z_low, z_upp)**: tuple of floats

Critical limits in intermediate steps temporary return, will be changed

Notes

In small samples the power for the `discrete` version, has a sawtooth pattern as a function of the number of observations. As a consequence, small changes in the number of observations or in the normal approximation can have a large effect on the power.

continuity and critval_continuity are added to match some results of PASS, and are mainly to investigate the sensitivity of the ztost power to small changes in the rejection region. From my interpretation of the equations in the SAS manual, both are zero in SAS.

works vectorized

**verification**: The `dist='binom'` results match PASS, The `dist='norm'` results look reasonable, but no benchmark is available.

References

find sample size to get desired confidence interval length

Parameters

- proportion : float in (0, 1)
  proportion or quantile

- half_length : float in (0, 1)
  desired half length of the confidence interval

- alpha : float in (0, 1)
  significance level, default 0.05, coverage of the two-sided interval is (approximately) 1
  - alpha

- method : string in ['normal']
  method to use for confidence interval, currently only normal approximation

Returns

- n : float
  sample size to get the desired half length of the confidence interval

Notes

this is mainly to store the formula. possible application: number of replications in bootstrap samples

### 3.8.10 Moment Helpers

When there are missing values, then it is possible that a correlation or covariance matrix is not positive semi-definite. The following three functions can be used to find a correlation or covariance matrix that is positive definite and close to the original matrix.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>corr_nearest(corr[, threshold, n_fact])</td>
<td>Find the nearest correlation matrix that is positive semi-definite.</td>
</tr>
<tr>
<td>corr_clipped(corr[, threshold])</td>
<td>Find a near correlation matrix that is positive semi-definite</td>
</tr>
<tr>
<td>cov_nearest(cov[, method, threshold, ...])</td>
<td>Find the nearest covariance matrix that is positive (semi-) definite</td>
</tr>
</tbody>
</table>

statsmodels.stats.correlation_tools.corr_nearest

Find the nearest correlation matrix that is positive semi-definite.

The function iteratively adjust the correlation matrix by clipping the eigenvalues of a difference matrix. The diagonal elements are set to one.

Parameters

- corr : ndarray, (k, k)
  initial correlation matrix

- threshold : float
  clipping threshold for smallest eigenvalue, see Notes
n_fact : int or float

factor to determine the maximum number of iterations. The maximum number of iterations is the integer part of the number of columns in the correlation matrix times n_fact.

Returns corr_new : ndarray, (optional)

corrected correlation matrix

See also:
corr_clipped, cov_nearest

Notes

The smallest eigenvalue of the corrected correlation matrix is approximately equal to the threshold. If the threshold=0, then the smallest eigenvalue of the correlation matrix might be negative, but zero within a numerical error, for example in the range of -1e-16.

Assumes input correlation matrix is symmetric.

Stops after the first step if correlation matrix is already positive semi-definite or positive definite, so that smallest eigenvalue is above threshold. In this case, the returned array is not the original, but is equal to it within numerical precision.

statsmodels.stats.correlation_tools.corr_clipped

statsmodels.stats.correlation_tools.corr_clipped(corr, threshold=1e-15)

Find a near correlation matrix that is positive semi-definite

This function clips the eigenvalues, replacing eigenvalues smaller than the threshold by the threshold. The new matrix is normalized, so that the diagonal elements are one. Compared to corr_nearest, the distance between the original correlation matrix and the positive definite correlation matrix is larger, however, it is much faster since it only computes eigenvalues once.

Parameters corr : ndarray, (k, k)

initial correlation matrix

threshold : float

clipping threshold for smallest eigenvalue, see Notes

Returns corr_new : ndarray, (optional)

corrected correlation matrix

See also:
corr_nearest, cov_nearest

Notes

The smallest eigenvalue of the corrected correlation matrix is approximately equal to the threshold. In examples, the smallest eigenvalue can be by a factor of 10 smaller than the threshold, e.g. threshold 1e-8 can result in smallest eigenvalue in the range between 1e-9 and 1e-8. If the threshold=0, then the smallest eigenvalue of the correlation matrix might be negative, but zero within a numerical error, for example in the range of -1e-16.
Assumes input correlation matrix is symmetric. The diagonal elements of returned correlation matrix is set to ones.

If the correlation matrix is already positive semi-definite given the threshold, then the original correlation matrix is returned.

cov_clipped is 40 or more times faster than cov_nearest in simple example, but has a slightly larger approximation error.

statsmodels.stats.correlation_tools.cov_nearest

statsmodels.stats.correlation_tools.cov_nearest(cov, method='clipped', threshold=1e-15, n_fact=100, return_all=False)

Find the nearest covariance matrix that is positive (semi-) definite

This leaves the diagonal, i.e. the variance, unchanged

Parameters:
- cov : ndarray, (k,k)
  initial covariance matrix
- method : string
  if “clipped”, then the faster but less accurate corr_clipped is used. if “nearest”, then corr_nearest is used
- threshold : float
  clipping threshold for smallest eigen value, see Notes
- nfact : int or float
  factor to determine the maximum number of iterations in corr_nearest. See its doc string
- return_all : bool
  if False (default), then only the covariance matrix is returned. If True, then correlation matrix and standard deviation are additionally returned.

Returns:
- cov_ : ndarray
  corrected covariance matrix
- corr_ : ndarray, (optional)
  corrected correlation matrix
- std_ : ndarray, (optional)
  standard deviation

See also:
- corr_nearest, corr_clipped

Notes

This converts the covariance matrix to a correlation matrix. Then, finds the nearest correlation matrix that is positive semidefinite and converts it back to a covariance matrix using the initial standard deviation.

The smallest eigenvalue of the intermediate correlation matrix is approximately equal to the threshold. If the threshold=0, then the smallest eigenvalue of the correlation matrix might be negative, but zero within a numerical error, for example in the range of -1e-16.
Assumes input covariance matrix is symmetric.

These are utility functions to convert between central and non-central moments, skew, kurtosis and cumulants.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cum2mc(kappa)</code></td>
<td>convert non-central moments to cumulants</td>
</tr>
<tr>
<td><code>mc2mnc(mc)</code></td>
<td>convert central to non-central moments, uses recursive formula</td>
</tr>
<tr>
<td><code>mc2mvsk(args)</code></td>
<td>convert central moments to mean, variance, skew, kurtosis</td>
</tr>
<tr>
<td><code>mnc2cum(mnc)</code></td>
<td>convert non-central moments to cumulants</td>
</tr>
<tr>
<td><code>mnc2mc(mnc[, wmean])</code></td>
<td>convert non-central to central moments, uses recursive formula optionally adjusts first moment to return mean</td>
</tr>
<tr>
<td><code>mnc2mvsk(args)</code></td>
<td>convert central moments to mean, variance, skew, kurtosis</td>
</tr>
<tr>
<td><code>mvs2mc(args)</code></td>
<td>convert mean, variance, skew, kurtosis to central moments</td>
</tr>
<tr>
<td><code>mvs2mnc(args)</code></td>
<td>convert mean, variance, skew, kurtosis to non-central moments</td>
</tr>
<tr>
<td><code>cov2corr(cov[, return_std])</code></td>
<td>convert covariance matrix to correlation matrix given standard deviation</td>
</tr>
<tr>
<td><code>corr2cov(corr, std)</code></td>
<td>convert correlation matrix to covariance matrix</td>
</tr>
<tr>
<td><code>se_cov(cov)</code></td>
<td>get standard deviation from covariance matrix</td>
</tr>
</tbody>
</table>

**References**

Kenneth Lange: Numerical Analysis for Statisticians, page 40 [link](http://books.google.ca/books?id=gm7kwtyRT0C&pg=PA40&lpg=PA40&dq=convert+cumulants+to+moments&source=web&ots=qyIaY6oaWH&sig=cShTDWl-YrWAzV7NlcMTRQV6y0A&hl=en&sa=X&oi=book_result&resnum=1&ct=result)

**Examples**

```python
from statsmodels.stats.moment_helpers import cum2mc
kappa = ...  # non-central moments
kappa_cum = cum2mc(kappa)
print(kappa_cum)  # cumulants
```

```python
from statsmodels.stats.moment_helpers import mc2mnc
mc = ...  # central moments
mnc = mc2mnc(mc)
print(mnc)  # non-central moments
```

```python
from statsmodels.stats.moment_helpers import mc2mvsk
args = (...)
mean, var, skew, kurt = mc2mvsk(args)
print(mean, var, skew, kurt)  # mean, variance, skew, kurtosis
```
statsmodels.stats.moment_helpers.mnc2mvsk

convert central moments to mean, variance, skew, kurtosis

statsmodels.stats.moment_helpers.mvsk2mc

convert mean, variance, skew, kurtosis to central moments

statsmodels.stats.moment_helpers.mvsk2mnc

convert mean, variance, skew, kurtosis to non-central moments

statsmodels.stats.moment_helpers.cov2corr

convert covariance matrix to correlation matrix

Parameters:
cov : array_like, 2d
covariance matrix, see Notes

Returns:
corr : ndarray (subclass)
correlation matrix

return_std : bool
If this is true then the standard deviation is also returned. By default only the correlation matrix is returned.

Notes

This function does not convert subclasses of ndarrays. This requires that division is defined elementwise. np.ma.array and np.matrix are allowed.

statsmodels.stats.moment_helpers.corr2cov

convert correlation matrix to covariance matrix given standard deviation

Parameters:
corr : array_like, 2d
correlation matrix, see Notes
std : array_like, 1d
standard deviation

Returns:
cov : ndarray (subclass)
covariance matrix
Notes

This function does not convert subclasses of ndarrays. This requires that multiplication is defined elementwise. np.ma.array are allowed, but not matrices.

\[ \text{statsmodels.stats.moment_helpers.se_cov} \]

\[ \text{statsmodels.stats.moment_helpers.se_cov}(cov) \]

get standard deviation from covariance matrix

just a shorthand function np.sqrt(np.diag(cov))

**Parameters**

cov : array_like, square

covariance matrix

**Returns**

std : ndarray

standard deviation from diagonal of cov

3.9 Nonparametric Methods nonparametric

This section collects various methods in nonparametric statistics. This includes kernel density estimation for univariate and multivariate data, kernel regression and locally weighted scatterplot smoothing (loess).

sandbox.nonparametric contains additional functions that are work in progress or don’t have unit tests yet. We are planning to include here nonparametric density estimators, especially based on kernel or orthogonal polynomials, smoothers, and tools for nonparametric models and methods in other parts of statsmodels.

3.9.1 Kernel density estimation

The kernel density estimation (KDE) functionality is split between univariate and multivariate estimation, which are implemented in quite different ways.

Univariate estimation (as provided by \texttt{KDEUnivariate}) uses FFT transforms, which makes it quite fast. Therefore it should be preferred for continuous, univariate data if speed is important. It supports using different kernels; bandwidth estimation is done only by a rule of thumb (Scott or Silverman).

Multivariate estimation (as provided by \texttt{KDEMultivariate}) uses product kernels. It supports least squares and maximum likelihood cross-validation for bandwidth estimation, as well as estimating mixed continuous, ordered and unordered data. The default kernels (Gaussian, Wang-Ryzin and Aitchison-Aitken) cannot be altered at the moment however. Direct estimation of the conditional density \(P(X|Y) = P(X, Y)/P(Y)\) is supported by \texttt{KDEMultivariateConditional}.

\texttt{KDEMultivariate} can do univariate estimation as well, but is up to two orders of magnitude slower than \texttt{KDEUnivariate}.

3.9.2 Kernel regression

Kernel regression (as provided by \texttt{KernelReg}) is based on the same product kernel approach as \texttt{KDEMultivariate}, and therefore has the same set of features (mixed data, cross-validated bandwidth estimation, kernels) as described above for \texttt{KDEMultivariate}. Censored regression is provided by \texttt{KernelCensoredReg}.

Note that code for semi-parametric partial linear models and single index models, based on \texttt{KernelReg}, can be found in the sandbox.
3.9.3 References

- B.W. Silverman, “Density Estimation for Statistics and Data Analysis”

3.9.4 Module Reference

The public functions and classes are

<table>
<thead>
<tr>
<th>Function/Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smoothers_lowess.lowess</td>
<td>LOWESS (Locally Weighted Scatterplot Smoothing)</td>
</tr>
<tr>
<td>kde.KDEUnivariate</td>
<td>Univariate Kernel Density Estimator.</td>
</tr>
<tr>
<td>kernel_density.KDEMultivariate</td>
<td>Multivariate kernel density estimator.</td>
</tr>
<tr>
<td>kernel_density.KDEMultivariateConditional</td>
<td>Conditional multivariate kernel density estimator.</td>
</tr>
<tr>
<td>kernel_density.EstimatorSettings</td>
<td>Object to specify settings for density estimation or regression.</td>
</tr>
<tr>
<td>kernel_regression.KernelReg</td>
<td>Nonparametric kernel regression class.</td>
</tr>
</tbody>
</table>

statsmodels.nonparametric.smoothers_lowess.lowess

```python
statsmodels.nonparametric.smoothers_lowess.lowess(endog, exog, frac=0.6666666666666666, it=3, delta=0.0, is_sorted=False, missing='drop', return_sorted=True)
```

LOWESS (Locally Weighted Scatterplot Smoothing)

A lowess function that outs smoothed estimates of endog at the given exog values from points (exog, endog)

**Parameters**

- `endog`: 1-D numpy array
  - The y-values of the observed points

- `exog`: 1-D numpy array
  - The x-values of the observed points

- `frac`: float
  - Between 0 and 1. The fraction of the data used when estimating each y-value.

- `it`: int
  - The number of residual-based reweightings to perform.

- `delta`: float
  - Distance within which to use linear-interpolation instead of weighted regression.

- `is_sorted`: bool
  - If False (default), then the data will be sorted by exog before calculating lowess. If True, then it is assumed that the data is already sorted by exog.

- `missing`: str
  -
Available options are ‘none’, ‘drop’, and ‘raise’. If ‘none’, no nan checking is done. If ‘drop’, any observations with nans are dropped. If ‘raise’, an error is raised. Default is ‘drop’.

return_sorted : bool

If True (default), then the returned array is sorted by exog and has missing (nan or infinite) observations removed. If False, then the returned array is in the same length and the same sequence of observations as the input array.

Returns out: ndarray, float :

The returned array is two-dimensional if return_sorted is True, and one dimensional if return_sorted is False. If return_sorted is True, then a numpy array with two columns. The first column contains the sorted x (exog) values and the second column the associated estimated y (endog) values. If return_sorted is False, then only the fitted values are returned, and the observations will be in the same order as the input arrays.

Notes

This lowess function implements the algorithm given in the reference below using local linear estimates.

Suppose the input data has N points. The algorithm works by estimating the smooth y_i by taking the frac*N closest points to (x_i,y_i) based on their x values and estimating y_i using a weighted linear regression. The weight for (x_j,y_j) is tricube function applied to abs(x_i-x_j).

If it > 1, then further weighted local linear regressions are performed, where the weights are the same as above times the _lowess_bisquare function of the residuals. Each iteration takes approximately the same amount of time as the original fit, so these iterations are expensive. They are most useful when the noise has extremely heavy tails, such as Cauchy noise. Noise with less heavy-tails, such as t-distributions with df>2, are less problematic. The weights downgrade the influence of points with large residuals. In the extreme case, points whose residuals are larger than 6 times the median absolute residual are given weight 0.

delta can be used to save computations. For each x_i, regressions are skipped for points closer than delta. The next regression is fit for the farthest point within delta of x_i and all points in between are estimated by linearly interpolating between the two regression fits.

Judicious choice of delta can cut computation time considerably for large data (N > 5000). A good choice is delta = 0.01 * range(exog).

Some experimentation is likely required to find a good choice of frac and iter for a particular dataset.

References


Examples

The below allows a comparison between how different the fits from lowess for different values of frac can be.

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> lowess = sm.nonparametric.lowess
>>> x = np.random.uniform(low = -2*np.pi, high = 2*np.pi, size=500)
>>> y = np.sin(x) + np.random.normal(size=len(x))
```
```python
>>> z = lowess(y, x)
>>> w = lowess(y, x, frac=1./3)

This gives a similar comparison for when it is 0 vs not.

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluate</td>
<td>Evaluate density at a single point.</td>
</tr>
<tr>
<td>fit</td>
<td>Attach the density estimate to the KDEUnivariate class.</td>
</tr>
<tr>
<td>icdf()</td>
<td>Inverse Cumulative Distribution (Quantile) Function</td>
</tr>
<tr>
<td>sf()</td>
<td>Returns the survival function evaluated at the support.</td>
</tr>
</tbody>
</table>

**statsmodels.nonparametric.kde.KDEUnivariate.cdf**

```python
static KDEUnivariate.cdf()
```

Returns the cumulative distribution function evaluated at the support.

**Notes**

Will not work if fit has not been called.

**statsmodels.nonparametric.kde.KDEUnivariate.cumhazard**

```python
static KDEUnivariate.cumhazard()
```

Returns the hazard function evaluated at the support.

**Notes**

Will not work if fit has not been called.

**statsmodels.nonparametric.kde.KDEUnivariate.entropy**

```python
static KDEUnivariate.entropy()
```

Returns the differential entropy evaluated at the support.

**Notes**

Will not work if fit has not been called. 1e-12 is added to each probability to ensure that log(0) is not called.

**statsmodels.nonparametric.kde.KDEUnivariate.evaluate**

```python
KDEUnivariate.evaluate(point)
```

Evaluate density at a single point.

**Parameters**

- **point**: float
  - Point at which to evaluate the density.
KDEUnivariate.fit(kernel='gau', bw='normal_reference', fft=True, weights=None, gridsize=None, adjust=1, cut=3, clip=(-inf, inf))

Attach the density estimate to the KDEUnivariate class.

Parameters:

**kernel**: str

The Kernel to be used. Choices are:

- “biw” for biweight
- “cos” for cosine
- “epa” for Epanechnikov
- “gau” for Gaussian.
- “tri” for triangular
- “triw” for triweight
- “uni” for uniform

**bw**: str, float

The bandwidth to use. Choices are:

- “scott” - 1.059 * A * nobs ** (-1/5.), where A is min(std(X), IQR/1.34)
- “silverman” - .9 * A * nobs ** (-1/5.), where A is min(std(X), IQR/1.34)
- “normal_reference” - C * A * nobs ** (-1/5.), where C is calculated from the kernel. Equivalent (up to 2 dp) to the “scott” bandwidth for gaussian kernels. See bandwidths.py

- If a float is given, it is the bandwidth.

**fft**: bool

Whether or not to use FFT. FFT implementation is more computationally efficient. However, only the Gaussian kernel is implemented. If FFT is False, then a ‘nobs’ x ‘gridsize’ intermediate array is created.

**gridsize**: int

If gridsize is None, max(len(X), 50) is used.

**cut**: float

Defines the length of the grid past the lowest and highest values of X so that the kernel goes to zero. The end points are -/+ cut*bw*{min(X) or max(X)}

**adjust**: float

An adjustment factor for the bw. Bandwidth becomes bw * adjust.

KDEUnivariate.icdf()

Inverse Cumulative Distribution (Quantile) Function
Notes

Will not work if fit has not been called. Uses scipy.stats.mstats.mquantiles.

statsmodels.nonparametric.kde.KDEUnivariate.sf

```python
static KDEUnivariate.sf()
```

Returns the survival function evaluated at the support.

Notes

Will not work if fit has not been called.

statsmodels.nonparametric.kernel_density.KDEMultivariate

class statsmodels.nonparametric.kernel_density.KDEMultivariate(data, var_type, bw=None, defaults=<statsmodels.nonparametric._kernel_base.EstimatorSettings object at 0x081ED30>)

Multivariate kernel density estimator.

This density estimator can handle univariate as well as multivariate data, including mixed continuous / ordered discrete / unordered discrete data. It also provides cross-validated bandwidth selection methods (least squares, maximum likelihood).

Parameters  data: list of ndarrays or 2-D ndarray :

The training data for the Kernel Density Estimation, used to determine the bandwidth(s). If a 2-D array, should be of shape (num_observations, num_variables). If a list, each list element is a separate observation.

var_type: str :

The type of the variables:

- c : continuous
- u : unordered (discrete)
- o : ordered (discrete)

The string should contain a type specifier for each variable, so for example `var_type='ccuo'`.

bw: array_like or str, optional :

If an array, it is a fixed user-specified bandwidth. If a string, should be one of:

- normal_reference: normal reference rule of thumb (default)
- cv_ml: cross validation maximum likelihood
- cv_ls: cross validation least squares

defaults: EstimatorSettings instance, optional :

The default values for (efficient) bandwidth estimation.
See also:
KDEMultivariateConditional

Examples

```python
>>> import statsmodels.api as sm
>>> nobs = 300
>>> np.random.seed(1234)  # Seed random generator
>>> c1 = np.random.normal(size=(nobs,1))
>>> c2 = np.random.normal(2, 1, size=(nobs,1))

Estimate a bivariate distribution and display the bandwidth found:

```python
dens_u = sm.nonparametric.KDEMultivariate(data=[c1,c2],
... var_type='cc', bw='normal_reference')
```  
```python
dens_u.bw
array([ 0.39967419, 0.38423292])
```

Attributes

- **bw**: array_like  The bandwidth parameters.

Methods

- **cdf([data_predict])**  Evaluate the cumulative distribution function.
- **imse(bw)**  Returns the Integrated Mean Square Error for the unconditional KDE.
- **loo_likelihood(bw[, func])**  Returns the leave-one-out likelihood function.
- **pdf([data_predict])**  Evaluate the probability density function.

---

(statsmodels.api.KDEMultivariate)`

**.cdf** *(data_predict=None)*

Evaluate the cumulative distribution function.

**Parameters**

- **data_predict**: array_like, optional

  Points to evaluate at. If unspecified, the training data is used.

**Returns**

- **cdf_est**: array_like

  The estimate of the cdf.

**Notes**


The multivariate CDF for mixed data (continuous and ordered/unordered discrete) is estimated by:

.. math:: F(x^c, x^d) = n^{-1} \sum_{i=1}^{n} \left[ G\left( \frac{x^c - X_{i}}{h} \right) \sum_{u \leq x^d} L(X_{i}^{d}, x_{i}^{d}, \lambda) \right]

```
where \( G() \) is the product kernel CDF estimator for the continuous and \( L() \) for the discrete variables.

Used bandwidth is `self.bw`.

**statsmodels.nonparametric.kernel_density.KDEMultivariate.imse**

\[
\text{KDEMultivariate.imse}(bw) \\
\text{Returns the Integrated Mean Square Error for the unconditional KDE.}
\]

**Parameters**

`bw: array_like`

The bandwidth parameter(s).

**Returns**

`CV: float`

The cross-validation objective function.

**Notes**

See p. 27 in [1] For details on how to handle the multivariate estimation with mixed data types see p.6 in [3]

The formula for the cross-validation objective function is:

\[
CV = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{N} \tilde{K}_h(X_i, X_j) - \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{N} K_h(X_i, X_j)
\]

Where \( \tilde{K}_h \) is the multivariate product convolution kernel (consult [3] for mixed data types).

**statsmodels.nonparametric.kernel_density.KDEMultivariate.loo_likelihood**

\[
\text{KDEMultivariate.loo_likelihood}(bw, func=<function \text{<lambda> at 0x07FDD330}>)
\]

Returns the leave-one-out likelihood function.

The leave-one-out likelihood function for the unconditional KDE.

**Parameters**

`bw: array_like`

The value for the bandwidth parameter(s).

`func: callable, optional`

Function to transform the likelihood values (before summing); for the log likelihood, use `func=np.log`. Default is \( f(x) = x \).

**Notes**

The leave-one-out kernel estimator of \( f_{-i} \) is:

\[
f_{-i}(X_i) = \frac{1}{(n-1)h_i} \sum_{j=1, j \neq i} K_h(X_i, X_j)
\]
where \( K_h \) represents the generalized product kernel estimator:

\[
K_h(X_i, X_j) = \prod_{s=1}^{q} h_s^{-1} k \left( \frac{X_{is} - X_{js}}{h_s} \right)
\]

**statsmodels.nonparametric.kernel_density.KDEMultivariate.pdf**

*KDEMultivariate.pdf*(\(data\_predict=None\))

Evaluate the probability density function.

Parameters

- **data_predict**: array_like, optional
  Points to evaluate at. If unspecified, the training data is used.

Returns

- **pdf_est**: array_like
  Probability density function evaluated at \(data\_predict\).

**Notes**

The probability density is given by the generalized product kernel estimator:

\[
K_h(X_i, X_j) = \prod_{s=1}^{q} h_s^{-1} k \left( \frac{X_{is} - X_{js}}{h_s} \right)
\]

**statsmodels.nonparametric.kernel_density.KDEMultivariateConditional**

class *statsmodels.nonparametric.kernel_density.KDEMultivariateConditional*(\(endog, exog, dep\_type, indep\_type, bw, defaults=<statsmodels.nonparametrics._kernel_base.EstimatorSettings object at 0x081EDAF0>\))

Conditional multivariate kernel density estimator.

Calculates \(P(Y_1, Y_2, \ldots Y_n \mid X_1, X_2, \ldots X_n) = \frac{P(X_1, X_2, \ldots X_n, Y_1, Y_2, \ldots, Y_n)}{P(X_1, X_2, \ldots, X_m)}\). The conditional density is by definition the ratio of the two densities, see \[R4\].

Parameters

- **endog**: list of ndarrays or 2-D ndarrays
  The training data for the dependent variables, used to determine the bandwidth(s). If a 2-D array, should be of shape (num_observations, num_variables). If a list, each list element is a separate observation.

- **exog**: list of ndarrays or 2-D ndarrays
The training data for the independent variable; same shape as `endog`.

**dep_type:** `str`

The type of the dependent variables:

- `c`: Continuous
- `u`: Unordered (Discrete)
- `o`: Ordered (Discrete)

The string should contain a type specifier for each variable, so for example `dep_type='ccuo'`.

**indep_type:** `str`

The type of the independent variables; specified like `dep_type`.

**bw:** `array_like` or `str`, optional

If an array, it is a fixed user-specified bandwidth. If a string, should be one of:

- `normal_reference`: normal reference rule of thumb (default)
- `cv_ml`: cross validation maximum likelihood
- `cv_ls`: cross validation least squares

**defaults:** Instance of class `EstimatorSettings`

The default values for the efficient bandwidth estimation

**Attributes**

- `bw`: `array_like`

The bandwidth parameters

**See also:**

`KDEMultivariate`

**References**

[R4]

**Examples**

```python
>>> import statsmodels.api as sm
>>> nobs = 300
>>> c1 = np.random.normal(size=(nobs,1))
>>> c2 = np.random.normal(2,1,size=(nobs,1))
>>> dens_c = sm.nonparametric.KDEMultivariateConditional(endog=[c1],
... exog=[c2], dep_type='c', indep_type='c', bw='normal_reference')
>>> dens_c.bw  # show computed bandwidth
array([ 0.41223484, 0.40976931])
```
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdf([endog_predict, exog_predict])</td>
<td>Cumulative distribution function for the conditional density.</td>
</tr>
<tr>
<td>imse(bw)</td>
<td>The integrated mean square error for the conditional KDE.</td>
</tr>
<tr>
<td>loo_likelihood(bw[, func])</td>
<td>Returns the leave-one-out conditional likelihood of the data.</td>
</tr>
<tr>
<td>pdf([endog_predict, exog_predict])</td>
<td>Evaluate the probability density function.</td>
</tr>
</tbody>
</table>

**statsmodels.nonparametric.kernel_density.KDEMultivariateConditional.cdf**

KDEMultivariateConditional.cdf (endog_predict=None, exog_predict=None)
Cumulative distribution function for the conditional density.

**Parameters**

- endog_predict: array_like, optional:
  
The evaluation dependent variables at which the cdf is estimated. If not specified the training dependent variables are used.

- exog_predict: array_like, optional:
  
The evaluation independent variables at which the cdf is estimated. If not specified the training independent variables are used.

**Returns**

cdf_est: array_like:

The estimate of the cdf.

**Notes**

For more details on the estimation see [5], and p.181 in [1].

The multivariate conditional CDF for mixed data (continuous and ordered/unordered discrete) is estimated by:

\[
F(y|x) = \frac{n^{-1}\sum_{i=1}^{n}G\left(frac{y-Y_{i}}{h_{0}}\right)W_{h}(X_{i},x)}{\hat{\mu}(x)}
\]

where \(G()\) is the product kernel CDF estimator for the dependent \((y)\) variable(s) and \(W()\) is the product kernel CDF estimator for the independent variable(s).

**statsmodels.nonparametric.kernel_density.KDEMultivariateConditional.imse**

KDEMultivariateConditional.imse (bw)
The integrated mean square error for the conditional KDE.

**Parameters**

- bw: array_like:
  
The bandwidth parameter(s).

**Returns**

CV: float:

The cross-validation objective function.
Notes

For more details see pp. 156-166 in [1]. For details on how to handle the mixed variable types see [3].

The formula for the cross-validation objective function for mixed variable types is:

\[ CV(h, \lambda) = \frac{1}{n} \sum_{l=1}^{n} \frac{G_{-l}(X_l)}{[\mu_{-l}(X_l)]^2} - \frac{2}{n} \sum_{l=1}^{n} \frac{f_{-l}(X_l, Y_l)}{\mu_{-l}(X_l)} \]

where

\[ G_{-l}(X_l) = n^{-2} \sum_{i \neq l} \sum_{j \neq l} K_{X_i, X_l} K_{X_j, X_l} K_{Y_i, Y_j}^{(2)} \]

where \( K_{X_i, X_l} \) is the multivariate product kernel and \( \mu_{-l}(X_l) \) is the leave-one-out estimator of the pdf.

\( K_{Y_i, Y_j}^{(2)} \) is the convolution kernel.

The value of the function is minimized by the \_cv_ls method of the GenericKDE class to return the bw estimates that minimize the distance between the estimated and “true” probability density.

**statsmodels.nonparametric.kernel_density.KDEMultivariateConditional.loo_likelihood**

KDEMultivariateConditional.loo_likelihood(bw, func=<function <lambda> at 0x07FDD530>)

Returns the leave-one-out conditional likelihood of the data.

If `func` is not equal to the default, what’s calculated is a function of the leave-one-out conditional likelihood.

**Parameters bw: array_like** :

The bandwidth parameter(s).

**func: callable, optional** :

Function to transform the likelihood values (before summing); for the log likelihood, use `func=np.log`. Default is `f(x) = x`.

**Returns L: float** :

The value of the leave-one-out function for the data.

**Notes**

Similar to KDE.loo_likelihood’, but substitute \( \ell(y|x) = f(x, y) / f(y) \) for \( f(x) \).

**statsmodels.nonparametric.kernel_density.KDEMultivariateConditional.pdf**

KDEMultivariateConditional.pdf(endog_predict=None, exog_predict=None)

Evaluate the probability density function.

**Parameters endog_predict: array_like, optional** :
Evaluation data for the dependent variables. If unspecified, the training data is used.

**exog_predict**: array_like, optional:
Evaluation data for the independent variables.

**Returns pdf**: array_like:
The value of the probability density at `endog_predict` and `exog_predict`.

**Notes**
The formula for the conditional probability density is:

\[
f(X|Y) = \frac{f(X, Y)}{f(Y)}
\]

with

\[
f(X) = \prod_{s=1}^{q} h_s^{-1} k \left( \frac{X_is - X_js}{h_s} \right)
\]

where \( k \) is the appropriate kernel for each variable.

### `statsmodels.nonparametric.kernel_density.EstimatorsSettings`

```python
class statsmodels.nonparametric.kernel_density.EstimatorsSettings (efficient=False, randomize=False, n_res=25, n_sub=50, return_median=True, return_only_bw=False, n_jobs=-1)
```

Object to specify settings for density estimation or regression.

`EstimatorsSettings` has several properties related to how bandwidth estimation for the `KDEMultivariate`, `KDEMultivariateConditional`, `KernelReg` and `CensoredKernelReg` classes behaves.

**Parameters efficient**: bool, optional:
If True, the bandwidth estimation is to be performed efficiently – by taking smaller sub-samples and estimating the scaling factor of each subsample. This is useful for large samples (nobs >> 300) and/or multiple variables (k_vars > 3). If False (default), all data is used at the same time.

**randomize**: bool, optional:
If True, the bandwidth estimation is to be performed by taking `n_res` random resamples (with replacement) of size `n_sub` from the full sample. If set to False (default), the estimation is performed by slicing the full sample in sub-samples of size `n_sub` so that all samples are used once.

**n_sub**: int, optional:
Size of the sub-samples. Default is 50.

**n_res**: int, optional

The number of random re-samples used to estimate the bandwidth. Only has an effect if `randomize == True`. Default value is 25.

**return_median**: bool, optional

If True (default), the estimator uses the median of all scaling factors for each sub-sample to estimate the bandwidth of the full sample. If False, the estimator uses the mean.

**return_only_bw**: bool, optional

If True, the estimator is to use the bandwidth and not the scaling factor. This is not theoretically justified. Should be used only for experimenting.

**n_jobs**: int, optional

The number of jobs to use for parallel estimation with `joblib.Parallel`. Default is -1, meaning `n_cores - 1`, with `n_cores` the number of available CPU cores. See the joblib documentation for more details.

**Examples**

```python
>>> settings = EstimatorSettings(randomize=True, n_jobs=3)
>>> k_dens = KDEMultivariate(data, var_type, defaults=settings)
```

**Methods**

```python
class statsmodels.nonparametric.kernel_regression.KernelReg
```

Nonparametric kernel regression class.

Calculates the conditional mean $E[y|X]$ where $y = g(X) + e$. Note that the “local constant” type of regression provided here is also known as Nadaraya-Watson kernel regression; “local linear” is an extension of that which suffers less from bias issues at the edge of the support.

**Parameters**

- **endog**: list with one element which is array_like
  
  This is the dependent variable.

- **exog**: list
  
  The training data for the independent variable(s) Each element in the list is a separate variable

- **var_type**: str
  
  The type of the variables, one character per variable:
- c: continuous
- u: unordered (discrete)
- o: ordered (discrete)

reg_type: {'lc', 'll'}, optional:
Type of regression estimator. 'lc' means local constant and 'll' local Linear estimator. Default is 'll'

bw: str or array_like, optional:
Either a user-specified bandwidth or the method for bandwidth selection. If a string, valid values are 'cv_ls' (least-squares cross-validation) and 'aic' (AIC Hurvich bandwidth estimation). Default is 'cv_ls'.

defaults: EstimatorSettings instance, optional:
The default values for the efficient bandwidth estimation.

Attributes:

**Methods**:

r-squared: calculates the R-Squared coefficient for the model.
fit: calculates the conditional mean and marginal effects.

Methods

- aic_hurvich(bw[, func])
  Computes the AIC Hurvich criteria for the estimation of the bandwidth.
- cv_loo(bw, func)
  The cross-validation function with leave-one-out estimator.
- fit([data_predict])
  Returns the mean and marginal effects at the data_predict points.
- loo_likelihood()
- r_squared()
- sig_test(var_pos[, nboot, nested_res, pivot])
  Significance test for the variables in the regression.

statsmodels.nonparametric.kernel_regression.KernelReg.aic_hurvich

KernelReg.aic_hurvich(bw, func=None)
Computes the AIC Hurvich criteria for the estimation of the bandwidth.

Parameters bw : str or array_like
  See the bw parameter of KernelReg for details.

Returns aic : ndarray
  The AIC Hurvich criteria, one element for each variable.

func : None
  Unused here, needed in signature because it’s used in cv_loo.
References

See ch.2 in [1] and p.35 in [2].

`statsmodels.nonparametric.kernel_regression.KernelReg.cv_loo`

`KernelReg.cv_loo(bw, func)`
The cross-validation function with leave-one-out estimator.

**Parameters**

*bw*: array_like:
Vector of bandwidth values.

*func*: callable function:
Returns the estimator of g(x). Can be either _est_loc_constant (local constant) or _est_loc_linear (local_linear).

**Returns**

*L*: float:
The value of the CV function.

**Notes**

Calculates the cross-validation least-squares function. This function is minimized by compute_bw to calculate the optimal value of *bw*.

For details see p.35 in [2]

.. math::
   CV(h) = n^{-1} \sum_{i=1}^{n} (Y_{i} - g_{-i}(X_{i}))^{2}

where \( g_{-i}(X_{i}) \) is the leave-one-out estimator of g(X) and \( h \) is the vector of bandwidths

`statsmodels.nonparametric.kernel_regression.KernelReg.fit`

`KernelReg.fit(data_predict=None)`
Returns the mean and marginal effects at the *data_predict* points.

**Parameters**

*data_predict*: array_like, optional
   Points at which to return the mean and marginal effects. If not given, *data_predict* == exog.

**Returns**

*mean*: ndarray
   The regression result for the mean (i.e. the actual curve).

*mfx*: ndarray
   The marginal effects, i.e. the partial derivatives of the mean.

`statsmodels.nonparametric.kernel_regression.KernelReg.loo_likelihood`

`KernelReg.loo_likelihood()`
**statsmodels.nonparametric.kernel_regression.KernelReg.r_squared**

KernelReg.r_squared()

Returns the R-Squared for the nonparametric regression.

**Notes**

For more details see p.45 in [2] The R-Squared is calculated by:

\[
\frac{\sum_{i=1}^{n}(Y_i - \bar{y})(\hat{Y}_i - \bar{y})^2}{\sum_{i=1}^{n}(Y_i - \bar{y})^2 \sum_{i=1}^{n}(Y_i - \bar{y})^2},
\]

where \(\hat{Y}_i\) is the mean calculated in fit at the exog points.

**statsmodels.nonparametric.kernel_regression.KernelReg.sig_test**

KernelReg.sig_test(var_pos, nboot=50, nested_res=25, pivot=False)

Significance test for the variables in the regression.

- **Parameters var_pos:** sequence :
  The position of the variable in exog to be tested.

- **Returns sig:** str :
  The level of significance:
  - * : at 90% confidence level
  - ** : at 95% confidence level
  - *** : at 99% confidence level
  - “Not Significant” : if not significant

**statsmodels.nonparametric.kernel_regression.KernelCensoredReg**

class statsmodels.nonparametric.kernel_regression.KernelCensoredReg(endog, exog, var_type, reg_type, bw='cv_ls', censor_val=0, defaults=<statsmodels.nonparametric._kernel_base.EstimatorSettings object at 0x07FE12F0>)

Nonparametric censored regression.

Calculates the conditional mean \(E[y|X]\) where \(y = g(X) + e\), where \(y\) is left-censored. Left censored variable \(Y\) is defined as \(Y = \min(Y', L)\) where \(L\) is the value at which \(Y\) is censored and \(Y'\) is the true value of the variable.

- **Parameters endog:** list with one element which is array_like :
This is the dependent variable.

**exog**: list :
The training data for the independent variable(s) Each element in the list is a separate variable

**dep_type**: str :
The type of the dependent variable(s) c: Continuous u: Unordered (Discrete) o: Ordered (Discrete)

**reg_type**: str :
Type of regression estimator lc: Local Constant Estimator ll: Local Linear Estimator

**bw**: array_like :
Either a user-specified bandwidth or the method for bandwidth selection. cv_ls: cross-validation least squares aic: AIC Hurvich Estimator

**censor_val**: float :
Value at which the dependent variable is censored

**defaults**: EstimatorSettings instance, optional :
The default values for the efficient bandwidth estimation

**Attributes** :

**bw**: array_like :
The bandwidth parameters

**Methods** :

*Methods* :

**r-squared** : calculates the R-Squared coefficient for the model.

**fit** : calculates the conditional mean and marginal effects.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aic_hurvich(bw[, func])</td>
<td>Computes the AIC Hurvich criteria for the estimation of the bandwidth.</td>
</tr>
<tr>
<td>censored(censor_val)</td>
<td>The cross-validation function with leave-one-out</td>
</tr>
<tr>
<td>cv_loo(bw, func)</td>
<td>Returns the marginal effects at the data_predict points.</td>
</tr>
<tr>
<td>fit([data_predict])</td>
<td>Returns the R-Squared for the nonparametric regression.</td>
</tr>
<tr>
<td>loo_likelihood()</td>
<td>Significance test for the variables in the regression.</td>
</tr>
<tr>
<td>r_squared()</td>
<td></td>
</tr>
<tr>
<td>sig_test(var_pos[, nboot, nested_res, pivot])</td>
<td></td>
</tr>
</tbody>
</table>

```python
statsmodels.nonparametric.kernel_regression.KernelCensoredReg.aic_hurvich
```

**KernelCensoredReg.aic_hurvich (bw, func=‘None)**

Computes the AIC Hurvich criteria for the estimation of the bandwidth.

**Parameters**

bw : str or array_like

See the bw parameter of KernelReg for details.

**Returns**

aic : ndarray

3.9. Nonparametric Methods nonparametric
The AIC Hurvich criteria, one element for each variable.

```
func : None

Unused here, needed in signature because it’s used in cv_loo.
```

**References**

See ch.2 in [1] and p.35 in [2].

```
statsmodels.nonparametric.kernel_regression.KernelCensoredReg.censored

KernelCensoredReg.censored(censor_val)
```

```
statsmodels.nonparametric.kernel_regression.KernelCensoredReg.cv_loo

KernelCensoredReg.cv_loo(bw, func)

The cross-validation function with leave-one-out estimator

**Parameters**

- **bw**: array_like
  Vector of bandwidth values

- **func**: callable function
  Returns the estimator of g(x). Can be either _est_loc_constant (local constant) or _est_loc_linear (local_linear).

**Returns**

- **L**: float
  The value of the CV function

**Notes**

Calculates the cross-validation least-squares function. This function is minimized by compute_bw to calculate the optimal value of bw

For details see p.35 in [2]

```
.. math:: CV(h)=n^{-1}\sum_{i=1}^{n}(Y_{i}-g_{-i}(X_{i}))^{2}
```

where $g_{-i}(X_{i})$ is the leave-one-out estimator of g(X) and $h$ is the vector of bandwidths

```
statsmodels.nonparametric.kernel_regression.KernelCensoredReg.fit

KernelCensoredReg.fit(data_predict=None)

Returns the marginal effects at the data_predict points.
```

```
statsmodels.nonparametric.kernel_regression.KernelCensoredReg.loo_likelihood

KernelCensoredReg.loo_likelihood()
```
KernelCensoredReg.r_squared()

Returns the R-Squared for the nonparametric regression.

Notes

For more details see p.45 in [2] The R-Squared is calculated by:

\[
R^2 = \frac{\sum_{i=1}^{n} (Y_i - \bar{y})(\hat{Y}_i - \bar{y})^2}{\sum_{i=1}^{n} (Y_i - \bar{y})^2 \sum_{i=1}^{n} (Y_i - \bar{y})^2},
\]

where \( \hat{Y}_i \) is the mean calculated in fit at the exog points.

KernelCensoredReg.sig_test(var_pos, nboot=50, nested_res=25, pivot=False)

Significance test for the variables in the regression.

Parameters var_pos: sequence :

The position of the variable in exog to be tested.

Returns sig: str :

The level of significance:

- *: at 90% confidence level
- **: at 95% confidence level
- ***: at 99% confidence level
- "Not Significant": if not significant

helper functions for kernel bandwidths

bandwidths.bw_scott(x[, kernel]) Scott’s Rule of Thumb
bandwidths.bw_silverman(x[, kernel]) Silverman’s Rule of Thumb
bandwidths.select_bandwidth(x, bw, kernel) Selects bandwidth for a selection rule bw

statsmodels.nonparametric.bandwidths.bw_scott

statsmodels.nonparametric.bandwidths.bw_scott(x, kernel=None)

Scott’s Rule of Thumb

Parameters x : array-like

Array for which to get the bandwidth

kernel : CustomKernel object

Unused

Returns bw : float
The estimate of the bandwidth

Notes

Returns $1.059 \times A \times n^{-(1/5)}$ where

$$A = \min\left(\text{std}(x, \text{ddof}=1), \frac{\text{IQR}}{1.349}\right)$$

$$\text{IQR} = \text{np.subtract.reduce}(\text{np.percentile}(x, [75, 25]))$$

References


statsmodels.nonparametric.bandwidths.bw_silverman

statsmodels.nonparametric.bandwidths.bw_silverman(x, kernel=None)
Silverman’s Rule of Thumb

Parameters

- **x**: array-like
  
  Array for which to get the bandwidth

- **kernel**: CustomKernel object
  
  Unused

Returns

- **bw**: float
  
  The estimate of the bandwidth

Notes

Returns $0.9 \times A \times n^{-(1/5)}$ where

$$A = \min\left(\text{std}(x, \text{ddof}=1), \frac{\text{IQR}}{1.349}\right)$$

$$\text{IQR} = \text{np.subtract.reduce}(\text{np.percentile}(x, [75, 25]))$$

References


statsmodels.nonparametric.bandwidths.select_bandwidth

statsmodels.nonparametric.bandwidths.select_bandwidth(x, bw, kernel)

Selects bandwidth for a selection rule bw

this is a wrapper around existing bandwidth selection rules

Parameters

- **x**: array-like
  
  Array for which to get the bandwidth

- **bw**: string
name of bandwidth selection rule, currently supported are: normal_reference, scott, silverman

**kernel** : not used yet

**Returns**  
**bw** : float

The estimate of the bandwidth

There are some examples for nonlinear functions in `statsmodels.nonparametric.dgp_examples`

The sandbox.nonparametric contains additional insufficiently tested classes for testing functional form and for semi-linear and single index models.

### 3.10 Generalized Method of Moments `gmm`

`statsmodels.gmm` contains model classes and functions that are based on estimation with Generalized Method of Moments. Currently the general non-linear case is implemented. An example class for the standard linear instrumental variable model is included. This has been introduced as a test case, it works correctly but it does not take the linear structure into account. For the linear case we intend to introduce a specific implementation which will be faster and numerically more accurate.

Currently, GMM takes arbitrary non-linear moment conditions and calculates the estimates either for a given weighting matrix or iteratively by alternating between estimating the optimal weighting matrix and estimating the parameters. Implementing models with different moment conditions is done by subclassing GMM. In the minimal implementation only the moment conditions, `momcond` have to be defined.

#### 3.10.1 Module Reference

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GMM</code></td>
<td>Class for estimation by Generalized Method of Moments needs to be subclassed, where the subclass defined the moment conditions <code>momcond</code></td>
</tr>
<tr>
<td><code>GMMResults</code></td>
<td>just a storage class right now</td>
</tr>
<tr>
<td><code>IV2SLS</code></td>
<td>Class for instrumental variables estimation using Two-Stage Least-Squares</td>
</tr>
<tr>
<td><code>IVGMM</code></td>
<td>Basic class for instrumental variables estimation using GMM</td>
</tr>
<tr>
<td><code>IVGMMResults</code></td>
<td>Results class for for an OLS model.</td>
</tr>
<tr>
<td><code>IVRegressionResults</code></td>
<td>class for linear instrumental variables models estimated with GMM</td>
</tr>
<tr>
<td><code>NonlinearIVGMM</code></td>
<td>Class for non-linear instrumental variables estimation wusing GMM</td>
</tr>
</tbody>
</table>

**statsmodels.sandbox.regression.gmm.GMM**

**class** `statsmodels.sandbox.regression.gmm.GMM`(*endog, exog, instrument[, k_moms, ...])`

Class for estimation by Generalized Method of Moments needs to be subclassed, where the subclass defined the moment conditions `momcond`

**Parameters**

- **endog** : array
  - endogenous variable, see notes

- **exog** : array
  - array of exogenous variables, see notes

- **instrument** : array
  - array of instruments, see notes

#### 3.10. Generalized Method of Moments `gmm`
**nnoms** : None or int

number of moment conditions, if None then it is set equal to the number of columns of instruments. Mainly needed to determine the shape or size of start parameters and starting weighting matrix.

**kwds** : anything

this is mainly if additional variables need to be stored for the calculations of the moment conditions

**Returns**

*Attributes* :

- **results** : instance of GMMResults
currently just a storage class for params and cov_params without its own methods
- **bse** : property
  return bse

**Notes**

The GMM class only uses the moment conditions and does not use any data directly. endog, exog, instrument and kwds in the creation of the class instance are only used to store them for access in the moment conditions. Which of this is required and how they are used depends on the moment conditions of the subclass.

**Warning:**

Options for various methods have not been fully implemented and are still missing in several methods.

TODO: currently onestep (maxiter=0) still produces an updated estimate of bse and cov_params.

**Methods**

- **calc_weightmatrix**(moms[, weights_method, ...])
calculate omega or the weighting matrix
- **fit**([start_params, maxiter, inv_weights, ...])
Estimate parameters using GMM and return GMMResults
- **fitgmm**(start[, weights, optim_method, ...])
estimate parameters using GMM
- **fitgmm_cu**(start[, optim_method, optim_args])
estimate parameters using continuously updating GMM
- **fititer**(start[, maxiter, start_invweights, ...])
itative estimation with updating of optimal weighting matrix
- **from_formula**(formula, data[, subset])
Create a Model from a formula and dataframe.
- **gmmobjective**(params, weights)
objective function for GMM minimization
- **gmmobjective_cu**(params[, weights_method, wargs])
objective function for continuously updating GMM minimization
- **gradient_momcond**(params[, epsilon, centered])
gradient of moment conditions.
- **momcond_mean**(params)
mean of moment conditions.
- **predict**(params[, exog])
After a model has been fit predict returns the fitted values.
- **score**(params, weights[, epsilon, centered])
- **score_cu**(params[, epsilon, centered])
- **start_weights**(inv)

```python
statsmodels.sandbox.regression.gmm.GMM.calc_weightmatrix

GMM.calc_weightmatrix(moms, weights_method='cov', wargs=(), params=None)
calculate omega or the weighting matrix
```

Parameters

**moms** : array, (nobs, nmoms)
moment conditions for all observations evaluated at a parameter value

weights_method : string 'cov'

If method='cov' is cov then the matrix is calculated as simple covariance of the moment conditions. see fit method for available aoptions for the weight and covariance matrix

wargs : tuple or dict

parameters that are required by some kernel methods to estimate the long-run covariance. Not used yet.

Returns w : array (nmoms, nmoms)

estimate for the weighting matrix or covariance of the moment condition

Notes

currently a constant cutoff window is used TODO: implement long-run cov estimators, kernel-based Newey-West Andrews Andrews-Moy

References

Greene Hansen, Bruce

statsmodels.sandbox.regression.gmm.GMM.fit

GMM.fit(start_params=None, maxiter=10, inv_weights=None, weights_method='cov', wargs=(), has_optimal_weights=True, optim_method='bfgs', optim_args=None)

Estimate parameters using GMM and return GMMResults

TODO: weight and covariance arguments still need to be made consistent with similar options in other models, see RegressionResult.get_robustcov_results

Parameters start_params : array (optional)

starting value for parameters ub minimization. If None then fitstart method is called for the starting values.

maxiter : int or ‘cue’

Number of iterations in iterated GMM. The onestep estimate can be obtained with maxiter=0 or 1. If maxiter is large, then the iteration will stop either at maxiter or on convergence of the parameters (TODO: no options for convergence criteria yet.) If maxiter == 'cue', the the continuously updated GMM is calculated which updates the weight matrix during the minimization of the GMM objective function. The CUE estimation uses the onestep parameters as starting values.

inv_weights : None or ndarray

inverse of the starting weighting matrix. If inv_weights are not given then the method start_weights is used which depends on the subclass, for IV subclasses inv_weights = z'z where z are the instruments, otherwise an identity matrix is used.

weights_method : string, defines method for robust
Options here are similar to `statsmodels.stats.robust_covariance` default is heteroscedasticity consistent, HC0
currently available methods are

- `cov`: HC0, optionally with degrees of freedom correction
- `hac`:
- `iid`: untested, only for Z*u case, IV cases with u as error indep of Z
- `ac`: not available yet
- `cluster`: not connected yet
- others from robust_covariance

`wargs`: tuple or dict,
required and optional arguments for weights_method

- `centered`: bool, indicates whether moments are centered for the calculation of the weights and covariance matrix, applies to all weight_methods
- `ddof`: int degrees of freedom correction, applies currently only to `cov`
- `maxlag`: int number of lags to include in HAC calculation, applies only to `hac`
- others not yet, e.g. groups for cluster robust

`has_optimal_weights`: If true, then the calculation of the covariance:
matrix assumes that we have optimal GMM with \( W = S^{-1} \). Default is True.
TODO: do we want to have a different default after `onestep`?

`optim_method`: string, default is ‘bfgs’
numerical optimization method. Currently not all optimizers that are available in LikelihoodModels are connected.

`optim_args`: dict
keyword arguments for the numerical optimizer.

**Returns**
`results`: instance of GMMResults
this is also attached as attribute results

**Notes**

Warning: One-step estimation, `maxiter` either 0 or 1, still has problems (at least compared to Stata’s gmm).
By default it uses a heteroscedasticity robust covariance matrix, but uses the assumption that the weight matrix is optimal. See options for `cov_params` in the results instance.

The same options as for weight matrix also apply to the calculation of the estimate of the covariance matrix of the parameter estimates.

`statsmodels.sandbox.regression.gmm.GMM.fitgmm`

```python
GMM.fitgmm(start, weights=None, optim_method='bfgs', optim_args=None)
```
estimate parameters using GMM

**Parameters**
`start`: array_like
starting values for minimization

weights : array

weighting matrix for moment conditions. If weights is None, then the identity
matrix is used

Returns paramest : array

estimated parameters

Notes

todo: add fixed parameter option, not here ???
uses scipy.optimize.fmin

statsmodels.sandbox.regression.gmm.GMM.fitgmm_cu

GMM . fitgmm_cu (start, optim_method=‘bfgs’, optim_args=None)
estimate parameters using continuously updating GMM

Parameters start : array_like

starting values for minimization

Returns paramest : array

estimated parameters

Notes

todo: add fixed parameter option, not here ???
uses scipy.optimize.fmin

statsmodels.sandbox.regression.gmm.GMM.fititer

GMM . fititer (start, maxiter=2, start_invweights=None, weights_method=‘cov’, wargs=(), optim_method=‘bfgs’, optim_args=None)
iterative estimation with updating of optimal weighting matrix

stopping criteria are maxiter or change in parameter estimate less than self.epsilon_iter, with default 1e-6.

Parameters start : array

starting value for parameters

maxiter : int

maximum number of iterations

start_weights : array (nmoms, nmoms)

initial weighting matrix; if None, then the identity matrix is used

weights_method : {‘cov’, ‘…’}

method to use to estimate the optimal weighting matrix, see calc_weightmatrix
for details
Returns  **params** : array

estimated parameters

**weights** : array

optimal weighting matrix calculated with final parameter estimates

### statsmodels.sandbox.regression.gmm.GMM.from_formula

**classmethod** `GMM.from_formula(formula, data, subset=None, *args, **kwargs)`

Create a Model from a formula and dataframe.

**Parameters**  

**formula** : str or generic Formula object

The formula specifying the model

**data** : array-like

The data for the model. See Notes.

**subset** : array-like

An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

**args** : extra arguments

These are passed to the model

**kwargs** : extra keyword arguments

These are passed to the model.

**Returns**  

**model** : Model instance

**Notes**

data must define `__getitem__` with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

### statsmodels.sandbox.regression.gmm.GMM.gmmobjective

`GMM.gmmobjective(params, weights)`

objective function for GMM minimization

**Parameters**  

**params** : array

parameter values at which objective is evaluated

**weights** : array

weighting matrix

**Returns**  

**jval** : float

value of objective function
statsmodels.sandbox.regression.gmm.GMM.gmmobjective_cu

GMM.gmmobjective_cu (params, weights_method='cov', wargs=())
objective function for continuously updating GMM minimization

Parameters  params : array
    parameter values at which objective is evaluated

Returns  jval : float
    value of objective function

statsmodels.sandbox.regression.gmm.GMM.gradient_momcond

GMM.gradient_momcond (params, epsilon=0.0001, centered=True)
gradient of moment conditions

Parameters  params : ndarray
    parameter at which the moment conditions are evaluated

epsilon : float
    stepsize for finite difference calculation

centered : bool
    This refers to the finite difference calculation. If centered is true, then the centered
    finite difference calculation is used. Otherwise the one-sided forward differences
    are used.

TODO: looks like not used yet :
        missing argument weights

statsmodels.sandbox.regression.gmm.GMM.momcond_mean

GMM.momcond_mean (params)
mean of moment conditions,

statsmodels.sandbox.regression.gmm.GMM.predict

GMM.predict (params, exog=None, *args, **kwargs)
After a model has been fit predict returns the fitted values.
This is a placeholder intended to be overwritten by individual models.

statsmodels.sandbox.regression.gmm.GMM.score

GMM.score (params, weights, epsilon=None, centered=True)

statsmodels.sandbox.regression.gmm.GMM.score_cu

GMM.score_cu (params, epsilon=None, centered=True)
GMM.

### start_weights

**Signature**

```python
GMM.start_weights(inv=True)
```

**Attributes**

- `endog_names`
- `exog_names`
- `results_class`

```python
str(object) -> string
```

### GMMResults

**Class**

```python
class statsmodels.sandbox.regression.gmm.GMMResults(*args, **kwds)
```

**Description**

Just a storage class right now.

### Methods

- `calc_cov_params(moms, gradmoms[, weights, ...])`
  
  Calculate covariance of parameter estimates.

- `compare_j(other)`
  
  Compute the F-test for a joint linear hypothesis.

- `conf_int([alpha, cols, method])`
  
  Returns the confidence interval of the fitted parameters.

- `cov_params(**kwds)`
  
  Standard error of the parameter estimates with options.

- `f_test(r_matrix[, q_matrix, cov_p, scale, ...])`
  
  Compute the F-test for a joint linear hypothesis.

- `get_bse(**kwds)`
  
  Compute the F-test for a joint linear hypothesis.

- `jtest()`
  
  Overidentification test.

- `jval()`
  
  Overidentification test.

- `llf()`
  
  Load a pickle, (class method).

- `normalized_cov_params()`
  
  Call self.model.predict with self.params as the first argument.

- `predict([exog, transform])`
  
  Remove data arrays, all nobs arrays from result and model.

- `pvalues()`
  
  Save a pickle of this instance.

- `remove_data()`
  
  Summarize the Regression Results.

- `save(fname[, remove_data])`
  
  Compute a t-test for a joint linear hypothesis of the form Rb = q.

- `summary([yname, xname, title, alpha])`
  
  Return the t-statistic for a given parameter estimate.

- `t_test(r_matrix[, q_matrix, cov_p, scale, use_t])`
  
  Compute a Wald-test for a joint linear hypothesis.

- `tvalues()`
  
  Compute a Wald-test for a joint linear hypothesis.

```python
statsmodels.sandbox.regression.gmm.GMMResults.calc_cov_params
```

**Signature**

```python
GMMResults.calc_cov_params(moms, gradmoms[, weights, ...])
```

**Description**

Calculate covariance of parameter estimates.

Not all options tried out yet.

If weights matrix is given, then the formula use to calculate cov_params depends on whether has_optimal_weights is true. If no weights are given, then the weight matrix is calculated with the given method, and has_optimal_weights is assumed to be true.
(API Note: The latter assumption could be changed if we allow for has_optimal_weights=None.)

```python
statsmodels.sandbox.regression.gmm.GMMResults.compare_j
```

**GMMResults.compare_j(other)**

Overidentification test for comparing two nested gmm estimates

This assumes that some moment restrictions have been dropped in one of the GMM estimates relative to the other.

Not tested yet

We are comparing two separately estimated models, that use different weighting matrices. It is not guaranteed that the resulting difference is positive.

TODO: Check in which cases Stata programs use the same weights

```python
statsmodels.sandbox.regression.gmm.GMMResults.conf_int
```

**GMMResults.conf_int(alpha=0.05, cols=None, method='default')**

Returns the confidence interval of the fitted parameters.

**Parameters**

- `alpha`: float, optional
  
  The `alpha` level for the confidence interval. i.e., The default `alpha = .05` returns a 95% confidence interval.

- `cols`: array-like, optional
  
  `cols` specifies which confidence intervals to return

- `method`: string
  
  Not Implemented Yet Method to estimate the confidence interval. “Default” : uses self.bse which is based on inverse Hessian for MLE  “jhj” : “jac” : “boot-bse” “boot_quant” “profile”

**Returns**

`conf_int`: array

Each row contains [lower, upper] confidence interval

**Notes**

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

**Examples**

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[[-5496529.48322745, -1467987.78596704],
       [-177.02903529, 207.15277984],
       [-0.1115811, 0.03994274],
       [-3.12506664, -0.91539297],
       [3.10. Generalized Method of Moments gmm
```
>>> results.conf_int(cols=(2,3))
array([[ -0.1115811 , 0.03994274],
       [-3.12506664, -0.91539297]])

statsmodels.sandbox.regression.gmm.GMMResults.cov_params

GMMResults.cov_params(**kwds)

statsmodels.sandbox.regression.gmm.GMMResults.f_test

GMMResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
Compute the F-test for a joint linear hypothesis.

Parameters:
- **r_matrix** : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.
- **q_matrix** : array-like
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.
- **cov_p** : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
- **scale** : float, optional
  Default is 1.0 for no scaling.
- **invcov** : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes
The matrix r_matrix is assumed to be non-singular. More precisely,
r_matrix (pX pX.T) r_matrix.T
is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10,
df_denom=9, df_num=6>

Compare this to
```n
```python
>>> results.F
330.2853392346658
>>> results.F_p
4.98403096572e-10
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the
5th and 6th regressors are equal.
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
df_num=2>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)
```

```python
statsmodels.sandbox.regression.gmm.GMMResults.get_bse
```

GMMResults.get_bse(**kwds)
standard error of the parameter estimates with options

Parameters

- **kwds** : optional keywords
  options for calculating cov_params

Returns

- **bse** : ndarray
  estimated standard error of parameter estimates
statsmodels.sandbox.regression.gmm.GMMResults.initialize

GMMResults.initialize(model, params, **kwd)

statsmodels.sandbox.regression.gmm.GMMResults.jtest

GMMResults.jtest()
overidentification test

I guess this is missing a division by nobs, what's the normalization in jval?

statsmodels.sandbox.regression.gmm.GMMResults.jval

static GMMResults.jval()

statsmodels.sandbox.regression.gmm.GMMResults.llf

static GMMResults.llf()

statsmodels.sandbox.regression.gmm.GMMResults.load

classmethod GMMResults.load(fname)
load a pickle, (class method)

Parameters fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance :

statsmodels.sandbox.regression.gmm.GMMResults.normalized_cov_params

GMMResults.normalized_cov_params()

statsmodels.sandbox.regression.gmm.GMMResults.predict

GMMResults.predict(exog=None, transform=True, *args, **kwargs)
Call self.model.predict with self.params as the first argument.

Parameters exog : array-like, optional
The values for which you want to predict.

transform : bool, optional
If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you'd need to log the data first.

Returns See self.model.predict :
GMMResults.pvalues

GMMResults.q

GMMResults.remove_data

GMMResults.save

GMMResults.summary
Default is y

default 

xname : list of strings, optional

Default is $\text{var_{##}}$ for ## in p the number of regressors

title : string, optional

Title for the top table. If not None, then this replaces the default title

alpha : float

significance level for the confidence intervals

Returns smry : Summary instance

this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results

statsmodels.sandbox.regression.gmm.GMMResults.t_test

gmm.GMMResults.t_test$(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)$

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

Parameters r_matrix : array-like, str, tuple

• array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.

• str : The full hypotheses to test can be given as a string. See the examples.

• tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like or scalar, optional

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

An optional scale to use. Default is the scale specified by the model fit.

use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues individual t statistics

f_test for F tests

patsy.DesignInfo.linear_constraint
Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0.  0.  0.  0.  0.  1. -1.]
```

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>> print(T_test) 
<T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

```python
statsmodels.sandbox.regression.gmm.GMMResults.tvalues
static
GMMResults.tvalues()
Return the t-statistic for a given parameter estimate.
```

```python
statsmodels.sandbox.regression.gmm.GMMResults.wald_test
GMMResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)
Compute a Wald-test for a joint linear hypothesis.

Parameters r_matrix : array-like, str, or tuple
    • array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
    • str : The full hypotheses to test can be given as a string. See the examples.
    • tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

q_matrix : array-like
    This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional
    An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
```

3.10. Generalized Method of Moments gmm
scale : float, optional

Default is 1.0 for no scaling.

invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,

\[ r_matrix \cdot (pX \cdot pX^T) \cdot r_matrix^T \]

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Attributes

- **bse** standard error of the parameter estimates

statsmodels.sandbox.regression.gmm.IV2SLS

class statsmodels.sandbox.regression.gmm.IV2SLS (endog, exog, instrument=None)

Class for instrumental variables estimation using Two-Stage Least-Squares

Parameters

- **endog**: array 1d
  
  endogenous variable

- **exog**: array
  
  explanatory variables

- **instruments**: array
  
  instruments for explanatory variables, needs to contain those exog variables that are not instrumented out

Notes

All variables in exog are instrumented in the calculations. If variables in exog are not supposed to be instrumented out, then these variables need also to be included in the instrument array.

Degrees of freedom in the calculation of the standard errors uses \( df_{resid} = (nobs - k_{vars}) \). (This corresponds to the small option in Stata’s ivreg2.)
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit()</code></td>
<td>estimate model using 2SLS IV regression</td>
</tr>
<tr>
<td><code>from_formula</code>(formula, data[, subset])</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td><code>hessian</code>(params)</td>
<td>The Hessian matrix of the model</td>
</tr>
<tr>
<td><code>information</code>(params)</td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td><code>initialize()</code></td>
<td></td>
</tr>
<tr>
<td><code>loglike</code>(params)</td>
<td>Log-likelihood of model.</td>
</tr>
<tr>
<td><code>predict</code>(params[, exog])</td>
<td>Return linear predicted values from a design matrix.</td>
</tr>
<tr>
<td><code>score</code>(params)</td>
<td>Score vector of model.</td>
</tr>
<tr>
<td><code>whiten</code>(X)</td>
<td></td>
</tr>
</tbody>
</table>

`statsmodels.sandbox.regression.gmm.IV2SLS.fit`

**IV2SLS.fit()**

estimate model using 2SLS IV regression

**Returns**  
results : instance of RegressionResults  
regression result

**Notes**

This returns a generic RegressionResults instance as defined for the linear models.

Parameter estimates and covariance are correct, but other results haven’t been tested yet, to see whether they apply without changes.

`statsmodels.sandbox.regression.gmm.IV2SLS.from_formula`

**classmethod IV2SLS.from_formula**(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

**Parameters**  
formula : str or generic Formula object  
The formula specifying the model

data : array-like  
The data for the model. See Notes.

subset : array-like  
An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

args : extra arguments  
These are passed to the model

kwargs : extra keyword arguments  
These are passed to the model.

**Returns**  
model : Model instance
Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

```
statsmodels.sandbox.regression.gmm.IV2SLS.hessian
```

```
IV2SLS.hessian(params)
The Hessian matrix of the model
```

```
statsmodels.sandbox.regression.gmm.IV2SLS.information
```

```
IV2SLS.information(params)
Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.
```

```
statsmodels.sandbox.regression.gmm.IV2SLS.initialize
```

```
IV2SLS.initialize()
```

```
statsmodels.sandbox.regression.gmm.IV2SLS.loglike
```

```
IV2SLS.loglike(params)
Log-likelihood of model.
```

```
statsmodels.sandbox.regression.gmm.IV2SLS.predict
```

```
IV2SLS.predict(params, exog=None)
Return linear predicted values from a design matrix.

Parameters

- **exog**: array-like
  Design / exogenous data

- **params**: array-like, optional after fit has been called
  Parameters of a linear model

Returns

- An array of fitted values

Notes

If the model as not yet been fit, params is not optional.

```
statsmodels.sandbox.regression.gmm.IV2SLS.score
```

```
IV2SLS.score(params)
Score vector of model.

The gradient of logL with respect to each parameter.
```
IV2SLS\texttt{.whiten}(X)

Attributes

\begin{verbatim}
  endog_names
  exog_names
\end{verbatim}

\texttt{IVGMM\class}

\begin{verbatim}
class \texttt{IVGMM}(\texttt{endog, exog, instrument, k_moms=None,}
  k_params=None, missing='none', **kwds)

Basic class for instrumental variables estimation using GMM

A linear function for the conditional mean is defined as default but the methods should be overwritten by subclasses, currently \texttt{LinearIVGMM} and \texttt{NonlinearIVGMM} are implemented as subclasses.

See also: \texttt{LinearIVGMM, NonlinearIVGMM}
\end{verbatim}

Methods

\begin{verbatim}
calc_weightmatrix(moms[, weights_method, ...])
fit([start_params, maxiter, inv_weights, ...])
fitgmm(start[, weights, optim_method, ...])
fitgmm_cu(start[, optim_method, optim_args])
fitstart()
from_formula(formula, data[, subset])
gmmobjective(params, weights)
gmmobjective_cu(params[, weights_method, wargs])
gradient_momcond(params[, epsilon, centered])
momcond(params)
momcond_mean(params)
predict(params[, exog])
score(params, weights[, epsilon, centered])
score_cu(params[, epsilon, centered])
start_weights([inv])
\end{verbatim}

\texttt{IVGMM\class\calc_weightmatrix\method}

\begin{verbatim}
IVGMM\calc_weightmatrix(moms, weights_method='cov', wargs=(), params=None)

calculate omega or the weighting matrix

Parameters moms : array, (nobs, nmoms)
moment conditions for all observations evaluated at a parameter value

weights_method : string 'cov'

3.10. Generalized Method of Moments gmm
If method='cov' is cov then the matrix is calculated as simple covariance of the
moment conditions. see fit method for available aoptions for the weight and co-
variance matrix

\texttt{wargs} : tuple or dict

parameters that are required by some kernel methods to estimate the long-run
covariance. Not used yet.

\textbf{Returns} \texttt{w} : array (nmoms, nmoms)

estimate for the weighting matrix or covariance of the moment condition

\textbf{Notes}

currently a constant cutoff window is used TODO: implement long-run cov estimators, kernel-based
Newey-West Andrews Andrews-Moy???

\textbf{References}

Greene Hansen, Bruce

\texttt{statsmodels.sandbox.regression.gmm.IVGMM.fit}

\texttt{IVGMM.fit} (start_params=None, maxiter=10, inv_weights=None, weights_method='cov', wargs=(),
has_optimal_weights=True, optim_method='bfgs', optim_args=None)

Estimate parameters using GMM and return GMMResults

TODO: weight and covariance arguments still need to be made consistent with similar options in other
models, see RegressionResult.get_robustcov_results

\textbf{Parameters} \texttt{start_params} : array (optional)

starting value for parameters ub minimization. If None then fitstart method is
called for the starting values.

\texttt{maxiter} : int or 'cue'

Number of iterations in iterated GMM. The onestep estimate can be obtained with
maxiter=0 or 1. If maxiter is large, then the iteration will stop either at maxiter
or on convergence of the parameters (TODO: no options for convergence crite-
ria yet.) If maxiter == 'cue', the the continuously updated GMM is calculated
which updates the weight matrix during the minimization of the GMM objective
function. The CUE estimation uses the onestep parameters as starting values.

\texttt{inv_weights} : None or ndarray

inverse of the starting weighting matrix. If inv_weights are not given then the
method \texttt{start_weight} is used which depends on the subclass, for IV subclasses
\texttt{inv_weights} = \texttt{z'z} where \texttt{z} are the instruments, otherwise an identity matrix is
used.

\texttt{weights_method} : string, defines method for robust

Options here are similar to \texttt{statsmodels.stats.robust_covariance}
default is heteroscedasticity consistent, HC0

currently available methods are
• **cov**: HC0, optionally with degrees of freedom correction

• **hac**:

• **iid**: untested, only for Z*u case, IV cases with u as error indep of Z

• **ac**: not available yet

• **cluster**: not connected yet

• others from robust_covariance

**wargs**

- tuple or dict,
  - required and optional arguments for weights_method
  - **centered**: bool, indicates whether moments are centered for the calculation of the weights and covariance matrix, applies to all weight_methods
  - **ddof**: int degrees of freedom correction, applies currently only to cov
  - **maxlag**: int number of lags to include in HAC calculation, applies only to hac
  - others not yet, e.g. groups for cluster robust

**has_optimal_weights**: If true, then the calculation of the covariance:

matrix assumes that we have optimal GMM with \( W = S^{-1} \). Default is True.

TODO: do we want to have a different default after onestep?

**optim_method**: string, default is ‘bfgs’

- numerical optimization method. Currently not all optimizers that are available in LikelihoodModels are connected.

**optim_args**: dict

- keyword arguments for the numerical optimizer.

**Returns results**: instance of GMMResults

- this is also attached as attribute results

**Notes**

Warning: One-step estimation, **maxiter** either 0 or 1, still has problems (at least compared to Stata’s gmm). By default it uses a heteroscedasticity robust covariance matrix, but uses the assumption that the weight matrix is optimal. See options for cov_params in the results instance.

The same options as for weight matrix also apply to the calculation of the estimate of the covariance matrix of the parameter estimates.

**statsmodels.sandbox.regression.gmm.IVGMM.fitgmm**

```py
IVGMM.fitgmm(start, weights=None, optim_method='bfgs', optim_args=)
```

estimate parameters using GMM

**Parameters start**: array_like

- starting values for minimization

**weights**: array
weighting matrix for moment conditions. If weights is None, then the identity matrix is used.

Returns: paramest : array
estimated parameters

Notes

todo: add fixed parameter option, not here ???
uses scipy.optimize.fmin

```
statsmodels.sandbox.regression.gmm.IVGMM.fitgmm_cu

IVGMM.fitgmm_cu(start, optim_method='bfgs', optim_args=None)
estimate parameters using continuously updating GMM

Parameters: start : array_like
starting values for minimization

Returns: paramest : array
estimated parameters

Notes

todo: add fixed parameter option, not here ???
uses scipy.optimize.fmin

```

```
statsmodels.sandbox.regression.gmm.IVGMM.fititer

IVGMM.fititer(start, maxiter=2, start_invweights=None, weights_method='cov', wargs=(), optim_method='bfgs', optim_args=None)
iterative estimation with updating of optimal weighting matrix
stopping criteria are maxiter or change in parameter estimate less than self.epsilon_iter, with default 1e-6.

Parameters: start : array
starting value for parameters

maxiter : int
maximum number of iterations

start_invweights : array (nmoms, nmoms)
initial weighting matrix; if None, then the identity matrix is used

weights_method : {'cov', ...}
method to use to estimate the optimal weighting matrix, see calc_weightmatrix for details

Returns: params : array
estimated parameters

```
weights : array
    optimal weighting matrix calculated with final parameter estimates

IVGMM.fitstart

IVGMM.from_formula

classmethod IVGMM.from_formula(formula, data, subset=None, *args, **kwargs)
    Create a Model from a formula and dataframe.

    Parameters
    formula : str or generic Formula object
        The formula specifying the model
    data : array-like
        The data for the model. See Notes.
    subset : array-like
        An array-like object of booleans, integers, or index values that indicate the subset
        of df to use in the model. Assumes df is a pandas.DataFrame
    args : extra arguments
        These are passed to the model
    kwargs : extra keyword arguments
        These are passed to the model.
    
    Returns
    model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the
model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

IVGMM.get_error

IVGMM.gmmobjective

IVGMM.gmmobjective(params, weights)
    objective function for GMM minimization

Parameters
params : array
    parameter values at which objective is evaluated
weights : array
    weighting matrix
Returns \(jval\) : float

value of objective function

\[\text{IVGMM.gmmobjective_cu}\]

\(\text{IVGMM.gmmobjective_cu}(\text{params}, \text{weights_method}='cov', \text{wargs}())\)

objective function for continuously updating GMM minimization

Parameters \(params\) : array

parameter values at which objective is evaluated

Returns \(jval\) : float

value of objective function

\[\text{IVGMM.gradient_momcond}\]

\(\text{IVGMM.gradient_momcond}(\text{params}, \epsilon=0.0001, \text{centered}=\text{True})\)

gradient of moment conditions

Parameters \(params\) : ndarray

parameter at which the moment conditions are evaluated

\(\epsilon\) : float

stepsize for finite difference calculation

\(\text{centered}\) : bool

This refers to the finite difference calculation. If \(\text{centered}\) is true, then the centered finite difference calculation is used. Otherwise the one-sided forward differences are used.

TODO: looks like not used yet:

missing argument \(weights\)

\[\text{IVGMM.momcond}\]

\(\text{IVGMM.momcond}(\text{params})\)

\[\text{IVGMM.momcond_mean}\]

\(\text{IVGMM.momcond_mean}(\text{params})\)

mean of moment conditions,

\[\text{IVGMM.predict}\]

\(\text{IVGMM.predict}(\text{params}, \text{exog}=\text{None})\)
statsmodels.sandbox.regression.gmm.IVGMM.score

IVGMM.score(params, weights, epsilon=None, centered=True)

statsmodels.sandbox.regression.gmm.IVGMM.score_cu

IVGMM.score_cu(params, epsilon=None, centered=True)

statsmodels.sandbox.regression.gmm.IVGMM.start_weights

IVGMM.start_weights(inv=True)

Attributes

- endog_names
- exog_names
- results_class str(object) -> string

statsmodels.sandbox.regression.gmm.IVGMMResults

class statsmodels.sandbox.regression.gmm.IVGMMResults(*args, **kwds)

Methods

calc_cov_params(moms, gradmoms[, weights, ...]) calculate covariance of parameter estimates
compare_j(other) overidentification test for comparing two nested gmm estimates
conf_int([alpha, cols, method]) Returns the confidence interval of the fitted parameters.
cov_params(**kwds) standard error of the parameter estimates with options
f_test(r_matrix[, q_matrix, cov_p, scale, ...]) Compute the F-test for a joint linear hypothesis.
fittedvalues() overidentification test
get_bse(**kwds) standard error of the parameter estimates with options
initialize(model, params, **kwd) overidentification test
jtest() load a pickle, (class method)
jval() Call self.model.predict with self.params as the first argument.
llf() remove data arrays, all nobs arrays from result and model
load(fname) save(fname[, remove_data])
normalized_cov_params() save a pickle of this instance
predict([exog, transform]) ssr()
pvalues() t_test(r_matrix[, q_matrix, cov_p, scale, use_t])
t_test(r_matrix[, q_matrix, cov_p, scale, use_t])
tvalues() Summarize the Regression Results

Continued on next page
statsmodels Documentation, Release 0.6.0

Table 3.232 – continued from previous page

<table>
<thead>
<tr>
<th>wald_test(r_matrix[, q_matrix, cov_p, ...])</th>
<th>Compute a Wald-test for a joint linear hypothesis.</th>
</tr>
</thead>
</table>

**statsmodels.sandbox.regression.gmm.IVGMResults.calc_cov_params**

IVGMMResults.calc_cov_params (moms, gradmoms, weights=None, use_weights=False, has_optimal_weights=True, weights_method='cov', wargs=())

calculate covariance of parameter estimates

not all options tried out yet

If weights matrix is given, then the formula use to calculate cov_params depends on whether has_optimal_weights is true. If no weights are given, then the weight matrix is calculated with the given method, and has_optimal_weights is assumed to be true.

(API Note: The latter assumption could be changed if we allow for has_optimal_weights=None.)

**statsmodels.sandbox.regression.gmm.IVGMResults.compare_j**

IVGMMResults.compare_j (other)

overidentification test for comparing two nested gmm estimates

This assumes that some moment restrictions have been dropped in one of the GMM estimates relative to the other.

Not tested yet

We are comparing two separately estimated models, that use different weighting matrices. It is not guaranteed that the resulting difference is positive.

TODO: Check in which cases Stata programs use the same weights

**statsmodels.sandbox.regression.gmm.IVGMResults.conf_int**

IVGMMResults.conf_int (alpha=0.05, cols=None, method='default')

Returns the confidence interval of the fitted parameters.

**Parameters** alpha : float, optional

The alpha level for the confidence interval. i.e., The default alpha = .05 returns a 95% confidence interval.

cols : array-like, optional

cols specifies which confidence intervals to return

method : string

Not Implemented Yet Method to estimate the confidence interval. “Default” : uses self.bse which is based on inverse Hessian for MLE “j*h” : “jac” : “boot-bse” “boot_quant” “profile”

Returns conf_int : array

Each row contains [lower, upper] confidence interval
Notes

The confidence interval is based on the standard normal distribution. Models wish to use a different distribution should overwrite this method.

Examples

```python
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> results.conf_int()
array([[-5496529.48322745, -1467987.78596704],
      [ -177.02903529,  207.15277984],
      [ -0.1115811 ,  0.03994274],
      [ -3.12506664, -0.91539297],
      [ -1.5179487 , -0.54850503],
      [ -0.56251721,  0.460309  ],
      [  798.7875153 ,  2859.51541392]])

>>> results.conf_int(cols=(2,3))
array([[-0.1115811 ,  0.03994274],
      [ -3.12506664, -0.91539297]])
```

statsmodels.sandbox.regression.gmm.IVGMMResults.cov_params

IVGMMResults.cov_params(**kwd)

statsmodels.sandbox.regression.gmm.IVGMMResults.f_test

IVGMMResults.f_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)

Compute the F-test for a joint linear hypothesis.

Parameters

- **r_matrix** : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- **q_matrix** : array-like
  - This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- **cov_p** : array-like, optional
  - An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- **scale** : float, optional
  - Default is 1.0 for no scaling.
invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix \( r_matrix \) is assumed to be non-singular. More precisely,
\( r_matrix (pX pX.T) r_matrix.T \)
is assumed invertible. Here, \( pX \) is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> A = np.identity(len(results.params))
>>> A = A[1:, :]
This tests that each coefficient is jointly statistically significantly different from zero.
>>> print(results.f_test(A))
<F contrast: F=330.28533923463488, p=4.98403052872e-10, df_denom=9, df_num=6>

Compare this to

```python
>>> results.F
330.2853392346658
```  
```python
>>> results.F_p
4.98403096572e-10
```  

```
>>> B = np.array([[0, 0, 1, -1, 0, 0, 0], [0, 0, 0, 0, 0, 1, -1]])
This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the 5th and 6th regressors are equal.
```

```python
>>> print(results.f_test(B))
<F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9, df_num=2>
```  

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()```
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)

statsmodels.sandbox.regression.gmm.IVGMMResults.fittedvalues

```
static IVGMMResults.fittedvalues()
```

statsmodels.sandbox.regression.gmm.IVGMMResults.get_bse

```
IVGMMResults.get_bse(**kwds)
standard error of the parameter estimates with options

Parameters kwds : optional keywords
    options for calculating cov_params

Returns bse : ndarray
    estimated standard error of parameter estimates
```

statsmodels.sandbox.regression.gmm.IVGMMResults.initialize

```
IVGMMResults.initialize(model, params, **kwd)
```

statsmodels.sandbox.regression.gmm.IVGMMResults.jtest

```
IVGMMResults.jtest()
overidentification test
    I guess this is missing a division by nobs, what’s the normalization in jval ?
```

statsmodels.sandbox.regression.gmm.IVGMMResults.jval

```
static IVGMMResults.jval()
```

statsmodels.sandbox.regression.gmm.IVGMMResults.llf

```
static IVGMMResults.llf()
```

statsmodels.sandbox.regression.gmm.IVGMMResults.load

```
classmethod IVGMMResults.load(fname)
    load a pickle, (class method)

Parameters fname : string or filehandle
    fname can be a string to a file path or filename, or a filehandle.

Returns unpickled instance :
```
IVGMMResults.normalized_cov_params()

IVGMMResults.predict(exog=None, transform=True, *args, **kwargs)
Call self.model.predict with self.params as the first argument.

Parameters:
- **exog**: array-like, optional
  The values for which you want to predict.
- **transform**: bool, optional
  If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model \( y \sim \log(x1) + \log(x2) \), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

Returns:
See self.model.predict:

IVGMMResults.pvalues()

IVGMMResults.q()

IVGMMResults.remove_data()
remove data arrays, all nobs arrays from result and model
This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

IVGMMResults.resid()

IVGMMResults.sandwich()

IVGMMResults.summary()

IVGMMResults.tvalues()

IVGMMResults.wald_test(matrix, r_matrix=None, inv_weights=None, use_f=False, cov_p=None, cov_type='nonrobust', **kwargs)
Perform Wald tests for the effects of an array of parameters.

Parameters:
- **matrix**: array
  An array giving the r by k (k <= nobs) restrictions over the coefficients
- **r_matrix**: array-like
  The restrictions over the coefficients to test in a Wald test. Default is None.
- **inv_weights**: array-like
  The inverse of the weighting matrix. Default is None.
- **use_f**: bool, optional
  If True, then the F-statistic is used; otherwise the \( \chi^2 \) statistic is used. Default is False.
- **cov_p**: array-like
  The covariance matrix of the parameters. If None, the inverse of the Hessian matrix is used. Default is None.
- **cov_type**: str
  Choose the type of covariance matrix. Only used if cov_p is not given. Default is 'nonrobust'.
- ****kwargs**: dict
  Additional keyword arguments for Wald test function.

Returns:
A results instance that contains the Wald test statistic and p-value for each hypothesis tested.

IVGMMResults.wald_test_pairwise(term, other, method='pairwise', alpha=0.05, correct=True, **kwargs)
Perform a pairwise Wald test on the model matrix.

Parameters:
- **term**: string
  The term to test against other.
- **other**: string
  The other term to test against term.
- **method**: str, optional
  The method to use for the test. Default is 'pairwise'.
- **alpha**: float, optional
  The significance level for the tests. Default is 0.05.
- **correct**: bool, optional
  If True, the p-values are adjusted for multiple testing. Default is True.
- ****kwargs**: dict
  Additional keyword arguments for Wald test function.

Returns:
A results instance that contains the Wald test statistic and p-value for each hypothesis tested.

IVGMMResults.wald_test_terms(alpha=0.05, correct=True, **kwargs)
Perform pairwise Wald tests between each term and the mean.

Parameters:
- **alpha**: float, optional
  The significance level for the tests. Default is 0.05.
- **correct**: bool, optional
  If True, the p-values are adjusted for multiple testing. Default is True.
- ****kwargs**: dict
  Additional keyword arguments for Wald test function.

Returns:
A results instance that contains the Wald test statistic and p-value for each hypothesis tested.
IVGMMResults.save (fname, remove_data=False)

save a pickle of this instance

Parameters
fname : string or filehandle
fname can be a string to a file path or filename, or a filehandle.
remove_data : bool
If False (default), then the instance is pickled without changes. If True, then all
arrays with length nobs are set to None before pickling. See the remove_data
method. In some cases not all arrays will be set to None.

Notes
If remove_data is true and the model result does not implement a remove_data method then this will raise
an exception.

IVGMMResults.ssr

static IVGMMResults.ssr()

IVGMMResults.summary (yname=None, xname=None, title=None, alpha=0.05)

Summarize the Regression Results

Parameters
yname : string, optional
Default is y
xname : list of strings, optional
Default is var_## for ## in p the number of regressors
title : string, optional
Title for the top table. If not None, then this replaces the default title
alpha : float
significance level for the confidence intervals

Returns
smry : Summary instance
this holds the summary tables and text, which can be printed or converted to
various output formats.

See also:
statsmodels.iolib.summary.Summary class to hold summary results
IVGMMResults.t_test(r_matrix, q_matrix=None, cov_p=None, scale=None, use_t=None)
Compute a t-test for a joint linear hypothesis of the form Rb = q

Parameters
r_matrix : array-like, str, tuple
  - array : If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.
q_matrix : array-like or scalar, optional
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.
cov_p : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.
scale : float, optional
  An optional scale to use. Default is the scale specified by the model fit.
use_t : bool, optional
  If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:
tvalues individual t statistics
f_test for F tests
patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]
>>> print(results.t_test(r))
{T contrast: effect=-1829.2025687192481, sd=455.3907492193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
>>> T_test.effect -1829.2025687192481 >>> T_test.sd 455.3907492193762 >>> T_test.tvalue -4.0167754636411717 >>> T_test.pvalue 0.0015163772380899498
```

r tests that the coefficients on the 5th and 6th independent variable are the same.
Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
>>> hypotheses = 'GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1'
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

```
statsmodels.sandbox.regression.gmm.IVGMMResults.tvalues

**static** IVGMMResults.tvalues()  
Return the t-statistic for a given parameter estimate.

statsmodels.sandbox.regression.gmm.IVGMMResults.wald_test

IVGMMResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)  
Compute a Wald-test for a joint linear hypothesis.

**Parameters** r_matrix : array-like, str, or tuple

- array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
- str : The full hypotheses to test can be given as a string. See the examples.
- tuple : A tuple of arrays in the form (R, q). since q_matrix is deprecated.

q_matrix : array-like

This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

cov_p : array-like, optional

An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

scale : float, optional

Default is 1.0 for no scaling.

invcov : array-like, optional

A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

use_f : bool

If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:

statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

3.10. Generalized Method of Moments gmm
Notes

The matrix \( r_{\text{matrix}} \) is assumed to be non-singular. More precisely,
\[
r_{\text{matrix}} \text{(pX pX.T)} r_{\text{matrix}.T}
\]
is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.

Attributes

- **bse** standard error of the parameter estimates

```python
class statsmodels.sandbox.regression.gmm.IVRegressionResults(model, params, normalized_cov_params=None, scale=1.0)
```

Results class for for an OLS model.

Most of the methods and attributes are inherited from RegressionResults. The special methods that are only available for OLS are:
- get_influence
- outlier_test
- el_test
- conf_int_el

See also:
RegressionResults

Methods

- **HC0_se**
- **HC1_se**
- **HC2_se**
- **HC3_se**
- **aic**
- **bic**
- **bse**
- **centered_tss**
- **compare_f_test** (restricted)
- **compare_lm_test** (restricted[, demean, use_lr])
- **compare_lr_test** (restricted[, large_sample])
- **condition_number**
- **conf_int** ([alpha, cols])
- **cov_HC0**
- **cov_HCl**

use F test to test whether restricted model is correct
Use Lagrange Multiplier test to test whether restricted model is correct
Likelihood ratio test to test whether restricted model is correct
Return condition number of exogenous matrix, calculated as ratio of largest to
Returns the confidence interval of the fitted parameters.
See statsmodels.RegressionResults
See statsmodels.RegressionResults
statsmodels.sandbox.regression.gmm.IVRegressionResults.HC0_se

static IVRegressionResults.HC0_se()
See statsmodels.RegressionResults

statsmodels.sandbox.regression.gmm.IVRegressionResults.HC1_se

static IVRegressionResults.HC1_se()
See statsmodels.RegressionResults

statsmodels.sandbox.regression.gmm.IVRegressionResults.HC2_se

static IVRegressionResults.HC2_se()
See statsmodels.RegressionResults

3.10. Generalized Method of Moments  gmm  811
IVRegressionResults.HC3_se()  
See statsmodels.RegressionResults

IVRegressionResults.aic()  

IVRegressionResults.bic()  

IVRegressionResults.bse()  

IVRegressionResults.centered_tss()  

IVRegressionResults.compare_f_test(restricted)  
use F test to test whether restricted model is correct

Parameters restricted : Result instance

The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, ssr, residual degrees of freedom, df_resid.

Returns f_value : float

test statistic, F distributed

p_value : float

p-value of the test statistic

df_diff : int

degrees of freedom of the restriction, i.e. difference in df between models

Notes

See mailing list discussion October 17,

This test compares the residual sum of squares of the two models. This is not a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results under the assumption of homoscedasticity and no autocorrelation (sphericity).
IVRegressionResults.compare_lm_test

Use Lagrange Multiplier test to test whether restricted model is correct

**Parameters**

- **restricted**: Result instance
  - The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, `ssr`, residual degrees of freedom, `df_resid`.

- **demean**: bool
  - Flag indicating whether the demean the scores based on the residuals from the restricted model. If True, the covariance of the scores are used and the LM test is identical to the large sample version of the LR test.

**Returns**

- **lm_value**: float
  - Test statistic, chi2 distributed
- **p_value**: float
  - p-value of the test statistic
- **df_diff**: int
  - Degrees of freedom of the restriction, i.e. difference in df between models

**Notes**

TODO: explain LM text

IVRegressionResults.compare_lr_test

Likelihood ratio test to test whether restricted model is correct

**Parameters**

- **restricted**: Result instance
  - The restricted model is assumed to be nested in the current model. The result instance of the restricted model is required to have two attributes, residual sum of squares, `ssr`, residual degrees of freedom, `df_resid`.

- **large_sample**: bool
  - Flag indicating whether to use a heteroskedasticity robust version of the LR test, which is a modified LM test.

**Returns**

- **lr_stat**: float
  - Likelihood ratio, chi-square distributed with `df_diff` degrees of freedom
- **p_value**: float
  - p-value of the test statistic
- **df_diff**: int
  - Degrees of freedom of the restriction, i.e. difference in df between models
Notes

The exact likelihood ratio is valid for homoskedastic data, and is defined as

\[ D = -2 \log \left( \frac{\mathcal{L}_{null}}{\mathcal{L}_{alternative}} \right) \]

where \( \mathcal{L} \) is the likelihood of the model. With \( D \) distributed as chisquare with \( df \) equal to difference in number of parameters or equivalently difference in residual degrees of freedom.

The large sample version of the likelihood ratio is defined as

\[ D = n s' S^{-1} s \]

where \( s = n^{-1} \sum_{i=1}^{n} s_i \)

\[ s_i = x_{i,alternative} \hat{\epsilon}_{i,null} \]

is the average score of the model evaluated using the residuals from null model and the regressors from the alternative model and \( S \) is the covariance of the scores, \( s_i \). The covariance of the scores is estimated using the same estimator as in the alternative model.

This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results without taking unspecified heteroscedasticity or correlation into account.

This test compares the loglikelihood of the two models. This may not be a valid test, if there is unspecified heteroscedasticity or correlation. This method will issue a warning if this is detected but still return the results without taking unspecified heteroscedasticity or correlation into account.

is the average score of the model evaluated using the residuals from null model and the regressors from the alternative model and \( S \) is the covariance of the scores, \( s_i \). The covariance of the scores is estimated using the same estimator as in the alternative model.

TODO: put into separate function, needs tests

statsmodels.sandbox.regression.gmm.IVRegressionResults.condition_number

static IVRegressionResults.condition_number()

Return condition number of exogenous matrix, calculated as ratio of largest to smallest eigenvalue.

statsmodels.sandbox.regression.gmm.IVRegressionResults.conf_int

IVRegressionResults.conf_int(alpha=0.05, cols=None)

Returns the confidence interval of the fitted parameters.

Parameters alpha : float, optional

The alpha level for the confidence interval. ie., The default alpha = .05 returns a 95% confidence interval.

cols : array-like, optional

cols specifies which confidence intervals to return
Notes

The confidence interval is based on Student’s t-distribution.

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.cov_HC0

static IVRegressionResults.cov_HC0()
See statsmodels.RegressionResults

statsmodels.sandbox.regression.gmm.IVRegressionResults.cov_HC1

static IVRegressionResults.cov_HC1()
See statsmodels.RegressionResults

statsmodels.sandbox.regression.gmm.IVRegressionResults.cov_HC2

static IVRegressionResults.cov_HC2()
See statsmodels.RegressionResults

statsmodels.sandbox.regression.gmm.IVRegressionResults.cov_HC3

static IVRegressionResults.cov_HC3()
See statsmodels.RegressionResults

statsmodels.sandbox.regression.gmm.IVRegressionResults.cov_params

IVRegressionResults.cov_params(r_matrix=None, column=None, scale=None, cov_p=None, other=None)
Returns the variance/covariance matrix.
The variance/covariance matrix can be of a linear contrast of the estimates of params or all params multiplied by scale which will usually be an estimate of sigma^2. Scale is assumed to be a scalar.

Parameters

r_matrix : array-like
    Can be 1d, or 2d. Can be used alone or with other.

column : array-like, optional
    Must be used on its own. Can be 0d or 1d see below.

scale : float, optional
    Can be specified or not. Default is None, which means that the scale argument is taken from the model.

other : array-like, optional
    Can be used when r_matrix is specified.

Returns

The below are assumed to be in matrix notation.:

cov : ndarray
    If no argument is specified returns the covariance matrix of a model:
\[(\text{scale}) \times (X^T X)^{-1} : \]

If contrast is specified it pre and post-multiples as follows:
\[(\text{scale}) \times \text{r\_matrix} \times (X^T X)^{-1} \times \text{r\_matrix}.T : \]

If contrast and other are specified returns:
\[(\text{scale}) \times \text{r\_matrix} \times (X^T X)^{-1} \times \text{other}.T : \]

If column is specified returns:
\[(\text{scale}) \times (X^T X)^{-1}[[\text{column},\text{column}]] \text{ if column is 0d} : \]

OR:
\[(\text{scale}) \times (X^T X)^{-1}[[\text{column}],[\text{column}]] \text{ if column is 1d} : \]

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.eigenvals

static IVRegressionResults.eigenvals()
Return eigenvalues sorted in decreasing order.
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.ess

static IVRegressionResults.ess()
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.f_pvalue

static IVRegressionResults.f_pvalue()
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.f_test

IVRegressionResults.f_test (r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None)
Compute the F-test for a joint linear hypothesis.

Parameters

- `r_matrix`: array-like, str, or tuple
  - array: An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- `q_matrix`: array-like
  This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.

- `cov_p`: array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, `self.normalized_cov_params` is used.

- `scale`: float, optional
  Default is 1.0 for no scaling.
invco : array-like, optional
        A q x q array to specify an inverse covariance matrix based on a restrictions
        matrix.

See also:
statsmodels.contrasts,statsmodels.model.LikelihoodModelResults.wald_test,
statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,

\( r_matrix (pX pX^T) r_matrix^T \)

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be
problems in non-OLS models where the rank of the covariance of the noise is not full.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()

>>> A = np.identity(len(results.params))
>>> A = A[1:, :]

This tests that each coefficient is jointly statistically significantly different from zero.

```python
>>> print(results.f_test(A))
(F contrast: F=330.28533923463488, p=4.98403052872e-10,
 df_denom=9, df_num=6)
```  

Compare this to

```python
>>> results.F
330.2853392346658
```  

```python
>>> results.F_p
4.98403096572e-10
```  

```python
>>> B = np.array([[0, 0, 1, -1, 0, 0], [0, 0, 0, 0, 0, 1]])

This tests that the coefficient on the 2nd and 3rd regressors are equal and jointly that the coefficient on the
5th and 6th regressors are equal.

```python
>>> print(results.f_test(B))
(F contrast: F=9.740461873303655, p=0.00560528853174, df_denom=9,
 df_num=2)
```  

Alternatively, you can specify the hypothesis tests using a string

```python
>>> from statsmodels.datasets import longley
>>> from statsmodels.formula.api import ols
>>> dta = longley.load_pandas().data
>>> formula = 'TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR'
>>> results = ols(formula, dta).fit()
```
>>> hypotheses = '(GNPDEFL = GNP), (UNEMP = 2), (YEAR/1829 = 1)'
>>> f_test = results.f_test(hypotheses)
>>> print(f_test)

`statsmodels.sandbox.regression.gmm.IVRegressionResults.fittedvalues`

```python
static IVRegressionResults.fittedvalues()
```

`statsmodels.sandbox.regression.gmm.IVRegressionResults.fvalue`

```python
static IVRegressionResults.fvalue()
```

`statsmodels.sandbox.regression.gmm.IVRegressionResults.get_robustcov_results`

```python
IVRegressionResults.get_robustcov_results(cov_type='HC1', use_t=None, **kwds)
```

create new results instance with robust covariance as default

**Parameters**

- `cov_type` : string
  the type of robust sandwich estimator to use. see Notes below

- `use_t` : bool
  If true, then the t distribution is used for inference. If false, then the normal
distribution is used.

- `kwds` : depends on `cov_type`
  Required or optional arguments for robust covariance calculation. see Notes be-
dow

**Returns**

- `results` : results instance
  This method creates a new results instance with the requested robust covariance
  as the default covariance of the parameters. Inferential statistics like p-values and
  hypothesis tests will be based on this covariance matrix.

**Notes**

The following covariance types and required or optional arguments are currently available:

- **‘HC0’, ‘HC1’, ‘HC2’, ‘HC3’ and no keyword arguments:** heteroscedasticity robust covariance

- **‘HAC’ and keywords**
  - `-maxlag` integer (required) : number of lags to use
  - `-kernel` string (optional) : kernel, default is Bartlett
  - `-use_correction` bool (optional) : [If true, use small sample] correction

- **‘cluster’ and required keyword `groups`, integer group indicator**
  - `-groups` array_like, integer (required) : index of clusters or groups
  - `-use_correction` bool (optional) : If True the sandwich covariance is calculated with a small
    sample correction. If False the the sandwich covariance is calculated without small sam-
    ple correction.
--df_correction bool (optional) If True (default), then the degrees of freedom for the inferential statistics and hypothesis tests, such as pvalues, f_pvalue, conf_int, and t_test and f_test, are based on the number of groups minus one instead of the total number of observations minus the number of explanatory variables. df_resid of the results instance is adjusted. If False, then df_resid of the results instance is not adjusted.

**‘hac-groupsum’ Driscoll and Kraay, heteroscedasticity and autocorrelation robust standard errors in panel data keywords**

- time array_like (required) : index of time periods
- maxlag integer (required) : number of lags to use
- kernel string (optional) : kernel, default is Bartlett
- use_correction False or string in ['hac', 'cluster'] (optional) : If False the the sandwich covariance is calculated without small sample correction. If use_correction = 'cluster' (default), then the same small sample correction as in the case of ‘covtype='cluster’’ is used.
- df_correction bool (optional) adjustment to df_resid, see cov_type ‘cluster’ above
  
  #TODO: we need more options here

**‘hac-panel’ heteroscedasticity and autocorrelation robust standard errors in panel data. The data needs to be sorted in this case, the time series for each panel unit or cluster need to be stacked. keywords**

- time array_like (required) : index of time periods
- maxlag integer (required) : number of lags to use
- kernel string (optional) : kernel, default is Bartlett
- use_correction False or string in ['hac', 'cluster'] (optional) : If False the the sandwich covariance is calculated without small sample correction.
- df_correction bool (optional) adjustment to df_resid, see cov_type ‘cluster’ above
  
  #TODO: we need more options here

Reminder: use_correction in “nw-groupsum” and “nw-panel” is not bool, needs to be in [False, ‘hac’, ‘cluster’]

TODO: Currently there is no check for extra or misspelled keywords, except in the case of cov_type HCy

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.initialize

IVRegressionResults.initialize(model, params, **kwds)
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.llf

static IVRegressionResults.llf()
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.load

classmethod IVRegressionResults.load(fname)
  
  load a pickle, (class method)

Parameters fname : string or filehandle
```
 fname can be a string to a file path or filename, or a filehandle.

 Returns  unpickled instance :

 statsmodels.sandbox.regression.gmm.IVRegressionResults.mse_model

 static  IVRegressionResults.mse_model()

 statsmodels.sandbox.regression.gmm.IVRegressionResults.mse_resid

 static  IVRegressionResults.mse_resid()

 statsmodels.sandbox.regression.gmm.IVRegressionResults.mse_total

 static  IVRegressionResults.mse_total()

 statsmodels.sandbox.regression.gmm.IVRegressionResults.nobs

 static  IVRegressionResults.nobs()

 statsmodels.sandbox.regression.gmm.IVRegressionResults.normalized_cov_params

 IVRegressionResults.normalized_cov_params()

 statsmodels.sandbox.regression.gmm.IVRegressionResults.predict

 IVRegressionResults.predict(exog=None, transform=True, *args, **kwargs)
 Call self.model.predict with self.params as the first argument.

 Parameters  exog : array-like, optional

 The values for which you want to predict.

 transform : bool, optional

 If the model was fit via a formula, do you want to pass exog through the formula. Default is True. E.g., if you fit a model y ~ log(x1) + log(x2), and transform is True, then you can pass a data structure that contains x1 and x2 in their original form. Otherwise, you’d need to log the data first.

 Returns  See self.model.predict :

 statsmodels.sandbox.regression.gmm.IVRegressionResults.pvalues

 static  IVRegressionResults.pvalues()
IVRegressionResults.remove_data

remove data arrays, all nobs arrays from result and model

This reduces the size of the instance, so it can be pickled with less memory. Currently tested for use with predict from an unpickled results and model instance.

**Warning:** Since data and some intermediate results have been removed calculating new statistics that require them will raise exceptions. The exception will occur the first time an attribute is accessed that has been set to None.

Not fully tested for time series models, tsa, and might delete too much for prediction or not all that would be possible.

The list of arrays to delete is maintained as an attribute of the result and model instance, except for cached values. These lists could be changed before calling remove_data.

IVRegressionResults.resid

static IVRegressionResults.resid()

IVRegressionResults.resid_pearson

static IVRegressionResults.resid_pearson()

Residuals, normalized to have unit variance.

**Returns** An array wresid/sqrt(scale):

IVRegressionResults.rsquared

static IVRegressionResults.rsquared()

IVRegressionResults.rsquared_adj

static IVRegressionResults.rsquared_adj()

IVRegressionResults.save

IVRegressionResults.save (fname, remove_data=False)

save a pickle of this instance

**Parameters** fname : string or filehandle

fname can be a string to a file path or filename, or a filehandle.

remove_data : bool

If False (default), then the instance is pickled without changes. If True, then all arrays with length nobs are set to None before pickling. See the remove_data method. In some cases not all arrays will be set to None.
Notes

If remove_data is true and the model result does not implement a remove_data method then this will raise an exception.

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.scale

static IVRegressionResults.scale()
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.spec_hausman

IVRegressionResults.spec_hausman(dof=None)

See also:

spec_hausman  generic function for Hausman’s specification test
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.ssr

static IVRegressionResults.ssr()
```

```python
statsmodels.sandbox.regression.gmm.IVRegressionResults.summary

IVRegressionResults.summary(yname=None, xname=None, title=None, alpha=0.05)

Summarize the Regression Results

Parameters yname : string, optional
    Default is y
xname : list of strings, optional
    Default is var_## for ## in p the number of regressors
title : string, optional
    Title for the top table. If not None, then this replaces the default title
alpha : float
    significance level for the confidence intervals

Returns smry : Summary instance
    this holds the summary tables and text, which can be printed or converted to various output formats.

See also:

statsmodels.iolib.summary.Summary class to hold summary results
```
**IVRegressionResults.summary2**

Experimental summary function to summarize the regression results

**Parameters**

- **xname**: List of strings of length equal to the number of parameters
  - Names of the independent variables (optional)
- **yname**: string
  - Name of the dependent variable (optional)
- **title**: string, optional
  - Title for the top table. If not None, then this replaces the default title
- **alpha**: float
  - Significance level for the confidence intervals
- **float_format**: string
  - Print format for floats in parameters summary

**Returns**

- **smry**: Summary instance
  - This holds the summary tables and text, which can be printed or converted to various output formats.

**See also**

- `statsmodels.iolib.summary.Summary` class to hold summary results

**IVRegressionResults.t_test**

Compute a t-test for a joint linear hypothesis of the form $Rb = q$

**Parameters**

- **r_matrix**: array-like, str, tuple
  - If an array is given, a p x k 2d array or length k 1d array specifying the linear restrictions.
  - str: The full hypotheses to test can be given as a string. See the examples.
  - tuple: A tuple of arrays in the form (R, q), since `q_matrix` is deprecated.
- **q_matrix**: array-like or scalar, optional
  - This is deprecated. See `r_matrix` and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and `r_matrix` is an array, `q_matrix` is assumed to be a conformable array of zeros.
- **cov_p**: array-like, optional
  - An alternative estimate for the parameter covariance matrix. If None is given, `self.normalized_cov_params` is used.
- **scale**: float, optional
  - An optional `scale` to use. Default is the scale specified by the model fit.
use_t : bool, optional

If use_t is None, then the default of the model is used. If use_t is True, then the p-values are based on the t distribution. If use_t is False, then the p-values are based on the normal distribution.

See also:

tvalues  individual t statistics
f_test  for F tests
patsy.DesignInfo.linear_constraint

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> results = sm.OLS(data.endog, data.exog).fit()
>>> r = np.zeros_like(results.params)
>>> r[5:] = [1,-1]
>>> print(r)
[ 0. 0. 0. 0. 0. 1. -1.]
```

r tests that the coefficients on the 5th and 6th independent variable are the same.

```python
>>> T_Test = results.t_test(r) >>>print(T_Test) <T contrast: effect=-1829.2025687192481, sd=455.39079425193762, t=-4.0167754636411717, p=0.0015163772380899498, df_denom=9>
>>> T_test.effect -1829.2025687192481 >>> T_test.sd 455.39079425193762 >>> T_test.tvalue -4.0167754636411717 >>> T_test.pvalue 0.0015163772380899498
```

Alternatively, you can specify the hypothesis tests using a string

```python
>>> dta = sm.datasets.longley.load_pandas().data
>>> formula = "TOTEMP ~ GNPDEFL + GNP + UNEMP + ARMED + POP + YEAR"
>>> results = ols(formula, dta).fit()
>>> hypotheses = "GNPDEFL = GNP, UNEMP = 2, YEAR/1829 = 1"
>>> t_test = results.t_test(hypotheses)
>>> print(t_test)
```

statsmodels.sandbox.regression.gmm.IVRegressionResults.tvalues

```python
static IVRegressionResults.tvalues()  
Return the t-statistic for a given parameter estimate.
```

statsmodels.sandbox.regression.gmm.IVRegressionResults.uncentered_tss

```python
static IVRegressionResults.uncentered_tss()  
```
statsmodels.sandbox.regression.gmm.IVRegressionResults.wald_test

IVRegressionResults.wald_test(r_matrix, q_matrix=None, cov_p=None, scale=1.0, invcov=None, use_f=None)

Compute a Wald-test for a joint linear hypothesis.

Parameters

- r_matrix : array-like, str, or tuple
  - array : An r x k array where r is the number of restrictions to test and k is the number of regressors.
  - str : The full hypotheses to test can be given as a string. See the examples.
  - tuple : A tuple of arrays in the form (R, q), since q_matrix is deprecated.

- q_matrix : array-like
  This is deprecated. See r_matrix and the examples for more information on new usage. Can be either a scalar or a length p row vector. If omitted and r_matrix is an array, q_matrix is assumed to be a conformable array of zeros.

- cov_p : array-like, optional
  An alternative estimate for the parameter covariance matrix. If None is given, self.normalized_cov_params is used.

- scale : float, optional
  Default is 1.0 for no scaling.

- invcov : array-like, optional
  A q x q array to specify an inverse covariance matrix based on a restrictions matrix.

- use_f : bool
  If True, then the F-distribution is used. If False, then the asymptotic distribution, chisquare is used. The test statistic is proportionally adjusted for the distribution by the number of constraints in the hypothesis.

See also:
statsmodels.contrasts, statsmodels.model.LikelihoodModelResults.f_test, statsmodels.model.LikelihoodModelResults.t_test, patsy.DesignInfo.linear_constraint

Notes

The matrix r_matrix is assumed to be non-singular. More precisely,

r_matrix (pX pX.T) r_matrix.T

is assumed invertible. Here, pX is the generalized inverse of the design matrix of the model. There can be problems in non-OLS models where the rank of the covariance of the noise is not full.
class for linear instrumental variables models estimated with GMM

Uses closed form expression instead of nonlinear optimizers for each step of the iterative GMM.

The model is assumed to have the following moment condition

\[ E( z * (y - x \beta)) = 0 \]

Where \( y \) is the dependent endogenous variable, \( x \) are the explanatory variables and \( z \) are the instruments. Variables in \( x \) that are exogenous need also be included in \( z \).

Notation Warning: our name exog stands for the explanatory variables, and includes both exogenous and explanatory variables that are endogenous, i.e. included endogenous variables

**Parameters**

- **endog**: array_like
  - dependent endogenous variable

- **exog**: array_like
  - explanatory, right hand side variables, including explanatory variables that are endogenous

- **instruments**: array_like
  - Instrumental variables, variables that are exogenous to the error in the linear model containing both included and excluded exogenous variables

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc_weightmatrix</td>
<td>calculate omega or the weighting matrix</td>
</tr>
<tr>
<td>fit</td>
<td>Estimate parameters using GMM and return GMMResults</td>
</tr>
<tr>
<td>fitgmm</td>
<td>estimate parameters using GMM for linear model</td>
</tr>
<tr>
<td>fitgmm_cu</td>
<td>estimate parameters using continuously updating GMM</td>
</tr>
<tr>
<td>fititer</td>
<td>iterative estimation with updating of optimal weighting matrix</td>
</tr>
<tr>
<td>fitstart</td>
<td></td>
</tr>
<tr>
<td>from_formula</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>get_error</td>
<td>objective function for GMM minimization</td>
</tr>
<tr>
<td>gmmobjective</td>
<td>objective function for continuously updating GMM minimization</td>
</tr>
<tr>
<td>gmmobjective_cu</td>
<td></td>
</tr>
<tr>
<td>gradient_momcond</td>
<td>mean of moment conditions.</td>
</tr>
<tr>
<td>momcond</td>
<td></td>
</tr>
<tr>
<td>momcond_mean</td>
<td></td>
</tr>
<tr>
<td>predict</td>
<td></td>
</tr>
<tr>
<td>score</td>
<td></td>
</tr>
<tr>
<td>score_cu</td>
<td></td>
</tr>
<tr>
<td>start_weights</td>
<td></td>
</tr>
</tbody>
</table>

```python
LinearIVGMM.calc_weightmatrix(moms[, weights_method, ...])
```

calculate omega or the weighting matrix
Parameters `moms` : array, (nobs, nmoms)
    moment conditions for all observations evaluated at a parameter value

`weights_method` : string 'cov'
    If method='cov' is cov then the matrix is calculated as simple covariance of the
    moment conditions. see fit method for available aotions for the weight and co-
    variance matrix

`wargs` : tuple or dict
    parameters that are required by some kernel methods to estimate the long-run
    covariance. Not used yet.

Returns `w` : array (nmoms, nmoms)
    estimate for the weighting matrix or covariance of the moment condition

Notes

currently a constant cutoff window is used TODO: implement long-run cov estimators, kernel-based
Newey-West Andrews Andrews-Moy????

References

Greene Hansen, Bruce

`statsmodels.sandbox.regression.gmm.LinearIVGMM.fit`

`LinearIVGMM.fit` (start_params=None, maxiter=10, inv_weights=None, weights_method='cov',
                 wargs=(), has_optimal_weights=True, optim_method='bfgs', op-
                 tim_args=None)
Estimate parameters using GMM and return GMMResults

TODO: weight and covariance arguments still need to be made consistent with similar options in other
models, see RegressionResult.get_robustcov_results

Parameters `start_params` : array (optional)
    starting value for parameters ub minimization. If None then fitstart method is
called for the starting values.

`maxiter` : int or 'cue'
    Number of iterations in iterated GMM. The onestep estimate can be obtained with
maxiter=0 or 1. If maxiter is large, then the iteration will stop either at maxiter
or on convergence of the parameters (TODO: no options for convergence crite-
ria yet.) If maxiter == 'cue', the the continuously updated GMM is calculated
which updates the weight matrix during the minimization of the GMM objective
function. The CUE estimation uses the onestep parameters as starting values.

`inv_weights` : None or ndarray
    inverse of the starting weighting matrix. If inv_weights are not given then the
method start_weights is used which depends on the subclass, for IV subclasses
inv_weights = z'z where z are the instruments, otherwise an identity matrix is
used.
weights_method : string, defines method for robust

Options here are similar to statsmodels.stats.robust_covariance
default is heteroscedasticity consistent, HC0
currently available methods are
• cov : HC0, optionally with degrees of freedom correction
• hac :
• iid : untested, only for Z^u case, IV cases with u as error indep of Z
• ac : not available yet
• cluster : not connected yet
• others from robust_covariance

wargs' : tuple or dict,

required and optional arguments for weights_method

• centered : bool, indicates whether moments are centered for the calculation of
  the weights and covariance matrix, applies to all weight_methods
• ddof : int degrees of freedom correction, applies currently only to cov
• maxlag : int number of lags to include in HAC calculation, applies only to hac
• others not yet, e.g. groups for cluster robust

has_optimal_weights : If true, then the calculation of the covariance :

matrix assumes that we have optimal GMM with W = S^{-1}. Default is True.
TODO: do we want to have a different default after onestep?

optim_method : string, default is ‘bfgs’
umerical optimization method. Currently not all optimizers that are available in
LikelihoodModels are connected.

optim_args : dict

keyword arguments for the numerical optimizer.

Returns results : instance of GMMResults

this is also attached as attribute results

Notes

Warning: One-step estimation, maxiter either 0 or 1, still has problems (at least compared to Stata’s gmm).
By default it uses a heteroscedasticity robust covariance matrix, but uses the assumption that the weight
matrix is optimal. See options for cov_params in the results instance.
The same options as for weight matrix also apply to the calculation of the estimate of the covariance
matrix of the parameter estimates.

statsmodels.sandbox.regression.gmm.LinearIVGMM.fitgmm

LinearIVGMM.fitgmm(start, weights=None, optim_method=None, **kwds)
estimate parameters using GMM for linear model
Uses closed form expression instead of nonlinear optimizers

**Parameters**

- **start**: not used
  - starting values for minimization, not used, only for consistency of method signature
- **weights**: array
  - weighting matrix for moment conditions. If weights is None, then the identity matrix is used
- **optim_method**: not used,
  - optimization method, not used, only for consistency of method signature
- **kwds**: keyword arguments
  - not used, will be silently ignored (for compatibility with generic)

**Returns**

- **params**: array
  - estimated parameters

---

**statsmodels.sandbox.regression.gmm.LinearIVGMM.fitgmm_cu**

**LinearIVGMM.fitgmm_cu**

```
LinearIVGMM.fitgmm_cu(start, optim_method='bfgs', optim_args=None)
```

estimate parameters using continuously updating GMM

**Parameters**

- **start**: array_like
  - starting values for minimization

**Returns**

- **params**: array
  - estimated parameters

---

**Notes**

todo: add fixed parameter option, not here ???

uses scipy.optimize.fmin

---

**statsmodels.sandbox.regression.gmm.LinearIVGMM.fititer**

**LinearIVGMM.fititer**

```
LinearIVGMM.fititer(start, maxiter=2, start_invweights=None, weights_method='cov',
                     wargs=(), optim_method='bfgs', optim_args=None)
```

iterative estimation with updating of optimal weighting matrix

stopping criteria are maxiter or change in parameter estimate less than self.epsilon_iter, with default 1e-6.

**Parameters**

- **start**: array
  - starting value for parameters
- **maxiter**: int
  - maximum number of iterations
- **start_weights**: array (nmoms, nmoms)
  - initial weighting matrix; if None, then the identity matrix is used
weights_method : {'cov', ...}
            method to use to estimate the optimal weighting matrix, see calc_weightmatrix
            for details

Returns params : array
            estimated parameters

weights : array
            optimal weighting matrix calculated with final parameter estimates

statsmodels.sandbox.regression.gmm.LinearIVGMM.fitstart

LinearIVGMM.fitstart()

statsmodels.sandbox.regression.gmm.LinearIVGMM.from_formula

classmethod LinearIVGMM.from_formula(formula, data, subset=None, *args, **kwargs)
            Create a Model from a formula and dataframe.

Parameters formula : str or generic Formula object
            The formula specifying the model
data : array-like
            The data for the model. See Notes.
subset : array-like
            An array-like object of booleans, integers, or index values that indicate the subset
            of df to use in the model. Assumes df is a pandas.DataFrame
args : extra arguments
            These are passed to the model
kwargs : extra keyword arguments
            These are passed to the model.

Returns model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the
model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

statsmodels.sandbox.regression.gmm.LinearIVGMM.get_error

LinearIVGMM.get_error(params)
statsmodels.sandbox.regression.gmm.LinearIVGMM

**gmmobjective** *(params, weights)*
objective function for GMM minimization

**Parameters**
- **params** : array
  parameter values at which objective is evaluated
- **weights** : array
  weighting matrix

**Returns**
- **jval** : float
  value of objective function

**gmmobjective_cu** *(params, weights_method='cov', wargs=())*
objective function for continuously updating GMM minimization

**Parameters**
- **params** : array
  parameter values at which objective is evaluated

**Returns**
- **jval** : float
  value of objective function

**gradient_momcond** *(params, **kwds)*

**momcond** *(params)*

**momcond_mean** *(params)*
  mean of moment conditions,

**predict** *(params, exog=None)*

**score** *(params, weights, **kwds)*
statsmodels.sandbox.regression.gmm.LinearIVGMM.score_cu

LinearIVGMM.score_cu(params, epsilon=None, centered=True)

statsmodels.sandbox.regression.gmm.LinearIVGMM.start_weights

LinearIVGMM.start_weights(inv=True)

Attributes

endog_names
exog_names
results_class str(object) -> string

statsmodels.sandbox.regression.gmm.NonlinearIVGMM

class statsmodels.sandbox.regression.gmm.NonlinearIVGMM(endog, exog, instrument, func, **kwds)

Class for non-linear instrumental variables estimation using GMM

The model is assumed to have the following moment condition

\[ E[ z * (y - f(X, \beta)) ] = 0 \]

Where \( y \) is the dependent endogenous variable, \( x \) are the explanatory variables and \( z \) are the instruments. Variables in \( x \) that are exogenous need also be included in \( z \). \( f \) is a nonlinear function.

Notation Warning: our name \textit{exog} stands for the explanatory variables, and includes both exogenous and explanatory variables that are endogenous, i.e. included endogenous variables

\textbf{Parameters}

- **endog**: array_like
  dependent endogenous variable

- **exog**: array_like
  explanatory, right hand side variables, including explanatory variables that are endogenous.

- **instruments**: array_like
  Instrumental variables, variables that are exogenous to the error in the linear model containing both included and excluded exogenous variables

- **func**: callable
  function for the mean or conditional expectation of the endogenous variable. The function will be called with parameters and the array of explanatory, right hand side variables, \( func(params, exog) \)

\textbf{Notes}

This class uses numerical differences to obtain the derivative of the objective function. If the jacobian of the conditional mean function, \( func \) is available, then it can be used by subclassing this class and defining a method \( jac_func \).
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc_weightmatrix(moms[, weights_method, ...])</td>
<td>calculate omega or the weighting matrix</td>
</tr>
<tr>
<td>fit([start_params, maxiter, inv_weights, ...])</td>
<td>Estimate parameters using GMM and return GMMResults</td>
</tr>
<tr>
<td>fitgmm(start[, weights, optim_method, ...])</td>
<td>estimate parameters using GMM</td>
</tr>
<tr>
<td>fitgmm_cu(start[, optim_method, optim_args])</td>
<td>estimate parameters using continuously updating GMM</td>
</tr>
<tr>
<td>fititer(start[, maxiter, start_invweights, ...])</td>
<td>iterative estimation with updating of optimal weighting matrix</td>
</tr>
<tr>
<td>from_formula(formula, data[, subset])</td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td>get_error(params)</td>
<td>objective function for GMM minimization</td>
</tr>
<tr>
<td>gmmobjective(params, weights)</td>
<td>objective function for continuously updating GMM minimization</td>
</tr>
<tr>
<td>gradient_momcond(params[, epsilon, centered])</td>
<td>gradient of moment conditions</td>
</tr>
<tr>
<td>jac_error(params, weights[, args, centered, ...])</td>
<td></td>
</tr>
<tr>
<td>jac_func(params, weights[, args, centered, ...])</td>
<td></td>
</tr>
<tr>
<td>momcond(params)</td>
<td>mean of moment conditions</td>
</tr>
<tr>
<td>momcond_mean(params)</td>
<td></td>
</tr>
<tr>
<td>predict(params[, exog])</td>
<td></td>
</tr>
<tr>
<td>score(params, weights, **kwds)</td>
<td></td>
</tr>
<tr>
<td>score_cu(params[, epsilon, centered])</td>
<td></td>
</tr>
<tr>
<td>start_weights([inv])</td>
<td></td>
</tr>
</tbody>
</table>

**NonlinearIVGMM.calc_weightmatrix**

NonlinearIVGMM.calc_weightmatrix(moms, weights_method='cov', wargs=(), params=None)
calculate omega or the weighting matrix

**Parameters**
- **moms** : array, (nobs, nmoms)
  moment conditions for all observations evaluated at a parameter value
- **weights_method** : string ‘cov’
  If method=’cov’ is cov then the matrix is calculated as simple covariance of the moment conditions. see fit method for available aoptions for the weight and covariance matrix
- **wargs** : tuple or dict
  parameters that are required by some kernel methods to estimate the long-run covariance. Not used yet.

**Returns**
- **w** : array (nmoms, nmoms)
  estimate for the weighting matrix or covariance of the moment condition

**Notes**

currently a constant cutoff window is used TODO: implement long-run cov estimators, kernel-based Newey-West Andrews Andrews-Moy???
statsmodels.sandbox.regression.gmm.NonlinearIVGMM.fit

NonlinearIVGMM.fit(start_params=None, maxiter=10, inv_weights=None, weights_method='cov', wargs=(), has_optimal_weights=True, optim_method='bfgs', optim_args=None)

Estimate parameters using GMM and return GMMResults

TODO: weight and covariance arguments still need to be made consistent with similar options in other models, see RegressionResult.get_robustcov_results

Parameters

start_params : array (optional)
    starting value for parameters ub minimization. If None then fitstart method is called for the starting values.

maxiter : int or 'cue'
    Number of iterations in iterated GMM. The onestep estimate can be obtained with maxiter=0 or 1. If maxiter is large, then the iteration will stop either at maxiter or on convergence of the parameters (TODO: no options for convergence criteria yet.) If maxiter == 'cue', the the continuously updated GMM is calculated which updates the weight matrix during the minimization of the GMM objective function. The CUE estimation uses the onestep parameters as starting values.

inv_weights : None or ndarray
    inverse of the starting weighting matrix. If inv_weights are not given then the method start_weights is used which depends on the subclass, for IV subclasses inv_weights = z'z where z are the instruments, otherwise an identity matrix is used.

weights_method : string, defines method for robust
    Options here are similar to statsmodels.stats.robust_covariance
default is heteroscedasticity consistent, HC0

currently available methods are
• cov : HC0, optionally with degrees of freedom correction
• hac :
• iid : untested, only for Z*u case, IV cases with u as error indep of Z
• ac : not available yet
• cluster : not connected yet
• others from robust_covariance

wargs : tuple or dict,
    required and optional arguments for weights_method
• centered : bool, indicates whether moments are centered for the calculation of the weights and covariance matrix, applies to all weight_methods
• ddof : int degrees of freedom correction, applies currently only to cov

References

Greene Hansen, Bruce
• maxlag : int number of lags to include in HAC calculation, applies only to hac

• others not yet, e.g. groups for cluster robust

**has_optimal_weights**: If true, then the calculation of the covariance:

matrix assumes that we have optimal GMM with \( W = S^{-1} \). Default is True.

TODO: do we want to have a different default after onestep?

**optim_method** : string, default is ‘bfgs’

numerical optimization method. Currently not all optimizers that are available in LikelihoodModels are connected.

**optim_args** : dict

keyword arguments for the numerical optimizer.

**Returns results** : instance of GMMResults

this is also attached as attribute results

**Notes**

Warning: One-step estimation, maxiter either 0 or 1, still has problems (at least compared to Stata’s gmm).
By default it uses a heteroscedasticity robust covariance matrix, but uses the assumption that the weight matrix is optimal. See options for cov_params in the results instance.

The same options as for weight matrix also apply to the calculation of the estimate of the covariance matrix of the parameter estimates.

```
statsmodels.sandbox.regression.gmm.NonlinearIVGMM.fitgmm
```

NonlinearIVGMM.**fitgmm**(start, weights=None, optim_method='bfgs', optim_args=None)

estimate parameters using GMM

**Parameters start** : array_like

starting values for minimization

**weights** : array

weighting matrix for moment conditions. If weights is None, then the identity
matrix is used

**Returns paramest** : array

estimated parameters

**Notes**

todo: add fixed parameter option, not here ???
uses scipy.optimize.fmin
NonlinearIVGMM.fitgmm_cu

Parameters

- **start**: array_like
  - starting values for minimization

Returns

- **params**: array
  - estimated parameters

Notes

todo: add fixed parameter option, not here ???

uses scipy.optimize.fmin

NonlinearIVGMM.fititer

Parameters

- **start**: array
  - starting value for parameters
- **maxiter**: int
  - maximum number of iterations
- **start_weights**: array (nmoms, nmoms)
  - initial weighting matrix; if None, then the identity matrix is used
- **weights_method**: {'cov', ...}
  - method to use to estimate the optimal weighting matrix, see calc_weightmatrix for details

Returns

- **params**: array
  - estimated parameters
- **weights**: array
  - optimal weighting matrix calculated with final parameter estimates

NonlinearIVGMM.fitstart

NonlinearIVGMM.fitstart()
```python
statsmodels.sandbox.regression.gmm.NonlinearIVGMM.from_formula

classmethod NonlinearIVGMM.from_formula(formula, data, subset=None, *args, **kwargs)
Create a Model from a formula and dataframe.

Parameters
formula : str or generic Formula object
    The formula specifying the model
data : array-like
    The data for the model. See Notes.
subset : array-like
    An array-like object of booleans, integers, or index values that indicate the subset
    of df to use in the model. Assumes df is a pandas.DataFrame
args : extra arguments
    These are passed to the model
kwargs : extra keyword arguments
    These are passed to the model.

Returns
model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the
model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.
```

```python
statsmodels.sandbox.regression.gmm.NonlinearIVGMM.get_error

NonlinearIVGMM.get_error(params)
```

```python
statsmodels.sandbox.regression.gmm.NonlinearIVGMM.gmmobjective

NonlinearIVGMM.gmmobjective(params, weights)
objective function for GMM minimization

Parameters
params : array
    parameter values at which objective is evaluated
weights : array
    weighting matrix

Returns
jval : float
    value of objective function
```
NonlinearIVGMM.gmmobjective_cu \((\text{params}, \text{weights\_method}='\text{cov}', \text{wargs}=())\)

objective function for continuously updating GMM minimization

**Parameters**
- **params**: array
  - parameter values at which objective is evaluated

**Returns**
- **jval**: float
  - value of objective function

NonlinearIVGMM.gradient_momcond \((\text{params}, \epsilon=0.0001, \text{centered}=\text{True})\)

gradient of moment conditions

**Parameters**
- **params**: ndarray
  - parameter at which the moment conditions are evaluated
- **epsilon**: float
  - stepsize for finite difference calculation
- **centered**: bool
  - This refers to the finite difference calculation. If `centered` is true, then the centered finite difference calculation is used. Otherwise the one-sided forward differences are used.

TODO: looks like not used yet:

- missing argument `weights`

NonlinearIVGGM.jac_error \((\text{params}, \text{weights}, \text{args}=\text{None}, \text{centered}=\text{True}, \epsilon=\text{None})\)

NonlinearIVGGM.jac_func \((\text{params}, \text{weights}, \text{args}=\text{None}, \text{centered}=\text{True}, \epsilon=\text{None})\)

NonlinearIVGGM.momcond \((\text{params})\)

NonlinearIVGGM.momcond_mean \((\text{params})\)

mean of moment conditions,
3.11 Empirical Likelihood emplike

3.11.1 Introduction

Empirical likelihood is a method of nonparametric inference and estimation that lifts the obligation of having to specify a family of underlying distributions. Moreover, empirical likelihood methods do not require re-sampling but still uniquely determine confidence regions whose shape mirrors the shape of the data. In essence, empirical likelihood attempts to combine the benefits of parametric and nonparametric methods while limiting their shortcomings. The main difficulties of empirical likelihood is the computationally intensive methods required to conduct inference. statsmodels.emplike attempts to provide a user-friendly interface that allows the end user to effectively conduct empirical likelihood analysis without having to concern themselves with the computational burdens.

Currently, emplike provides methods to conduct hypothesis tests and form confidence intervals for descriptive statistics. Empirical likelihood estimation and inference in a regression, accelerated failure time and instrumental variable model are currently under development.

References

The main reference for empirical likelihood is:

3.11.2 Examples

```python
import numpy as np
import statsmodels.api as sm

# Generate Data
x = np.random.standard_normal(50)

# initiate EL
el = sm.emplike.DescStat(x)

# confidence interval for the mean
el.ci_mean()

# test variance is 1
el.test_var(1)
```

3.11.3 Module Reference

- **descriptive.DescStatUV**(endog) A class to compute confidence intervals and hypothesis tests involving mean, variance, kurtosis and skewness of a univariate random variable.

**statsmodels.emplike.descriptive.DescStat**

**statsmodels.emplike.descriptive.DescStat**(endog)

Returns an instance to conduct inference on descriptive statistics via empirical likelihood. See DescStatUV and DescStatMV for more information.

**Parameters endog : ndarray**

Array of data

**Returns :** DescStat instance

If k=1, the function returns a univariate instance, DescStatUV. If k>1, the function returns a multivariate instance, DescStatMV.

**statsmodels.emplike.descriptive.DescStatUV**

**class statsmodels.emplike.descriptive.DescStatUV**(endog)

A class to compute confidence intervals and hypothesis tests involving mean, variance, kurtosis and skewness of a univariate random variable.

**Parameters endog : 1darray**

Data to be analyzed

**Attributes**

<table>
<thead>
<tr>
<th><strong>Attribute</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>endog</td>
<td>Data to be analyzed</td>
</tr>
<tr>
<td>nobs</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>Number of observations</td>
</tr>
</tbody>
</table>
**Methods**

- `ci_kurt([sig, upper_bound, lower_bound])`: Returns the confidence interval for kurtosis.
- `ci_mean([sig, method, epsilon, gamma_low, ...])`: Returns the confidence interval for the mean.
- `ci_skew([sig, upper_bound, lower_bound])`: Returns the confidence interval for skewness.
- `ci_var([lower_bound, upper_bound, sig])`: Returns the confidence interval for the variance.
- `plot_contour(mu_low, mu_high, var_low, ...)`: Returns a plot of the confidence region for a univariate mean and variance.
- `test_joint_skew_kurt(skew0, kurt0[, ...])`: Returns -2 x log-likelihood and the p-value for the joint test.
- `test_kurt(kurt0[, return_weights])`: Returns -2 x log-likelihood and the p-value for the hypothesized kurtosis.
- `test_mean(mu0[, return_weights])`: Returns -2 x log-likelihood ratio and p-value for a hypothesis test of the mean.
- `test_skew(skew0[, return_weights])`: Returns -2 x log-likelihood ratio and p-value for the hypothesized skewness.
- `test_var(sig2_0[, return_weights])`: Returns -2 x log-likelihood ratio and the p-value for the hypothesized variance.

---

**statsmodels.emplike.descriptive.DescStatUV.ci_kurt**

DescStatUV.ci_kurt(sig=0.05, upper_bound=None, lower_bound=None)

Returns the confidence interval for kurtosis.

**Parameters**

- `sig`: float
  The significance level. Default is .05
- `upper_bound`: float
  Maximum value of kurtosis the upper limit can be. Default is .99 confidence limit assuming normality.
- `lower_bound`: float
  Minimum value of kurtosis the lower limit can be. Default is .99 confidence limit assuming normality.

**Returns**

- `Interval`: tuple
  Lower and upper confidence limit

**Notes**

For small n, upper_bound and lower_bound may have to be provided by the user. Consider using test_kurt to find values close to the desired significance level.

If function returns f(a) and f(b) must have different signs, consider expanding the bounds.

---

**statsmodels.emplike.descriptive.DescStatUV.ci_mean**

DescStatUV.ci_mean(sig=0.05, method='gamma', epsilon=1e-08, gamma_low=-10000000000L, gamma_high=10000000000L)

Returns the confidence interval for the mean.

**Parameters**

- `sig`: float
  significance level. Default is .05
- `method`: str
Root finding method, Can be ‘nested-brent’ or ‘gamma’. Default is ‘gamma’
‘gamma’ Tries to solve for the gamma parameter in the Lagrange (see Owen pg 22) and then determine the weights.
‘nested brent’ uses brents method to find the confidence intervals but must maximize the likelihood ratio on every iteration.
gamma is generally much faster. If the optimizations does not converge, try expanding the gamma_high and gamma_low variable.

**gamma_low** : float
Lower bound for gamma when finding lower limit. If function returns f(a) and f(b) must have different signs, consider lowering gamma_low.

**gamma_high** : float
Upper bound for gamma when finding upper limit. If function returns f(a) and f(b) must have different signs, consider raising gamma_high.

**epsilon** : float
When using ‘nested-brent’, amount to decrease (increase) from the maximum (minimum) of the data when starting the search. This is to protect against the likelihood ratio being zero at the maximum (minimum) value of the data. If data is very small in absolute value (<10 ** -6) consider shrinking epsilon

When using ‘gamma’, amount to decrease (increase) the minimum (maximum) by to start the search for gamma. If function returns f(a) and f(b) must have different signs, consider lowering epsilon.

Returns **Interval** : tuple
Confidence interval for the mean

---

**statsmodels.emplike.descriptive.DescStatUV.ci_skew**

DescStatUV.ci_skew(sig=0.05, upper_bound=None, lower_bound=None)
Returns the confidence interval for skewness.

**Parameters**

**sig** : float
The significance level. Default is .05

**upper_bound** : float
Maximum value of skewness the upper limit can be. Default is .99 confidence limit assuming normality.

**lower_bound** : float
Minimum value of skewness the lower limit can be. Default is .99 confidence level assuming normality.

Returns **Interval** : tuple
Confidence interval for the skewness

**Notes**

If function returns f(a) and f(b) must have different signs, consider expanding lower and upper bounds
DescStatUV.ci_var  

DescStatUV.ci_var(lower_bound=None, upper_bound=None, sig=0.05)  

Returns the confidence interval for the variance.

Parameters  lower_bound : float

The minimum value the lower confidence interval can take. The p-value from

\text{test\_var}(\text{lower\_bound})\ must\ be\ lower\ than\ 1 - \ \text{significance\ level}\.

Default is .99 confidence limit assuming normality

upper_bound : float

The maximum value the upper confidence interval can take. The p-value from

\text{test\_var}(\text{upper\_bound})\ must\ be\ lower\ than\ 1 - \ \text{significance\ level}\.

Default is .99 confidence limit assuming normality

sig : float

The significance level. Default is .05

Returns  Interval : tuple

Confidence interval for the variance

Notes

If the function returns the error \(f(a)\) and \(f(b)\) must have different signs, consider lowering lower_bound and raising upper_bound.

Examples

```python
>>> random_numbers = np.random.standard_normal(100)
>>> el_analysis = sm.emplike.DescStat(random_numbers)
>>> el_analysis.ci_var()

'f(a) and f(b) must have different signs'
>>> el_analysis.ci_var(.5, 2)
```

DescStatUV.plot_contour  

DescStatUV.plot_contour(mu_low, mu_high, var_low, var_high, mu_step, var_step, levs=[0.2, 0.1, 0.05, 0.01, 0.001])  

Returns a plot of the confidence region for a univariate mean and variance.

Parameters  mu_low : float

Lowest value of the mean to plot

mu_high : float

Highest value of the mean to plot

var_low : float

Lowest value of the variance to plot

var_high : float

Highest value of the variance to plot
mu_step : float
Increments to evaluate the mean

var_step : float
Increments to evaluate the mean

levs : list
Which values of significance the contour lines will be drawn. Default is [.2, .1, .05, .01, .001]

Returns fig : matplotlib figure instance
The contour plot

statsmodels.emplike.descriptive.DescStatUV.test_joint_skew_kurt

DescStatUV.test_joint_skew_kurt (skew0, kurt0, return_weights=False)
Returns - 2 x log-likelihood and the p-value for the joint hypothesis test for skewness and kurtosis

Parameters skew0 : float
Skewness value to be tested

kurt0 : float
Kurtosis value to be tested

return_weights : bool
If True, function also returns the weights that maximize the likelihood ratio. Default is False.

Returns test_results : tuple
The log-likelihood ratio and p-value of the joint hypothesis test.

statsmodels.emplike.descriptive.DescStatUV.test_kurt

DescStatUV.test_kurt (kurt0, return_weights=False)
Returns -2 x log-likelihood and the p-value for the hypothesized kurtosis.

Parameters kurt0 : float
Kurtosis value to be tested

return_weights : bool
If True, function also returns the weights that maximize the likelihood ratio. Default is False.

Returns test_results : tuple
The log-likelihood ratio and p-value of kurt0

statsmodels.emplike.descriptive.DescStatUV.test_mean

DescStatUV.test_mean (mu0, return_weights=False)
Returns - 2 x log-likelihood ratio, p-value and weights for a hypothesis test of the mean.
Parameters mu0 : float
    Mean value to be tested

return_weights : bool
    If return_weights is True the function returns the weights of the observations under
    the null hypothesis. Default is False

Returns test_results : tuple
    The log-likelihood ratio and p-value of mu0

statsmodels.emplike.descriptive.DescStatUV.test_skew

DescStatUV.test_skew(skew0, return_weights=False)
    Returns -2 x log-likelihood and p-value for the hypothesized skewness.

Parameters skew0 : float
    Skewness value to be tested

return_weights : bool
    If True, function also returns the weights that maximize the likelihood ratio. De-
    fault is False.

Returns test_results : tuple
    The log-likelihood ratio and p_value of skew0

statsmodels.emplike.descriptive.DescStatUV.test_var

DescStatUV.test_var(sig2_0, return_weights=False)
    Returns -2 x log-likelihood ratio and the p-value for the hypothesized variance

Parameters sig2_0 : float
    Hypothesized variance to be tested

return_weights : bool
    If True, returns the weights that maximize the likelihood of observing sig2_0.
    Default is False

Returns test_results : tuple
    The log-likelihood ratio and the p_value of sig2_0

Examples

>>> random_numbers = np.random.standard_normal(1000)*100
>>> el_analysis = sm.emplike.DescStat(random_numbers)
>>> hyp_test = el_analysis.test_var(9500)
class statsmodels.emplike.descriptive.DescStatMV (endog)
A class for conducting inference on multivariate means and correlation.

Parameters endog : ndarray
Data to be analyzed

Attributes

<table>
<thead>
<tr>
<th>endog</th>
<th>ndarray</th>
<th>Data to be analyzed</th>
</tr>
</thead>
<tbody>
<tr>
<td>nobs</td>
<td>float</td>
<td>Number of observations</td>
</tr>
</tbody>
</table>

Methods

- **ci_corr** ([sig, upper_bound, lower_bound])
  Returns the confidence intervals for the correlation coefficient
  Parameters sig : float
  The significance level. Default is .05
  upper_bound : float
  Maximum value the upper confidence limit can be. Default is 99% confidence limit assuming normality.
  lower_bound : float
  Minimum value the lower confidence limit can be. Default is 99% confidence limit assuming normality.
  Returns interval : tuple
  Confidence interval for the correlation

- **mv_mean_contour** (mu1_low, mu1_upp, mu2_low, ...)
  Creates a confidence region plot for the mean of bivariate data
  Parameters mu1_low : float
  Minimum value of the mean for variable 1
  mu1_upp : float
  Minimum value of the mean for variable 1
  Returns -2 x log likelihood and the p-value

- **test_corr** (corr0[, return_weights])
  Returns -2 x log-likelihood ratio and p-value for the correlation
  Parameters corr0 : float
  The correlation coefficient
  Returns -2 x log-likelihood ratio and p-value for the correlation
Maximum value of the mean for variable 1

**mu2_low** : float

Minimum value of the mean for variable 2

**mu2_upp** : float

Maximum value of the mean for variable 2

**step1** : float

Increment of evaluations for variable 1

**step2** : float

Increment of evaluations for variable 2

**levs** : list

Levels to be drawn on the contour plot. Default = [.2, .1, .05, .01, .001]

**plot_data** : bool

If True, makes a scatter plot of the data on top of the contour plot. Default is False.

**var1_name** : str

Name of variable 1 to be plotted on the x-axis

**var2_name** : str

Name of variable 2 to be plotted on the y-axis

**Notes**

The smaller the step size, the more accurate the intervals will be

If the function returns optimization failed, consider narrowing the boundaries of the plot

**Examples**

```python
>>> two_rvs = np.random.standard_normal((20,2))
>>> el_analysis = sm.empllike.DescStat(two_rvs)
>>> contourp = el_analysis.mv_mean_contour(-2, 2, -2, 2, .1, .1)
>>> contourp.show()
```

**statsmodels.emplike.descriptive.DescStatMV.mv_test_mean**

**DescStatMV.mv_test_mean** *(mu_array, return_weights=False)*

Returns -2 x log likelihood and the p-value for a multivariate hypothesis test of the mean

**Parameters**

- **mu_array** : 1d array
  
  Hypothesized values for the mean. Must have same number of elements as columns in endog

- **return_weights** : bool
  
  If True, returns the weights that maximize the likelihood of mu_array. Default is False.
Returns test_results : tuple
The log-likelihood ratio and p-value for mu_array

statsmodels.emplike.descriptive.DescStatMV.test_corr

DescStatMV.test_corr(corr0, return_weights=0)
Returns -2 x log-likelihood ratio and p-value for the correlation coefficient between 2 variables

Parameters corr0 : float
Hypothesized value to be tested

return_weights : bool
If true, returns the weights that maximize the log-likelihood at the hypothesized value

3.12 Other Models miscmodels

statsmodels.miscmodels contains model classes and that do not yet fit into any other category, or are basic implementations that are not yet polished and will most likely still change. Some of these models were written as examples for the generic maximum likelihood framework, and there will be others that might be based on general method of moments.

The models in this category have been checked for basic cases, but might be more exposed to numerical problems than the complete implementation. For example, count.Poisson has been added using only the generic maximum likelihood framework, the standard errors are based on the numerical evaluation of the Hessian, while discretemod.Poisson uses analytical Gradients and Hessian and will be more precise, especially in cases when there is strong multicollinearity. On the other hand, by subclassing GenericLikelihoodModel, it is easy to add new models, another example can be seen in the zero inflated Poisson model, miscmodels.count.

3.12.1 Count Models count

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PoissonGMLE(endog[, exog, loglike, score, ...])</td>
<td>Maximum Likelihood Estimation of Poisson Model</td>
</tr>
<tr>
<td>PoissonOffsetGMLE(endog[, exog, offset, missing])</td>
<td>Maximum Likelihood Estimation of Poisson Model</td>
</tr>
<tr>
<td>PoissonZiGMLE(endog[, exog, offset, missing])</td>
<td>Maximum Likelihood Estimation of Poisson Model</td>
</tr>
</tbody>
</table>

statsmodels.miscmodels.count.PoissonGMLE

class statsmodels.miscmodels.count.PoissonGMLE(endog, exog=None, loglike=None, score=None, hessian=None, missing='none', extra_params_names=None, **kwds)

Maximum Likelihood Estimation of Poisson Model

This is an example for generic MLE which has the same statistical model as discretemod.Poisson.

Except for defining the negative log-likelihood method, all methods and results are generic. Gradients and Hessian and all resulting statistics are based on numerical differentiation.

Methods
**statsmodels.miscmodels.count.PoissonGMLE.expandparams**

 expand to full parameter array when some parameters are fixed

**Parameters**

- **params**: array
  - reduced parameter array

**Returns**

- **paramsfull**: array
  - expanded parameter array where fixed parameters are included

**Notes**

Calling this requires that self.fixed_params and self.fixed_paramsmask are defined.

**developer notes:**

This can be used in the log-likelihood to ...

this could also be replaced by a more general parameter transformation.

**statsmodels.miscmodels.count.PoissonGMLE.fit**

**PoissonGMLE.fit**(start_params=None, method='nm', maxiter=500, full_output=1, disp=1, callback=None, retall=0, **kwargs)

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

**statsmodels.miscmodels.count.PoissonGMLE.from_formula**

**classmethod PoissonGMLE.from_formula**(formula, data[, subset], *args, **kwargs)

Create a Model from a formula and dataframe.

**Parameters**

- **formula**: str or generic Formula object
  - The formula specifying the model
data : array-like
    The data for the model. See Notes.
subset : array-like
    An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame
args : extra arguments
    These are passed to the model
kwargs : extra keyword arguments
    These are passed to the model.

Returns model : Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.
PoissonGMLE\texttt{.nloglike}(\texttt{params})

PoissonGMLE\texttt{.nloglikeobs}(\texttt{params})

\texttt{PoissonGMLE.nloglikeobs(params)}

Loglikelihood of Poisson model

\textbf{Parameters} \texttt{params} : array-like

The parameters of the model.

\textbf{Returns} The log likelihood of the model evaluated at ‘params’ :

\[
\ln L = \sum_{i=1}^{n} \left[ -\lambda_i + y_i x'_i \beta - \ln y_i! \right]
\]

PoissonGMLE\texttt{.predict}(\texttt{params, exog=None, *args, **kwargs})

After a model has been fit predict returns the fitted values.

This is a placeholder intended to be overwritten by individual models.

PoissonGMLE\texttt{.predict_distribution}(\texttt{exog})

\texttt{return frozen scipy.stats distribution with mu at estimated prediction}

PoissonGMLE\texttt{.reduceparams}(\texttt{params})

PoissonGMLE\texttt{.score}(\texttt{params})

Gradient of log-likelihood evaluated at params

\textbf{Attributes}

\begin{verbatim}
endog_names
exog_names
\end{verbatim}
Maximum Likelihood Estimation of Poisson Model

This is an example for generic MLE which has the same statistical model as discretemod.Poisson but adds offset. Except for defining the negative log-likelihood method, all methods and results are generic. Gradients and Hessian and all resulting statistics are based on numerical differentiation.

Methods

*expandparams*(params) expand to full parameter array when some parameters are fixed

*fit*(start_params, method, maxiter, ...) Fit the model using maximum likelihood.

*from_formula*(formula, data[, subset]) Create a Model from a formula and dataframe.

*hessian*(params) Hessian of log-likelihood evaluated at params

*information*(params) Fisher information matrix of model

*initialize*()  

*jac*(params, **kwds) Jacobian/Gradient of log-likelihood evaluated at params for each

*loglike*(params) Loglikelihood of Poisson model

*loglikeobs*(params) After a model has been fit predict returns the fitted values.

*nloglike*(params)  

*nloglikeobs*(params)  

*predict*(params[, exog])  

*reduceparams*(params)  

*score*(params) Gradient of log-likelihood evaluated at params

Notes

Calling this requires that self.fixed_params and self.fixed_paramsmask are defined.

**developer notes:**

This can be used in the log-likelihood to ...

could also be replaced by a more general parameter transformation.
**statsmodels.misccmodels.count.PoissonOffsetGMLE.fit**

PoissonOffsetGMLE.fit(start_params=None, method='nm', maxiter=500, full_output=1, disp=1, callback=None, retall=0, **kwargs)

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit

**statsmodels.misccmodels.count.PoissonOffsetGMLE.from_formula**

classmethod PoissonOffsetGMLE.from_formula(formula, data, subset=None, *args, **kwargs)

Create a Model from a formula and dataframe.

Parameters formula : str or generic Formula object

The formula specifying the model
data : array-like

The data for the model. See Notes.
subset : array-like

An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

args : extra arguments

These are passed to the model
kwargs : extra keyword arguments

These are passed to the model.

Returns model : Model instance

Notes
data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**statsmodels.misccmodels.count.PoissonOffsetGMLE.hessian**

PoissonOffsetGMLE.hessian(params)

Hessian of log-likelihood evaluated at params

**statsmodels.misccmodels.count.PoissonOffsetGMLE.information**

PoissonOffsetGMLE.information(params)

Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

3.12. Other Models misccmodels
statsmodels.miscmodels.count.PoissonOffsetGMLE.initialize

PoissonOffsetGMLE.initialize()

statsmodels.miscmodels.count.PoissonOffsetGMLE.jac

PoissonOffsetGMLE.jac(params, **kwds)

Jacobian/Gradient of log-likelihood evaluated at params for each observation.

statsmodels.miscmodels.count.PoissonOffsetGMLE.loglike

PoissonOffsetGMLE.loglike(params)

statsmodels.miscmodels.count.PoissonOffsetGMLE.loglikeobs

PoissonOffsetGMLE.loglikeobs(params)

statsmodels.miscmodels.count.PoissonOffsetGMLE.nloglike

PoissonOffsetGMLE.nloglike(params)

statsmodels.miscmodels.count.PoissonOffsetGMLE.nloglikeobs

PoissonOffsetGMLE.nloglikeobs(params)

Loglikelihood of Poisson model

Parameters params : array-like

The parameters of the model.

Returns The log likelihood of the model evaluated at ‘params’ :

Notes

\[ \ln L = \sum_{i=1}^{n} \left[ -\lambda_i + y_i x_i' \beta - \ln y_i! \right] \]

statsmodels.miscmodels.count.PoissonOffsetGMLE.predict

PoissonOffsetGMLE.predict(params, exog=None, *args, **kwargs)

After a model has been fit predict returns the fitted values.

This is a placeholder intended to be overwritten by individual models.
.. _count.poisson_offset_gmle:

.. currentmodule:: statsmodels.miscmodels.count

.. automodule:: PoissonOffsetGMLE
   :members: reduceparams

.. automodule:: PoissonOffsetGMLE
   :members: score

.. currentmodule:: statsmodels.miscmodels.count

Attributes

- `endog_names`
- `exog_names`

.. currentmodule:: statsmodels.miscmodels.count

.. class:: PoissonZiGMLE
   :module: statsmodels.miscmodels.count
   :members: expandparams

Maximum Likelihood Estimation of Poisson Model

This is an example for generic MLE which has the same statistical model as discretemod.Poisson but adds offset and zero-inflation.

Except for defining the negative log-likelihood method, all methods and results are generic. Gradients and Hessian and all resulting statistics are based on numerical differentiation.

There are numerical problems if there is no zero-inflation.

Methods

- `expandparams(params)`
  expand to full parameter array when some parameters are fixed
- `fit([start_params, method, maxiter, ...])` Fit the model using maximum likelihood.
- `from_formula(formula, data[, subset])` Create a Model from a formula and dataframe.
- `hessian(params)` Hessian of log-likelihood evaluated at params
- `information(params)` Fisher information matrix of model
- `initialize()`
- `jac(params, **kwds)` Jacobian/Gradient of log-likelihood evaluated at params for each
- `loglike(params)`
- `loglikeobs(params)`
- `nloglike(params)`
- `nloglikeobs(params)`
- `predict(params[, exog])` After a model has been fit predict returns the fitted values.
- `reduceparams(params)` Gradient of log-likelihood evaluated at params

.. currentmodule:: statsmodels.miscmodels.count

.. automodule:: PoissonZiGMLE
   :members: expandparams

- `PoissonZiGMLE.expandparams(params)`
  expand to full parameter array when some parameters are fixed
Parameters params: array
    reduced parameter array

Returns paramsfull: array
    expanded parameter array where fixed parameters are included

Notes

Calling this requires that self.fixed_params and self.fixed_paramsmask are defined.

developer notes:
This can be used in the log-likelihood to ...
this could also be replaced by a more general parameter transformation.

statsmodels.miscmodels.count.PoissonZiGMLE.fit

PoissonZiGMLE.fit(start_params=None, method='nm', maxiter=500, full_output=1, disp=1, callback=None, retall=0, **kwargs)
    Fit the model using maximum likelihood.
    The rest of the docstring is from statsmodels.LikelihoodModel.fit

statsmodels.miscmodels.count.PoissonZiGMLE.from_formula

classmethod PoissonZiGMLE.from_formula(formula, data, subset=None, *args, **kwargs)
    Create a Model from a formula and dataframe.

Parameters formula: str or generic Formula object
    The formula specifying the model

data: array-like
    The data for the model. See Notes.

subset: array-like
    An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a pandas.DataFrame

args: extra arguments
    These are passed to the model

kwargs: extra keyword arguments
    These are passed to the model.

Returns model: Model instance

Notes

data must define __getitem__ with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.
PoissonZiGMLE.hessian(params)
   Hessian of log-likelihood evaluated at params

PoissonZiGMLE.information(params)
   Fisher information matrix of model
   Returns -Hessian of loglike evaluated at params.

PoissonZiGMLE.initialize()

PoissonZiGMLE.jac(params, **kwds)
   Jacobian/Gradient of log-likelihood evaluated at params for each observation.

PoissonZiGMLE.loglike(params)

PoissonZiGMLE.loglikeobs(params)

PoissonZiGMLE.nloglike(params)

PoissonZiGMLE.nloglikeobs(params)
   Loglikelihood of Poisson model
   Parameters params : array-like
      The parameters of the model.
   Returns The log likelihood of the model evaluated at 'params' :

3.12. Other Models miscmodels
\[ \ln L = \sum_{i=1}^{n} \left[ -\lambda_i + y_i x_i^\prime \beta - \ln y_i! \right] \]

\section*{Notes}

\begin{verbatim}
statsmodels.miscmodels.count.PoissonZiGMLE.predict

PoissonZiGMLE.predict (params, exog=None, *args, **kwargs)

After a model has been fit predict returns the fitted values.

This is a placeholder intended to be overwritten by individual models.

statsmodels.miscmodels.count.PoissonZiGMLE.reduceparams

PoissonZiGMLE.reduceparams (params)

statsmodels.miscmodels.count.PoissonZiGMLE.score

PoissonZiGMLE.score (params)

Gradient of log-likelihood evaluated at params
\end{verbatim}

\section*{Attributes}

\begin{verbatim}
    endog_names
    exog_names
\end{verbatim}

\subsection*{3.12.2 Linear Model with t-distributed errors}

This is a class that shows that a new model can be defined by only specifying the method for the loglikelihood. All result statistics are inherited from the generic likelihood model and result classes. The results have been checked against R for a simple case.

\begin{verbatim}
TLinearModel(endog[, exog, loglike, score, ...]) Maximum Likelihood Estimation of Linear Model with t-distributed errors
\end{verbatim}

\begin{verbatim}
class statsmodels.miscmodels.tmodel.TLinearModel (endog, exog=None, loglike=None, score=None, hessian=None, missing='none', extra_params_names=None, **kwds)

Maximum Likelihood Estimation of Linear Model with t-distributed errors

This is an example for generic MLE.

Except for defining the negative log-likelihood method, all methods and results are generic. Gradients and
Hessian and all resulting statistics are based on numerical differentiation.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expandparams(params)</code></td>
<td>expand to full parameter array when some parameters are fixed</td>
</tr>
<tr>
<td><code>fit([start_params, method, maxiter, ...])</code></td>
<td>Fit the model using maximum likelihood.</td>
</tr>
<tr>
<td><code>from_formula(formula, data[, subset])</code></td>
<td>Create a Model from a formula and dataframe.</td>
</tr>
<tr>
<td><code>hessian(params)</code></td>
<td>Hessian of log-likelihood evaluated at params</td>
</tr>
<tr>
<td><code>information(params)</code></td>
<td>Fisher information matrix of model</td>
</tr>
<tr>
<td><code>jac(params, **kwds)</code></td>
<td>Jacobian/Gradient of log-likelihood evaluated at params for each</td>
</tr>
<tr>
<td><code>loglike(params)</code></td>
<td>Loglikelihood of linear model with t distributed errors.</td>
</tr>
<tr>
<td><code>loglikeobs(params)</code></td>
<td></td>
</tr>
<tr>
<td><code>nloglike(params)</code></td>
<td></td>
</tr>
<tr>
<td><code>nloglikeobs(params)</code></td>
<td></td>
</tr>
<tr>
<td><code>predict([params, exog])</code></td>
<td></td>
</tr>
<tr>
<td><code>reduceparams(params)</code></td>
<td></td>
</tr>
<tr>
<td><code>score(params)</code></td>
<td>Gradient of log-likelihood evaluated at params</td>
</tr>
</tbody>
</table>

**statsmodels.miscmodels.tmodel.TLinearModel.expandparams**

`TLinearModel.expandparams(params)`

expand to full parameter array when some parameters are fixed

**Parameters**

- **params** : array
  - reduced parameter array

**Returns**

- **paramsfull** : array
  - expanded parameter array where fixed parameters are included

**Notes**

Calling this requires that self.fixed_params and self.fixed_paramsmask are defined.

developer notes:

This can be used in the log-likelihood to ...

this could also be replaced by a more general parameter transformation.

**statsmodels.miscmodels.tmodel.TLinearModel.fit**

`TLinearModel.fit(start_params=None, method='nm', maxiter=500, full_output=1, disp=1, callback=None, retall=0, **kwags)`

Fit the model using maximum likelihood.

The rest of the docstring is from statsmodels.LikelihoodModel.fit
**statsmodels.misceconomics.tmodel.TLinearModel.from_formula**

**classmethod** `TLinearModel.from_formula(formula, data, subset=None, *args, **kwargs)`

Create a Model from a formula and dataframe.

**Parameters**
- **formula** : str or generic Formula object
  - The formula specifying the model
- **data** : array-like
  - The data for the model. See Notes.
- **subset** : array-like
  - An array-like object of booleans, integers, or index values that indicate the subset of df to use in the model. Assumes df is a `pandas.DataFrame`
- **args** : extra arguments
  - These are passed to the model
- **kwargs** : extra keyword arguments
  - These are passed to the model.

**Returns**
- **model** : Model instance

**Notes**

data must define `__getitem__` with the keys in the formula terms args and kwargs are passed on to the model instantiation. E.g., a numpy structured or rec array, a dictionary, or a pandas DataFrame.

**statsmodels.misceconomics.tmodel.TLinearModel.hessian**

`TLinearModel.hessian(params)`

Hessian of log-likelihood evaluated at params

**statsmodels.misceconomics.tmodel.TLinearModel.information**

`TLinearModel.information(params)`

Fisher information matrix of model

Returns -Hessian of loglike evaluated at params.

**statsmodels.misceconomics.tmodel.TLinearModel.initialize**

`TLinearModel.initialize()`

**statsmodels.misceconomics.tmodel.TLinearModel.jac**

`TLinearModel.jac(params, **kwds)`

Jacobian/Gradient of log-likelihood evaluated at params for each observation.
Loglikelihood of linear model with t distributed errors.

**Parameters** `params`: array

The parameters of the model. The last 2 parameters are degrees of freedom and scale.

**Returns** `loglike`: array, (nobs,)

The log likelihood of the model evaluated at `params` for each observation defined by `self.endog` and `self.exog`.

**Notes**

\[
\ln L = \sum_{i=1}^{n} [-\lambda_i + y_i x_i' \beta - \ln y_i !]
\]

The t distribution is the standard t distribution and not a standardized t distribution, which means that the scale parameter is not equal to the standard deviation.

self.fixed_params and self.expandparams can be used to fix some parameters. (I doubt this has been tested in this model.)
statsmodels.miscmodels.tmodel.TLinearModel.score

\texttt{TLinearModel.score(params)}

Gradient of log-likelihood evaluated at params

Attributes

\begin{itemize}
\item \texttt{endog\_names}
\item \texttt{exog\_names}
\end{itemize}

3.13 Distributions

This section collects various additional functions and methods for statistical distributions.

3.13.1 Empirical Distributions

\begin{itemize}
\item \texttt{ECDF(x[, side])} Return the Empirical CDF of an array as a step function.
\item \texttt{StepFunction(x, y[, ival, sorted, side])} A basic step function.
\end{itemize}

\texttt{statsmodels.distributions.empirical_distribution.ECDF}

\texttt{class statsmodels.distributions.empirical_distribution.ECDF (x, side=\textquoteleft right\textquoteright )} Return the Empirical CDF of an array as a step function.

\textbf{Parameters}

\begin{itemize}
\item \texttt{x} : array-like
  Observations
\item \texttt{side} : \{'left’, ‘right’}, optional
  Default is ‘right’. Defines the shape of the intervals constituting the steps. ‘right’ correspond to \([a, b)\) intervals and ‘left’ to \((a, b]\).
\end{itemize}

\textbf{Returns}

Empirical CDF as a step function.

\textbf{Examples}

\begin{verbatim}
>>> import numpy as np
>>> from statsmodels.distributions.empirical_distribution import ECDF
>>> >>> ecdf = ECDF([3, 3, 1, 4])
>>> >>> ecdf([3, 55, 0.5, 1.5])
array([ 0.75, 1. , 0. , 0.25])
\end{verbatim}

\textbf{Methods}
class statsmodels.distributions.empirical_distribution.StepFunction(x, y, ival=0.0, sorted=False, side='left')

A basic step function.
Values at the ends are handled in the simplest way possible: everything to the left of x[0] is set to ival; everything to the right of x[-1] is set to y[-1].

Parameters

x : array-like
  Values at which the steps are defined.

y : array-like
  Values of the step function at the x points.

ival : float
  ival is the value given to the values to the left of x[0]. Default is 0.

sorted : bool
  Default is False.

side : {'left', 'right'}, optional
  Default is 'left'. Defines the shape of the intervals constituting the steps. ‘right’ correspond to [a, b) intervals and ‘left’ to (a, b).

Examples

>>> import numpy as np
>>> from statsmodels.distributions.empirical_distribution import StepFunction
>>> x = np.arange(20)
>>> y = np.arange(20)
>>> f = StepFunction(x, y)
>>> print(f(3.2))
3.0
>>> print(f([[3.2,4.5],[24,-3.1]]))
[[ 3.  4.]
 [19.  0.]]
>>> f2 = StepFunction(x, y, side='right')
>>> print(f(3.0))
2.0
>>> print(f2(3.0))
3.0

Methods

3.13.2 Distribution Extras

Skew Distributions
SkewNorm_gen() univariate Skew-Normal distribution of Azzalini
SkewNorm2_gen([momtype, a, b, xtol, ...]) univariate Skew-Normal distribution of Azzalini
ACSkewT_gen() univariate Skew-T distribution of Azzalini
skewnorm2 univariate Skew-Normal distribution of Azzalini

statsmodels.sandbox.distributions.extras.SkewNorm_gen

class statsmodels.sandbox.distributions.extras.SkewNorm_gen univariate Skew-Normal distribution of Azzalini
class follows scipy.stats.distributions pattern but with __init__

Methods

cdf(x, *args, **kwds) Cumulative distribution function of the given RV.
entropy(*args, **kwds) Differential entropy of the RV.
est_loc_scale(*args, **kwds) est_loc_scale is deprecated!
expect([func, args, loc, scale, lb, ub, ...]) Calculate expected value of a function with respect to the distribution
fit(data, *args, **kwds) Estimate MLEs for shape, location, and scale parameters from data.
fit_loc_scale(data, *args) Estimate loc and scale parameters from data using 1st and 2nd moments.
freeze(*args, **kwds) Freeze the distribution for the given arguments.
interval(alpha, *args, **kwds) Confidence interval with equal areas around the median.
isf(q, *args, **kwds) Inverse survival function at q of the given RV.
logcdf(x, *args, **kwds) Log of the cumulative distribution function at x of the given RV.
logpdf(x, *args, **kwds) Log of the probability density function at x of the given RV.
logsf(x, *args, **kwds) Log of the survival function of the given RV.
mean(*args, **kwds) Mean of the distribution
median(*args, **kwds) Median of the distribution.
moment(n, *args, **kwds) n'th order non-central moment of distribution.
nnlf(theta, x) Return negative loglikelihood function..
pdf(x, *args, **kwds) Probability density function at x of the given RV.
ppf(q, *args, **kwds) Percent point function (inverse of cdf) at q of the given RV.
rvs(*args, **kwds) Random variates of given type.
sf(x, *args, **kwds) Survival function (1-cdf) at x of the given RV.
stats(*args, **kwds) Some statistics of the given RV
std(*args, **kwds) Standard deviation of the distribution.
var(*args, **kwds) Variance of the distribution

statsmodels.sandbox.distributions.extras.SkewNorm_gen.cdf

SkewNorm_gen.cdf (x, *args, **kwds)
Cumulative distribution function of the given RV.

Parameters x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
...scale parameter (default=1)

Returns cdf : ndarray
Cumulative distribution function evaluated at x

statsmodels.sandbox.distributions.extras.SkewNorm_gen.entropy

SkewNorm_gen.entropy(*args, **kwds)
Differential entropy of the RV.

Parameters arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
Location parameter (default=0).
scale : array_like, optional
Scale parameter (default=1).

statsmodels.sandbox.distributions.extras.SkewNorm_gen.est_loc_scale

SkewNorm_gen.est_loc_scale(*args, **kwds)
est_loc_scale is deprecated!
This function is deprecated, use self.fit_loc_scale(data) instead.

statsmodels.sandbox.distributions.extras.SkewNorm_gen.expect

SkewNorm_gen.expect(func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)
Calculate expected value of a function with respect to the distribution
The expected value of a function \( f(x) \) with respect to a distribution dist is defined as:

\[
E[x] = \int_{lb}^{ub} \int_{dist.pdf(x)} f(x)
\]

Parameters func : callable, optional
Function for which integral is calculated. Takes only one argument. The default is the identity mapping \( f(x) = x \).
args : tuple, optional
Argument (parameters) of the distribution.
lb, ub : scalar, optional
Lower and upper bound for integration. default is set to the support of the distribution.
conditional : bool, optional

If True, the integral is corrected by the conditional probability of the integration
interval. The return value is the expectation of the function, conditional on being
in the given interval. Default is False.

Additional keyword arguments are passed to the integration routine.

Returns expect : float

The calculated expected value.

Notes

The integration behavior of this function is inherited from integrate.quad.

statsmodels.sandbox.distributions.extras.SkewNorm_gen.fit

SkewNorm_gen.fit(data, *args, **kwds)

Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input argu-
ments; for any arguments not provided with starting estimates, self._fitstart(data) is called to
generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments f0, f1, ..., fn
(for shape parameters) and floc and fscale (for location and scale parameters, respectively).

Parameters data : array_like

Data to use in calculating the MLEs.

args : floats, optional

Starting value(s) for any shape-characterizing arguments (those not provided will
be determined by a call to _fitstart(data)). No default value.

kwds : floats, optional

Starting values for the location and scale parameters; no default. Special keyword
arguments are recognized as holding certain parameters fixed:

f0...fn : hold respective shape parameters fixed.
floc : hold location parameter fixed to specified value.
fscale : hold scale parameter fixed to specified value.

optimizer [The optimizer to use. The optimizer must take func.] and starting
position as the first two arguments, plus args (for extra arguments to pass to the
function to be optimized) and disp=0 to suppress output as keyword arguments.

Returns shape, loc, scale : tuple of floats

MLEs for any shape statistics, followed by those for location and scale.
Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.fit_loc_scale**

`SkewNorm_gen.fit_loc_scale(data, *args)`
Estimate loc and scale parameters from data using 1st and 2nd moments.

**Parameters**
- `data` : array_like
  Data to fit.
- `arg1, arg2, arg3,...` : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).

**Returns**
- `Lhat` : float
  Estimated location parameter for the data.
- `Shat` : float
  Estimated scale parameter for the data.

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.freeze**

`SkewNorm_gen.freeze(*args, **kwds)`
Freeze the distribution for the given arguments.

**Parameters**
- `arg1, arg2, arg3,...` : array_like
  The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include `loc` and `scale`.

**Returns**
- `rv_frozen` : rv_frozen instance
  The frozen distribution.

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.interval**

`SkewNorm_gen.interval(alpha, *args, **kwds)`
Confidence interval with equal areas around the median.

**Parameters**
- `alpha` : array_like of float
  Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].
- `arg1, arg2,...` : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- `loc` : array_like, optional
  location parameter, Default is 0.
scale : array_like, optional
    scale parameter, Default is 1.

**Returns**

a, b : ndarray of float
    end-points of range that contain 100 * alpha % of the rv's possible values.

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.isf**

SkewNorm_gen.isf(q, *args, **kwds)
Inverse survival function at q of the given RV.

**Parameters**

q : array_like
    upper tail probability

args, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

**Returns**

x : ndarray or scalar
    Quantile corresponding to the upper tail probability q.

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.logcdf**

SkewNorm_gen.logcdf(x, *args, **kwds)
Log of the cumulative distribution function at x of the given RV.

**Parameters**

x : array_like
    quantiles

args, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

**Returns**

logcdf : array_like
    Log of the cumulative distribution function evaluated at x
SkewNorm_gen.logpdf(x, *args, **kwds)

    Log of the probability density function at x of the given RV.
    This uses a more numerically accurate calculation if available.

    Parameters
    x : array_like
        quantiles
    arg1, arg2, arg3,... : array_like
        The shape parameter(s) for the distribution (see docstring of the instance object
        for more information)
    loc : array_like, optional
        location parameter (default=0)
    scale : array_like, optional
        scale parameter (default=1)

    Returns
    logpdf : array_like
        Log of the probability density function evaluated at x

SkewNorm_gen.logsf(x, *args, **kwds)

    Log of the survival function of the given RV.

    Returns the log of the “survival function,” defined as (1 - cdf), evaluated at x.

    Parameters
    x : array_like
        quantiles
    arg1, arg2, arg3,... : array_like
        The shape parameter(s) for the distribution (see docstring of the instance object
        for more information)
    loc : array_like, optional
        location parameter (default=0)
    scale : array_like, optional
        scale parameter (default=1)

    Returns
    logsf : ndarray
        Log of the survival function evaluated at x.

SkewNorm_gen.mean(*args, **kwds)

    Mean of the distribution

    Parameters
    arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object
for more information)

**loc**: array_like, optional
location parameter (default=0)

**scale**: array_like, optional
scale parameter (default=1)

**Returns**

**mean**: float
the mean of the distribution

---

`statsmodels.sandbox.distributions.extras.SkewNorm_gen.median`

`SkewNorm_gen.median(*args, **kwds)`
Median of the distribution.

**Parameters**

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object
for more information)

**loc**: array_like, optional
Location parameter, Default is 0.

**scale**: array_like, optional
Scale parameter, Default is 1.

**Returns**

**median**: float
The median of the distribution.

**See also:**

`stats.distributions.rv_discrete.ppf` Inverse of the CDF

---

`statsmodels.sandbox.distributions.extras.SkewNorm_gen.moment`

`SkewNorm_gen.moment(n, *args, **kwds)`
n’th order non-central moment of distribution.

**Parameters**

**n**: int, n>=1
Order of moment.

arg1, arg2, arg3,... : float
The shape parameter(s) for the distribution (see docstring of the instance object
for more information).

**kwds** : keyword arguments, optional
These can include “loc” and “scale”, as well as other keyword arguments relevant
for a given distribution.
**statsmodels.sandbox.distributions.extras.SkewNorm_gen.nnlf**

`SkewNorm_gen.nnlf(theta, x)`

Return negative loglikelihood function

**Notes**

This is `-sum(log pdf(x, theta), axis=0)` where theta are the parameters (including loc and scale).

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.pdf**

`SkewNorm_gen.pdf(x, *args, **kwds)`

Probability density function at x of the given RV.

**Parameters**

- `x`: array_like
  - quantiles
- `arg1, arg2, arg3,...`: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- `loc`: array_like, optional
  - location parameter (default=0)
- `scale`: array_like, optional
  - scale parameter (default=1)

**Returns**

- `pdf`: ndarray
  - Probability density function evaluated at x

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.ppf**

`SkewNorm_gen.ppf(q, *args, **kwds)`

Percent point function (inverse of cdf) at q of the given RV.

**Parameters**

- `q`: array_like
  - lower tail probability
- `arg1, arg2, arg3,...`: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- `loc`: array_like, optional
  - location parameter (default=0)
- `scale`: array_like, optional
  - scale parameter (default=1)

**Returns**

- `x`: array_like
  - quantile corresponding to the lower tail probability q.
**statsmodels.sandbox.distributions.extras.SkewNorm_gen.rvs**

`SkewNorm_gen.rvs(*args, **kwds)`

Random variates of given type.

**Parameters**
- `arg1, arg2, arg3,...` : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- `loc` : array_like, optional
  - Location parameter (default=0).
- `scale` : array_like, optional
  - Scale parameter (default=1).
- `size` : int or tuple of ints, optional
  - Defining number of random variates (default=1).

**Returns**
- `rvs` : ndarray or scalar
  - Random variates of given `size`.

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.sf**

`SkewNorm_gen.sf(x, *args, **kwds)`

Survival function (1-cdf) at `x` of the given RV.

**Parameters**
- `x` : array_like
  - quantiles
- `arg1, arg2, arg3,...` : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- `loc` : array_like, optional
  - location parameter (default=0)
- `scale` : array_like, optional
  - scale parameter (default=1)

**Returns**
- `sf` : array_like
  - Survival function evaluated at `x`

**statsmodels.sandbox.distributions.extras.SkewNorm_gen.stats**

`SkewNorm_gen.stats(*args, **kwds)`

Some statistics of the given RV

**Parameters**
- `arg1, arg2, arg3,...` : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- `loc` : array_like, optional
location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

moments : str, optional
    composed of letters ['mvsk'] defining which moments to compute: ‘m’ = mean, ‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default=’mv’)

Returns stats : sequence
    of requested moments.

statsmodels.sandbox.distributions.extras.SkewNorm_gen.std

SkewNorm_gen.std(*args, **kwds)
    Standard deviation of the distribution.

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns std : float
    standard deviation of the distribution

statsmodels.sandbox.distributions.extras.SkewNorm_gen.var

SkewNorm_gen.var(*args, **kwds)
    Variance of the distribution

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns var : float
    the variance of the distribution
statsmodels.sandbox.distributions.extras.SkewNorm2_gen

class statsmodels.sandbox.distributions.extras.SkewNorm2_gen (momtype=1, a=None, b=None, xtol=1e-14, badvalue=None, name=None, longname=None, shapes=None, extradoc=None)

univariate Skew-Normal distribution of Azzalini
class follows scipy.stats.distributions pattern

Methods

- cdf(x, *args, **kwds): Cumulative distribution function of the given RV.
- entropy(*args, **kwds): Differential entropy of the RV.
- est_loc_scale(*args, **kwds): est_loc_scale is deprecated!
- expect([func, args, loc, scale, lb, ub, ...]): Calculate expected value of a function with respect to the distribution
- fit(data, *args, **kwds): Return MLEs for shape, location, and scale parameters from data.
- fit_loc_scale(data, *args): Estimate loc and scale parameters from data using 1st and 2nd moments.
- freeze(*args, **kwds): Freeze the distribution for the given arguments.
- interval(alpha, *args, **kwds): Confidence interval with equal areas around the median.
- isf(q, *args, **kwds): Inverse survival function at q of the given RV.
- logcdf(x, *args, **kwds): Log of the cumulative distribution function at x of the given RV.
- logpdf(x, *args, **kwds): Log of the probability density function at x of the given RV.
- logsf(x, *args, **kwds): Log of the survival function of the given RV.
- mean(*args, **kwds): Mean of the distribution
- median(*args, **kwds): Median of the distribution.
- moment(n, *args, **kwds): n’th order non-central moment of distribution.
- nnlf(theta, x): Return negative loglikelihood function..
- pdf(x, *args, **kwds): Probability density function at x of the given RV.
- ppf(q, *args, **kwds): Percent point function (inverse of cdf) at q of the given RV.
- rvs(*args, **kwds): Random variates of given type.
- sf(x, *args, **kwds): Survival function (1-cdf) at x of the given RV.
- stats(*args, **kwds): Some statistics of the given RV
- std(*args, **kwds): Standard deviation of the distribution.
- var(*args, **kwds): Variance of the distribution

statsmodels.sandbox.distributions.extras.SkewNorm2_gen.cdf

SkewNorm2_gen.cdf (x, *args, **kwds)
Cumulative distribution function of the given RV.

Parameters x : array_like
quantiles

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)

**scale** : array_like, optional

scale parameter (default=1)

**Returns**

cdf : ndarray

Cumulative distribution function evaluated at \( x \)

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.entropy
```

*args, **kwds

Differential entropy of the RV.

**Parameters**

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

Location parameter (default=0).

scale : array_like, optional

Scale parameter (default=1).

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.est_loc_scale
```

*args, **kwds

`est_loc_scale` is deprecated!

This function is deprecated, use `self.fit_loc_scale(data)` instead.

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.expect
```

func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds

Calculate expected value of a function with respect to the distribution

The expected value of a function \( f(x) \) with respect to a distribution \( dist \) is defined as:

\[
\mathbb{E}[x] = \int f(x) \cdot dist.pdf(x)
\]

ubound

lbound

**Parameters**

func : callable, optional

Function for which integral is calculated. Takes only one argument. The default is the identity mapping \( f(x) = x \).

args : tuple, optional

Argument (parameters) of the distribution.

lb, ub : scalar, optional

Lower and upper bound for integration. default is set to the support of the distribution.
**conditional**: bool, optional

If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.

*Additional keyword arguments are passed to the integration routine.*

**Returns**

- **expect**: float
  
The calculated expected value.

**Notes**

The integration behavior of this function is inherited from `integrate.quad`.

---

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.fit**

*SkewNorm2_gen.fit*(data, *args, **kwds)*

Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, `self._fitstart(data)` is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments `f0, f1, ..., fn` (for shape parameters) and `floc` and `fscale` (for location and scale parameters, respectively).

**Parameters**

- **data**: array_like
  
  Data to use in calculating the MLEs.

- **args**: floats, optional
  
  Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to `self._fitstart(data)`). No default value.

- **kwds**: floats, optional
  
  Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:

  - `f0...fn`: hold respective shape parameters fixed.
  - `floc`: hold location parameter fixed to specified value.
  - `fscale`: hold scale parameter fixed to specified value.

- **optimizer** [The optimizer to use. The optimizer must take func.] and starting position as the first two arguments, plus args (for extra arguments to pass to the function to be optimized) and disp=0 to suppress output as keyword arguments.

**Returns**

- **shape, loc, scale**: tuple of floats
  
  MLEs for any shape statistics, followed by those for location and scale.
Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.fit_loc_scale

SkewNorm2_gen.fit_loc_scale(data, *args)
 Estimate loc and scale parameters from data using 1st and 2nd moments.

Parameters
  data : array_like
    Data to fit.
  arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information).

Returns
  Lhat : float
    Estimated location parameter for the data.
  Shat : float
    Estimated scale parameter for the data.
```

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.freeze

SkewNorm2_gen.freeze(*args, **kwds)
 Freeze the distribution for the given arguments.

Parameters
  arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include loc and scale.

Returns
  rv_frozen : rv_frozen instance
    The frozen distribution.
```

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.interval

SkewNorm2_gen.interval(alpha, *args, **kwds)
 Confidence interval with equal areas around the median.

Parameters
  alpha : array_like of float
    Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].
  arg1, arg2, ... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information).
  loc : array_like, optional
    location parameter, Default is 0.
```
scale : array_like, optional
scale parameter, Default is 1.

Returns a, b : ndarray of float
end-points of range that contain $100 \times \alpha \%$ of the rv's possible values.

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.isf**

SkewNorm2_gen.isf(q, *args, **kwds)
Inverse survival function at q of the given RV.

Parameters q : array_like
upper tail probability

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
location parameter (default=0)

scale : array_like, optional
scale parameter (default=1)

Returns x : ndarray or scalar
Quantile corresponding to the upper tail probability q.

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.logcdf**

SkewNorm2_gen.logcdf(x, *args, **kwds)
Log of the cumulative distribution function at x of the given RV.

Parameters x : array_like
quantiles

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
location parameter (default=0)

scale : array_like, optional
scale parameter (default=1)

Returns logcdf : array_like
Log of the cumulative distribution function evaluated at x
**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.logpdf**

SkewNorm2_gen.logpdf(x, *args, **kwds)

Log of the probability density function at x of the given RV.

This uses a more numerically accurate calculation if available.

**Parameters**
- x : array_like
  - quantiles
- *arg1, arg2, arg3,...*: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- loc : array_like, optional
  - location parameter (default=0)
- scale : array_like, optional
  - scale parameter (default=1)

**Returns**
- logpdf : array_like
  - Log of the probability density function evaluated at x

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.logsf**

SkewNorm2_gen.logsf(x, *args, **kwds)

Log of the survival function of the given RV.

Returns the log of the “survival function,” defined as \((1 - cdf)\), evaluated at x.

**Parameters**
- x : array_like
  - quantiles
- *arg1, arg2, arg3,...*: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- loc : array_like, optional
  - location parameter (default=0)
- scale : array_like, optional
  - scale parameter (default=1)

**Returns**
- logsf : ndarray
  - Log of the survival function evaluated at x.

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.mean**

SkewNorm2_gen.mean(*args, **kwds)

Mean of the distribution

**Parameters**
- *arg1, arg2, arg3,...* : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc**: array_like, optional
- Location parameter (default=0)

**scale**: array_like, optional
- Scale parameter (default=1)

**Returns**
- **mean**: float
  - The mean of the distribution

---

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.median
```

**SkewNorm2_gen.median(*args, **kwds)**
- Median of the distribution.

**Parameters**
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)

- **loc**: array_like, optional
  - Location parameter, Default is 0.

- **scale**: array_like, optional
  - Scale parameter, Default is 1.

**Returns**
- **median**: float
  - The median of the distribution.

**See also:**
- `stats.distributions.rv_discrete.ppf` Inverse of the CDF

---

```python
statsmodels.sandbox.distributions.extras.SkewNorm2_gen.moment
```

**SkewNorm2_gen.moment(n, *args, **kwds)**
- n’th order non-central moment of distribution.

**Parameters**
- **n**: int, n>=1
  - Order of moment.

- **arg1, arg2, arg3,...**: float
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).

- **kwds**: keyword arguments, optional
  - These can include “loc” and “scale”, as well as other keyword arguments relevant for a given distribution.
**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.nnlf**

`SkewNorm2_gen.nnlf(theta, x)`

Return negative loglikelihood function

**Notes**

This is \(-\sum(\log \text{pdf}(x, \theta), \text{axis}=0)\) where \(\theta\) are the parameters (including loc and scale).

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.pdf**

`SkewNorm2_gen.pdf(x, *args, **kwds)`

Probability density function at \(x\) of the given RV.

**Parameters**

- **x**: array_like
  - quantiles
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

**Returns**

- **pdf**: ndarray
  - Probability density function evaluated at \(x\)

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.ppf**

`SkewNorm2_gen.ppf(q, *args, **kwds)`

Percent point function (inverse of cdf) at \(q\) of the given RV.

**Parameters**

- **q**: array_like
  - lower tail probability
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

**Returns**

- **x**: array_like
  - quantile corresponding to the lower tail probability \(q\).
**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.rvs**

`SkewNorm2_gen.rvs`(*args, **kwds)

Random variates of given type.

**Parameters**
arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

Location parameter (default=0).

scale : array_like, optional

Scale parameter (default=1).

size : int or tuple of ints, optional

Defining number of random variates (default=1).

**Returns**
rvs : ndarray or scalar

Random variates of given size.

---

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.sf**

`SkewNorm2_gen.sf`(*x, *args, **kwds)

Survival function (1-cdf) at x of the given RV.

**Parameters**
x : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

**Returns**
sf : array_like

Survival function evaluated at x

---

**statsmodels.sandbox.distributions.extras.SkewNorm2_gen.stats**

`SkewNorm2_gen.stats`(*args, **kwds)

Some statistics of the given RV

**Parameters**
arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

moments : str, optional
    composed of letters ['mvsk'] defining which moments to compute: ‘m’ = mean,
    ‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default='mv')

Returns stats : sequence
    of requested moments.

statsmodels.sandbox.distributions.extras.SkewNorm2_gen.std

SkewNorm2_gen.std(*args, **kwds)
    Standard deviation of the distribution.

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns std : float
    standard deviation of the distribution

statsmodels.sandbox.distributions.extras.SkewNorm2_gen.var

SkewNorm2_gen.var(*args, **kwds)
    Variance of the distribution

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns var : float
    the variance of the distribution
statsmodels.sandbox.distributions.extras.ACSkewT_gen

class statsmodels.sandbox.distributions.extras.ACSkewT_gen
univariate Skew-T distribution of Azzalini

class follows scipy.stats.distributions pattern but with __init__

Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdf(x, *args, **kwds)</td>
<td>Cumulative distribution function of the given RV.</td>
</tr>
<tr>
<td>entropy(*args, **kwds)</td>
<td>Differential entropy of the RV.</td>
</tr>
<tr>
<td>est_loc_scale(*args, **kwds)</td>
<td>Estimate loc and scale parameters from data.</td>
</tr>
<tr>
<td>expect([func, args, loc, scale, lb, ub, ...])</td>
<td>Calculate expected value of a function with respect to the distribution</td>
</tr>
<tr>
<td>fit(data, *args, **kwds)</td>
<td>Return MLEs for shape, location, and scale parameters from data.</td>
</tr>
<tr>
<td>fit_loc_scale(data, *args)</td>
<td>Estimate loc and scale parameters from data using 1st and 2nd moments.</td>
</tr>
<tr>
<td>freeze(*args, **kwds)</td>
<td>Freeze the distribution for the given arguments.</td>
</tr>
<tr>
<td>interval(alpha, *args, **kwds)</td>
<td>Confidence interval with equal areas around the median.</td>
</tr>
<tr>
<td>isf(q, *args, **kwds)</td>
<td>Inverse survival function at q of the given RV.</td>
</tr>
<tr>
<td>logcdf(x, *args, **kwds)</td>
<td>Log of the cumulative distribution function at x of the given RV.</td>
</tr>
<tr>
<td>logpdf(x, *args, **kwds)</td>
<td>Log of the probability density function at x of the given RV.</td>
</tr>
<tr>
<td>logsf(x, *args, **kwds)</td>
<td>Log of the survival function of the given RV.</td>
</tr>
<tr>
<td>mean(*args, **kwds)</td>
<td>Mean of the distribution.</td>
</tr>
<tr>
<td>median(*args, **kwds)</td>
<td>Median of the distribution.</td>
</tr>
<tr>
<td>moment(n, *args, **kwds)</td>
<td>n'th order non-central moment of distribution.</td>
</tr>
<tr>
<td>nnlf(theta, x)</td>
<td>Return negative loglikelihood function.</td>
</tr>
<tr>
<td>pdf(x, *args, **kwds)</td>
<td>Probability density function at x of the given RV.</td>
</tr>
<tr>
<td>ppf(q, *args, **kwds)</td>
<td>Percent point function (inverse of cdf) at q of the given RV.</td>
</tr>
<tr>
<td>rvs(*args, **kwds)</td>
<td>Random variates of given type.</td>
</tr>
<tr>
<td>sf(x, *args, **kwds)</td>
<td>Survival function (1-cdf) at x of the given RV.</td>
</tr>
<tr>
<td>stats(*args, **kwds)</td>
<td>Some statistics of the given RV</td>
</tr>
<tr>
<td>std(*args, **kwds)</td>
<td>Standard deviation of the distribution.</td>
</tr>
<tr>
<td>var(*args, **kwds)</td>
<td>Variance of the distribution.</td>
</tr>
</tbody>
</table>

statsmodels.sandbox.distributions.extras.ACSkewT_gen.cdf

ACSkewT_gen.cdf(x, *args, **kwds)
Cumulative distribution function of the given RV.

Parameters  
x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns  
cdf : ndarray
Cumulative distribution function evaluated at $x$

**statsmodels.sandbox.distributions.extras.ACSkewT_gen.entropy**

$\text{ACSkewT\_gen.entropy}(\ast\text{args}, \ast\ast\text{kwds})$

Differential entropy of the RV.

**Parameters**

- arg1, arg2, arg3,... : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- loc : array_like, optional
  - Location parameter (default=0).
- scale : array_like, optional
  - Scale parameter (default=1).

**statsmodels.sandbox.distributions.extras.ACSkewT_gen.est_loc_scale**

$\text{ACSkewT\_gen.est\_loc\_scale}(\ast\text{args}, \ast\ast\text{kwds})$

(est_loc_scale is deprecated!)

This function is deprecated, use self.fit_loc_scale(data) instead.

**statsmodels.sandbox.distributions.extras.ACSkewT_gen.expect**

$\text{ACSkewT\_gen.expect}(\text{func=None, args=()}, \text{loc=0, scale=1, lb=None, ub=None, conditional=False, }\ast\ast\text{kwds})$

Calculate expected value of a function with respect to the distribution

The expected value of a function $f(x)$ with respect to a distribution $\text{dist}$ is defined as:

\[
E[x] = \int_{\text{lbound}}^{\text{ubound}} f(x) \times \text{dist.pdf}(x)
\]

**Parameters**

- **func** : callable, optional
  - Function for which integral is calculated. Takes only one argument. The default is the identity mapping $f(x) = x$.
- **args** : tuple, optional
  - Argument (parameters) of the distribution.
- **lb, ub** : scalar, optional
  - Lower and upper bound for integration. default is set to the support of the distribution.
- **conditional** : bool, optional
  - If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.

Additional keyword arguments are passed to the integration routine.:
Returns `expect` : float

The calculated expected value.

Notes

The integration behavior of this function is inherited from `integrate.quad`.

`statsmodels.sandbox.distributions.extras.ACSkewT_gen.fit`

ACSkewT_gen.fit (data, *args, **kwds)

Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, self._fitstart(data) is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments f0, f1, ..., fn (for shape parameters) and floc and fscale (for location and scale parameters, respectively).

Parameters data : array_like

Data to use in calculating the MLEs.

args : floats, optional

Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to _fitstart(data)). No default value.

kwds : floats, optional

Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:

f0...fn : hold respective shape parameters fixed.
floc : hold location parameter fixed to specified value.
fscale : hold scale parameter fixed to specified value.

optimizer [The optimizer to use. The optimizer must take func,] and starting position as the first two arguments, plus args (for extra arguments to pass to the function to be optimized) and disp=0 to suppress output as keyword arguments.

Returns shape, loc, scale : tuple of floats

MLEs for any shape statistics, followed by those for location and scale.

Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.
ACSkewT_gen.fit_loc_scale(data, *args)
Estimate loc and scale parameters from data using 1st and 2nd moments.

Parameters
data : array_like
   Data to fit.
arg1, arg2, arg3,... : array_like
   The shape parameter(s) for the distribution (see docstring of the instance object for more information).

Returns
Lhat : float
   Estimated location parameter for the data.
S hat : float
   Estimated scale parameter for the data.

ACSkewT_gen.freeze(*args, **kwds)
Freeze the distribution for the given arguments.

Parameters
arg1, arg2, arg3,... : array_like
   The shape parameter(s) for the distribution. Should include all the non-optinal arguments, may include loc and scale.

Returns
rv_frozen : rv_frozen instance
   The frozen distribution.

ACSkewT_gen.interval(alpha, *args, **kwds)
Confidence interval with equal areas around the median.

Parameters
alpha : array_like of float
   Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].
arg1, arg2, ... : array_like
   The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
   location parameter, Default is 0.
scale : array_like, optional
   scale parameter, Default is 1.

Returns
a, b : ndarray of float
   end-points of range that contain 100 * alpha % of the rv’s possible values.
ACSkewT_gen.isf(q, *args, **kwds)
Inverse survival function at q of the given RV.

Parameters:
- **q**: array_like
  - upper tail probability
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

Returns:
- **x**: ndarray or scalar
  - Quantile corresponding to the upper tail probability q.

ACSkewT_gen.logcdf(x, *args, **kwds)
Log of the cumulative distribution function at x of the given RV.

Parameters:
- **x**: array_like
  - quantiles
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

Returns:
- **logcdf**: array_like
  - Log of the cumulative distribution function evaluated at x

ACSkewT_gen.logpdf(x, *args, **kwds)
Log of the probability density function at x of the given RV.
This uses a more numerically accurate calculation if available.

Parameters:
- **x**: array_like
  - quantiles
- **arg1, arg2, arg3,...**: array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc**: array_like, optional
location parameter (default=0)

**scale**: array_like, optional
scale parameter (default=1)

Returns **logpdf**: array_like
Log of the probability density function evaluated at x

```
statsmodels.sandbox.distributions.extras.ACSkewT_gen.logsf

ACSkewT_gen.logsf(x, *args, **kwds)
Log of the survival function of the given RV.
Returns the log of the “survival function,” defined as (1 - \text{cdf}), evaluated at \text{x}.

Parameters 
\text{x} : array_like
quantiles
\text{arg1, arg2, arg3,...} : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

\text{loc} : array_like, optional
location parameter (default=0)

\text{scale} : array_like, optional
scale parameter (default=1)

Returns **logsf**: ndarray
Log of the survival function evaluated at \text{x}.
```

```
statsmodels.sandbox.distributions.extras.ACSkewT_gen.mean

ACSkewT_gen.mean(*args, **kwds)
Mean of the distribution

Parameters \text{arg1, arg2, arg3,...} : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

\text{loc} : array_like, optional
location parameter (default=0)

\text{scale} : array_like, optional
scale parameter (default=1)

Returns **mean**: float
the mean of the distribution
```
ACSkewT_gen.median(*args, **kwds)
Median of the distribution.

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)
loc : array_like, optional
    Location parameter, Default is 0.
scale : array_like, optional
    Scale parameter, Default is 1.

Returns median : float
    The median of the distribution.

See also:
stats.distributions.rv_discrete.ppf Inverse of the CDF

ACSkewT_gen.moment(n, *args, **kwds)
n'th order non-central moment of distribution.

Parameters n : int, n>=1
    Order of moment.
    arg1, arg2, arg3,... : float
        The shape parameter(s) for the distribution (see docstring of the instance object
        for more information).
kwds : keyword arguments, optional
    These can include “loc” and “scale”, as well as other keyword arguments relevant
    for a given distribution.

ACSkewT_gen.nnlf(theta, x)
Return negative loglikelihood function

Notes
This is \(-\text{sum} (\log \text{pdf}(x, \theta), \text{axis}=0)\) where \(\theta\) are the parameters (including loc and scale).
ACSkewT_gen.pdf

ACSkewT_gen.pdf \( (x, *\text{args}, **\text{kwds}) \)
Probability density function at \( x \) of the given RV.

**Parameters**
- \( x \): array_like
  - quantiles
- \( \text{arg1}, \text{arg2}, \text{arg3},... \): array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- \( \text{loc} \): array_like, optional
  - location parameter (default=0)
- \( \text{scale} \): array_like, optional
  - scale parameter (default=1)

**Returns**
- \( \text{pdf} \): ndarray
  - Probability density function evaluated at \( x \)

ACSkewT_gen.ppf

ACSkewT_gen.ppf \( (q, *\text{args}, **\text{kwds}) \)
Percent point function (inverse of cdf) at \( q \) of the given RV.

**Parameters**
- \( q \): array_like
  - lower tail probability
- \( \text{arg1}, \text{arg2}, \text{arg3},... \): array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- \( \text{loc} \): array_like, optional
  - location parameter (default=0)
- \( \text{scale} \): array_like, optional
  - scale parameter (default=1)

**Returns**
- \( x \): array_like
  - quantile corresponding to the lower tail probability \( q \)

ACSkewT_gen.rvs

ACSkewT_gen.rvs \( (*\text{args}, **\text{kwds}) \)
Random variates of given type.

**Parameters**
- \( \text{arg1}, \text{arg2}, \text{arg3},... \): array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- \( \text{loc} \): array_like, optional
Location parameter (default=0).

**scale** : array_like, optional
Scale parameter (default=1).

**size** : int or tuple of ints, optional
Defining number of random variates (default=1).

**Returns**

**rvs** : ndarray or scalar
Random variates of given size.

```python
statsmodels.sandbox.distributions.extras.ACSkewT_gen.sf
```

**ACSkewT_gen.sf***(x, *args, **kwds)**
Survival function (1-cdf) at x of the given RV.

**Parameters**

**x** : array_like
quantiles

**arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc** : array_like, optional
location parameter (default=0)

**scale** : array_like, optional
scale parameter (default=1)

**Returns**

**sf** : array_like
Survival function evaluated at x

```python
statsmodels.sandbox.distributions.extras.ACSkewT_gen.stats
```

**ACSkewT_gen.stats***(*args, **kwds)**
Some statistics of the given RV

**Parameters**

**arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc** : array_like, optional
location parameter (default=0)

**scale** : array_like, optional
scale parameter (default=1)

**moments** : str, optional
composed of letters ['mvsk'] defining which moments to compute: ‘m’ = mean, ‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default='mv')

**Returns**

**stats** : sequence
of requested moments.

\texttt{statsmodels.sandbox.distributions.extras.ACSkewT\_gen.std}

\texttt{ACSkewT\_gen.std(*args, **kwds)}

Standard deviation of the distribution.

**Parameters**
- \texttt{arg1, arg2, arg3,...} : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- \texttt{loc} : array_like, optional
  - location parameter (default=0)
- \texttt{scale} : array_like, optional
  - scale parameter (default=1)

**Returns**
- \texttt{std} : float
  - standard deviation of the distribution

\texttt{statsmodels.sandbox.distributions.extras.ACSkewT\_gen.var}

\texttt{ACSkewT\_gen.var(*args, **kwds)}

Variance of the distribution

**Parameters**
- \texttt{arg1, arg2, arg3,...} : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- \texttt{loc} : array_like, optional
  - location parameter (default=0)
- \texttt{scale} : array_like, optional
  - scale parameter (default=1)

**Returns**
- \texttt{var} : float
  - the variance of the distribution

\texttt{statsmodels.sandbox.distributions.extras.skewnorm2}

\texttt{skewnorm2 = <statsmodels.sandbox.distributions.extras.SkewNorm2\_gen object at 0x082AA730>}

univariate Skew-Normal distribution of Azzalini

class follows scipy.stats.distributions pattern

**Distributions based on Gram-Charlier expansion**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{pdf_moments_st}(cnt)</td>
<td>Return the Gaussian expanded pdf function given the list of central moments</td>
</tr>
<tr>
<td>\texttt{pdf_mwsk}(mwsk)</td>
<td>Return the Gaussian expanded pdf function given the list of 1st, 2nd moments</td>
</tr>
<tr>
<td>\texttt{pdf_moments}(cnt)</td>
<td>Return the Gaussian expanded pdf function given the list of central moments</td>
</tr>
<tr>
<td>\texttt{NormExpan_gen}(args, **kwds)</td>
<td>Gram-Charlier Expansion of Normal distribution</td>
</tr>
</tbody>
</table>
statsmodels.sandbox.distributions.extras.pdf_moments_st

statsmodels.sandbox.distributions.extras.pdf_moments_st(cnt)
Return the Gaussian expanded pdf function given the list of central moments (first one is mean).
version of scipy.stats, any changes ? the scipy.stats version has a bug and returns normal distribution

statsmodels.sandbox.distributions.extras.pdf_mvsk

statsmodels.sandbox.distributions.extras.pdf_mvsk(mvsk)
Return the Gaussian expanded pdf function given the list of 1st, 2nd moment and skew and Fisher (excess) kurtosis.

Parameters mvsk : list of mu, mc2, skew, kurt
distribution is matched to these four moments

Returns pdffunc : function
function that evaluates the pdf(x), where x is the non-standardized random variable.

Notes

Changed so it works only if four arguments are given. Uses explicit formula, not loop.
This implements a Gram-Charlier expansion of the normal distribution where the first 2 moments coincide with those of the normal distribution but skew and kurtosis can deviate from it.
In the Gram-Charlier distribution it is possible that the density becomes negative. This is the case when the deviation from the normal distribution is too large.

References


statsmodels.sandbox.distributions.extras.pdf_moments

statsmodels.sandbox.distributions.extras.pdf_moments(cnt)
Return the Gaussian expanded pdf function given the list of central moments (first one is mean).
Changed so it works only if four arguments are given. Uses explicit formula, not loop.

Notes

This implements a Gram-Charlier expansion of the normal distribution where the first 2 moments coincide with those of the normal distribution but skew and kurtosis can deviate from it.
In the Gram-Charlier distribution it is possible that the density becomes negative. This is the case when the deviation from the normal distribution is too large.
References


statsmodels.sandbox.distributions.extras.NormExpan_gen

class NormExpan_gen (args, **kwds)

Gram-Charlier Expansion of Normal distribution

class follows scipy.stats.distributions pattern but with __init__

Methods

cdf(x, *args, **kwds) Cumulative distribution function of the given RV.
entropy(*args, **kwds) Differential entropy of the RV.
est_loc_scale(*args, **kwds) est_loc_scale is deprecated!
expect(func, args, loc, scale, lb, ub, ...]) Calculate expected value of a function with respect to the distribution
fit(data, *args, **kwds) Return MLEs for shape, location, and scale parameters from data.
fit_loc_scale(data, *args) Estimate loc and scale parameters from data using 1st and 2nd moments.
freeze(*args, **kwds) Freeze the distribution for the given arguments.
interval(alpha, *args, **kwds) Confidence interval with equal areas around the median.
isf(q, *args, **kwds) Inverse survival function at q of the given RV.
logcdf(x, *args, **kwds) Log of the cumulative distribution function at x of the given RV.
logpdf(x, *args, **kwds) Log of the probability density function at x of the given RV.
logsf(x, *args, **kwds) Log of the survival function of the given RV.
mean(*args, **kwds) Mean of the distribution
median(*args, **kwds) Median of the distribution.
moment(n, *args, **kwds) n’th order non-central moment of distribution.
nllf(theta, x) Return negative loglikelihood function..
pdf(x, *args, **kwds) Probability density function at x of the given RV.
ppf(q, *args, **kwds) Percent point function (inverse cdf) at q of the given RV.
rvs(*args, **kwds) Random variates of given type.
sf(x, *args, **kwds) Survival function (1-cdf) at x of the given RV.
stats(*args, **kwds) Some statistics of the given RV
std(*args, **kwds) Standard deviation of the distribution.
var(*args, **kwds) Variance of the distribution

3.13. Distributions
...location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns cdf : ndarray

Cumulative distribution function evaluated at x

statsmodels.sandbox.distributions.extras.NormExpan_gen.entropy

NormExpan_gen.entropy (*args, **kwds)

Differential entropy of the RV.

Parameters arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

Location parameter (default=0).

scale : array_like, optional

Scale parameter (default=1).

statsmodels.sandbox.distributions.extras.NormExpan_gen.est_loc_scale

NormExpan_gen.est_loc_scale (*args, **kwds)

est_loc_scale is deprecated!

This function is deprecated, use self.fit_loc_scale(data) instead.

statsmodels.sandbox.distributions.extras.NormExpan_gen.expect

NormExpan_gen.expect (func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)

Calculate expected value of a function with respect to the distribution

The expected value of a function f(x) with respect to a distribution dist is defined as:

\[ E[x] = \int f(x) \cdot dist.pdf(x) \]

Parameters func : callable, optional

Function for which integral is calculated. Takes only one argument. The default is the identity mapping f(x) = x.

args : tuple, optional

Argument (parameters) of the distribution.

lb, ub : scalar, optional

Lower and upper bound for integration. default is set to the support of the distribution.
conditional : bool, optional
If True, the integral is corrected by the conditional probability of the integration
interval. The return value is the expectation of the function, conditional on being
in the given interval. Default is False.

Additional keyword arguments are passed to the integration routine.

Returns expect : float
The calculated expected value.

Notes
The integration behavior of this function is inherited from integrate.quad.

statsmodels.sandbox.distributions.extras.NormExpan_gen.fit

NormExpan_gen.fit(data, *args, **kwds)
Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input argu-
ments; for any arguments not provided with starting estimates, self._fitstart(data) is called to
generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments f0, f1, ..., fn
(for shape parameters) and floc and fscale (for location and scale parameters, respectively).

Parameters data : array_like
Data to use in calculating the MLEs.

args : floats, optional
Starting value(s) for any shape-characterizing arguments (those not provided will
be determined by a call to _fitstart(data)). No default value.

kwds : floats, optional
Starting values for the location and scale parameters; no default. Special keyword
arguments are recognized as holding certain parameters fixed:

f0...fn : hold respective shape parameters fixed.
floc : hold location parameter fixed to specified value.
scale : hold scale parameter fixed to specified value.

optimizer [The optimizer to use. The optimizer must take func.] and starting
position as the first two arguments, plus args (for extra arguments to pass to the
function to be optimized) and disp=0 to suppress output as keyword arguments.

Returns shape, loc, scale : tuple of floats
MLEs for any shape statistics, followed by those for location and scale.
Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.

```python
statsmodels.sandbox.distributions.extras.NormExpan_gen.fit_loc_scale

NormExpan_gen.fit_loc_scale(data, *args)
Estimate loc and scale parameters from data using 1st and 2nd moments.

Parameters
- data : array_like
  Data to fit.
- arg1, arg2, arg3,... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).

Returns
- Lhat : float
  Estimated location parameter for the data.
- Shat : float
  Estimated scale parameter for the data.
```

```python
statsmodels.sandbox.distributions.extras.NormExpan_gen.freeze

NormExpan_gen.freeze(*args, **kwds)
Freeze the distribution for the given arguments.

Parameters
- arg1, arg2, arg3,... : array_like
  The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include loc and scale.

Returns
- rv_frozen : rv_frozen instance
  The frozen distribution.
```

```python
statsmodels.sandbox.distributions.extras.NormExpan_gen.interval

NormExpan_gen.interval(alpha, *args, **kwds)
Confidence interval with equal areas around the median.

Parameters
- alpha : array_like of float
  Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].
- arg1, arg2, ... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- loc : array_like, optional
  location parameter, Default is 0.
scale : array_like, optional
    scale parameter, Default is 1.

Returns a, b : ndarray of float
    end-points of range that contain 100 * alpha % of the rv's possible values.

**statsmodels.sandbox.distributions.extras.NormExpan_gen.isf**

NormExpan_gen.isf(q, *args, **kwds)
Inverse survival function at q of the given RV.

Parameters q : array_like
    upper tail probability
arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
    location parameter (default=0)
scale : array_like, optional
    scale parameter (default=1)

Returns x : ndarray or scalar
    Quantile corresponding to the upper tail probability q.

**statsmodels.sandbox.distributions.extras.NormExpan_gen.logcdf**

NormExpan_gen.logcdf(x, *args, **kwds)
Log of the cumulative distribution function at x of the given RV.

Parameters x : array_like
    quantiles
arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
    location parameter (default=0)
scale : array_like, optional
    scale parameter (default=1)

Returns logcdf : array_like
    Log of the cumulative distribution function evaluated at x
statsmodels.sandbox.distributions.extras.NormExpan_gen.logpdf

NormExpan_gen.logpdf(x, *args, **kwds)
Log of the probability density function at x of the given RV.
This uses a more numerically accurate calculation if available.

Parameters:
- x : array_like
  quantiles
- arg1, arg2, arg3,... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- loc : array_like, optional
  location parameter (default=0)
- scale : array_like, optional
  scale parameter (default=1)

Returns:
- logpdf : array_like
  Log of the probability density function evaluated at x

statsmodels.sandbox.distributions.extras.NormExpan_gen.logsf

NormExpan_gen.logsf(x, *args, **kwds)
Log of the survival function of the given RV.
Returns the log of the “survival function,” defined as (1 - cdf), evaluated at x.

Parameters:
- x : array_like
  quantiles
- arg1, arg2, arg3,... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- loc : array_like, optional
  location parameter (default=0)
- scale : array_like, optional
  scale parameter (default=1)

Returns:
- logsf : ndarray
  Log of the survival function evaluated at x.

statsmodels.sandbox.distributions.extras.NormExpan_gen.mean

NormExpan_gen.mean(*args, **kwds)
Mean of the distribution

Parameters:
- arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc**: array_like, optional  
Location parameter (default=0)

**scale**: array_like, optional  
Scale parameter (default=1)

**Returns**

- **mean**: float  
The mean of the distribution

```python
statsmodels.sandbox.distributions.extras.NormExpan_gen.median
```

**NormExpan_gen.median** (*args, **kwds*)  
Median of the distribution.

**Parameters**

- **arg1, arg2, arg3,**...: array_like  
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

- **loc**: array_like, optional  
Location parameter, Default is 0.

- **scale**: array_like, optional  
Scale parameter, Default is 1.

**Returns**

- **median**: float  
The median of the distribution.

**See also:**

- **stats.distributions.rv_discrete.ppf** Inverse of the CDF

```python
statsmodels.sandbox.distributions.extras.NormExpan_gen.moment
```

**NormExpan_gen.moment** (*n*, *args, **kwds*)  
n’th order non-central moment of distribution.

**Parameters**

- **n**: int, n>=1  
Order of moment.

- **arg1, arg2, arg3,**...: float  
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

- **kwds**: keyword arguments, optional  
These can include “loc” and “scale”, as well as other keyword arguments relevant for a given distribution.
**statsmodels.sandbox.distributions.extras.NormExpan_gen.nnlf**

`NormExpan_gen.nnlf(theta, x)`

Return negative loglikelihood function

**Notes**

This is \(-\text{sum}(\log \text{ pdf}(x, \theta), \text{axis}=0)\) where \(\theta\) are the parameters (including loc and scale).

**statsmodels.sandbox.distributions.extras.NormExpan_gen.pdf**

`NormExpan_gen.pdf(x, *args, **kwds)`

Probability density function at \(x\) of the given RV.

**Parameters**

\(x\) : array_like
  quantiles

\(arg1, arg2, arg3,...\) : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)

\(loc\) : array_like, optional
  location parameter (default=0)

\(scale\) : array_like, optional
  scale parameter (default=1)

**Returns**

\(pdf\) : ndarray
  Probability density function evaluated at \(x\)

**statsmodels.sandbox.distributions.extras.NormExpan_gen.ppf**

`NormExpan_gen.ppf(q, *args, **kwds)`

Percent point function (inverse of cdf) at \(q\) of the given RV.

**Parameters**

\(q\) : array_like
  lower tail probability

\(arg1, arg2, arg3,...\) : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)

\(loc\) : array_like, optional
  location parameter (default=0)

\(scale\) : array_like, optional
  scale parameter (default=1)

**Returns**

\(x\) : array_like
  quantile corresponding to the lower tail probability \(q\).
statsmodels.sandbox.distributions.extras.NormExpan_gen.rvs

NormExpan_gen.rvs(*args, **kwds)
Random variates of given type.

Parameters arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object
for more information).
loc : array_like, optional
Location parameter (default=0).
scale : array_like, optional
Scale parameter (default=1).
size : int or tuple of ints, optional
Defining number of random variates (default=1).

Returns rvs : ndarray or scalar
Random variates of given size.

statsmodels.sandbox.distributions.extras.NormExpan_gen.sf

NormExpan_gen.sf(x, *args, **kwds)
Survival function (1-cdf) at x of the given RV.

Parameters x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object
for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns sf : array_like
Survival function evaluated at x

statsmodels.sandbox.distributions.extras.NormExpan_gen.stats

NormExpan_gen.stats(*args, **kwds)
Some statistics of the given RV

Parameters arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object
for more information)
loc : array_like, optional
location parameter (default=0)

**scale**: array_like, optional

scale parameter (default=1)

**moments**: str, optional

composed of letters [‘mvsk’] defining which moments to compute: ‘m’ = mean,
‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default=’mv’)

**Returns**

**stats**: sequence

of requested moments.

```python
statsmodels.sandbox.distributions.extras.NormExpan_gen.std
```

**NormExpan_gen.std** (*args, **kwds*)

Standard deviation of the distribution.

**Parameters**

**arg1, arg2, arg3,**... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc** : array_like, optional

location parameter (default=0)

**scale** : array_like, optional

scale parameter (default=1)

**Returns**

**std** : float

standard deviation of the distribution

```python
statsmodels.sandbox.distributions.extras.NormExpan_gen.var
```

**NormExpan_gen.var** (*args, **kwds*)

Variance of the distribution

**Parameters**

**arg1, arg2, arg3,**... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc** : array_like, optional

location parameter (default=0)

**scale** : array_like, optional

scale parameter (default=1)

**Returns**

**var** : float

the variance of the distribution

**cdf of multivariate normal** wrapper for scipy.stats

```python
mvdtnormcdf(lower, upper, corrcoef, **kwds)  
mvnormcdf(upper, mu, cov[, lower])
```

standardized multivariate normal cumulative distribution function

multivariate normal cumulative distribution function
standardized multivariate normal cumulative distribution function

This is a wrapper for scipy.stats.kde.mvn.mvndst which calculates a rectangular integral over a standardized multivariate normal distribution.

This function assumes standardized scale, that is the variance in each dimension is one, but correlation can be arbitrary, covariance = correlation matrix

Parameters

lower, upper : array_like, 1d
  lower and upper integration limits with length equal to the number of dimensions of the multivariate normal distribution. It can contain -np.inf or np.inf for open integration intervals

corrcoef : float or array_like
  specifies correlation matrix in one of three ways, see notes

Notes

The correlation matrix corrcoef can be given in 3 different ways If the multivariate normal is two-dimensional than only the correlation coefficient needs to be provided. For general dimension the correlation matrix can be provided either as a one-dimensional array of the upper triangular correlation coefficients stacked by rows, or as full square correlation matrix

Examples

```python
>>> print mvstdnormcdf([-np.inf,-np.inf], [0.0,np.inf], 0.5)
0.5
>>> corr = [[1.0, 0, 0.5],[0,1,0],[0.5,0,1]]
>>> print mvstdnormcdf([-np.inf,-np.inf,-100.0], [0.0,0.0,0.0], corr, abseps=1e-6)
0.166666399198
>>> print mvstdnormcdf([-np.inf,-np.inf,-100.0], [0.0,0.0,0.0], corr, abseps=1e-8)
something wrong completion with ERROR > EPS and MAXPTS function values used; increase MAXPTS to decrease ERROR; 1.048330348e-006
0.166666546218
>>> print mvstdnormcdf([-np.inf,-np.inf,-100.0], [0.0,0.0,0.0], corr,
```
statsmodels Documentation, Release 0.6.0

```python
0.166666588293

maxpts=100000, abseps=1e-8)
```

```python
statsmodels.sandbox.distributions.extras.mvnormcdf
```

```python
statsmodels.sandbox.distributions.extras.mvnormcdf(upper, mu, cov, lower=None,
**kwds)
```

multivariate normal cumulative distribution function

This is a wrapper for scipy.stats.kde.mvn.mvndst which calculates a rectangular integral over a multivariate normal distribution.

**Parameters**

- `lower`, `upper` : array_like, 1d
  lower and upper integration limits with length equal to the number of dimensions of the multivariate normal distribution. It can contain -np.inf or np.inf for open integration intervals

- `mu` : array_like, 1d
  list or array of means

- `cov` : array_like, 2d
  specifies covariance matrix

**optional keyword parameters to influence integration** :

- `maxpts` [int, maximum number of function values allowed. This parameter can be used to limit the time. A sensible strategy is to start with maxpts = 1000*N, and then increase maxpts if ERROR is too large.

- `abseps` : float absolute error tolerance.

- `releps` : float relative error tolerance.

**Returns**

- `cdfvalue` : float
  value of the integral

**See also**:

- `mvstnormcdf` location and scale standardized multivariate normal cdf

**Notes**

This function normalizes the location and scale of the multivariate normal distribution and then uses `mvstnormcdf` to call the integration.

### 3.13.3 Univariate Distributions by non-linear Transformations

Univariate distributions can be generated from a non-linear transformation of an existing univariate distribution. `Transf_gen` is a class that can generate a new distribution from a monotonic transformation. `TransfTwo_gen` can use hump-shaped or u-shaped transformation, such as abs or square. The remaining objects are special cases.

```python
TransfTwo_gen(kls, func, funcinvplus, ...)
```

Distribution based on a non-monotonic (u- or hump-shaped transformation) the

```python
Transf_gen(kls, func, funcinv, *args, **kwargs)
```

a class for non-linear monotonic transformation of a continuous random variable
Table 3.262 – continued from previous page

<table>
<thead>
<tr>
<th>Distribution Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExpTransf_gen</td>
<td>Distribution based on log/exp transformation</td>
</tr>
<tr>
<td>LogTransf_gen</td>
<td>Distribution based on log/exp transformation</td>
</tr>
<tr>
<td>SquareFunc</td>
<td>class to hold quadratic function with inverse function and derivative</td>
</tr>
<tr>
<td>absnormalg</td>
<td>Distribution based on a non-monotonic (u- or hump-shaped transformation)</td>
</tr>
<tr>
<td>invdnormalg</td>
<td>a class for non-linear monotonic transformation of a continuous random variable</td>
</tr>
<tr>
<td>loggammaexpg</td>
<td>a class for non-linear monotonic transformation of a continuous random variable</td>
</tr>
<tr>
<td>lognormalg</td>
<td>a class for non-linear monotonic transformation of a continuous random variable</td>
</tr>
<tr>
<td>negsquarenormalg</td>
<td>Distribution based on a non-monotonic (u- or hump-shaped transformation)</td>
</tr>
<tr>
<td>squarenormalg</td>
<td>Distribution based on a non-monotonic (u- or hump-shaped transformation)</td>
</tr>
<tr>
<td>squaretg</td>
<td>Distribution based on a non-monotonic (u- or hump-shaped transformation)</td>
</tr>
</tbody>
</table>

**statsmodels.sandbox.distributions.transformed.TransfTwo_gen**

```python
class statsmodels.sandbox.distributions.transformed.TransfTwo_gen(kls, func, funcinvplus, funcinvminus, derivplus, derivminus, *args, **kwargs)
```

Distribution based on a non-monotonic (u- or hump-shaped transformation)

the constructor can be called with a distribution class, and functions that define the non-linear transformation.
and generates the distribution of the transformed random variable

Note: the transformation, it’s inverse and derivatives need to be fully specified: func, funcinvplus, funcinvminus, derivplus, derivminus. Currently no numerical derivatives or inverse are calculated

This can be used to generate distribution instances similar to the distributions in scipy.stats.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdf(x, *args, **kws)</td>
<td>Cumulative distribution function of the given RV.</td>
</tr>
<tr>
<td>entropy(*args, **kws)</td>
<td>Differential entropy of the RV.</td>
</tr>
<tr>
<td>est_loc_scale(*args, **kws)</td>
<td>Calculate expected value of a function with respect to the distribution</td>
</tr>
<tr>
<td>expect([func, args, loc, scale, lb, ub, ...])</td>
<td>Return MLEs for shape, location, and scale parameters from data.</td>
</tr>
<tr>
<td>fit(data, *args, **kws)</td>
<td>Estimate loc and scale parameters from data using 1st and 2nd moments.</td>
</tr>
<tr>
<td>fit_loc_scale(data, *args)</td>
<td>Freeze the distribution for the given arguments.</td>
</tr>
<tr>
<td>interval(alpha, *args, **kws)</td>
<td>Confidence interval with equal areas around the median.</td>
</tr>
<tr>
<td>isf(q, *args, **kws)</td>
<td>Inverse survival function at q of the given RV.</td>
</tr>
<tr>
<td>logcdf(x, *args, **kws)</td>
<td>Log of the cumulative distribution function at x of the given RV.</td>
</tr>
<tr>
<td>logpdf(x, *args, **kws)</td>
<td>Log of the probability density function at x of the given RV.</td>
</tr>
<tr>
<td>logsf(x, *args, **kws)</td>
<td>Log of the survival function of the given RV.</td>
</tr>
<tr>
<td>mean(*args, **kws)</td>
<td>Mean of the distribution</td>
</tr>
<tr>
<td>median(*args, **kws)</td>
<td>Median of the distribution</td>
</tr>
<tr>
<td>moment(n, *args, **kws)</td>
<td>n'th order non-central moment of distribution.</td>
</tr>
<tr>
<td>nnlf(theta, x)</td>
<td>Return negative loglikelihood function ..</td>
</tr>
<tr>
<td>pdf(x, *args, **kws)</td>
<td>Probability density function at x of the given RV.</td>
</tr>
<tr>
<td>ppf(q, *args, **kws)</td>
<td>Percent point function (inverse of cdf) at q of the given RV.</td>
</tr>
<tr>
<td>rvs(*args, **kws)</td>
<td>Random variates of given type.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.263 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sf(*args, **kwds)</code></td>
<td>Survival function (1-cdf) at x of the given RV.</td>
</tr>
<tr>
<td><code>stats(*args, **kwds)</code></td>
<td>Some statistics of the given RV.</td>
</tr>
<tr>
<td><code>std(*args, **kwds)</code></td>
<td>Standard deviation of the distribution.</td>
</tr>
<tr>
<td><code>var(*args, **kwds)</code></td>
<td>Variance of the distribution.</td>
</tr>
</tbody>
</table>

**statsmodels.sandbox.distributions.transformed.TransfTwo_gen.cdf**

```python
TransfTwo_gen.cdf(x, *args, **kwds)
```
Cumulative distribution function of the given RV.

**Parameters**

- `x`: array_like
- `arg1, arg2, arg3,...`: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- `loc`: array_like, optional
  - Location parameter (default=0).
- `scale`: array_like, optional
  - Scale parameter (default=1).

**Returns**

- `cdf`: ndarray
  - Cumulative distribution function evaluated at x

**statsmodels.sandbox.distributions.transformed.TransfTwo_gen.entropy**

```python
TransfTwo_gen.entropy(*args, **kwds)
```
Differential entropy of the RV.

**Parameters**

- `arg1, arg2, arg3,...`: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- `loc`: array_like, optional
  - Location parameter (default=0).
- `scale`: array_like, optional
  - Scale parameter (default=1).

**statsmodels.sandbox.distributions.transformed.TransfTwo_gen.est_loc_scale**

```python
TransfTwo_gen.est_loc_scale(*args, **kwds)
```
est_loc_scale is deprecated!

This function is deprecated, use self.fit_loc_scale(data) instead.
statsmodels.sandbox.distributions.transformed.TransfTwo_gen.expect

TransfTwo_gen.expect(func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)

Calculate expected value of a function with respect to the distribution

The expected value of a function \( f(x) \) with respect to a distribution \( \text{dist} \) is defined as:

\[
E[x] = \int_{lbound}^{ubound} f(x) \times \text{dist.pdf}(x)
\]

Parameters

- **func**: callable, optional
  Function for which integral is calculated. Takes only one argument. The default is the identity mapping \( f(x) = x \).

- **args**: tuple, optional
  Argument (parameters) of the distribution.

- **lb**, **ub**: scalar, optional
  Lower and upper bound for integration. default is set to the support of the distribution.

- **conditional**: bool, optional
  If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.

Additional keyword arguments are passed to the integration routine.

Returns

- **expect**: float
  The calculated expected value.

Notes

The integration behavior of this function is inherited from \( \text{integrate.quad} \).

statsmodels.sandbox.distributions.transformed.TransfTwo_gen.fit

TransfTwo_gen.fit(data, *args, **kwds)

Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, self._fitstart(data) is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments \( f_0, f_1, \ldots, f_n \) (for shape parameters) and \( floc \) and \( fscale \) (for location and scale parameters, respectively).

Parameters

- **data**: array_like
  Data to use in calculating the MLEs.

- **args**: floats, optional
Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to \texttt{_fitstart(data)}). No default value.

**kwds: floats, optional**

Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:

- \texttt{f0...fn}: hold respective shape parameters fixed.
- \texttt{floc}: hold location parameter fixed to specified value.
- \texttt{fscale}: hold scale parameter fixed to specified value.

**optimizer** [The optimizer to use. The optimizer must take \texttt{func,} and starting position as the first two arguments, plus \texttt{args} (for extra arguments to pass to the function to be optimized) and \texttt{disp=0} to suppress output as keyword arguments.]

**Returns** \texttt{shape, loc, scale}: tuple of floats

MLEs for any shape statistics, followed by those for location and scale.

**Notes**

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.

### \texttt{statsmodels.sandbox.distributions.transformed.TransfTwo_gen.fit_loc_scale}

\texttt{TransfTwo_gen.fit_loc_scale(data, *args)}

Estimate loc and scale parameters from data using 1st and 2nd moments.

**Parameters** \texttt{data}: array_like

Data to fit.

\texttt{arg1, arg2, arg3,...}: array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

**Returns** \texttt{Lhat}: float

Estimated location parameter for the data.

\texttt{Shat}: float

Estimated scale parameter for the data.

### \texttt{statsmodels.sandbox.distributions.transformed.TransfTwo_gen.freeze}

\texttt{TransfTwo_gen.freeze(*args, **kwds)}

Freeze the distribution for the given arguments.

**Parameters** \texttt{arg1, arg2, arg3,...}: array_like

The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include \texttt{loc} and \texttt{scale}.

**Returns** \texttt{rv_frozen}: rv_frozen instance
The frozen distribution.

**statsmodels.sandbox.distributions.transformed.TransfTwo_gen.interval**

```python
TransfTwo_gen.interval(alpha, *args, **kwds)
```

Confidence interval with equal areas around the median.

**Parameters**

- **alpha**: array_like of float
  Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].
- **arg1, arg2, ...**: array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- **loc**: array_like, optional
  location parameter, Default is 0.
- **scale**: array_like, optional
  scale parameter, Default is 1.

**Returns**

- **a, b**: ndarray of float
  end-points of range that contain \(100 \times \alpha\) % of the rv’s possible values.

**statsmodels.sandbox.distributions.transformed.TransfTwo_gen.isf**

```python
TransfTwo_gen.isf(q, *args, **kwds)
```

Inverse survival function at q of the given RV.

**Parameters**

- **q**: array_like
  upper tail probability
- **arg1, arg2, arg3,...**: array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  location parameter (default=0)
- **scale**: array_like, optional
  scale parameter (default=1)

**Returns**

- **x**: ndarray or scalar
  Quantile corresponding to the upper tail probability q.

**statsmodels.sandbox.distributions.transformed.TransfTwo_gen.logcdf**

```python
TransfTwo_gen.logcdf(x, *args, **kwds)
```

Log of the cumulative distribution function at x of the given RV.

**Parameters**

- **x**: array_like
quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns logcdf : array_like

Log of the cumulative distribution function evaluated at x

statsmodels.sandbox.distributions.transformed.TransfTwo_gen.logpdf

TransfTwo_gen.logpdf(x, *args, **kwds)

Log of the probability density function at x of the given RV.

This uses a more numerically accurate calculation if available.

Parameters x : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns logpdf : array_like

Log of the probability density function evaluated at x

statsmodels.sandbox.distributions.transformed.TransfTwo_gen.logsf

TransfTwo_gen.logsf(x, *args, **kwds)

Log of the survival function of the given RV.

Returns the log of the “survival function,” defined as (1 - pdf), evaluated at x.

Parameters x : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns logsf : ndarray
    Log of the survival function evaluated at x.

statsmodels.sandbox.distributions.transformed.TransfTwo_gen.mean

TransfTwo_gen.mean(*args, **kwds)
Mean of the distribution

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns mean : float
    the mean of the distribution

statsmodels.sandbox.distributions.transformed.TransfTwo_gen.median

TransfTwo_gen.median(*args, **kwds)
Median of the distribution.

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
    Location parameter, Default is 0.

scale : array_like, optional
    Scale parameter, Default is 1.

Returns median : float
    The median of the distribution.

See also:

stats.distributions.rv_discrete.ppf Inverse of the CDF
statsmodels.sandbox.distributions.transformed.TransfTwo_gen.moment

TransfTwo_gen.moment(n, *args, **kwds)

n'th order non-central moment of distribution.

Parameters

n : int, n>=1

Order of moment.

arg1, arg2, arg3,... : float

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

kwds : keyword arguments, optional

These can include “loc” and “scale”, as well as other keyword arguments relevant for a given distribution.

statsmodels.sandbox.distributions.transformed.TransfTwo_gen.nnlf

TransfTwo_gen.nnlf(theta, x)

Return negative loglikelihood function

Notes

This is -sum(log pdf(x, theta), axis=0) where theta are the parameters (including loc and scale).

statsmodels.sandbox.distributions.transformed.TransfTwo_gen.pdf

TransfTwo_gen.pdf(x, *args, **kwds)

Probability density function at x of the given RV.

Parameters

x : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns

pdf : ndarray

Probability density function evaluated at x
TransfTwo_gen.ppf(q, *args, **kwds)
Percent point function (inverse of cdf) at q of the given RV.

Parameters
- q : array_like
  lower tail probability
- arg1, arg2, arg3,... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- loc : array_like, optional
  Location parameter (default=0)
- scale : array_like, optional
  Scale parameter (default=1)

Returns
x : array_like
quantile corresponding to the lower tail probability q.

TransfTwo_gen.rvs(*args, **kwds)
Random variates of given type.

Parameters
- arg1, arg2, arg3,... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- loc : array_like, optional
  Location parameter (default=0).
- scale : array_like, optional
  Scale parameter (default=1).
- size : int or tuple of ints, optional
  Defining number of random variates (default=1).

Returns
rvs : ndarray or scalar
Random variates of given size.

TransfTwo_gen.sf(x, *args, **kwds)
Survival function (1-cdf) at x of the given RV.

Parameters
- x : array_like
  quantiles
- arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc**: array_like, optional
location parameter (default=0)

**scale**: array_like, optional
scale parameter (default=1)

**Returns**

**sf**: array_like
Survival function evaluated at x

### statsmodels.sandbox.distributions.transformed.TransfTwo_gen.stats

TransfTwo_gen.stats(*args, **kwds)
Some statistics of the given RV

**Parameters**

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc**: array_like, optional
location parameter (default=0)

**scale**: array_like, optional
scale parameter (default=1)

**moments**: str, optional
composed of letters ['mvsk'] defining which moments to compute: ‘m’ = mean, ‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default='mv')

**Returns**

**stats**: sequence
of requested moments.

### statsmodels.sandbox.distributions.transformed.TransfTwo_gen.std

TransfTwo_gen.std(*args, **kwds)
Standard deviation of the distribution.

**Parameters**

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc**: array_like, optional
location parameter (default=0)

**scale**: array_like, optional
scale parameter (default=1)

**Returns**

**std**: float
standard deviation of the distribution
Statsmodels Documentation, Release 0.6.0

**transformation**: distributions

### TransfTwo_gen.var

**transformation**: distributions

**TransfTwo_gen.var**(*args, **kwds)

Variance of the distribution

**Parameters**

- **arg1, arg2, arg3,...** : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  - location parameter (default=0)
- **scale** : array_like, optional
  - scale parameter (default=1)

**Returns**

- **var** : float
  - the variance of the distribution

### Transf_gen

**class**

**statsmodels.sandbox.distributions.transformed.Transf_gen**(*kls, func, funcinv, *args, **kwargs)

- a class for non-linear monotonic transformation of a continuous random variable

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdf</td>
<td>Cumulative distribution function of the given RV.</td>
</tr>
<tr>
<td>entropy</td>
<td>Differential entropy of the RV.</td>
</tr>
<tr>
<td>est_loc_scale</td>
<td>Calculate expected value of a function with respect to the distribution</td>
</tr>
<tr>
<td>fit</td>
<td>Return MLEs for shape, location, and scale parameters from data.</td>
</tr>
<tr>
<td>fit_loc_scale</td>
<td>Estimate loc and scale parameters from data using 1st and 2nd moments.</td>
</tr>
<tr>
<td>freeze</td>
<td>Freeze the distribution for the given arguments.</td>
</tr>
<tr>
<td>interval</td>
<td>Confidence interval with equal areas around the median.</td>
</tr>
<tr>
<td>isf</td>
<td>Inverse survival function at q of the given RV.</td>
</tr>
<tr>
<td>logcdf</td>
<td>Log of the cumulative distribution function at x of the given RV.</td>
</tr>
<tr>
<td>logpdf</td>
<td>Log of the probability density function at x of the given RV.</td>
</tr>
<tr>
<td>logsf</td>
<td>Log of the survival function of the given RV.</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of the distribution</td>
</tr>
<tr>
<td>median</td>
<td>Median of the distribution</td>
</tr>
<tr>
<td>moment</td>
<td>n'th order non-central moment of distribution.</td>
</tr>
<tr>
<td>nnlf</td>
<td>Return negative loglikelihood function ..</td>
</tr>
<tr>
<td>pdf</td>
<td>Probability density function at x of the given RV.</td>
</tr>
<tr>
<td>ppf</td>
<td>Percent point function (inverse of cdf) at q of the given RV.</td>
</tr>
<tr>
<td>rvs</td>
<td>Random variates of given type.</td>
</tr>
<tr>
<td>sf</td>
<td>Survival function (1-cdf) at x of the given RV.</td>
</tr>
<tr>
<td>stats</td>
<td>Some statistics of the given RV</td>
</tr>
<tr>
<td>std</td>
<td>Standard deviation of the distribution.</td>
</tr>
<tr>
<td>var</td>
<td>Variance of the distribution.</td>
</tr>
</tbody>
</table>
statsmodels.sandbox.distributions.transformed.Transf_gen.cdf

Transf_gen.cdf(x, *args, **kws)
Cumulative distribution function of the given RV.

Parameters  x : array_like
    quantiles

    arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)

    loc : array_like, optional
    location parameter (default=0)

    scale : array_like, optional
    scale parameter (default=1)

Returns  cdf : ndarray
    Cumulative distribution function evaluated at x

statsmodels.sandbox.distributions.transformed.Transf_gen.entropy

Transf_gen.entropy(*args, **kws)
Differential entropy of the RV.

Parameters  arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information).

    loc : array_like, optional
    Location parameter (default=0).

    scale : array_like, optional
    Scale parameter (default=1).

statsmodels.sandbox.distributions.transformed.Transf_gen.est_loc_scale

Transf_gen.est_loc_scale(*args, **kws)
est_loc_scale is deprecated!
This function is deprecated, use self.fit_loc_scale(data) instead.

statsmodels.sandbox.distributions.transformed.Transf_gen.expect

Transf_gen.expect(func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kws)
Calculate expected value of a function with respect to the distribution
The expected value of a function \( f(x) \) with respect to a distribution dist is defined as:
\begin{equation}
E[x] = \int f(x) \ast dist.pdf(x)
\end{equation}

**Parameters**

- **func**: callable, optional
  - Function for which integral is calculated. Takes only one argument. The default is the identity mapping \( f(x) = x \).

- **args**: tuple, optional
  - Argument (parameters) of the distribution.

- **lb, ub**: scalar, optional
  - Lower and upper bound for integration. Default is set to the support of the distribution.

- **conditional**: bool, optional
  - If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.

**Returns**

- **expect**: float
  - The calculated expected value.

**Notes**

The integration behavior of this function is inherited from `integrate.quad`.

```python
statsmodels.sandbox.distributions.transformed.Transf_gen.fit
```

**Transf_gen.fit**

Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, `self._fitstart(data)` is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments `f0, f1, ..., fn` (for shape parameters) and `floc` and `fscale` (for location and scale parameters, respectively).

- **Parameters**
  - **data**: array_like
    - Data to use in calculating the MLEs.
  - **args**: floats, optional
    - Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to `__fitstart(data)`). No default value.
  - **kwds**: floats, optional
    - Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:
      - `f0...fn`: hold respective shape parameters fixed.
floc : hold location parameter fixed to specified value.
fscale : hold scale parameter fixed to specified value.

optimizer [The optimizer to use. The optimizer must take func,] and starting
position as the first two arguments, plus args (for extra arguments to pass to the
function to be optimized) and disp=0 to suppress output as keyword arguments.

Returns shape, loc, scale : tuple of floats
MLEs for any shape statistics, followed by those for location and scale.

Notes
This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of
range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may
only be locally optimal, or the optimization may fail altogether.

statsmodels.sandbox.distributions.transformed.Transf_gen.fit_loc_scale

Transf_gen.fit_loc_scale(data, *args)
Estimate loc and scale parameters from data using 1st and 2nd moments.

Parameters data : array_like
Data to fit.
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object
for more information).

Returns Lhat : float
Estimated location parameter for the data.
Shat : float
Estimated scale parameter for the data.

statsmodels.sandbox.distributions.transformed.Transf_gen.freeze

Transf_gen.freeze(*args, **kwds)
Freeze the distribution for the given arguments.

Parameters arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution. Should include all the non-optional
arguments, may include loc and scale.

Returns rv_frozen : rv_frozen instance
The frozen distribution.

statsmodels.sandbox.distributions.transformed.Transf_gen.interval

Transf_gen.interval(alpha, *args, **kwds)
Confidence interval with equal areas around the median.
Parameters alpha : array_like of float

Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].

arg1, arg2, ... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

location parameter, Default is 0.

scale : array_like, optional

scale parameter, Default is 1.

Returns a, b : ndarray of float

end-points of range that contain 100 * alpha % of the rv’s possible values.

statsmodels.sandbox.distributions.transformed.Transf_gen.isf

Transf_gen.isf(q, *args, **kwds)
Inverse survival function at q of the given RV.

Parameters q : array_like

upper tail probability

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns x : ndarray or scalar

Quantile corresponding to the upper tail probability q.

statsmodels.sandbox.distributions.transformed.Transf_gen.logcdf

Transf_gen.logcdf(x, *args, **kwds)
Log of the cumulative distribution function at x of the given RV.

Parameters x : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns logcdf : array_like
Log of the cumulative distribution function evaluated at x

statsmodels.sandbox.distributions.transformed.Transf_gen.logpdf

Transf_gen.logpdf(x, *args, **kwds)
Log of the probability density function at x of the given RV.
This uses a more numerically accurate calculation if available.

Parameters x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns logpdf : array_like
Log of the probability density function evaluated at x

statsmodels.sandbox.distributions.transformed.Transf_gen.logsf

Transf_gen.logsf(x, *args, **kwds)
Log of the survival function of the given RV.
Returns the log of the “survival function,” defined as (1 - cdf), evaluated at x.

Parameters x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns logsf : ndarray
Log of the survival function evaluated at x.
**statsmodels.sandbox.distributions.transformed.Transf_gen.mean**

`Transf_gen.mean(*args, **kwds)`

Mean of the distribution

Parameters `arg1, arg2, arg3,...` : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

`loc` : array_like, optional

Location parameter (default=0)

`scale` : array_like, optional

Scale parameter (default=1)

Returns `mean` : float

The mean of the distribution

**statsmodels.sandbox.distributions.transformed.Transf_gen.median**

`Transf_gen.median(*args, **kwds)`

Median of the distribution.

Parameters `arg1, arg2, arg3,...` : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

`loc` : array_like, optional

Location parameter, Default is 0.

`scale` : array_like, optional

Scale parameter, Default is 1.

Returns `median` : float

The median of the distribution.

See also:

- `stats.distributions.rv_discrete.ppf` Inverse of the CDF

**statsmodels.sandbox.distributions.transformed.Transf_gen.moment**

`Transf_gen.moment(n, *args, **kwds)`

n'th order non-central moment of distribution.

Parameters `n` : int, n>=1

Order of moment.

`arg1, arg2, arg3,...` : float

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

`kwds` : keyword arguments, optional
These can include “loc” and “scale”, as well as other keyword arguments relevant for a given distribution.

```
statsmodels.sandbox.distributions.transformed.Transf_gen.nnlf
```

**Transf_gen.nnlf**(theta, x)

Return negative loglikelihood function

**Notes**

This is \(-\text{sum}(\log \text{ pdf}(x, \theta), \text{axis}=0)\) where \(\theta\) are the parameters (including loc and scale).

```
statsmodels.sandbox.distributions.transformed.Transf_gen.pdf
```

**Transf_gen.pdf**(x, \*args, \**kwds)

Probability density function at x of the given RV.

**Parameters**

- **x**: array_like
  - quantiles
- **arg1, arg2, arg3,..**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

**Returns**

- **pdf**: ndarray
  - Probability density function evaluated at x

```
statsmodels.sandbox.distributions.transformed.Transf_gen.ppf
```

**Transf_gen.ppf**(q, \*args, \**kwds)

Percent point function (inverse of cdf) at q of the given RV.

**Parameters**

- **q**: array_like
  - lower tail probability
- **arg1, arg2, arg3,..**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
scale parameter (default=1)

Returns x : array_like
    quantile corresponding to the lower tail probability q.

statsmodels.sandbox.distributions.transformed.Transf_gen.rvs

Transf_gen.rvs(*args, **kwds)
Random variates of given type.

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
    Location parameter (default=0).
scale : array_like, optional
    Scale parameter (default=1).
size : int or tuple of ints, optional
    Defining number of random variates (default=1).

Returns rvs : ndarray or scalar
    Random variates of given size.

statsmodels.sandbox.distributions.transformed.Transf_gen.sf

Transf_gen.sf(x, *args, **kwds)
Survival function (1-cdf) at x of the given RV.

Parameters x : array_like
    quantiles
arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
    location parameter (default=0)
scale : array_like, optional
    scale parameter (default=1)

Returns sf : array_like
    Survival function evaluated at x
statsmodels.sandbox.distributions.transformed.Transf_gen.stats

**Transf_gen.stats** (*args, **kwds*)

Some statistics of the given RV

**Parameters**

- **arg1, arg2, arg3,**... : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  - location parameter (default=0)
- **scale** : array_like, optional
  - scale parameter (default=1)
- **moments** : str, optional
  - composed of letters ['mvsk'] defining which moments to compute: ‘m’ = mean, ‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default=’mv’)

**Returns**

- **stats** : sequence
  - of requested moments.

statsmodels.sandbox.distributions.transformed.Transf_gen.std

**Transf_gen.std** (*args, **kwds*)

Standard deviation of the distribution.

**Parameters**

- **arg1, arg2, arg3,**... : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  - location parameter (default=0)
- **scale** : array_like, optional
  - scale parameter (default=1)

**Returns**

- **std** : float
  - standard deviation of the distribution

statsmodels.sandbox.distributions.transformed.Transf_gen.var

**Transf_gen.var** (*args, **kwds*)

Variance of the distribution

**Parameters**

- **arg1, arg2, arg3,**... : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  - location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns var : float
the variance of the distribution

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen**

class statsmodels.sandbox.distributions.transformed.ExpTransf_gen(kls, *args, **kwargs)

Distribution based on log/exp transformation
the constructor can be called with a distribution class and generates the distribution of the transformed random variable

Methods

cdf(x, *args, **kwds) Cumulative distribution function of the given RV.
entropy(*args, **kwds) Differential entropy of the RV.
est_loc_scale(*args, **kwds) est_loc_scale is deprecated!
expect([func, args, loc, scale, lb, ub, ...]) Calculate expected value of a function with respect to the distribution
fit(data, *args, **kwds) Return MLEs for shape, location, and scale parameters from data.
fit_loc_scale(data, *args) Estimate loc and scale parameters from data using 1st and 2nd moments.
freeze(*args, **kwds) Freeze the distribution for the given arguments.
interval(alpha, *args, **kwds) Confidence interval with equal areas around the median.
isf(q, *args, **kwds) Inverse survival function at q of the given RV.
logcdf(x, *args, **kwds) Log of the cumulative distribution function at x of the given RV.
logpdf(x, *args, **kwds) Log of the probability density function at x of the given RV.
logsf(x, *args, **kwds) Log of the survival function of the given RV.
mean(*args, **kwds) Mean of the distribution
median(*args, **kwds) Median of the distribution.
moment(n, *args, **kwds) n’th order non-central moment of distribution.
nllf(theta, x) Return negative loglikelihood function ..
pdf(x, *args, **kwds) Probability density function at x of the given RV.
ppf(q, *args, **kwds) Percent point function (inverse of cdf) at q of the given RV.
rvs(*args, **kwds) Random variates of given type.
sf(x, *args, **kwds) Survival function (1-cdf) at x of the given RV.
stats(*args, **kwds) Some statistics of the given RV
std(*args, **kwds) Standard deviation of the distribution.
var(*args, **kwds) Variance of the distribution

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.cdf**

ExpTransf_gen.cdf(x, *args, **kwds)
Cumulative distribution function of the given RV.

Parameters x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc**: array_like, optional
  location parameter (default=0)

**scale**: array_like, optional
  scale parameter (default=1)

**Returns**

**cdf**: ndarray
  Cumulative distribution function evaluated at \( x \)

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.entropy**

ExpTransf_gen.entropy(*args, **kwds)
  Differential entropy of the RV.

  **Parameters**
  
  **arg1, arg2, arg3,...**: array_like
  
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).

  **loc**: array_like, optional
  
  Location parameter (default=0).

  **scale**: array_like, optional
  
  Scale parameter (default=1).

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.est_loc_scale**

ExpTransf_gen.est_loc_scale(*args, **kwds)
  est_loc_scale is deprecated!

  This function is deprecated, use self.fit_loc_scale(data) instead.

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.expect**

ExpTransf_gen.expect(func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)
  Calculate expected value of a function with respect to the distribution

  The expected value of a function \( f(x) \) with respect to a distribution \( \text{dist} \) is defined as:

  \[
  \mathbb{E}[x] = \int_{\text{lb}}^{\text{ub}} f(x) \times \text{dist.pdf}(x) \, dx
  \]

  **Parameters**
  
  **func**: callable, optional
  
  Function for which integral is calculated. Takes only one argument. The default is the identity mapping \( f(x) = x \).

  **args** : tuple, optional
  
  Argument (parameters) of the distribution.
lb, ub : scalar, optional

Lower and upper bound for integration. default is set to the support of the distribution.

conditional : bool, optional

If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.

Additional keyword arguments are passed to the integration routine.

Returns

expect : float

The calculated expected value.

Notes

The integration behavior of this function is inherited from integrate.quad.

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.fit**

ExpTransf_gen.fit(data, *args, **kwds)

Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, self._fitstart(data) is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments f0, f1, ..., fn (for shape parameters) and floc and fscale (for location and scale parameters, respectively).

Parameters

data : array_like

Data to use in calculating the MLEs.

args : floats, optional

Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to _fitstart(data)). No default value.

kwds : floats, optional

Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:

f0...fn : hold respective shape parameters fixed.
floc : hold location parameter fixed to specified value.
fscale : hold scale parameter fixed to specified value.

optimizer [The optimizer to use. The optimizer must take func, and starting position as the first two arguments, plus args (for extra arguments to pass to the function to be optimized) and disp=0 to suppress output as keyword arguments.

Returns

shape, loc, scale : tuple of floats

MLEs for any shape statistics, followed by those for location and scale.
Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.fit_loc_scale

ExpTransf_gen.fit_loc_scale(data, *args)

Estimate loc and scale parameters from data using 1st and 2nd moments.

Parameters

- data: array_like
  Data to fit.
- arg1, arg2, arg3,...: array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).

Returns

- Lhat: float
  Estimated location parameter for the data.
- Shat: float
  Estimated scale parameter for the data.

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.freeze

ExpTransf_gen.freeze(*args, **kwds)

Freeze the distribution for the given arguments.

Parameters

- arg1, arg2, arg3,...: array_like
  The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include loc and scale.

Returns

- rv_frozen: rv_frozen instance
  The frozen distribution.

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.interval

ExpTransf_gen.interval(alpha, *args, **kwds)

Confidence interval with equal areas around the median.

Parameters

- alpha: array_like of float
  Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].
- arg1, arg2, ...: array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- loc: array_like, optional
  location parameter, Default is 0.
scale : array_like, optional
        scale parameter, Default is 1.

Returns a, b : ndarray of float
        end-points of range that contain $100 \times \alpha \%$ of the rv’s possible values.

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.isf

ExpTransf_gen.isf(q, *args, **kwds)
        Inverse survival function at q of the given RV.

Parameters q : array_like
    upper tail probability

arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns x : ndarray or scalar
    Quantile corresponding to the upper tail probability q.

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.logcdf

ExpTransf_gen.logcdf(x, *args, **kwds)
        Log of the cumulative distribution function at x of the given RV.

Parameters x : array_like
    quantiles

arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns logcdf : array_like
    Log of the cumulative distribution function evaluated at x
**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.logpdf**

ExpTransf_gen.logpdf(x, *args, **kwds)

Log of the probability density function at x of the given RV. This uses a more numerically accurate calculation if available.

**Parameters**
- **x**: array_like
  - quantiles
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

**Returns**
- **logpdf**: array_like
  - Log of the probability density function evaluated at x

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.logsf**

ExpTransf_gen.logsf(x, *args, **kwds)

Log of the survival function of the given RV.

Returns the log of the “survival function,” defined as (1 - cdf), evaluated at x.

**Parameters**
- **x**: array_like
  - quantiles
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

**Returns**
- **logsf**: ndarray
  - Log of the survival function evaluated at x.

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.mean**

ExpTransf_gen.mean(*args, **kwds)

Mean of the distribution

**Parameters**
- **arg1, arg2, arg3,...**: array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

```
loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)
```

Returns `mean` : float

the mean of the distribution

```python
statsmodels.sandbox.distributions.transformed.ExpTransf_gen.median
```

```
ExpTransf_gen.median(*args, **kwds)
```

Median of the distribution.

**Parameters**

```
arg1, arg2, arg3,... : array_like
```

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

```
loc : array_like, optional
```

Location parameter, Default is 0.

```
scale : array_like, optional
```

Scale parameter, Default is 1.

Returns `median` : float

The median of the distribution.

See also:

```
stats.distributions.rv_discrete.ppf
```

Inverse of the CDF

```python
statsmodels.sandbox.distributions.transformed.ExpTransf_gen.moment
```

```
ExpTransf_gen.moment(n, *args, **kwds)
```

n’th order non-central moment of distribution.

**Parameters**

```
n : int, n>=1
```

Order of moment.

```
arg1, arg2, arg3,... : float
```

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

```
kwds : keyword arguments, optional
```

These can include “loc” and “scale”, as well as other keyword arguments relevant for a given distribution.
Statsmodels Documentation, Release 0.6.0

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.nnlf**

ExpTransf_gen.nnlf(theta, x)

Return negative loglikelihood function

**Notes**

This is \(-\text{sum}(\log \text{ pdf}(x, \theta), \text{ axis}=0)\) where \(\theta\) are the parameters (including loc and scale).

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.pdf**

ExpTransf_gen.pdf(x, *args, **kwds)

Probability density function at \(x\) of the given RV.

**Parameters**

- **x**: array_like
  quantiles
- **arg1, arg2, arg3,...**: array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  location parameter (default=0)
- **scale**: array_like, optional
  scale parameter (default=1)

**Returns**

- **pdf**: ndarray
  Probability density function evaluated at \(x\)

**statsmodels.sandbox.distributions.transformed.ExpTransf_gen.ppf**

ExpTransf_gen.ppf(q, *args, **kwds)

Percent point function (inverse of cdf) at \(q\) of the given RV.

**Parameters**

- **q**: array_like
  lower tail probability
- **arg1, arg2, arg3,...**: array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  location parameter (default=0)
- **scale**: array_like, optional
  scale parameter (default=1)

**Returns**

- **x**: array_like
  quantile corresponding to the lower tail probability \(q\).
statsmodels.sandbox.distributions.transformed.ExpTransf_gen.rvs

ExpTransf_gen.rvs(*args, **kwds)
Random variates of given type.

Parameters arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
Location parameter (default=0).
scale : array_like, optional
Scale parameter (default=1).
size : int or tuple of ints, optional
Defining number of random variates (default=1).

Returns rvs : ndarray or scalar
Random variates of given size.

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.sf

ExpTransf_gen.sf(x, *args, **kwds)
Survival function (1-cdf) at x of the given RV.

Parameters x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns sf : array_like
Survival function evaluated at x

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.stats

ExpTransf_gen.stats(*args, **kwds)
Some statistics of the given RV

Parameters arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional

3.13. Distributions 935
location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

moments : str, optional
    composed of letters [‘mvsk’] defining which moments to compute: ‘m’ = mean,
    ‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default=’mv’)

Returns stats : sequence
    of requested moments.

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.std

ExpTransf_gen.std(*args, **kwds)
    Standard deviation of the distribution.

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns std : float
    standard deviation of the distribution

statsmodels.sandbox.distributions.transformed.ExpTransf_gen.var

ExpTransf_gen.var(*args, **kwds)
    Variance of the distribution

Parameters arg1, arg2, arg3,... : array_like
    The shape parameter(s) for the distribution (see docstring of the instance object
    for more information)

loc : array_like, optional
    location parameter (default=0)

scale : array_like, optional
    scale parameter (default=1)

Returns var : float
    the variance of the distribution
class statsmodels.sandbox.distributions.transformed.LogTransf_gen(kls, *args, **kwds)

Distribution based on log/exp transformation
the constructor can be called with a distribution class and generates the distribution of the transformed random variable

Methods

cdf(x, *args, **kwds) Cumulative distribution function of the given RV.
entropy(*args, **kwds) Differential entropy of the RV.
est_loc_scale(*args, **kwds) est_loc_scale is deprecated!
expect(func, args, loc, scale, lb, ub, ...) Calculate expected value of a function with respect to the distribution
fit(data, *args, **kwds) Return MLEs for shape, location, and scale parameters from data.
fit_loc_scale(data, *args) Estimate loc and scale parameters from data using 1st and 2nd moments.
freeze(*args, **kwds) Freeze the distribution for the given arguments.
interval(alpha, *args, **kwds) Confidence interval with equal areas around the median.
isf(q, *args, **kwds) Inverse survival function at q of the given RV.
logcdf(x, *args, **kwds) Log of the cumulative distribution function at x of the given RV.
logpdf(x, *args, **kwds) Log of the probability density function at x of the given RV.
logsf(x, *args, **kwds) Log of the survival function of the given RV.
mean(*args, **kwds) Mean of the distribution
median(*args, **kwds) Median of the distribution.
moment(n, *args, **kwds) n’th order non-central moment of distribution.
nnlf(theta, x) Return negative loglikelihood function..
pdf(x, *args, **kwds) Probability density function at x of the given RV.
ppf(q, *args, **kwds) Percent point function (inverse of cdf) at q of the given RV.
rvs(*args, **kwds) Random variates of given type.
.sf(x, *args, **kwds) Survival function (1-cdf) at x of the given RV.
stats(*args, **kwds) Some statistics of the given RV
std(*args, **kwds) Standard deviation of the distribution.
var(*args, **kwds) Variance of the distribution

parameters

3.13. Distributions
**Returns**

```
cdf : ndarray
```

Cumulative distribution function evaluated at x

---

**statsmodels.sandbox.distributions.transformed.LogTransf_gen.entropy**

```
LogTransf_gen.entropy(*args, **kwds)
```

Differential entropy of the RV.

**Parameters**

```
arg1, arg2, arg3,... : array_like
```

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

```
loc : array_like, optional
```

Location parameter (default=0).

```
scale : array_like, optional
```

Scale parameter (default=1).

---

**statsmodels.sandbox.distributions.transformed.LogTransf_gen.est_loc_scale**

```
LogTransf_gen(est_loc_scale(*args, **kwds)
```

*est_loc_scale* is deprecated!

This function is deprecated, use self.fit_loc_scale(data) instead.

---

**statsmodels.sandbox.distributions.transformed.LogTransf_gen.expect**

```
LogTransf_gen.expect(func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)
```

Calculate expected value of a function with respect to the distribution

The expected value of a function $f(x)$ with respect to a distribution $dist$ is defined as:

```
\begin{align*}
\text{E}[x] &= \int f(x) \cdot dist.pdf(x) \\
\text{lbound} \quad \text{ubound}
\end{align*}
```

**Parameters**

```
func : callable, optional
```

Function for which integral is calculated. Takes only one argument. The default is the identity mapping $f(x) = x$.

```
args : tuple, optional
```

Argument (parameters) of the distribution.

```
lb, ub : scalar, optional
```

Lower and upper bound for integration. Default is set to the support of the distribution.

```
conditional : bool, optional
```

If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.
Additional keyword arguments are passed to the integration routine. :

Returns `expect` : float

The calculated expected value.

Notes

The integration behavior of this function is inherited from `integrate.quad`.

statsmodels.sandbox.distributions.transformed.LogTransf_gen.fit

LogTransf_gen.fit(data, *args, **kwds)

Return MLEs for shape, location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, `self._fitstart(data)` is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments `f0, f1, ..., fn` (for shape parameters) and `floc` and `fscale` (for location and scale parameters, respectively).

Parameters `data` : array_like

Data to use in calculating the MLEs.

`args` : floats, optional

Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to `self._fitstart(data)`). No default value.

`kwds` : floats, optional

Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:

- `f0...fn` : hold respective shape parameters fixed.
- `floc` : hold location parameter fixed to specified value.
- `fscale` : hold scale parameter fixed to specified value.

`optimizer` [The optimizer to use. The optimizer must take func[,] and starting position as the first two arguments, plus args (for extra arguments to pass to the function to be optimized) and disp=0 to suppress output as keyword arguments.

Returns `shape, loc, scale` : tuple of floats

MLEs for any shape statistics, followed by those for location and scale.

Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.
**statsmodels.sandbox.distributions.transformed.LogTransf_gen.fit_loc_scale**

LogTransf_gen.fit_loc_scale(data, *args)

Estimate loc and scale parameters from data using 1st and 2nd moments.

**Parameters**

- **data** : array_like
  - Data to fit.

- **arg1, arg2, arg3,...** : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).

**Returns**

- **Lhat** : float
  - Estimated location parameter for the data.

- **Shat** : float
  - Estimated scale parameter for the data.

**statsmodels.sandbox.distributions.transformed.LogTransf_gen.freeze**

LogTransf_gen.freeze(*args, **kwds)

Freeze the distribution for the given arguments.

**Parameters**

- **arg1, arg2, arg3,...** : array_like
  - The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include `loc` and `scale`.

**Returns**

- **rv_frozen** : rv_frozen instance
  - The frozen distribution.

**statsmodels.sandbox.distributions.transformed.LogTransf_gen.interval**

LogTransf_gen.interval(alpha, *args, **kwds)

Confidence interval with equal areas around the median.

**Parameters**

- **alpha** : array_like of float
  - Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].

- **arg1, arg2, ...** : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).

- **loc** : array_like, optional
  - location parameter, Default is 0.

- **scale** : array_like, optional
  - scale parameter, Default is 1.

**Returns**

- **a, b** : ndarray of float
  - end-points of range that contain $100 \times \alpha$ % of the rv's possible values.
LogTransf_gen.isf(q, *args, **kwds)
Inverse survival function at q of the given RV.

Parameters  
q : array_like  
   upper tail probability

arg1, arg2, arg3,... : array_like  
The shape parameter(s) for the distribution (see docstring of the instance object
   for more information)

loc : array_like, optional  
   location parameter (default=0)

scale : array_like, optional  
   scale parameter (default=1)

Returns  
x : ndarray or scalar  
   Quantile corresponding to the upper tail probability q.

LogTransf_gen.logcdf(x, *args, **kwds)
Log of the cumulative distribution function at x of the given RV.

Parameters  
x : array_like  
   quantiles

arg1, arg2, arg3,... : array_like  
The shape parameter(s) for the distribution (see docstring of the instance object
   for more information)

loc : array_like, optional  
   location parameter (default=0)

scale : array_like, optional  
   scale parameter (default=1)

Returns  
logcdf : array_like  
   Log of the cumulative distribution function evaluated at x

LogTransf_gen.logpdf(x, *args, **kwds)
Log of the probability density function at x of the given RV.

This uses a more numerically accurate calculation if available.

Parameters  
x : array_like  
   quantiles

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

**loc** : array_like, optional
location parameter (default=0)

**scale** : array_like, optional
scale parameter (default=1)

**Returns**  
**logpdf** : array_like
Log of the probability density function evaluated at x

```
statsmodels.sandbox.distributions.transformed.LogTransf_gen.logsf
```

LogTransf_gen.logsf(x, *args, **kwds)
Log of the survival function of the given RV.

Returns the log of the “survival function,” defined as \(1 - \text{cdf}\), evaluated at \(x\).

**Parameters**

- **x** : array_like
  quantiles
- **arg1, arg2, arg3,**... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  location parameter (default=0)
- **scale** : array_like, optional
  scale parameter (default=1)

**Returns**  
**logsf** : ndarray
Log of the survival function evaluated at \(x\).

```
statsmodels.sandbox.distributions.transformed.LogTransf_gen.mean
```

LogTransf_gen.mean(*args, **kwds)
Mean of the distribution

**Parameters**

- **arg1, arg2, arg3,**... : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  location parameter (default=0)
- **scale** : array_like, optional
  scale parameter (default=1)

**Returns**  
**mean** : float
the mean of the distribution
LogTransf_gen.median(*args, **kwds)
Median of the distribution.

Parameters arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional
Location parameter, Default is 0.

scale : array_like, optional
Scale parameter, Default is 1.

Returns median : float
The median of the distribution.

See also:
stats.distributions.rv_discrete.ppf Inverse of the CDF

LogTransf_gen.moment(n, *args, **kwds)
n’th order non-central moment of distribution.

Parameters n : int, n>=1
Order of moment.

arg1, arg2, arg3,... : float
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

kwds : keyword arguments, optional
These can include “loc” and “scale”, as well as other keyword arguments relevant for a given distribution.

LogTransf_gen.nnlf(theta, x)
Return negative loglikelihood function

Notes
This is -sum(log pdf(x, theta), axis=0) where theta are the parameters (including loc and scale).
LogTransf_gen.pdf(x,*args,**kwds)
Probability density function at x of the given RV.

Parameters

- **x**: array_like
  - quantiles
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

Returns

- **pdf**: ndarray
  - Probability density function evaluated at x

LogTransf_gen.ppf(q,*args,**kwds)
Percent point function (inverse of cdf) at q of the given RV.

Parameters

- **q**: array_like
  - lower tail probability
- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc**: array_like, optional
  - location parameter (default=0)
- **scale**: array_like, optional
  - scale parameter (default=1)

Returns

- **x**: array_like
  - quantile corresponding to the lower tail probability q.

LogTransf_gen.rvs(*args,**kwds)
Random variates of given type.

Parameters

- **arg1, arg2, arg3,...**: array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information).
- **loc**: array_like, optional
Location parameter (default=0).

**scale** : array_like, optional
Scale parameter (default=1).

**size** : int or tuple of ints, optional
Defining number of random variates (default=1).

**Returns**

**rvs** : ndarray or scalar
Random variates of given size.

### statsmodels.sandbox.distributions.transformed.LogTransf_gen.sf

**LogTransf_gen.sf** *(x, *args, **kwds)*
Survival function (1-cdf) at x of the given RV.

**Parameters**

- **x** : array_like
  quantiles
- **arg1, arg2, arg3,...** : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  location parameter (default=0)
- **scale** : array_like, optional
  scale parameter (default=1)

**Returns**

**sf** : array_like
Survival function evaluated at x

### statsmodels.sandbox.distributions.transformed.LogTransf_gen.stats

**LogTransf_gen.stats** *(*args, **kwds)*
Some statistics of the given RV

**Parameters**

- **arg1, arg2, arg3,...** : array_like
  The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- **loc** : array_like, optional
  location parameter (default=0)
- **scale** : array_like, optional
  scale parameter (default=1)
- **moments** : str, optional
  composed of letters ['mvsk'] defining which moments to compute: ‘m’ = mean, ‘v’ = variance, ‘s’ = (Fisher’s) skew, ‘k’ = (Fisher’s) kurtosis. (default='mv')

**Returns**

**stats** : sequence
of requested moments.

**statsmodels.sandbox.distributions.transformed.LogTransf_gen.std**

\[ \text{LogTransf_gen.std}(*args, **kwds) \]

Standard deviation of the distribution.

**Parameters**

- `arg1, arg2, arg3,...` : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- `loc` : array_like, optional
  - location parameter (default=0)
- `scale` : array_like, optional
  - scale parameter (default=1)

**Returns**

- `std` : float
  - standard deviation of the distribution

**statsmodels.sandbox.distributions.transformed.LogTransf_gen.var**

\[ \text{LogTransf_gen.var}(*args, **kwds) \]

Variance of the distribution

**Parameters**

- `arg1, arg2, arg3,...` : array_like
  - The shape parameter(s) for the distribution (see docstring of the instance object for more information)
- `loc` : array_like, optional
  - location parameter (default=0)
- `scale` : array_like, optional
  - scale parameter (default=1)

**Returns**

- `var` : float
  - the variance of the distribution

**statsmodels.sandbox.distributions.transformed.SquareFunc**

**class**

\[ \text{statsmodels.sandbox.distributions.transformed.SquareFunc} \]

- class to hold quadratic function with inverse function and derivative

  using instance methods instead of class methods, if we want extension to parameterized function

**Methods**

- `derivminus(x)`
- `derivplus(x)`

Continued on next page
Table 3.267 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverseminus(x)</td>
</tr>
<tr>
<td>inverseplus(x)</td>
</tr>
<tr>
<td>squarefunc(x)</td>
</tr>
</tbody>
</table>

```
statsmodels.sandbox.distributions.transformed.SquareFunc.derivminus

SquareFunc.derivminus(x)
```

```
statsmodels.sandbox.distributions.transformed.SquareFunc.derivplus

SquareFunc.derivplus(x)
```

```
statsmodels.sandbox.distributions.transformed.SquareFunc.inverseminus

SquareFunc.inverseminus(x)
```

```
statsmodels.sandbox.distributions.transformed.SquareFunc.inverseplus

SquareFunc.inverseplus(x)
```

```
statsmodels.sandbox.distributions.transformed.SquareFunc.squarefunc

SquareFunc.squarefunc(x)
```

```
statsmodels.sandbox.distributions.transformed.absnormalg

statsmodels.sandbox.distributions.transformed.absnormalg = <statsmodels.sandbox.distributions.transformed.TransfTwo_gen object at 0x0818E250>

Distribution based on a non-monotonic (u- or hump-shaped transformation)
the constructor can be called with a distribution class, and functions that define the non-linear transformation.
and generates the distribution of the transformed random variable
Note: the transformation, it’s inverse and derivatives need to be fully specified: func, funcinvplus, funcinvminus,
derivplus, derivminus. Currently no numerical derivatives or inverse are calculated
This can be used to generate distribution instances similar to the distributions in scipy.stats.
```

```
statsmodels.sandbox.distributions.transformed.invdnormalg

statsmodels.sandbox.distributions.transformed.invdnormalg = <statsmodels.sandbox.distributions.transformed.Transf_gen object at 0x0792A210>
a class for non-linear monotonic transformation of a continuous random variable
```

```
statsmodels.sandbox.distributions.transformed.loggammaexpg

statsmodels.sandbox.distributions.transformed.loggammaexpg = <statsmodels.sandbox.distributions.transformed.Transf_gen object at 0x0818E510>
univariate distribution of a non-linear monotonic transformation of a random variable
```
statsmodels.sandbox.distributions.transformed.lognormalg

statsmodels.sandbox.distributions.transformed.lognormalg = <statsmodels.sandbox.distributions.transformed.Transf_gen object at 0x0818C9F0>

a class for non-linear monotonic transformation of a continuous random variable

statsmodels.sandbox.distributions.transformed.negsquarenormalg

statsmodels.sandbox.distributions.transformed.negsquarenormalg = <statsmodels.sandbox.distributions.transformed.TransfTwo_gen object at 0x08127750>

Distribution based on a non-monotonic (u- or hump-shaped transformation)

the constructor can be called with a distribution class, and functions that define the non-linear transformation.
and generates the distribution of the transformed random variable

Note: the transformation, it’s inverse and derivatives need to be fully specified: func, funcinvplus, funcinvminus,
derivplus, derivminus. Currently no numerical derivatives or inverse are calculated

This can be used to generate distribution instances similar to the distributions in scipy.stats.

statsmodels.sandbox.distributions.transformed.squarenormalg

statsmodels.sandbox.distributions.transformed.squarenormalg = <statsmodels.sandbox.distributions.transformed.TransfTwo_gen object at 0x082AA450>

Distribution based on a non-monotonic (u- or hump-shaped transformation)

the constructor can be called with a distribution class, and functions that define the non-linear transformation.
and generates the distribution of the transformed random variable

Note: the transformation, it’s inverse and derivatives need to be fully specified: func, funcinvplus, funcinvminus,
derivplus, derivminus. Currently no numerical derivatives or inverse are calculated

This can be used to generate distribution instances similar to the distributions in scipy.stats.

statsmodels.sandbox.distributions.transformed.squaretg

statsmodels.sandbox.distributions.transformed.squaretg = <statsmodels.sandbox.distributions.transformed.TransfTwo_gen object at 0x082AACB0>

Distribution based on a non-monotonic (u- or hump-shaped transformation)

the constructor can be called with a distribution class, and functions that define the non-linear transformation.
and generates the distribution of the transformed random variable

Note: the transformation, it’s inverse and derivatives need to be fully specified: func, funcinvplus, funcinvminus,
derivplus, derivminus. Currently no numerical derivatives or inverse are calculated

This can be used to generate distribution instances similar to the distributions in scipy.stats.

3.14 Graphics

3.14.1 Goodness of Fit Plots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gofplots.qqplot(data[, dist, distargs, a, ...])</td>
<td>Q-Q plot of the quantiles of x versus the quantiles/ppf of a distribution.</td>
</tr>
<tr>
<td>gofplots.qqline(ax, line[, x, y, dist, fmt])</td>
<td>Plot a reference line for a qqplot.</td>
</tr>
<tr>
<td>gofplots.qqplot_2samples(data1, data2[, ...])</td>
<td>Q-Q Plot of two samples’ quantiles.</td>
</tr>
<tr>
<td>gofplots.ProbPlot(data[, dist, fit, ...])</td>
<td>Class for convenient construction of Q-Q, P-P, and probability plots.</td>
</tr>
</tbody>
</table>
statsmodels.graphics.gofplots.qqplot

`statsmodels.graphics.gofplots.qqplot(data, dist=<scipy.stats.distributions.norm_gen object at 0x0534E8D0>, distargs=(), a=0, loc=0, scale=1, fit=False, line=None, ax=None)`

Q-Q plot of the quantiles of x versus the quantiles/ppf of a distribution.

Can take arguments specifying the parameters for dist or fit them automatically. (See fit under Parameters.)

**Parameters**

- **data**: array-like
  - 1d data array

- **dist**: A scipy.stats or statsmodels distribution
  - Compare x against dist. The default is scipy.stats.distributions.norm (a standard normal).

- **distargs**: tuple
  - A tuple of arguments passed to dist to specify it fully so dist.ppf may be called.

- **loc**: float
  - Location parameter for dist

- **a**: float
  - Offset for the plotting position of an expected order statistic, for example. The plotting positions are given by (i - a)/(nobs - 2*a + 1) for i in range(0,nobs+1)

- **scale**: float
  - Scale parameter for dist

- **fit**: boolean
  - If fit is false, loc, scale, and distargs are passed to the distribution. If fit is True then the parameters for dist are fit automatically using dist.fit. The quantiles are formed from the standardized data, after subtracting the fitted loc and dividing by the fitted scale.

- **line**: str {'45', 's', 'r', q'} or None
  - Options for the reference line to which the data is compared:
    - ‘45’ - 45-degree line
    - ‘s’ - standardized line, the expected order statistics are scaled by the standard deviation of the given sample and have the mean added to them
    - ‘r’ - A regression line is fit
    - ‘q’ - A line is fit through the quartiles.
    - None - by default no reference line is added to the plot.

- **ax**: Matplotlib AxesSubplot instance, optional
  - If given, this subplot is used to plot in instead of a new figure being created.

**Returns**

- **fig**: Matplotlib figure instance
  - If ax is None, the created figure. Otherwise the figure to which ax is connected.

**See also:**

- `scipy.stats.probplot`
Notes

Depends on matplotlib. If `fit` is True then the parameters are fit using the distribution’s fit() method.

Examples

```python
>>> import statsmodels.api as sm
>>> from matplotlib import pyplot as plt

>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> mod_fit = sm.OLS(data.endog, data.exog).fit()
>>> res = mod_fit.resid  # residuals
>>> fig = sm.qqplot(res)
>>> plt.show()

qqplot of the residuals against quantiles of t-distribution with 4 degrees of freedom:

```python
>>> import scipy.stats as stats
>>> fig = sm.qqplot(res, stats.t, distargs=(4,))
>>> plt.show()
```

qqplot against same as above, but with mean 3 and std 10:

```python
>>> fig = sm.qqplot(res, stats.t, distargs=(4,), loc=3, scale=10)
>>> plt.show()
```

Automatically determine parameters for t distribution including the loc and scale:

```python
>>> fig = sm.qqplot(res, stats.t, fit=True, line='45')
>>> plt.show()
```

The following plot displays some options, follow the link to see the code.
Theoretical Quantiles

Sample Quantiles

no keywords

line='s'

dist=stats.t, line='45', fit=True

dist=stats.t, line='45', fit=True
statsmodels.graphics.gofplots.qqline

Plot a reference line for a qqplot.

Parameters:
- `ax`: matplotlib axes instance
  - The axes on which to plot the line
- `line`: str (`'45'`, `'r'`, `'s'`, `'q'`)
  - Options for the reference line to which the data is compared:
    - `'45'` - 45-degree line
    - `'s'` - standardized line, the expected order statistics are scaled by the standard deviation of the given sample and have the mean added to them
    - `'r'` - A regression line is fit
    - `'q'` - A line is fit through the quartiles.
    - `None` - By default no reference line is added to the plot.

- `x`: array
  - X data for plot. Not needed if line is `'45'`.
- `y`: array
  - Y data for plot. Not needed if line is `'45'`. 
**dist**: scipy.stats.distribution

A scipy.stats distribution, needed if line is ‘q’.

---

**Notes**

There is no return value. The line is plotted on the given `ax`.

---

**statsmodels.graphics.gofplots.qqplot_2samples**

```python
statsmodels.graphics.gofplots.qqplot_2samples(data1, data2, xlabel=None, ylabel=None, line=None, ax=None)
```

Q-Q Plot of two samples’ quantiles.

Can take either two `ProbPlot` instances or two array-like objects. In the case of the latter, both inputs will be converted to `ProbPlot` instances using only the default values - so use `ProbPlot` instances if finer-grained control of the quantile computations is required.

**Parameters**

- `data1`, `data2` : array-like (1d) or `ProbPlot` instances

- `xlabel`, `ylabel` : str or None
  
  User-provided labels for the x-axis and y-axis. If None (default), other values are used.

- `line` : str {‘45’, ‘s’, ‘r’, ‘q’} or None
  
  Options for the reference line to which the data is compared:
  
  - ‘45’ - 45-degree line
  
  - ‘s’ - standardized line, the expected order statistics are scaled by the standard deviation of the given sample and have the mean added to them
  
  - ‘r’ - A regression line is fit
  
  - ‘q’ - A line is fit through the quartiles.
  
  - None - by default no reference line is added to the plot.

- `ax` : Matplotlib AxesSubplot instance, optional
  
  If given, this subplot is used to plot in instead of a new figure being created.

**Returns**

- `fig` : Matplotlib figure instance
  
  If `ax` is None, the created figure. Otherwise the figure to which `ax` is connected.

**See also**

- `scipy.stats.probplot`

---

**Notes**

1. Depends on matplotlib.

2. If `data1` and `data2` are not `ProbPlot` instances, instances will be created using the default parameters. Therefore, it is recommended to use `ProbPlot` instance if fine-grained control is needed in the computation of the quantiles.
Examples

```python
>>> x = np.random.normal(loc=8.5, scale=2.5, size=37)
>>> y = np.random.normal(loc=8.0, scale=3.0, size=37)
>>> pp_x = sm.ProbPlot(x)
>>> pp_y = sm.ProbPlot(y)
>>> qqplot_2samples(data1, data2, xlabel=None, ylabel=None, line=None, ax=None):
```

statsmodels.graphics.gofplots.ProbPlot

class statsmodels.graphics.gofplots.ProbPlot(data, dist=<scipy.stats.distributions.norm_gen object at 0x0534E8D0>, fit=False, distargs=(), a=0, loc=0, scale=1)

Class for convenient construction of Q-Q, P-P, and probability plots.
Can take arguments specifying the parameters for dist or fit them automatically. (See fit under kwargs.)

Parameters:
- **data**: array-like
  1d data array
- **dist**: A scipy.stats or statsmodels distribution
  Compare x against dist. The default is scipy.stats.distributions.norm (a standard normal).
- **distargs**: tuple
  A tuple of arguments passed to dist to specify it fully so dist.ppf may be called.
- **loc**: float
  Location parameter for dist
- **a**: float
  Offset for the plotting position of an expected order statistic, for example. The plotting positions are given by (i - a)/(nobs - 2*a + 1) for i in range(0,nobs+1)
- **scale**: float
  Scale parameter for dist
- **fit**: boolean
  If fit is false, loc, scale, and distargs are passed to the distribution. If fit is True then the parameters for dist are fit automatically using dist.fit. The quantiles are formed from the standardized data, after subtracting the fitted loc and dividing by the fitted scale.

See also:
- scipy.stats.probplot

Notes

1. Depends on matplotlib.
2. If **fit** is True then the parameters are fit using the distribution’s `fit()` method.
3. The call signatures for the `qqplot`, `ppplot`, and `probplot` methods are similar, so examples 1 through 4 apply to all three methods.
4. The three plotting methods are summarized below:

- **ppplot** [Probability-Probability plot] Compares the sample and theoretical probabilities (percentiles).
- **qqplot** [Quantile-Quantile plot] Compares the sample and theoretical quantiles
- **probplot** [Probability plot] Same as a Q-Q plot, however probabilities are shown in the scale of the theoretical distribution (x-axis) and the y-axis contains unscaled quantiles of the sample data.

**Examples**

```python
>>> import statsmodels.api as sm
>>> from matplotlib import pyplot as plt

>>> # example 1
>>> data = sm.datasets.longley.load()
>>> data.exog = sm.add_constant(data.exog)
>>> model = sm.OLS(data.endog, data.exog)
>>> mod_fit = model.fit()
>>> res = mod_fit.resid # residuals
>>> probplot = sm.ProbPlot(res)
>>> probplot.qqplot()
>>> plt.show()
```

qqplot of the residuals against quantiles of t-distribution with 4 degrees of freedom:

```python
>>> # example 2
>>> import scipy.stats as stats

>>> probplot = sm.ProbPlot(res, stats.t, distargs=(4,))
>>> fig = probplot.qqplot()
>>> plt.show()
```

qqplot against same as above, but with mean 3 and std 10:

```python
>>> # example 3
>>> probplot = sm.ProbPlot(res, stats.t, distargs=(4,), loc=3, scale=10)
>>> fig = probplot.qqplot()
>>> plt.show()
```

Automatically determine parameters for t distribution including the loc and scale:

```python
>>> # example 4
>>> probplot = sm.ProbPlot(res, stats.t, fit=True)
>>> fig = probplot.qqplot(line='45')
>>> plt.show()
```

A second ProbPlot object can be used to compare two separate sample sets by using the other kwarg in the qqplot and ppplot methods.

```python
>>> # example 5
>>> import numpy as np

>>> x = np.random.normal(loc=8.25, scale=2.75, size=37)
>>> y = np.random.normal(loc=8.75, scale=3.25, size=37)
>>> pp_x = sm.ProbPlot(x, fit=True)
>>> pp_y = sm.ProbPlot(y, fit=True)
>>> fig = pp_x.qqplot(line='45', other=pp_y)
>>> plt.show()
```
The following plot displays some options, follow the link to see the code.
Sample Quantiles vs. Probability of Exceedance (%)
Methods

ppplot(xlabel=None, ylabel=None, line=None, other=None, ax=None, **plotkwargs)
ProbPlot ppplot

ProbPlot ppplot (xlabel=None, ylabel=None, line=None, other=None, ax=None, **plotkwargs)
P-P plot of the percentiles (probabilities) of x versus the probabilities of a
Distribution.

Parameters

- xlabel, ylabel : str or None, optional
  User-provided labels for the x-axis and y-axis. If None (default), other values
  are used depending on the status of the kwarg other.
- line : str {'45', 's', 'r', 'q'} or None, optional
  Options for the reference line to which the data is compared:
`statsmodels.graphics.gofplots.ProbPlot.probplot`

Probability plot of the unscaled quantiles of x versus the probabilities of a distribution (not to be confused with a P-P plot).

The x-axis is scaled linearly with the quantiles, but the probabilities are used to label the axis.

**Parameters**

- **xlabel**, **ylabel** : str or None, optional
  User-provided labels for the x-axis and y-axis. If None (default), other values are used depending on the status of the kwarg `other`.

- **line** : str (‘45’, ‘s’, ‘r’, ‘q’) or None, optional
  Options for the reference line to which the data is compared:
  - ‘45’ - 45-degree line
  - ‘s’ - standardized line, the expected order statistics are scaled by the standard deviation of the given sample and have the mean added to them
  - ‘r’ - A regression line is fit
  - ‘q’ - A line is fit through the quartiles.
  - None - by default no reference line is added to the plot.

- **exceed** : boolean, optional
  - If False (default) the raw sample quantiles are plotted against the theoretical quantiles, show the probability that a sample will not exceed a given value
  - If True, the theoretical quantiles are flipped such that the figure displays the probability that a sample will exceed a given value.
**ax** : Matplotlib AxesSubplot instance, optional

If given, this subplot is used to plot in instead of a new figure being created.

**plotkwargs** : additional matplotlib arguments to be passed to the

plot command.

Returns **fig** : Matplotlib figure instance

If *ax* is None, the created figure. Otherwise the figure to which *ax* is connected.

```python
statsmodels.graphics.gofplots.ProbPlot.qqplot

ProbPlot.qqplot (xlabel=None, ylabel=None, line=None, other=None, ax=None, **plotkwargs)

Q-Q plot of the quantiles of *x* versus the quantiles/ppf of a distribution or the quantiles of another ProbPlot instance.

Parameters **xlabel** *, ylabel** : str or None, optional

User-provided lables for the x-axis and y-axis. If None (default), other values are used depending on the status of the kwarg *other*.

**line** : str {'45', 's', 'r', 'q'} or None, optional

Options for the reference line to which the data is compared:

- ‘45’ - 45-degree line
- ‘s’ - standardized line, the expected order statistics are scaled by the standard deviation of the given sample and have the mean added to them
- ‘r’ - A regression line is fit
- ‘q’ - A line is fit through the quartiles.
- None - by default no reference line is added to the plot.

**other** : ProbPlot instance, array-like, or None, optional

If provided, the sample quantiles of this ProbPlot instance are plotted against the sample quantiles of the other ProbPlot instance. If an array-like object is provided, it will be turned into a ProbPlot instance using default parameters. If not provided (default), the theoretical quantiles are used.

**ax** : Matplotlib AxesSubplot instance, optional

If given, this subplot is used to plot in instead of a new figure being created.

**plotkwargs** : additional matplotlib arguments to be passed to the

plot command.

Returns **fig** : Matplotlib figure instance

If *ax* is None, the created figure. Otherwise the figure to which *ax* is connected.

```python
statsmodels.graphics.gofplots.ProbPlot.sample_percentiles

static ProbPlot.sample_percentiles ()

statsmodels.graphics.gofplots.ProbPlot.sample_quantiles

static ProbPlot.sample_quantiles()

statsmodels.graphics.gofplots.ProbPlot.sorted_data

static ProbPlot.sorted_data()

statsmodels.graphics.gofplots.ProbPlot.theoretical_percentiles

static ProbPlot.theoretical_percentiles()

statsmodels.graphics.gofplots.ProbPlot.theoretical_quantiles

static ProbPlot.theoretical_quantiles()

### 3.14.2 Boxplots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boxplots.violinplot</code></td>
<td>Make a violin plot of each dataset in the <code>data</code> sequence.</td>
</tr>
<tr>
<td><code>boxplots.beanplot</code></td>
<td>Make a bean plot of each dataset in the <code>data</code> sequence.</td>
</tr>
</tbody>
</table>

#### `statsmodels.graphics.boxplots.violinplot`

`statsmodels.graphics.boxplots.violinplot(data[, ax, labels, ...])`

Make a violin plot of each dataset in the `data` sequence.

A violin plot is a boxplot combined with a kernel density estimate of the probability density function per point.

**Parameters**
- `data`: sequence of ndarrays
  - Data arrays, one array per value in `positions`.
- `ax`: Matplotlib AxesSubplot instance, optional
  - If given, this subplot is used to plot in instead of a new figure being created.
- `labels`: list of str, optional
  - Tick labels for the horizontal axis. If not given, integers `1..len(data)` are used.
- `positions`: array_like, optional
  - Position array, used as the horizontal axis of the plot. If not given, spacing of the violins will be equidistant.
- `side`: {'both', 'left', 'right'}, optional
  - How to plot the violin. Default is ‘both’. The ‘left’, ‘right’ options can be used to create asymmetric violin plots.
- `show_boxplot`: bool, optional
  - Whether or not to show normal box plots on top of the violins. Default is True.
- `plot_opts`: dict, optional
A dictionary with plotting options. Any of the following can be provided, if not present in `plot_opts` the defaults will be used:

- `'violin_fc'`, MPL color. Fill color for violins. Default is 'y'.
- `'violin_ec'`, MPL color. Edge color for violins. Default is 'k'.
- `'violin_lw'`, scalar. Edge linewidth for violins. Default is 1.
- `'violin_alpha'`, float. Transparency of violins. Default is 0.5.
- `'cutoff'`, bool. If True, limit violin range to data range.
  Default is False.
- `'cutoff_val'`, scalar. Where to cut off violins if 'cutoff' is True. Default is 1.5 standard deviations.
- `'cutoff_type'`, {'std', 'abs'}. Whether cutoff value is absolute, or in standard deviations. Default is 'std'.
- `'violin_width'`: float. Relative width of violins. Max available space is 1, default is 0.8.
- `'label_fontsize'`, MPL fontsize. Adjusts fontsize only if given.
- `'label_rotation'`, scalar. Adjusts label rotation only if given.
  Specify in degrees.

**Returns**  
fig : Matplotlib figure instance

If `ax` is None, the created figure. Otherwise the figure to which `ax` is connected.

See also:

- `beanplot`  
  Bean plot, builds on `violinplot`.

- `matplotlib.pyplot.boxplot`  
  Standard boxplot.

Notes

The appearance of violins can be customized with `plot_opts`. If customization of boxplot elements is required, set `show_boxplot` to False and plot it on top of the violins by calling the Matplotlib `boxplot` function directly. For example:

```python
violinplot(data, ax=ax, show_boxplot=False)
ax.boxplot(data, sym='cv', whis=2.5)
```

It can happen that the axis labels or tick labels fall outside the plot area, especially with rotated labels on the horizontal axis. With Matplotlib 1.1 or higher, this can easily be fixed by calling `ax.tight_layout()`. With older Matplotlib one has to use `plt.rc` or `plt.rcParams` to fix this, for example:

```python
plt.rc('figure.subplot', bottom=0.25)
violinplot(data, ax=ax)
```

References


Examples

We use the American National Election Survey 1996 dataset, which has Party Identification of respondents as independent variable and (among other data) age as dependent variable.
Group age by party ID, and create a violin plot with it:

```python
>>> plt.rcParams['figure.subplot.bottom'] = 0.23  # keep labels visible
>>> age = [data.exog['age'][data.endog == id] for id in party_ID]
>>> ax = fig.figure()
>>> ax.set_xlabel("Party identification of respondent.")
>>> ax.set_ylabel("Age")
```

A bean plot is a combination of a violinplot (kernel density estimate of the probability density function per
point) with a line-scatter plot of all individual data points.

Parameters

- **data**: sequence of ndarrays
  Data arrays, one array per value in `positions`.

- **ax**: Matplotlib AxesSubplot instance, optional
  If given, this subplot is used to plot in instead of a new figure being created.

- **labels**: list of str, optional
  Tick labels for the horizontal axis. If not given, integers `1..len(data)` are used.

- **positions**: array_like, optional
  Position array, used as the horizontal axis of the plot. If not given, spacing of the violins will be equidistant.

- **side**: {'both', 'left', 'right'}, optional
  How to plot the violin. Default is ‘both’. The ‘left’, ‘right’ options can be used to create asymmetric violin plots.

- **jitter**: bool, optional
  If True, jitter markers within violin instead of plotting regular lines around the center. This can be useful if the data is very dense.

- **plot_opts**: dict, optional
  A dictionary with plotting options. All the options for `violinplot` can be specified, they will simply be passed to `violinplot`. Options specific to `beanplot` are:

  - **bean_color**, MPL color. Color of bean plot lines. Default is ‘k’. Also used for jitter marker edge color if `jitter` is True.
  - **bean_size**, scalar. Line length as a fraction of maximum length. Default is 0.5.
  - **bean_lw**, scalar. Linewidth, default is 0.5.
  - **bean_show_mean**, bool. If True (default), show mean as a line.
  - **bean_show_median**, bool. If True (default), show median as a marker.
  - **bean_mean_color**, MPL color. Color of mean line. Default is ‘b’.
  - **bean_mean_lw**, scalar. Linewidth of mean line, default is 2.
  - **bean_median_color**, MPL color. Color of median marker. Default is ‘r’.
  - **bean_median_marker**, MPL marker. Marker type, default is ‘+’.
  - **jitter_marker**, MPL marker. Marker type for `jitter=True`. Default is ‘o’.
  - **jitter_fc**, MPL color. Jitter marker face color. Default is None.
  - **bean_legend_text**, str. If given, add a legend with given text.

Returns

- **fig**: Matplotlib figure instance
  If `ax` is None, the created figure. Otherwise the figure to which `ax` is connected.

See also:
**violinplot** Violin plot, also used internally in *beanplot*.

**matplotlib.pyplot.boxplot** Standard boxplot.

### References


### Examples

We use the American National Election Survey 1996 dataset, which has Party Identification of respondents as independent variable and (among other data) age as dependent variable.

```python
>>> data = sm.datasets.anes96.load_pandas()
>>> party_ID = np.arange(7)
>>> labels = ['Strong Democrat', 'Weak Democrat', 'Independent-Democrat', ...
            'Independent-Independent', 'Independent-Republican', ...
            'Weak Republican', 'Strong Republican']

Group age by party ID, and create a violin plot with it:

```python
>>> plt.rcParams['figure.subplot.bottom'] = 0.23  # keep labels visible
>>> age = [data.exog['age'][data.endog == id] for id in party_ID]
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> sm.graphics.beanplot(age, ax=ax, labels=labels,
                        plot_opts={'cutoff_val':5, 'cutoff_type':'abs',
                                   'label_fontsize':'small',
                                   'label_rotation':30})
>>> ax.set_xlabel("Party identification of respondent.")
>>> ax.set_ylabel("Age")
>>> plt.show()
```
3.14.3 Correlation Plots

```
correlation.plot_corr(dcorr[, xnames, ...])  # Plot correlation of many variables in a tight color grid.
correlation.plot_corr_grid(dcorrs[, titles, ...])  # Create a grid of correlation plots.
plot_grids.scatter_ellipse(data[, level, ...])  # Create a grid of scatter plots with confidence ellipses.
```

**statsmodels.graphics.correlation.plot_corr**

```
plot_corr(dcorr, xnames=None, ynames=None, title=None, normcolor=False, ax=None, cmap='RdYlBu_r')
```

Plot correlation of many variables in a tight color grid.

**Parameters**
- `dcorr`: ndarray
  - Correlation matrix, square 2-D array.
- `xnames`: list of str, optional
  - Labels for the horizontal axis. If not given (None), then the matplotlib defaults (integers) are used. If it is an empty list, [], then no ticks and labels are added.
- `ynames`: list of str, optional
  - Labels for the vertical axis. Works the same way as `xnames`. If not given, the same names as for `xnames` are re-used.
**title**: str, optional

The figure title. If None, the default ('Correlation Matrix') is used. If title='', then no title is added.

**normcolor**: bool or tuple of scalars, optional

If False (default), then the color coding range corresponds to the range of dcorr. If True, then the color range is normalized to (-1, 1). If this is a tuple of two numbers, then they define the range for the color bar.

**ax**: Matplotlib AxesSubplot instance, optional

If ax is None, then a figure is created. If an axis instance is given, then only the main plot but not the colorbar is created.

**cmap**: str or Matplotlib Colormap instance, optional

The colormap for the plot. Can be any valid Matplotlib Colormap instance or name.

**Returns**

**fig**: Matplotlib figure instance

If ax is None, the created figure. Otherwise the figure to which ax is connected.

**Examples**

```python
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import statsmodels.api as sm

>>> hie_data = sm.datasets.randhie.load_pandas()
>>> corr_matrix = np.corrcoef(hie_data.data.T)
>>> sm.graphics.plot_corr(corr_matrix, xnames=hie_data.names)
>>> plt.show()
```

**statsmodels.graphics.correlation.plot_corr_grid**

Create a grid of correlation plots.

The individual correlation plots are assumed to all have the same variables, axis labels can be specified only once.

**Parameters**

**dcorrs**: list or iterable of ndarrays

List of correlation matrices.

**titles**: list of str, optional

List of titles for the subplots. By default no title are shown.

**ncols**: int, optional

Number of columns in the subplot grid. If not given, the number of columns is determined automatically.

**normcolor**: bool or tuple, optional
If False (default), then the color coding range corresponds to the range of \textit{dcorr}. If True, then the color range is normalized to (-1, 1). If this is a tuple of two numbers, then they define the range for the color bar.

\textbf{xnames} : list of str, optional
Labels for the horizontal axis. If not given (None), then the matplotlib defaults (integers) are used. If it is an empty list, [], then no ticks and labels are added.

\textbf{ynames} : list of str, optional
Labels for the vertical axis. Works the same way as \textit{xnames}. If not given, the same names as for \textit{xnames} are re-used.

\textbf{fig} : Matplotlib figure instance, optional
If given, this figure is simply returned. Otherwise a new figure is created.

\textbf{cmap} : str or Matplotlib Colormap instance, optional
The colormap for the plot. Can be any valid Matplotlib Colormap instance or name.

\textbf{Returns} \textbf{fig} : Matplotlib figure instance
If \textit{ax} is None, the created figure. Otherwise the figure to which \textit{ax} is connected.

\textbf{Examples}

```python
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import statsmodels.api as sm

In this example we just reuse the same correlation matrix several times. Of course in reality one would show a different correlation (measuring a another type of correlation, for example Pearson (linear) and Spearman, Kendall (nonlinear) correlations) for the same variables.

```python
>>> hie_data = sm.datasets.randhie.load_pandas()
>>> corr_matrix = np.corrcoef(hie_data.data.T)
>>> sm.graphics.plot_corr_grid([corr_matrix] * 8, xnames=hie_data.names)
>>> plt.show()
```

\textbf{statsmodels.graphics.plot_grids.scatter_ellipse}

Create a grid of scatter plots with confidence ellipses. ell_kwds, plot_kdes not used yet

\textbf{looks ok with 5 or 6 variables, too crowded with 8, too empty with 1

\textbf{Parameters} \textbf{data} : array_like
Input data.

\textbf{level} : scalar, optional
Default is 0.9.

\textbf{varnames} : list of str, optional
Variable names. Used for y-axis labels, and if add_titles is True also for titles. If not given, integers 1..data.shape[1] are used.

ell_kwds : dict, optional
UNUSED

plot_kwds : dict, optional
UNUSED

add_titles : bool, optional
Whether or not to add titles to each subplot. Default is False. Titles are constructed from varnames.

keep_ticks : bool, optional
If False (default), remove all axis ticks.

fig : Matplotlib figure instance, optional
If given, this figure is simply returned. Otherwise a new figure is created.

Returns fig : Matplotlib figure instance
If fig is None, the created figure. Otherwise fig itself.

## 3.14.4 Functional Plots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional.fboxplot</td>
<td>Plot functional boxplot.</td>
</tr>
<tr>
<td>functional.rainbowplot</td>
<td>Create a rainbow plot for a set of curves.</td>
</tr>
<tr>
<td>functional.banddepth</td>
<td>Calculate the band depth for a set of functional curves.</td>
</tr>
</tbody>
</table>

### `statsmodels.graphics.functional.fboxplot`

Plot functional boxplot.

A functional boxplot is the analog of a boxplot for functional data. Functional data is any type of data that varies over a continuum, i.e. curves, probability distributions, seasonal data, etc.

The data is first ordered, the order statistic used here is *banddepth*. Plotted are then the median curve, the envelope of the 50% central region, the maximum non-outlying envelope and the outlier curves.

**Parameters**

- **data**: sequence of ndarrays or 2-D ndarray
  - The vectors of functions to create a functional boxplot from. If a sequence of 1-D arrays, these should all be the same size. The first axis is the function index, the second axis the one along which the function is defined. So `data[0, :]` is the first functional curve.

- **xdata**: ndarray, optional
  - The independent variable for the data. If not given, it is assumed to be an array of integers 0..N with N the length of the vectors in `data`.

- **labels**: sequence of scalar or str, optional
The labels or identifiers of the curves in `data`. If given, outliers are labeled in the plot.

**depth** : ndarray, optional

A 1-D array of band depths for `data`, or equivalent order statistic. If not given, it will be calculated through `banddepth`.

**method** : {‘MBD’, ‘BD2’}, optional

The method to use to calculate the band depth. Default is ‘MBD’.

**wfactor** : float, optional

Factor by which the central 50% region is multiplied to find the outer region (analog of “whiskers” of a classical boxplot).

**ax** : Matplotlib AxesSubplot instance, optional

If given, this subplot is used to plot in instead of a new figure being created.

**plot_opts** : dict, optional

A dictionary with plotting options. Any of the following can be provided, if not present in `plot_opts` the defaults will be used:

- ‘cmap_outliers’, a Matplotlib LinearSegmentedColormap instance.
- ‘c_inner’, valid MPL color. Color of the central 50% region
- ‘c_outer’, valid MPL color. Color of the non-outlying region
- ‘c_median’, valid MPL color. Color of the median.
- ‘lw_outliers’, scalar. Linewidth for drawing outlier curves.
- ‘lw_median’, scalar. Linewidth for drawing the median curve.
- ‘draw_nonout’, bool. If True, also draw non-outlying curves.

**Returns**

**fig** : Matplotlib figure instance

If `ax` is None, the created figure. Otherwise the figure to which `ax` is connected.

**depth** : ndarray

1-D array containing the calculated band depths of the curves.

**ix_depth** : ndarray

1-D array of indices needed to order curves (or `depth`) from most to least central curve.

**ix_outliers** : ndarray

1-D array of indices of outlying curves in `data`.

**See also:**

`banddepth`, `rainbowplot`

**Notes**

The median curve is the curve with the highest band depth.

Outliers are defined as curves that fall outside the band created by multiplying the central region by `wfactor`. Note that the range over which they fall outside this band doesn’t matter, a single data point outside the band is enough. If the data is noisy, smoothing may therefore be required.

The non-outlying region is defined as the band made up of all the non-outlying curves.
References


Examples

Load the El Nino dataset. Consists of 60 years worth of Pacific Ocean sea surface temperature data.

```python
>>> import matplotlib.pyplot as plt
>>> import statsmodels.api as sm

data = sm.datasets.elnino.load()
```

Create a functional boxplot. We see that the years 1982-83 and 1997-98 are outliers; these are the years where El Nino (a climate pattern characterized by warming up of the sea surface and higher air pressures) occurred with unusual intensity.

```python
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> res = sm.graphics.fboxplot(data.raw_data[:, 1:], wfactor=2.58,
...                             labels=data.raw_data[:, 0].astype(int),
...                             ax=ax)
```

```python
>>> ax.set_xlabel("Month of the year")
>>> ax.set_ylabel("Sea surface temperature (C)")
>>> ax.set_xticks(np.arange(13, step=3) - 1)
>>> ax.set_xticklabels(['', 'Mar', 'Jun', 'Sep', 'Dec'])
>>> ax.set_xlim([-0.2, 11.2])
```

```python
>>> plt.show()
```
statsmodels.graphics.functional.rainbowplot

statsmodels.graphics.functional.rainbowplot(data, xdata=None, depth=None, method='MBD', ax=None, cmap=None)

Create a rainbow plot for a set of curves.

A rainbow plot contains line plots of all curves in the dataset, colored in order of functional depth. The median curve is shown in black.

Parameters data : sequence of ndarrays or 2-D ndarray

The vectors of functions to create a functional boxplot from. If a sequence of 1-D arrays, these should all be the same size. The first axis is the function index, the second axis the one along which the function is defined. So data[0, :] is the first functional curve.

xdata : ndarray, optional

The independent variable for the data. If not given, it is assumed to be an array of integers 0..N with N the length of the vectors in data.

depth : ndarray, optional

A 1-D array of band depths for data, or equivalent order statistic. If not given, it will be calculated through banddepth.

method : {'MBD', 'BD2'}, optional

The method to use to calculate the band depth. Default is ‘MBD’.
ax : Matplotlib AxesSubplot instance, optional

If given, this subplot is used to plot in instead of a new figure being created.

cmap : Matplotlib LinearSegmentedColormap instance, optional

The colormap used to color curves with. Default is a rainbow colormap, with red
used for the most central and purple for the least central curves.

Returns fig : Matplotlib figure instance

If ax is None, the created figure. Otherwise the figure to which ax is connected.

See also:
banddepth, fboxplot

References


Examples

Load the El Nino dataset. Consists of 60 years worth of Pacific Ocean sea surface temperature data.

```python
>>> import matplotlib.pyplot as plt
>>> import statsmodels.api as sm
>>> data = sm.datasets.elnino.load()

Create a rainbow plot:

```
Statsmodels Graphics Univariate Functionals

**statsmodels.graphics.functional.banddepth**

`statsmodels.graphics.functional.banddepth(data, method='MBD')`

Calculate the band depth for a set of functional curves.

Band depth is an order statistic for functional data (see *fboxplot*), with a higher band depth indicating larger “centrality”. In analog to scalar data, the functional curve with highest band depth is called the median curve, and the band made up from the first N/2 of N curves is the 50% central region.

**Parameters**

- **data** : ndarray
  
  The vectors of functions to create a functional boxplot from. The first axis is the function index, the second axis the one along which the function is defined. So `data[0, :]` is the first functional curve.

- **method** : {'MBD', 'BD2'}, optional
  
  Whether to use the original band depth (with J=2) of [R1] or the modified band depth. See Notes for details.

**Returns**

- **depth** : ndarray
  
  Depth values for functional curves.
Notes

Functional band depth as an order statistic for functional data was proposed in [R1] and applied to functional
boxplots and bagplots in [R2].

The method ‘BD2’ checks for each curve whether it lies completely inside bands constructed from two curves.
All permutations of two curves in the set of curves are used, and the band depth is normalized to one. Due to
the complete curve having to fall within the band, this method yields a lot of ties.

The method ‘MBD’ is similar to ‘BD2’, but checks the fraction of the curve falling within the bands. It therefore
generates very few ties.

References

[R1], [R2]

3.14.5 Regression Plots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regressionplots.plot_fit</td>
<td>Plot fit against one regressor.</td>
</tr>
<tr>
<td>regressionplots.plot_regress_exog</td>
<td>Plot regression results against one regressor.</td>
</tr>
<tr>
<td>regressionplots.plot_partregress</td>
<td>Plot partial regression for a single regressor.</td>
</tr>
<tr>
<td>regressionplots.plot_ccpr</td>
<td>Plot CCPR against one regressor.</td>
</tr>
<tr>
<td>regressionplots.abline_plot</td>
<td>Plots a line given an intercept and slope.</td>
</tr>
<tr>
<td>regressionplots.influence_plot</td>
<td>Plot of influence in regression.</td>
</tr>
<tr>
<td>regressionplots.plot_leverage_resid2</td>
<td>Plots leverage statistics vs.</td>
</tr>
</tbody>
</table>

statsmodels.graphics.regressionplots.plot_fit

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statsmodels.graphics.regressionplots.plot_fit</td>
<td>Plot fit against one regressor.</td>
</tr>
</tbody>
</table>

This creates one graph with the scatterplot of observed values compared to fitted values.

**Parameters**

- **results**: result instance
  - result instance with resid, model.endog and model.exog as attributes
- **x_var**: int or str
  - Name or index of regressor in exog matrix.
- **y_true**: array_like
  - (optional) If this is not None, then the array is added to the plot
- **ax**: Matplotlib AxesSubplot instance, optional
  - If given, this subplot is used to plot in instead of a new figure being created.
- **kwargs**: The keyword arguments are passed to the plot command for the fitted values points.

**Returns**

- **fig**: Matplotlib figure instance
  - If `ax` is None, the created figure. Otherwise the figure to which `ax` is connected.
Examples

Load the Statewide Crime data set and perform linear regression with poverty and hs_grad as variables and murder as the response

```python
>>> import statsmodels.api as sm
>>> import matplotlib.pyplot as plt

>>> data = sm.datasets.statecrime.load_pandas().data

... murder = data['murder']
>>> X = data[['poverty', 'hs_grad']]

... X['constant'] = 1
>>> y = murder
>>> model = sm.OLS(y, X)

... results = model.fit()

Create a plot just for the variable ‘Poverty’:

```python
>>> fig, ax = plt.subplots()
>>> fig = sm.graphics.plot_fit(results, 0, ax=ax)
>>> ax.set_ylabel('Murder Rate')
>>> ax.set_xlabel('Poverty Level')
>>> ax.set_title('Linear Regression')
```
statsmodels.graphics.regressionplots.plot_regress_exog

`statsmodels.graphics.regressionplots.plot_regress_exog(results, exog_idx, fig=None)`

Plot regression results against one regressor.

This plots four graphs in a 2 by 2 figure: ‘endog versus exog’, ‘residuals versus exog’, ‘fitted versus exog’ and ‘fitted plus residual versus exog’

**Parameters**

- **results**: result instance
  - result instance with resid, model.endog and model.exog as attributes
- **exog_idx**: int
  - index of regressor in exog matrix
- **fig**: Matplotlib figure instance, optional
  - If given, this figure is simply returned. Otherwise a new figure is created.

**Returns**

- **fig**: matplotlib figure instance

statsmodels.graphics.regressionplots.plot_partregress

`statsmodels.graphics.regressionplots.plot_partregress(endog, exog_i, exog_others, data=None, title_kwargs={}, obs_labels=True, label_kwargs={}, ax=None, ret_coords=False, **kwargs)`

Plot partial regression for a single regressor.

**Parameters**

- **endog**: ndarray or string
  - endogenous or response variable. If string is given, you can use a arbitrary translations as with a formula.
- **exog_i**: ndarray or string
  - exogenous, explanatory variable. If string is given, you can use a arbitrary translations as with a formula.
- **exog_others**: ndarray or list of strings
  - other exogenous, explanatory variables. If a list of strings is given, each item is a term in formula. You can use a arbitrary translations as with a formula. The effect of these variables will be removed by OLS regression.
- **data**: DataFrame, dict, or recarray
  - Some kind of data structure with names if the other variables are given as strings.
- **title_kwargs**: dict
  - Keyword arguments to pass on for the title. The key to control the fonts is fontdict.
- **obs_labels**: bool or array-like
  - Whether or not to annotate the plot points with their observation labels. If obs_labels is a boolean, the point labels will try to do the right thing. First it will try to use the index of data, then fall back to the index of exog_i. Alternatively, you may give an array-like object corresponding to the obseveration numbers.
- **label_kwargs**: dict
  - Keyword arguments to pass on for the labels. The key to control the fonts is fontdict.
Keyword arguments that control annotate for the observation labels.

**ax**: Matplotlib AxesSubplot instance, optional
If given, this subplot is used to plot in instead of a new figure being created.

**ret_coords**: bool
If True will return the coordinates of the points in the plot. You can use this to add your own annotations.

**kwargs**:
The keyword arguments passed to plot for the points.

**Returns**

- **fig**: Matplotlib figure instance
  If `ax` is None, the created figure. Otherwise the figure to which `ax` is connected.

- **coords**: list, optional
  If `ret_coords` is True, return a tuple of arrays (x_coords, y_coords).

See also:

- **plot_partregress_grid**: Plot partial regression for a set of regressors.

**Notes**

The slope of the fitted line is the that of `exog_i` in the full multiple regression. The individual points can be used to assess the influence of points on the estimated coefficient.

```python
statsmodels.graphics.regressionplots.plot_ccpr
```

Generates a CCPR (component and component-plus-residual) plot.

**Parameters**

- **results**: result instance
  A regression results instance.

- **exog_idx**: int or string
  Exogenous, explanatory variable. If string is given, it should be the variable name that you want to use, and you can use arbitrary translations as with a formula.

- **ax**: Matplotlib AxesSubplot instance, optional
  If given, it is used to plot in instead of a new figure being created.

**Returns**

- **fig**: Matplotlib figure instance
  If `ax` is None, the created figure. Otherwise the figure to which `ax` is connected.

See also:

- **plot_ccpr_grid**: Creates CCPR plot for multiple regressors in a plot grid.
Notes

The CCPR plot provides a way to judge the effect of one regressor on the response variable by taking into account the effects of the other independent variables. The partial residuals plot is defined as Residuals + B_i*X_i versus X_i. The component adds the B_i*X_i versus X_i to show where the fitted line would lie. Care should be taken if X_i is highly correlated with any of the other independent variables. If this is the case, the variance evident in the plot will be an underestimate of the true variance.

References


statsmodels.graphics.regressionplots.abline_plot

`statsmodels.graphics.regressionplots.abline_plot (intercept=None, slope=None, horiz=None, vert=None, model_results=None, **kwargs)`

Plots a line given an intercept and slope.

**intercept**  [float] The intercept of the line

**slope**  [float] The slope of the line

**horiz**  [float or array-like] Data for horizontal lines on the y-axis

**vert**  [array-like] Data for vertical lines on the x-axis

**model_results**  [statsmodels results instance] Any object that has a two-value `params` attribute. Assumed that it is (intercept, slope)

**ax**  [axes, optional] Matplotlib axes instance

**kwargs**  Options passed to matplotlib.pyplot.plt

**Returns**  **fig** : Figure

The figure given by `ax.figure` or a new instance.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm
>>> np.random.seed(12345)
>>> X = sm.add_constant(np.random.normal(0, 20, size=30))
>>> y = np.dot(X, [25, 3.5]) + np.random.normal(0, 30, size=30)
>>> mod = sm.OLS(y,X).fit()
>>> fig = abline_plot(model_results=mod)
>>> ax = fig.axes[0]
>>> ax.scatter(X[:,1], y)
>>> ax.margins(.1)
>>> import matplotlib.pyplot as plt
>>> plt.show()
```
Plot of influence in regression. Plots studentized residuals vs. leverage.

**Parameters**

- **results**: results instance
  A fitted model.
- **external**: bool
  Whether to use externally or internally studentized residuals. It is recommended to leave external as True.
- **alpha**: float
  The alpha value to identify large studentized residuals. Large means abs(resid_studentized) > t.ppf(1-alpha/2, dof=results.df_resid)
- **criterion**: str {'DFFITS', 'Cooks'}
  Which criterion to base the size of the points on. Options are DFFITS or Cook’s D.
- **size**: float
  The range of criterion is mapped to 10**2 - size**2 in points.
- **plot_alpha**: float
  The alpha of the plotted points.
- **ax**: matplotlib Axes instance
  An instance of a matplotlib Axes.

**Returns**

- **fig**: matplotlib figure
  The matplotlib figure that contains the Axes.

**Notes**

Row labels for the observations in which the leverage, measured by the diagonal of the hat matrix, is high or the residuals are large, as the combination of large residuals and a high influence value indicates an influence point. The value of large residuals can be controlled using the alpha parameter. Large leverage points are identified as hat_i > 2 * (df_model + 1)/nobs.
Specifies the cut-off for large-standardized residuals. Residuals are assumed to be distributed $N(0, 1)$ with $\alpha=\alpha$.

**label_kwargs**: dict

The keywords to pass to annotate for the labels.

**ax**: Axes instance

Matplotlib Axes instance

**Returns**

**fig**: matplotlib Figure

A matplotlib figure instance.

### 3.14.6 Time Series Plots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tsaplots.plot_acf</code></td>
<td>Plot the autocorrelation function</td>
</tr>
<tr>
<td><code>tsaplots.plot_pacf</code></td>
<td>Plot the partial autocorrelation function</td>
</tr>
<tr>
<td><code>tsaplots.month_plot</code></td>
<td>Seasonal plot of monthly data</td>
</tr>
<tr>
<td><code>tsaplots.quarter_plot</code></td>
<td>Seasonal plot of quarterly data</td>
</tr>
</tbody>
</table>

#### statsmodels.graphics.tsaplots.plot_acf

Plot the autocorrelation function

Plots lags on the horizontal and the correlations on vertical axis.

**Parameters**

- **x**: array_like
  
  Array of time-series values

- **ax**: Matplotlib AxesSubplot instance, optional
  
  If given, this subplot is used to plot in instead of a new figure being created.

- **lags**: array_like, optional
  
  Array of lag values, used on horizontal axis. If not given, `lags=np.arange(len(corr))` is used.

- **alpha**: scalar, optional
  
  If a number is given, the confidence intervals for the given level are returned. For instance if `alpha=.05`, 95 % confidence intervals are returned where the standard deviation is computed according to Bartlett’s formula. If None, no confidence intervals are plotted.

- **use_vlines**: bool, optional
  
  If True, vertical lines and markers are plotted. If False, only markers are plotted. The default marker is ‘o’; it can be overridden with a `marker` kwarg.

- **unbiased**: bool
  
  If True, then denominators for autocovariance are n-k, otherwise n

- **fft**: bool, optional
If True, computes the ACF via FFT.

**kwargs : kwargs, optional

Optional keyword arguments that are directly passed on to the Matplotlib plot and axhline functions.

Returns  
fig : Matplotlib figure instance

If ax is None, the created figure. Otherwise the figure to which ax is connected.

See also:

matplotlib.pyplot.xcorr, matplotlib.pyplot.acorr, mpl_examples

Notes

Adapted from matplotlib’s xcorr.

Data are plotted as plot(lags, corr, **kwargs)

stsmodels.graphics.tsaplots.plot_pacf

stsmodels.graphics.tsaplots.plot_pacf(x, ax=None, lags=None, alpha=0.05, method='ywunbiased', use_vlines=True, **kwargs)

Plot the partial autocorrelation function

Plots lags on the horizontal and the correlations on vertical axis.

Parameters  
x : array_like

Array of time-series values

ax : Matplotlib AxesSubplot instance, optional

If given, this subplot is used to plot in instead of a new figure being created.

lags : array_like, optional

Array of lag values, used on horizontal axis. If not given, lags=np.arange(len(corr)) is used.

alpha : scalar, optional

If a number is given, the confidence intervals for the given level are returned. For instance if alpha=.05, 95 % confidence intervals are returned where the standard deviation is computed according to 1/sqrt(len(x))

method : ‘ywunbiased’ (default) or ‘ywmle’ or ‘ols’

specifies which method for the calculations to use:

• yw or ywunbiased : yule walker with bias correction in denominator for acovf
• ywm or ywmle : yule walker without bias correction
• ols - regression of time series on lags of it and on constant
• ld or ldunbiased : Levinson-Durbin recursion with bias correction
• ldb or ldbiased : Levinson-Durbin recursion without bias correction

use_vlines : bool, optional
If True, vertical lines and markers are plotted. If False, only markers are plotted. The default marker is ‘o’; it can be overridden with a marker kwarg.

**kwargs : kwargs, optional
Optional keyword arguments that are directly passed on to the Matplotlib plot and axhline functions.

Returns fig : Matplotlib figure instance
If ax is None, the created figure. Otherwise the figure to which ax is connected.

See also:
matplotlib.pyplot.xcorr, matplotlib.pyplot.acorr, mpl_examples

Notes
Adapted from matplotlib’s xcorr.
Data are plotted as plot(lags, corr, **kwargs)

statsmodels.graphics.tsaplots.month_plot

statsmodels.graphics.tsaplots.month_plot (x, dates=None, ylabel=None, ax=None)
Seasonal plot of monthly data

Parameters x : array-like
Seasonal data to plot. If dates is None, x must be a pandas object with a PeriodIndex or DatetimeIndex with a monthly frequency.

dates : array-like, optional
If x is not a pandas object, then dates must be supplied.

ylabel : str, optional
The label for the y-axis. Will attempt to use the name attribute of the Series.

ax : matplotlib.axes, optional
Existing axes instance.

Returns matplotlib.Figure :

Examples

```python
>>> import statsmodels.api as sm
>>> import pandas as pd

>>> dta = sm.datasets.elnino.load_pandas().data
>>> dta['YEAR'] = dta.YEAR.astype(int).astype(str)
>>> dta = dta.set_index('YEAR').T.unstack()
>>> dates = map(lambda x : pd.datetools.parse('1 '+', '.join(x)),
...     dta.index.values)

>>> dta.index = pd.DatetimeIndex(dates, freq='M')
>>> fig = sm.graphics.tsa.month_plot(dta)
```
statsmodels.graphics.tsaplots.quarter_plot

statsmodels.graphics.tsaplots.quarter_plot(x, dates=None, ylabel=None, ax=None)
Seasonal plot of quarterly data

**Parameters**

- **x**: array-like
  
  Seasonal data to plot. If dates is None, x must be a pandas object with a PeriodIndex or DatetimeIndex with a monthly frequency.

- **dates**: array-like, optional
  If x is not a pandas object, then dates must be supplied.

- **ylabel**: str, optional
  The label for the y-axis. Will attempt to use the name attribute of the Series.

- **ax**: matplotlib.axes, optional
  Existing axes instance.

**Returns**

matplotlib.pyplot.Figure:

3.14.7 Other Plots

factorplots.interaction_plot(x, trace, response) Interaction plot for factor level statistics.
mosaicplot.mosaic(data[, index, ax, ...]) Create a mosaic plot from a contingency table.

statsmodels.graphics.factorplots.interaction_plot

statsmodels.graphics.factorplots.interaction_plot(x, trace, response, func=<function mean at 0x02E014F0>, ax=None, plottype='b', xlabel=None, ylabel=None, colors=[], markers=[], linestyles=[], legendloc='best', legendtitle=None, **kwargs)

Interaction plot for factor level statistics.

Note. If categorical factors are supplied levels will be internally recoded to integers. This ensures matplotlib compatibility.

uses pandas.DataFrame to calculate an aggregate statistic for each level of the factor or group given by trace.

**Parameters**

- **x**: array-like
  
The x factor levels constitute the x-axis. If a pandas.Series is given its name will be used in xlabel if xlabel is None.

- **trace**: array-like
  
The trace factor levels will be drawn as lines in the plot. If trace is a pandas.Series its name will be used as the legendtitle if legendtitle is None.

- **response**: array-like
  
The response or dependent variable. If a pandas.Series is given its name will be used in ylabel if ylabel is None.

- **func**: function
  
  Anything accepted by pandas.DataFrame.aggregate. This is applied to the response variable grouped by the trace levels.

- **plottype**: str {'line', 'scatter', 'both'}, optional
  
The type of plot to return. Can be ‘l’, ‘s’, or ‘b’

- **ax**: axes, optional
  
  Matplotlib axes instance

- **xlabel**: str, optional
  
  Label to use for x. Default is ‘X’. If x is a pandas.Series it will use the series names.

- **ylabel**: str, optional
  
  Label to use for response. Default is ‘func of response’. If response is a pandas.Series it will use the series names.

- **colors**: list, optional
  
  If given, must have length == number of levels in trace.

- **linestyles**: list, optional
  
  If given, must have length == number of levels in trace.

- **markers**: list, optional
  
  If given, must have length == number of levels in trace.
If given, must have length == number of levels in trace

**kwargs:

These will be passed to the plot command used either plot or scatter. If you want to control the overall plotting options, use **kwargs.

**Returns** fig : Figure

The figure given by ax.figure or a new instance.

**Examples**

```python
>>> import numpy as np
>>> np.random.seed(12345)
>>> weight = np.random.randint(1,4,size=60)
>>> duration = np.random.randint(1,3,size=60)
>>> days = np.log(np.random.randint(1,30, size=60))
>>> fig = interaction_plot(weight, duration, days, ...
                colors=['red','blue'], markers=['D','^'], ms=10)
>>> import matplotlib.pyplot as plt
>>> plt.show()
```

![Interaction Plot Example](image-url)
Create a mosaic plot from a contingency table.

It allows to visualize multivariate categorical data in a rigorous and informative way.

**Parameters**

- **data**: dict, pandas.Series, np.ndarray, pandas.DataFrame
  
The contingency table that contains the data. Each category should contain a non-negative number with a tuple as index. It expects that all the combination of keys to be representes; if that is not true, will automatically consider the missing values as 0. The order of the keys will be the same as the one of insertion. If a dict of a Series (or any other dict like object) is used, it will take the keys as labels. If a np.ndarray is provided, it will generate a simple numerical labels.

- **index**: list, optional
  
  Gives the preferred order for the category ordering. If not specified will default to the given order. It doesn’t support named indexes for hierarchical Series. If a DataFrame is provided, it expects a list with the name of the columns.

- **ax**: matplotlib.Axes, optional
  
  The graph where display the mosaic. If not given, will create a new figure

- **horizontal**: bool, optional (default True)
  
  The starting direction of the split (by default along the horizontal axis)

- **gap**: float or array of floats
  
  The list of gaps to be applied on each subdivision. If the lenght of the given array is less of the number of subcategories (or if it’s a single number) it will extend it with exponentially decreasing gaps

- **labelizer**: function (key) -> string, optional
  
  A function that generate the text to display at the center of each tile base on the key of that tile

- **properties**: function (key) -> dict, optional
  
  A function that for each tile in the mosaic take the key of the tile and returns the dictionary of properties of the generated Rectangle, like color, hatch or similar. A default properties set will be provided fot the keys whose color has not been defined, and will use color variation to help visually separates the various categories. It should return None to indicate that it should use the default property for the tile. A dictionary of the properties for each key can be passed, and it will be internally converted to the correct function

- **statistic**: bool, optional (default False)
  
  if true will use a crude statistical model to give colors to the plot. If the tile has a containt that is more than 2 standard deviation from the expected value under independence hipotesys, it will go from green to red (for positive deviations, blue otherwise) and will acquire an hatching when crosses the 3 sigma.

- **title**: string, optional
  
  The title of the axis
axes_label: boolean, optional:
    Show the name of each value of each category on the axis (default) or hide them.

label_rotation: float or list of float:
    the rotation of the axis label (if present). If a list is given each axis can have a
different rotation

Returns
    fig : matplotlib.Figure
        The generate figure

    rects : dict
        A dictionary that has the same keys of the original dataset, that holds a reference to
        the coordinates of the tile and the Rectangle that represent it

See also:

A Michael Friendly, York University, Psychology Department Journal of Computational and Graphical Statistics,
2001

Mosaic Michael Friendly, York University, Psychology Department Proceedings of the Statistical Graphics
Section, 1992, 61-68.

Mosaic Michael Friendly, York University, Psychology Department Journal of the american statistical association

Examples

The most simple use case is to take a dictionary and plot the result

```python
>>> data = {'a': 10, 'b': 15, 'c': 16}
>>> mosaic(data, title='basic dictionary')
>>> pylab.show()
```

A more useful example is given by a dictionary with multiple indices. In this case we use a wider gap to a better
visual separation of the resulting plot

```python
>>> data = {('a', 'b'): 1, ('a', 'c'): 2, ('d', 'b'): 3, ('d', 'c'): 4}
>>> mosaic(data, gap=0.05, title='complete dictionary')
>>> pylab.show()
```

The same data can be given as a simple or hierarchical indexed Series

```python
>>> rand = np.random.random
>>> from itertools import product
>>> tuples = list(product(['bar', 'baz', 'foo', 'qux'], ['one', 'two']))
>>> index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
>>> data = pd.Series(rand(8), index=index)
>>> mosaic(data, title='hierarchical index series')
>>> pylab.show()
```

The third accepted data structure is the np array, for which a very simple index will be created.

```python
>>> rand = np.random.random
>>> data = 1+rand((2,2))
>>> mosaic(data, title='random non-labeled array')
>>> pylab.show()
```
If you need to modify the labeling and the coloring you can give a function to create the labels and one with the graphical properties starting from the key tuple

```python
data = {'a': 10, 'b': 15, 'c': 16}
props = lambda key: {'color': 'r' if 'a' in key else 'gray'}
labelizer = lambda k: {('a',): 'first', ('b',): 'second', ('c',): 'third'}[k]
mosaic(data, title='colored dictionary', properties=props, labelizer=labelizer)
```  

Using a DataFrame as source, specifying the name of the columns of interest

```python
gender = ['male', 'male', 'male', 'female', 'female', 'female']
pet = ['cat', 'dog', 'dog', 'cat', 'dog', 'cat']
data = pandas.DataFrame({'gender': gender, 'pet': pet})
mosaic(data, ['pet', 'gender'])
```  

## 3.15 Input-Output

statsmodels offers some functions for input and output. These include a reader for STATA files, a class for generating tables for printing in several formats and two helper functions for pickling.

Users can also leverage the powerful input/output functions provided by pandas.io. Among other things, pandas (a statsmodels dependency) allows reading and writing to Excel, CSV, and HDF5 (PyTables).

### 3.15.1 Examples

**SimpleTable**

**3.15.2 Module Reference**

```python
foreign.StataReader

foreign.StataReader(stata_file, missing_values=False, encoding=None) Stata .dta file reader.

foreign.StataWriter

class foreign.StataWriter(stata_file, data[, ...]) A class for writing Stata binary dta files from array-like objects

foreign.genfromdta

class foreign.genfromdta(stata_file[, missing_flt, ...]) Returns an ndarray or DataFrame from a Stata .dta file.

foreign.savetxt

class foreign.savetxt(stata_file[, missing_flt, ...]) Save an array to a text file.

table.SimpleTable

class table.SimpleTable(data[, headers, stubs, ...]) Produce a simple ASCII, CSV, HTML, or LaTeX table from a DataFrame

table.csv2st

class table.csv2st(csv_file[, headers, stubs, title]) Return SimpleTable instance,

smpickle.save_pickle

class smpickle.save_pickle(obj, fname) Save the object to file via pickling.

smpickle.load_pickle

class smpickle.load_pickle(fname) Load a previously saved object from file
```
encoding : string, optional

Used for Python 3 only. Encoding to use when reading the .dta file. Defaults to locale.getpreferredencoding

See also:
statsmodels.lib.io.genfromdta

Notes

This is known only to work on file formats 113 (Stata 8/9), 114 (Stata 10/11), and 115 (Stata 12). Needs to be tested on older versions. Known not to work on format 104, 108. If you have the documentation for older formats, please contact the developers.

For more information about the .dta format see http://www.stata.com/help.cgi?dta
http://www.stata.com/help.cgi?dta_113

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset([[as_dict])</td>
<td>Returns a Python generator object for iterating over the dataset.</td>
</tr>
<tr>
<td>file_format()</td>
<td>Returns the file format.</td>
</tr>
<tr>
<td>file_headers()</td>
<td>Returns all .dta file headers.</td>
</tr>
<tr>
<td>file_label()</td>
<td>Returns the dataset's label.</td>
</tr>
<tr>
<td>file_timestamp()</td>
<td>Returns the date and time Stata recorded on last file save.</td>
</tr>
<tr>
<td>variables()</td>
<td>Returns a list of the dataset's StataVariables objects.</td>
</tr>
</tbody>
</table>

statsmodels.iolib.foreign.StataReader.dataset

StataReader.dataset (as_dict=False)

Returns a Python generator object for iterating over the dataset.

Parameters as_dict : bool, optional

If as_dict is True, yield each row of observations as a dict. If False, yields each row of observations as a list.

Returns Generator object for iterating over the dataset. Yields each row of observations as a list by default.

Notes

If missing_values is True during instantiation of StataReader then observations with _StataMissing-Value(s) are not filtered and should be handled by your application.

statsmodels.iolib.foreign.StataReader.file_format

StataReader.file_format()

Returns the file format.

Returns out : int
Notes

Format 113: Stata 8/9 Format 114: Stata 10/11 Format 115: Stata 12

statsmodels.iolib.foreign.StataReader.file_headers

StataReader.file_headers()  
Returns all .dta file headers.  

**out**: dict  
Has keys typlist, data_label, lbllist, varlist, nvar, filetype, ds_format, nosb, fmtlist, vlblist,  
time_stamp, srtlist, byteorder

statsmodels.iolib.foreign.StataReader.file_label

StataReader.file_label()  
Returns the dataset’s label.  

Returns **out**: string :

statsmodels.iolib.foreign.StataReader.file_timestamp

StataReader.file_timestamp()  
Returns the date and time Stata recorded on last file save.  

Returns **out**: str

statsmodels.iolib.foreign.StataReader.variables

StataReader.variables()  
Returns a list of the dataset’s StataVariables objects.

Attributes

<table>
<thead>
<tr>
<th>DTYPE_MAP</th>
<th>MISSING_VALUES</th>
<th>TYPE_MAP</th>
<th>list() -&gt; new empty list</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>int(x[, base])</td>
<td>i</td>
<td>int(x[, base]) -&gt; integer</td>
</tr>
</tbody>
</table>

statsmodels.iolib.foreign.StataWriter

class statsmodels.iolib.foreign.StataWriter(fname, data, convert_dates=None, encoding='latin-1', byteorder=None)

A class for writing Stata binary dta files from array-like objects

Parameters **fname**: file path or buffer  
Where to save the dta file.

**data**: array-like  
Array-like input to save. Pandas objects are also accepted.
convert_dates : dict

Dictionary mapping column of datetime types to the stata internal format that you want to use for the dates. Options are ‘tc’, ‘td’, ‘tm’, ‘tw’, ‘th’, ‘tq’, ‘ty’. Column can be either a number or a name.

encoding : str

Default is latin-1. Note that Stata does not support unicode.

byteorder : str

Can be “>”, “<”, “little”, or “big”. The default is None which uses sys.byteorder

Returns writer : StataWriter instance

The StataWriter instance has a write_file method, which will write the file to the given fname.

Examples

>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()

Or with dates

>>> writer = StataWriter('./date_data_file.dta', date, {2 : 'tw'})
>>> writer.write_file()

Methods

write_file()

Attributes

DTYPE_MAP
MISSING_VALUES
TYPE_MAP

list() -> new empty list

int(x[, base]) -> integer

Attributes

statsmodels.iolib.foreign.StataWriter.write_file

StataWriter.write_file()

Attributes

DTYPE_MAP
MISSING_VALUES
TYPE_MAP

list() -> new empty list

int(x[, base]) -> integer

statsmodels.iolib.foreign.genfromdta

Returns an ndarray or DataFrame from a Stata .dta file.

Parameters fname : str or filehandle

Stata .dta file.
missing_flf : numeric

    The numeric value to replace missing values with. Will be used for any numeric value.

encoding : string, optional

    Used for Python 3 only. Encoding to use when reading the .dta file. Defaults to
    locale.getpreferredencoding

pandas : bool

    Optionally return a DataFrame instead of an ndarray

convert_dates : bool

    If convert_dates is True, then Stata formatted dates will be converted to datetime types according to the variable’s format.

statsmodels.iolib.foreign.savetxt

statsmodels.iolib.foreign.savetxt (fname, X, names=None, fmt='%.18e', delimiter=' ')

Save an array to a text file.

This is just a copy of numpy.savetxt patched to support structured arrays or a header of names. Does not include py3 support now in savetxt.

Parameters

fname : filename or file handle

    If the filename ends in .gz, the file is automatically saved in compressed gzip format. loadtxt understands gzipped files transparently.

X : array_like

    Data to be saved to a text file.

names : list, optional

    If given names will be the column header in the text file. If None and X is a structured or recarray then the names are taken from X.dtype.names.

fmt : str or sequence of strs

    A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. ‘Iteration %d – %10.5f’, in which case delimiter is ignored.

delimiter : str

    Character separating columns.

See also:

save  Save an array to a binary file in NumPy .npy format

savez  Save several arrays into a .npz compressed archive

Notes

Further explanation of the fmt parameter (%[flag]width[.precision]specifier):
flags:  - : left justify
   + : Forces to preceed result with + or -.
   0 : Left pad the number with zeros instead of space (see width).

width:  Minimum number of characters to be printed. The value is not truncated if it has more characters.

precision:
   • For integer specifiers (eg. d, i, o, x), the minimum number of digits.
   • For e, E and f specifiers, the number of digits to print after the decimal point.
   • For g and G, the maximum number of significant digits.
   • For s, the maximum number of characters.

specifiers:  c : character
            d or i : signed decimal integer
            e or E : scientific notation with e or E.
            f : decimal floating point
            g, G : use the shorter of e, E or f
            o : signed octal
            s : string of characters
            u : unsigned decimal integer
            x, X : unsigned hexadecimal integer

This explanation of fmt is not complete, for an exhaustive specification see [R3].

References

[R3]

Examples

```python
>>> savetxt('test.out', x, delimiter=',')  # x is an array
>>> savetxt('test.out', (x,y,z))          # x,y,z equal sized 1D arrays
>>> savetxt('test.out', x, fmt='%1.4e')  # use exponential notation
```

statsmodels.iolib.table.SimpleTable

class statsmodels.iolib.table.SimpleTable(data, headers=None, stubs=None, title='', datatypes=None, csv_fmt=None, txt_fmt=None, ltx_fmt=None, html_fmt=None, celltype=None, rowtype=None, **fmt_dict)

Produce a simple ASCII, CSV, HTML, or LaTeX table from a rectangular (2d!) array of data, not necessarily numerical. Directly supports at most one header row, which should be the length of data[0]. Directly supports at most one stubs column, which must be the length of data. (But see insert_stubs method.) See globals default_txt_fmt, default_csv_fmt, default_html_fmt, and default_latex_fmt for formatting options.

Sample uses:
mydata = [[11,12],[21,22]] # data MUST be 2-dimensional
myheaders = [ "Column 1", "Column 2" ]
mystubs = [ "Row 1", "Row 2" ]
tbl = text.SimpleTable(mydata, myheaders, mystubs, title="Title")
print( tbl )
print( tbl.as_html() )
# set column specific data formatting
tbl = text.SimpleTable(mydata, myheaders, mystubs,
    data_fmts=["%3.2f","%d"])
print( tbl.as_csv() )
with open('c:/temp/temp.tex','w') as fh:
   fh.write( tbl.as_latex_tabular() )

Methods

append
\L.append(object) – append object to end

as_csv(**fmt_dict)
Return string, the table in CSV format.

as_html(**fmt_dict)
Return string.

as_latex_tabular([center])
Return string, the table as a LaTeX tabular environment.

as_text(**fmt_dict)
Return string, the table as text.

count(...)
L.extend(iterable) – extend list by appending elements from the iterable
Return None.

extend
Return list, the widths of each column.

get_colwidths(output_format, **fmt_dict)
Raises ValueError if the value is not present.

index((value, [start, ...) Raises ValueError if the value is not present.

insert(idx, row[, datatype])
Return None. Insert a row into a table.

insert_header_row(rownum, headers[, dec_below])
Return None. Insert a row of headers.

insert_stubs(loc, stubs)
Return None. Insert column of stubs at column loc.

label_cells(func)
Return None. Labels cells based on func.

pad(s, width, align)
DEPRECATED: just use the pad function

pop(...)
Raises IndexError if list is empty or index is out of range.

remove
L.remove(value) – remove first occurrence of value.

reverse
L.reverse() – reverse IN PLACE.

sort
L.sort(cmp=None, key=None, reverse=False) – stable sort IN PLACE;

statsmodels.iolib.table.SimpleTable.append

SimpleTable.append()
L.append(object) – append object to end

statsmodels.iolib.table.SimpleTable.as_csv

SimpleTable.as_csv(**fmt_dict)
Return string. The table in CSV format. Currently only supports comma separator.

statsmodels.iolib.table.SimpleTable.as_html

SimpleTable.as_html(**fmt_dict)
Return string. This is the default formatter for HTML tables. An HTML table formatter must accept as arguments a table and a format dictionary.
Statsmodels Documentation, Release 0.6.0

**statsmodels.iolib.table.SimpleTable.as_latex_tabular**

*SimpleTable.as_latex_tabular*(center=True, **fmt_dict)*

Return string, the table as a LaTeX tabular environment. Note: will require the booktabs package.

**statsmodels.iolib.table.SimpleTable.as_text**

*SimpleTable.as_text**(fmt_dict)*

Return string, the table as text.

**statsmodels.iolib.table.SimpleTable.count**

*SimpleTable.count*(value) → integer – return number of occurrences of value

**statsmodels.iolib.table.SimpleTable.extend**

*SimpleTable.extend()*

L.extend(iterable) – extend list by appending elements from the iterable

**statsmodels.iolib.table.SimpleTable.extend_right**

*SimpleTable.extend_right*(table)*

Return None. Extend each row of self with corresponding row of table. Does not import formatting from table. This generally makes sense only if the two tables have the same number of rows, but that is not enforced. :note: To extend append a table below, just use extend, which is the ordinary list method. This generally makes sense only if the two tables have the same number of columns, but that is not enforced.

**statsmodels.iolib.table.SimpleTable.get_colwidths**

*SimpleTable.get_colwidths*(output_format, **fmt_dict)*

Return list, the widths of each column.

**statsmodels.iolib.table.SimpleTable.index**

*SimpleTable.index*(value[, start[, stop]]) → integer – return first index of value.

Raises ValueError if the value is not present.

**statsmodels.iolib.table.SimpleTable.insert**

*SimpleTable.insert*(idx, row, datatype=None)*

Return None. Insert a row into a table.
statsmodels.iolib.table.SimpleTable.insert_header_row

SimpleTable.insert_header_row(rownum, headers, dec_below='header_dec_below')

Return None. Insert a row of headers, where headers is a sequence of strings. (The strings may contain newlines, to indicated multiline headers.)

statsmodels.iolib.table.SimpleTable.insert_stubs

SimpleTable.insert_stubs(loc, stubs)

Return None. Insert column of stubs at column loc. If there is a header row, it gets an empty cell. So len(stubs) should equal the number of non-header rows.

statsmodels.iolib.table.SimpleTable.label_cells

SimpleTable.label_cells(func)

Return None. Labels cells based on func. If func(cell) is None then its datatype is not changed; otherwise it is set to func(cell).

statsmodels.iolib.table.SimpleTable.pad

SimpleTable.pad(s, width, align)

DEPRECATED: just use the pad function

statsmodels.iolib.table.SimpleTable.pop

SimpleTable.pop(index) → item – remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

statsmodels.iolib.table.SimpleTable.remove

SimpleTable.remove()

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

statsmodels.iolib.table.SimpleTable.reverse

SimpleTable.reverse()

L.reverse() – reverse IN PLACE

statsmodels.iolib.table.SimpleTable.sort

SimpleTable.sort()

L.sort(cmp=None, key=None, reverse=False) – stable sort IN PLACE; cmp(x, y) -> -1, 0, 1

Attributes
3.16 Tools

Our tool collection contains some convenience functions for users and functions that were written mainly for internal use.

Additional to this tools directory, several other subpackages have their own tools modules, for example 
\texttt{statsmodels.tsa.tsatools}

3.16.1 Module Reference

Basic tools \texttt{tools}

These are basic and miscellaneous tools. The full import path is \texttt{statsmodels.tools.tools}.

\begin{verbatim}
\texttt{tools.add_constant(data[, prepend])}  This appends a column of ones to an array if prepend==False.
\end{verbatim}
**statsmodels Documentation, Release 0.6.0**

**statsmodels.tools.tools.add_constant**

*statsmodels.tools.tools.add_constant*(data, prepend=True)  
This appends a column of ones to an array if prepend=False.

For ndarrays and pandas.DataFrame, checks to make sure a constant is not already included. If there is at least one column of ones then the original object is returned. Does not check for a constant if a structured or recarray is given.

**Parameters**  
- **data**: array-like  
  *data* is the column-ordered design matrix  
- **prepend**: bool  
  True and the constant is prepended rather than appended.

**Returns**  
- **data**: array  
  The original array with a constant (column of ones) as the first or last column.

The next group are mostly helper functions that are not separately tested or insufficiently tested.

---

**tools.categorical**  
*tools.categorical*(data[, col, dictnames, drop])  
Returns a dummy matrix given an array of categorical variables.

**Parameters**  
- **data**: array  
  A structured array, recarray, or array. This can be either a 1d vector of the categorical variable or a 2d array with the column specifying the categorical variable specified by the col argument.  
- **col**: ‘string’, int, or None  
  If data is a structured array or a recarray, *col* can be a string that is the name of the column that contains the variable. For all arrays *col* can be an int that is the (zero-based) column index number. *col* can only be None for a 1d array. The default is None.  
- **dictnames**: bool, optional  
  If True, a dictionary mapping the column number to the categorical name is returned. Used to have information about plain arrays.  
- **drop**: bool  
  Whether or not keep the categorical variable in the returned matrix.
Returns dummy_matrix, [dictnames, optional]:

A matrix of dummy (indicator/binary) float variables for the categorical data. If
dictnames is True, then the dictionary is returned as well.

Notes

This returns a dummy variable for EVERY distinct variable. If a structured or recarray is provided, the names
for the new variable is the old variable name - underscore - category name. So if the a variable ‘vote’ had
answers as ‘yes’ or ‘no’ then the returned array would have to new variables-- ‘vote_yes’ and ‘vote_no’. There
is currently no name checking.

Examples

```python
>>> import numpy as np
>>> import statsmodels.api as sm

Univariate examples

>>> import string

>>> string_var = [string.lowercase[0:5], string.lowercase[5:10], string.lowercase[10:15], string.lowercase[15:20], string.lowercase[20:25]]
>>> string_var *= 5
>>> string_var = np.asarray(sorted(string_var))
>>> design = sm.tools.categorical(string_var, drop=True)

Or for a numerical categorical variable

>>> instr = np.floor(np.arange(10,60, step=2)/10)
>>> design = sm.tools.categorical(instr, drop=True)

With a structured array

>>> num = np.random.randn(25,2)
>>> struct_ar = np.zeros((25,1), dtype=[('var1', 'f4'), ('var2', 'f4'), ('instrument','f4'),('str_instr','a5')])
>>> struct_ar['var1'] = num[:,0][,None]
>>> struct_ar['var2'] = num[:,1][,None]
>>> struct_ar['instrument'] = instr[:,None]
>>> struct_ar['str_instr'] = string_var[:,None]
>>> design = sm.tools.categorical(struct_ar, col='instrument', drop=True)

Or

>>> design2 = sm.tools.categorical(struct_ar, col='str_instr', drop=True)
```

statsmodels.tools.tools.ECDF

statsmodels.tools.tools.ECDF(*args, **kwds)

statsmodels.tools.tools.ECDF is deprecated, use statsmodels.distributions.ECDF instead!

Return the Empirical CDF of an array as a step function.

Parameters x : array-like

Observations

side : {'left', 'right'}, optional
Default is ‘right’. Defines the shape of the intervals constituting the steps. ‘right’
correspond to [a, b) intervals and ‘left’ to (a, b).

Returns Empirical CDF as a step function.

Examples

```python
>>> import numpy as np
>>> from statsmodels.distributions.empirical_distribution import ECDF
>>> ecdf = ECDF([3, 3, 1, 4])
>>> ecdf([3, 55, 0.5, 1.5])
array([ 0.75, 1. , 0. , 0.25])
```

```python
statsmodels.tools.tools.clean0

statsmodels.tools.tools.clean0(matrix)
Erase columns of zeros: can save some time in pseudoinverse.

statsmodels.tools.tools.fullrank

statsmodels.tools.tools.fullrank(X, r=None)
Return a matrix whose column span is the same as X.
If the rank of X is known it can be specified as r – no check is made to ensure that this really is the rank of X.

statsmodels.tools.tools.isestimable

statsmodels.tools.tools.isestimable(C, D)
True if (Q, P) contrast C is estimable for (N, P) design D
From an Q x P contrast matrix C and an N x P design matrix D, checks if the contrast C is estimable by looking
at the rank of vstack([C, D]) and verifying it is the same as the rank of D.

Parameters C: (Q, P) array-like
contrast matrix. If C has is 1 dimensional assume shape (1, P)

D: (N, P) array-like :
design matrix

Returns tf: bool
True if the contrast C is estimable on design D

Examples

```python
>>> D = np.array([[1, 1, 1, 0, 0, 0],
...                [0, 0, 0, 1, 1, 1],
...                [1, 1, 1, 1, 1, 1]]).T
>>> isestimable([1, 0, 0], D)
False
```
>>> isestimable([1, -1, 0], D)
True

statsmodels.tools.tools.monotone_fn_inverter

statsmodels.tools.tools.monotone_fn_inverter(*args, **kwds)

statsmodels.tools.tools.monotone_fn_inverter is deprecated, use statsmodels.distributions.monotone_fn_inverter instead!

Given a monotone function x (no checking is done to verify monotonicity) and a set of x values, return an linearly interpolated approximation to its inverse from its values on x.

statsmodels.tools.tools.rank

statsmodels.tools.tools.rank(X, cond=1e-12)

Return the rank of a matrix X based on its generalized inverse, not the SVD.

statsmodels.tools.tools.recipr

statsmodels.tools.tools.recipr(X)

Return the reciprocal of an array, setting all entries less than or equal to 0 to 0. Therefore, it presumes that X should be positive in general.

statsmodels.tools.tools.recipr0

statsmodels.tools.tools.recipr0(X)

Return the reciprocal of an array, setting all entries equal to 0 as 0. It does not assume that X should be positive in general.

statsmodels.tools.tools.unsqueez

statsmodels.tools.tools.unsqueez(data, axis, oldshape)

Unsqueeze a collapsed array

>>> from numpy import mean
>>> from numpy.random import standard_normal
>>> x = standard_normal((3,4,5))
>>> m = mean(x, axis=1)
>>> m.shape
(3, 5)
>>> m = unsqueez(m, 1, x.shape)
>>> m.shape
(3, 1, 5)

Numerical Differentiation

numdiff.approx_fprime(x, f[, epsilon, args, ...])  Gradient of function, or Jacobian if function f returns 1d array
Table 3.285 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numdiff.approx_fprime_cs</code></td>
<td>Calculate gradient or Jacobian with complex step derivative approximation</td>
</tr>
<tr>
<td><code>numdiff.approx_hess1</code></td>
<td>Calculate Hessian with finite difference derivative approximation</td>
</tr>
<tr>
<td><code>numdiff.approx_hess2</code></td>
<td>Calculate Hessian with finite difference derivative approximation</td>
</tr>
<tr>
<td><code>numdiff.approx_hess3</code></td>
<td>Calculate Hessian with finite difference derivative approximation</td>
</tr>
<tr>
<td><code>numdiff.approx_hess_cs</code></td>
<td>Calculate Hessian with complex-step derivative approximation</td>
</tr>
</tbody>
</table>

**statsmodels.tools.numdiff.approx_fprime**

`statsmodels.tools.numdiff.approx_fprime(x, f, epsilon=None, args=(), kwargs={}, centered=False)`

Gradient of function, or Jacobian if function f returns 1d array

**Parameters**

- `x`: array
  - parameters at which the derivative is evaluated
- `f`: function
  - `f*((x,)+args, **kwargs)` returning either one value or 1d array
- `epsilon`: float, optional
  - Stepsize, if None, optimal stepsize is used. This is EPS**(1/2)*x for `centered` == False and EPS**(1/3)*x for `centered` == True.
- `args`: tuple
  - Tuple of additional arguments for function `f`
- `kwargs`: dict
  - Dictionary of additional keyword arguments for function `f`
- `centered`: bool
  - Whether central difference should be returned. If not, does forward differencing.

**Returns**

- `grad`: array
  - gradient or Jacobian

**Notes**

If `f` returns a 1d array, it returns a Jacobian. If a 2d array is returned by `f` (e.g., with a value for each observation), it returns a 3d array with the Jacobian of each observation with shape `xk x noobs x xk`. I.e., the Jacobian of the first observation would be `[:, 0, :]`

**statsmodels.tools.numdiff.approx_fprime_cs**

`statsmodels.tools.numdiff.approx_fprime_cs(x, f, epsilon=None, args=(), kwargs={})`

Calculate gradient or Jacobian with complex step derivative approximation

**Parameters**

- `x`: array
  - parameters at which the derivative is evaluated
- `f`: function
  - `f*((x,)+args, **kwargs)` returning either one value or 1d array
epsilon : float, optional
    Stepsize, if None, optimal stepsize is used. Optimal step-size is EPS*x. See note.
args : tuple
    Tuple of additional arguments for function f.
kwvars : dict
    Dictionary of additional keyword arguments for function f.

Returns  partials : ndarray
    array of partial derivatives, Gradient or Jacobian

Notes
The complex-step derivative has truncation error O(epsilon**2), so truncation error can be eliminated by choosing epsilon to be very small. The complex-step derivative avoids the problem of round-off error with small epsilon because there is no subtraction.

```
statsmodels.tools.numdiff.approx_hess1
```

Calculate Hessian with finite difference derivative approximation

Parameters  x : array_like
    value at which function derivative is evaluated
f : function
    function of one array f(x, *args, **kwargs)
epsilon : float or array-like, optional
    Stepsize used, if None, then stepsize is automatically chosen according to EPS**(1/3)*x.
args : tuple
    Arguments for function f.
kwvars : dict
    Keyword arguments for function f.
return_grad : bool
    Whether or not to also return the gradient

Returns  hess : ndarray
    array of partial second derivatives, Hessian
grad : nparray
    Gradient if return_grad == True
Notes

Equation (7) in Ridout. Computes the Hessian as:

\[ \frac{1}{d_j d_k} \left( \frac{f(x + d[j]e[j] + d[k]e[k]) - f(x + d[j]e[j])}{d[j]e[j]} \right) \]

where \( e[j] \) is a vector with element \( j = 1 \) and the rest are zero and \( d[i] \) is epsilon[i].

References


\[
\text{statsmodels.tools.numdiff.approx_hess2}
\]

\[
\text{statsmodels.tools.numdiff.approx_hess2}(x, f, \text{epsilon=None, args=(), kwargs={}, return_grad=False})
\]

Calculate Hessian with finite difference derivative approximation

Parameters

- **x**: array_like
  - value at which function derivative is evaluated
- **f**: function
  - function of one array \( f(x, \text{*args, **kwargs}) \)
- **epsilon**: float or array-like, optional
  - Stepsize used, if None, then stepsize is automatically chosen according to \( \text{EPS}^{(1/3)}x \).
- **args**: tuple
  - Arguments for function \( f \).
- **kwargs**: dict
  - Keyword arguments for function \( f \).
- **return_grad**: bool
  - Whether or not to also return the gradient

Returns

- **hess**: ndarray
  - array of partial second derivatives, Hessian
- **grad**: ndarray
  - Gradient if \( \text{return_grad == True} \)

Notes

Equation (8) in Ridout. Computes the Hessian as:

\[ \frac{1}{2d_j d_k} \left( \frac{f(x + d[j]e[j] + d[k]e[k]) - f(x + d[j]e[j]) - (f(x - d[j]e[j] - d[k]e[k]) - f(x + d[j]e[j]))}{d[j]e[j]} \right) + \]
where e[j] is a vector with element j == 1 and the rest are zero and d[i] is epsilon[i].

References


```
statsmodels.tools.numdiff.approx_hess3
```

Calculate Hessian with finite difference derivative approximation

Parameters

- **x**: array_like
  value at which function derivative is evaluated
- **f**: function
  function of one array f(x, *args, **kwargs)
- **epsilon**: float or array-like, optional
  Stepsize used, if None, then stepsize is automatically chosen according to EPS**(1/4)*x.
- **args**: tuple
  Arguments for function f.
- **kwargs**: dict
  Keyword arguments for function f.

Returns

- **hess**: ndarray
  array of partial second derivatives, Hessian

Notes

Equation (9) in Ridout. Computes the Hessian as:

\[
\frac{1}{4*d_j*d_k} \left( (f(x + d[j]*e[j] + d[k]*e[k]) - f(x + d[j]*e[j]) - d[k]*e[k])) - (f(x - d[j]*e[j] + d[k]*e[k]) - f(x - d[j]*e[j]) - d[k]*e[k]) \right)
\]

where e[j] is a vector with element j == 1 and the rest are zero and d[i] is epsilon[i].

References


This is an alias for approx_hess3
Calculate Hessian with complex-step derivative approximation

Equation (10) in Ridout. Computes the Hessian as:

\[
\frac{1}{2d_j d_k} \times \text{imag}(f(x + i d[j] e[j] + d[k] e[k]) - f(x + i d[j] e[j] - d[k] e[k]))
\]

where e[j] is a vector with element j == 1 and the rest are zero and d[i] is epsilon[i].

References


Measure for fit performance eval_measures

The first group of function in this module are standalone versions of information criteria, aic bic and hqic. The function with _sigma suffix take the error sum of squares as argument, those without, take the value of the log-likelihood, llf, as argument.

The second group of function are measures of fit or prediction performance, which are mostly one liners to be used as helper functions. All of those calculate a performance or distance statistic for the difference between two arrays. For example in the case of Monte Carlo or cross-validation, the first array would be the estimation results for the different replications or draws, while the second array would be the true or observed values.
### Table 3.286 – continued from previous page

<table>
<thead>
<tr>
<th>eval_measures.aic_sigma</th>
<th>Akaike information criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval_measures.aicc</td>
<td>Akaike information criterion (AIC) with small sample correction</td>
</tr>
<tr>
<td>eval_measures.aicc_sigma</td>
<td>Akaike information criterion (AIC) with small sample correction</td>
</tr>
<tr>
<td>eval_measures.bic</td>
<td>Bayesian information criterion (BIC) or Schwarz criterion</td>
</tr>
<tr>
<td>eval_measures.bic_sigma</td>
<td>Bayesian information criterion (BIC) or Schwarz criterion</td>
</tr>
<tr>
<td>eval_measures.hqic</td>
<td>Hannan-Quinn information criterion (HQC)</td>
</tr>
<tr>
<td>eval_measures.hqic_sigma</td>
<td>Hannan-Quinn information criterion (HQC)</td>
</tr>
<tr>
<td>eval_measures.bias</td>
<td>bias, mean error</td>
</tr>
<tr>
<td>eval_measures.iqr</td>
<td>interquartile range of error</td>
</tr>
<tr>
<td>eval_measures.maxabs</td>
<td>maximum absolute error</td>
</tr>
<tr>
<td>eval_measures.meanabs</td>
<td>mean absolute error</td>
</tr>
<tr>
<td>eval_measures.medianabs</td>
<td>median absolute error</td>
</tr>
<tr>
<td>eval_measures.medianbias</td>
<td>median bias, median error</td>
</tr>
<tr>
<td>eval_measures.mse</td>
<td>mean squared error</td>
</tr>
<tr>
<td>eval_measures.rmse</td>
<td>root mean squared error</td>
</tr>
<tr>
<td>eval_measures.stde</td>
<td>standard deviation of error</td>
</tr>
<tr>
<td>eval_measures.vare</td>
<td>variance of error</td>
</tr>
</tbody>
</table>

### statsmodels.tools.eval_measures.aic

**statsmodels.tools.eval_measures.aic(llf, obs, df_modelwc)**  
Akaike information criterion  

**Parameters**  
- **llf** : float  
  value of the loglikelihood  
- **obs** : int  
  number of observations  
- **df_modelwc** : int  
  number of parameters including constant  

**Returns**  
- **aic** : float  
  information criterion

**References**  
http://en.wikipedia.org/wiki/Akaike_information_criterion

### statsmodels.tools.eval_measures.aic_sigma

**statsmodels.tools.eval_measures.aic_sigma(sigma2, obs, df_modelwc, islog=False)**  
Akaike information criterion  

**Parameters**  
- **sigma2** : float  
  estimate of the residual variance or determinant of Sigma_hat in the multivariate case. If islog is true, then it is assumed that sigma is already log-ed, for example logdetSigma.  
- **obs** : int
number of observations

def_modelwc : int
    number of parameters including constant

Returns aic : float
    information criterion

Notes

A constant has been dropped in comparison to the loglikelihood base information criteria. The information criteria should be used to compare only comparable models.

For example, AIC is defined in terms of the loglikelihood as

\[ -2\log l + 2k \]

in terms of \( \hat{\sigma}^2 \)

\[ \log(\hat{\sigma}^2) + 2k/n \]

in terms of the determinant of \( \hat{\Sigma} \)

\[ \log(||\hat{\Sigma}||) + 2k/n \]

Note: In our definition we do not divide by n in the log-likelihood version.

TODO: Latex math

reference for example lecture notes by Herman Bierens

References

http://en.wikipedia.org/wiki/Akaike_information_criterion

statsmodels.tools.eval_measures.aicc

statsmodels.tools.eval_measures.aicc(llf, nobs, df_modelwc)
    Akaike information criterion (AIC) with small sample correction

Parameters llf : float
    value of the loglikelihood

nobs : int
    number of observations

def_modelwc : int
    number of parameters including constant

Returns aicc : float
    information criterion

References

http://en.wikipedia.org/wiki/Akaike_information_criterion#AICc
statsmodels.tools.eval_measures.aicc_sigma

statsmodels.tools.eval_measures.aicc_sigma(sigma2, nobs, df_modelwc, islog=False)

Akaike information criterion (AIC) with small sample correction

Parameters

sigma2 : float
    estimate of the residual variance or determinant of Sigma_hat in the multivariate case. If islog is true, then it is assumed that sigma is already log-ed, for example logdetSigma.

nobs : int
    number of observations

df_modelwc : int
    number of parameters including constant

Returns

aicc : float
    information criterion

Notes

A constant has been dropped in comparison to the loglikelihood base information criteria. These should be used to compare for comparable models.

References

http://en.wikipedia.org/wiki/Akaike_information_criterion#AICc

statsmodels.tools.eval_measures.bic

statsmodels.tools.eval_measures.bic(llf, nobs, df_modelwc)

Bayesian information criterion (BIC) or Schwarz criterion

Parameters

llf : float
    value of the loglikelihood

nobs : int
    number of observations

df_modelwc : int
    number of parameters including constant

Returns

bic : float
    information criterion

References

http://en.wikipedia.org/wiki/Bayesian_information_criterion
statsmodels.tools.eval_measures.bic_sigma

**statsmodels.tools.eval_measures.bic_sigma***(sigma2, nobs, df_modelwc, islog=False)***

Bayesian information criterion (BIC) or Schwarz criterion

**Parameters**

- **sigma2**: float
  estimate of the residual variance or determinant of Sigma_hat in the multivariate case. If islog is true, then it is assumed that sigma is already log-ed, for example logdetSigma.

- **nobs**: int
  number of observations

- **df_modelwc**: int
  number of parameters including constant

**Returns**

- **bic**: float
  information criterion

**Notes**

A constant has been dropped in comparison to the loglikelihood base information criteria. These should be used to compare for comparable models.

**References**


statsmodels.tools.eval_measures.hqic

**statsmodels.tools.eval_measures.hqic***(llf, nobs, df_modelwc)***

Hannan-Quinn information criterion (HQC)

**Parameters**

- **llf**: float
  value of the loglikelihood

- **nobs**: int
  number of observations

- **df_modelwc**: int
  number of parameters including constant

**Returns**

- **hqic**: float
  information criterion

**References**

- Wikipedia doesn’t say much
statsmodels.tools.eval_measures.hqic_sigma

statsmodels.tools.eval_measures.hqic_sigma(sigma2, nobs, df_modelwc, islog=False)

Hannan-Quinn information criterion (HQC)

Parameters

- **sigma2**: float
  estimate of the residual variance or determinant of Sigma_hat in the multivariate case. If islog is true, then it is assumed that sigma is already log-ed, for example logdetSigma.

- **nobs**: int
  number of observations

- **df_modelwc**: int
  number of parameters including constant

Returns

- **hqic**: float
  information criterion

Notes

A constant has been dropped in comparison to the loglikelihood base information criteria. These should be used to compare for comparable models.

References

xxx

statsmodels.tools.eval_measures.bias

statsmodels.tools.eval_measures.bias(x1, x2, axis=0)

bias, mean error

Parameters

- **x1, x2**: array_like
  The performance measure depends on the difference between these two arrays.

- **axis**: int
  axis along which the summary statistic is calculated

Returns

- **bias**: ndarray or float
  bias, or mean difference along given axis.

Notes

If x1 and x2 have different shapes, then they need to broadcast. This uses numpy.asanyarray to convert the input. Whether this is the desired result or not depends on the array subclass.
statsmodels.tools.eval_measures.iqr

\[ \text{statsmodels.tools.eval_measures.iqr}(x1, x2, axis=0) \]

interquartile range of error
rounded index, no interpolations
this could use newer numpy function instead

**Parameters**

- \( x1, x2 \) : array_like
  The performance measure depends on the difference between these two arrays.
- \( axis \) : int
  axis along which the summary statistic is calculated

**Returns**

- \( \text{mse} \) : ndarray or float
  mean squared error along given axis.

**Notes**

If \( x1 \) and \( x2 \) have different shapes, then they need to broadcast.

This uses `numpy.asarray` to convert the input, in contrast to the other functions in this category.

---

statsmodels.tools.eval_measures.maxabs

\[ \text{statsmodels.tools.eval_measures.maxabs}(x1, x2, axis=0) \]

maximum absolute error

**Parameters**

- \( x1, x2 \) : array_like
  The performance measure depends on the difference between these two arrays.
- \( axis \) : int
  axis along which the summary statistic is calculated

**Returns**

- \( \text{maxabs} \) : ndarray or float
  maximum absolute difference along given axis.

**Notes**

If \( x1 \) and \( x2 \) have different shapes, then they need to broadcast. This uses `numpy.asanyarray` to convert the input. Whether this is the desired result or not depends on the array subclass.

---

statsmodels.tools.eval_measures.meanabs

\[ \text{statsmodels.tools.eval_measures.meanabs}(x1, x2, axis=0) \]

mean absolute error

**Parameters**

- \( x1, x2 \) : array_like
  The performance measure depends on the difference between these two arrays.
- \( axis \) : int
axis along which the summary statistic is calculated

**Returns meanabs**: ndarray or float
mean absolute difference along given axis.

**Notes**

If \( x_1 \) and \( x_2 \) have different shapes, then they need to broadcast. This uses `numpy.asanyarray` to convert the input. Whether this is the desired result or not depends on the array subclass.

**statsmodels.tools.eval_measures.medianabs**

```python
statsmodels.tools.eval_measures.medianabs(x1, x2, axis=0)
```

median absolute error

**Parameters**

- **x1, x2**: array_like
  The performance measure depends on the difference between these two arrays.
- **axis**: int
  axis along which the summary statistic is calculated

**Returns medianabs**: ndarray or float
median absolute difference along given axis.

**Notes**

If \( x_1 \) and \( x_2 \) have different shapes, then they need to broadcast. This uses `numpy.asanyarray` to convert the input. Whether this is the desired result or not depends on the array subclass.

**statsmodels.tools.eval_measures.medianbias**

```python
statsmodels.tools.eval_measures.medianbias(x1, x2, axis=0)
```

median bias, median error

**Parameters**

- **x1, x2**: array_like
  The performance measure depends on the difference between these two arrays.
- **axis**: int
  axis along which the summary statistic is calculated

**Returns medianbias**: ndarray or float
median bias, or median difference along given axis.

**Notes**

If \( x_1 \) and \( x_2 \) have different shapes, then they need to broadcast. This uses `numpy.asanyarray` to convert the input. Whether this is the desired result or not depends on the array subclass.
statsmodels tools.eval_measures.mse

statsmodels.tools.eval_measures.mse(x1, x2, axis=0)
mean squared error

Parameters x1, x2 : array_like
The performance measure depends on the difference between these two arrays.

axis : int
axis along which the summary statistic is calculated

Returns mse : ndarray or float
mean squared error along given axis.

Notes
If x1 and x2 have different shapes, then they need to broadcast. This uses numpy.asanyarray to convert the input. Whether this is the desired result or not depends on the array subclass, for example numpy matrices will silently produce an incorrect result.

statsmodels tools.eval_measures.rmse

statsmodels.tools.eval_measures.rmse(x1, x2, axis=0)
root mean squared error

Parameters x1, x2 : array_like
The performance measure depends on the difference between these two arrays.

axis : int
axis along which the summary statistic is calculated

Returns rmse : ndarray or float
root mean squared error along given axis.

Notes
If x1 and x2 have different shapes, then they need to broadcast. This uses numpy.asanyarray to convert the input. Whether this is the desired result or not depends on the array subclass, for example numpy matrices will silently produce an incorrect result.

statsmodels tools.eval_measures.stde

statsmodels.tools.eval_measures.stde(x1, x2, ddof=0, axis=0)
standard deviation of error

Parameters x1, x2 : array_like
The performance measure depends on the difference between these two arrays.

axis : int
axis along which the summary statistic is calculated
Returns stde : ndarray or float
standard deviation of difference along given axis.

Notes
If x1 and x2 have different shapes, then they need to broadcast. This uses numpy.asanyarray to convert the input. Whether this is the desired result or not depends on the array subclass.

statsmodels.tools.eval_measures.vare

statsmodels.tools.eval_measures.vare(x1, x2, ddof=0, axis=0)
variance of error

Parameters x1, x2 : array_like
The performance measure depends on the difference between these two arrays.
axis : int
axis along which the summary statistic is calculated

Returns vare : ndarray or float
variance of difference along given axis.

Notes
If x1 and x2 have different shapes, then they need to broadcast. This uses numpy.asanyarray to convert the input. Whether this is the desired result or not depends on the array subclass.

3.17 The Datasets Package

statsmodels provides data sets (i.e. data and meta-data) for use in examples, tutorials, model testing, etc.

3.17.1 Using Datasets from R

The Rdatasets project gives access to the datasets available in R’s core datasets package and many other common R packages. All of these datasets are available to statsmodels by using the get_rdataset function. The actual data is accessible by the data attribute. For example:

In [1]: import statsmodels.api as sm
In [2]: duncan_prestige = sm.datasets.get_rdataset("Duncan", "car")
In [3]: print duncan_prestige.__doc__
+----------+-------------------+
| Duncan | R Documentation   |
+----------+-------------------+
Duncan’s Occupational Prestige Data
Description
~~~~~~~~~~~

The `Duncan` data frame has 45 rows and 4 columns. Data on the prestige and other characteristics of 45 U. S. occupations in 1950.

Usage
~~~~~~

::

    Duncan

Format
~~~~~~

This data frame contains the following columns:

- **type**
  - Type of occupation. A factor with the following levels: 'prof', professional and managerial; 'wc', white-collar; 'bc', blue-collar.

- **income**
  - Percent of males in occupation earning $3500 or more in 1950.

- **education**
  - Percent of males in occupation in 1950 who were high-school graduates.

- **prestige**
  - Percent of raters in NORC study rating occupation as excellent or good in prestige.

Source
~~~~~~


References
~~~~~~~~~~


In [4]: duncan_prestige.data.head(5)
Out[4]:

<table>
<thead>
<tr>
<th>type</th>
<th>income</th>
<th>education</th>
<th>prestige</th>
</tr>
</thead>
<tbody>
<tr>
<td>accountant</td>
<td>prof</td>
<td>62</td>
<td>86</td>
</tr>
<tr>
<td>pilot</td>
<td>prof</td>
<td>72</td>
<td>76</td>
</tr>
<tr>
<td>architect</td>
<td>prof</td>
<td>75</td>
<td>92</td>
</tr>
<tr>
<td>author</td>
<td>prof</td>
<td>55</td>
<td>90</td>
</tr>
</tbody>
</table>
### 3.17.2 R Datasets Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statsmodels.datasets.get_rdataset</td>
<td>download and return R dataset</td>
</tr>
<tr>
<td>statsmodels.datasets.get_rdataset</td>
<td>(dataname[, package, cache])</td>
</tr>
<tr>
<td>statsmodels.datasets.get_rdataset</td>
<td>Return the path of the statsmodels data dir.</td>
</tr>
<tr>
<td>statsmodels.datasets.get_rdataset</td>
<td>(data_home)</td>
</tr>
<tr>
<td>statsmodels.datasets.get_rdataset</td>
<td>Delete all the content of the data home cache.</td>
</tr>
</tbody>
</table>

#### statsmodels.datasets.get_rdataset

```python
statsmodels.datasets.get_rdataset(dataname, package='datasets', cache=False)
```

download and return R dataset

**Parameters**

- **dataname** : str
  - The name of the dataset you want to download

- **package** : str
  - The package in which the dataset is found. The default is the core ‘datasets’ package.

- **cache** : bool or str
  - If True, will download this data into the STATSMODELS_DATA folder. The default location is a folder called statsmodels_data in the user home folder. Otherwise, you can specify a path to a folder to use for caching the data. If False, the data will not be cached.

**Returns**

- **dataset** : Dataset instance
  - A statsmodels.data.utils.Dataset instance. This objects has attributes:
    - * data - A pandas DataFrame containing the data
    - * title - The dataset title
    - * package - The package from which the data came
    - * from_cache - Whether not cached data was retrieved
    - * __doc__ - The verbatim R documentation.

**Notes**

If the R dataset has an integer index. This is reset to be zero-based. Otherwise the index is preserved. The caching facilities are dumb. That is, no download dates, e-tags, or otherwise identifying information is checked to see if the data should be downloaded again or not. If the dataset is in the cache, it’s used.

#### statsmodels.datasets.get_data_home

```python
statsmodels.datasets.get_data_home(data_home=None)
```
Return the path of the statsmodels data dir.

This folder is used by some large dataset loaders to avoid downloading the data several times.

By default the data dir is set to a folder named ‘statsmodels_data’ in the user home folder.
Alternatively, it can be set by the ‘STATSMODELS_DATA’ environment variable or programatically by giving an explicit folder path. The ‘~’ symbol is expanded to the user home folder.

If the folder does not already exist, it is automatically created.

```
statsmodels.datasets.clear_data_home
```

Delete all the content of the data home cache.

### 3.17.3 Available Datasets

### 3.17.4 Usage

Load a dataset:

```
In [5]: import statsmodels.api as sm

In [6]: data = sm.datasets.longley.load()
```

The `Dataset` object follows the bunch pattern explained in proposal. The full dataset is available in the `data` attribute.

```
In [7]: data.data
Out[7]:
rec.array([(60323.0, 83.0, 234289.0, 2356.0, 1590.0, 107608.0, 1947.0),
(61122.0, 88.5, 259426.0, 2325.0, 1456.0, 108632.0, 1948.0),
(60171.0, 88.2, 258054.0, 3682.0, 1616.0, 109773.0, 1949.0),
(61187.0, 89.5, 284599.0, 3351.0, 1650.0, 110929.0, 1950.0),
(63221.0, 96.2, 328975.0, 2099.0, 3099.0, 112075.0, 1951.0),
(63639.0, 98.1, 346999.0, 1932.0, 3594.0, 113270.0, 1952.0),
(64998.0, 99.0, 365385.0, 1870.0, 3547.0, 115094.0, 1953.0),
(63761.0, 100.0, 363112.0, 3578.0, 3350.0, 116219.0, 1954.0),
(66019.0, 101.2, 397469.0, 2904.0, 3048.0, 117388.0, 1955.0),
(67857.0, 104.6, 419180.0, 2822.0, 2857.0, 119834.0, 1956.0),
(68169.0, 108.4, 442769.0, 2936.0, 2798.0, 120445.0, 1957.0),
(66513.0, 110.8, 444546.0, 4681.0, 2637.0, 121950.0, 1958.0),
(68655.0, 112.6, 482704.0, 3813.0, 2552.0, 123366.0, 1959.0),
(69564.0, 114.2, 502601.0, 3931.0, 2514.0, 125368.0, 1960.0),
(69331.0, 115.7, 518173.0, 4806.0, 2572.0, 127852.0, 1961.0),
(70551.0, 116.9, 554894.0, 4007.0, 2827.0, 130081.0, 1962.0)],
dtype=[('TOTEMP', '<f8'), ('GNPDEFL', '<f8'), ('GNP', '<f8'), ('UNEMP', '<f8'), ('ARMED', '<f8')])
```

Most datasets hold convenient representations of the data in the attributes `endog` and `exog`:

```
In [8]: data.endog[:5]
Out[8]:
array([ 83. , 234289. , 2356. , 1590. , 107608. , 1947. ],
[ 88.5, 259426. , 2325.0, 1456.0, 108632.0, 1948.0],
[ 88.2, 258054.0, 3682.0, 1616.0, 109773.0, 1949.0],
[ 89.5, 284599.0, 3351.0, 1650.0, 110929.0, 1950.0],
[ 96.2, 328975.0, 2099.0, 3099.0, 112075.0, 1951.0])
```

1022  Chapter 3.  Table of Contents
Univariate datasets, however, do not have an \texttt{exog} attribute.

Variable names can be obtained by typing:

\begin{verbatim}
In [10]: data.endog_name
Out[10]: 'TOTEMP'

In [11]: data.exog_name
Out[11]: ['GNPDEFL', 'GNP', 'UNEMP', 'ARMED', 'POP', 'YEAR']
\end{verbatim}

If the dataset does not have a clear interpretation of what should be an \textit{endog} and \textit{exog}, then you can always access the \texttt{data} or \texttt{raw_data} attributes. This is the case for the \texttt{macrodata} dataset, which is a collection of US macroeconomic data rather than a dataset with a specific example in mind. The \texttt{data} attribute contains a record array of the full dataset and the \texttt{raw_data} attribute contains an ndarray with the names of the columns given by the \texttt{names} attribute.

\begin{verbatim}
In [12]: type(data.data)
Out[12]: numpy.core.records.recarray

In [13]: type(data.raw_data)
Out[13]: numpy.ndarray

In [14]: data.names
Out[14]: ['TOTEMP', 'GNPDEFL', 'GNP', 'UNEMP', 'ARMED', 'POP', 'YEAR']
\end{verbatim}

\section*{Loading data as pandas objects}

For many users it may be preferable to get the datasets as a pandas DataFrame or Series object. Each of the dataset modules is equipped with a \texttt{load_pandas} method which returns a \texttt{Dataset} instance with the data readily available as pandas objects:

\begin{verbatim}
In [15]: data = sm.datasets.longley.load_pandas()

In [16]: data.exog
Out[16]:
   GNPDEFL   GNP  UNEMP  ARMED   POP  YEAR
0   83.0   234289  2356  1590  107608  1947
1   88.5   259426  2325  1456  108632  1948
2   88.2   258054  3682  1616  109773  1949
3   89.5   284599  3351  1650  110929  1950
4   96.2   328975  2099  3099  112075  1951
5   98.1   346999  1932  3594  113270  1952
6   99.0   365385  1870  3547  115094  1953
7  100.0   363112  3578  3350  116219  1954
8  101.2   397469  2904  3048  117388  1955
9  104.6   419180  2822  2857  118734  1956
10  108.4   442769  2936  2798  120445  1957
11  110.8   444546  4681  2637  121950  1958
12  112.6   482704  3813  2552  123366  1959
13  114.2   502601  3931  2514  125368  1960
14  115.7   518173  4806  2572  127852  1961
15  116.9   554894  4007  2827  130081  1962

In [17]: data.endog
Out[17]:
0   60323
1   61122
\end{verbatim}
The full DataFrame is available in the `data` attribute of the Dataset object

In [18]: data.data

Out[18]:

<table>
<thead>
<tr>
<th>TOTEMP</th>
<th>GNPDEFL</th>
<th>GNP</th>
<th>UNEMP</th>
<th>ARMED</th>
<th>POP</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>60323</td>
<td>83.0</td>
<td>234289</td>
<td>2356</td>
<td>1590</td>
<td>107608</td>
<td>1947</td>
</tr>
<tr>
<td>61122</td>
<td>88.5</td>
<td>259426</td>
<td>2325</td>
<td>1456</td>
<td>108632</td>
<td>1948</td>
</tr>
<tr>
<td>60171</td>
<td>88.2</td>
<td>258054</td>
<td>3682</td>
<td>1616</td>
<td>109773</td>
<td>1949</td>
</tr>
<tr>
<td>61187</td>
<td>89.5</td>
<td>284599</td>
<td>3351</td>
<td>1650</td>
<td>110929</td>
<td>1950</td>
</tr>
<tr>
<td>63221</td>
<td>96.2</td>
<td>32975</td>
<td>2099</td>
<td>3099</td>
<td>112075</td>
<td>1951</td>
</tr>
<tr>
<td>63639</td>
<td>98.1</td>
<td>346999</td>
<td>1932</td>
<td>3594</td>
<td>113270</td>
<td>1952</td>
</tr>
<tr>
<td>64989</td>
<td>99.0</td>
<td>365385</td>
<td>1870</td>
<td>3547</td>
<td>115094</td>
<td>1953</td>
</tr>
<tr>
<td>63761</td>
<td>100.0</td>
<td>363112</td>
<td>3578</td>
<td>3350</td>
<td>116219</td>
<td>1954</td>
</tr>
<tr>
<td>66019</td>
<td>101.2</td>
<td>397469</td>
<td>2904</td>
<td>3048</td>
<td>117388</td>
<td>1955</td>
</tr>
<tr>
<td>67857</td>
<td>104.6</td>
<td>419180</td>
<td>2822</td>
<td>2857</td>
<td>118734</td>
<td>1956</td>
</tr>
<tr>
<td>68169</td>
<td>108.4</td>
<td>442769</td>
<td>2936</td>
<td>2798</td>
<td>120445</td>
<td>1957</td>
</tr>
<tr>
<td>66513</td>
<td>110.8</td>
<td>444546</td>
<td>4681</td>
<td>2637</td>
<td>121950</td>
<td>1958</td>
</tr>
<tr>
<td>68655</td>
<td>112.6</td>
<td>482704</td>
<td>3813</td>
<td>2552</td>
<td>123366</td>
<td>1959</td>
</tr>
<tr>
<td>69564</td>
<td>114.2</td>
<td>502601</td>
<td>3931</td>
<td>2514</td>
<td>125368</td>
<td>1960</td>
</tr>
<tr>
<td>69331</td>
<td>115.7</td>
<td>518173</td>
<td>4806</td>
<td>2572</td>
<td>127852</td>
<td>1961</td>
</tr>
<tr>
<td>70551</td>
<td>116.9</td>
<td>554894</td>
<td>4007</td>
<td>2827</td>
<td>130081</td>
<td>1962</td>
</tr>
</tbody>
</table>

With pandas integration in the estimation classes, the metadata will be attached to model results:

**Extra Information**

If you want to know more about the dataset itself, you can access the following, again using the Longley dataset as an example

```bash
>>> dir(sm.datasets.longley)[:6]
['COPYRIGHT', 'DESCRLONG', 'DESCRSHORT', 'NOTE', 'SOURCE', 'TITLE']
```

### 3.17.5 Additional information

- The idea for a datasets package was originally proposed by David Cournapeau and can be found [here](#) with updates by Skipper Seabold.
- To add datasets, see the [notes on adding a dataset](#).
3.18 Sandbox

This sandbox contains code that is for various reasons not ready to be included in statsmodels proper. It contains modules from the old stats.models code that have not been tested, verified and updated to the new statsmodels structure: cox survival model, mixed effects model with repeated measures, generalized additive model and the formula framework. The sandbox also contains code that is currently being worked on until it fits the pattern of statsmodels or is sufficiently tested.

All sandbox modules have to be explicitly imported to indicate that they are not yet part of the core of statsmodels. The quality and testing of the sandbox code varies widely. This is sandbox code

3.18.1 Examples

There are some examples in the sandbox.examples folder. Additional examples are directly included in the modules and in subfolders of the sandbox.

3.18.2 Module Reference

Time Series analysis tsa

In this part we develop models and functions that will be useful for time series analysis. Most of the models and function have been moved to statsmodels.tsa. Currently, GARCH models remain in development stage in sandbox.tsa.

Moving Window Statistics

Most moving window statistics, like rolling mean, moments (up to 4th order), min, max, mean, and variance, are covered by the functions for Moving (rolling) statistics/moments in Pandas.

| tsa.movstat | using scipy signal and numpy correlate to calculate some time series |

| statsmodels.sandbox.tsa.movstat | using scipy signal and numpy correlate to calculate some time series statistics |

original developer notes

see also scikits.timeseries (movstat is partially inspired by it) added 2009-08-29 timeseries moving stats are in c, autocorrelation similar to here I thought I saw moving stats somewhere in python, maybe not)

TODO

Moving statistics - filters don’t handle boundary conditions nicely (correctly ?) e.g. minimum order filter uses 0 for out of bounds value -> append and prepend with last resp. first value - enhance for nd arrays, with axis = 0

Note: Equivalence for 1D signals >>> np.all(signal.correlate(x,[1,1,1],'valid')==np.correlate(x,[1,1,1])) True >>> np.all(ndimage.filters.correlate(x,[1,1,1], origin = -1)[:-3+1]==np.correlate(x,[1,1,1])) True

# multidimensional, but, it looks like it uses common filter across time series, no VAR ndimage.filters.correlate(np.vstack([x,x]),np.array([[1,1,1],[0,0,0]]), origin = 1) ndimage.filters.correlate(x,[1,1,1],origin = 1)) ndimage.filters.correlate(np.vstack([x,x]),np.array([[0.5,0.5,0.5],[0.5,0.5,0.5]]), origin = 1)

>>> np.all(ndimage.filters.correlate(np.vstack([x,x]),np.array([[1,1,1],[0,0,0]]), origin = 1)[0]==ndimage.filters.correlate(x,[1,1,1],origin = 1))
True

>>> np.all(ndimage.filters.correlate(np.vstack([x,x]),np.array([[0.5,0.5,0.5],[0.5,0.5,0.5]]), origin = 1))

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert_array_almost_equal(x, y[, decimal, ...])</td>
<td>Raise an assertion if two objects are not equal up to desired precision.</td>
</tr>
<tr>
<td>assert_array_equal(x, y[, err_msg, verbose])</td>
<td>Raise an assertion if two array_like objects are not equal.</td>
</tr>
<tr>
<td>checkMovorder()</td>
<td>graphical test for movorder</td>
</tr>
<tr>
<td>expandarr(x, k)</td>
<td></td>
</tr>
<tr>
<td>movmean(x[, windowsize, lag])</td>
<td>moving window mean</td>
</tr>
<tr>
<td>movmoment(x, k[, windowsize, lag])</td>
<td>non-central moment</td>
</tr>
<tr>
<td>movorder(x[, order, windsize, lag])</td>
<td>moving order statistics</td>
</tr>
<tr>
<td>movvar(x[, windowsize, lag])</td>
<td>moving window variance</td>
</tr>
</tbody>
</table>

Regression and ANOVA

The following two ANOVA functions are fully tested against the NIST test data for balanced one-way ANOVA. anova_oneway follows the same pattern as the oneway anova function in scipy.stats but with higher precision for badly scaled problems. anova_ols produces the same results as the one way anova however using the OLS model class. It also verifies against the NIST tests, with some problems in the worst scaled cases. It shows how to do simple ANOVA using statsmodels in three lines and is also best taken as a recipe.

```python
anova_oneway(y, x[, seq])
anova_ols(y, x)
```

The following are helper functions for working with dummy variables and generating ANOVA results with OLS. They are best considered as recipes since they were written with a specific use in mind. These function will eventually be rewritten or reorganized.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>try_ols_anova.data2dummy(x[, returnall])</td>
<td>convert array of categories to dummy variables</td>
</tr>
<tr>
<td>try_ols_anova.data2groupcont(x1, x2)</td>
<td>create dummy continuous variable</td>
</tr>
<tr>
<td>try_ols_anova.data2prodummy(x)</td>
<td>creates product dummy variables from 2 columns of 2d array</td>
</tr>
<tr>
<td>try_ols_anova.dropname(ss, li)</td>
<td>drop names from a list of strings,</td>
</tr>
<tr>
<td>try_ols_anova.form2design(ss, data)</td>
<td>convert string formula to data dictionary</td>
</tr>
</tbody>
</table>

statsmodels.sandbox.regression.try_ols_anova.data2dummy

convert array of categories to dummy variables by default drops dummy variable for last category uses ravel, 1d only
create dummy continuous variable

**Parameters**
- **x1**: 1d array
  - label or group array
- **x2**: 1d array (float)
  - continuous variable

**Notes**
useful for group specific slope coefficients in regression

creates product dummy variables from 2 columns of 2d array

drops last dummy variable, but not from each category singular with simple dummy variable but not with constant
quickly written, no safeguards

drop names from a list of strings, names to drop are in space delimited list does not change original list

convert string formula to data dictionary

**ss** [string]
- I : add constant
- varname : for simple varnames data is used as is
- F:varname : create dummy variables for factor varname
- P:varname1*varname2 : create product dummy variables for varnames
- G:varname1*varname2 : create product between factor and continuous variable

**data** [dict or structured array] data set, access of variables by name as in dictionaries

**Returns**
- **vars**: dictionary
  - dictionary of variables with converted dummy variables
- **names**: list
list of names, product (P:) and grouped continuous variables (G:) have name by joining individual names sorted according to input

**Notes**

with sorted dict, separate name list wouldn’t be necessary

**Examples**

```python
>>> xx, n = form2design('I a F:b P:c*d G:c*f', testdata)
>>> xx.keys()
['a', 'b', 'const', 'cf', 'cd']
>>> n
['const', 'a', 'b', 'cd', 'cf']
```

The following are helper functions for group statistics where groups are defined by a label array. The qualifying comments for the previous group apply also to this group of functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat2dummy(y[, nonseq])</code></td>
<td>Converts labels based on multiple variables or string labels to unique index labels 0,1,2,...,nk-1 where nk is the number of distinct labels</td>
</tr>
<tr>
<td><code>convertlabels(ys[, indices])</code></td>
<td>Converts labels to unique index labels 0,1,2,...,nk-1 where nk is the number of distinct labels</td>
</tr>
<tr>
<td><code>groupsstats_1d(y, x, labelsunique)</code></td>
<td>Uses ndimage to get fast mean and variance</td>
</tr>
<tr>
<td><code>groupsstats_dummy(y, x[, nonseq])</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
<tr>
<td><code>groupstatsbin(factors, values)</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
<tr>
<td><code>labelmeanfilter(y, x)</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
<tr>
<td><code>labelmeanfilter_nd(y, x)</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
</tbody>
</table>

**Statsmodels.sandbox.regression.try_catdata.cat2dummy**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat2dummy(y[, nonseq])</code></td>
<td>Converts labels based on multiple variables or string labels to unique index labels 0,1,2,...,nk-1 where nk is the number of distinct labels</td>
</tr>
</tbody>
</table>

**Statsmodels.sandbox.regression.try_catdata.convertlabels**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convertlabels(ys[, indices])</code></td>
<td>Converts labels to unique index labels 0,1,2,...,nk-1 where nk is the number of distinct labels</td>
</tr>
</tbody>
</table>

**Statsmodels.sandbox.regression.try_catdata.groupsstats_1d**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>groupsstats_1d(y, x[, labelsunique])</code></td>
<td>Uses ndimage to get fast mean and variance</td>
</tr>
<tr>
<td><code>groupsstats_dummy(y, x[, nonseq])</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
</tbody>
</table>

**Statsmodels.sandbox.regression.try_catdata.groupstatsbin**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>groupstatsbin(factors, values)</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
<tr>
<td><code>labelmeanfilter(y, x)</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
<tr>
<td><code>labelmeanfilter_nd(y, x)</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
<tr>
<td><code>labelmeanfilter_str(ys, x)</code></td>
<td>Uses np.bincount, assumes factors/labels are integers</td>
</tr>
</tbody>
</table>
statsmodels.sandbox.regression.try_catdata.groupstatsbin

statsmodels.sandbox.regression.try_catdata.groupstatsbin(factors, values)
uses np.bincount, assumes factors/labels are integers

statsmodels.sandbox.regression.try_catdata.labelmeanfilter

statsmodels.sandbox.regression.try_catdata.labelmeanfilter(y, x)

statsmodels.sandbox.regression.try_catdata.labelmeanfilter_nd

statsmodels.sandbox.regression.try_catdata.labelmeanfilter_nd(y, x)

statsmodels.sandbox.regression.try_catdata.labelmeanfilter_str

statsmodels.sandbox.regression.try_catdata.labelmeanfilter_str(ys, x)

Additional to these functions, sandbox regression still contains several examples, that are illustrative of the use of the regression models of statsmodels.

Systems of Regression Equations and Simultaneous Equations

The following are for fitting systems of equations models. Though the returned parameters have been verified as accurate, this code is still very experimental, and the usage of the models will very likely change significantly before they are added to the main codebase.

\[
\begin{align*}
\text{SUR}(\text{sys[, sigma, dfk]}) & \quad \text{Seemingly Unrelated Regression} \\
\text{Sem2SLS}(\text{sys[, indep_endog, instruments]}) & \quad \text{Two-Stage Least Squares for Simultaneous equations}
\end{align*}
\]

statsmodels.sandbox.sysreg.SUR

class statsmodels.sandbox.sysreg.SUR(sys, sigma=None, dfk=None)
Seemingly Unrelated Regression

Parameters sys : list
[ endog1, exog1, endog2, exog2,... ] It will be of length 2 x M, where M is the number of equations endog = exog.

sigma : array-like
M x M array where sigma[i,j] is the covariance between equation i and j

dfk : None, ‘dfk1’, or ‘dfk2’
Default is None. Correction for the degrees of freedom should be specified for small samples. See the notes for more information.
Notes

All individual equations are assumed to be well-behaved, homoskedastic iid errors. This is basically an extension of GLS, using sparse matrices.

\[ \Sigma = \begin{bmatrix}
\sigma_{11} & \sigma_{12} & \cdots & \sigma_{1M} \\
\sigma_{21} & \sigma_{22} & \cdots & \sigma_{2M} \\
\vdots & \vdots & \ddots & \vdots \\
\sigma_{M1} & \sigma_{M2} & \cdots & \sigma_{MM}
\end{bmatrix} \]

References


Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholsigmainv</td>
<td>array</td>
<td>The transpose of the Cholesky decomposition of pinv_wexog</td>
</tr>
<tr>
<td>df_model</td>
<td>array</td>
<td>Model degrees of freedom of each equation. ( p_(m) - 1 ) where ( p ) is the number of regressors for each equation ( m ) and one is subtracted for the constant.</td>
</tr>
<tr>
<td>df_resid</td>
<td>array</td>
<td>Residual degrees of freedom of each equation. Number of observations less the number of parameters.</td>
</tr>
<tr>
<td>endog</td>
<td>array</td>
<td>The LHS variables for each equation in the system. It is a ( M \times \text{nobs} ) array where ( M ) is the number of equations.</td>
</tr>
<tr>
<td>exog</td>
<td>array</td>
<td>The RHS variable for each equation in the system. It is a ( \text{nobs} \times \sum(p_(m)) ) array. Which is just each RHS array stacked next to each other in columns.</td>
</tr>
<tr>
<td>history</td>
<td>dict</td>
<td>Contains the history of fitting the model. Probably not of interest if the model is fit with ( \text{igls} = \text{False} ).</td>
</tr>
<tr>
<td>iterations</td>
<td>int</td>
<td>The number of iterations until convergence if the model is fit iteratively.</td>
</tr>
<tr>
<td>nobss</td>
<td>float</td>
<td>The number of observations of the equations.</td>
</tr>
<tr>
<td>normalized_cov_params</td>
<td>array</td>
<td>( \sum(p_(m)) \times \sum(p_(m)) ) array ( [X^T (\Sigma^{-1} \otimes I) X]^{-1} )</td>
</tr>
<tr>
<td>pinv_wexog</td>
<td>array</td>
<td>The pseudo-inverse of the ( \text{wexog} )</td>
</tr>
<tr>
<td>sigma</td>
<td>array</td>
<td>( M \times M ) covariance matrix of the cross-equation disturbances. See notes.</td>
</tr>
<tr>
<td>sp_exog</td>
<td>CSR</td>
<td>Contains a block diagonal sparse matrix of the design so that ( \text{exog1} \ldots \text{exogM} ) are on the diagonal.</td>
</tr>
<tr>
<td>wendog</td>
<td>array</td>
<td>( M \times \text{nobs} \times 1 ) array of the endogenous variables whitened by ( \text{cholsigmainv} ) and stacked into a single column.</td>
</tr>
<tr>
<td>wexog</td>
<td>array</td>
<td>( M\times\text{nobs} \times \sum(p_(m)) ) array of the whitened exogenous variables.</td>
</tr>
</tbody>
</table>

Methods

- \( \text{fit}([\text{igls}, \text{tol}, \text{maxiter}]) \)  \( \text{igls} : \text{bool} \)
- \( \text{initialize}() \)
- \( \text{predict}(\text{design}) \)
- \( \text{whiten}(\text{X}) \)  \( \text{SUR whiten method.} \)
**statsmodels.sandbox.sysreg.SUR.fit**

`SUR.fit(igls=False, tol=1e-05, maxiter=100)`

- **igls** [bool] Iterate until estimates converge if sigma is None instead of two-step GLS, which is the default is sigma is None.
- **tol** : float
- **maxiter** : int

**Notes**

This ia naive implementation that does not exploit the block diagonal structure. It should work for ill-conditioned `sigma` but this is untested.

**statsmodels.sandbox.sysreg.SUR.initialize**

`SUR.initialize()`

**statsmodels.sandbox.sysreg.SUR.predict**

`SUR.predict(design)`

**statsmodels.sandbox.sysreg.SUR.whiten**

`SUR.whiten(X)`

SUR whiten method.

- **Parameters** `X` : list of arrays
  
  Data to be whitened.

- **Returns** If `X` is the exogenous RHS of the system. :

  "np.dot(np.kron(cholsigmainv,np.eye(M)),np.diag(X))"

  If `X` is the endogenous LHS of the system. :

**statsmodels.sandbox.sysreg.Sem2SLS**

**class** `statsmodels.sandbox.sysreg.Sem2SLS(sys, indep_endog=None, instruments=None)`

Two-Stage Least Squares for Simultaneous equations

- **Parameters** `sys` : list
  
  [endog1, exog1, endog2, exog2,...] It will be of length 2 x M, where M is the number of equations endog = exog.

  **indep_endog** : dict

  A dictionary mapping the equation to the column numbers of the the independent endogenous regressors in each equation. It is assumed that the system is inputed as broken up into LHS and RHS. For now, the values of the dict have to be sequences. Note that the keys for the equations should be zero-indexed.

  **instruments** : array

  Array of the exogenous independent variables.
Notes

This is unfinished, and the design should be refactored. Estimation is done by brute force and there is no exploitation of the structure of the system.

Methods

\[
\begin{align*}
\text{fit()} & \quad \text{Runs the first stage of the 2SLS.} \\
\text{whiten}(Y) & \quad \text{Runs the first stage of the 2SLS.}
\end{align*}
\]

\textit{statsmodels.sandbox.sysreg.Sem2SLS.fit}
Sem2SLS.\texttt{fit}()

\textit{statsmodels.sandbox.sysreg.Sem2SLS.whiten}
Sem2SLS.\texttt{whiten}(Y)

- Runs the first stage of the 2SLS.
- Returns the RHS variables that include the instruments.

Miscellaneous

Tools for Time Series Analysis

nothing left in here

Tools: Principal Component Analysis

\[
\text{pca}(\text{data[, keepdim, normalize, demean]}) \quad \text{principal components with eigenvector decomposition}
\]

\[
\text{pcasvd}(\text{data[, keepdim, demean]}) \quad \text{principal components with svd}
\]

\textit{statsmodels.sandbox.tools.tools_pca.pca}
\textit{statsmodels.sandbox.tools.tools_pca.pca (data, keepdim=0, normalize=0, demean=True)}

principal components with eigenvector decomposition similar to princomp in matlab

**Parameters**

- **data**: ndarray, 2d
  - data with observations by rows and variables in columns

- **keepdim**: integer
  - number of eigenvectors to keep if keepdim is zero, then all eigenvectors are included

- **normalize**: boolean
  - if true, then eigenvectors are normalized by sqrt of eigenvalues

- **demean**: boolean
  - if true, then the column mean is subtracted from the data

**Returns**

- **xreduced**: ndarray, 2d, (nobs, nvars)
  - projection of the data x on the kept eigenvectors
**factors**: ndarray, 2d, (nobs, nfactors)
   factor matrix, given by np.dot(x, evecs)

**evals**: ndarray, 2d, (nobs, nfactors)
eigenvalues

**evecs**: ndarray, 2d, (nobs, nfactors)
eigenvectors, normalized if normalize is true

See also:

**pcasvd** principal component analysis using svd

```python
statsmodels.sandbox.tools.tools_pca.pcasvd
```

`statsmodels.sandbox.tools.tools_pca.pcasvd(data, keepdim=0, demean=True)`

principal components with svd

Parameters

**data**: ndarray, 2d
data with observations by rows and variables in columns

**keepdim**: integer
   number of eigenvectors to keep if keepdim is zero, then all eigenvectors are included

**demean**: boolean
   if true, then the column mean is subtracted from the data

Returns

**xreduced**: ndarray, 2d, (nobs, nvars)
   projection of the data x on the kept eigenvectors

**factors**: ndarray, 2d, (nobs, nfactors)
factor matrix, given by np.dot(x, evecs)

**evals**: ndarray, 2d, (nobs, nfactors)
eigenvalues

**evecs**: ndarray, 2d, (nobs, nfactors)
eigenvectors, normalized if normalize is true

See Also:

---

**pca** principal component analysis using eigenvector decomposition

Notes

This doesn’t have yet the normalize option of pca.

**Descriptive Statistics Printing**

```
  descstats.sign_test(samp[, mu0])  Signs test.
  descstats.descstats(data[, cols, axis])  Prints descriptive statistics for one or multiple variables.
```

3.18. Sandbox
statsmodels.sandbox.descstats.sign_test
statsmodels.sandbox.descstats.sign_test(samp, mu0=0)

Signs test.

Parameters samp : array-like

1d array. The sample for which you want to perform the signs test.

mu0 : float

See Notes for the definition of the sign test. mu0 is 0 by default, but it is common to
set it to the median.

Returns M, p-value :

See also:
scepy.stats.wilcoxon

Notes

The signs test returns
M = (N(+) - N(-))/2

where N(+) is the number of values above mu0, N(-) is the number of values below. Values equal to mu0 are
discarded.

The p-value for M is calculated using the binomial distribution and can be interpreted the same as for a t-test.
The test-statistic is distributed Binom(min(N(+), N(-)), n_trials, .5) where n_trials equals N(+) + N(-).

statsmodels.sandbox.descstats.descstats
statsmodels.sandbox.descstats.descstats(data, cols=None, axis=0)

Prints descriptive statistics for one or multiple variables.

Parameters data: numpy array :

x is the data

v: list, optional :

A list of the column number or field names (for a recarray) of variables. Default is
all columns.

axis: 1 or 0 :

axis order of data. Default is 0 for column-ordered data.

Examples

>>> descstats(data.exog,v=['x_1','x_2','x_3'])

Original stats.models

None of these are fully working. The formula framework is used by cox and mixed.

Mixed Effects Model with Repeated Measures using an EM Algorithm

statsmodels.sandbox.mixed
Cox Proportional Hazards Model
statsmodels.sandbox.cox

Generalized Additive Models
statsmodels.sandbox.gam

Formula
statsmodels.sandbox.formula
CHAPTER
FOUR

INDICES AND TABLES

• genindex
• modindex
• search


S

statsmodels.graphics, 948
statsmodels.sandbox, 1025
statsmodels.sandbox.tsa.movstat, 1025
S
statsmodels.graphics, 948
statsmodels.sandbox, 1025
statsmodels.sandbox.tsa.movstat, 1025