arch Documentation

Release 4.7.0

Kevin Sheppard

Dec 13, 2018

Contents

The ARCH toolbox currently contains routines for

- Univariate volatility models
- Bootstrapping
- Multiple comparison procedures
- Unit root tests

Future plans are to continue to expand this toolbox to include additional routines relevant for the analysis of financial data.

CHAPTER 1

Contents

1.1 Univariate Volatility Models

1.1.1 Introduction to ARCH Models

ARCH models are a popular class of volatility models that use observed values of returns or residuals as volatility shocks. A basic GARCH model is specified as

$$r_t = \mu + \epsilon_t \tag{1.1}$$
$$\epsilon_t = \sigma_t e_t \tag{1.2}$$
$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2 \tag{1.3}$$

A complete ARCH model is divided into three components:

However, the simplest method to construct this model is to use the constructor function *arch_model()*

```
import datetime as dt

import pandas_datareader.data as web

from arch import arch_model

start = dt.datetime(2000, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
returns = 100 * sp500['Adj Close'].pct_change().dropna()
am = arch_model(returns)
```

Alternatively, the same model can be manually assembled from the building blocks of an ARCH model

```
from arch import ConstantMean, GARCH, Normal

am = ConstantMean(returns)
```

(continues on next page)

(continued from previous page)

```
am.volatility = GARCH(1, 0, 1)
am.distribution = Normal()
```

In either case, model parameters are estimated using

```
res = am.fit()
```

with the following output

```
Iteration:      1,   Func. Count:      6,   Neg. LLF: 5159.58323938
Iteration:      2,   Func. Count:     16,   Neg. LLF: 5156.09760149
Iteration:      3,   Func. Count:     24,   Neg. LLF: 5152.29989336
Iteration:      4,   Func. Count:     31,   Neg. LLF: 5146.47531817
Iteration:      5,   Func. Count:     38,   Neg. LLF: 5143.86337547
Iteration:      6,   Func. Count:     45,   Neg. LLF: 5143.02096168
Iteration:      7,   Func. Count:     52,   Neg. LLF: 5142.24105141
Iteration:      8,   Func. Count:     60,   Neg. LLF: 5142.07138907
Iteration:      9,   Func. Count:     67,   Neg. LLF: 5141.416653
Iteration:     10,   Func. Count:     73,   Neg. LLF: 5141.39212288
Iteration:     11,   Func. Count:     79,   Neg. LLF: 5141.39023885
Iteration:     12,   Func. Count:     85,   Neg. LLF: 5141.39023359
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 5141.39023359
            Iterations: 12
            Function evaluations: 85
            Gradient evaluations: 12
```

```
print(res.summary())
```

yields

```
                     Constant Mean - GARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                      -0.001
Mean Model:             Constant Mean   Adj. R-squared:                 -0.001
Vol Model:                      GARCH   Log-Likelihood:               -5141.39
Distribution:                  Normal   AIC:                           10290.8
Method:            Maximum Likelihood   BIC:                           10315.4
                                        No. Observations:                 3520
Date:                Fri, Dec 02 2016   Df Residuals:                     3516
Time:                        22:22:28   Df Model:                            4
                                Mean Model
==============================================================================
                 coef    std err          t      P>|t|        95.0% Conf. Int.
------------------------------------------------------------------------------
mu             0.0531  1.487e-02      3.569  3.581e-04   [2.392e-02,8.220e-02]
                              Volatility Model
==============================================================================
                 coef    std err          t      P>|t|        95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.0156  4.932e-03      3.155  1.606e-03   [5.892e-03,2.523e-02]
alpha[1]       0.0879  1.140e-02      7.710  1.260e-14     [6.554e-02,  0.110]
beta[1]        0.9014  1.183e-02     76.163      0.000       [  0.878,  0.925]
==============================================================================

Covariance estimator: robust
```

Core Model Constructor

While models can be carefully specified using the individual components, most common specifications can be specified using a simple model constructor.

arch.**arch_model**(*y*, *x=None*, *mean='Constant'*, *lags=0*, *vol='Garch'*, *p=1*, *o=0*, *q=1*, *power=2.0*, *dist='Normal'*, *hold_back=None*)

Convenience function to simplify initialization of ARCH models

Parameters

- **y** (*{ndarray, Series, None}*) – The dependent variable
- **x** (*{np.array, DataFrame}, optional*) – Exogenous regressors. Ignored if model does not permit exogenous regressors.
- **mean** (*str, optional*) – Name of the mean model. Currently supported options are: 'Constant', 'Zero', 'ARX' and 'HARX'
- **lags** (*int or list (int), optional*) – Either a scalar integer value indicating lag length or a list of integers specifying lag locations.
- **vol** (*str, optional*) – Name of the volatility model. Currently supported options are: 'GARCH' (default), 'ARCH', 'EGARCH', 'FIARCH' and 'HARCH'
- **p** (*int, optional*) – Lag order of the symmetric innovation
- **o** (*int, optional*) – Lag order of the asymmetric innovation
- **q** (*int, optional*) – Lag order of lagged volatility or equivalent
- **power** (*float, optional*) – Power to use with GARCH and related models
- **dist** (*int, optional*) – Name of the error distribution. Currently supported options are:
  - Normal: 'normal', 'gaussian' (default)
  - Students's t: 't', 'studentst'
  - Skewed Student's t: 'skewstudent', 'skewt'
  - Generalized Error Distribution: 'ged', 'generalized error"
- **hold_back** (*int*) – Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.

Returns **model** – Configured ARCH model

Return type *ARCHModel*

Examples

```
>>> import datetime as dt
>>> import pandas_datareader.data as web
>>> djia = web.get_data_fred('DJIA')
>>> returns = 100 * djia['DJIA'].pct_change().dropna()
```

A basic GARCH(1,1) with a constant mean can be constructed using only the return data

```
>>> from arch.univariate import arch_model
>>> am = arch_model(returns)
```

Alternative mean and volatility processes can be directly specified

```
>>> am = arch_model(returns, mean='AR', lags=2, vol='harch', p=[1, 5, 22])
```

This example demonstrates the construction of a zero mean process with a TARCH volatility process and Student t error distribution

```
>>> am = arch_model(returns, mean='zero', p=1, o=1, q=1,
...                 power=1.0, dist='StudentsT')
```

Notes

Input that are not relevant for a particular specification, such as *lags* when *mean='zero'*, are silently ignored.

Model Results

All model return the same object, a results class (`ARCHModelResult`)

class arch.univariate.base.**ARCHModelResult**(*params*, *param_cov*, *r2*, *resid*, *volatility*, *cov_type*, *dep_var*, *names*, *loglikelihood*, *is_pandas*, *optim_output*, *fit_start*, *fit_stop*, *model*)

Results from estimation of an ARCHModel model

Parameters

- **params** (*ndarray*) – Estimated parameters
- **param_cov** (*{ndarray, None}*) – Estimated variance-covariance matrix of params. If none, calls method to compute variance from model when parameter covariance is first used from result
- **r2** (*float*) – Model R-squared
- **resid** (*ndarray*) – Residuals from model. Residuals have same shape as original data and contain nan-values in locations not used in estimation
- **volatility** (*ndarray*) – Conditional volatility from model
- **cov_type** (*str*) – String describing the covariance estimator used
- **dep_var** (*Series*) – Dependent variable
- **names** (*list (str)*) – Model parameter names
- **loglikelihood** (*float*) – Loglikelihood at estimated parameters
- **is_pandas** (*bool*) – Whether the original input was pandas
- **fit_start** (*int*) – Integer index of the first observation used to fit the model
- **fit_stop** (*int*) – Integer index of the last observation used to fit the model using slice notation *fit_start:fit_stop*
- **model** (`ARCHModel`) – The model object used to estimate the parameters

summary()

Produce a summary of the results

plot()

Produce a plot of the volatility and standardized residuals

conf_int()
Confidence intervals

loglikelihood
float – Value of the log-likelihood

params
Series – Estimated parameters

param_cov
DataFrame – Estimated variance-covariance of the parameters

resid
{ndarray, Series} – nobs element array containing model residuals

model
ARCHModel – Model instance used to produce the fit

conf_int(*alpha=0.05*)

Parameters **alpha** (*float, optional*) – Size (prob.) to use when constructing the confidence interval.

Returns **ci** – Array where the ith row contains the confidence interval for the ith parameter

Return type ndarray

forecast(*params=None*, *horizon=1*, *start=None*, *align='origin'*, *method='analytic'*, *simulations=1000*, *rng=None*)
Construct forecasts from estimated model

Parameters

- **params** (*ndarray, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'.
- **align** (*str, optional*) – Either 'origin' or 'target'. When set of 'origin', the t-th row of forecasts contains the forecasts for t+1, t+2, ..., t+h. When set to 'target', the t-th row contains the 1-step ahead forecast from time t-1, the 2 step from time t-2, ..., and the h-step from time t-h. 'target' simplified computing forecast errors since the realization and h-step forecast are aligned.
- **method** (*{'analytic', 'simulation', 'bootstrap'}, optional*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.
- **simulations** (*int, optional*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** (*callable, optional*) – Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

Returns **forecasts** – t by h data frame containing the forecasts. The alignment of the forecasts is controlled by *align*.

Return type *ARCHModelForecast*

Notes

The most basic 1-step ahead forecast will return a vector with the same length as the original data, where the t-th value will be the time-t forecast for time t + 1. When the horizon is > 1, and when using the default value for *align*, the forecast value in position [t, h] is the time-t, h+1 step ahead forecast.

If model contains exogenous variables (*model.x is not None*), then only 1-step ahead forecasts are available. Using horizon > 1 will produce a warning and all columns, except the first, will be nan-filled.

If *align* is 'origin', forecast[t,h] contains the forecast made using y[:t] (that is, up to but not including t) for horizon h + 1. For example, y[100,2] contains the 3-step ahead forecast using the first 100 data points, which will correspond to the realization y[100 + 2]. If *align* is 'target', then the same forecast is in location [102, 2], so that it is aligned with the observation to use when evaluating, but still in the same column.

hedgehog_plot(*params=None*, *horizon=10*, *step=10*, *start=None*, *type='volatility'*, *method='analytic'*, *simulations=1000*)

Plot forecasts from estimated model

Parameters

- **params** (*{ndarray, Series}*) – Alternative parameters to use. If not provided, the parameters computed by fitting the model are used. Must be 1-d and identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **step** (*int, optional*) – Non-negative number of forecasts to skip between spines
- **start** (*int, datetime or str, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'. If not provided, the start is set to the earliest forecastable date.
- **type** (*{'volatility', 'mean'}*) – Quantity to plot, the forecast volatility or the forecast mean
- **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.

Returns **fig** – Handle to the figure

Return type figure

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],
↪vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot(type='mean')
```

plot(*annualize=None*, *scale=None*)

Plot standardized residuals and conditional volatility

Parameters

- **annualize** (*str, optional*) – String containing frequency of data that indicates plot should contain annualized volatility. Supported values are 'D' (daily), 'W' (weekly) and 'M' (monthly), which scale variance by 252, 52, and 12, respectively.
- **scale** (*float, optional*) – Value to use when scaling returns to annualize. If scale is provides, annualize is ignored and the value in scale is used.

Returns **fig** – Handle to the figure

Return type figure

Examples

```
>>> from arch import arch_model
>>> am = arch_model(None)
>>> sim_data = am.simulate([0.0, 0.01, 0.07, 0.92], 2520)
>>> am = arch_model(sim_data['data'])
>>> res = am.fit(update_freq=0, disp='off')
>>> fig = res.plot()
```

Produce a plot with annualized volatility

```
>>> fig = res.plot(annualize='D')
```

Override the usual scale of 252 to use 360 for an asset that trades most days of the year

```
>>> fig = res.plot(scale=360)
```

summary()

Constructs a summary of the results from a fit model.

Returns **summary** – Object that contains tables and facilitated export to text, html or latex

Return type Summary instance

When using the `fix` method, a (*ARCHModelFixedResult*) is produced that lacks some properties of a (*ARCHModelResult*) that are not relevant when parameters are not estimated.

class arch.univariate.base.**ARCHModelFixedResult**(*params*, *resid*, *volatility*, *dep_var*, *names*, *loglikelihood*, *is_pandas*, *model*)

Results for fixed parameters for an ARCHModel model

Parameters

- **params** (*ndarray*) – Estimated parameters
- **resid** (*ndarray*) – Residuals from model. Residuals have same shape as original data and contain nan-values in locations not used in estimation
- **volatility** (*ndarray*) – Conditional volatility from model
- **dep_var** (*Series*) – Dependent variable
- **names** (*list (str)*) – Model parameter names
- **loglikelihood** (*float*) – Loglikelihood at specified parameters
- **is_pandas** (*bool*) – Whether the original input was pandas
- **model** (ARCHModel) – The model object used to estimate the parameters

summary()
Produce a summary of the results

plot()
Produce a plot of the volatility and standardized residuals

forecast()
Construct forecasts from a model

loglikelihood
float – Value of the log-likelihood

params
Series – Estimated parameters

resid
{ndarray, Series} – nobs element array containing model residuals

model
ARCHModel – Model instance used to produce the fit

forecast(*params=None*, *horizon=1*, *start=None*, *align='origin'*, *method='analytic'*, *simulations=1000*, *rng=None*)
Construct forecasts from estimated model

Parameters

- **params** (*ndarray, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'.
- **align** (*str, optional*) – Either 'origin' or 'target'. When set of 'origin', the t-th row of forecasts contains the forecasts for t+1, t+2, ..., t+h. When set to 'target', the t-th row contains the 1-step ahead forecast from time t-1, the 2 step from time t-2, ..., and the h-step from time t-h. 'target' simplified computing forecast errors since the realization and h-step forecast are aligned.
- **method** (*{'analytic', 'simulation', 'bootstrap'}, optional*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In

particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.

- **simulations** (*int, optional*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** (*callable, optional*) – Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

Returns **forecasts** – t by h data frame containing the forecasts. The alignment of the forecasts is controlled by *align*.

Return type *ARCHModelForecast*

Notes

The most basic 1-step ahead forecast will return a vector with the same length as the original data, where the t-th value will be the time-t forecast for time t + 1. When the horizon is > 1, and when using the default value for *align*, the forecast value in position [t, h] is the time-t, h+1 step ahead forecast.

If model contains exogenous variables (*model.x is not None*), then only 1-step ahead forecasts are available. Using horizon > 1 will produce a warning and all columns, except the first, will be nan-filled.

If *align* is 'origin', forecast[t,h] contains the forecast made using y[:t] (that is, up to but not including t) for horizon h + 1. For example, y[100,2] contains the 3-step ahead forecast using the first 100 data points, which will correspond to the realization y[100 + 2]. If *align* is 'target', then the same forecast is in location [102, 2], so that it is aligned with the observation to use when evaluating, but still in the same column.

hedgehog_plot (*params=None, horizon=10, step=10, start=None, type='volatility', method='analytic', simulations=1000*)

Plot forecasts from estimated model

Parameters

- **params** (*{ndarray, Series}*) – Alternative parameters to use. If not provided, the parameters computed by fitting the model are used. Must be 1-d and identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **step** (*int, optional*) – Non-negative number of forecasts to skip between spines
- **start** (*int, datetime or str, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'. If not provided, the start is set to the earliest forecastable date.
- **type** (*{'volatility', 'mean'}*) – Quantity to plot, the forecast volatility or the forecast mean
- **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.

Returns **fig** – Handle to the figure

Return type figure

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22], ␣
↪vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot(type='mean')
```

plot(*annualize=None*, *scale=None*)

Plot standardized residuals and conditional volatility

Parameters

- **annualize** (*str, optional*) – String containing frequency of data that indicates plot should contain annualized volatility. Supported values are 'D' (daily), 'W' (weekly) and 'M' (monthly), which scale variance by 252, 52, and 12, respectively.
- **scale** (*float, optional*) – Value to use when scaling returns to annualize. If scale is provides, annualize is ignored and the value in scale is used.

Returns **fig** – Handle to the figure

Return type figure

Examples

```
>>> from arch import arch_model
>>> am = arch_model(None)
>>> sim_data = am.simulate([0.0, 0.01, 0.07, 0.92], 2520)
>>> am = arch_model(sim_data['data'])
>>> res = am.fit(update_freq=0, disp='off')
>>> fig = res.plot()
```

Produce a plot with annualized volatility

```
>>> fig = res.plot(annualize='D')
```

Override the usual scale of 252 to use 360 for an asset that trades most days of the year

```
>>> fig = res.plot(scale=360)
```

summary()

Constructs a summary of the results from a fit model.

Returns **summary** – Object that contains tables and facilitated export to text, html or latex

Return type Summary instance

1.1.2 ARCH Modeling

This setup code is required to run in an IPython notebook

```
In [1]: import warnings
        warnings.simplefilter('ignore')

        %matplotlib inline
        import seaborn
        seaborn.set_style('darkgrid')
```

```
In [2]: seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
        seaborn.mpl.rcParams['savefig.dpi'] = 90
        seaborn.mpl.rcParams['font.family'] = 'serif'
        seaborn.mpl.rcParams['font.size'] = 14
```

Setup

These examples will all make use of financial data from Yahoo! Finance imported using `pandas-datareader`.

```
In [3]: import datetime as dt
        import pandas_datareader.data as web
        st = dt.datetime(1988,1,1)
        en = dt.datetime(2018,1,1)
        data = web.get_data_famafrench('F-F_Research_Data_Factors_daily', start=st, end=en)
        mkt_returns = data[0]['Mkt-RF'] +  data[0]['RF']
        returns = mkt_returns
        figure = returns.plot()
```

Specifying Common Models

The simplest way to specify a model is to use the model constructor `arch.arch_model` which can specify most common models. The simplest invocation of `arch` will return a model with a constant mean, GARCH(1,1) volatility process and normally distributed errors.

$$r_t = \mu + \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2$$

$$\epsilon_t = \sigma_t e_t, \;\; e_t \sim N(0,1)$$

The model is estimated by calling `fit`. The optional inputs `iter` controls the frequency of output form the optimizer, and `disp` controls whether convergence information is returned. The results class returned offers direct access to the estimated parameters and related quantities, as well as a `summary` of the estimation results.

GARCH (with a Constant Mean)

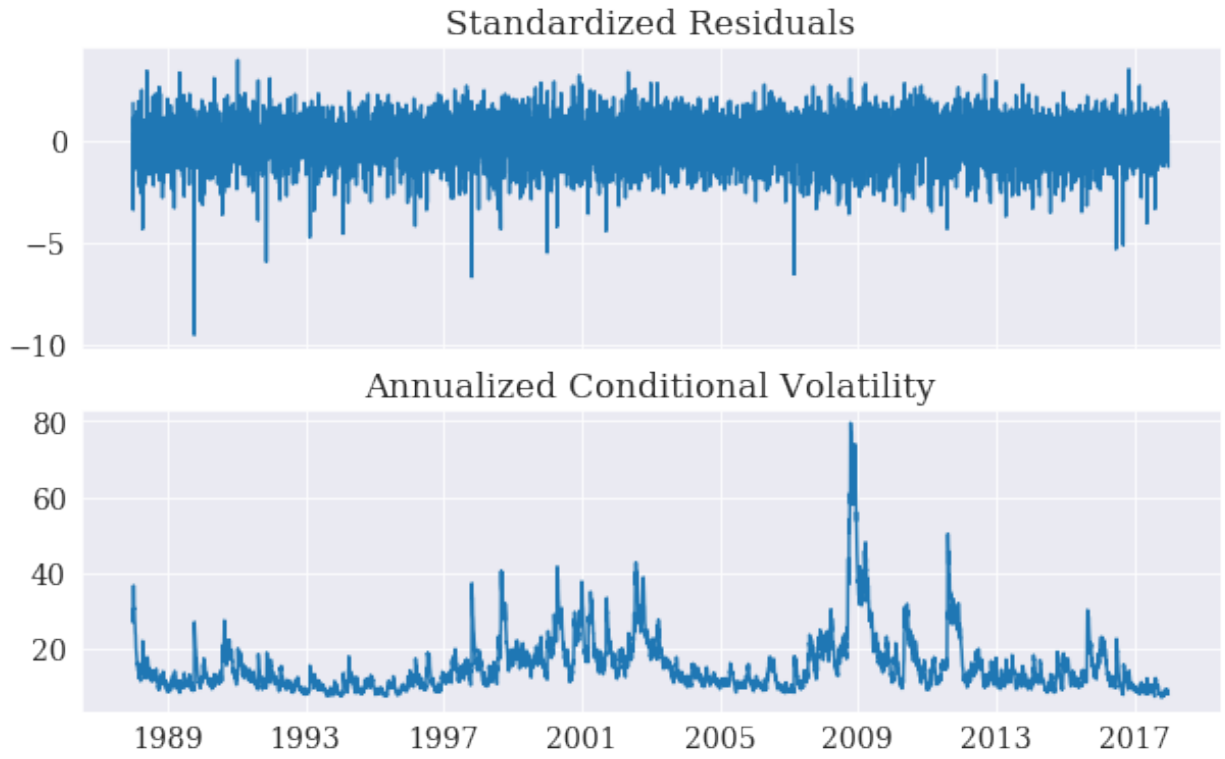
The default set of options produces a model with a constant mean, GARCH(1,1) conditional variance and normal errors.

```
In [4]: from arch import arch_model
        am = arch_model(returns)
        res = am.fit(update_freq=5)
        print(res.summary())
```

```
Iteration:      5,   Func. Count:     41,   Neg. LLF: 9820.63969763662
Iteration:     10,   Func. Count:     74,   Neg. LLF: 9817.274895653107
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 9817.274187120329
            Iterations: 13
            Function evaluations: 92
            Gradient evaluations: 13
                    Constant Mean - GARCH Model Results
==============================================================================
Dep. Variable:                    None   R-squared:                      -0.000
Mean Model:             Constant Mean   Adj. R-squared:                 -0.000
Vol Model:                      GARCH   Log-Likelihood:               -9817.27
Distribution:                  Normal   AIC:                           19642.5
Method:            Maximum Likelihood   BIC:                           19670.3
                                        No. Observations:                 7561
Date:                Thu, Sep 27 2018   Df Residuals:                     7557
Time:                        15:25:43   Df Model:                            4
                                Mean Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
mu             0.0687  9.098e-03      7.555  4.176e-14 [5.091e-02,8.657e-02]
                              Volatility Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.0151  4.174e-03      3.622  2.923e-04 [6.937e-03,2.330e-02]
alpha[1]       0.0836  1.231e-02      6.788  1.138e-11   [5.943e-02,  0.108]
beta[1]        0.9014  1.436e-02     62.771      0.000     [  0.873,  0.930]
==============================================================================

Covariance estimator: robust
```

`plot()` can be used to quickly visualize the standardized residuals and conditional volatility.

```
In [5]: fig = res.plot(annualize='D')
```



GJR-GARCH

Additional inputs can be used to construct other models. This example sets o to 1, which includes one lag of an asymmetric shock which transforms a GARCH model into a GJR-GARCH model with variance dynamics given by

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \gamma\epsilon_{t-1}^2 I_{[\epsilon_{t-1}<0]} + \beta\sigma_{t-1}^2$$

where I is an indicator function that takes the value 1 when its argument is true.

The log likelihood improves substantially with the introduction of an asymmetric term, and the parameter estimate is highly significant.

```
In [6]: am = arch_model(returns, p=1, o=1, q=1)
        res = am.fit(update_freq=5, disp='off')
        print(res.summary())
```

```
                    Constant Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:                    None   R-squared:                      -0.000
Mean Model:             Constant Mean   Adj. R-squared:                 -0.000
Vol Model:                  GJR-GARCH   Log-Likelihood:               -9713.51
Distribution:                  Normal   AIC:                           19437.0
Method:            Maximum Likelihood   BIC:                           19471.7
                                        No. Observations:                 7561
Date:                Thu, Sep 27 2018   Df Residuals:                     7556
Time:                        15:25:43   Df Model:                            5
                            Mean Model
==============================================================================
```

```
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0442  8.759e-03      5.051  4.398e-07 [2.707e-02,6.141e-02]
                            Volatility Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0186  4.608e-03      4.044  5.252e-05  [9.603e-03,2.766e-02]
alpha[1]   5.9351e-03  5.981e-03      0.992      0.321 [-5.788e-03,1.766e-02]
gamma[1]       0.1355  2.086e-02      6.495  8.291e-11    [9.459e-02,  0.176]
beta[1]        0.9044  1.464e-02     61.778      0.000      [  0.876,  0.933]
============================================================================

Covariance estimator: robust
```

TARCH/ZARCH

TARCH (also known as ZARCH) model the *volatility* using absolute values. This model is specified using `power=1.0` since the default power, 2, corresponds to variance processes that evolve in squares.

The volatility process in a TARCH model is given by

$$\sigma_t = \omega + \alpha \left|\epsilon_{t-1}\right| + \gamma \left|\epsilon_{t-1}\right| I_{[\epsilon_{t-1}<0]} + \beta\sigma_{t-1}$$

More general models with other powers (κ) have volatility dynamics given by

$$\sigma_t^\kappa = \omega + \alpha \left|\epsilon_{t-1}\right|^\kappa + \gamma \left|\epsilon_{t-1}\right|^\kappa I_{[\epsilon_{t-1}<0]} + \beta\sigma_{t-1}^\kappa$$

where the conditional variance is $\left(\sigma_t^\kappa\right)^{2/\kappa}$.

The TARCH model also improves the fit, although the change in the log likelihood is less dramatic.

```
In [7]: am = arch_model(returns, p=1, o=1, q=1, power=1.0)
        res = am.fit(update_freq=5)
        print(res.summary())
```

```
Iteration:      5,   Func. Count:     54,   Neg. LLF: 9701.46915224492
Iteration:     10,   Func. Count:     94,   Neg. LLF: 9685.575185675976
Iteration:     15,   Func. Count:    130,   Neg. LLF: 9683.248100354442
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 9683.248099008848
            Iterations: 16
            Function evaluations: 137
            Gradient evaluations: 16
                  Constant Mean - TARCH/ZARCH Model Results
==============================================================================
Dep. Variable:                    None   R-squared:                      -0.000
Mean Model:             Constant Mean   Adj. R-squared:                 -0.000
Vol Model:                TARCH/ZARCH   Log-Likelihood:               -9683.25
Distribution:                  Normal   AIC:                           19376.5
Method:            Maximum Likelihood   BIC:                           19411.1
                                        No. Observations:                 7561
Date:                Thu, Sep 27 2018   Df Residuals:                     7556
Time:                        15:25:43   Df Model:                            5
                            Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0386  8.899e-03      4.338  1.437e-05 [2.116e-02,5.604e-02]
```

```
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0215  4.333e-03      4.961  6.998e-07 [1.301e-02,2.999e-02]
alpha[1]       0.0137  6.461e-03      2.114  3.448e-02 [9.980e-04,2.632e-02]
gamma[1]       0.1195  1.440e-02      8.301  1.031e-16   [9.130e-02,  0.148]
beta[1]        0.9213  1.010e-02     91.219      0.000     [  0.901,  0.941]
============================================================================

Covariance estimator: robust
```

Student's T Errors

Financial returns are often heavy tailed, and a Student's T distribution is a simple method to capture this feature. The call to `arch` changes the distribution from a Normal to a Students's T.

The standardized residuals appear to be heavy tailed with an estimated degree of freedom near 10. The log-likelihood also shows a large increase.

```
In [8]: am = arch_model(returns, p=1, o=1, q=1, power=1.0, dist='StudentsT')
        res = am.fit(update_freq=5)
        print(res.summary())
```

```
Iteration:      5,   Func. Count:     56,   Neg. LLF: 9535.100867863783
Iteration:     10,   Func. Count:    100,   Neg. LLF: 9506.165300845836
Iteration:     15,   Func. Count:    140,   Neg. LLF: 9505.38232521017
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 9505.381447531525
            Iterations: 17
            Function evaluations: 156
            Gradient evaluations: 17
                  Constant Mean - TARCH/ZARCH Model Results
==============================================================================================
Dep. Variable:                             None   R-squared:                       -0.000
Mean Model:                     Constant Mean   Adj. R-squared:                  -0.000
Vol Model:                        TARCH/ZARCH   Log-Likelihood:                -9505.38
Distribution:      Standardized Student's t   AIC:                             19022.8
Method:                  Maximum Likelihood   BIC:                             19064.3
                                              No. Observations:                   7561
Date:                       Thu, Sep 27 2018   Df Residuals:                       7555
Time:                               15:25:43   Df Model:                             6
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0593  8.031e-03      7.383  1.553e-13 [4.355e-02,7.503e-02]
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0153  3.142e-03      4.875  1.087e-06  [9.159e-03,2.148e-02]
alpha[1]   7.9884e-03  4.902e-03      1.630      0.103 [-1.619e-03,1.760e-02]
gamma[1]       0.1258  1.335e-02      9.420  4.514e-21    [9.959e-02,  0.152]
beta[1]        0.9291  8.308e-03    111.830      0.000      [  0.913,  0.945]
                              Distribution
============================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
----------------------------------------------------------------------------
```

```
nu              6.8360      0.575     11.896  1.241e-32 [  5.710,  7.962]
============================================================================

Covariance estimator: robust
```

Fixing Parameters

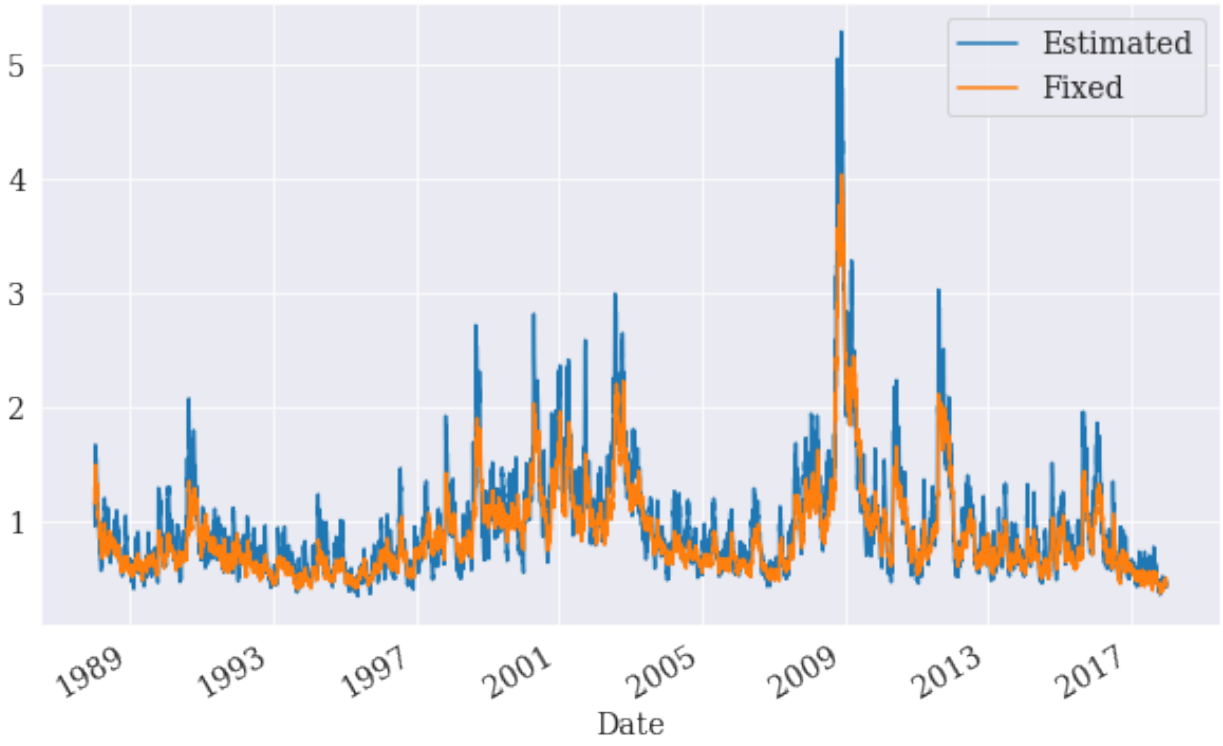
In some circumstances, fixed rather than estimated parameters might be of interest. A model-result-like class can be generated using the `fix()` method. The class returend is identical to the usual model result class except that information about inference (standard errors, t-stats, etc) is not available.

In the example, I fix the parameters to a symmetric version of the previously estimated model.

```
In [9]: fixed_res = am.fix([0.0235, 0.01, 0.06, 0.0, 0.9382, 8.0])
        print(fixed_res.summary())
```

```
                     Constant Mean - TARCH/ZARCH Model Results
==============================================================================================
Dep. Variable:                             None   R-squared:                               --
Mean Model:                      Constant Mean   Adj. R-squared:                          --
Vol Model:                         TARCH/ZARCH   Log-Likelihood:                    -9690.73
Distribution:       Standardized Student's t   AIC:                                 19393.5
Method:            User-specified Parameters   BIC:                                 19435.1
                                               No. Observations:                       7561
Date:                         Thu, Sep 27 2018
Time:                                 15:25:43
      Mean Model
=====================
                  coef
---------------------
mu              0.0235
   Volatility Model
=====================
                  coef
---------------------
omega           0.0100
alpha[1]        0.0600
gamma[1]        0.0000
beta[1]         0.9382
     Distribution
=====================
                  coef
---------------------
nu              8.0000
=====================

Results generated with user-specified parameters.
Since the model was not estimated, there are no std. errors.
```

```
In [10]: import pandas as pd
         df = pd.concat([res.conditional_volatility,fixed_res.conditional_volatility],1)
         df.columns = ['Estimated', 'Fixed']
         subplot = df.plot()
```



Building a Model From Components

Models can also be systematically assembled from the three model components:

- A mean model (`arch.mean`)
  - Zero mean (`ZeroMean`) - useful if using residuals from a model estimated separately
  - Constant mean (`ConstantMean`) - common for most liquid financial assets
  - Autoregressive (`ARX`) with optional exogenous regressors
  - Heterogeneous (`HARX`) autoregression with optional exogenous regressors
  - Exogenous regressors only (`LS`)
- A volatility process (`arch.volatility`)
  - ARCH (`ARCH`)
  - GARCH (`GARCH`)
  - GJR-GARCH (`GARCH` using `o` argument)
  - TARCH/ZARCH (`GARCH` using `power` argument set to `1`)
  - Power GARCH and Asymmetric Power GARCH (`GARCH` using `power`)
  - Exponentially Weighted Moving Average Variance with estimated coefficient (`EWMAVariance`)
  - Heterogeneous ARCH (`HARCH`)
  - Parameterless Models
    - Exponentially Weighted Moving Average Variance, known as RiskMetrics (`EWMAVariance`)

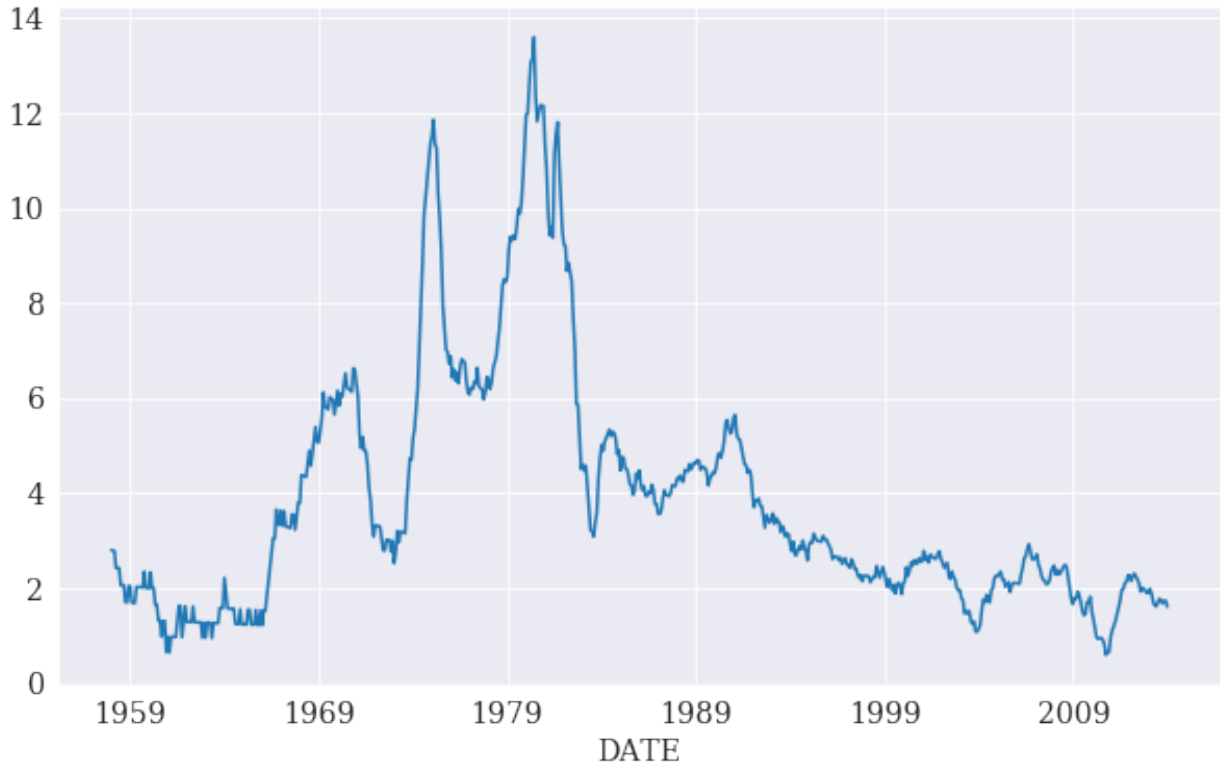
    - Weighted averages of EWMAs, known as the RiskMetrics 2006 methodology (`RiskMetrics2006`)

- A distribution (`arch.distribution`)
  - Normal (`Normal`)
  - Standardized Students's T (`StudentsT`)

Mean Models

The first choice is the mean model. For many liquid financial assets, a constant mean (or even zero) is adequate. For other series, such as inflation, a more complicated model may be required. These examples make use of Core CPI downloaded from the Federal Reserve Economic Data site.

```
In [11]: core_cpi = web.DataReader("CPILFESL", "fred", dt.datetime(1957,1,1), dt.datetime(2014,1,1))
         ann_inflation = 100 * core_cpi.CPILFESL.pct_change(12).dropna()
         fig = ann_inflation.plot()
```



All mean models are initialized with constant variance and normal errors. For `ARX` models, the `lags` argument specifies the lags to include in the model.

```
In [12]: from arch.univariate import ARX
         ar = ARX(ann_inflation, lags = [1, 3, 12])
         print(ar.fit().summary())
```

```
                     AR - Constant Variance Model Results
==============================================================================
Dep. Variable:                CPILFESL   R-squared:                       0.991
Mean Model:                         AR   Adj. R-squared:                  0.991
Vol Model:          Constant Variance   Log-Likelihood:                -13.7178
Distribution:                  Normal   AIC:                            37.4356
```

```
Method:            Maximum Likelihood   BIC:                           59.9043
                                        No. Observations:                  661
Date:                Thu, Sep 27 2018   Df Residuals:                      656
Time:                        15:25:44   Df Model:                            5
                                Mean Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
Const            0.0424  2.197e-02      1.929  5.372e-02 [-6.781e-04,8.542e-02]
CPILFESL[1]      1.1927  3.513e-02     33.950 1.229e-252      [  1.124,  1.262]
CPILFESL[3]     -0.1803  4.123e-02     -4.374  1.218e-05   [ -0.261,-9.954e-02]
CPILFESL[12]    -0.0235  1.384e-02     -1.695  9.001e-02 [-5.060e-02,3.663e-03]
                           Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
sigma2         0.0610  6.992e-03      8.728  2.590e-18 [4.733e-02,7.474e-02]
============================================================================

Covariance estimator: White's Heteroskedasticity Consistent Estimator
```

Volatility Processes

Volatility processes can be added a a mean model using the `volatility` property. This example adds an ARCH(5) process to model volatility. The arguments `iter` and `disp` are used in `fit()` to suppress estimation output.

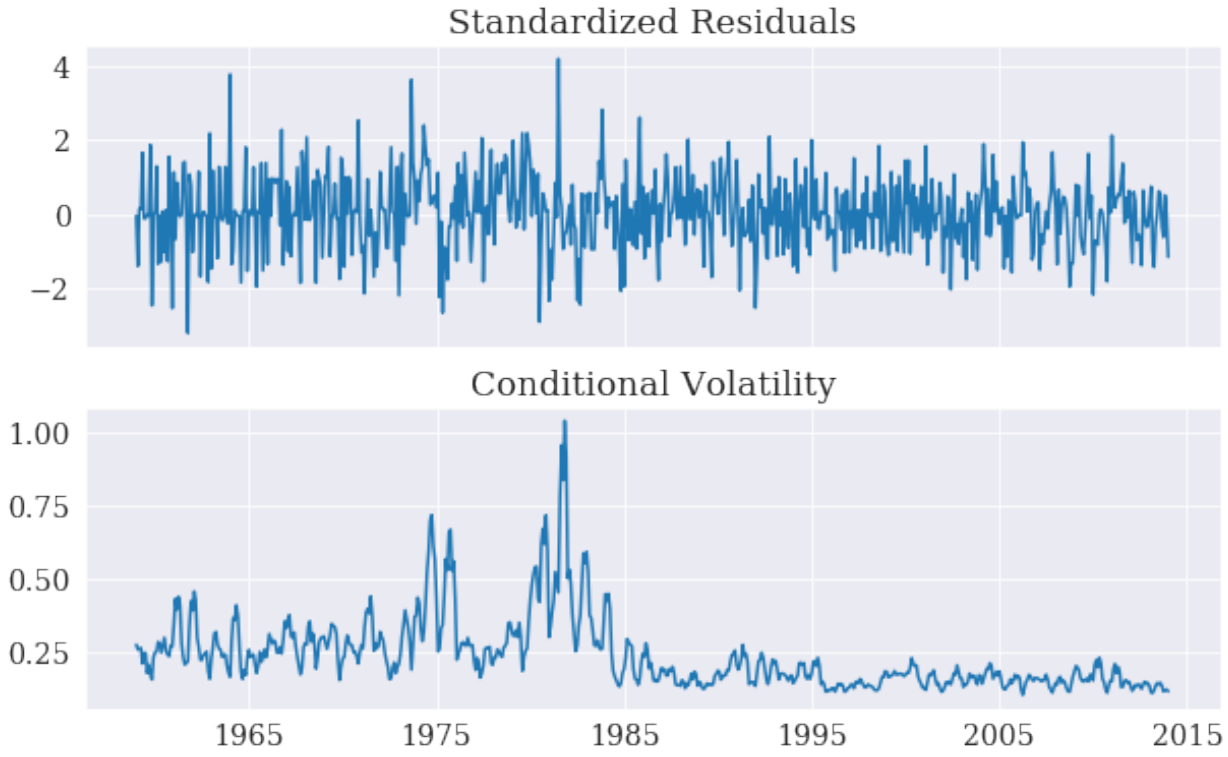
```
In [13]: from arch.univariate import ARCH, GARCH
         ar.volatility = ARCH(p=5)
         res = ar.fit(update_freq=0, disp='off')
         print(res.summary())
```

```
                          AR - ARCH Model Results
============================================================================
Dep. Variable:               CPILFESL   R-squared:                       0.991
Mean Model:                        AR   Adj. R-squared:                  0.991
Vol Model:                       ARCH   Log-Likelihood:                83.8447
Distribution:                  Normal   AIC:                           -147.689
Method:            Maximum Likelihood   BIC:                           -102.752
                                        No. Observations:                  661
Date:                Thu, Sep 27 2018   Df Residuals:                      651
Time:                        15:25:44   Df Model:                           10
                                Mean Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
Const            0.0268  2.207e-02      1.212      0.225 [-1.651e-02,7.002e-02]
CPILFESL[1]      1.0854  3.840e-02     28.266 9.157e-176      [  1.010,  1.161]
CPILFESL[3]     -0.0755  4.148e-02     -1.821  6.854e-02    [ -0.157,5.747e-03]
CPILFESL[12]    -0.0210  1.195e-02     -1.761  7.829e-02 [-4.446e-02,2.381e-03]
                           Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega      9.9111e-03  2.195e-03      4.515  6.342e-06 [5.608e-03,1.421e-02]
alpha[1]       0.1291  4.090e-02      3.157  1.594e-03   [4.895e-02,  0.209]
alpha[2]       0.2279  6.468e-02      3.524  4.258e-04     [  0.101,  0.355]
alpha[3]       0.1698  7.121e-02      2.385  1.709e-02   [3.026e-02,  0.309]
alpha[4]       0.2643  8.318e-02      3.177  1.489e-03     [  0.101,  0.427]
```

```
alpha[5]       0.1686  7.426e-02      2.271  2.316e-02    [2.309e-02,  0.314]
================================================================================

Covariance estimator: robust
```

Plotting the standardized residuals and the conditional volatility shows some large (in magnitude) errors, even when standardized.

```
In [14]: fig = res.plot()
```



Distributions

Finally the distribution can be changed from the default normal to a standardized Student's T using the `distribution` property of a mean model.

The Student's t distribution improves the model, and the degree of freedom is estimated to be near 8.

```
In [15]: from arch.univariate import StudentsT
         ar.distribution = StudentsT()
         res = ar.fit(update_freq=0, disp='off')
         print(res.summary())
```

```
                           AR - ARCH Model Results
==============================================================================
Dep. Variable:                CPILFESL   R-squared:                       0.991
Mean Model:                         AR   Adj. R-squared:                  0.991
Vol Model:                        ARCH   Log-Likelihood:                89.4699
Distribution:      Standardized Student's t   AIC:                    -156.940
Method:                  Maximum Likelihood   BIC:                    -107.509
                                              No. Observations:              661
Date:                     Thu, Sep 27 2018   Df Residuals:                  650
Time:                             15:25:44   Df Model:                       11
```

```
                             Mean Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
Const            0.0281  2.252e-02      1.249      0.212 [-1.600e-02,7.227e-02]
CPILFESL[1]      1.0851  3.920e-02     27.683 1.134e-168      [  1.008,  1.162]
CPILFESL[3]     -0.0697  4.277e-02     -1.629      0.103    [ -0.154,1.416e-02]
CPILFESL[12]    -0.0266  1.525e-02     -1.742  8.157e-02 [-5.643e-02,3.328e-03]
                          Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0117  3.142e-03      3.710  2.071e-04 [5.499e-03,1.782e-02]
alpha[1]       0.1669  5.233e-02      3.189  1.426e-03   [6.433e-02,  0.269]
alpha[2]       0.2185  6.648e-02      3.287  1.012e-03   [8.823e-02,  0.349]
alpha[3]       0.1370  6.928e-02      1.977  4.805e-02   [1.178e-03,  0.273]
alpha[4]       0.2186  7.738e-02      2.825  4.734e-03   [6.691e-02,  0.370]
alpha[5]       0.1596  8.579e-02      1.861  6.275e-02  [-8.496e-03,  0.328]
                          Distribution
============================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
----------------------------------------------------------------------------
nu             9.1012      3.834      2.374  1.759e-02 [  1.588, 16.615]
============================================================================

Covariance estimator: robust
```

WTI Crude

The next example uses West Texas Intermediate Crude data from FRED. Theee models are fit using alternative distributional assumptions. The results are printed, where we can see that the normal has a much lower log-likelihood than either the Standard Student's T or the Standardized Skew Student's T – however, these two are fairly close. The closeness of the T and the Skew T indicate that returns are not heavily skewed.

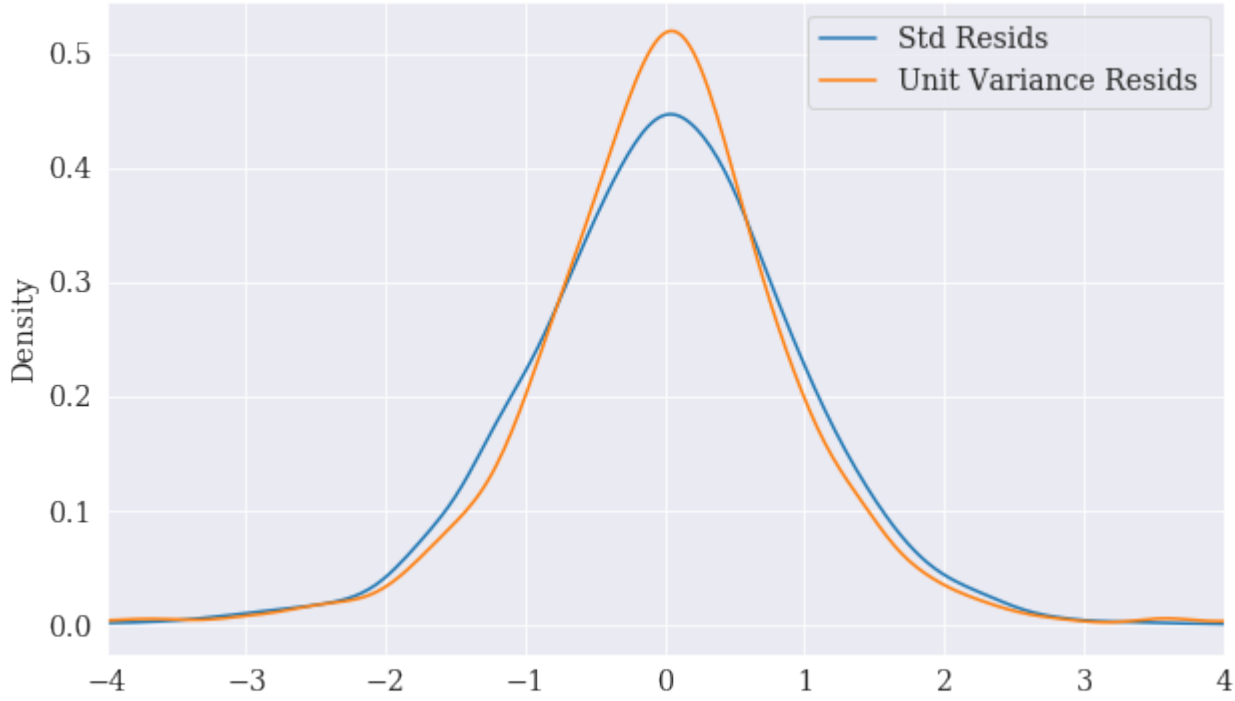
```
In [16]: from collections import OrderedDict
         crude=web.get_data_fred('DCOILWTICO',dt.datetime(2000, 1, 1),dt.datetime(2015, 1, 1))
         crude_ret = 100 * crude.dropna().pct_change().dropna()
         res_normal = arch_model(crude_ret).fit(disp='off')
         res_t = arch_model(crude_ret, dist='t').fit(disp='off')
         res_skewt = arch_model(crude_ret, dist='skewt').fit(disp='off')
         lls = pd.Series(OrderedDict((('normal', res_normal.loglikelihood),
                         ('t', res_t.loglikelihood),
                         ('skewt', res_skewt.loglikelihood))))
         print(lls)
         params = pd.DataFrame(OrderedDict((('normal', res_normal.params),
                         ('t', res_t.params),
                         ('skewt', res_skewt.params))))
         print(params)
```

```
normal   -8227.359031
t        -8128.534732
skewt    -8126.303934
dtype: float64
            normal         t     skewt
alpha[1]  0.054488  0.046069  0.045908
beta[1]   0.940953  0.949954  0.950364
lambda         NaN       NaN -0.048593
mu        0.065643  0.076392  0.057599
```

```
nu              NaN  6.841504  6.889653
omega      0.034733  0.026497  0.025239
```

The standardized residuals can be computed by dividing the residuals by the conditional volatility. These are plotted along with the (unstandardized, but scaled) residuals. The non-standardized residuals are more peaked in the center indicating that the distribution is somewhat more heavy tailed than that of the standardized residuals.

```
In [17]: std_resid = res_normal.resid / res_normal.conditional_volatility
         unit_var_resid = res_normal.resid / res_normal.resid.std()
         df = pd.concat([std_resid, unit_var_resid],1)
         df.columns = ['Std Resids', 'Unit Variance Resids']
         subplot = df.plot(kind='kde', xlim=(-4,4))
```



1.1.3 Forecasting

Multi-period forecasts can be easily produced for ARCH-type models using forward recursion, with some caveats. In particular, models that are non-linear in the sense that they do not evolve using squares or residuals do not normally have analytically tractable multi-period forecasts available.

All models support three methods of forecasting:

- Analytical: analytical forecasts are always available for the 1-step ahead forecast due to the structure of ARCH-type models. Multi-step analytical forecasts are only available for model which are linear in the square of the residual, such as GARCH or HARCH.
- Simulation: simulation-based forecasts are always available for any horizon, although they are only useful for horizons larger than 1 since the first out-of-sample forecast from an ARCH-type model is always fixed. Simulation-based forecasts make use fo the structure of an ARCH-type model to forward simulate using the assumed distribution of residuals, e.g., a Normal or Student's t.
- Bootstrap: bootstrap-based forecasts are similar to simulation based forecasts except that they make use of the standardized residuals from the actual data used in the estimation rather than assuming a specific distribution.

Like simulation-base forecasts, bootstrap-based forecasts are only useful for horizons larger than 1. Additionally, the bootstrap forecasting method requires a minimal amount of in-sample data to use prior to producing the forecasts.

This document will use a standard GARCH(1,1) with a constant mean to explain the choices available for forecasting. The model can be described as

$$\begin{aligned} r_t &= \mu + \epsilon_t & (1.4)\\ \epsilon_t &= \sigma_t e_t & (1.5)\\ \sigma_t^2 &= \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2 & (1.6)\\ e_t &\sim N(0,1) & (1.7) \end{aligned}$$

In code this model can be constructed using data from the S&P 500 using

```
from arch import arch_model
import datetime as dt
import pandas_datareader.data as web
start = dt.datetime(2000,1,1)
end = dt.datetime(2014,1,1)
sp500 = web.get_data_yahoo('^GSPC', start=start, end=end)
returns = 100 * sp500['Adj Close'].pct_change().dropna()
am = arch_model(returns, vol='Garch', p=1, o=0, q=1, dist='Normal')
```

The model will be estimated using the first 10 years to estimate parameters and then forecasts will be produced for the final 5.

```
split_date = dt.datetime(2010,1,1)
res = am.fit(last_obs=split_date)
```

Analytical Forecasts

Analytical forecasts are available for most models that evolve in terms of the squares of the model residuals, e.g., GARCH, HARCH, etc. These forecasts exploit the relationship $E_t[\epsilon_{t+1}^2] = \sigma_{t+1}^2$ to recursively compute forecasts.

Variance forecasts are constructed for the conditional variances as

$$\begin{aligned} \sigma_{t+1}^2 &= \omega + \alpha\epsilon_t^2 + \beta\sigma_{t-1}^2 & (1.8)\\ \sigma_{t+h}^2 &= \omega + \alpha E_t[\epsilon_{t+h-1}^2] + \beta E_t[\sigma_{t+h-1}^2]\, h \geq 2 & (1.9)\\ &= \omega + (\alpha+\beta)\, E_t[\sigma_{t+h-1}^2]\, h \geq 2 & (1.10) \end{aligned}$$

```
forecasts = res.forecast(horizon=5, start=split_date)
forecasts.variance[split_date:].plot()
```

Simulation Forecasts

Simulation-based forecasts use the model random number generator to simulate draws of the standardized residuals, e_{t+h}. These are used to generate a pre-specified number of paths fof the variances which are then averaged to produce the forecasts. In models like GARCH which evolve in the squares of the residuals, there are few advantages to simulation-based forecasting. These methods are more valuable when producing multi-step forecasts from models that do not have closed form multi-step forecasts such as EGARCH models.

Assume there are B simulated paths. A single simulated path is generated using

$$\begin{aligned} \sigma_{t+h,b}^2 &= \omega + \alpha\epsilon_{t+h-1,b}^2 + \beta\sigma_{t+h-1,b}^2 & (1.11)\\ \epsilon_{t+h,b} &= e_{t+h,b}\sqrt{\sigma_{t+h,b}^2} & (1.12) \end{aligned}$$

where the simulated shocks are $e_{t+1,b}, e_{t+2,b}, \ldots, e_{t+h,b}$ where b is included to indicate that the simulations are independent across paths. Note that the first residual, ϵ_t, is in-sample and so is not simulated.

The final variance forecasts are then computed using the B simulations

$$E_t[\epsilon_{t+h}^2] = \sigma_{t+h}^2 = B^{-1}\sum_{b=1}^{B}\sigma_{t+h,b}^2. \tag{1.13}$$

```
forecasts = res.forecast(horizon=5, start=split_date, method='simulation')
```

Bootstrap Forecasts

Bootstrap-based forecasts are virtually identical to simulation-based forecasts except that the standardized residuals are generated by the model. These standardized residuals are generated using the observed data and the estimated parameters as

$$\hat{e}_t = \frac{r_t - \hat{\mu}}{\hat{\sigma}_t} \tag{1.14}$$

The generation scheme is identical to the simulation-based method except that the simulated shocks are drawn (i.i.d., with replacement) from $\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_t$. so that only data available at time t are used to simulate the paths.

Forecasting Options

The *forecast()* method is attached to a model fit result.‘

- `params` - The model parameters used to forecast the mean and variance. If not specified, the parameters estimated during the call to `fit` the produced the result are used.
- `horizon` - A positve integer value indicating the maximum horizon to produce forecasts.
- `start` - A positive integer or, if the input to the mode is a DataFrame, a date (string, datetime, datetime64 or Timestamp). Forecasts are produced from `start` until the end of the sample. If not provided, `start` is set to the length of the input data minus 1 so that only 1 forecast is produced.
- `align` - One of ‘origin’ (default) or ‘target’ that describes how the forecasts aligned in the output. Origin aligns forecasts to the last observation used in producing the forecast, while target aligns forecasts to the observation index that is being forecast.
- `method` - One of ‘analytic’ (default), ‘simulation’ or ‘bootstrap’ that describes the method used to produce the forecasts. Not all methods are available for all horizons.
- `simulations` - A non-negative integer indicating the number of simulation to use when `method` is ‘simulation’ or ‘bootstrap’

Understanding Forecast Output

Any call to *forecast()* returns a *ARCHModelForecast* object with has 3 core attributes and 1 which may be useful when using simulation- or bootstrap-based forecasts.

The three core attributes are

- `mean` - The forecast conditional mean.
- `variance` - The forecast conditional variance.
- `residual_variance` - The forecast conditional variance of residuals. This will differ from `variance` whenever the model has dynamics (e.g. an AR model) for horizons larger than 1.

Each attribute contains a `DataFrame` with a common structure.

```
print(forecasts.variance.tail())
```

which returns

```
                h.1       h.2       h.3       h.4       h.5
Date
2013-12-24  0.489534  0.495875  0.501122  0.509194  0.518614
2013-12-26  0.474691  0.480416  0.483664  0.491932  0.502419
2013-12-27  0.447054  0.454875  0.462167  0.467515  0.475632
2013-12-30  0.421528  0.430024  0.439856  0.448282  0.457368
2013-12-31  0.407544  0.415616  0.422848  0.430246  0.439451
```

The values in the columns `h.1` are one-step ahead forecast, while values in `h.2`, ..., `h.5` are 2, ..., 5-observation ahead forecasts. The output is aligned so that the Date column is the final data used to generate the forecast, so that `h.1` in row `2013-12-31` is the one-step ahead forecast made using data **up to and including** December 31, 2013.

By default forecasts are only produced for observations after the final observation used to estimate the model.

```
day = dt.timedelta(1)
print(forecasts.variance[split_date - 5 * day:split_date + 5 * day])
```

which produces

```
                h.1       h.2       h.3       h.4       h.5
Date
2009-12-28      NaN       NaN       NaN       NaN       NaN
2009-12-29      NaN       NaN       NaN       NaN       NaN
2009-12-30      NaN       NaN       NaN       NaN       NaN
2009-12-31      NaN       NaN       NaN       NaN       NaN
2010-01-04  0.739303  0.741100  0.744529  0.746940  0.752688
2010-01-05  0.695349  0.702488  0.706812  0.713342  0.721629
2010-01-06  0.649343  0.654048  0.664055  0.672742  0.681263
```

The output will always have as many rows as the data input. Values that are not forecast are `nan` filled.

Output Classes

class arch.univariate.base.**ARCHModelForecast**(*index*, *mean*, *variance*, *residual_variance*, *simulated_paths=None*, *simulated_variances=None*, *simulated_residual_variances=None*, *simulated_residuals=None*, *align='origin'*)

Container for forecasts from an ARCH Model

Parameters

- **index** (*{list, ndarray}*) –
- **mean** (*ndarray*) –
- **variance** (*ndarray*) –
- **residual_variance** (*ndarray*) –
- **simulated_paths** (*ndarray, optional*) –
- **simulated_variances** (*ndarray, optional*) –

- **simulated_residual_variances** (*ndarray, optional*) –
- **simulated_residuals** (*ndarray, optional*) –
- **align** (*{'origin', 'target'}*) –

mean
DataFrame – Forecast values for the conditional mean of the process

variance
DataFrame – Forecast values for the conditional variance of the process

residual_variance
DataFrame – Forecast values for the conditional variance of the residuals

class arch.univariate.base.**ARCHModelForecastSimulation**(*values*, *residuals*, *variances*, *residual_variances*)
Container for a simulation or bootstrap-based forecasts from an ARCH Model

Parameters

- **values** –
- **residuals** –
- **variances** –
- **residual_variances** –

values
DataFrame – Simulated values of the process

residuals
DataFrame – Simulated residuals used to produce the values

variances
DataFrame – Simulated variances of the values

residual_variances
DataFrame – Simulated variance of the residuals

1.1.4 Volatility Forecasting

This setup code is required to run in an IPython notebook

```
In [1]: import warnings
        warnings.simplefilter('ignore')

        %matplotlib inline
        import seaborn
        seaborn.set_style('darkgrid')
```

```
In [2]: seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
        seaborn.mpl.rcParams['savefig.dpi'] = 90
        seaborn.mpl.rcParams['font.family'] = 'serif'
        seaborn.mpl.rcParams['font.size'] = 14
```

Data

These examples make use of S&P 500 data from Yahoo! using the pandas-datareader package to manage data download.

```
In [3]: import datetime as dt
        import sys

        import numpy as np
        import pandas as pd
        import pandas_datareader.data as web

        from arch import arch_model

        start = dt.datetime(2000,1,1)
        end = dt.datetime(2017,1,1)
        data = web.get_data_famafrench('F-F_Research_Data_Factors_daily', start=start, end=end)
        mkt_returns = data[0]['Mkt-RF'] +  data[0]['RF']
        returns = mkt_returns
```

Basic Forecasting

Forecasts can be generated for standard GARCH(p,q) processes using any of the three forecast generation methods:

- Analytical
- Simulation-based
- Bootstrap-based

Be default forecasts will only be produced for the final observation in the sample so that they are out-of-sample.

Forecasts start with specifying the model and estimating parameters.

```
In [4]: am = arch_model(returns, vol='Garch', p=1, o=0, q=1, dist='Normal')
        res = am.fit(update_freq=5)
```

```
Iteration:      5,   Func. Count:     39,   Neg. LLF: 6130.463290480577
Iteration:     10,   Func. Count:     71,   Neg. LLF: 6128.47317714312
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 6128.473168195432
            Iterations: 11
            Function evaluations: 77
            Gradient evaluations: 11
```

```
In [5]: forecasts = res.forecast()
```

Forecasts are contained in an `ARCHModelForecast` object which has 4 attributes:

- `mean` - The forecast means
- `residual_variance` - The forecast residual variances, that is $E_t[\epsilon_{t+h}^2]$
- `variance` - The forecast variance of the process, $E_t[r_{t+h}^2]$. The variance will differ from the residual variance whenever the model has mean dynamics, e.g., in an AR process.
- `simulations` - An object that contains detailed information about the simulations used to generate forecasts. Only used if the forecast `method` is set to `'simulation'` or `'bootstrap'`. If using `'analytical'` (the default), this is `None`.

The three main outputs are all returned in `DataFrames` with columns of the form `h.#` where `#` is the number of steps ahead. That is, `h.1` corresponds to one-step ahead forecasts while `h.10` corresponds to 10-steps ahead.

The default forecast only produces 1-step ahear forecasts.

```
In [6]: print(forecasts.mean.iloc[-3:])
        print(forecasts.residual_variance.iloc[-3:])
        print(forecasts.variance.iloc[-3:])
```

```
                h.1
Date
2016-12-28       NaN
2016-12-29       NaN
2016-12-30  0.061285
                h.1
Date
2016-12-28       NaN
2016-12-29       NaN
2016-12-30  0.400956
                h.1
Date
2016-12-28       NaN
2016-12-29       NaN
2016-12-30  0.400956
```

Longer horizon forecasts can be computed by passing the parameter `horizon`.

```
In [7]: forecasts = res.forecast(horizon=5)
        print(forecasts.residual_variance.iloc[-3:])
```

```
                h.1       h.2       h.3       h.4       h.5
Date
2016-12-28       NaN       NaN       NaN       NaN       NaN
2016-12-29       NaN       NaN       NaN       NaN       NaN
2016-12-30  0.400956  0.416563  0.431896  0.446961  0.461762
```

Values that are not computed are `nan`-filled.

Alternative Forecast Generation Schemes

Fixed Window Forecasting

Fixed-windows forecasting uses data up to a specified date to generate all forecasts after that date. This cna be implemented by passing the entire data in when initializing the model and then using `last_obs` when calling `fit`. `forecast()` will, by default, produce forecasts after this final date.

Note `last_obs` follow Python sequence rules so that the actual date in `last_obs` is not in the sample.

```
In [8]: res = am.fit(last_obs = '2011-1-1', update_freq=5)
        forecasts = res.forecast(horizon=5)
        print(forecasts.variance.dropna().head())
```

```
Iteration:      5,   Func. Count:     38,   Neg. LLF: 4204.91956121224
Iteration:     10,   Func. Count:     72,   Neg. LLF: 4202.815024845146
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 4202.812110685669
            Iterations: 12
            Function evaluations: 84
            Gradient evaluations: 12
                h.1       h.2       h.3       h.4       h.5
Date
2010-12-31  0.365727  0.376462  0.387106  0.397660  0.408124
2011-01-03  0.451526  0.461532  0.471453  0.481290  0.491043
2011-01-04  0.432131  0.442302  0.452387  0.462386  0.472300
2011-01-05  0.430051  0.440239  0.450341  0.460358  0.470289
2011-01-06  0.407841  0.418219  0.428508  0.438710  0.448825
```

Rolling Window Forecasting

Rolling window forecasts use a fixed sample length and then produce one-step from the final observation. These can be implemented using `first_obs` and `last_obs`.

```
In [9]: index = returns.index
        start_loc = 0
        end_loc = np.where(index >= '2010-1-1')[0].min()
        forecasts = {}
        for i in range(20):
            sys.stdout.write('.')
            sys.stdout.flush()
            res = am.fit(first_obs=i, last_obs=i+end_loc, disp='off')
            temp = res.forecast(horizon=3).variance
            fcast = temp.iloc[i+end_loc-1]
            forecasts[fcast.name] = fcast
        print()
        print(pd.DataFrame(forecasts).T)

...
                h.1       h.2       h.3
2009-12-31  0.598199  0.605960  0.613661
2010-01-04  0.771974  0.778431  0.784837
2010-01-05  0.724185  0.731008  0.737781
2010-01-06  0.674237  0.681423  0.688555
2010-01-07  0.637534  0.644995  0.652399
2010-01-08  0.601684  0.609451  0.617161
2010-01-11  0.562393  0.570450  0.578447
2010-01-12  0.613401  0.621098  0.628738
2010-01-13  0.623059  0.630676  0.638236
2010-01-14  0.584403  0.592291  0.600119
2010-01-15  0.654097  0.661483  0.668813
2010-01-19  0.725471  0.732355  0.739187
2010-01-20  0.758532  0.765176  0.771770
2010-01-21  0.958742  0.964005  0.969229
2010-01-22  1.272999  1.276121  1.279220
2010-01-25  1.182257  1.186084  1.189883
2010-01-26  1.110357  1.114637  1.118885
2010-01-27  1.044077  1.048777  1.053442
2010-01-28  1.085489  1.089873  1.094223
2010-01-29  1.088349  1.092875  1.097367
```

Recursive Forecast Generation

Recursive is similar to rolling except that the initial observation doesn't change. This can be easily implemented by dropping the `first_obs` input.

```
In [10]: import pandas as pd
         import numpy as np
         index = returns.index
         start_loc = 0
         end_loc = np.where(index >= '2010-1-1')[0].min()
         forecasts = {}
         for i in range(20):
             sys.stdout.write('.')
             sys.stdout.flush()
             res = am.fit(last_obs=i+end_loc, disp='off')
             temp = res.forecast(horizon=3).variance
```

```
            fcast = temp.iloc[i+end_loc-1]
            forecasts[fcast.name] = fcast
        print()
        print(pd.DataFrame(forecasts).T)

...
                h.1       h.2       h.3
2009-12-31  0.598199  0.605960  0.613661
2010-01-04  0.772200  0.778629  0.785009
2010-01-05  0.723347  0.730126  0.736853
2010-01-06  0.673796  0.680934  0.688017
2010-01-07  0.637555  0.644959  0.652306
2010-01-08  0.600834  0.608511  0.616129
2010-01-11  0.561436  0.569411  0.577324
2010-01-12  0.612214  0.619798  0.627322
2010-01-13  0.622095  0.629604  0.637055
2010-01-14  0.583425  0.591215  0.598945
2010-01-15  0.652960  0.660231  0.667447
2010-01-19  0.724212  0.730968  0.737673
2010-01-20  0.757280  0.763797  0.770264
2010-01-21  0.956394  0.961508  0.966583
2010-01-22  1.268445  1.271402  1.274337
2010-01-25  1.177405  1.180991  1.184549
2010-01-26  1.106326  1.110404  1.114450
2010-01-27  1.040930  1.045462  1.049959
2010-01-28  1.082130  1.086370  1.090577
2010-01-29  1.082251  1.086487  1.090690
```

TARCH

Analytical Forecasts

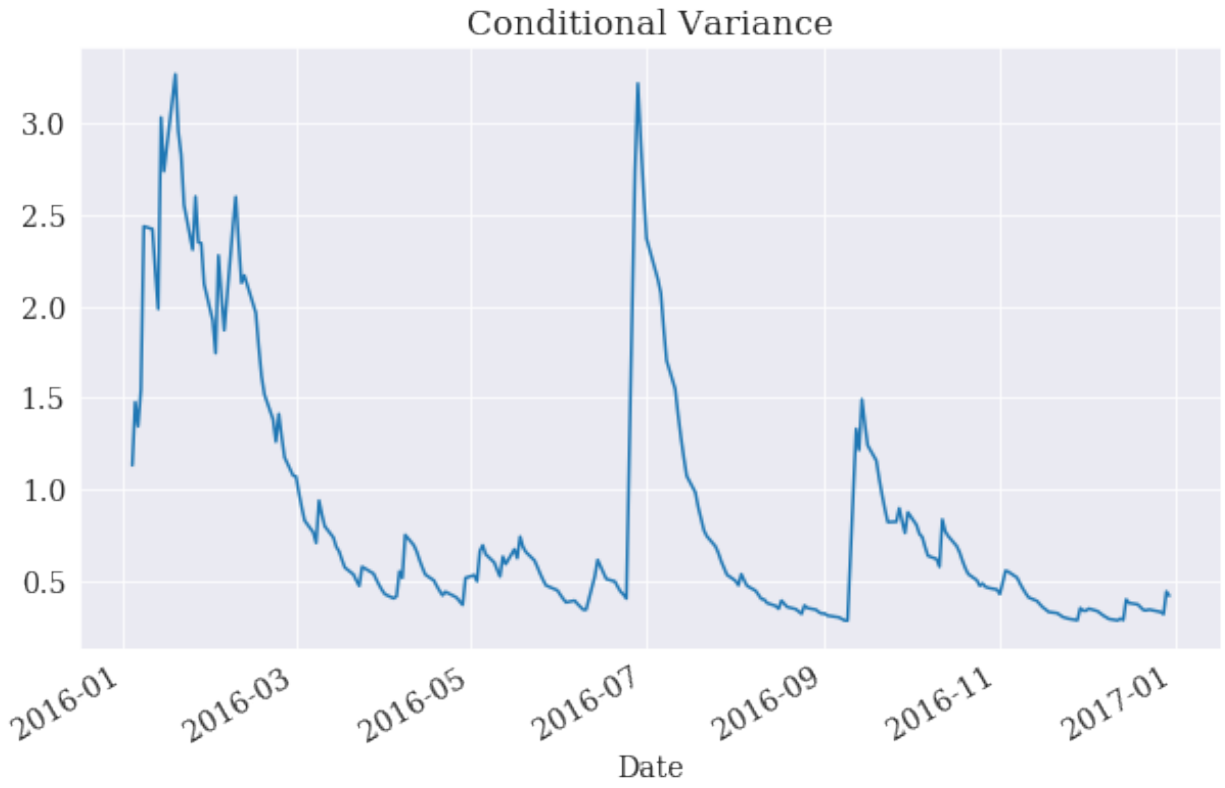
All ARCH-type models have one-step analytical forecasts. Longer horizons only have closed forms for specific models. TARCH models do not have closed-form (analytical) forecasts for horizons larger than 1, and so simulation or bootstrapping is required. Attempting to produce forecasts for horizons larger than 1 using `method='analytical'` results in a `ValueError`.

```
In [11]: # TARCH specification
         am = arch_model(returns, vol='GARCH', power=2.0, p=1, o=1, q=1)
         res = am.fit(update_freq=5)
         forecasts = res.forecast()
         print(forecasts.variance.iloc[-1])

Iteration:      5,   Func. Count:     44,   Neg. LLF: 6037.930348216891
Iteration:     10,   Func. Count:     82,   Neg. LLF: 6034.462050551085
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 6034.461795250723
            Iterations: 12
            Function evaluations: 96
            Gradient evaluations: 12
h.1    0.449483
Name: 2016-12-30 00:00:00, dtype: float64
```

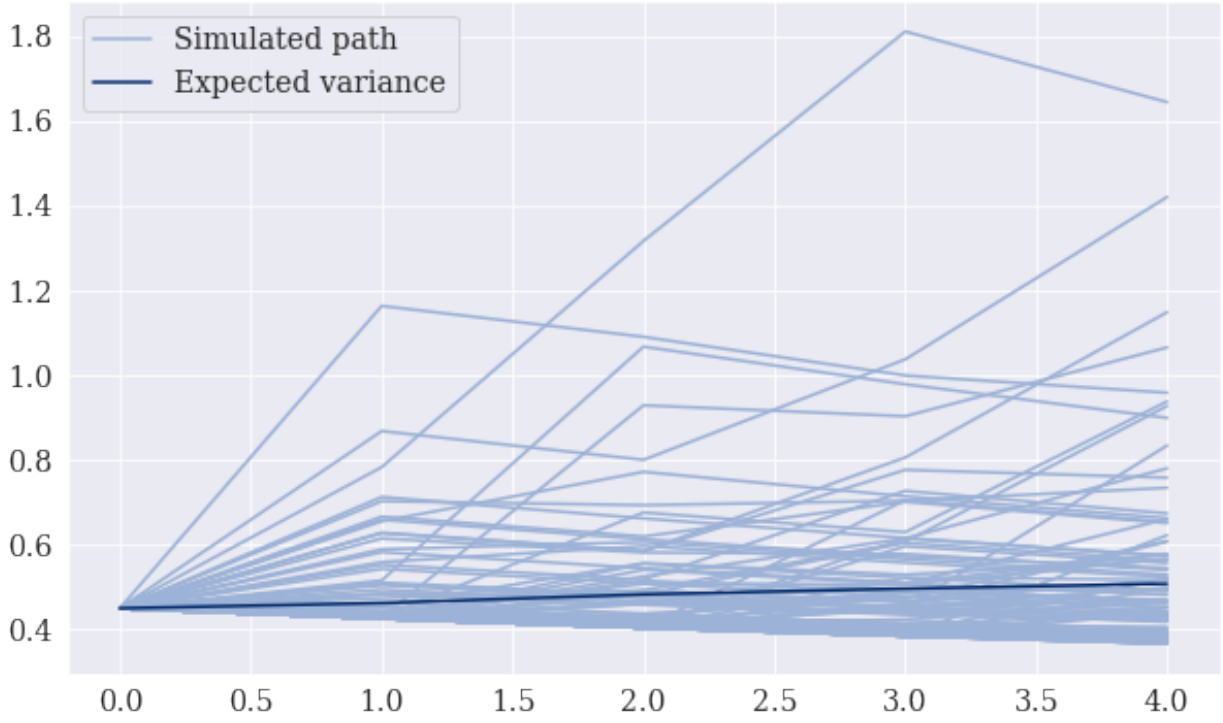
Simulation Forecasts

When using simulation- or bootstrap-based forecasts, an additional attribute of an `ARCHModelForecast` object is meaningful – `simulation`.

```
In [12]: import matplotlib.pyplot as plt
         fig, ax = plt.subplots(1,1)
         subplot = (res.conditional_volatility['2016'] ** 2.0).plot(ax=ax, title='Conditional Varianc
```

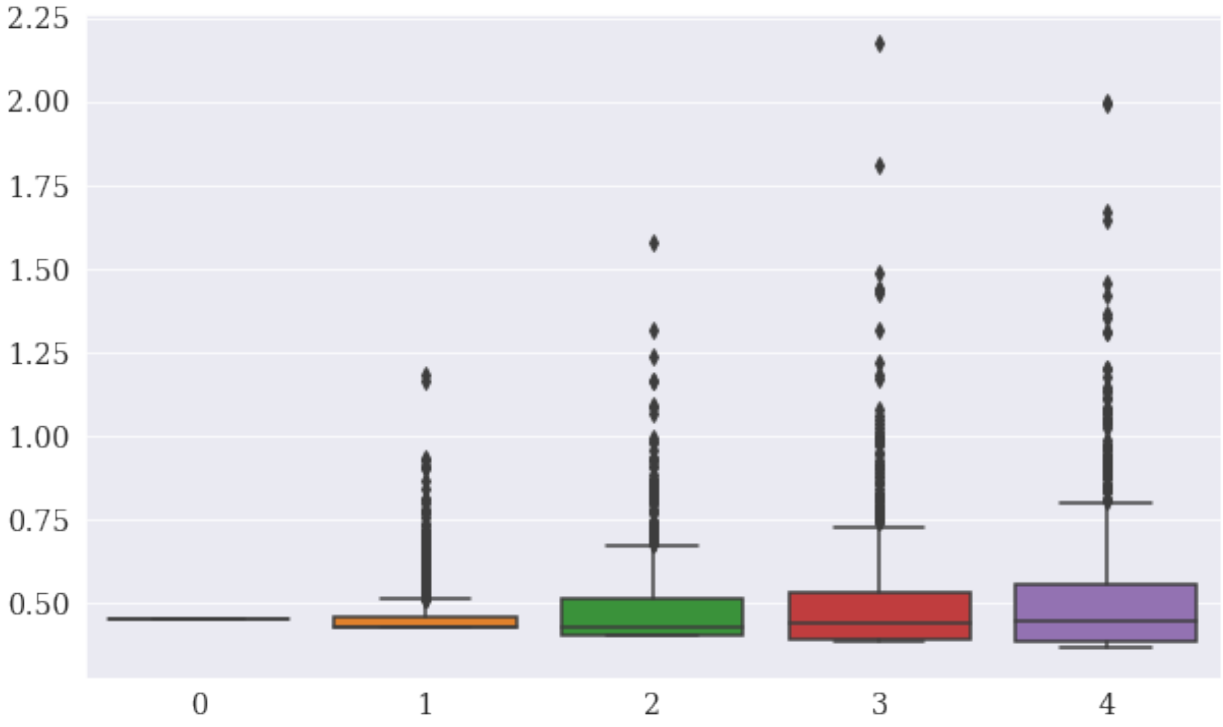



```
In [13]: forecasts = res.forecast(horizon=5, method='simulation')
         sims = forecasts.simulations

         lines = plt.plot(sims.residual_variances[-1,::10].T, color='#9cb2d6')
         lines[0].set_label('Simulated path')
         line = plt.plot(forecasts.variance.iloc[-1].values, color='#002868')
         line[0].set_label('Expected variance')
         legend = plt.legend()
```




```
In [14]: import seaborn as sns
         sns.boxplot(data=sims.variances[-1])
```

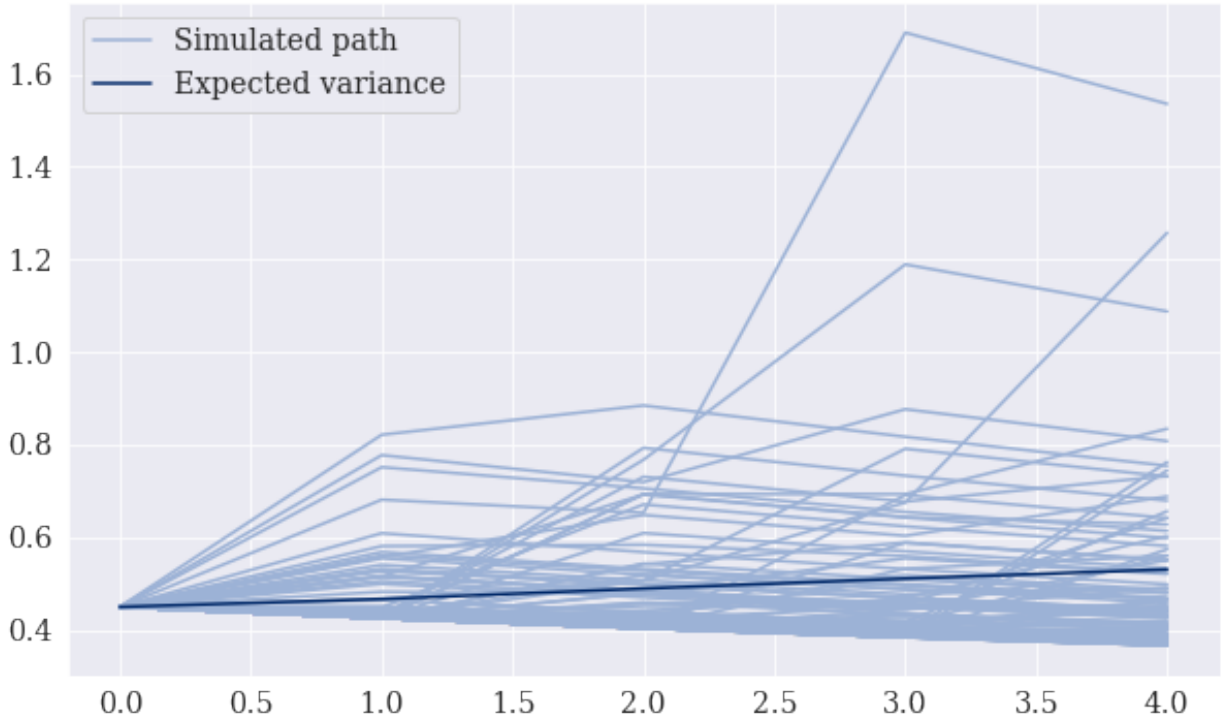
```
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x1ba000f9cf8>
```



Bootstrap Forecasts

Bootstrap-based forecasts are nearly identical to simulation-based forecasts except that the values used to simulate the process are computed from historical data rather than using the assumed distribution of the residuals. Forecasts produced using this method also return an `ARCHModelForecastSimulation` containing information about the simulated paths.

```
In [15]: forecasts = res.forecast(horizon=5, method='bootstrap')
         sims = forecasts.simulations

         lines = plt.plot(sims.residual_variances[-1,::10].T, color='#9cb2d6')
         lines[0].set_label('Simulated path')
         line = plt.plot(forecasts.variance.iloc[-1].values, color='#002868')
         line[0].set_label('Expected variance')
         legend = plt.legend()
```



1.1.5 Volatilty Scenarios

Custom random-number generators can be used to implement scenarios where shock follow a particular pattern. For example, suppose you wanted to find out what would happen if there were 5 days of shocks that were larger than average. In most circumstances, the shocks in a GARCH model have unit variance. This could be changed so that the first 5 shocks have variance 4, or twice the standard deviation.

Another scenario would be to over sample a specific period for the shocks. When using the standard bootstrap method (filtered historical simulation) the shocks are drawn using iid sampling from the history. While this approach is standard and well-grounded, it might be desirable to sample from a specific period. This can be implemented using a custom random number generator. This strategy is precisely how the filtered historical simulation is implemented internally, only where the draws are uniformly sampled from the entire history.

First, some preliminaries

```
In [1]: %matplotlib inline
        from arch.univariate import ConstantMean, GARCH, Normal
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn
        import pandas_datareader as pdr
        seaborn.set_style('darkgrid')
```

```
In [2]: seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
        seaborn.mpl.rcParams['savefig.dpi'] = 90
        seaborn.mpl.rcParams['font.family'] = 'serif'
        seaborn.mpl.rcParams['font.size'] = 14
```

This example makes use of returns from the NASDAQ index. The scenario bootstrap will make use of returns in the run-up to and during the Financial Crisis of 2008.

```
In [3]: nasdaq = pdr.fred.FredReader('NASDAQCOM',start='1-1-2000',end='1-1-2018').read()
        print(nasdaq.head())
```

```
            NASDAQCOM
DATE
2000-01-03    4131.15
2000-01-04    3901.69
2000-01-05    3877.54
2000-01-06    3727.13
2000-01-07    3882.62
```

Next, the returns are computed and the model is constructed. The model is constructed from the building blocks. It is a standard model and could have been (almost) equivalently constructed using

```
mod = arch_model(rets, mean='constant', p=1, o=1, q=1)
```

The one advantage of constructing the model using the components is that the NumPy `RandomState` that is used to simulate from the model can be externally set. This allows the generator seed to be easily set and for the state to reset, if needed.

NOTE: It is always a good idea to scale return by 100 before estimating ARCH-type models. This helps the optimizer converse since the scale of the volatility intercept is much closer to the scale of the other parameters in the model.

```
In [4]: rets = 100 * nasdaq.pct_change().dropna()

        # Build components to set the state for the distribution
        random_state = np.random.RandomState(1)
        dist = Normal(random_state=random_state)
        volatility = GARCH(1, 1, 1)

        mod = ConstantMean(rets, volatility=volatility, distribution=dist)
```

Fitting the model is standard.

```
In [5]: res = mod.fit(disp='off')
        res
```

```
Out[5]:                     Constant Mean - GJR-GARCH Model Results
        ==============================================================================
        Dep. Variable:              NASDAQCOM   R-squared:                      -0.000
        Mean Model:             Constant Mean   Adj. R-squared:                 -0.000
        Vol Model:                  GJR-GARCH   Log-Likelihood:               -7472.03
        Distribution:                  Normal   AIC:                           14954.1
        Method:            Maximum Likelihood   BIC:                           14986.3
                                                No. Observations:                 4695
```

```
Date:                Thu, Sep 27 2018   Df Residuals:                     4690
Time:                        15:26:52   Df Model:                           5
                                Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0297  1.491e-02      1.992  4.643e-02 [4.704e-04,5.893e-02]
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0169  4.601e-03      3.670  2.424e-04  [7.868e-03,2.590e-02]
alpha[1]       0.0000  7.140e-03      0.000      1.000 [-1.399e-02,1.399e-02]
gamma[1]       0.1265  1.744e-02      7.251  4.137e-13    [9.228e-02,  0.161]
beta[1]        0.9256  1.244e-02     74.415      0.000      [  0.901,  0.950]
============================================================================

Covariance estimator: robust
ARCHModelResult, id: 0x1b05cb127f0
```

GJR-GARCH models support analytical forecasts, which is the default. The forecasts are produced for all of 2017 using the estimated model parameters.

```
In [6]: forecasts = res.forecast(start='1-1-2017', horizon=10)
        print(forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
DATE
2017-01-02  0.580045  0.590476  0.600792  0.610993  0.621080  0.631055
2017-01-03  0.553798  0.564522  0.575126  0.585613  0.595982  0.606237
2017-01-04  0.529503  0.540497  0.551369  0.562120  0.572751  0.583264
2017-01-05  0.507015  0.518259  0.529378  0.540374  0.551247  0.561999
2017-01-06  0.486199  0.497675  0.509023  0.520245  0.531342  0.542315

                h.07      h.08      h.09      h.10
DATE
2017-01-02  0.640919  0.650674  0.660320  0.669858
2017-01-03  0.616377  0.626404  0.636320  0.646126
2017-01-04  0.593659  0.603940  0.614105  0.624158
2017-01-05  0.572631  0.583145  0.593542  0.603824
2017-01-06  0.553167  0.563897  0.574509  0.585002
```

All GARCH specification are complete models in the sense that they specify a distribution. This allows simulations to be produced using the assumptions in the model. The `forecast` function can be made to produce simulations using the assumed distribution by setting `method='simulation'`.

These forecasts are similar to the analytical forecasts above. As the number of simulation increases towards ∞, the simulation-based forecasts will converge to the analytical values above.

```
In [7]: sim_forecasts = res.forecast(start='1-1-2017', method='simulation', horizon=10)
        print(sim_forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
DATE
2017-01-02  0.580045  0.590403  0.598096  0.609930  0.616253  0.626874
2017-01-03  0.553798  0.567418  0.581017  0.591527  0.598513  0.603279
2017-01-04  0.529503  0.541442  0.552064  0.563919  0.569974  0.580534
2017-01-05  0.507015  0.516400  0.526907  0.538021  0.545443  0.554512
2017-01-06  0.486199  0.495461  0.505577  0.518608  0.534521  0.545152

                h.07      h.08      h.09      h.10
```

```
DATE
2017-01-02  0.633377  0.640617  0.647898  0.656073
2017-01-03  0.618454  0.627658  0.633192  0.641493
2017-01-04  0.590986  0.600379  0.612958  0.622361
2017-01-05  0.561961  0.576138  0.587727  0.593812
2017-01-06  0.552813  0.565625  0.574218  0.588634
```

Custom Random Generators

`forecast` supports replacing the generator based on the assumed distribution of residuals in the model with any other generator. A shock generator should usually produce unit variance shocks. However, in this example the first 5 shocks generated have variance 2, and the remainder are standard normal. This scenario consists of a period of consistently surprising volatility where the volatility has shifted for some reason.

The forecast variances are much larger and grow faster than those from either method previously illustrated. This reflects the increase in volatility in the first 5 days.

```
In [8]: import numpy as np
        random_state = np.random.RandomState(1)

        def scenario_rng(size):
            shocks = random_state.standard_normal(size)
            shocks[:, :5] *= np.sqrt(2)
            return shocks

        scenario_forecasts = res.forecast(start='1-1-2017', method='simulation', horizon=10, rng=scen
        print(scenario_forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
DATE
2017-01-02  0.580045  0.627008  0.670593  0.726040  0.770887  0.827587
2017-01-03  0.553798  0.605332  0.660221  0.709851  0.755653  0.794870
2017-01-04  0.529503  0.575869  0.622036  0.672072  0.712424  0.767144
2017-01-05  0.507015  0.546602  0.590346  0.637349  0.677891  0.727246
2017-01-06  0.486199  0.523992  0.564631  0.615177  0.677983  0.728223

                h.07      h.08      h.09      h.10
DATE
2017-01-02  0.830443  0.835040  0.839207  0.846161
2017-01-03  0.809932  0.816872  0.818132  0.823916
2017-01-04  0.776076  0.782297  0.794378  0.800691
2017-01-05  0.731660  0.744623  0.754475  0.757505
2017-01-06  0.732785  0.743466  0.750469  0.764813
```

Bootstrap Scenarios

`forecast` supports Filtered Historical Simulation (FHS) using `method='bootstrap'`. This is effectively a simulation method where the simulated shocks are generated using iid sampling from the history of the demeaned and standardized return data. Custom bootstraps are another application of `rng`. Here an object is used to hold the shocks. This object exposes a method (`rng`) the acts like a random number generator, except that it only returns values that were provided in the `shocks` parameter.

The internal implementation of the FHS uses a method almost identical to this where `shocks` contain the entire history.

```
In [9]: class ScenarioBootstrapRNG(object):
            def __init__(self, shocks, random_state):
                self._shocks = np.asarray(shocks)  # 1d
```

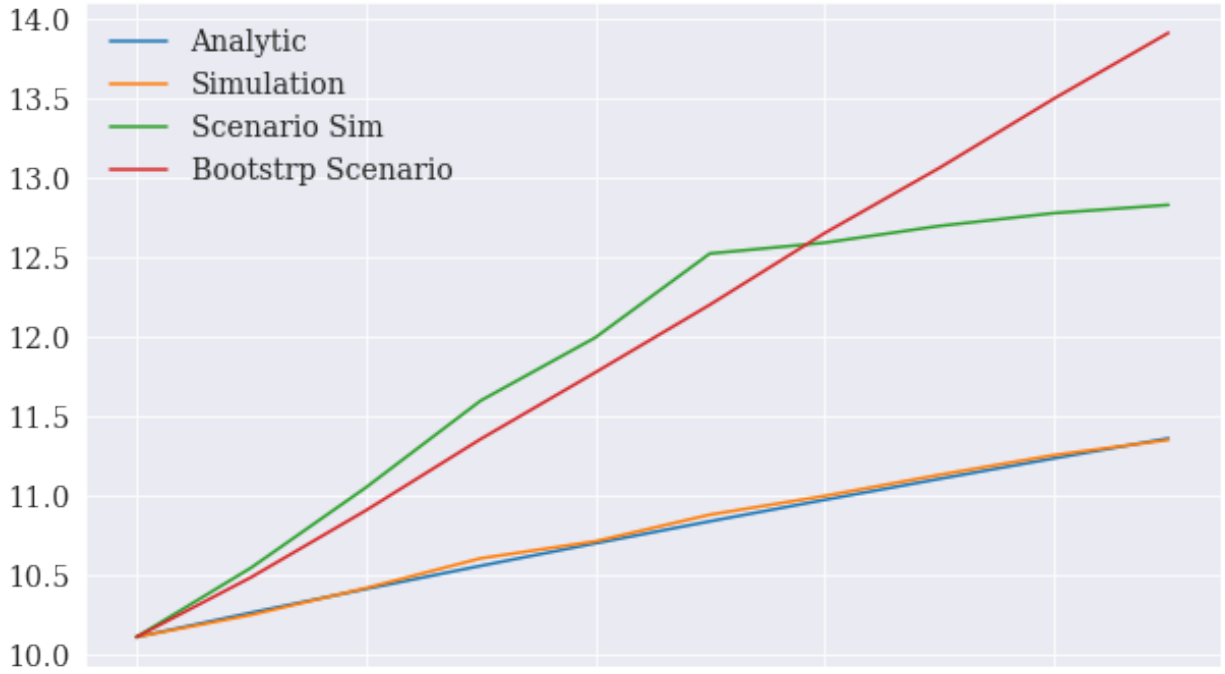
```
                self._rs = random_state
                self.n = shocks.shape[0]

            def rng(self, size):
                idx = self._rs.randint(0, self.n, size=size)
                return self._shocks[idx]

        random_state = np.random.RandomState(1)
        std_shocks = res.resid / res.conditional_volatility
        shocks = std_shocks['2008-08-01':'2008-11-10']
        scenario_bootstrap = ScenarioBootstrapRNG(shocks, random_state)
        bs_forecasts = res.forecast(start='1-1-2017', method='simulation', horizon=10, rng=scenario_b
        print(bs_forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
DATE
2017-01-02  0.580045  0.627731  0.671806  0.710768  0.768906  0.808972
2017-01-03  0.553798  0.601034  0.643942  0.686704  0.733541  0.783090
2017-01-04  0.529503  0.573559  0.615961  0.660616  0.704348  0.755170
2017-01-05  0.507015  0.551867  0.598752  0.643247  0.681701  0.727702
2017-01-06  0.486199  0.520973  0.557449  0.602373  0.648293  0.701349

                h.07      h.08      h.09      h.10
DATE
2017-01-02  0.872246  0.916471  0.973876  1.038470
2017-01-03  0.822505  0.876280  0.930863  0.984509
2017-01-04  0.819018  0.870309  0.931131  0.988081
2017-01-05  0.780660  0.833911  0.883571  0.947947
2017-01-06  0.762154  0.814151  0.863550  0.909780
```

Visualizing the differences

The final forecast values are used to illustrate how these are different. The analytical and standard simulation are virtually identical. The simulated scenario grows rapidly for the first 5 periods and then more slowly. The bootstrap scenario grows quickly and consistently due to the magnitude of the shocks in the financial crisis.

```
In [10]: import pandas as pd
         df = pd.concat([forecasts.residual_variance.iloc[-1],
                         sim_forecasts.residual_variance.iloc[-1],
                         scenario_forecasts.residual_variance.iloc[-1],
                         bs_forecasts.residual_variance.iloc[-1]], 1)
         df.columns = ['Analytic','Simulation','Scenario Sim','Bootstrp Scenario']
         # Plot annualized vol
         subplot = np.sqrt(252 * df).plot(legend=False)
         legend = subplot.legend(frameon=False)
```

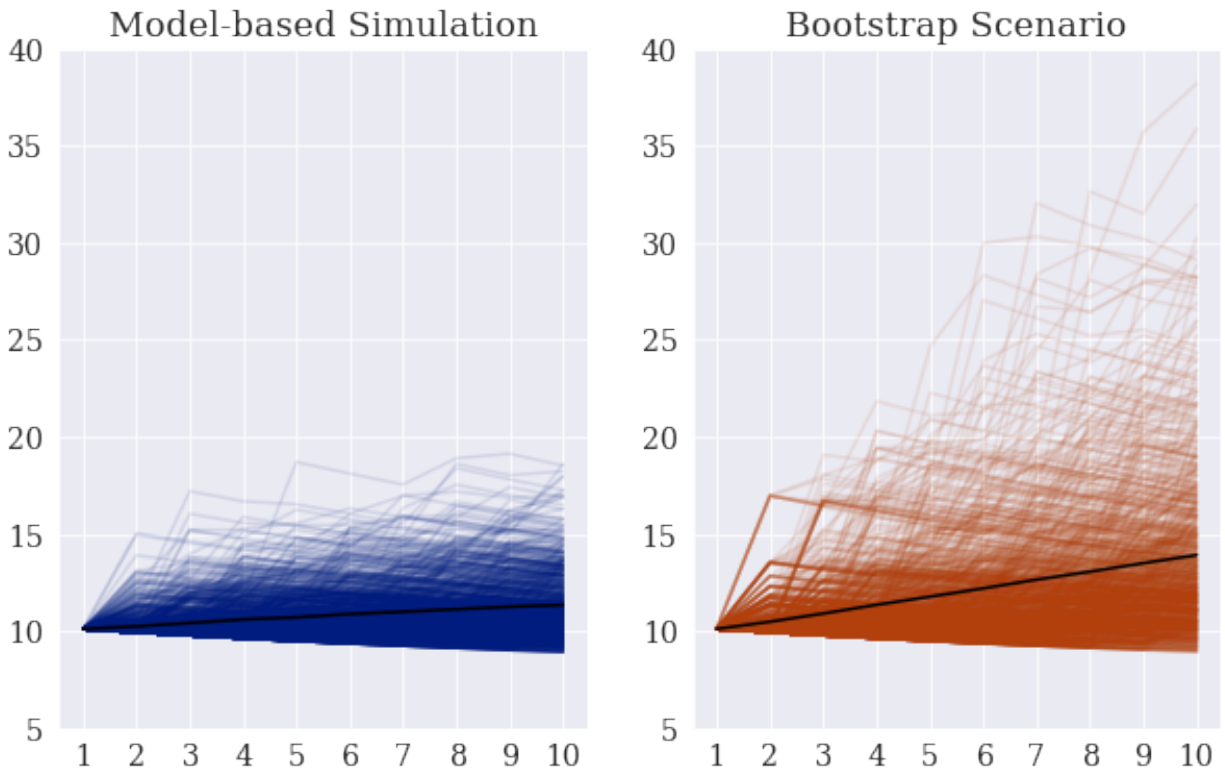



```
In [11]: subplot = np.sqrt(252 * df).plot
```

Comparing the paths

The paths are available on the attribute `simulations`. Plotting the paths shows important differences between the two scenarios beyond the average differences plotted above. Both start at the same point.

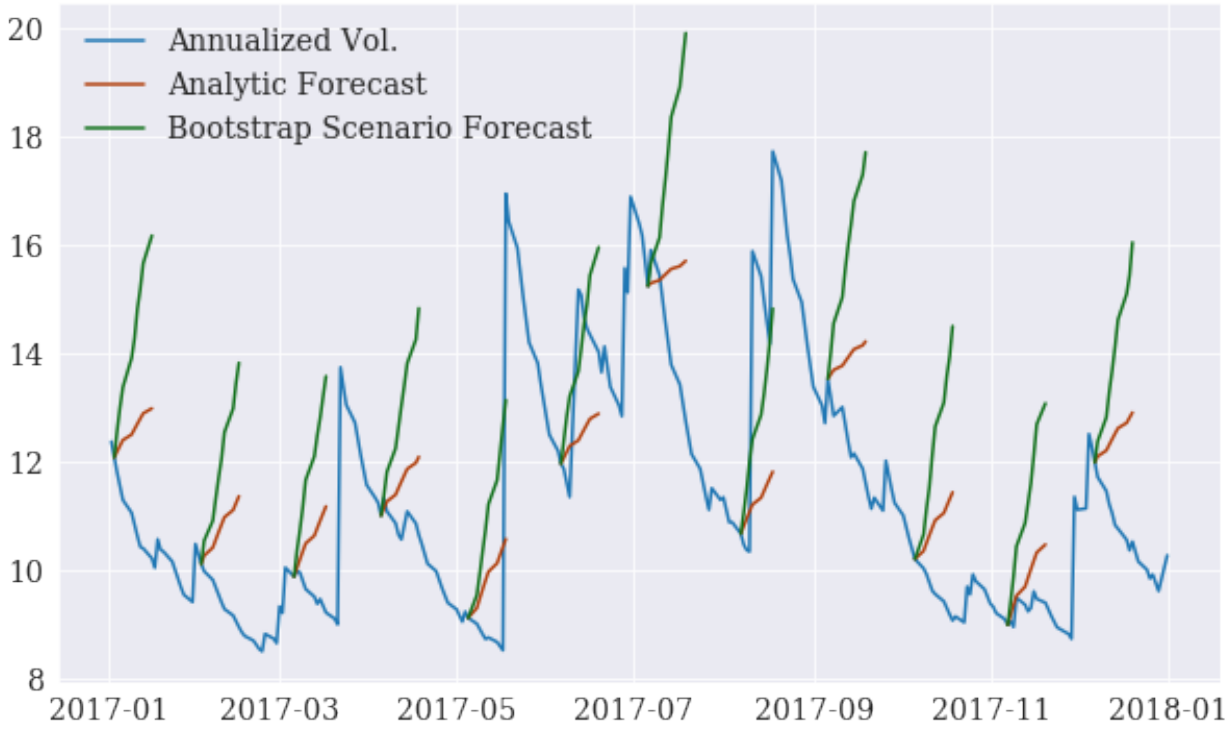
```
In [12]: fig, axes = plt.subplots(1, 2)
         colors = seaborn.color_palette('dark')
         # The paths for the final observation
         sim_paths = sim_forecasts.simulations.residual_variances[-1].T
         bs_paths = bs_forecasts.simulations.residual_variances[-1].T

         # Plot the paths and the mean, set the axis to have the same limit
         axes[0].plot(np.sqrt(252 * sim_paths), color=colors[0], alpha=0.1)
         axes[0].plot(np.sqrt(252 * sim_forecasts.residual_variance.iloc[-1]),
                      color='k', alpha=1)
         axes[0].set_title('Model-based Simulation')
         axes[0].set_xticklabels(np.arange(1, 11))
         axes[0].set_ylim(5, 40)

         axes[1].plot(np.sqrt(252 * bs_paths), color=colors[1], alpha=0.1)
         axes[1].plot(np.sqrt(252 * bs_forecasts.residual_variance.iloc[-1]),
                      color='k', alpha=1)
         axes[1].set_xticklabels(np.arange(1, 11))
         axes[1].set_ylim(5, 40)
         title = axes[1].set_title('Bootstrap Scenario')
```



Comparing across the year

A hedgehog plot is useful for showing the differences between the two forecasting methods across the year, instead of a single day.

```
In [13]: analytic = forecasts.residual_variance.dropna()
         bs = bs_forecasts.residual_variance.dropna()
         fig, ax = plt.subplots(1, 1)
         vol = res.conditional_volatility['2017-1-1':'2018-1-1']
         idx = vol.index
         ax.plot(np.sqrt(252) * vol)
         for i in range(0, len(vol), 22):
             a = analytic.iloc[i]
             b = bs.iloc[i]
             loc = idx.get_loc(a.name)
             new_idx = idx[loc + 1:loc + 11]
             a.index = new_idx
             b.index = new_idx
             ax.plot(np.sqrt(252 * a), color=colors[1])
             ax.plot(np.sqrt(252 * b), color=colors[2])
         labels = ['Annualized Vol.', 'Analytic Forecast',
                   'Bootstrap Scenario Forecast']
         legend = ax.legend(labels, frameon=False)
```



1.1.6 Mean Models

All ARCH models start by specifying a mean model.

No Mean

class arch.univariate.**ZeroMean**(*y=None*, *hold_back=None*, *volatility=None*, *distribution=None*)

Model with zero conditional mean estimation and simulation

Parameters

- **y** (*{ndarray, Series}*) – nobs element vector containing the dependent variable
- **hold_back** (*int*) – Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.
- **volatility** (VolatilityProcess, *optional*) – Volatility process to use in the model
- **distribution** (Distribution, *optional*) – Error distribution to use in the model

Examples

```
>>> import numpy as np
>>> from arch.univariate import ZeroMean
```

(continues on next page)

(continued from previous page)

```
>>> y = np.random.randn(100)
>>> zm = ZeroMean(y)
>>> res = zm.fit()
```

Notes

The zero mean model is described by

$$y_t = \epsilon_t$$

fit(*update_freq=1*, *disp='final'*, *starting_values=None*, *cov_type='robust'*, *show_warning=True*, *first_obs=None*, *last_obs=None*, *tol=None*, *options=None*, *backcast=None*)
Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **update_freq** (*int, optional*) – Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.
- **disp** (*str*) – Either 'final' to print optimization result or 'off' to display nothing
- **starting_values** (*ndarray, optional*) – Array of starting values to use. If not provided, starting values are constructed by the model components.
- **cov_type** (*str, optional*) – Estimation method of parameter covariance. Supported options are 'robust', which does not assume the Information Matrix Equality holds and 'classic' which does. In the ARCH literature, 'robust' corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning** (*bool, optional*) – Flag indicating whether convergence warnings should be shown.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when estimating model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when estimating model
- **tol** (*float, optional*) – Tolerance for termination.
- **options** (*dict, optional*) – Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.
- **backcast** (*float, optional*) – Value to use as backcast. Should be measure σ_0^2 since model-specific non-linear transformations are applied to value before computing the variance recusions.

Returns **results** – Object containing model results

Return type *ARCHModelResult*

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

fix(*params*, *first_obs=None*, *last_obs=None*)
Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

- **params** (*{ndarray, Series}*) – User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when fixing model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when fixing model

Returns **results** – Object containing model results

Return type *ARCHModelFixedResult*

Notes

Parameters are not checked against model-specific constraints.

forecast(*params*, *horizon=1*, *start=None*, *align='origin'*, *method='analytic'*, *simulations=1000*, *rng=None*)
Construct forecasts from estimated model

Parameters

- **params** (*{ndarray, Series}, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'.
- **align** (*str, optional*) – Either 'origin' or 'target'. When set of 'origin', the t-th row of forecasts contains the forecasts for t+1, t+2, ..., t+h. When set to 'target', the t-th row contains the 1-step ahead forecast from time t-1, the 2 step from time t-2, ..., and the h-step from time t-h. 'target' simplified computing forecast errors since the realization and h-step forecast are aligned.
- **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** (*callable, optional*) – Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

Returns **forecasts** – t by h data frame containing the forecasts. The alignment of the forecasts is controlled by *align*.

Return type *ARCHModelForecast*

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],
↪vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

Notes

The most basic 1-step ahead forecast will return a vector with the same length as the original data, where the t-th value will be the time-t forecast for time t + 1. When the horizon is > 1, and when using the default value for *align*, the forecast value in position [t, h] is the time-t, h+1 step ahead forecast.

If model contains exogenous variables (model.x is not None), then only 1-step ahead forecasts are available. Using horizon > 1 will produce a warning and all columns, except the first, will be nan-filled.

If *align* is 'origin', forecast[t,h] contains the forecast made using y[:t] (that is, up to but not including t) for horizon h + 1. For example, y[100,2] contains the 3-step ahead forecast using the first 100 data points, which will correspond to the realization y[100 + 2]. If *align* is 'target', then the same forecast is in location [102, 2], so that it is aligned with the observation to use when evaluating, but still in the same column.

resids (*params*, *y=None*, *regressors=None*)
Compute model residuals

Parameters

- **params** (*ndarray*) – Model parameters
- **y** (*ndarray, optional*) – Alternative values to use when computing model residuals
- **regressors** (*ndarray, optional*) – Alternative regressor values to use when computing model residuals

Returns **resids** – Model residuals

Return type ndarray

simulate (*params*, *nobs*, *burn=500*, *initial_value=None*, *x=None*, *initial_value_vol=None*)
Simulated data from a zero mean model

Parameters

- **params** (*{ndarray, DataFrame}*) – Parameters to use when simulating the model. Parameter order is [volatility distribution]. There are no mean parameters.
- **nobs** (*int*) – Length of series to simulate
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values.
- **initial_value** (*None*) – This value is not used.
- **x** (*None*) – This value is not used.

- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process.

Returns simulated_data – DataFrame with columns data containing the simulated values, volatility, containing the conditional volatility and errors containing the errors used in the simulation

Return type DataFrame

Examples

Basic data simulation with no mean and constant volatility

```
>>> from arch.univariate import ZeroMean
>>> zm = ZeroMean()
>>> sim_data = zm.simulate([1.0], 1000)
```

Simulating data with a non-trivial volatility process

```
>>> from arch.univariate import GARCH
>>> zm.volatility = GARCH(p=1, o=1, q=1)
>>> sim_data = zm.simulate([0.05, 0.1, 0.1, 0.8], 300)
```

Constant Mean

class arch.univariate.**ConstantMean**(*y=None, hold_back=None, volatility=None, distribution=None*)

Constant mean model estimation and simulation.

Parameters

- **y** (*{ndarray, Series}*) – nobs element vector containing the dependent variable
- **hold_back** (*int*) – Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.
- **volatility** (VolatilityProcess, *optional*) – Volatility process to use in the model
- **distribution** (Distribution, *optional*) – Error distribution to use in the model

Examples

```
>>> import numpy as np
>>> from arch.univariate import ConstantMean
>>> y = np.random.randn(100)
>>> cm = ConstantMean(y)
>>> res = cm.fit()
```

Notes

The constant mean model is described by

$$y_t = \mu + \epsilon_t$$

fit(*update_freq=1*, *disp='final'*, *starting_values=None*, *cov_type='robust'*, *show_warning=True*, *first_obs=None*, *last_obs=None*, *tol=None*, *options=None*, *backcast=None*)
Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **update_freq** (*int, optional*) – Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.
- **disp** (*str*) – Either 'final' to print optimization result or 'off' to display nothing
- **starting_values** (*ndarray, optional*) – Array of starting values to use. If not provided, starting values are constructed by the model components.
- **cov_type** (*str, optional*) – Estimation method of parameter covariance. Supported options are 'robust', which does not assume the Information Matrix Equality holds and 'classic' which does. In the ARCH literature, 'robust' corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning** (*bool, optional*) – Flag indicating whether convergence warnings should be shown.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when estimating model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when estimating model
- **tol** (*float, optional*) – Tolerance for termination.
- **options** (*dict, optional*) – Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.
- **backcast** (*float, optional*) – Value to use as backcast. Should be measure σ_0^2 since model-specific non-linear transformations are applied to value before computing the variance recusions.

Returns **results** – Object containing model results

Return type *ARCHModelResult*

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

forecast(*params*, *horizon=1*, *start=None*, *align='origin'*, *method='analytic'*, *simulations=1000*, *rng=None*)
Construct forecasts from estimated model

Parameters

- **params** (*{ndarray, Series}, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'.
- **align** (*str, optional*) – Either 'origin' or 'target'. When set of 'origin', the t-th row of forecasts contains the forecasts for t+1, t+2, ..., t+h. When set to 'target', the t-th row contains the 1-step ahead forecast from time t-1, the 2 step from time t-2, ..., and the h-step from time t-h. 'target' simplified computing forecast errors since the realization and h-step forecast are aligned.
- **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** (*callable, optional*) – Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

Returns **forecasts** – t by h data frame containing the forecasts. The alignment of the forecasts is controlled by *align*.

Return type *ARCHModelForecast*

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],
↪vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

Notes

The most basic 1-step ahead forecast will return a vector with the same length as the original data, where the t-th value will be the time-t forecast for time t + 1. When the horizon is > 1, and when using the default value for *align*, the forecast value in position [t, h] is the time-t, h+1 step ahead forecast.

If model contains exogenous variables (model.x is not None), then only 1-step ahead forecasts are available. Using horizon > 1 will produce a warning and all columns, except the first, will be nan-filled.

If *align* is 'origin', forecast[t,h] contains the forecast made using y[:t] (that is, up to but not including t) for horizon h + 1. For example, y[100,2] contains the 3-step ahead forecast using the first 100 data points, which will correspond to the realization y[100 + 2]. If *align* is 'target', then the same forecast is in location [102, 2], so that it is aligned with the observation to use when evaluating, but still in the same column.

resids (*params*, *y=None*, *regressors=None*)
Compute model residuals

Parameters

- **params** (*ndarray*) – Model parameters
- **y** (*ndarray, optional*) – Alternative values to use when computing model residuals
- **regressors** (*ndarray, optional*) – Alternative regressor values to use when computing model residuals

Returns **resids** – Model residuals

Return type ndarray

simulate (*params*, *nobs*, *burn=500*, *initial_value=None*, *x=None*, *initial_value_vol=None*)
Simulated data from a constant mean model

Parameters

- **params** (*ndarray*) – Parameters to use when simulating the model. Parameter order is [mean volatility distribution]. There is one parameter in the mean model, mu.
- **nobs** (*int*) – Length of series to simulate
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values.
- **initial_value** (*None*) – This value is not used.
- **x** (*None*) – This value is not used.
- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process.

Returns **simulated_data** – DataFrame with columns data containing the simulated values, volatility, containing the conditional volatility and errors containing the errors used in the simulation

Return type DataFrame

Examples

Basic data simulation with a constant mean and volatility

```
>>> import numpy as np
>>> from arch.univariate import ConstantMean, GARCH
>>> cm = ConstantMean()
>>> cm.volatility = GARCH()
>>> cm_params = np.array([1])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((cm_params, garch_params))
>>> sim_data = cm.simulate(params, 1000)
```

Autoregressions

class arch.univariate.**ARX**(*y=None*, *x=None*, *lags=None*, *constant=True*, *hold_back=None*, *volatility=None*, *distribution=None*)

Autoregressive model with optional exogenous regressors estimation and simulation

Parameters

- **y** (*{ndarray, Series}*) – nobs element vector containing the dependent variable
- **x** (*{ndarray, DataFrame}, optional*) – nobs by k element array containing exogenous regressors
- **lags** (*scalar, 1-d array, optional*) – Description of lag structure of the HAR. Scalar included all lags between 1 and the value. A 1-d array includes the AR lags lags[0], lags[1], ...
- **constant** (*bool, optional*) – Flag whether the model should include a constant
- **hold_back** (*int*) – Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.

Examples

```
>>> import numpy as np
>>> from arch.univariate import ARX
>>> y = np.random.randn(100)
>>> arx = ARX(y, lags=[1, 5, 22])
>>> res = arx.fit()
```

Estimating an AR with GARCH(1,1) errors >>> from arch.univariate import GARCH >>> arx.volatility = GARCH() >>> res = arx.fit(update_freq=0, disp='off')

Notes

The AR-X model is described by

$$y_t = \mu + \sum_{i=1}^{p} \phi_{L_i} y_{t-L_i} + \gamma' x_t + \epsilon_t$$

fit(*update_freq=1*, *disp='final'*, *starting_values=None*, *cov_type='robust'*, *show_warning=True*, *first_obs=None*, *last_obs=None*, *tol=None*, *options=None*, *backcast=None*)

Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **update_freq** (*int, optional*) – Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.
- **disp** (*str*) – Either 'final' to print optimization result or 'off' to display nothing
- **starting_values** (*ndarray, optional*) – Array of starting values to use. If not provided, starting values are constructed by the model components.

- **cov_type** (*str, optional*) – Estimation method of parameter covariance. Supported options are 'robust', which does not assume the Information Matrix Equality holds and 'classic' which does. In the ARCH literature, 'robust' corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning** (*bool, optional*) – Flag indicating whether convergence warnings should be shown.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when estimating model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when estimating model
- **tol** (*float, optional*) – Tolerance for termination.
- **options** (*dict, optional*) – Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.
- **backcast** (*float, optional*) – Value to use as backcast. Should be measure σ_0^2 since model-specific non-linear transformations are applied to value before computing the variance recusions.

Returns **results** – Object containing model results

Return type *ARCHModelResult*

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

fix(*params*, *first_obs=None*, *last_obs=None*)
Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

- **params** (*{ndarray, Series}*) – User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when fixing model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when fixing model

Returns **results** – Object containing model results

Return type *ARCHModelFixedResult*

Notes

Parameters are not checked against model-specific constraints.

forecast(*params*, *horizon=1*, *start=None*, *align='origin'*, *method='analytic'*, *simulations=1000*, *rng=None*)
Construct forecasts from estimated model

Parameters

- **params** (*{ndarray, Series}, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'.
- **align** (*str, optional*) – Either 'origin' or 'target'. When set of 'origin', the t-th row of forecasts contains the forecasts for t+1, t+2, ..., t+h. When set to 'target', the t-th row contains the 1-step ahead forecast from time t-1, the 2 step from time t-2, ..., and the h-step from time t-h. 'target' simplified computing forecast errors since the realization and h-step forecast are aligned.
- **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** (*callable, optional*) – Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

Returns **forecasts** – t by h data frame containing the forecasts. The alignment of the forecasts is controlled by *align*.

Return type *ARCHModelForecast*

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],
↪vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

Notes

The most basic 1-step ahead forecast will return a vector with the same length as the original data, where the t-th value will be the time-t forecast for time t + 1. When the horizon is > 1, and when using the default value for *align*, the forecast value in position [t, h] is the time-t, h+1 step ahead forecast.

If model contains exogenous variables (model.x is not None), then only 1-step ahead forecasts are available. Using horizon > 1 will produce a warning and all columns, except the first, will be nan-filled.

If *align* is 'origin', forecast[t,h] contains the forecast made using y[:t] (that is, up to but not including t) for horizon h + 1. For example, y[100,2] contains the 3-step ahead forecast using the first 100 data points, which will correspond to the realization y[100 + 2]. If *align* is 'target', then the same forecast is in location [102, 2], so that it is aligned with the observation to use when evaluating, but still in the same column.

resids (*params*, *y=None*, *regressors=None*)
Compute model residuals

Parameters

- **params** (*ndarray*) – Model parameters
- **y** (*ndarray, optional*) – Alternative values to use when computing model residuals
- **regressors** (*ndarray, optional*) – Alternative regressor values to use when computing model residuals

Returns **resids** – Model residuals

Return type ndarray

simulate (*params*, *nobs*, *burn=500*, *initial_value=None*, *x=None*, *initial_value_vol=None*)
Simulates data from a linear regression, AR or HAR models

Parameters

- **params** (*ndarray*) – Parameters to use when simulating the model. Parameter order is [mean volatility distribution] where the parameters of the mean model are ordered [constant lag[0] lag[1] ... lag[p] ex[0] ... ex[k-1]] where lag[j] indicates the coefficient on the jth lag in the model and ex[j] is the coefficient on the jth exogenous variable.
- **nobs** (*int*) – Length of series to simulate
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values.
- **initial_value** (*{ndarray, float}, optional*) – Either a scalar value or *max(lags)* array set of initial values to use when initializing the model. If omitted, 0.0 is used.
- **x** (*{ndarray, DataFrame}, optional*) – nobs + burn by k array of exogenous variables to include in the simulation.
- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process.

Returns **simulated_data** – DataFrame with columns data containing the simulated values, volatility, containing the conditional volatility and errors containing the errors used in the simulation

Return type DataFrame

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX, GARCH
>>> harx = HARX(lags=[1, 5, 22])
>>> harx.volatility = GARCH()
>>> harx_params = np.array([1, 0.2, 0.3, 0.4])
>>> garch_params = np.array([0.01, 0.07, 0.92])
```

(continues on next page)

(continued from previous page)

```
>>> params = np.concatenate((harx_params, garch_params))
>>> sim_data = harx.simulate(params, 1000)
```

Simulating models with exogenous regressors requires the regressors to have nobs plus burn data points

```
>>> nobs = 100
>>> burn = 200
>>> x = np.random.randn(nobs + burn, 2)
>>> x_params = np.array([1.0, 2.0])
>>> params = np.concatenate((harx_params, x_params, garch_params))
>>> sim_data = harx.simulate(params, nobs=nobs, burn=burn, x=x)
```

Heterogeneous Autoregressions

class arch.univariate.**HARX**(*y=None*, *x=None*, *lags=None*, *constant=True*, *use_rotated=False*, *hold_back=None*, *volatility=None*, *distribution=None*)

Heterogeneous Autoregression (HAR), with optional exogenous regressors, model estimation and simulation

Parameters

- **y** (*{ndarray, Series}*) – nobs element vector containing the dependent variable
- **x** (*{ndarray, DataFrame}, optional*) – nobs by k element array containing exogenous regressors
- **lags** (*{scalar, ndarray}, optional*) – Description of lag structure of the HAR. Scalar included all lags between 1 and the value. A 1-d array includes the HAR lags 1:lags[0], 1:lags[1], ... A 2-d array includes the HAR lags of the form lags[0,j]:lags[1,j] for all columns of lags.
- **constant** (*bool, optional*) – Flag whether the model should include a constant
- **use_rotated** (*bool, optional*) – Flag indicating to use the alternative rotated form of the HAR where HAR lags do not overlap
- **hold_back** (*int*) – Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.
- **volatility** (*VolatilityProcess, optional*) – Volatility process to use in the model
- **distribution** (*Distribution, optional*) – Error distribution to use in the model

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX
>>> y = np.random.randn(100)
>>> harx = HARX(y, lags=[1, 5, 22])
>>> res = harx.fit()
```

```
>>> from pandas import Series, date_range
>>> index = date_range('2000-01-01', freq='M', periods=y.shape[0])
>>> y = Series(y, name='y', index=index)
>>> har = HARX(y, lags=[1, 6], hold_back=10)
```

Notes

The HAR-X model is described by

$$y_t = \mu + \sum_{i=1}^{p} \phi_{L_i} \bar{y}_{t-L_{i,0}:L_{i,1}} + \gamma' x_t + \epsilon_t$$

where $\bar{y}_{t-L_{i,0}:L_{i,1}}$ is the average value of y_t between $t - L_{i,0}$ and $t - L_{i,1}$.

fit(*update_freq=1*, *disp='final'*, *starting_values=None*, *cov_type='robust'*, *show_warning=True*, *first_obs=None*, *last_obs=None*, *tol=None*, *options=None*, *backcast=None*)
Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **update_freq** (*int, optional*) – Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.
- **disp** (*str*) – Either 'final' to print optimization result or 'off' to display nothing
- **starting_values** (*ndarray, optional*) – Array of starting values to use. If not provided, starting values are constructed by the model components.
- **cov_type** (*str, optional*) – Estimation method of parameter covariance. Supported options are 'robust', which does not assume the Information Matrix Equality holds and 'classic' which does. In the ARCH literature, 'robust' corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning** (*bool, optional*) – Flag indicating whether convergence warnings should be shown.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when estimating model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when estimating model
- **tol** (*float, optional*) – Tolerance for termination.
- **options** (*dict, optional*) – Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.
- **backcast** (*float, optional*) – Value to use as backcast. Should be measure σ_0^2 since model-specific non-linear transformations are applied to value before computing the variance recusions.

Returns **results** – Object containing model results

Return type *ARCHModelResult*

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

fix(*params*, *first_obs=None*, *last_obs=None*)
Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

- **params** (*{ndarray, Series}*) – User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when fixing model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when fixing model

Returns **results** – Object containing model results

Return type *ARCHModelFixedResult*

Notes

Parameters are not checked against model-specific constraints.

forecast(*params*, *horizon=1*, *start=None*, *align='origin'*, *method='analytic'*, *simulations=1000*, *rng=None*)
Construct forecasts from estimated model

Parameters

- **params** (*{ndarray, Series}, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in '1945-01-01'.
- **align** (*str, optional*) – Either 'origin' or 'target'. When set of 'origin', the t-th row of forecasts contains the forecasts for t+1, t+2, ..., t+h. When set to 'target', the t-th row contains the 1-step ahead forecast from time t-1, the 2 step from time t-2, ..., and the h-step from time t-h. 'target' simplified computing forecast errors since the realization and h-step forecast are aligned.
- **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the 'analytic' method for horizons > 1.
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** (*callable, optional*) – Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

Returns **forecasts** – t by h data frame containing the forecasts. The alignment of the forecasts is controlled by *align*.

Return type *ARCHModelForecast*

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],
↪vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

Notes

The most basic 1-step ahead forecast will return a vector with the same length as the original data, where the t-th value will be the time-t forecast for time t + 1. When the horizon is > 1, and when using the default value for *align*, the forecast value in position [t, h] is the time-t, h+1 step ahead forecast.

If model contains exogenous variables (model.x is not None), then only 1-step ahead forecasts are available. Using horizon > 1 will produce a warning and all columns, except the first, will be nan-filled.

If *align* is 'origin', forecast[t,h] contains the forecast made using y[:t] (that is, up to but not including t) for horizon h + 1. For example, y[100,2] contains the 3-step ahead forecast using the first 100 data points, which will correspond to the realization y[100 + 2]. If *align* is 'target', then the same forecast is in location [102, 2], so that it is aligned with the observation to use when evaluating, but still in the same column.

resids (*params*, *y=None*, *regressors=None*)
Compute model residuals

Parameters

- **params** (*ndarray*) – Model parameters
- **y** (*ndarray, optional*) – Alternative values to use when computing model residuals
- **regressors** (*ndarray, optional*) – Alternative regressor values to use when computing model residuals

Returns **resids** – Model residuals

Return type ndarray

simulate (*params*, *nobs*, *burn=500*, *initial_value=None*, *x=None*, *initial_value_vol=None*)
Simulates data from a linear regression, AR or HAR models

Parameters

- **params** (*ndarray*) – Parameters to use when simulating the model. Parameter order is [mean volatility distribution] where the parameters of the mean model are ordered [constant lag[0] lag[1] … lag[p] ex[0] … ex[k-1]] where lag[j] indicates the coefficient on the jth lag in the model and ex[j] is the coefficient on the jth exogenous variable.
- **nobs** (*int*) – Length of series to simulate
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values.

- **initial_value** (*{ndarray, float}, optional*) – Either a scalar value or *max(lags)* array set of initial values to use when initializing the model. If omitted, 0.0 is used.
- **x** (*{ndarray, DataFrame}, optional*) – nobs + burn by k array of exogenous variables to include in the simulation.
- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process.

Returns simulated_data – DataFrame with columns data containing the simulated values, volatility, containing the conditional volatility and errors containing the errors used in the simulation

Return type DataFrame

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX, GARCH
>>> harx = HARX(lags=[1, 5, 22])
>>> harx.volatility = GARCH()
>>> harx_params = np.array([1, 0.2, 0.3, 0.4])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((harx_params, garch_params))
>>> sim_data = harx.simulate(params, 1000)
```

Simulating models with exogenous regressors requires the regressors to have nobs plus burn data points

```
>>> nobs = 100
>>> burn = 200
>>> x = np.random.randn(nobs + burn, 2)
>>> x_params = np.array([1.0, 2.0])
>>> params = np.concatenate((harx_params, x_params, garch_params))
>>> sim_data = harx.simulate(params, nobs=nobs, burn=burn, x=x)
```

Least Squares

class arch.univariate.**LS**(*y=None*, *x=None*, *constant=True*, *hold_back=None*)

Least squares model estimation and simulation

Parameters

- **y** (*{ndarray, DataFrame}, optional*) – nobs element vector containing the dependent variable
- **y** – nobs by k element array containing exogenous regressors
- **constant** (*bool, optional*) – Flag whether the model should include a constant
- **hold_back** (*int*) – Number of observations at the start of the sample to exclude when estimating model parameters. Used when comparing models with different lag lengths to estimate on the common sample.

Examples

```
>>> import numpy as np
>>> from arch.univariate import LS
>>> y = np.random.randn(100)
>>> x = np.random.randn(100,2)
>>> ls = LS(y, x)
>>> res = ls.fit()
```

Notes

The LS model is described by

$$y_t = \mu + \gamma' x_t + \epsilon_t$$

fit (*update_freq=1*, *disp='final'*, *starting_values=None*, *cov_type='robust'*, *show_warning=True*, *first_obs=None*, *last_obs=None*, *tol=None*, *options=None*, *backcast=None*)
Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **update_freq** (*int, optional*) – Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.
- **disp** (*str*) – Either 'final' to print optimization result or 'off' to display nothing
- **starting_values** (*ndarray, optional*) – Array of starting values to use. If not provided, starting values are constructed by the model components.
- **cov_type** (*str, optional*) – Estimation method of parameter covariance. Supported options are 'robust', which does not assume the Information Matrix Equality holds and 'classic' which does. In the ARCH literature, 'robust' corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning** (*bool, optional*) – Flag indicating whether convergence warnings should be shown.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when estimating model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when estimating model
- **tol** (*float, optional*) – Tolerance for termination.
- **options** (*dict, optional*) – Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.
- **backcast** (*float, optional*) – Value to use as backcast. Should be measure σ_0^2 since model-specific non-linear transformations are applied to value before computing the variance recusions.

Returns **results** – Object containing model results

Return type *ARCHModelResult*

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

fix(*params*, *first_obs=None*, *last_obs=None*)
Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

- **params** (*{ndarray, Series}*) – User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.
- **first_obs** (*{int, str, datetime, Timestamp}*) – First observation to use when fixing model
- **last_obs** (*{int, str, datetime, Timestamp}*) – Last observation to use when fixing model

Returns **results** – Object containing model results

Return type *ARCHModelFixedResult*

Notes

Parameters are not checked against model-specific constraints.

resids(*params*, *y=None*, *regressors=None*)
Compute model residuals

Parameters

- **params** (*ndarray*) – Model parameters
- **y** (*ndarray, optional*) – Alternative values to use when computing model residuals
- **regressors** (*ndarray, optional*) – Alternative regressor values to use when computing model residuals

Returns **resids** – Model residuals

Return type ndarray

simulate(*params*, *nobs*, *burn=500*, *initial_value=None*, *x=None*, *initial_value_vol=None*)
Simulates data from a linear regression, AR or HAR models

Parameters

- **params** (*ndarray*) – Parameters to use when simulating the model. Parameter order is [mean volatility distribution] where the parameters of the mean model are ordered [constant lag[0] lag[1] … lag[p] ex[0] … ex[k-1]] where lag[j] indicates the coefficient on the jth lag in the model and ex[j] is the coefficient on the jth exogenous variable.
- **nobs** (*int*) – Length of series to simulate
- **burn** (*int, optional*) – Number of values to simulate to initialize the model and remove dependence on initial values.
- **initial_value** (*{ndarray, float}, optional*) – Either a scalar value or *max(lags)* array set of initial values to use when initializing the model. If omitted, 0.0 is used.

- **x** (*{ndarray, DataFrame}, optional*) – nobs + burn by k array of exogenous variables to include in the simulation.
- **initial_value_vol** (*{ndarray, float}, optional*) – An array or scalar to use when initializing the volatility process.

Returns simulated_data – DataFrame with columns data containing the simulated values, volatility, containing the conditional volatility and errors containing the errors used in the simulation

Return type DataFrame

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX, GARCH
>>> harx = HARX(lags=[1, 5, 22])
>>> harx.volatility = GARCH()
>>> harx_params = np.array([1, 0.2, 0.3, 0.4])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((harx_params, garch_params))
>>> sim_data = harx.simulate(params, 1000)
```

Simulating models with exogenous regressors requires the regressors to have nobs plus burn data points

```
>>> nobs = 100
>>> burn = 200
>>> x = np.random.randn(nobs + burn, 2)
>>> x_params = np.array([1.0, 2.0])
>>> params = np.concatenate((harx_params, x_params, garch_params))
>>> sim_data = harx.simulate(params, nobs=nobs, burn=burn, x=x)
```

Writing New Mean Models

All mean models must inherit from :class:ARCHModel and provide all public methods. There are two optional private methods that should be provided if applicable.

class arch.univariate.base.**ARCHModel**(*y=None*, *volatility=None*, *distribution=None*, *hold_back=None*)

Abstract base class for mean models in ARCH processes. Specifies the conditional mean process.

All public methods that raise NotImplementedError should be overridden by any subclass. Private methods that raise NotImplementedError are optional to override but recommended where applicable.

1.1.7 Volatility Processes

A volatility process is added to a mean model to capture time-varying volatility.

Constant Variance

class arch.univariate.**ConstantVariance**

Constant volatility process

Notes

Model has the same variance in all periods

backcast (*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds (*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance (*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints ()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate (*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers

- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

GARCH

class arch.univariate.**GARCH**(*p=1*, *o=0*, *q=1*, *power=2.0*)
GARCH and related model estimation

The following models can be specified using GARCH:

- ARCH(p)
- GARCH(p,q)
- GJR-GARCH(p,o,q)
- AVARCH(p)
- AVGARCH(p,q)
- TARCH(p,o,q)
- Models with arbitrary, pre-specified powers

Parameters

- **p** (*int*) – Order of the symmetric innovation
- **o** (*int*) – Order of the asymmetric innovation
- **q** (*int*) – Order of the lagged (transformed) conditional variance
- **power** (*float, optional*) – Power to use with the innovations, abs(e) ** power. Default is 2.0, which produces ARCH and related models. Using 1.0 produces AVARCH and related models. Other powers can be specified, although these should be strictly positive, and usually larger than 0.25.

num_params
int – The number of parameters in the model

Examples

```
>>> from arch.univariate import GARCH
```

Standard GARCH(1,1)

```
>>> garch = GARCH(p=1, q=1)
```

Asymmetric GJR-GARCH process

```
>>> gjr = GARCH(p=1, o=1, q=1)
```

Asymmetric TARCH process

```
>>> tarch = GARCH(p=1, o=1, q=1, power=1.0)
```

Notes

In this class of processes, the variance dynamics are

$$\sigma_t^\lambda = \omega + \sum_{i=1}^{p} \alpha_i \left|\epsilon_{t-i}\right|^\lambda + \sum_{j=1}^{o} \gamma_j \left|\epsilon_{t-j}\right|^\lambda I\left[\epsilon_{t-j} < 0\right] + \sum_{k=1}^{q} \beta_k \sigma_{t-k}^\lambda$$

backcast (*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds (*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance (*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints ()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate(*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

Fractionally Integrated (FI) GARCH

class arch.univariate.**FIGARCH**(*p=1*, *q=1*, *power=2.0*, *truncation=1000*)
FIGARCH model

Parameters

- **p** (*{0, 1}*) – Order of the symmetric innovation
- **q** (*{0, 1}*) – Order of the lagged (transformed) conditional variance
- **power** (*float, optional*) – Power to use with the innovations, abs(e) ** power. Default is 2.0, which produces FIGARCH and related models. Using 1.0 produces FIAVARCH and related models. Other powers can be specified, although these should be strictly positive, and usually larger than 0.25.

- **truncation** (*int, optional*) – Truncation point to use in ARCH(∞) representation. Default is 1000.

num_params
int – The number of parameters in the model

Examples

```
>>> from arch.univariate import FIGARCH
```

Standard FIGARCH

```
>>> figarch = FIGARCH()
```

FIARCH

```
>>> fiarch = FIGARCH(p=0)
```

FIAVGARCH process

```
>>> fiavarch = FIGARCH(power=1.0)
```

Notes

In this class of processes, the variance dynamics are

$$h_t = \omega + [1 - \beta L - \phi L(1-L)^d]\epsilon_t^2 + \beta h_{t-1}$$

where `L` is the lag operator and `d` is the fractional differencing parameter. The model is estimated using the ARCH(∞) representation,

$$h_t = (1-\beta)^{-1}\omega + \sum_{i=1}^{\infty} \lambda_i \epsilon_{t-i}^2$$

The weights are constructed using

$$\begin{aligned} \delta_1 &= d \\ \lambda_1 &= d - \beta + \phi \end{aligned}$$

and the recursive equations

$$\begin{aligned} \delta_j &= \frac{j-1-d}{j}\delta_{j-1} \\ \lambda_j &= \beta\lambda_{j-1} + \delta_j - \phi\delta_{j-1}. \end{aligned}$$

When *power* is not 2, the ARCH(∞) representation is still used where ϵ_t^2 is replaced by $|\epsilon_t|^p$ and `p` is the power.

backcast (*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds(*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance(*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate(*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

EGARCH

class arch.univariate.**EGARCH**(*p=1*, *o=0*, *q=1*)
EGARCH model estimation

Parameters

- **p** (*int*) – Order of the symmetric innovation
- **o** (*int*) – Order of the asymmetric innovation
- **q** (*int*) – Order of the lagged (transformed) conditional variance

num_params
int – The number of parameters in the model

Examples

```
>>> from arch.univariate import EGARCH
```

Symmetric EGARCH(1,1)

```
>>> egarch = EGARCH(p=1, q=1)
```

Standard EGARCH process

```
>>> egarch = EGARCH(p=1, o=1, q=1)
```

Exponential ARCH process

```
>>> earch = EGARCH(p=5)
```

Notes

In this class of processes, the variance dynamics are

$$\ln \sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i \left(|e_{t-i}| - \sqrt{2/\pi} \right) + \sum_{j=1}^{o} \gamma_j e_{t-j} + \sum_{k=1}^{q} \beta_k \ln \sigma_{t-k}^2$$

where $e_t = \epsilon_t / \sigma_t$.

backcast(*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds (*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance (*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints ()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate (*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals

- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

HARCH

class arch.univariate.**HARCH**(*lags=1*)
Heterogeneous ARCH process

Parameters **lags** (*{list, array, int}*) – List of lags to include in the model, or if scalar, includes all lags up the value

num_params
int – The number of parameters in the model

Examples

```
>>> from arch.univariate import HARCH
```

Lag-1 HARCH, which is identical to an ARCH(1)

```
>>> harch = HARCH()
```

More useful and realistic lag lengths

```
>>> harch = HARCH(lags=[1, 5, 22])
```

Notes

In a Heterogeneous ARCH process, variance dynamics are

$$\sigma_t^2 = \omega + \sum_{i=1}^{m} \alpha_{l_i} \left(l_i^{-1} \sum_{j=1}^{l_i} \epsilon_{t-j}^2 \right)$$

In the common case where lags=[1,5,22], the model is

$$\sigma_t^2 = \omega + \alpha_1 \epsilon_{t-1}^2 + \alpha_5 \left(\frac{1}{5} \sum_{j=1}^{5} \epsilon_{t-j}^2 \right) + \alpha_{22} \left(\frac{1}{22} \sum_{j=1}^{22} \epsilon_{t-j}^2 \right)$$

A HARCH process is a special case of an ARCH process where parameters in the more general ARCH process have been restricted.

backcast(*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds(*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance(*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate(*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

MIDAS Hyperbolic

class arch.univariate.**MIDASHyperbolic**(*m=22*, *asym=False*)
MIDAS Hyperbolic ARCH process

Parameters

- **m** (*int*) – Length of maximum lag to include in the model
- **asym** (*bool*) – Flag indicating whether to include an asymmetric term

num_params
int – The number of parameters in the model

Examples

```
>>> from arch.univariate import MIDASHyperbolic
```

22-lag MIDAS Hyperbolic process

```
>>> harch = MIDASHyperbolic()
```

Longer 66-period lag

```
>>> harch = MIDASHyperbolic(m=66)
```

Asymmetric MIDAS Hyperbolic process

```
>>> harch = MIDASHyperbolic(asym=True)
```

Notes

In a MIDAS Hyperbolic process, the variance evolves according to

$$\sigma_t^2 = \omega + \sum_{i=1}^{m} \left(\alpha + \gamma I\left[\epsilon_{t-j} < 0\right]\right) \phi_i(\theta) \epsilon_{t-i}^2$$

where

$$\phi_i(\theta) \propto \Gamma(i+\theta)/(\Gamma(i+1)\Gamma(\theta))$$

where Γ is the gamma function. $\{\phi_i(\theta)\}$ is normalized so that $\sum \phi_i(\theta) = 1$

References

backcast (*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds (*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance (*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints ()
Constraints

Notes

Parameters are (omega, alpha, gamma, theta)

A.dot(parameters) - b >= 0

1. omega >0
2. alpha>0 or alpha + gamma > 0
3. alpha<1 or alpha+0.5*gamma<1
4. theta > 0
5. theta < 1

simulate (*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate

- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

ARCH

class arch.univariate.**ARCH**(*p=1*)
ARCH process

Parameters **p** (*int*) – Order of the symmetric innovation

num_params
int – The number of parameters in the model

Examples

ARCH(1) process

```
>>> from arch.univariate import ARCH
```

ARCH(5) process

```
>>> arch = ARCH(p=5)
```

Notes

The variance dynamics of the model estimated

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2$$

backcast(*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds(*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance(*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate(*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

Parameterless Variance Processes

Some volatility processes use fixed parameters and so have no parameters that are estimable.

EWMA Variance

class arch.univariate.**EWMAVariance**(*lam=0.94*)
Bases: *arch.univariate.volatility.VolatilityProcess*

Exponentially Weighted Moving-Average (RiskMetrics) Variance process

Parameters **lam** (*{float, None}, optional*) – Smoothing parameter. Default is 0.94. Set to None to estimate lam jointly with other model parameters

num_params
int – The number of parameters in the model

Examples

Daily RiskMetrics EWMA process

```
>>> from arch.univariate import EWMAVariance
>>> rm = EWMAVariance(0.94)
```

Notes

The variance dynamics of the model

$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)\epsilon_{t-1}^2$$

When lam is provided, this model has no parameters since the smoothing parameter is treated as fixed. Sel lam to `None` to jointly estimate this parameter when fitting the model.

backcast(*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Value to use in backcasting in the volatility recursion

Return type float

bounds(*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance(*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate(*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

RiskMetrics (2006)

class arch.univariate.**RiskMetrics2006**(*tau0=1560*, *tau1=4*, *kmax=14*, *rho=1.4142135623730951*)
Bases: *arch.univariate.volatility.VolatilityProcess*

RiskMetrics 2006 Variance process

Parameters

- **tau0** (*int, optional*) – Length of long cycle. Default is 1560.
- **tau1** (*int, optional*) – Length of short cycle. Default is 4.
- **kmax** (*int, optional*) – Number of components. Default is 14.
- **rho** (*float, optional*) – Relative scale of adjacent cycles. Default is sqrt(2)

num_params
int – The number of parameters in the model

Examples

Daily RiskMetrics 2006 process

```
>>> from arch.univariate import RiskMetrics2006
>>> rm = RiskMetrics2006()
```

Notes

The variance dynamics of the model are given as a weighted average of kmax EWMA variance processes where the smoothing parameters and weights are determined by tau0, tau1 and rho.

This model has no parameters since the smoothing parameter is fixed.

backcast(*resids*)
Construct values for backcasting to start the recursion

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **backcast** – Backcast values for each EWMA component

Return type ndarray

bounds(*resids*)
Returns bounds for parameters

Parameters **resids** (*ndarray*) – Vector of (approximate) residuals

Returns **bounds** – List of bounds where each element is (lower, upper).

Return type list[tuple[float,float]]

compute_variance(*parameters*, *resids*, *sigma2*, *backcast*, *var_bounds*)
Compute the variance for the ARCH model

Parameters

- **parameters** (*ndarray*) – Model parameters
- **resids** (*ndarray*) – Vector of mean zero residuals
- **sigma2** (*ndarray*) – Array with same size as resids to store the conditional variance
- **backcast** (*{float, ndarray}*) – Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.
- **var_bounds** (*ndarray*) – Array containing columns of lower and upper bounds

constraints()
Construct parameter constraints arrays for parameter estimation

Returns

- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

simulate(*parameters*, *nobs*, *rng*, *burn=500*, *initial_value=None*)
Simulate data from the model

Parameters

- **parameters** (*{ndarray, Series}*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance

starting_values(*resids*)
Returns starting values for the ARCH model

Parameters **resids** (*ndarray*) – Array of (approximate) residuals to use when computing starting values

Returns **sv** – Array of starting values

Return type ndarray

FixedVariance

The `FixedVariance` class is a special-purpose volatility process that allows the so-called zig-zag algorithm to be used. See the example for usage.

class arch.univariate.**FixedVariance**(*variance*, *unit_scale=False*)

Bases: *arch.univariate.volatility.VolatilityProcess*

Fixed volatility process

Parameters

- **variance** (*{array, Series}*) – Array containing the variances to use. Should have the same shape as the data used in the model.
- **unit_scale** (*bool, optional*) – Flag whether to enforce a unit scale. If False, a scale parameter will be estimated so that the model variance will be proportional to `variance`. If True, the model variance is set of `variance`

Notes

Allows a fixed set of variances to be used when estimating a mean model, allowing GLS estimation.

Writing New Volatility Processes

All volatility processes must inherit from :class:VolatilityProcess and provide all public methods.

class arch.univariate.volatility.**VolatilityProcess**

Abstract base class for ARCH models. Allows the conditional mean model to be specified separately from the conditional variance, even though parameters are estimated jointly.

1.1.8 Using the Fixed Variance process

The `FixedVariance` volatility process can be used to implement zig-zag model estimation where two steps are repeated until convergence. This can be used to estimate models which may not be easy to estimate as a single process due to numerical issues or a high-dimensional parameter space.

This setup code is required to run in an IPython notebook

```
In [1]: import warnings
        warnings.simplefilter('ignore')

        %matplotlib inline
        import seaborn
        seaborn.set_style('darkgrid')

In [2]: seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
        seaborn.mpl.rcParams['savefig.dpi'] = 90
        seaborn.mpl.rcParams['font.family'] = 'serif'
        seaborn.mpl.rcParams['font.size'] = 14
```

Setup

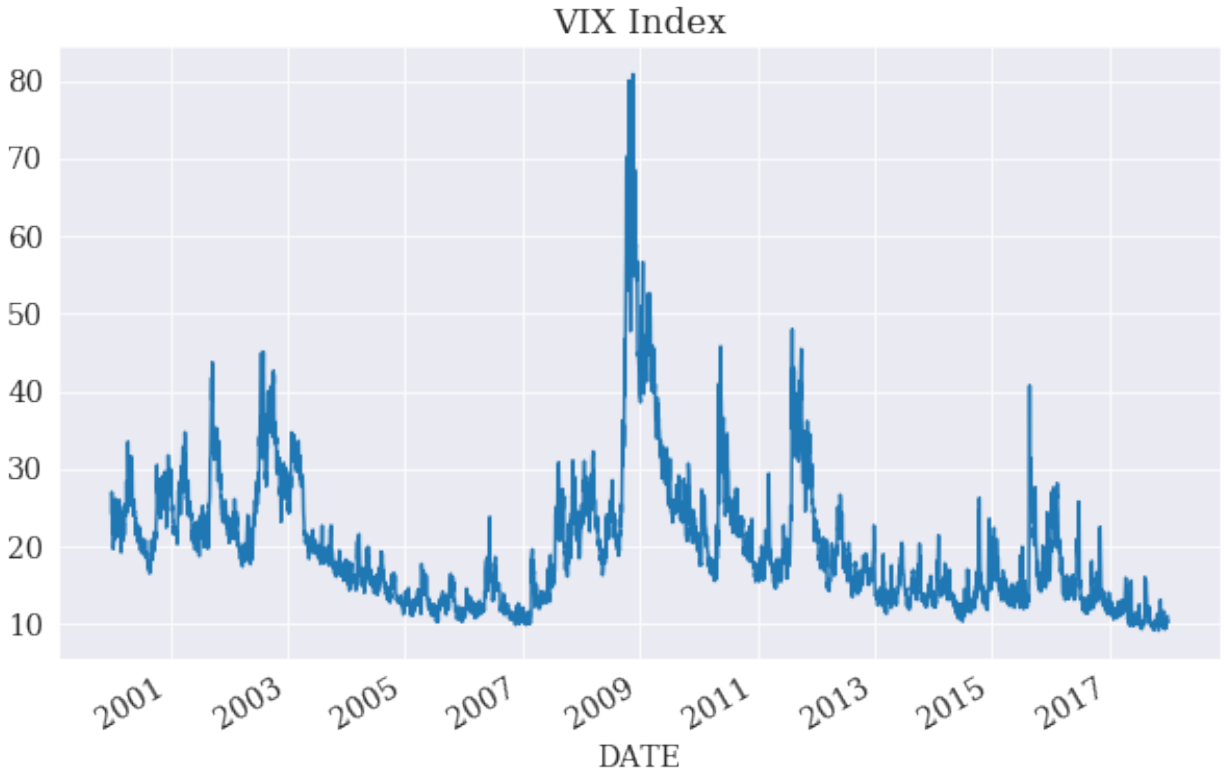
Imports used in this example.

```
In [3]: import datetime as dt
        import numpy as np
        from pandas_datareader import data
```

Data

The VIX index will be used to illustrate the use of the `FixedVariance` process. The data is read using `pandas-datareader`.

```
In [4]: dr = data.FredReader('VIXCLS', dt.datetime(2000, 1, 1), dt.datetime(2017, 12, 31))
        vix_data = dr.read()
        vix = vix_data.VIXCLS.dropna()
        vix.name = 'VIX Index'
        ax = vix.plot(title='VIX Index')
```



Initial Mean Model Estimation

The first step is to estimate the mean to filter the residuals using a constant variance.

```
In [5]: from arch.univariate.mean import HARX, ZeroMean
        from arch.univariate.volatility import GARCH, FixedVariance
        mod = HARX(vix, lags=[1,5,22])
        res = mod.fit()
        print(res.summary())
```

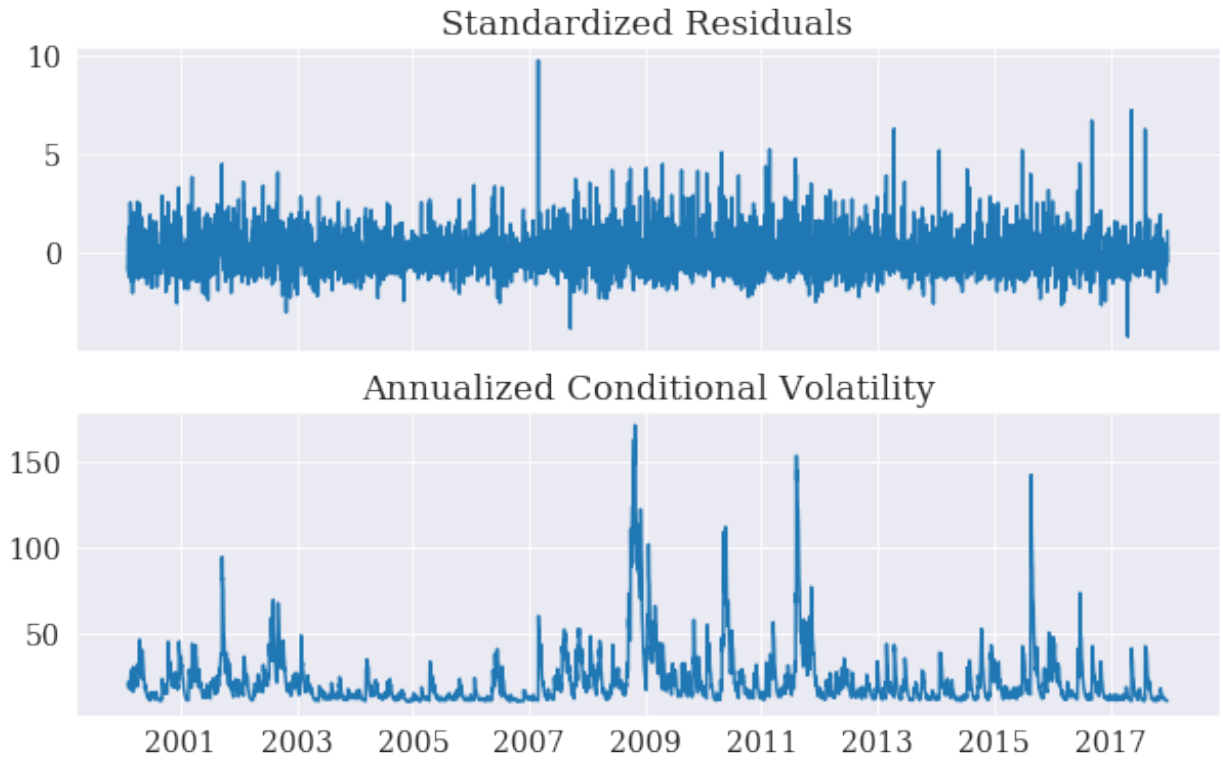
```
                     HAR - Constant Variance Model Results
==============================================================================
Dep. Variable:              VIX Index   R-squared:                       0.965
Mean Model:                       HAR   Adj. R-squared:                  0.965
Vol Model:          Constant Variance   Log-Likelihood:                -8600.27
Distribution:                  Normal   AIC:                            17210.5
Method:            Maximum Likelihood   BIC:                            17242.6
                                        No. Observations:                 4506
Date:                Thu, Sep 27 2018   Df Residuals:                     4501
Time:                        15:27:49   Df Model:                            5
                                Mean Model
==============================================================================
                     coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
Const              0.2202      0.112      1.959  5.013e-02   [-1.208e-04,  0.440]
VIX Index[0:1]     0.8418  4.112e-02     20.473  3.722e-93      [  0.761,  0.922]
VIX Index[0:5]     0.1150  4.780e-02      2.407  1.610e-02    [2.135e-02,  0.209]
VIX Index[0:22]    0.0319  2.484e-02      1.283      0.199 [-1.681e-02,8.058e-02]
                            Volatility Model
=============================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
-----------------------------------------------------------------------------
sigma2         2.6628      0.176     15.110  1.396e-51 [  2.317,  3.008]
=============================================================================

Covariance estimator: White's Heteroskedasticity Consistent Estimator
```

Initial Volatility Model Estimation

Using the previously estimated residuals, a volatility model can be estimated using a `ZeroMean`. In this example, a GJR-GARCH process is used for the variance.

```
In [6]: vol_mod = ZeroMean(res.resid.dropna(), volatility=GARCH(p=1,o=1,q=1))
        vol_res = vol_mod.fit(disp='off')
        print(vol_res.summary())
        ax = vol_res.plot('D')
```

```
                    Zero Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:                   resid   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                  GJR-GARCH   Log-Likelihood:                -7167.57
Distribution:                  Normal   AIC:                            14343.1
Method:            Maximum Likelihood   BIC:                            14368.8
                                        No. Observations:                 4506
Date:                Thu, Sep 27 2018   Df Residuals:                     4502
Time:                        15:27:49   Df Model:                            4
                            Volatility Model
=============================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
-----------------------------------------------------------------------------
omega          0.0738  1.683e-02      4.382  1.174e-05 [4.077e-02,  0.107]
alpha[1]       0.2680  3.863e-02      6.937  4.004e-12   [  0.192,  0.344]
gamma[1]      -0.2680  3.575e-02     -7.496  6.599e-14   [ -0.338, -0.198]
beta[1]        0.8234  2.399e-02     34.318 4.173e-258   [  0.776,  0.870]
=============================================================================

Covariance estimator: robust
```



Re-estimating the mean with a `FixedVariance`

The `FixedVariance` requires that the variance is provided when initializing the object. The variance provided should have the same shape as the original data. Since the variance estimated from the GJR-GARCH model is missing the first 22 observations due to the HAR lags, we simply fill these with 1. These values will not be used to estimate the model, and so the value is not important.

The summary shows that there is a single parameter, `scale`, which is close to 1. The mean parameters have changed which reflects the GLS-like weighting that this re-estimaion imposes.

```
In [7]: variance = np.empty_like(vix)
        variance.fill(1.0)
        variance[22:] = vol_res.conditional_volatility ** 2.0
        fv = FixedVariance(variance)
        mod = HARX(vix, lags=[1,5,22], volatility=fv)
        res = mod.fit()
        print(res.summary())
```

```
Iteration:      1,   Func. Count:      7,   Neg. LLF: 7167.56600290707
Iteration:      2,   Func. Count:     21,   Neg. LLF: 7167.542807019507
Iteration:      3,   Func. Count:     32,   Neg. LLF: 7164.798624995907
Iteration:      4,   Func. Count:     43,   Neg. LLF: 7162.605127839062
Iteration:      5,   Func. Count:     53,   Neg. LLF: 7162.600913736296
Iteration:      6,   Func. Count:     63,   Neg. LLF: 7162.600864075792
Iteration:      7,   Func. Count:     70,   Neg. LLF: 7162.600859729137
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 7162.6008597291375
            Iterations: 7
            Function evaluations: 70
            Gradient evaluations: 7
```

```
                     HAR - Fixed Variance Model Results
==============================================================================
Dep. Variable:              VIX Index   R-squared:                       0.965
Mean Model:                       HAR   Adj. R-squared:                  0.965
Vol Model:            Fixed Variance   Log-Likelihood:                -7162.60
Distribution:                 Normal   AIC:                            14335.2
Method:           Maximum Likelihood   BIC:                            14367.3
                                       No. Observations:                 4506
Date:                Thu, Sep 27 2018   Df Residuals:                     4501
Time:                        15:27:49   Df Model:                            5
                            Mean Model
==============================================================================
                     coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
Const              0.2256  5.095e-02      4.427  9.560e-06     [  0.126,  0.325]
VIX Index[0:1]     0.8969  1.780e-02     50.383      0.000     [  0.862,  0.932]
VIX Index[0:5]     0.0570  2.204e-02      2.587  9.681e-03   [1.382e-02,  0.100]
VIX Index[0:22]    0.0347  1.201e-02      2.891  3.836e-03 [1.119e-02,5.826e-02]
                          Volatility Model
===========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
---------------------------------------------------------------------------
scale          0.9978  4.106e-02     24.299 2.020e-130 [  0.917,  1.078]
===========================================================================

Covariance estimator: robust
```

Zig-Zag estimation

A small repetitions of the previous two steps can be used to implement a so-called zig-zag estimation strategy.

```
In [8]: for i in range(5):
            print(i)
            vol_mod = ZeroMean(res.resid.dropna(), volatility=GARCH(p=1,o=1,q=1))
            vol_res = vol_mod.fit(disp='off')
            variance[22:] = vol_res.conditional_volatility ** 2.0
            fv = FixedVariance(variance, unit_scale=True)
            mod = HARX(vix, lags=[1,5,22], volatility=fv)
            res = mod.fit(disp='off')
        print(res.summary())
```

```
0
1
2
3
4
                  HAR - Fixed Variance (Unit Scale) Model Results
==============================================================================
Dep. Variable:                      VIX Index   R-squared:                       0.965
Mean Model:                               HAR   Adj. R-squared:                  0.965
Vol Model:         Fixed Variance (Unit Scale)   Log-Likelihood:                -7165.87
Distribution:                           Normal   AIC:                            14339.7
Method:                    Maximum Likelihood   BIC:                            14365.4
                                                No. Observations:                 4506
Date:                         Thu, Sep 27 2018   Df Residuals:                     4502
Time:                                 15:27:50   Df Model:                            4
                                     Mean Model
==============================================================================
```

```
                     coef    std err          t      P>|t|      95.0% Conf. Int.
-----------------------------------------------------------------------------
Const               0.2237  5.102e-02      4.385  1.159e-05     [  0.124,  0.324]
VIX Index[0:1]      0.8971  1.781e-02     50.362      0.000     [  0.862,  0.932]
VIX Index[0:5]      0.0567  2.206e-02      2.571  1.015e-02 [1.348e-02,9.995e-02]
VIX Index[0:22]     0.0349  1.203e-02      2.901  3.720e-03 [1.132e-02,5.849e-02]
=============================================================================

Covariance estimator: robust
```

Direct Estimation

This model can be directly estimated. The results are provided for comparisson to the previous `FixedVariance` estimates of the mean parameters.

```
In [9]: mod = HARX(vix, lags=[1,5,22], volatility=GARCH(1,1,1))
        res = mod.fit(disp='off')
        print(res.summary())
```

```
                       HAR - GJR-GARCH Model Results
==============================================================================
Dep. Variable:              VIX Index   R-squared:                       0.965
Mean Model:                       HAR   Adj. R-squared:                  0.965
Vol Model:                  GJR-GARCH   Log-Likelihood:               -7159.40
Distribution:                  Normal   AIC:                           14334.8
Method:            Maximum Likelihood   BIC:                           14386.1
                                        No. Observations:                4506
Date:                Thu, Sep 27 2018   Df Residuals:                    4498
Time:                        15:27:50   Df Model:                           8
                                 Mean Model
=============================================================================
                     coef    std err          t      P>|t|      95.0% Conf. Int.
-----------------------------------------------------------------------------
Const               0.3235  9.679e-02      3.342  8.313e-04     [  0.134,  0.513]
VIX Index[0:1]      0.8709  1.993e-02     43.693      0.000     [  0.832,  0.910]
VIX Index[0:5]      0.0753  2.353e-02      3.201  1.370e-03   [2.920e-02,  0.121]
VIX Index[0:22]     0.0348  1.733e-02      2.009  4.449e-02 [8.581e-04,6.879e-02]
                          Volatility Model
===========================================================================
                 coef    std err          t      P>|t|    95.0% Conf. Int.
---------------------------------------------------------------------------
omega          0.0779  1.888e-02      4.128  3.662e-05 [4.093e-02,  0.115]
alpha[1]       0.2672  4.465e-02      5.985  2.164e-09   [  0.180,  0.355]
gamma[1]      -0.2672  4.949e-02     -5.400  6.664e-08   [ -0.364, -0.170]
beta[1]        0.8149  3.654e-02     22.300 3.713e-110   [  0.743,  0.886]
===========================================================================

Covariance estimator: robust
```

1.1.9 Distributions

A distribution is the final component of an ARCH Model.

Normal

class arch.univariate.**Normal**(*random_state=None*)
Standard normal distribution for use with ARCH models

bounds(*resids*)

Parameters **resids** (*ndarray*) – Residuals to use when computing the bounds

Returns **bounds** – List containing a single tuple with (lower, upper) bounds

Return type list

constraints()

Returns

- **A** (*ndarray*) – Constraint loadings
- **b** (*ndarray*) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

loglikelihood(*parameters*, *resids*, *sigma2*, *individual=False*)
Computes the log-likelihood of assuming residuals are normally distributed, conditional on the variance

Parameters

- **parameters** (*ndarray*) – The normal likelihood has no shape parameters. Empty since the standard normal has no shape parameters.
- **resids** (*ndarray*) – The residuals to use in the log-likelihood calculation
- **sigma2** (*ndarray*) – Conditional variances of resids
- **individual** (*bool, optional*) – Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns **ll** – The log-likelihood

Return type float

Notes

The log-likelihood of a single data point x is

$$\ln f(x) = -\frac{1}{2}\left(\ln 2\pi + \ln \sigma^2 + \frac{x^2}{\sigma^2}\right)$$

simulate(*parameters*)
Simulates i.i.d. draws from the distribution

Parameters **parameters** (*ndarray*) – Distribution parameters

Returns **simulator** – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type callable

starting_values(*std_resid*)

Parameters **std_resid** (*ndarray*) – Estimated standardized residuals to use in computing starting values for the shape parameter

Returns **sv** – The estimated shape parameters for the distribution

Return type ndarray

Notes

Size of sv depends on the distribution

Student's t

class arch.univariate.**StudentsT**(*random_state=None*)

Standardized Student's distribution for use with ARCH models

bounds(*resids*)

Parameters **resids** (*ndarray*) – Residuals to use when computing the bounds

Returns **bounds** – List containing a single tuple with (lower, upper) bounds

Return type list

constraints()

Returns

- **A** (*ndarray*) – Constraint loadings
- **b** (*ndarray*) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

loglikelihood(*parameters*, *resids*, *sigma2*, *individual=False*)

Computes the log-likelihood of assuming residuals are have a standardized (to have unit variance) Student's t distribution, conditional on the variance.

Parameters

- **parameters** (*ndarray*) – Shape parameter of the t distribution
- **resids** (*ndarray*) – The residuals to use in the log-likelihood calculation
- **sigma2** (*ndarray*) – Conditional variances of resids
- **individual** (*bool, optional*) – Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns **ll** – The log-likelihood

Return type float

Notes

The log-likelihood of a single data point x is

$$\ln \Gamma\left(\frac{\nu+1}{2}\right) - \ln \Gamma\left(\frac{\nu}{2}\right) - \frac{1}{2}\ln(\pi\,(\nu-2)\,\sigma^2) - \frac{\nu+1}{2}\ln(1+x^2/(\sigma^2(\nu-2)))$$

where Γ is the gamma function.

simulate (*parameters*)
Simulates i.i.d. draws from the distribution

Parameters **parameters** (*ndarray*) – Distribution parameters

Returns **simulator** – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type callable

starting_values (*std_resid*)

Parameters **std_resid** (*ndarray*) – Estimated standardized residuals to use in computing starting values for the shape parameter

Returns **sv** – Array containing starting valuer for shape parameter

Return type ndarray

Notes

Uses relationship between kurtosis and degree of freedom parameter to produce a moment-based estimator for the starting values.

Skew Student's t

class arch.univariate.**SkewStudent** (*random_state=None*)
Standardized Skewed Student's[1] distribution for use with ARCH models

Notes

The Standardized Skewed Student's distribution takes two parameters, η and λ. η controls the tail shape and is similar to the shape parameter in a Standardized Student's t. λ controls the skewness. When $\lambda = 0$ the distribution is identical to a standardized Student's t.

References

bounds (*resids*)

Parameters **resids** (*ndarray*) – Residuals to use when computing the bounds

Returns **bounds** – List containing a single tuple with (lower, upper) bounds

Return type list

[1] Hansen, B. E. (1994). Autoregressive conditional density estimation. *International Economic Review*, 35(3), 705–730. <http://www.ssc.wisc.edu/~bhansen/papers/ier_94.pdf>

constraints()

Returns

- **A** (*ndarray*) – Constraint loadings
- **b** (*ndarray*) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

loglikelihood(*parameters*, *resids*, *sigma2*, *individual=False*)

Computes the log-likelihood of assuming residuals are have a standardized (to have unit variance) Skew Student's t distribution, conditional on the variance.

Parameters

- **parameters** (*ndarray*) – Shape parameter of the skew-t distribution
- **resids** (*ndarray*) – The residuals to use in the log-likelihood calculation
- **sigma2** (*ndarray*) – Conditional variances of resids
- **individual** (*bool, optional*) – Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns **ll** – The log-likelihood

Return type float

Notes

The log-likelihood of a single data point x is

$$\ln\left[\frac{bc}{\sigma}\left(1+\frac{1}{\eta-2}\left(\frac{a+bx/\sigma}{1+sgn(x/\sigma+a/b)\lambda}\right)^2\right)^{-(\eta+1)/2}\right],$$

where $2<\eta<\infty$, and $-1<\lambda<1$. The constants a, b, and c are given by

$$a=4\lambda c\frac{\eta-2}{\eta-1},\quad b^2=1+3\lambda^2-a^2,\quad c=\frac{\Gamma\left(\frac{\eta+1}{2}\right)}{\sqrt{\pi\left(\eta-2\right)}\Gamma\left(\frac{\eta}{2}\right)},$$

and Γ is the gamma function.

simulate(*parameters*)

Simulates i.i.d. draws from the distribution

Parameters **parameters** (*ndarray*) – Distribution parameters

Returns **simulator** – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type callable

starting_values(*std_resid*)

Parameters **std_resid** (*ndarray*) – Estimated standardized residuals to use in computing starting values for the shape parameter

Returns **sv** – Array containing starting valuer for shape parameter

Return type ndarray

Notes

Uses relationship between kurtosis and degree of freedom parameter to produce a moment-based estimator for the starting values.

Generalized Error (GED)

class arch.univariate.**GeneralizedError**(*random_state=None*)

Generalized Error distribution for use with ARCH models

bounds(*resids*)

Parameters **resids** (*ndarray*) – Residuals to use when computing the bounds

Returns **bounds** – List containing a single tuple with (lower, upper) bounds

Return type list

constraints()

Returns

- **A** (*ndarray*) – Constraint loadings
- **b** (*ndarray*) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

loglikelihood(*parameters*, *resids*, *sigma2*, *individual=False*)

Computes the log-likelihood of assuming residuals are have a Generalized Error Distribution, conditional on the variance.

Parameters

- **parameters** (*ndarray*) – Shape parameter of the GED distribution
- **resids** (*ndarray*) – The residuals to use in the log-likelihood calculation
- **sigma2** (*ndarray*) – Conditional variances of resids
- **individual** (*bool, optional*) – Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns **ll** – The log-likelihood

Return type float

Notes

The log-likelihood of a single data point x is

$$\ln \nu - \ln c - \ln \Gamma(\frac{1}{\nu}) + (1 + \frac{1}{\nu}) \ln 2 - \frac{1}{2} \ln \sigma^2 - \frac{1}{2} \left| \frac{x}{c\sigma} \right|^{\nu}$$

where Γ is the gamma function and $\ln c$ is

$$\ln c = \frac{1}{2}\left(\frac{-2}{\nu}\ln 2 + \ln\Gamma(\frac{1}{\nu}) - \ln\Gamma(\frac{3}{\nu})\right).$$

simulate (*parameters*)
Simulates i.i.d. draws from the distribution

Parameters **parameters** (*ndarray*) – Distribution parameters

Returns **simulator** – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type callable

starting_values (*std_resid*)

Parameters **std_resid** (*ndarray*) – Estimated standardized residuals to use in computing starting values for the shape parameter

Returns **sv** – Array containing starting valuer for shape parameter

Return type ndarray

Notes

Defaults to 1.5 which is implies heavier tails than a normal

Writing New Distributions

All distributions must inherit from :class:Distribution and provide all public methods.

class arch.univariate.**Distribution** (*name*, *random_state=None*)
Template for subclassing only

1.1.10 Theoretical Background

To be completed

1.2 Bootstrapping

The bootstrap module provides both high- and low-level interfaces for bootstrapping data contained in NumPy arrays or pandas Series or DataFrames.

All bootstraps have the same interfaces and only differ in their name, setup parameters and the (internally generated) sampling scheme.

1.2.1 Bootstrap Examples

This setup code is required to run in an IPython notebook

```
In [1]: import warnings
        warnings.simplefilter('ignore')

        %matplotlib inline
        import seaborn
        seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
        seaborn.mpl.rcParams['savefig.dpi'] = 90
```

Sharpe Ratio

The Sharpe Ratio is an important measure of return per unit of risk. The example shows how to estimate the variance of the Sharpe Ratio and how to construct confidence intervals for the Sharpe Ratio using a long series of U.S. equity data. First, the data is imported using pandas.

```
In [2]: import numpy as np
        import pandas as pd
        import pandas_datareader.data as web
        try:
            ff=web.DataReader('F-F_Research_Data_Factors', 'famafrench')
        except:
            ff=web.DataReader('F-F_Research_Data_Factors_TXT', 'famafrench')
        ff = ff[0]
```

The data set contains the Fama-French factors, including the excess market return.

```
In [3]: excess_market = ff.iloc[:,0]
        print(ff.describe())
```

```
            Mkt-RF         SMB         HML          RF
count  103.000000  103.000000  103.000000  103.000000
mean     1.139806    0.107379   -0.131068    0.020680
std      3.542553    2.273547    2.162328    0.037397
min     -7.890000   -4.370000   -4.500000    0.000000
25%     -0.985000   -1.720000   -1.530000    0.000000
50%      1.180000    0.320000   -0.290000    0.010000
75%      3.155000    1.230000    1.015000    0.020000
max     11.350000    5.490000    8.270000    0.160000
```

The next step is to construct a function that computes the Sharpe Ratio. This function also return the annualized mean and annualized standard deviation which will allow the covariance matrix of these parameters to be estimated using the bootstrap.

```
In [4]: def sharpe_ratio(x):
            mu, sigma = 12 * x.mean(), np.sqrt(12 * x.var())
            values = np.array([mu, sigma, mu / sigma ]).squeeze()
            index = ['mu', 'sigma', 'SR']
            return pd.Series(values, index=index)
```

The function can be called directly on the data to show full sample estimates.

```
In [5]: params = sharpe_ratio(excess_market)
        params
```

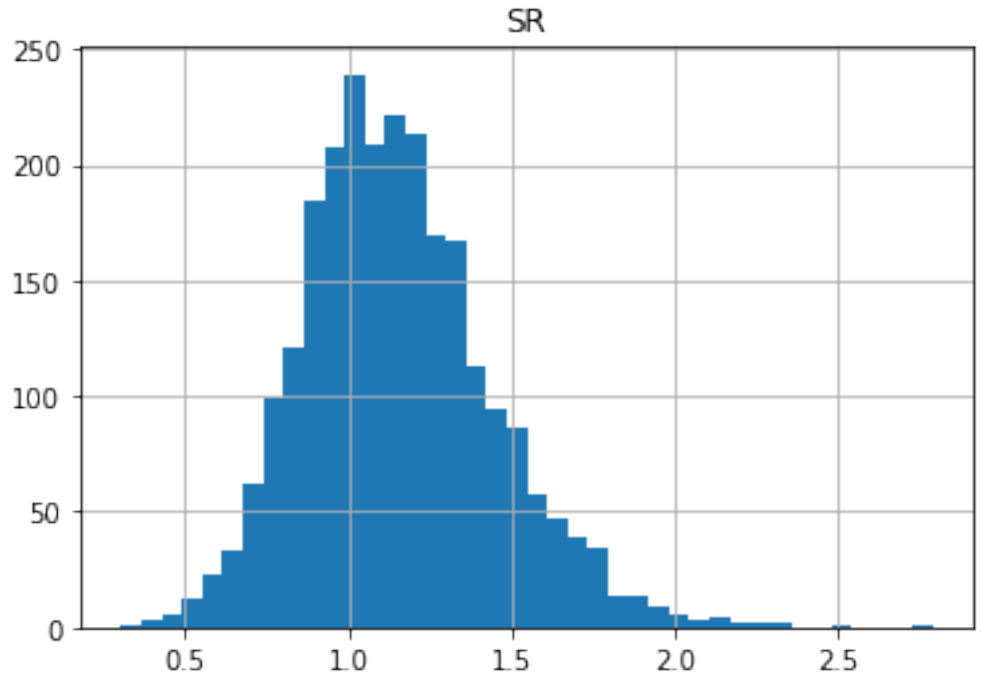
```
Out[5]: mu       13.677670
        sigma    12.271762
        SR        1.114564
        dtype: float64
```

Warning

The bootstrap chosen must be appropriate for the data. Squared returns are serially correlated, and so a time-series bootstrap is required.

Bootstraps are initialized with any bootstrap specific parameters and the data to be used in the bootstrap. Here the `12` is the average window length in the Stationary Bootstrap, and the next input is the data to be bootstrapped.

```
In [6]: from arch.bootstrap import StationaryBootstrap
        bs = StationaryBootstrap(12, excess_market)
        results = bs.apply(sharpe_ratio, 2500)
        SR = pd.DataFrame(results[:,-1:], columns=['SR'])
        fig = SR.hist(bins=40)
```




```
In [7]: cov = bs.cov(sharpe_ratio, 1000)
        cov = pd.DataFrame(cov, index=params.index, columns=params.index)
        print(cov)
        se = pd.Series(np.sqrt(np.diag(cov)), index=params.index)
        se.name = 'Std Errors'
        print('\n')
        print(se)
```

```
              mu     sigma        SR
mu     6.799678 -1.070593  0.673616
sigma -1.070593  1.785856 -0.264456
SR     0.673616 -0.264456  0.083816


mu       2.607619
sigma    1.336359
SR       0.289510
Name: Std Errors, dtype: float64
```

```
In [8]: ci = bs.conf_int(sharpe_ratio, 1000, method='basic')
        ci = pd.DataFrame(ci, index=['Lower','Upper'], columns=params.index)
```

```
        print(ci)
```

```
              mu      sigma        SR
Lower   8.300243  10.087436  0.472338
Upper  18.961893  14.968219  1.600511
```

Alternative confidence intervals can be computed using a variety of methods. Setting `reuse=True` allows the previous bootstrap results to be used when constructing confidence intervals using alternative methods.

```
In [9]: ci = bs.conf_int(sharpe_ratio, 1000, method='percentile', reuse=True)
        ci = pd.DataFrame(ci, index=['Lower','Upper'], columns=params.index)
        print(ci)
```

```
              mu      sigma        SR
Lower   8.393447   9.575305  0.628618
Upper  19.055097  14.456088  1.756791
```

Probit (Statsmodels)

The second example makes use of a Probit model from Statsmodels. The demo data is university admissions data which contains a binary variable for being admitted, GRE score, GPA score and quartile rank. This data is downloaded from the internet and imported using pandas.

```
In [10]: import numpy as np
         import pandas as pd
         try:
             import urllib2
             import StringIO
         except ImportError:
             import urllib.request as urllib2
             from io import StringIO

         url = 'https://stats.idre.ucla.edu/stat/stata/dae/binary.dta'
         binary = pd.read_stata(url)
         binary = binary.dropna()
         print(binary.describe())
```

```
            admit         gre         gpa        rank
count  400.000000  400.000000  400.000000  400.00000
mean     0.317500  587.700012    3.389900    2.48500
std      0.466087  115.516541    0.380567    0.94446
min      0.000000  220.000000    2.260000    1.00000
25%      0.000000  520.000000    3.130000    2.00000
50%      0.000000  580.000000    3.395000    2.00000
75%      1.000000  660.000000    3.670000    3.00000
max      1.000000  800.000000    4.000000    4.00000
```

Fitting the model directly

The first steps are to build the regressor and the dependent variable arrays. Then, using these arrays, the model can be estimated by calling `fit`

```
In [11]: endog = binary[['admit']]
         exog = binary[['gre','gpa']]
         const = pd.Series(np.ones(exog.shape[0]), index=endog.index)
         const.name = 'Const'
         exog = pd.DataFrame([const, exog.gre, exog.gpa]).T
         # Estimate the model
         import statsmodels.api as sm
```

```
         mod = sm.Probit(endog, exog)
         fit = mod.fit(disp=0)
         params = fit.params
         print(params)
```

```
Const   -3.003536
gre      0.001643
gpa      0.454575
dtype: float64
```

The wrapper function

Most models in Statsmodels are implemented as classes, require an explicit call to `fit` and return a class containing parameter estimates and other quantities. These classes cannot be directly used with the bootstrap methods. However, a simple wrapper can be written that takes the data as the only inputs and returns parameters estimated using a Statsmodel model.

```
In [12]: def probit_wrap(endog, exog):
             return sm.Probit(endog, exog).fit(disp=0).params
```

A call to this function should return the same parameter values.

```
In [13]: probit_wrap(endog, exog)
```

```
Out[13]: Const   -3.003536
         gre      0.001643
         gpa      0.454575
         dtype: float64
```

The wrapper can be directly used to estimate the parameter covariance or to construct confidence intervals.

```
In [14]: from arch.bootstrap import IIDBootstrap
         bs = IIDBootstrap(endog=endog, exog=exog)
         cov = bs.cov(probit_wrap, 1000)
         cov = pd.DataFrame(cov, index=exog.columns, columns=exog.columns)
         print(cov)
```

```
           Const           gre       gpa
Const   0.450423 -8.740087e-05 -0.114939
gre    -0.000087  4.328506e-07 -0.000050
gpa    -0.114939 -5.012678e-05  0.042329
```

```
In [15]: se = pd.Series(np.sqrt(np.diag(cov)), index=exog.columns)
         print(se)
         print('T-stats')
         print(params / se)
```

```
Const    0.671135
gre      0.000658
gpa      0.205740
dtype: float64
T-stats
Const   -4.475306
gre      2.496583
gpa      2.209460
dtype: float64
```

```
In [16]: ci = bs.conf_int(probit_wrap, 1000, method='basic')
         ci = pd.DataFrame(ci, index=['Lower','Upper'], columns=exog.columns)
         print(ci)
```

```
          Const       gre       gpa
Lower -4.205996  0.000348  0.044685
Upper -1.641689  0.003006  0.829251
```

Speeding things up

Starting values can be provided to `fit` which can save time finding starting values. Since the bootstrap parameter estimates should be close to the original sample estimates, the full sample estimated parameters are reasonable starting values. These can be passed using the `extra_kwargs` dictionary to a modified wrapper that will accept a keyword argument containing starting values.

```
In [17]: def probit_wrap_start_params(endog, exog, start_params=None):
             return sm.Probit(endog, exog).fit(start_params=start_params, disp=0).params
```

```
In [18]: bs.reset()  # Reset to original state for comparability
         cov = bs.cov(probit_wrap_start_params, 1000, extra_kwargs={'start_params': params.values})
         cov = pd.DataFrame(cov, index=exog.columns, columns=exog.columns)
         print(cov)
```

```
           Const           gre       gpa
Const   0.450423 -8.740087e-05 -0.114939
gre    -0.000087  4.328506e-07 -0.000050
gpa    -0.114939 -5.012678e-05  0.042329
```

1.2.2 Confidence Intervals

The confidence interval function allows three types of confidence intervals to be constructed:

- Nonparametric, which only resamples the data
- Semi-parametric, which use resampled residuals
- Parametric, which simulate residuals

Confidence intervals can then be computed using one of 6 methods:

- Basic (`basic`)
- Percentile (`percentile`)
- Studentized (`studentized`)
- Asymptotic using parameter covariance (`norm`, `var` or `cov`)
- Bias-corrected (`bc`, `bias-corrected` or `debiased`)
- Bias-corrected and accelerated (`bca`)

- *Percentile* (`percentile`)
- *Asymptotic Normal Approximation* (`norm`, `cov` or `var`)
- *Studentized* (`studentized`)
- *Bias-corrected* (`bc`, `bias-corrected` or `debiased`)
- *Bias-corrected and accelerated* (`bca`)

Setup

All examples will construct confidence intervals for the Sharpe ratio of the S&P 500, which is the ratio of the assumalized mean to the annualized standard deviation. The parameters will be the annualized mean, the annualized standard deviation and the Sharpe ratio.

The setup makes use of return data downloaded from Yahoo!

```
import datetime as dt

import pandas as pd
import pandas_datareader.data as web

start = dt.datetime(1951, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
low = sp500.index.min()
high = sp500.index.max()
monthly_dates = pd.date_range(low, high, freq='M')
monthly = sp500.reindex(monthly_dates, method='ffill')
returns = 100 * monthly['Adj Close'].pct_change().dropna()
```

The main function used will return a 3-element array containing the parameters.

```
def sharpe_ratio(x):
    mu, sigma = 12 * x.mean(), np.sqrt(12 * x.var())
    return np.array([mu, sigma, mu / sigma])
```

Note: Functions must return 1-d NumPy arrays or Pandas Series.

Confidence Interval Types

Three types of confidence intervals can be computed. The simplest are non-parametric; these only make use of parameter estimates from both the original data as well as the resampled data. Semi-parametric mix the original data with a limited form of resampling, usually for residuals. Finally, parametric bootstrap confidence intervals make use of a parametric distribution to construct "as-if" exact confidence intervals.

Nonparametric Confidence Intervals

Non-parametric sampling is the simplest method to construct confidence intervals.

This example makes use of the percentile bootstrap which is conceptually the simplest method - it constructs many bootstrap replications and returns order statistics from these empirical distributions.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='percentile')
```

Note: While returns have little serial correlation, squared returns are highly persistent. The IID bootstrap is not a good choice here. Instead a time-series bootstrap with an appropriately chosen block size should be used.

Semi-parametric Confidence Intervals

See *Semiparametric Bootstraps*

Parametric Confidence Intervals

See *Parametric Bootstraps*

Confidence Interval Methods

Note: `conf_int` can construct two-sided, upper or lower (one-sided) confidence intervals. All examples use two-sided, 95% confidence intervals (the default). This can be modified using the keyword inputs `type` (`'upper'`, `'lower'` or `'two-sided'`) and `size`.

Basic (`basic`)

Basic confidence intervals construct many bootstrap replications $\hat{\theta}_b^\star$ and then constructs the confidence interval as

$$\left[\hat{\theta} + \left(\hat{\theta} - \hat{\theta}_u^\star\right), \hat{\theta} + \left(\hat{\theta} - \hat{\theta}_l^\star\right)\right]$$

where $\hat{\theta}_l^\star$ and $\hat{\theta}_u^\star$ are the $\alpha/2$ and $1-\alpha/2$ empirical quantiles of the bootstrap distribution. When θ is a vector, the empirical quantiles are computed element-by-element.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='basic')
```

Percentile (`percentile`)

The percentile method directly constructs condifence intervals from the emprical CDF of the bootstrap parameter estimates, $\hat{\theta}_b^\star$. The confidence interval is then defined.

$$\left[\hat{\theta}_l^\star, \hat{\theta}_u^\star\right]$$

where $\hat{\theta}_l^\star$ and $\hat{\theta}_u^\star$ are the $\alpha/2$ and $1-\alpha/2$ empirical quantiles of the bootstrap distribution.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='percentile')
```

Asymptotic Normal Approximation (`norm`, `cov` or `var`)

The asymptotic normal approximation method estimates the covairance of the parameters and then combines this with the usual quantiles from a normal distribution. The confidence interval is then

$$\left[\hat{\theta} + \hat{\sigma}\Phi^{-1}\left(\alpha/2\right), \hat{\theta} - \hat{\sigma}\Phi^{-1}\left(\alpha/2\right),\right]$$

where $\hat{\sigma}$ is the bootstrap estimate of the parameter standard error.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='norm')
```

Studentized (`studentized`)

The studentized bootstrap may be more accurate than some of the other methods. The studentized bootstrap makes use of either a standard error function, when parameter standard errors can be analytically computed, or a nested bootstrap, to bootstrap studentized versions of the original statistic. This can produce higher-order refinements in some circumstances.

The confidence interval is then

$$\left[\hat{\theta} + \hat{\sigma}\hat{G}^{-1}\left(\alpha/2\right), \hat{\theta} + \hat{\sigma}\hat{G}^{-1}\left(1-\alpha/2\right),\right]$$

where $\hat{G}$ is the estimated quantile function for the studentized data and where $\hat{\sigma}$ is a bootstrap estimate of the parameter standard error.

The version that uses a nested bootstrap is simple to implement although it can be slow since it requires B inner bootstraps of each of the B outer bootstraps.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='studentized')
```

In order to use the standard error function, it is necessary to estimate the standard error of the parameters. In this example, this can be done using a method-of-moments argument and the delta-method. A detailed description of the mathematical formula is beyond the intent of this document.

```
def sharpe_ratio_se(params, x):
    mu, sigma, sr = params
    y = 12 * x
    e1 = y - mu
    e2 = y ** 2.0 - sigma ** 2.0
    errors = np.vstack((e1, e2)).T
    t = errors.shape[0]
    vcv = errors.T.dot(errors) / t
```

(continues on next page)

(continued from previous page)

```
    D = np.array([[1, 0],
                  [0, 0.5 * 1 / sigma],
                  [1.0 / sigma, - mu / (2.0 * sigma**3)]
                  ])
    avar = D.dot(vcv /t).dot(D.T)
    return np.sqrt(np.diag(avar))
```

The studentized bootstrap can then be implemented using the standard error function.

```
from arch.bootstrap import IIDBootstrap
bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='studentized',
                 std_err_func=sharpe_ratio_se)
```

Note: Standard error functions must return a 1-d array with the same number of element as params.

Note: Standard error functions must match the patters `std_err_func(params, *args, **kwargs)` where `params` is an array of estimated parameters constructed using `*args` and `**kwargs`.

Bias-corrected (`bc`, `bias-corrected` or `debiased`)

The bias corrected bootstrap makes use of a bootstrap estimate of the bias to improve confidence intervals.

```
from arch.bootstrap import IIDBootstrap
bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='bc')
```

The bias-corrected confidence interval is identical to the bias-corrected and accelerated where $a = 0$.

Bias-corrected and accelerated (`bca`)

Bias-corrected and accelerated confidence intervals make use of both a bootstrap bias estimate and a jackknife acceleration term. BCa intervals may offer higher-order accuracy if some conditions are satisfied. Bias-corrected confidence intervals are a special case of BCa intervals where the acceleration parameter is set to 0.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='bca')
```

The confidence interval is based on the empirical distribution of the bootstrap parameter estimates, $\hat{\theta}_b^\star$, where the percentiles used are

$$\Phi\left(\Phi^{-1}\left(\hat{b}\right)+\frac{\Phi^{-1}\left(\hat{b}\right)+z_\alpha}{1-\hat{a}\left(\Phi^{-1}\left(\hat{b}\right)+z_\alpha\right)}\right)$$

where z_α is the usual quantile from the normal distribution and b is the empirical bias estimate,

$$\hat{b}=\#\left\{\hat{\theta}_b^\star<\hat{\theta}\right\}/B$$

a is a skewness-like estimator using a leave-one-out jackknife.

1.2.3 Covariance Estimation

The bootstrap can be used to estimate parameter covariances in applications where analytical computation is challenging, or simply as an alternative to traditional estimators.

This example estimates the covariance of the mean, standard deviation and Sharpe ratio of the S&P 500 using Yahoo! Finance data.

```
import datetime as dt
import pandas as pd
import pandas_datareader.data as web

start = dt.datetime(1951, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
low = sp500.index.min()
high = sp500.index.max()
monthly_dates = pd.date_range(low, high, freq='M')
monthly = sp500.reindex(monthly_dates, method='ffill')
returns = 100 * monthly['Adj Close'].pct_change().dropna()
```

The function that returns the parameters.

```
def sharpe_ratio(r):
    mu = 12 * r.mean(0)
    sigma = np.sqrt(12 * r.var(0))
    sr = mu / sigma
    return np.array([mu, sigma, sr])
```

Like all applications of the bootstrap, it is important to choose a bootstrap that captures the dependence in the data. This example uses the stationary bootstrap with an average block size of 12.

```
import pandas as pd
from arch.bootstrap import StationaryBootstrap

bs = StationaryBootstrap(12, returns)
param_cov = bs.cov(sharpe_ratio)
index = ['mu', 'sigma', 'SR']
params = sharpe_ratio(returns)
params = pd.Series(params, index=index)
param_cov = pd.DataFrame(param_cov, index=index, columns=index)
```

The output is

```
>>> params
mu        8.148534
sigma    14.508540
SR        0.561637
dtype: float64

>>> param_cov
              mu     sigma        SR
mu     3.729435 -0.442891  0.273945
sigma -0.442891  0.495087 -0.049454
SR     0.273945 -0.049454  0.020830
```

Note: The covariance estimator is centered using the average of the bootstrapped estimators. The original sample estimator can be used to center using the keyword argument `recenter=False`.

1.2.4 Low-level Interfaces

Constructing Parameter Estimates

The bootstrap method apply can be use to directly compute parameter estimates from a function and the bootstrapped data.

This example makes use of monthly S&P 500 data.

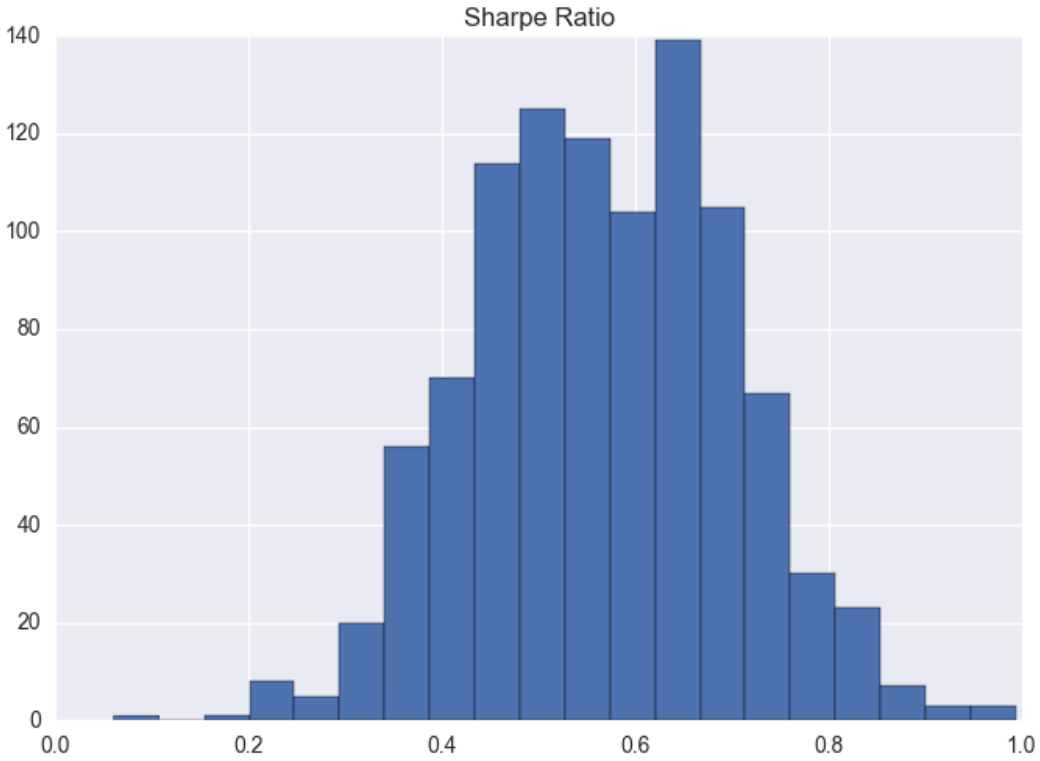
```
import datetime as dt

import pandas as pd
import pandas_datareader.data as web

start = dt.datetime(1951, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
low = sp500.index.min()
high = sp500.index.max()
monthly_dates = pd.date_range(low, high, freq='M')
monthly = sp500.reindex(monthly_dates, method='ffill')
returns = 100 * monthly['Adj Close'].pct_change().dropna()
```

The function will compute the Sharpe ratio – the (annualized) mean divided by the (annualized) standard deviation.

```
import numpy as np
def sharpe_ratio(x):
    return np.array([12 * x.mean() / np.sqrt(12 * x.var())])
```

The bootstrapped Sharpe ratios can be directly computed using *apply*.

```
import seaborn
from arch.bootstrap import IIDBootstrap
bs = IIDBootstrap(returns)
sharpe_ratios = bs.apply(sr, 1000)
sharpe_ratios = pd.DataFrame(sharp_ratios, columns=['Sharpe Ratio'])
sharpe_ratios.hist(bins=20)
```



The Bootstrap Iterator

The lowest-level method to use a bootstrap is the iterator. This is used internally in all higher-level methods that estimate a function using multiple bootstrap replications. The iterator returns a two-element tuple where the first element contains all positional arguments (in the order input) passed when constructing the bootstrap instance, and the second contains the all keyword arguments passed when constructing the instance.

This example makes uses of simulated data to demonstrate how to use the bootstrap iterator.

```
import pandas as pd
import numpy as np

from arch.bootstrap import IIDBootstrap

x = np.random.randn(1000, 2)
y = pd.DataFrame(np.random.randn(1000, 3))
z = np.random.rand(1000, 10)
bs = IIDBootstrap(x, y=y, z=z)

for pos, kw in bs.bootstrap(1000):
    xstar = pos[0]  # pos is always a tuple, even when a singleton
    ystar = kw['y']  # A dictionary
    zstar = kw['z']  # A dictionary
```

1.2.5 Semiparametric Bootstraps

Functions for semi-parametric bootstraps differ from those used in nonparametric bootstraps. At a minimum they must accept the keyword argument `params` which will contain the parameters estimated on the original (non-bootstrap) data. This keyword argument must be optional so that the function can be called without the keyword argument to estimate parameters. In most applications other inputs will also be needed to perform the semi-parametric step - these can be input using the `extra_kwargs` keyword input.

For simplicity, consider a semiparametric bootstrap of an OLS regression. The bootstrap step will combine the original parameter estimates and original regressors with bootstrapped residuals to construct a bootstrapped regressand. The bootstrap regressand and regressors can then be used to produce a bootstraped parameter estimate.

The user-provided function must:

- Estimate the parameters when `params` is not provided
- Estimate residuals from bootstrapped data when `params` is provided to construct bootstrapped residuals, simulate the regressand, and then estimate the bootstrapped parameters

```
import numpy as np
def ols(y, x, params=None, x_orig=None):
    if params is None:
        return np.linalg.pinv(x).dot(y)

    # When params is not None
    # Bootstrap residuals
    resids = y - x.dot(params)
    # Simulated data
    y_star = x_orig.dot(params) + resids
    # Parameter estimates
    return np.linalg.pinv(x_orig).dot(y_star)
```

This function can then be used to perform a semiparametric bootstrap

```
from arch.bootstrap import IIDBootstrap
x = np.random.randn(100,3)
e = np.random.randn(100,1)
b = np.arange(1,4)
y = x.dot(b) + e
bs = IIDBootstrap(y, x)
ci = bs.conf_int(ols, 1000, method='percentile',
                 sampling='semi', extra_kwargs={'x_orig': x})
```

Using `partial` instead of `extra_kwargs`

`functools.partial` can be used instead to provide a wrapper function which can then be used in the bootstrap. This example fixed the value of `x_orig` so that it is not necessary to use `extra_kwargs`.

```
from functools import partial
ols_partial = partial(ols, x_orig=x)
ci = bs.conf_int(ols_partial, 1000, sampling='semi')
```

Semiparametric Bootstrap (Alternative Method)

Since semiparametric bootstraps are effectively bootstrapping residuals, an alternative method can be used to conduct a semiparametric bootstrap. This requires passing both the data and the estimated residuals when intializing the

bootstrap.

First, the function used must be account for this structure.

```
def ols_semi_v2(y, x, resids=None, params=None, x_orig=None):
    if params is None:
        return np.linalg.pinv(x).dot(y)

    # Simulated data if params provided
    y_star = x_orig.dot(params) + resids
    # Parameter estimates
    return np.linalg.pinv(x_orig).dot(y_star)
```

This version can then be used to *directly* implement a semiparametric bootstrap, although ultimately it is not meaningfully simpler than the previous method.

```
resids = y - x.dot(ols_semi_v2(y,x))
bs = IIDBootstrap(y, x, resids=resids)
bs.conf_int(ols_semi_v2, 1000, sampling='semi', extra_kwargs={'x_orig': x})
```

Note: This alternative method is more useful when computing residuals is relatively expensive when compared to simulating data or estimating parameters. These circumstances are rarely encountered in actual problems.

1.2.6 Parametric Bootstraps

Parametric bootstraps are meaningfully different from their nonparametric or semiparametric cousins. Instead of sampling the data to simulate the data (or residuals, in the case of a semiparametric bootstrap), a parametric bootstrap makes use of a fully parametric model to simulate data using a pseudo-random number generator.

Warning: Parametric bootstraps are model-based methods to construct exact confidence intervals through integration. Since these confidence intervals should be exact, bootstrap methods which make use of asymptotic normality are required (and may not be desirable).

Implementing a parametric bootstrap, like implementing a semi-parametric bootstrap, requires specific keyword arguments. The first is `params`, which, when present, will contain the parameters estimated on the original data. The second is `rng` which will contain the `numpy.random.RandomState` instance that is used by the bootstrap. This is provided to facilitate simulation in a reproducible manner.

A parametric bootstrap function must:

- Estimate the parameters when `params` is not provided
- Simulate data when `params` is provided and then estimate the bootstrapped parameters on the simulated data

This example continues the OLS example from the semiparametric example, only assuming that residuals are normally distributed. The variance estimator is the MLE.

```
def ols_para(y, x, params=None, rng=None, x_orig=None):
    if params is None:
        beta = np.linalg.pinv(x).dot(y)
        e = y - x.dot(beta)
        sigma2 = e.dot(e) / e.shape[0]
        return np.hstack([beta,sigma2])
```

(continues on next page)

(continued from previous page)

```
    beta = params[:-1]
    sigma2 = params[-1]
    e = rng.standard_normal(x_orig.shape[0])
    ystar = x_orig.dot(params) + np.sqrt(sigma2) * e

    # Use the plain function to compute parameters
    return ols_para(ystar, x_orig)
```

This function can then be used to form parametric bootstrap confidence intervals.

```
bs = IIDBootstrap(y,x)
ci = bs.conf_int(ols_para, 1000, method='percentile',
                 sampling='parametric', extra_kwargs={'x_orig': x})
```

Note: The parameter vector in this example includes the variance since this is required when specifying a complete model.

1.2.7 Independent, Identical Distributed Data (i.i.d.)

IIDBootstrap is the standard bootstrap that is appropriate for data that is either i.i.d. or at least not serially dependant.

class arch.bootstrap.**IIDBootstrap**(**args*, ***kwargs*)

Bootstrap using uniform resampling

Parameters

- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

index

ndarray – The current index of the bootstrap

data

tuple – Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

pos_data

tuple – Tuple containing the positional arguments (in the order entered)

kw_data

dict – Dictionary containing the keyword arguments

random_state

RandomState – RandomState instance used by bootstrap

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import IIDBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500,2))
>>> z = standard_normal(500)
>>> bs = IIDBootstrap(x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

apply (*func*, *reps=1000*, *extra_kwargs=None*)
Applies a function to bootstrap replicated data

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns **results** – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type ndarray

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

bootstrap (*reps*)
Iterator for use when bootstrapping

Parameters **reps** (*int*) – Number of bootstrap replications

Returns **gen** – Generator to iterate over in bootstrap calculations

Return type generator

Example

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

conf_int (*func*, *reps=1000*, *method='basic'*, *size=0.95*, *tail='two'*, *extra_kwargs=None*, *reuse=False*, *sampling='nonparametric'*, *std_err_func=None*, *studentize_reps=1000*)

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **method** (*string, optional*) – One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'
- **size** (*float, optional*) – Coverage of confidence interval
- **tail** (*string, optional*) – One of 'two', 'upper' or 'lower'.
- **reuse** (*bool, optional*) – Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.
- **sampling** (*string, optional*) – Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
- **std_err_func** (*callable, optional*) – Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap

- **studentize_reps** (*int, optional*) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns **intervals** – Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type 2-d array

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

cov (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter covariance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications

- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.
- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **cov** – Bootstrap covariance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

get_state()
Gets the state of the bootstrap's random number generator

Returns **state** – Array containing the state

Return type RandomState state vector

reset(*use_seed=True*)
Resets the bootstrap to either its initial state or the last seed.

Parameters **use_seed** (*bool, optional*) – Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

seed (*value*)
Seeds the bootstrap's random number generator

Parameters **value** (*int*) – Integer to use as the seed

set_state (*state*)
Sets the state of the bootstrap's random number generator

Parameters **state** (*RandomState state vector*) – Array containing the state

var (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter variance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications
- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.
- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **var** – Bootstrap variance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

1.2.8 Time-series Bootstraps

Bootstraps for time-series data come in a variety of forms. The three contained in this package are the stationary bootstrap (*StationaryBootstrap*), which uses blocks with an exponentially distributed lengths, the circular block bootstrap (*CircularBlockBootstrap*), which uses fixed length blocks, and the moving block bootstrap which also uses fixed length blocks (*MovingBlockBootstrap*). The moving block bootstrap does *not* wrap around and so observations near the start or end of the series will be systematically under-sampled. It is not recommended for this reason.

The Stationary Bootstrap

class arch.bootstrap.**StationaryBootstrap**(*block_size*, **args*, ***kwargs*)

Politis and Romano (1994) bootstrap with expon. distributed block sizes

Parameters

- **block_size** (*int*) – Average size of block to use
- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

index

ndarray – The current index of the bootstrap

data

tuple – Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

pos_data

tuple – Tuple containing the positional arguments (in the order entered)

kw_data

dict – Dictionary containing the keyword arguments

random_state

RandomState – RandomState instance used by bootstrap

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import StationaryBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500,2))
>>> z = standard_normal(500)
>>> bs = StationaryBootstrap(12, x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

apply(*func*, *reps=1000*, *extra_kwargs=None*)
Applies a function to bootstrap replicated data

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns **results** – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type ndarray

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

bootstrap(*reps*)
Iterator for use when bootstrapping

Parameters **reps** (*int*) – Number of bootstrap replications

Returns **gen** – Generator to iterate over in bootstrap calculations

Return type generator

Example

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

conf_int (*func*, *reps=1000*, *method='basic'*, *size=0.95*, *tail='two'*, *extra_kwargs=None*, *reuse=False*, *sampling='nonparametric'*, *std_err_func=None*, *studentize_reps=1000*)

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **method** (*string, optional*) – One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'
- **size** (*float, optional*) – Coverage of confidence interval
- **tail** (*string, optional*) – One of 'two', 'upper' or 'lower'.
- **reuse** (*bool, optional*) – Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.
- **sampling** (*string, optional*) – Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
- **std_err_func** (*callable, optional*) – Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap

- **studentize_reps** (*int, optional*) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns **intervals** – Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type 2-d array

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

cov (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter covariance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications

- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.
- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **cov** – Bootstrap covariance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

get_state()
Gets the state of the bootstrap's random number generator

Returns **state** – Array containing the state

Return type RandomState state vector

reset(*use_seed=True*)
Resets the bootstrap to either its initial state or the last seed.

Parameters **use_seed** (*bool, optional*) – Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

seed (*value*)
Seeds the bootstrap's random number generator

Parameters **value** (*int*) – Integer to use as the seed

set_state (*state*)
Sets the state of the bootstrap's random number generator

Parameters **state** (*RandomState state vector*) – Array containing the state

var (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter variance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications
- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.
- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **var** – Bootstrap variance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

The Circular Block Bootstrap

class arch.bootstrap.**CircularBlockBootstrap**(*block_size*, **args*, ***kwargs*)

Bootstrap based on blocks of the same length with end-to-start wrap around

Parameters

- **block_size** (*int*) – Size of block to use
- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

index
ndarray – The current index of the bootstrap

data
tuple – Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

pos_data
tuple – Tuple containing the positional arguments (in the order entered)

kw_data
dict – Dictionary containing the keyword arguments

random_state
RandomState – RandomState instance used by bootstrap

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import CircularBlockBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500, 2))
```

(continues on next page)

(continued from previous page)

```
>>> z = standard_normal(500)
>>> bs = CircularBlockBootstrap(17, x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

apply(*func*, *reps=1000*, *extra_kwargs=None*)

Applies a function to bootstrap replicated data

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns **results** – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type ndarray

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

bootstrap(*reps*)

Iterator for use when bootstrapping

Parameters **reps** (*int*) – Number of bootstrap replications

Returns **gen** – Generator to iterate over in bootstrap calculations

Return type generator

Example

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

conf_int (*func*, *reps=1000*, *method='basic'*, *size=0.95*, *tail='two'*, *extra_kwargs=None*, *reuse=False*, *sampling='nonparametric'*, *std_err_func=None*, *studentize_reps=1000*)

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **method** (*string, optional*) – One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'
- **size** (*float, optional*) – Coverage of confidence interval
- **tail** (*string, optional*) – One of 'two', 'upper' or 'lower'.
- **reuse** (*bool, optional*) – Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.
- **sampling** (*string, optional*) – Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
- **std_err_func** (*callable, optional*) – Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap
- **studentize_reps** (*int, optional*) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns intervals – Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type 2-d array

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

cov (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter covariance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications
- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **cov** – Bootstrap covariance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

get_state()
Gets the state of the bootstrap's random number generator

Returns **state** – Array containing the state

Return type RandomState state vector

reset(*use_seed=True*)
Resets the bootstrap to either its initial state or the last seed.

Parameters **use_seed** (*bool, optional*) – Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

seed (*value*)
Seeds the bootstrap's random number generator

Parameters **value** (*int*) – Integer to use as the seed

set_state (*state*)
Sets the state of the bootstrap's random number generator

Parameters **state** (*RandomState state vector*) – Array containing the state

var (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter variance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications
- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.
- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **var** – Bootstrap variance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
```

(continues on next page)

(continued from previous page)

```
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

The Moving Block Bootstrap

class arch.bootstrap.**MovingBlockBootstrap**(*block_size*, **args*, ***kwargs*)

Bootstrap based on blocks of the same length without wrap around

Parameters

- **block_size** (*int*) – Size of block to use
- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

index
ndarray – The current index of the bootstrap

data
tuple – Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

pos_data
tuple – Tuple containing the positional arguments (in the order entered)

kw_data
dict – Dictionary containing the keyword arguments

random_state
RandomState – RandomState instance used by bootstrap

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import MovingBlockBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500,2))
>>> z = standard_normal(500)
>>> bs = MovingBlockBootstrap(7, x, y=y, z=z)
```

(continues on next page)

(continued from previous page)

```
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

apply(*func*, *reps=1000*, *extra_kwargs=None*)
Applies a function to bootstrap replicated data

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns **results** – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type ndarray

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

bootstrap(*reps*)
Iterator for use when bootstrapping

Parameters **reps** (*int*) – Number of bootstrap replications

Returns **gen** – Generator to iterate over in bootstrap calculations

Return type generator

Example

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

conf_int(*func*, *reps=1000*, *method='basic'*, *size=0.95*, *tail='two'*, *extra_kwargs=None*, *reuse=False*, *sampling='nonparametric'*, *std_err_func=None*, *studentize_reps=1000*)

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **method** (*string, optional*) – One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'
- **size** (*float, optional*) – Coverage of confidence interval
- **tail** (*string, optional*) – One of 'two', 'upper' or 'lower'.
- **reuse** (*bool, optional*) – Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.
- **sampling** (*string, optional*) – Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.
- **extra_kwargs** (*dict, optional*) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
- **std_err_func** (*callable, optional*) – Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap
- **studentize_reps** (*int, optional*) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns **intervals** – Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type 2-d array

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

cov (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter covariance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications
- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.
- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **cov** – Bootstrap covariance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

get_state()
Gets the state of the bootstrap's random number generator

Returns **state** – Array containing the state

Return type RandomState state vector

reset(*use_seed=True*)
Resets the bootstrap to either its initial state or the last seed.

Parameters **use_seed** (*bool, optional*) – Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

seed(*value*)
Seeds the bootstrap's random number generator

Parameters **value** (*int*) – Integer to use as the seed

set_state(*state*)
Sets the state of the bootstrap's random number generator

Parameters **state** (*RandomState state vector*) – Array containing the state

var (*func*, *reps=1000*, *recenter=True*, *extra_kwargs=None*)
Compute parameter variance using bootstrap

Parameters

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications
- **recenter** (*bool, optional*) – Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.
- **extra_kwargs** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

Returns **var** – Bootstrap variance estimator

Return type ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Example

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note: Note this is a generic example and so the class used should be the name of the required bootstrap

1.2.9 References

The bootstrap is a large area with a number of high-quality books. Leading references include

References

Articles used in the creation of this module include

1.3 Multiple Comparison Procedures

This module contains a set of bootstrap-based multiple comparison procedures. These are designed to allow multiple models to be compared while controlling a the Familywise Error Rate, which is similar to the size of a test.

1.3.1 Multiple Comparisons

This setup code is required to run in an IPython notebook

```
In [1]: import warnings
        warnings.simplefilter('ignore')

        %matplotlib inline
        import seaborn
        seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
        seaborn.mpl.rcParams['savefig.dpi'] = 90
        seaborn.mpl.rcParams['font.family'] = 'serif'
        seaborn.mpl.rcParams['font.size'] = 14

        # Reproducability
        import numpy as np
        np.random.seed(23456)
        # Common seed used throughout
        seed = np.random.randint(0,2**31-1)
```

The multiple comparison procedures all allow for examining aspects of superior predictive ability. There are three available:

- `SPA` - The test of Superior Predictive Ability, also known as the Reality Check (and accessible as `RealityCheck`) or the bootstrap data snooper, examines whether any model in a set of models can outperform a benchmark.
- `StepM` - The stepwise multiple testing procedure uses sequential testing to determine which models are superior to a benchmark.
- `MCS` - The model confidence set which computes the set of models which with performance indistinguishable from others in the set.

All procedures take **losses** as inputs. That is, smaller values are preferred to larger values. This is common when evaluating forecasting models where the loss function is usually defined as a positive function of the forecast error that is increasing in the absolute error. Leading examples are Mean Square Error (MSE) and Mean Absolute Deviation (MAD).

The test of Superior Predictive Ability (SPA)

This procedure requires a t-element array of benchmark losses and a t by k-element array of model losses. The null hypothesis is that no model is better than the benchmark, or

$$H_0 : \max_i E[L_i] \geq E[L_{bm}]$$

where L_i is the loss from model i and L_{bm} is the loss from the benchmark model.

This procedure is normally used when there are many competing forecasting models such as in the study of technical trading rules. The example below will make use of a set of models which are all equivalently good to a benchmark model and will serve as a *size study*.

Study Design

The study will make use of a measurement error in predictors to produce a large set of correlated variables that all have equal expected MSE. The benchmark will have identical measurement error and so all models have the same expected loss, although will have different forecasts.

The first block computed the series to be forecast.

```
In [2]: from numpy.random import randn
        import statsmodels.api as sm
        t = 1000
        factors = randn(t,3)
        beta = np.array([1,0.5,0.1])
        e = randn(t)
        y = factors.dot(beta)
```

The next block computes the benchmark factors and the model factors by contaminating the original factors with noise. The models are estimated on the first 500 observations and predictions are made for the second 500. Finally, losses are constructed from these predictions.

```
In [3]: # Measurement noise
        bm_factors = factors + randn(t,3)
        # Fit using first half, predict second half
        bm_beta = sm.OLS(y[:500],bm_factors[:500]).fit().params
        # MSE loss
        bm_losses = (y[500:] - bm_factors[500:].dot(bm_beta))**2.0
        # Number of models
        k = 500
        model_factors = np.zeros((k,t,3))
        model_losses = np.zeros((500,k))
        for i in range(k):
            # Add measurement noise
            model_factors[i] = factors + randn(1000,3)
            # Compute regression parameters
            model_beta  = sm.OLS(y[:500],model_factors[i,:500]).fit().params
            # Prediction and losses
            model_losses[:,i] = (y[500:] - model_factors[i,500:].dot(model_beta))**2.0
```

Finally the SPA can be used. The SPA requires the **losses** from the benchmark and the models as inputs. Other inputs allow the bootstrap sued to be changed or for various options regarding studentization of the losses. `compute` does the real work, and then `pvalues` contains the probability that the null is true given the realizations.

In this case, one would not reject. The three p-values correspond to different re-centerings of the losses. In general, the `consistent` p-value should be used. It should always be the case that

$$lower \leq consistent \leq upper.$$

See the original papers for more details.

```
In [4]: from arch.bootstrap import SPA
        spa = SPA(bm_losses, model_losses)
        spa.seed(seed)
        spa.compute()
        spa.pvalues
```

```
Out[4]: lower         0.520
        consistent    0.723
        upper         0.733
        dtype: float64
```

The same blocks can be repeated to perform a simulation study. Here I only use 100 replications since this should complete in a reasonable amount of time. Also I set `reps=250` to limit the number of bootstrap replications in each application of the SPA (the default is a more reasonable 1000).

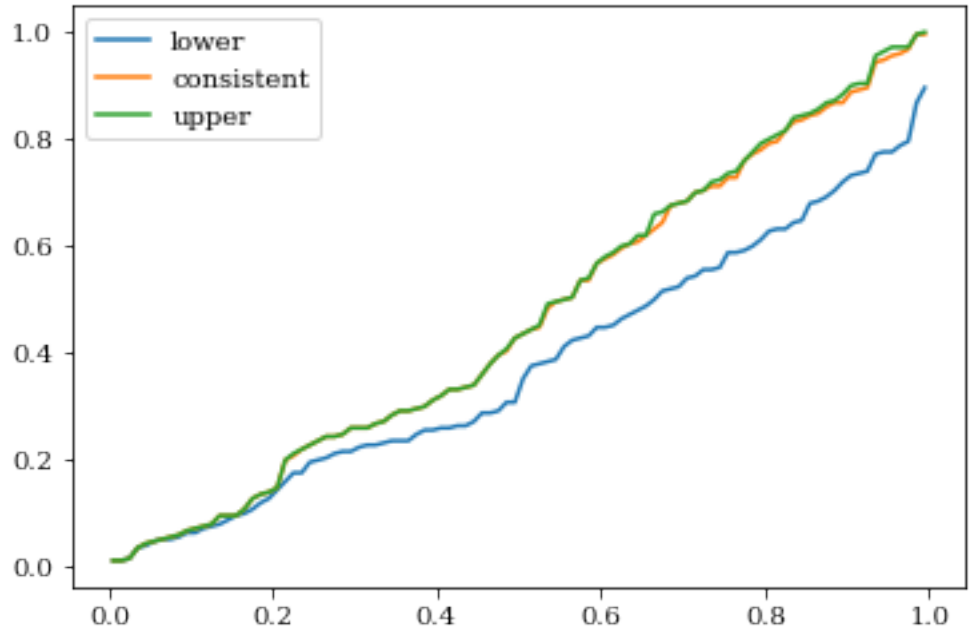
```
In [5]: # Save the pvalues
        pvalues = []
        b = 100
        seeds = np.random.randint(0, 2**31 - 1, b)
        # Repeat 100 times
        for j in range(b):
            if j % 10 == 0:
                print(j)
            factors = randn(t,3)
            beta = np.array([1,0.5,0.1])
            e = randn(t)
            y = factors.dot(beta)


            # Measurement noise
            bm_factors = factors + randn(t,3)
            # Fit using first half, predict second half
            bm_beta = sm.OLS(y[:500],bm_factors[:500]).fit().params
            # MSE loss
            bm_losses = (y[500:] - bm_factors[500:].dot(bm_beta))**2.0
            # Number of models
            k = 500
            model_factors = np.zeros((k,t,3))
            model_losses = np.zeros((500,k))
            for i in range(k):
                model_factors[i] = factors + randn(1000,3)
                model_beta  = sm.OLS(y[:500],model_factors[i,:500]).fit().params
                # MSE loss
                model_losses[:,i] = (y[500:] - model_factors[i,500:].dot(model_beta))**2.0
            # Lower the bootstrap replications to 250
            spa = SPA(bm_losses, model_losses, reps = 250)
            spa.seed(seeds[j])
            spa.compute()
            pvalues.append(spa.pvalues)
```

```
0
10
20
30
```

```
40
50
60
70
80
90
```

Finally the pvalues can be plotted. Ideally they should form a 45^o line indicating the size is correct. Both the consistent and upper perform well. The lower has too many small p-values.

```
In [6]: import pandas as pd

        pvalues = pd.DataFrame(pvalues)
        for col in pvalues:
            values = pvalues[col].values
            values.sort()
            pvalues[col] = values
        # Change the index so that the x-values are between 0 and 1
        pvalues.index = np.linspace(0.005,.995,100)
        fig = pvalues.plot()
```



Power

The SPA also has power to reject then the null is violated. The simulation will be modified so that the amount of measurement error differs across models, and so that some models are actually better than the benchmark. The p-values should be small indicating rejection of the null.

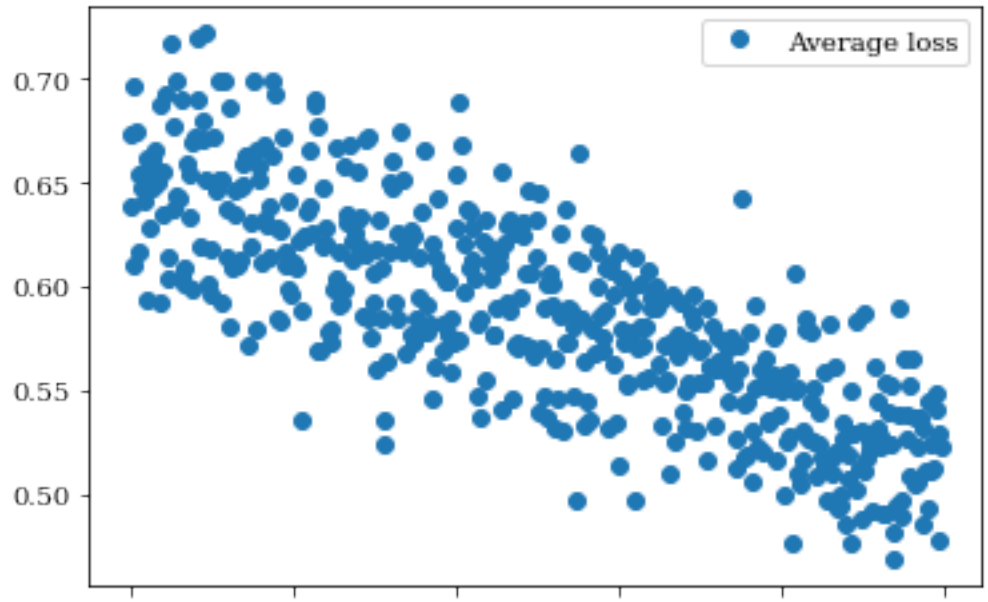
```
In [7]: # Number of models
        k = 500
        model_factors = np.zeros((k,t,3))
        model_losses = np.zeros((500,k))
        for i in range(k):
            scale = ((2500.0 - i) / 2500.0)
            model_factors[i] = factors + scale * randn(1000,3)
            model_beta  = sm.OLS(y[:500],model_factors[i,:500]).fit().params
```

```
            # MSE loss
            model_losses[:,i] = (y[500:] - model_factors[i,500:].dot(model_beta))**2.0

        spa = SPA(bm_losses, model_losses)
        spa.seed(seed)
        spa.compute()
        spa.pvalues
```

Out[7]:
```
lower          0.0
consistent     0.0
upper          0.0
dtype: float64
```

Here the average losses are plotted. The higher index models are clearly better than the lower index models – and the benchmark model (which is identical to model.0).

In [8]:
```
model_losses = pd.DataFrame(model_losses,columns=['model.' + str(i) for i in range(k)])
avg_model_losses = pd.DataFrame(model_losses.mean(0), columns=['Average loss'])
fig = avg_model_losses.plot(style=['o'])
```



Stepwise Multiple Testing (StepM)

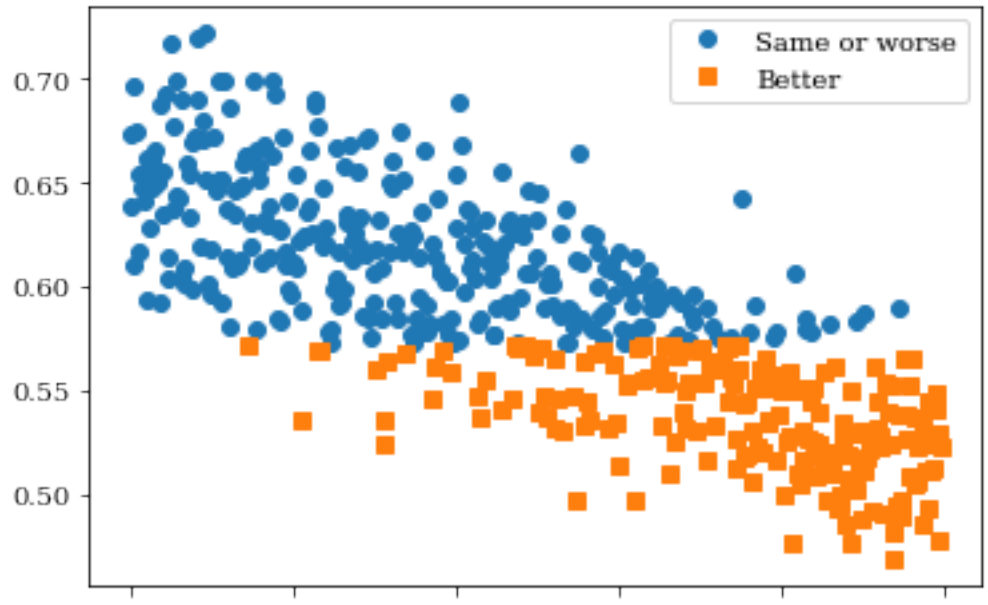
Stepwise Multiple Testing is similar to the SPA and has the same null. The primary difference is that it identifies the set of models which are better than the benchmark, rather than just asking the basic question if any model is better.

In [9]:
```
from arch.bootstrap import StepM
stepm = StepM(bm_losses, model_losses)
stepm.compute()
print('Model indices:')
print([model.split('.')[1] for model in stepm.superior_models])
```

```
Model indices:
['106', '115', '117', '152', '156', '157', '158', '169', '186', '187', '192', '197', '214', '215', '2
```

In [10]:
```
better_models = pd.concat([model_losses.mean(0),model_losses.mean(0)],1)
better_models.columns = ['Same or worse','Better']
better = better_models.index.isin(stepm.superior_models)
```

```
worse = np.logical_not(better)
better_models.loc[better,'Same or worse'] = np.nan
better_models.loc[worse,'Better'] = np.nan
fig = better_models.plot(style=['o','s'], rot=270)
```



The Model Confidence Set

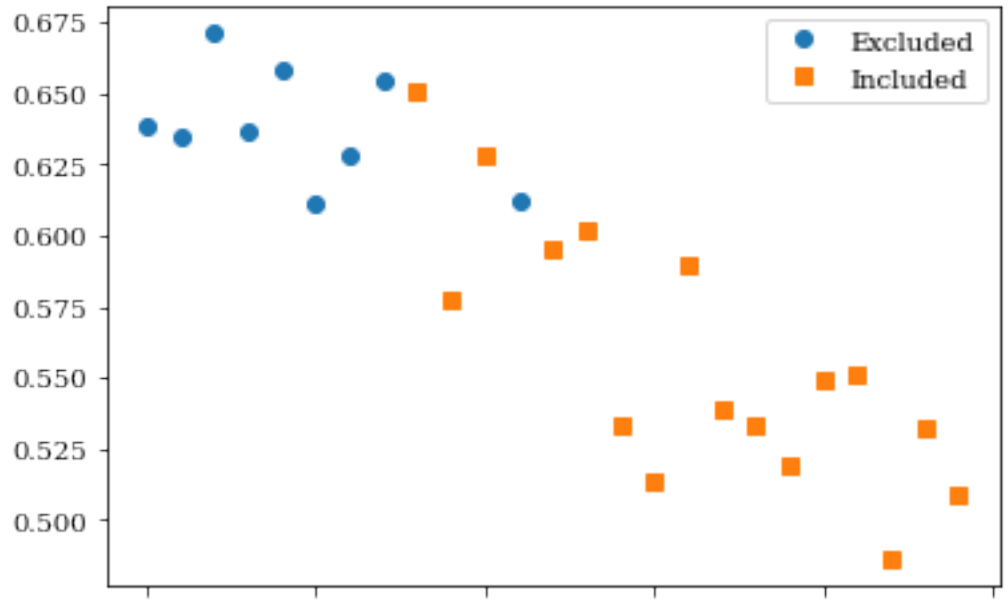
The model confidence set takes a set of **losses** as its input and finds the set which are not statistically different from each other while controlling the familywise error rate. The primary output is a set of p-values, where models with a pvalue above the size are in the MCS. Small p-values indicate that the model is easily rejected from the set that includes the best.

```
In [11]: from arch.bootstrap import MCS
         # Limit the size of the set
         losses = model_losses.iloc[:,::20]
         mcs = MCS(losses, size=0.10)
         mcs.compute()
         print('MCS P-values')
         print(mcs.pvalues)
         print('Included')
         included = mcs.included
         print([model.split('.')[1] for model in included])
         print('Excluded')
         excluded = mcs.excluded
         print([model.split('.')[1] for model in excluded])
```

```
MCS P-values
            Pvalue
Model name
model.60     0.000
model.80     0.001
model.40     0.002
model.140    0.002
model.20     0.005
model.100    0.005
model.120    0.013
```

```
model.0      0.013
model.220    0.025
model.160    0.128
model.240    0.130
model.260    0.146
model.200    0.146
model.180    0.403
model.320    0.460
model.420    0.483
model.400    0.717
model.360    0.866
model.340    0.866
model.280    0.866
model.460    0.866
model.380    0.866
model.300    0.866
model.480    0.866
model.440    1.000
Included
['160', '180', '200', '240', '260', '280', '300', '320', '340', '360', '380', '400', '420', '440', '4
Excluded
['0', '100', '120', '140', '20', '220', '40', '60', '80']
```

```
In [12]: status = pd.DataFrame([losses.mean(0),losses.mean(0)], index=['Excluded','Included']).T
         status.loc[status.index.isin(included), 'Excluded'] = np.nan
         status.loc[status.index.isin(excluded), 'Included'] = np.nan
         fig = status.plot(style=['o','s'])
```



1.3.2 Module Reference

Test of Superior Predictive Ability (SPA), Reality Check

The test of Superior Predictive Ability (Hansen 2005), or SPA, is an improved version of the Reality Check (White 2000). It tests whether the best forecasting performance from a set of models is better than that of the forecasts from a benchmark model. A model is "better" if its losses are smaller than those from the benchmark. Formally, it tests the

null

$$H_0 : \max_i E[L_i] \geq E[L_{bm}]$$

where L_i is the loss from model *i* and L_{bm} is the loss from the benchmark model. The alternative is

$$H_1 : \min_i E[L_i] < E[L_{bm}]$$

This procedure accounts for dependence between the losses and the fact that there are potentially alternative models being considered.

Note: Also callable using `RealityCheck`

class arch.bootstrap.**SPA**(*benchmark*, *models*, *block_size=None*, *reps=1000*, *bootstrap='stationary'*, *studentize=True*, *nested=False*)

Implementation of the Test of Superior Predictive Ability (SPA), which is also known as the Reality Check or Bootstrap Data Snooper.

Parameters

- **benchmark** (*{ndarray, Series}*) – T element array of benchmark model *losses*
- **models** (*{ndarray, DataFrame}*) – T by k element array of alternative model *losses*
- **block_size** (*int, optional*) – Length of window to use in the bootstrap. If not provided, sqrt(T) is used. In general, this should be provided and chosen to be appropriate for the data.
- **reps** (*int, optional*) – Number of bootstrap replications to uses. Default is 1000.
- **bootstrap** (*str, optional*) – Bootstrap to use. Options are 'stationary' or 'sb': Stationary bootstrap (Default) 'circular' or 'cbb': Circular block bootstrap 'moving block' or 'mbb': Moving block bootstrap
- **studentize** (*bool*) – Flag indicating to studentize loss differentials. Default is True
- **nested=False** – Flag indicating to use a nested bootstrap to compute variances for studentization. Default is False. Note that this can be slow since the procedure requires k extra bootstraps.

compute()
Compute the bootstrap pvalue. Must be called before accessing the pvalue

seed()
Pass seed to bootstrap implementation

reset()
Reset the bootstrap to its initial state

better_models()
Produce a list of column indices or names (if models is a DataFrame) that are rejected given a test size

References

White, H. (2000). "A reality check for data snooping." Econometrica 68, no. 5, 1097-1126.

Hansen, P. R. (2005). "A test for superior predictive ability." Journal of Business & Economic Statistics, 23(4)

Notes

The three p-value correspond to different re-centering decisions.

- Upper : Never recenter to all models are relevant to distribution
- Consistent : Only recenter if closer than a log(log(t)) bound
- Lower : Never recenter a model if worse than benchmark

See also:

StepM

compute()
Compute the bootstrap p-value

critical_values(*pvalue=0.05*)
Returns data-dependent critical values

Parameters **pvalue** (*float, optional*) – P-value in (0,1) to use when computing the critical values.

Returns **crit_vals** – Series containing critical values for the lower, consistent and upper methodologies

Return type Series

pvalues
P-values corresponding to the lower, consistent and upper p-values.

Returns **pvals** – Three p-values corresponding to the lower bound, the consistent estimator, and the upper bound.

Return type Series

Stepwise Multiple Testing (StepM)

The Stepwise Multiple Testing procedure (Romano & Wolf (2005)) is closely related to the SPA, except that it returns a set of models that are superior to the benchmark model, rather than the p-value from the null. They are so closely related that *StepM* is essentially a wrapper around *SPA* with some small modifications to allow multiple calls.

class arch.bootstrap.**StepM**(*benchmark*, *models*, *size=0.05*, *block_size=None*, *reps=1000*, *bootstrap='stationary'*, *studentize=True*, *nested=False*)
Implementation of Romano and Wolf's StepM multiple comparison procedure

Parameters

- **benchmark** (*{ndarray, Series}*) – T element array of benchmark model *losses*
- **models** (*{ndarray, DataFrame}*) – T by k element array of alternative model *losses*
- **size** (*float, optional*) – Value in (0,1) to use as the test size when implementing the comparison. Default value is 0.05.
- **block_size** (*int, optional*) – Length of window to use in the bootstrap. If not provided, sqrt(T) is used. In general, this should be provided and chosen to be appropriate for the data.
- **reps** (*int, optional*) – Number of bootstrap replications to uses. Default is 1000.

- **bootstrap** (*str, optional*) – Bootstrap to use. Options are 'stationary' or 'sb': Stationary bootstrap (Default) 'circular' or 'cbb': Circular block bootstrap 'moving block' or 'mbb': Moving block bootstrap
- **studentize** (*bool, optional*) – Flag indicating to studentize loss differentials. Default is True
- **nested** (*bool, optional*) – Flag indicating to use a nested bootstrap to compute variances for studentization. Default is False. Note that this can be slow since the procedure requires k extra bootstraps.

compute()
Compute the set of superior models.

References

Romano, J. P., & Wolf, M. (2005). "Stepwise multiple testing as formalized data snooping." Econometrica, 73(4), 1237-1282.

Notes

The size controls the Family Wise Error Rate (FWER) since this is a multiple comparison procedure. Uses SPA and the consistent selection procedure.

See also:

SPA

compute()
Computes the set of superior models

superior_models
List of the indices or column names of the superior models

Returns **superior_models** – List of superior models. Contains column indices if models is an array or contains column names if models is a DataFrame.

Return type list

Model Confidence Set (MCS)

The Model Confidence Set (Hansen, Lunde & Nason (2011)) differs from other multiple comparison procedures in that there is no benchmark. The MCS attempts to identify the set of models which produce the same expected loss, while controlling the probability that a model that is worse than the best model is in the model confidence set. Like the other MCPs, it controls the Familywise Error Rate rather than the usual test size.

class arch.bootstrap.**MCS**(*losses*, *size*, *reps=1000*, *block_size=None*, *method='R'*, *bootstrap='stationary'*)
Implementation of the Model Confidence Set (MCS)

Parameters

- **losses** (*{ndarray, DataFrame}*) – T by k array containing losses from a set of models
- **size** (*float, optional*) – Value in (0,1) to use as the test size when implementing the mcs. Default value is 0.05.

- **block_size** (*int, optional*) – Length of window to use in the bootstrap. If not provided, sqrt(T) is used. In general, this should be provided and chosen to be appropriate for the data.
- **method** (*{'max', 'R'}, optional*) – MCS test and elimination implementation method, either 'max' or 'R'. Default is 'R'.
- **reps** (*int, optional*) – Number of bootstrap replications to uses. Default is 1000.
- **bootstrap** (*str, optional*) – Bootstrap to use. Options are 'stationary' or 'sb': Stationary bootstrap (Default) 'circular' or 'cbb': Circular block bootstrap 'moving block' or 'mbb': Moving block bootstrap

compute()

References

Hansen, P. R., Lunde, A., & Nason, J. M. (2011). The model confidence set. Econometrica, 79(2), 453-497.

compute()
Computes the model confidence set

1.3.3 References

Articles used in the creation of this module include

1.4 Unit Root Testing

Many time series are highly persistent, and determining whether the data appear to be stationary or contains a unit root is the first step in many analyses. This module contains a number of routines:

- Augmented Dickey-Fuller (*ADF*)
- Dickey-Fuller GLS (*DFGLS*)
- Phillips-Perron (*PhillipsPerron*)
- Variance Ratio (*VarianceRatio*)
- KPSS (*KPSS*)

The first four all start with the null of a unit root and have an alternative of a stationary process. The final test, KPSS, has a null of a stationary process with an alternative of a unit root.

1.4.1 Introduction

All tests expect a 1-d series as the first input. The input can be any array that can *squeeze* into a 1-d array, a pandas *Series* or a pandas *DataFrame* that contains a single variable.

All tests share a common structure. The key elements are:

- *stat* - Returns the test statistic
- *pvalue* - Returns the p-value of the test statistic
- *lags* - Sets or gets the number of lags used in the model. In most test, can be *None* to trigger automatic selection.
- ***trend* - Sets of gets the trend used in the model. Supported trends vary by model, but include:**

  - *'nc'*: No constant
  - *'c'*: Constant
  - *'ct'*: Constant and time trend
  - *'ctt'*: Constant, time trend and quadratic time trend
- *summary()* - Returns a summary object that can be printed to get a formatted table

Basic Example

This basic example show the use of the Augmented-Dickey fuller to test whether the default premium, defined as the difference between the yields of large portfolios of BAA and AAA bonds. This example uses a constant and time trend.

```
import datetime as dt

import pandas_datareader.data as web
from arch.unitroot import ADF

start = dt.datetime(1919, 1, 1)
end = dt.datetime(2014, 1, 1)

df = web.DataReader(["AAA", "BAA"], "fred", start, end)
df['diff'] = df['BAA'] - df['AAA']
adf = ADF(df['diff'])
adf.trend = 'ct'

print(adf.summary())
```

which yields

```
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -3.448
P-value                         0.045
Lags                               21
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -3.97 (1%), -3.41 (5%), -3.13 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

1.4.2 Unit Root Testing

This setup code is required to run in an IPython notebook

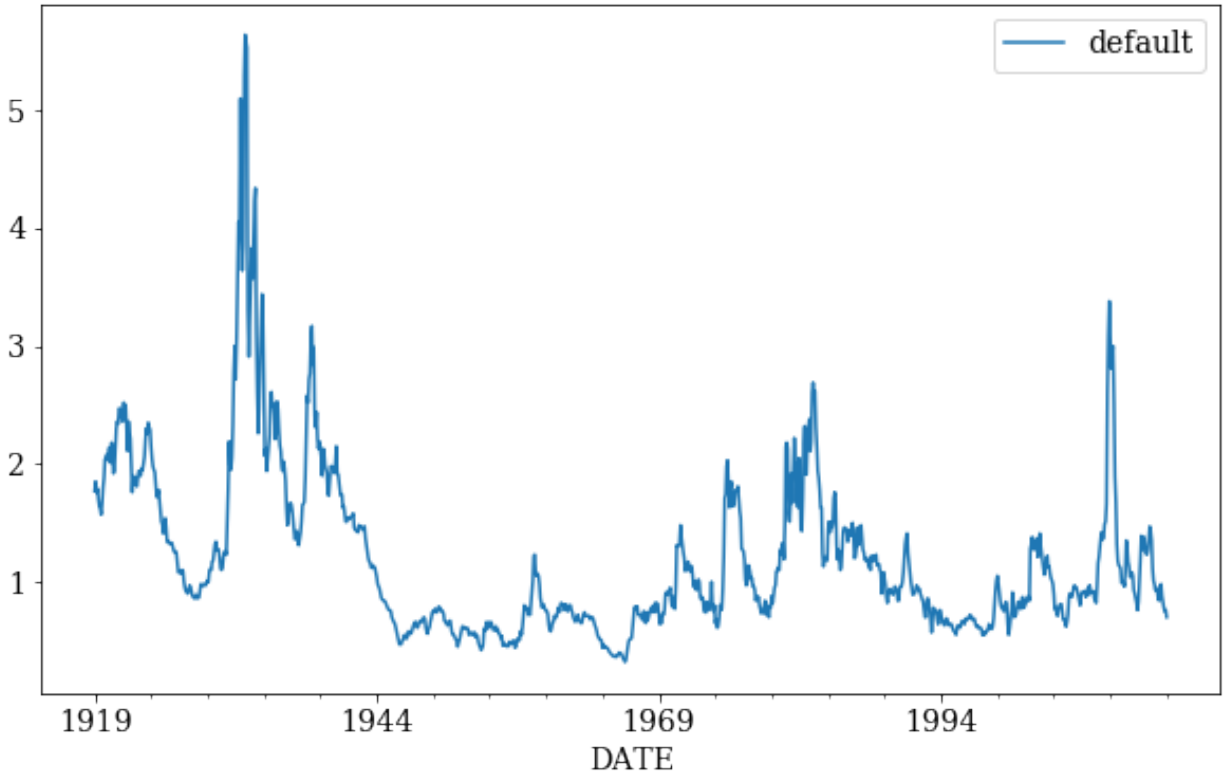
```
In [1]: import warnings
        warnings.simplefilter('ignore')

        %matplotlib inline
        import seaborn

In [2]: seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
        seaborn.mpl.rcParams['savefig.dpi'] = 90
```

```
seaborn.mpl.rcParams['font.family'] = 'serif'
seaborn.mpl.rcParams['font.size'] = 14
```

Setup

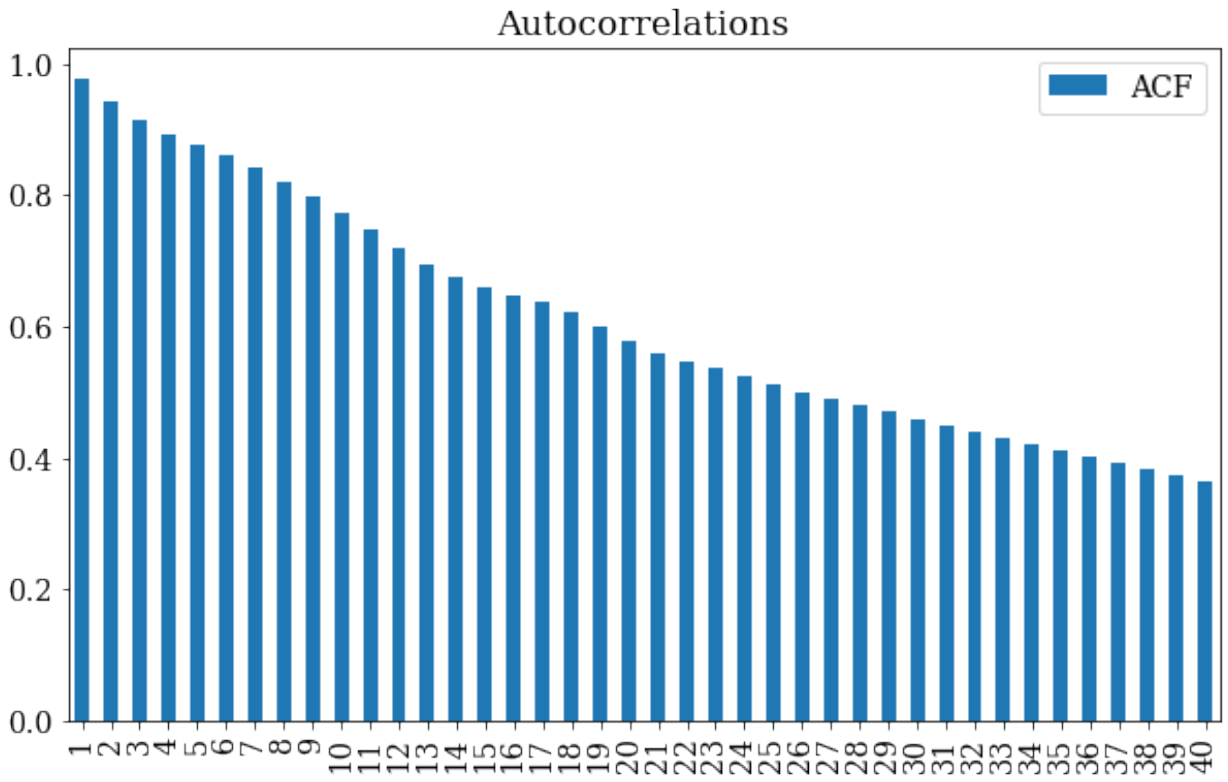
Most examples will make use of the Default premium, which is the difference between the yields of BAA and AAA rated corporate bonds. The data is downloaded from FRED using pandas.

```
In [3]: import datetime as dt
        import pandas as pd
        import statsmodels.api as sm
        import pandas_datareader.data as web
        aaa = web.DataReader("AAA", "fred", dt.datetime(1919,1,1), dt.datetime(2014,1,1))
        baa = web.DataReader("BAA", "fred", dt.datetime(1919,1,1), dt.datetime(2014,1,1))
        baa.columns = aaa.columns = ['default']
        default = baa - aaa
        fig = default.plot()
```



The Default premium is clearly highly persistent. A simple check of the autocorrelations confirms this.

```
In [4]: acf = pd.DataFrame(sm.tsa.stattools.acf(default), columns=['ACF'])
        fig = acf[1:].plot(kind='bar', title='Autocorrelations')
```



Augmented Dickey-Fuller Testing

The Augmented Dickey-Fuller test is the most common unit root test used. It is a regression of the first difference of the variable on its lagged level as well as additional lags of the first difference. The null is that the series contains a unit root, and the (one-sided) alternative is that the series is stationary.

By default, the number of lags is selected by minimizing the AIC across a range of lag lengths (which can be set using `max_lag` when initializing the model). Additionally, the basic test includes a constant in the ADF regression.

These results indicate that the Default premium is stationary.

```
In [5]: from arch.unitroot import ADF
        adf = ADF(default)
        print(adf.summary().as_text())
```

```
   Augmented Dickey-Fuller Results
====================================
Test Statistic                -3.241
P-value                        0.018
Lags                              21
------------------------------------

Trend: Constant
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

The number of lags can be directly set using `lags`. Changing the number of lags makes no difference to the conclusion.

Note: The ADF assumes residuals are white noise, and that the number of lags is sufficient to pick up any dependence in the data.

Setting the number of lags

```
In [6]: adf.lags = 5
        print(adf.summary().as_text())
```

```
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -3.427
P-value                         0.010
Lags                                5
-------------------------------------

Trend: Constant
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Deterministic terms

The deterministic terms can be altered using `trend`. The options are:

- `'nc'` : No deterministic terms
- `'c'` : Constant only
- `'ct'` : Constant and time trend
- `'ctt'` : Constant, time trend and time-trend squared

Changing the type of constant also makes no difference for this data.

```
In [7]: adf.trend = 'ct'
        print(adf.summary().as_text())
```

```
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -3.661
P-value                         0.025
Lags                                5
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -3.97 (1%), -3.41 (5%), -3.13 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Regression output

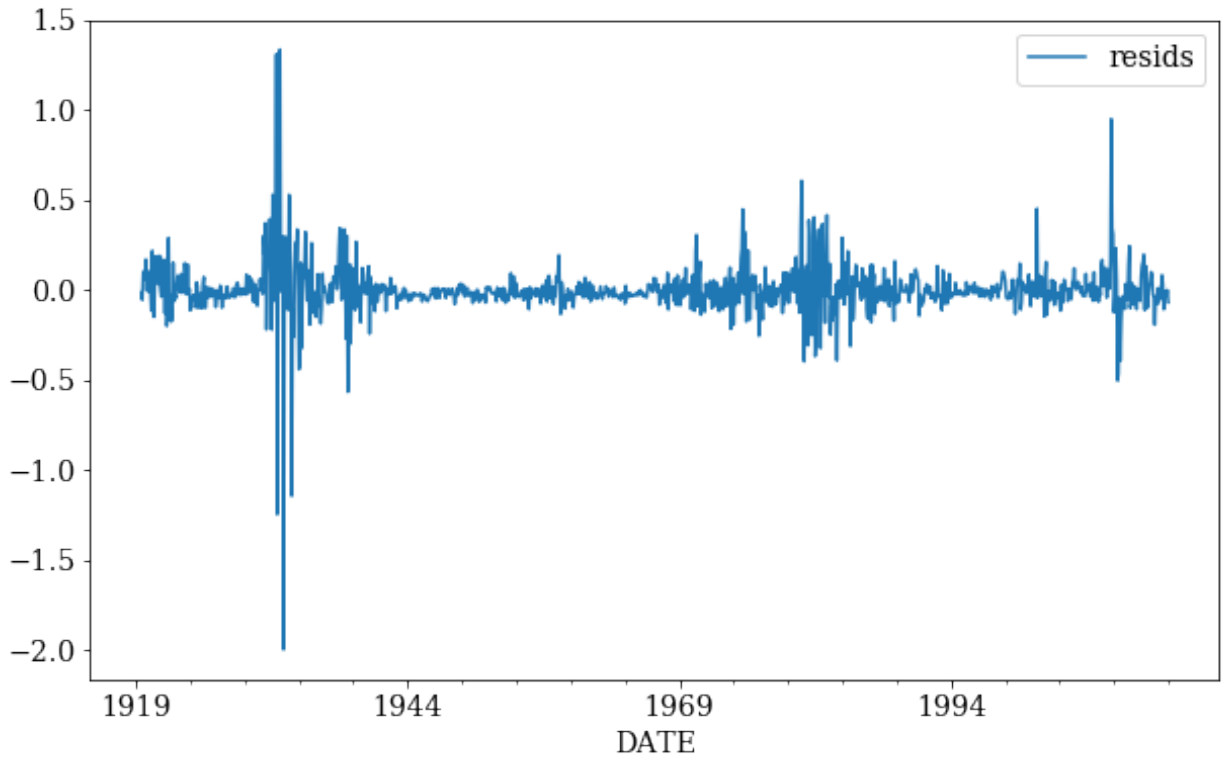
The ADF uses a standard regression when computing results. These can be accesses using `regression`.

```
In [8]: reg_res = adf.regression
        print(reg_res.summary().as_text())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.095
```

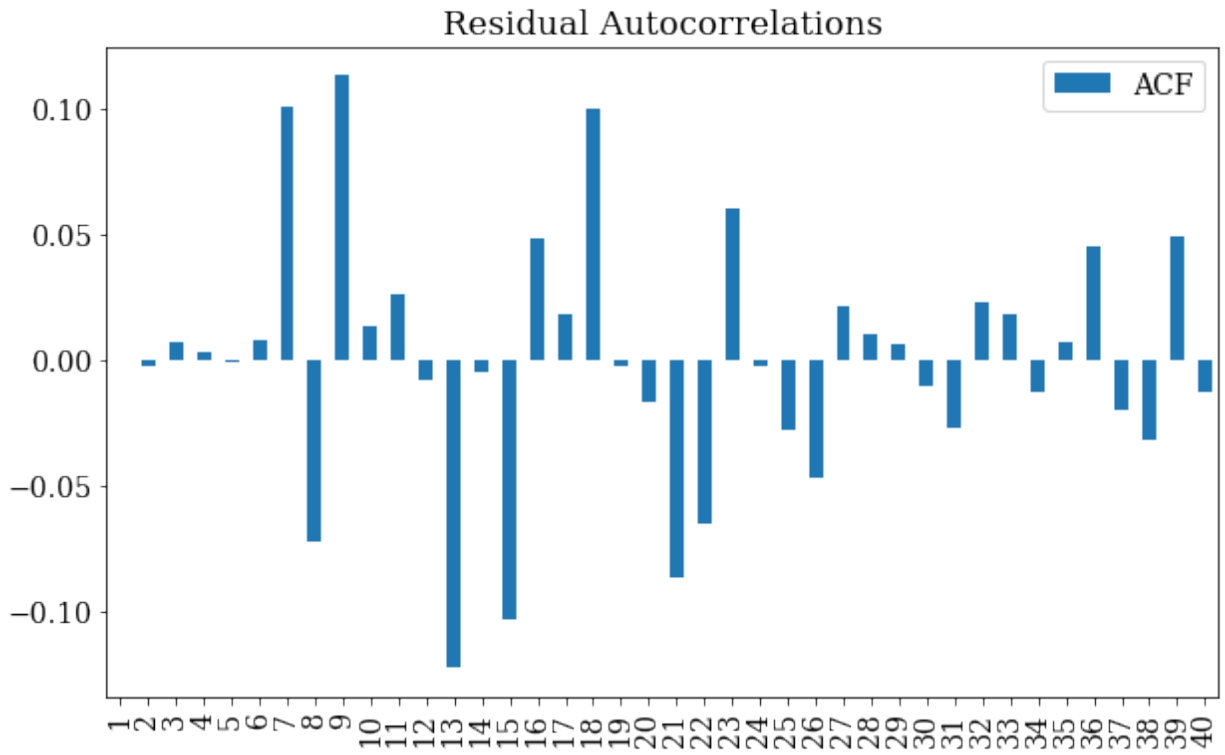
```
Model:                            OLS   Adj. R-squared:                  0.090
Method:                 Least Squares   F-statistic:                     17.00
Date:                Thu, 27 Sep 2018   Prob (F-statistic):           1.84e-21
Time:                        15:31:57   Log-Likelihood:                 575.20
No. Observations:                1135   AIC:                            -1134.
Df Residuals:                    1127   BIC:                            -1094.
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Level.L1      -0.0246      0.007     -3.661      0.000      -0.038      -0.011
Diff.L1        0.2190      0.030      7.344      0.000       0.160       0.277
Diff.L2       -0.0549      0.030     -1.802      0.072      -0.115       0.005
Diff.L3       -0.1398      0.030     -4.649      0.000      -0.199      -0.081
Diff.L4       -0.0519      0.030     -1.716      0.086      -0.111       0.007
Diff.L5        0.0426      0.030      1.431      0.153      -0.016       0.101
const          0.0389      0.014      2.796      0.005       0.012       0.066
trend      -1.817e-05   1.41e-05     -1.285      0.199   -4.59e-05    9.58e-06
==============================================================================
Omnibus:                      623.301   Durbin-Watson:                   2.001
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           128496.439
Skew:                          -1.393   Prob(JB):                         0.00
Kurtosis:                      55.051   Cond. No.                     5.30e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 5.3e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [9]:
```
import pandas as pd
import matplotlib.pyplot as plt
resids = pd.DataFrame(reg_res.resid)
resids.index = default.index[6:]
resids.columns=['resids']
fig = resids.plot()
```



Since the number lags was directly set, it is good to check whether the residuals appear to be white noise.

```
In [10]: acf = pd.DataFrame(sm.tsa.stattools.acf(reg_res.resid), columns=['ACF'])
         fig = acf[1:].plot(kind='bar', title='Residual Autocorrelations')
```



Dickey-Fuller GLS Testing

The Dickey-Fuller GLS test is an improved version of the ADF which uses a GLS-detrending regression before running an ADF regression with no additional deterministic terms. This test is only available with a constant or constant and time trend (`trend='c'` or `trend='ct'`).

The results of this test agree with the ADF results.

```
In [11]: from arch.unitroot import DFGLS
         dfgls = DFGLS(default)
         print(dfgls.summary().as_text())
```

```
      Dickey-Fuller GLS Results
====================================
Test Statistic                 -2.252
P-value                         0.024
Lags                               21
------------------------------------

Trend: Constant
Critical Values: -2.59 (1%), -1.96 (5%), -1.64 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

The trend can be altered using `trend`. The conclusion is the same.

```
In [12]: dfgls.trend = 'ct'
         print(dfgls.summary().as_text())
```

```
      Dickey-Fuller GLS Results
====================================
Test Statistic                 -3.406
P-value                         0.010
Lags                               21
------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -3.43 (1%), -2.86 (5%), -2.58 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Phillips-Perron Testing

The Phillips-Perron test is similar to the ADF except that the regression run does not include lagged values of the first differences. Instead, the PP test fixed the t-statistic using a long run variance estimation, implemented using a Newey-West covariance estimator.

By default, the number of lags is automatically set, although this can be overridden using `lags`.

```
In [13]: from arch.unitroot import PhillipsPerron
         pp = PhillipsPerron(default)
         print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-tau)
====================================
Test Statistic                 -3.761
P-value                         0.003
Lags                               23
------------------------------------

Trend: Constant
```

```
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

It is important that the number of lags is sufficient to pick up any dependence in the data.

```
In [14]: pp.lags = 12
         print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-tau)
=====================================
Test Statistic                 -3.876
P-value                         0.002
Lags                               12
-------------------------------------

Trend: Constant
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

The trend can be changed as well.

```
In [15]: pp.trend = 'ct'
         print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-tau)
=====================================
Test Statistic                 -4.142
P-value                         0.005
Lags                               12
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -3.97 (1%), -3.41 (5%), -3.13 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Finally, the PP testing framework includes two types of tests. One which uses an ADF-type regression of the first difference on the level, the other which regresses the level on the level. The default is the `tau` test, which is similar to an ADF regression, although this can be changed using `test_type='rho'`.

```
In [16]: pp.test_type = 'rho'
         print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-rho)
=====================================
Test Statistic                -33.974
P-value                         0.000
Lags                               12
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -29.15 (1%), -21.60 (5%), -18.16 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

KPSS Testing

The KPSS test differs from the three previous in that the null is a stationary process and the alternative is a unit root.

Note that here the null is rejected which indicates that the series might be a unit root.

```
In [17]: from arch.unitroot import KPSS
         kpss = KPSS(default)
         print(kpss.summary().as_text())
```

```
    KPSS Stationarity Test Results
=====================================
Test Statistic                  0.941
P-value                         0.003
Lags                               23
-------------------------------------

Trend: Constant
Critical Values: 0.74 (1%), 0.46 (5%), 0.35 (10%)
Null Hypothesis: The process is weakly stationary.
Alternative Hypothesis: The process contains a unit root.
```

Changing the trend does not alter the conclusion.

```
In [18]: kpss.trend = 'ct'
         print(kpss.summary().as_text())
```

```
    KPSS Stationarity Test Results
=====================================
Test Statistic                  0.361
P-value                         0.000
Lags                               23
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: 0.22 (1%), 0.15 (5%), 0.12 (10%)
Null Hypothesis: The process is weakly stationary.
Alternative Hypothesis: The process contains a unit root.
```

Variance Ratio Testing

Variance ratio tests are not usually used as unit root tests, and are instead used for testing whether a financial return series is a pure random walk versus having some predictability. This example uses the excess return on the market from Ken French's data.

```
In [19]: import numpy as np
         import pandas as pd
         import pandas_datareader as web
         try:
             ff=web.DataReader('F-F_Research_Data_Factors', 'famafrench')
         except:
             ff=web.DataReader('F-F_Research_Data_Factors_TXT', 'famafrench')
         ff = ff[0]
         excess_market = ff.iloc[:,0]  # Excess Market
         print(ff.describe())
```

```
            Mkt-RF         SMB         HML          RF
count  103.000000  103.000000  103.000000  103.000000
mean     1.139806    0.107379   -0.131068    0.020680
std      3.542553    2.273547    2.162328    0.037397
min     -7.890000   -4.370000   -4.500000    0.000000
25%     -0.985000   -1.720000   -1.530000    0.000000
50%      1.180000    0.320000   -0.290000    0.010000
75%      3.155000    1.230000    1.015000    0.020000
max     11.350000    5.490000    8.270000    0.160000
```

The variance ratio compares the variance of a 1-period return to that of a multi-period return. The comparison length has to be set when initializing the test.

This example compares 1-month to 12-month returns, and the null that the series is a pure random walk is rejected. Negative values indicate some positive autocorrelation in the returns (momentum).

```
In [20]: from arch.unitroot import VarianceRatio
         vr = VarianceRatio(excess_market, 12)
         print(vr.summary().as_text())
```

```
     Variance-Ratio Test Results
=====================================
Test Statistic                 -4.555
P-value                         0.000
Lags                               12
-------------------------------------

Computed with overlapping blocks (de-biased)
```

By default the VR test uses all overlapping blocks to estimate the variance of the long period's return. This can be changed by setting `overlap=False`. This lowers the power but doesn't change the conclusion.

```
In [21]: warnings.simplefilter('always')  # Restore warnings

         vr.overlap = False
         print(vr.summary().as_text())
```

```
     Variance-Ratio Test Results
=====================================
Test Statistic                 -1.944
P-value                         0.052
Lags                               12
-------------------------------------

Computed with non-overlapping blocks

c:\git\arch\arch\unitroot\unitroot.py:1228: InvalidLengthWarning:
The length of y is not an exact multiple of 12, and so the final
6 observations have been dropped.

  InvalidLengthWarning)
```

Note: The warning is intentional. It appears here since when it is not possible to use all data since the data length isn't an integer multiple of the long period when using non-overlapping blocks. There is little reason to use `overlap=False`.

1.4.3 The Unit Root Tests

Augmented-Dickey Fuller Testing

class arch.unitroot.**ADF**(*y*, *lags=None*, *trend='c'*, *max_lags=None*, *method='AIC'*, *low_memory=None*)

Augmented Dickey-Fuller unit root test

Parameters

- **y** (*{ndarray, Series}*) – The data to test for a unit root
- **lags** (*int, optional*) – The number of lags to use in the ADF regression. If omitted or None, *method* is used to automatically select the lag length with no more than *max_lags* are included.

- **trend** (*{'nc', 'c', 'ct', 'ctt'}, optional*) – The trend component to include in the ADF test 'nc' - No trend components 'c' - Include a constant (Default) 'ct' - Include a constant and linear time trend 'ctt' - Include a constant and linear and quadratic time trends
- **max_lags** (*int, optional*) – The maximum number of lags to use when selecting lag length
- **method** (*{'AIC', 'BIC', 't-stat'}, optional*) – The method to use when selecting the lag length 'AIC' - Select the minimum of the Akaike IC 'BIC' - Select the minimum of the Schwarz/Bayesian IC 't-stat' - Select the minimum of the Schwarz/Bayesian IC
- **low_memory** (*bool*) – Flag indicating whether to use a low memory implementation of the lag selection algorithm. The low memory algorithm is slower than the standard algorithm but will use 2-4% of the memory required for the standard algorithm. This options allows automatic lag selection to be used in very long time series. If None, use automatic selection of algorithm.

stat

pvalue

critical_values

null_hypothesis

alternative_hypothesis

summary

regression

valid_trends

y

trend

lags

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 the null cannot be rejected that there and the series appears to be a unit root.

The p-values are obtained through regression surface approximation from MacKinnon (1994) using the updated 2010 tables. If the p-value is close to significant, then the critical values should be used to judge whether to reject the null.

The autolag option and maxlag for it are described in Greene.

Examples

```
>>> from arch.unitroot import ADF
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
```

(continues on next page)

(continued from previous page)

```
>>> inflation = np.diff(np.log(data['cpi']))
>>> adf = ADF(inflation)
>>> print('{0:0.4f}'.format(adf.stat))
-3.0931
>>> print('{0:0.4f}'.format(adf.pvalue))
0.0271
>>> adf.lags
2
>>> adf.trend='ct'
>>> print('{0:0.4f}'.format(adf.stat))
-3.2111
>>> print('{0:0.4f}'.format(adf.pvalue))
0.0822
```

References

Greene, W. H. 2011. Econometric Analysis. Prentice Hall: Upper Saddle River, New Jersey.

Hamilton, J. D. 1994. Time Series Analysis. Princeton: Princeton University Press.

P-Values (regression surface approximation) MacKinnon, J.G. 1994. "Approximate asymptotic distribution functions for unit-root and cointegration bootstrap. *Journal of Business and Economic Statistics* 12, 167-76.

Critical values MacKinnon, J.G. 2010. "Critical Values for Cointegration Tests." Queen's University, Dept of Economics, Working Papers. Available at http://ideas.repec.org/p/qed/wpaper/1227.html

alternative_hypothesis
: The alternative hypothesis

critical_values
: Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

lags
: Sets or gets the number of lags used in the model. When bootstrap use DF-type regressions, lags is the number of lags in the regression model. When bootstrap use long-run variance estimators, lags is the number of lags used in the long-run variance estimator.

null_hypothesis
: The null hypothesis

pvalue
: Returns the p-value for the test statistic

regression
: Returns the OLS regression results from the ADF model estimated

stat
: The test statistic for a unit root

summary()
: Summary of test, containing statistic, p-value and critical values

trend
: Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

valid_trends
: List of valid trend terms.

Dickey-Fuller GLS Testing

class arch.unitroot.**DFGLS**(*y*, *lags=None*, *trend='c'*, *max_lags=None*, *method='AIC'*, *low_memory=None*)

Elliott, Rothenberg and Stock's GLS version of the Dickey-Fuller test

Parameters

- **y** (*{ndarray, Series}*) – The data to test for a unit root
- **lags** (*int, optional*) – The number of lags to use in the ADF regression. If omitted or None, *method* is used to automatically select the lag length with no more than *max_lags* are included.
- **trend** (*{'c', 'ct'}, optional*) – The trend component to include in the ADF test 'c' - Include a constant (Default) 'ct' - Include a constant and linear time trend
- **max_lags** (*int, optional*) – The maximum number of lags to use when selecting lag length
- **method** (*{'AIC', 'BIC', 't-stat'}, optional*) – The method to use when selecting the lag length 'AIC' - Select the minimum of the Akaike IC 'BIC' - Select the minimum of the Schwarz/Bayesian IC 't-stat' - Select the minimum of the Schwarz/Bayesian IC

stat

pvalue

critical_values

null_hypothesis

alternative_hypothesis

summary

regression

valid_trends

y

trend

lags

Notes

The null hypothesis of the Dickey-Fuller GLS 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 the null cannot be rejected and the series appears to be a unit root.

DFGLS differs from the ADF test in that an initial GLS detrending step is used before a trend-less ADF regression is run.

Critical values and p-values when trend is 'c' are identical to the ADF. When trend is set to 'ct, they are from ...

Examples

```
>>> from arch.unitroot import DFGLS
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
>>> inflation = np.diff(np.log(data['cpi']))
>>> dfgls = DFGLS(inflation)
>>> print('{0:0.4f}'.format(dfgls.stat))
-2.7611
>>> print('{0:0.4f}'.format(dfgls.pvalue))
0.0059
>>> dfgls.lags
2
>>> dfgls.trend = 'ct'
>>> print('{0:0.4f}'.format(dfgls.stat))
-2.9036
>>> print('{0:0.4f}'.format(dfgls.pvalue))
0.0447
```

References

Elliott, G. R., T. J. Rothenberg, and J. H. Stock. 1996. Efficient bootstrap for an autoregressive unit root. Econometrica 64: 813-836

summary()
Summary of test, containing statistic, p-value and critical values

Phillips-Perron Testing

class arch.unitroot.**PhillipsPerron**(*y*, *lags=None*, *trend='c'*, *test_type='tau'*)
Phillips-Perron unit root test

Parameters

- **y** (*{ndarray, Series}*) – The data to test for a unit root
- **lags** (*int, optional*) – The number of lags to use in the Newey-West estimator of the long-run covariance. If omitted or None, the lag length is set automatically to 12 * (nobs/100) ** (1/4)
- **trend** (*{'nc', 'c', 'ct'}, optional*) –

  The trend component to include in the ADF test 'nc' - No trend components 'c' - Include a constant (Default) 'ct' - Include a constant and linear time trend
- **test_type** (*{'tau', 'rho'}*) – The test to use when computing the test statistic. 'tau' is based on the t-stat and 'rho' uses a test based on nobs times the re-centered regression coefficient

stat

pvalue

critical_values

test_type

null_hypothesis

alternative_hypothesis

summary

valid_trends

y

trend

lags

Notes

The null hypothesis of the Phillips-Perron (PP) test 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 the null cannot be rejected that there and the series appears to be a unit root.

Unlike the ADF test, the regression estimated includes only one lag of the dependant variable, in addition to trend terms. Any serial correlation in the regression errors is accounted for using a long-run variance estimator (currently Newey-West).

The p-values are obtained through regression surface approximation from MacKinnon (1994) using the updated 2010 tables. If the p-value is close to significant, then the critical values should be used to judge whether to reject the null.

Examples

```
>>> from arch.unitroot import PhillipsPerron
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
>>> inflation = np.diff(np.log(data['cpi']))
>>> pp = PhillipsPerron(inflation)
>>> print('{0:0.4f}'.format(pp.stat))
-8.1356
>>> print('{0:0.4f}'.format(pp.pvalue))
0.0000
>>> pp.lags
15
>>> pp.trend = 'ct'
>>> print('{0:0.4f}'.format(pp.stat))
-8.2022
>>> print('{0:0.4f}'.format(pp.pvalue))
0.0000
>>> pp.test_type = 'rho'
>>> print('{0:0.4f}'.format(pp.stat))
-120.3271
>>> print('{0:0.4f}'.format(pp.pvalue))
0.0000
```

References

Hamilton, J. D. 1994. Time Series Analysis. Princeton: Princeton University Press.

Newey, W. K., and K. D. West. 1987. A simple, positive semidefinite, heteroskedasticity and autocorrelation consistent covariance matrix. Econometrica 55, 703-708.

Phillips, P. C. B., and P. Perron. 1988. Testing for a unit root in time series regression. Biometrika 75, 335-346.

P-Values (regression surface approximation) MacKinnon, J.G. 1994. "Approximate asymptotic distribution functions for unit-root and cointegration bootstrap. *Journal of Business and Economic Statistics* 12, 167-76.

Critical values MacKinnon, J.G. 2010. "Critical Values for Cointegration Tests." Queen's University, Dept of Economics, Working Papers. Available at http://ideas.repec.org/p/qed/wpaper/1227.html

summary()
Summary of test, containing statistic, p-value and critical values

Variance Ratios

class arch.unitroot.**VarianceRatio**(*y*, *lags=2*, *trend='c'*, *debiased=True*, *robust=True*, *overlap=True*)
Variance Ratio test of a random walk.

Parameters

- **y** (*{ndarray, Series}*) – The data to test for a random walk
- **lags** (*int*) – The number of periods to used in the multi-period variance, which is the numerator of the test statistic. Must be at least 2
- **trend** (*{'nc', 'c'}, optional*) – 'c' allows for a non-zero drift in the random walk, while 'nc' requires that the increments to y are mean 0
- **overlap** (*bool, optional*) – Indicates whether to use all overlapping blocks. Default is True. If False, the number of observations in y minus 1 must be an exact multiple of lags. If this condition is not satisfied, some values at the end of y will be discarded.
- **robust** (*bool, optional*) – Indicates whether to use heteroskedasticity robust inference. Default is True.
- **debiased** (*bool, optional*) – Indicates whether to use a debiased version of the test. Default is True. Only applicable if overlap is True.

stat

pvalue

critical_values

null_hypothesis

alternative_hypothesis

summary

valid_trends

y

trend

lags

overlap

robust

debiased

Notes

The null hypothesis of a VR is that the process is a random walk, possibly plus drift. Rejection of the null with a positive test statistic indicates the presence of positive serial correlation in the time series.

Examples

```
>>> from arch.unitroot import VarianceRatio
>>> import datetime as dt
>>> import pandas_datareader as pdr
>>> data = pdr.get_data_fred('DJIA')
>>> data = data.resample('M').last()  # End of month
>>> returns = data['DJIA'].pct_change().dropna()
>>> vr = VarianceRatio(returns, lags=12)
>>> print('{0:0.4f}'.format(vr.pvalue))
0.0000
```

References

Campbell, John Y., Lo, Andrew W. and MacKinlay, A. Craig. (1997) The Econometrics of Financial Markets. Princeton, NJ: Princeton University Press.

summary()
Summary of test, containing statistic, p-value and critical values

KPSS Testing

class arch.unitroot.**KPSS**(*y*, *lags=None*, *trend='c'*)
Kwiatkowski, Phillips, Schmidt and Shin (KPSS) stationarity test

Parameters

- **y** (*{ndarray, Series}*) – The data to test for stationarity
- **lags** (*int, optional*) – The number of lags to use in the Newey-West estimator of the long-run covariance. If omitted or None, the lag length is set automatically to 12 * (nobs/100) ** (1/4)
- **trend** (*{'c', 'ct'}, optional*) –

  The trend component to include in the ADF test 'c' - Include a constant (Default) 'ct' - Include a constant and linear time trend

stat

pvalue

critical_values

null_hypothesis

alternative_hypothesis

summary

valid_trends

y

trend

lags

Notes

The null hypothesis of the KPSS test is that the series is weakly stationary and the alternative is that it is non-stationary. If the p-value is above a critical size, then the null cannot be rejected that there and the series appears stationary.

The p-values and critical values were computed using an extensive simulation based on 100,000,000 replications using series with 2,000 observations.

Examples

```
>>> from arch.unitroot import KPSS
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
>>> inflation = np.diff(np.log(data['cpi']))
>>> kpss = KPSS(inflation)
>>> print('{0:0.4f}'.format(kpss.stat))
0.2870
>>> print('{0:0.4f}'.format(kpss.pvalue))
0.1474
>>> kpss.trend = 'ct'
>>> print('{0:0.4f}'.format(kpss.stat))
0.2075
>>> print('{0:0.4f}'.format(kpss.pvalue))
0.0128
```

References

Kwiatkowski, D.; Phillips, P. C. B.; Schmidt, P.; Shin, Y. (1992). "Testing the null hypothesis of stationarity against the alternative of a unit root". Journal of Econometrics 54 (1-3), 159-178

summary()
Summary of test, containing statistic, p-value and critical values

1.5 Change Logs

1.5.1 Changes since 4.0

- Added support for Fractionally Integrated GARCH (FIGARCH)
- Enable user to specify a specific value of the *backcast* in place of the automatically generated value.
- Fixed a big where parameter-less models where incorrectly reported as having constant variance (GH248)
- Added support for MIDAS volatility processes using Hyperbolic weighting (GH233)
- Added a parameter to forecast that allows a user-provided callable random generator to be used in place of the model random generator. (GH225)

- Added a low memory automatic lag selection method that can be used with very large time-series.
- Improved performance of automatic lag selection in ADF and related tests.
- Added named parameters to Dickey-Fuller regressions.
- Removed use of the module-level NumPy RandomState. All random number generators use separate RandomState instances.
- Fixed a bug that prevented 1-step forecasts with exogenous regressors
- Added the Generalized Error Distribution for univariate ARCH models
- Fixed a bug in MCS when using the max method that prevented all included models from being listed
- Added `FixedVariance` volatility process which allows pre-specified variances to be used with a mean model. This has been added to allow so-called zig-zag estimation where a mean model is estimated with a fixed variance, and then a variance model is estimated on the residuals using a `ZeroMean` variance process.
- Fixed a bug that prevented `fix` from being used with a new model (GH156)
- Added `first_obs` and `last_obs` parameters to `fix` to mimic `fit`
- Added ability to jointly estimate smoothing parameter in EWMA variance when fitting the model
- Added ability to pass optimization options to ARCH model estimation (GH195)

1.5.2 Changes since 3.0

- Added forecast code for mean forecasting
- Added volatility hedgehog plot
- Added `fix` to arch models which allows for user specified parameters instead of estimated parameters.
- Added Hansen's Skew T distribution to distribution (Stanislav Khrapov)
- Updated IPython notebooks to latest IPython version
- Bug and typo fixes to IPython notebooks
- Changed MCS to give a pvalue of 1.0 to best model. Previously was NaN
- Removed `hold_back` and `last_obs` from model initialization and to `fit` method to simplify estimating a model over alternative samples (e.g., rolling window estimation)
- Redefined `hold_back` to only accept integers so that is simply defined the number of observations held back. This number is now held out of the sample irrespective of the value of `first_obs`.

1.5.3 Changes since 2.1

- Added multiple comparison procedures
- Typographical and other small changes

1.5.4 Changes since 2.0

- Add unit root tests: * Augmented Dickey-Fuller * Dickey-Fuller GLS * Phillips-Perron * KPSS * Variance Ratio
- Removed deprecated locations for ARCH modeling functions

1.5.5 Changes since 1.0

- Refactored to move the univariate routines to *arch.univariate* and added deprecation warnings in the old locations
- Enable *numba* jit compilation in the python recursions
- Added a bootstrap framework, which will be used in future versions. The bootstrap framework is general purpose and can be used via high-level functions such as *conf_int* or *cov*, or as a low level iterator using *bootstrap*

CHAPTER 2

Citation

This package should be cited using Zenodo. For example, for the 4.6 release,

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[Chernick] Chernick, M. R. (2011). *Bootstrap methods: A guide for practitioners and researchers* (Vol. 619). John Wiley & Sons.

[Davidson] Davison, A. C. (1997). *Bootstrap methods and their application* (Vol. 1). Cambridge university press.

[EfronTibshirani] Efron, B., & Tibshirani, R. J. (1994). *An introduction to the bootstrap* (Vol. 57). CRC press.

[PolitisRomanoWolf] Politis, D. N., & Romano, J. P. M. Wolf, 1999. *Subsampling*.

[CarpenterBithell] Carpenter, J., & Bithell, J. (2000). "Bootstrap confidence intervals: when, which, what? A practical guide for medical statisticians." *Statistics in medicine*, 19(9), 1141-1164.

[DavidsonMacKinnon] Davidson, R., & MacKinnon, J. G. (2006). "Bootstrap methods in econometrics." *Palgrave Handbook of Econometrics*, 1, 812-38.

[DiCiccioEfron] DiCiccio, T. J., & Efron, B. (1996). "Bootstrap confidence intervals." *Statistical Science*, 189-212.

[Efron] Efron, B. (1987). "Better bootstrap confidence intervals." *Journal of the American statistical Association*, 82(397), 171-185.

[Hansen] Hansen, P. R. (2005). A test for superior predictive ability. *Journal of Business & Economic Statistics*, 23(4).

[HansenLundeNason] Hansen, P. R., Lunde, A., & Nason, J. M. (2011). The model confidence set. *Econometrica*, 79(2), 453-497.

[RomanoWolf] Romano, J. P., & Wolf, M. (2005). Stepwise multiple testing as formalized data snooping. *Econometrica*, 73(4), 1237-1282.

[White] White, H. (2000). A reality check for data snooping. *Econometrica*, 68(5), 1097-1126.

Python Module Index

a

Index

D

E

F

G

H

S

T

V

Y

Z