PyFlux Documentation

Release 0.4.7

Ross Taylor

Dec 16, 2018

Contents

CHAPTER 1

What is PyFlux?

PyFlux is a library for time series analysis and prediction. Users can choose from a flexible range of modelling and inference options, and use the output for forecasting and retrospection. Users can build a full probabilistic model where the data y and latent variables (parameters) z are treated as random variables through a joint probability $p(y, z)$. The advantage of a probabilistic approach is that it gives a more complete picture of uncertainty, which is important for time series tasks such as forecasting. Alternatively, for speed, users can simply use Maximum Likelihood estimation for speed within the same unified API.

CHAPTER 2

Installation

The latest release version of PyFlux is available on PyPi. Python 2.7 and Python 3.5 are supported, but development occurs primarily on 3.5. To install `pyflux`, simply call `pip`:

```
pip install pyflux
```

PyFlux requires a number of dependencies, in particular `numpy`, `pandas`, `scipy`, `patsy`, `matplotlib`, `numdifftools` and `seaborn`.

The development code can be accessed on GitHub. The project is open source, and contributions are welcome and encouraged. We also encourage you to check out other modelling libraries written in Python including `pymc3`, `edward` and `statsmodels`.

CHAPTER 3

Application Interface

The PyFlux API is designed to be as clear and concise as possible, meaning it takes a minimal number of steps to conduct the model building process. The high-level outline is detailed below.

The first step is to **create a model instance**, where the main arguments are (i) a data input, such as a pandas dataframe, (ii) design parameters, such as autoregressive lags for an ARIMA model, and (iii) a family, which specifies the distribution of the modelled time series, such as a Normal distribution.

```
my_model = pf.ARIMA(data=my_dataframe, ar=2, ma=0, family=pf.Normal())
```

The second step is **prior formation**, which involves specifying a family for each latent variable in the model using the `adjust_prior` method, for example we can a prior for the constant in the ARIMA model $N(0, 10)$. The latent variables can be viewed by printing the `latent_variables` object attached to the model. Prior formation be ignored if the user is intending to just do Maximum Likelihood.

```
print(my_model.latent_variables)

Index    Latent Variable     Prior           Prior Hyperparameters     V.I. Dist␣
↪Transform
======== ================== =============== ========================= ==========␣
↪==========
0        Constant            Normal          mu0: 0, sigma0: 3         Normal     None
1        AR(1)               Normal          mu0: 0, sigma0: 0.5       Normal     None
2        AR(2)               Normal          mu0: 0, sigma0: 0.5       Normal     None
3        Normal Scale        Flat            n/a (noninformative)      Normal     exp

my_model.adjust_prior(0, pf.Normal(0, 10))

print(my_model.latent_variables)

Index    Latent Variable     Prior           Prior Hyperparameters     V.I. Dist␣
↪Transform
======== ================== =============== ========================= ==========␣
↪==========
0        Constant            Normal          mu0: 0, sigma0: 10        Normal     None
```

(continues on next page)

(continued from previous page)

```
1        AR(1)              Normal          mu0: 0, sigma0: 0.5       Normal      None
2        AR(2)              Normal          mu0: 0, sigma0: 0.5       Normal      None
3        Normal Scale       Flat            n/a (noninformative)      Normal      exp
```

The third step is **model fitting (or inference)**, which involves using a `fit` method, specifying an inference option. Current options include Maximum Likelihood (MLE), Metropolis-Hastings (M-H), and black box variational inference (BBVI). Once complete, the model latent variable information will be updated, and the user can proceed to the post fitting methods.

```
x = my_model.fit('M-H')
Tuning complete! Now sampling.
Acceptance rate of Metropolis-Hastings is 0.2915

x.summary()
Normal ARIMA(2,0,0)
======================================␣
↪================================================
Dependent Variable: sunspot.year         Method: Metropolis Hastings
Start Date: 1702                         Unnormalized Log Posterior: -1219.7028
End Date: 1988                           AIC: 2447.40563132
Number of observations: 287              BIC: 2462.04356018
=============================================================================================
Latent Variable           Median             Mean               95% Credibility␣
↪Interval
========================= ================== ==================␣
↪=========================
Constant                  14.6129            14.5537            (11.8099 | 17.1807)
AR(1)                     1.3790             1.3796             (1.3105 | 1.4517)
AR(2)                     -0.6762            -0.6774            (-0.7484 | -0.6072)
Normal Scale              16.6720            16.6551            (15.5171 | 17.8696)
=============================================================================================
```

```
my_model.plot_z(figsize=(15, 7))
```

The fourth step is **model evaluation**, **retrospection** and **prediction**. Once the model has been fit, the user can look at historical fit, criticize with posterior predictive checks, predict out of sample, and perform a range of other tasks for their model.

```
# Some example tasks
my_model.plot_fit() # plots the fit of the model
my_model.plot_sample(nsims=10) # draws samples from the model
my_model.plot_ppc(T=np.mean) # plots histogram of posterior predictive check for mean
my_model.plot_predict(h=5) # plots predictions for next 5 time steps
my_model.plot_predict_is(h=5) # plots rolling in-sample prediction for past 5 time␣
↪steps

predictions = my_model.predict(h=5, intervals=True) # outputs dataframe of predictions
samples = my_model.sample(nsims=10) # returns 10 samples from the data
ppc_pvalue = my_model.ppc(T=np.mean) # p-value for mean posterior predictive test
```

CHAPTER 4

Tutorials

Want to learn how to use this library? Check out these tutorials:

4.1 Getting Started with Time Series

4.1.1 Introduction

Time series analysis is a subfield of statistics and econometrics. Time series data y_t is indexed by time t and ordered sequentially. This presents unique challenges including autocorrelation within the data, non-exchangeability of data points, and non-stationarity of data and parameters. Because of the sequential nature of the data, time series analysis has particular goals. We can summarize these goals into one of **description** of a time series in terms of latent components or features of interest, and **prediction**, which aims to produce reasonable forecasts of the future (Harvey, 1990).

From start to finish, we can place time series modelling in a framework in the spirit of **Box's Loop** (Blei, D.M. 2014). In particular, we:

1. Build a model for the time series data
2. Perform inference on the model
3. Check the model fit, performing evaluation & criticism
4. Revise the model, repeat until happy
5. Perform retrospection and prediction with the model

Below we outline an example model building process for JPMorgan Chase stock data, where the index is daily. Consider this time series data:

```
import pandas as pd
import numpy as np
from pandas_datareader.data import DataReader
from datetime import datetime
```

(continues on next page)

(continued from previous page)

```
a = DataReader('JPM',  'yahoo', datetime(2006,6,1), datetime(2016,6,1))
a_returns = pd.DataFrame(np.diff(np.log(a['Adj Close'].values)))
a_returns.index = a.index.values[1:a.index.values.shape[0]]
a_returns.columns = ["JPM Returns"]

a_returns.head()
```

The index of the data is meaningful for this data; we cannot simply 'shuffle the deck' otherwise we could lose meaningful dependencies such as seasonality, trends, cycles and other components.

4.1.2 Step One: Visualize the Data

Because time series is sequential, plotting the data allows us to obtain an idea of its properties. We can also plot autocorrelation plots of the data (and transformations of the data) to understand if autocorrelation exists in the series. Lastly, in this stage, we can reason about potential features that might explain variation in the series.

For our stock market data, we can first plot the data:

```
plt.figure(figsize=(15, 5))
plt.ylabel("Returns")
plt.plot(a_returns)
plt.show()
```

It appears that the volatility of the series changes over time, and is clustering in periods of market turbulence, such as in the financial crisis of 2008. We can obtain more insight by plotting autocorrelation functions of the returns and squared returns:

```
import pyflux as pf
import matplotlib.pyplot as plt
pf.acf_plot(a_returns.values.T[0])
pf.acf_plot(np.square(a_returns.values.T[0]))
```

The squared returns demonstrate strong evidence of autocorrelation. The fact that autocorrelation persists and decays over multiply lags is evidence of an autoregressive effect within volatility. For returns, there is less strong evidence of autocorrelation, although the first lag is significant.

4.1.3 Step Two: Propose a Model

We reason about a model that can explain the variation in the data and we specify any prior beliefs we have about the model parameters. We saw evidence of volatility clustering. One way to model this effect is through a GARCH model for volatility (Bollerslev, T. 1986).

$$y_t \sim N\left(\mu, \sigma_t\right)$$

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

We will perform Bayesian inference on this model, and so we will specify some priors. We will ensure $\omega > 0$ through a log transform, and we will use a Truncated Normal prior on α, β:

```
my_model = pf.GARCH(p=1, q=1, data=a_returns)
print(my_model.latent_variables)
```

(continues on next page)

(continued from previous page)

```
  Index    Latent Variable     Prior           Prior Hyperparameters   V.I. Dist
↪Transform
  ======== ================== =============== ======================= ==========
↪==========
  0        Vol Constant        Normal          mu0: 0, sigma0: 3       Normal     exp
  1        q(1)                Normal          mu0: 0, sigma0: 0.5     Normal
↪logit
  2        p(1)                Normal          mu0: 0, sigma0: 0.5     Normal
↪logit
  3        Returns Constant    Normal          mu0: 0, sigma0: 3       Normal     None

my_model.adjust_prior(1, pf.TruncatedNormal(0.01, 0.5, lower=0.0, upper=1.0))
my_model.adjust_prior(2, pf.TruncatedNormal(0.97, 0.5, lower=0.0, upper=1.0))
```

4.1.4 Step Three: Perform Inference

As a third step we need to decide how to perform inference for the model. Below we use Metropolis-Hastings for approximate inference on our GARCH model. We also plot the latent variables α and β:

```
result = my_model.fit('M-H', nsims=20000)

Tuning complete! Now sampling.
Acceptance rate of Metropolis-Hastings is 0.33865

my_model.plot_z([1,2])
```

4.1.5 Step Four: Evaluate Model Fit

We next evaluate the fit of the model and establish whether we can improve the model further. For time series, the simplest way to visualize fit is to plot the series against its predicted values; we can also check out-of-sample performance. If we seek further model improvements, we go back to **step two** and proceed. Once we are happy we go to **step five**.

Below we plot the fit of the GARCH model and observe that it picking up volatility clustering in the series:

```
my_model.plot_fit(figsize=(15,5))
```

We can also plot samples from the posterior predictive density:

```
my_model.plot_sample(nsims=10, figsize=(15,7))
```

We can see that the samples (colored) appear to be picking up variation in the data (the square datapoints).

We can also perform a posterior predictive check (PPC) on features of the generated series, for example the kurtosis:

```
from scipy.stats import kurtosis
my_model.plot_ppc(T=kurtosis)
```

It appears our generated data underestimates kurtosis in the series. This is not surprising as we are assuming normally distributed returns, so we may want to consider alternative volatility models.

4.1.6 Step Five: Analyse and Predict

Once we are happy with our model, we can use it to analyze the historical time series and make predictions. For our GARCH model, we can see from the previous fit plot that the main periods of volatility picked up are during the financial crisis of 2007-2008, and during the Eurozone crisis in late 2011. We can also obtain forward predictions with the model:

```
my_model.plot_predict(h=30, figsize=(15,5))
```

4.1.7 References

Blei, D. M. (2014). Build, compute, critique, repeat: Data analysis with latent variable models. Annual Review of Statistics and Its Application, 1, 203–232.

Bollerslev, T. (1986). Generalized Autoregressive Conditional Heteroskedasticity. Journal of Econometrics. April, 31:3, pp. 307–27.

Harvey A. C. (1990). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press.

CHAPTER 5

Model Guide

5.1 ARIMA models

5.1.1 Introduction

Autoregressive integrated moving average (ARIMA) models were popularised by Box and Jenkins (1970). An ARIMA model describes a univariate time series as a combination of autoregressive (AR) and moving average (MA) lags which capture the autocorrelation within the time series. The order of integration denotes how many times the series has been differenced to obtain a stationary series.

We write an $ARIMA(p, d, q)$ model for some time series data y_t, where p is the number of autoregressive lags, d is the degree of differencing and q is the number of moving average lags as:

$$\Delta^D y_t = \sum_{i=1}^{p} \phi_i \Delta^D y_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t$$

$$\epsilon_t \sim N\left(0, \sigma^2\right)$$

ARIMA models are associated with a Box-Jenkins approach to time series. According to this approach, you should difference the series until it is stationary, and then use information criteria and autocorrelation plots to choose the appropriate lag order for an $ARIMA$ process. You then apply inference to obtain latent variable estimates, and check the model to see whether the model has captured the autocorrelation in the time series. For example, you can plot the autocorrelation of the model residuals. Once you are happy, you can use the model for retrospection and forecasting.

5.1.2 Example

We'll run an ARIMA Model for yearly sunspot data. First we load the data:

```
import numpy as np
import pandas as pd
import pyflux as pf
from datetime import datetime
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
%matplotlib inline

data = pd.read_csv('https://vincentarelbundock.github.io/Rdatasets/csv/datasets/
↪sunspot.year.csv')
data.index = data['time'].values

plt.figure(figsize=(15,5))
plt.plot(data.index,data['sunspot.year'])
plt.ylabel('Sunspots')
plt.title('Yearly Sunspot Data');
```

We can build an ARIMA model as follows, specifying the order of model we want, as well as a pandas DataFrame or numpy array carrying the data. Here we specify an arbitrary $ARIMA(4,0,4)$ model:

```
model = pf.ARIMA(data=data, ar=4, ma=4, target='sunspot.year', family=pf.Normal())
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit("MLE")
x.summary()

ARIMA(4,0,4)
====================================== ========================================
Dependent Variable: sunspot.year       Method: MLE
Start Date: 1704                       Log Likelihood: -1189.488
End Date: 1988                         AIC: 2398.9759
Number of observations: 285            BIC: 2435.5008
================================================================================
Latent Variable      Estimate   Std Error  z         P>|z|    95% C.I.
==================== ========== ========== ========= ======== ======================
Constant             8.0092     3.2275     2.4816    0.0131   (1.6834 | 14.3351)
AR(1)                1.6255     0.0367     44.2529   0.0      (1.5535 | 1.6975)
AR(2)                -0.4345    0.2455     -1.7701   0.0767   (-0.9157 | 0.0466)
AR(3)                -0.8819    0.2295     -3.8432   0.0001   (-1.3317 | -0.4322)
AR(4)                0.5261     0.0429     12.2515   0.0      (0.4419 | 0.6103)
MA(1)                -0.5061    0.0383     -13.2153  0.0      (-0.5812 | -0.4311)
MA(2)                -0.481     0.1361     -3.533    0.0004   (-0.7478 | -0.2142)
MA(3)                0.2511     0.1093     2.2979    0.0216   (0.0369 | 0.4653)
MA(4)                0.2846     0.0602     4.7242    0.0      (0.1665 | 0.4027)
Sigma                15.7944
================================================================================
```

We can plot the latent variables z^{MLE}: using the *plot_z()*: method:

```
model.plot_z(figsize=(15,5))
```

We can plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

We can get an idea of the performance of our model by using rolling in-sample prediction through the `plot_predict_is()`: method:

```
model.plot_predict_is(h=50, figsize=(15,5))
```

If we want to plot predictions, we can use the `plot_predict()`: method:

```
model.plot_predict(h=20,past_values=20,figsize=(15,5))
```

If we want the predictions in a DataFrame form, then we can just use the `predict()`: method.

5.1.3 Class Description

class ARIMA(*data*, *ar*, *ma*, *integ*, *target*, *family*)

Autoregressive Integrated Moving Average Models (ARIMA).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
ar	int	The number of autoregressive lags
ma	int	The number of moving average lags
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc (*T*, *nsims*)
Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)
Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)
Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.1.4 References

Box, G; Jenkins, G. (1970). Time Series Analysis: Forecasting and Control. San Francisco: Holden-Day.

Hamilton, J.D. (1994). Time Series Analysis. Taylor & Francis US.

5.2 ARIMAX models

5.2.1 Introduction

Autoregressive integrated moving average (ARIMAX) models extend ARIMA models through the inclusion of exogenous variables X. We write an $ARIMAX(p,d,q)$ model for some time series data y_t and exogenous data X_t, where p is the number of autoregressive lags, d is the degree of differencing and q is the number of moving average lags as:

$$\Delta^D y_t = \sum_{i=1}^{p} \phi_i \Delta^D y_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \sum_{m=1}^{M} \beta_m X m, t + \epsilon_t$$

$$\epsilon_t \sim N\left(0, \sigma^2\right)$$

5.2.2 Example

We will combine ARIMA dynamics with intervention analysis for monthly UK driver death data. There are two interventions we are interested in: the 1974 oil crisis and the introduction of the seatbelt law in 1983. We will model the effects of these events as structural breaks.

```
import numpy as np
import pandas as pd
import pyflux as pf
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

data = pd.read_csv("https://vincentarelbundock.github.io/Rdatasets/csv/MASS/drivers.
↪csv")
data.index = data['time'];
data.loc[(data['time']>=1983.05), 'seat_belt'] = 1;
data.loc[(data['time']<1983.05), 'seat_belt'] = 0;
data.loc[(data['time']>=1974.00), 'oil_crisis'] = 1;
data.loc[(data['time']<1974.00), 'oil_crisis'] = 0;
plt.figure(figsize=(15,5));
plt.plot(data.index,data['drivers']);
plt.ylabel('Driver Deaths');
plt.title('Deaths of Car Drivers in Great Britain 1969-84');
plt.plot();
```

The structural breaks can be included via patsy notation. Below we estimate a point mass estimate z^{MLE} of the latent variables:

```
model = pf.ARIMAX(data=data, formula='drivers~1+seat_belt+oil_crisis',
                  ar=1, ma=1, family=pf.Normal())
x = model.fit("MLE")
x.summary()

ARIMAX(1,0,1)
=========================================␣
↪=============================================
Dependent Variable: drivers              Method: MLE
```

(continues on next page)

(continued from previous page)

```
Start Date: 1969.08333333                Log Likelihood: -1278.7616
End Date: 1984.91666667                  AIC: 2569.5232
Number of observations: 191              BIC: 2589.0368
============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|     95% C.I.
========================= ========== ========== ========= ========
↪=========================
AR(1)                     0.5002     0.0933     5.3607    0.0       (0.3173 | 0.6831)
MA(1)                     0.1713     0.0991     1.7275    0.0841    (-0.023 | 0.3656)
Beta 1                    946.585    176.9439   5.3496    0.0       (599.7749 | 1293.
↪3952)
Beta seat_belt            -57.8924   57.8211    -1.0012   0.3167    (-171.2217 | 55.437)
Beta oil_crisis           -151.673   44.119     -3.4378   0.0006    (-238.1462 | -65.
↪1998)
Sigma                     195.6042
============================================================================================
```

We can plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

To forecast forward we need exogenous variables for future dates. Since the interventions carry forward, we can just use a slice of the existing dataframe and use func:*plot_predict*:

```
model.plot_predict(h=10, oos_data=data.iloc[-12:], past_values=100, figsize=(15,5))
```

5.2.3 Class Description

class ARIMAX(*data*, *formula*, *ar*, *ma*, *integ*, *target*, *family*)

Autoregressive Integrated Moving Average Exogenous Variable Models (ARIMAX).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
formula	string	Patsy notation specifying the regression
ar	int	The number of autoregressive lags
ma	int	The number of moving average lags
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit (*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit (***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc (*T*, *nsims*)
Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict (*h*, *oos_data*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the h first rows from *oos_data* and take the values for the exogenous variables that you asked for in the patsy formula.

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *oos_data*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
intervals	boolean	Whether to return prediction intervals

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the 5 first rows from *oos_data* and take the values for the exogenous variables that you specified as exogenous variables in the patsy formula.

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.2.4 References

Box, G; Jenkins, G. (1970). Time Series Analysis: Forecasting and Control. San Francisco: Holden-Day.

Hamilton, J.D. (1994). Time Series Analysis. Taylor & Francis US.

5.3 DAR models

5.3.1 Introduction

Gaussian state space models - often called structural time series or unobserved component models - provide a way to decompose a time series into several distinct components. These components can be extracted in closed form using the Kalman filter if the errors are jointly Gaussian, and parameters can be estimated via the prediction error decomposition and Maximum Likelihood.

We can write a **dynamic autoregression model** in this framework as:

$$y_t = \sum_{i=1}^{p} \phi_{i,t} y_{t-i} + \epsilon_t$$

$$\phi_{i,t} = \phi_{i,t-1} + \eta_{i,t}$$

$$\epsilon_t \sim N\left(0, \sigma^2\right)$$

$$\eta_{i,t} \sim N\left(0, \sigma^2_{\eta_i}\right)$$

In other words the dynamic autoregression coefficients follow a random walk.

5.3.2 Example

We'll run an Dynamic Autoregressive (DAR) Model for yearly sunspot data:

```
import numpy as np
import pandas as pd
import pyflux as pf
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

data = pd.read_csv('https://vincentarelbundock.github.io/Rdatasets/csv/datasets/
↪sunspot.year.csv')
data.index = data['time'].values

plt.figure(figsize=(15,5))
plt.plot(data.index,data['sunspot.year'])
plt.ylabel('Sunspots')
plt.title('Yearly Sunspot Data');
```

Here we specify an arbitrary DAR(9) model (note: which is probably overspecified).

```
model = pf.DAR(data=data, ar=9, integ=0, target='sunspot.year')
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit("MLE")
x.summary()

DAR(9, integrated=0)
=====================================␣
↪================================================
Dependent Variable: sunspot.year       Method: MLE
Start Date: 1709                       Log Likelihood: -1179.097
End Date: 1988                         AIC: 2380.194
Number of observations: 280            BIC: 2420.1766
=============================================================================================
Latent Variable          Estimate   Std Error  z         P>|z|    95% C.I.
======================== ========== ========== ======== ========␣
↪========================
Sigma^2 irregular        0.301
Constant                 60.0568    23.83      2.5202   0.0117   (13.3499 | 106.7637)
Sigma^2 AR(1)            0.005
Sigma^2 AR(2)            0.0
Sigma^2 AR(3)            0.0005
Sigma^2 AR(4)            0.0001
Sigma^2 AR(5)            0.0002
Sigma^2 AR(6)            0.0011
Sigma^2 AR(7)            0.0002
Sigma^2 AR(8)            0.0003
Sigma^2 AR(9)            0.032
=============================================================================================
```

Note we have no standard errors in the results table because it shows the transformed parameters. If we want standard errors, we can call `x.summary(transformed=False)`. Next we will plot the in-sample fit and the dynamic coefficients using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

The sharp changes at the beginning reflect the diffuse initialization; together with high initial uncertainty, this leads to stronger updates towards the beginning of the series. We can predict forward using plot_predict:

We can predict forwards through the `plot_predict()`: method:

```
model.plot_predict(h=50, past_values=40, figsize=(15,5))
```

The prediction intervals here are unrealistic and reflect the Gaussian distributional assumption we've chosen – we can't have negative sunspots! – but if we are just want the predictions themselves, we can use the `predict()`: method.

5.3.3 Class Description

class DAR(*data*, *ar*, *integ*, *target*, *family*)

Dynamic Autoregression Models (DAR).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
ar	int	The number of autoregressive lags
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit (***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_predict (*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is (*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_z (*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

predict (*h*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

simulation_smoother (*beta*)
Returns np.ndarray of draws of the data from the Durbin and Koopman (2002) simulation smoother.

Parameter	Type	Description
beta	np.array	np.array of latent variables

Recommended just to use model.latent_variables.get_z_values() for the beta input, if you have already fit a model.

Returns : np.ndarray - samples from simulation smoother

5.3.4 References

Durbin, J. and Koopman, S. J. (2002). A simple and efficient simulation smoother for state space time series analysis. Biometrika, 89(3):603–615.

Harvey, A. C. (1989). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge.

5.4 Dynamic Linear regression models

5.4.1 Introduction

Gaussian state space models - often called structural time series or unobserved component models - provide a way to decompose a time series into several distinct components. These components can be extracted in closed form using the Kalman filter if the errors are jointly Gaussian, and parameters can be estimated via the prediction error decomposition and Maximum Likelihood.

We can support **Dynamic Linear Regression** in the state space framework:

$$y_t = \boldsymbol{x}_t^{'}\boldsymbol{\beta}_t + \epsilon_t$$

$$\boldsymbol{\beta}_t = \boldsymbol{\beta}_{t-1} + \boldsymbol{\eta}_t$$

$$\epsilon_t \sim N\left(0, \sigma_\epsilon^2\right)$$

$$\boldsymbol{\eta}_t \sim N\left(\mathbf{0}, \Sigma_\eta\right)$$

5.4.2 Example

In constructing portfolios in finance, we are often after the β of a stock which can be used to construct the systematic component of returns. But this may not be a static quantity. For normally distributed returns (!) we can use a dynamic linear regression model using the Kalman filter and smoothing algorithm to track its evolution. First let's get some data on excess returns. We'll look at Amazon stock (AMZN) and use the S&P500 as 'the market'.

```
from pandas_datareader import DataReader
from datetime import datetime

a = DataReader('AMZN',  'yahoo', datetime(2012,1,1), datetime(2016,6,1))
a_returns = pd.DataFrame(np.diff(np.log(a['Adj Close'].values)))
a_returns.index = a.index.values[1:a.index.values.shape[0]]
a_returns.columns = ["Amazon Returns"]

spy = DataReader('SPY',  'yahoo', datetime(2012,1,1), datetime(2016,6,1))
spy_returns = pd.DataFrame(np.diff(np.log(spy['Adj Close'].values)))
spy_returns.index = spy.index.values[1:spy.index.values.shape[0]]
spy_returns.columns = ['S&P500 Returns']

one_mon = DataReader('DGS1MO', 'fred',datetime(2012,1,1), datetime(2016,6,1))
one_day = np.log(1+one_mon)/365

returns = pd.concat([one_day,a_returns,spy_returns],axis=1).dropna()
excess_m = returns["Amazon Returns"].values - returns['DGS1MO'].values
excess_spy = returns["S&P500 Returns"].values - returns['DGS1MO'].values
final_returns = pd.DataFrame(np.transpose([excess_m,excess_spy, returns['DGS1MO'].
↪values]))
final_returns.columns=["Amazon","SP500","Risk-free rate"]
final_returns.index = returns.index

plt.figure(figsize=(15,5))
plt.title("Excess Returns")
x = plt.plot(final_returns);
plt.legend(iter(x), final_returns.columns);
```

Here we define a Dynamic Linear regression as follows:

```
model = pf.DynReg('Amazon ~ SP500', data=final_returns)
```

We can also use the higher-level wrapper which allows us to specify the family, although if we pick a non-Gaussian family then the model will be estimated in a different way (not through the Kalman filter):

```
model = pf.DynamicGLM('Amazon ~ SP500', data=USgrowth, family=pf.Normal())
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit()
x.summary()

Dynamic Linear Regression
=====================================␣
↪================================================
Dependent Variable: Amazon              Method: MLE
Start Date: 2012-01-04 00:00:00         Log Likelihood: 2871.5419
End Date: 2016-06-01 00:00:00           AIC: -5737.0838
Number of observations: 1101            BIC: -5722.0719
```

(continues on next page)

(continued from previous page)

```
==========================================================================================
Latent Variable          Estimate   Std Error  z         P>|z|     95% C.I.
======================== ========== ========== ======== ========
↪=========================
Sigma^2 irregular        0.0003
Sigma^2 1                0.0
Sigma^2 SP500            0.0024
==========================================================================================
```

We can plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,15))
```

The third plot shows β_{AMZN}. Following the burn-in period, the β hovered just above 1 in 2013, although it became very correlated with market performance in 2014. More recently it has settled down again to hover just above 1. The fourth plot shows the remaining residual component of return (not including α).

5.4.3 Class Description

class DynLin (*formula*, *data*)

Dynamic Linear Regression models

Parameter	Type	Description
formula	string	Patsy notation specifying the regression
data	pd.DataFrame	Contains the univariate time series

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior (*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit (*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit (***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc (*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict (*h*, *oos_data*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the h first rows from *oos_data* and take the values for the exogenous variables that you asked for in the patsy formula.

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is (*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *oos_data*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
intervals	boolean	Whether to return prediction intervals

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the 5 first rows from *oos_data* and take the values for the exogenous variables that you specified as exogenous variables in the patsy formula.

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is(*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

simulation_smoother (*beta*)

Returns np.ndarray of draws of the data from the Durbin and Koopman (2002) simulation smoother.

Parameter	Type	Description
beta	np.array	np.array of latent variables

Recommended just to use model.latent_variables.get_z_values() for the beta input, if you have already fit a model.

Returns : np.ndarray - samples from simulation smoother

5.4.4 References

Durbin, J. and Koopman, S. J. (2002). A simple and efficient simulation smoother for state space time series analysis. Biometrika, 89(3):603–615.

Harvey, A. C. (1989). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge.

5.5 Beta-t-EGARCH models

5.5.1 Introduction

Beta-t-EGARCH models were proposed by Harvey and Chakravarty (2008). They extend upon GARCH models by using the conditional score of a t-distribution drive the conditional variance. This allows for increased robustness to outliers through a 'trimming' property of the t-distribution score. Their formulation also follows that of an EGARCH model, see Nelson (1991), where the conditional volatility is log-transformed, which prevents the need for restrictive parameter constraints as in GARCH models.

Below is the formulation for a $Beta\text{-}t\text{-}EGARCH(p,q)$ model:

$$y_t = \mu + \exp\left(\lambda_{t|t-1}/2\right)\epsilon_t$$

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p}\alpha_i\lambda_{t-i} + \sum_{j=1}^{q}\beta_j\left(\frac{(\nu+1)\,y_{t-j}^2}{\nu\exp\left(\lambda_{t-j|t-j-1}\right)+y_{t-j}^2} - 1\right)$$

$$\epsilon_t \sim t_\nu$$

Past evidence also suggests a leverage effect in stock returns, see Black (1976), that observes that volatility increases more after bad news than good news. Following Harvey and Succarrat (2013), we can incorporate a leverage effect in the Beta-t-EGARCH model as follows:

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j} + \kappa \left(\text{sgn}\left(-\epsilon_{t-1}\right)\left(u_{t-1}+1\right)\right)$$

Where κ is the leverage coefficient.

5.5.2 Developer Note

- This model type has yet to be Cythonized so performance can be slow.

5.5.3 Example

First let us load some financial time series data from Yahoo Finance:

```
import numpy as np
import pyflux as pf
import pandas as pd
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

jpm = DataReader('JPM',  'yahoo', datetime(2006,1,1), datetime(2016,3,10))
returns = pd.DataFrame(np.diff(np.log(jpm['Adj Close'].values)))
returns.index = jpm.index.values[1:jpm.index.values.shape[0]]
returns.columns = ['JPM Returns']

plt.figure(figsize=(15,5));
plt.plot(returns.index,returns);
plt.ylabel('Returns');
plt.title('JPM Returns');
```

One way to visualize the underlying volatility of the series is to plot the absolute returns $\mid y \mid$:

```
plt.figure(figsize=(15,5))
plt.plot(returns.index, np.abs(returns))
plt.ylabel('Absolute Returns')
plt.title('JP Morgan Absolute Returns');
```

There appears to be some evidence of volatility clustering over this period. Let's fit a $Beta\text{-}t\text{-}EGARCH(1,1)$ model using a point mass estimate z^{MLE}:

```
model = pf.EGARCH(returns, p=1, q=1)
x = model.fit()
x.summary()

EGARCH(1,1)
=========================================␣
↪==============================================
Dependent Variable: JPM Returns          Method: MLE
```

(continues on next page)

(continued from previous page)

```
Start Date: 2006-01-05 00:00:00          Log Likelihood: 6663.2492
End Date: 2016-03-10 00:00:00            AIC: -13316.4985
Number of observations: 2562             BIC: -13287.2557
============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========␣
↪=========================
Vol Constant              -0.0575    0.0166     -3.4695   0.0005   (-0.0899 | -0.025)
p(1)                      0.9933
q(1)                      0.103
v                         6.0794
Returns Constant          0.0007     0.0247     0.0292    0.9767   (-0.0477 | 0.0492)
============================================================================================
```

The standard errors are not shown for transformed variables. You can pass through a `transformed=False` argument to `summary` to obtain this information for untransformed variables.

We can plot the EGARCH latent variables with *plot_z()*: :

```
model.plot_z([1,2],figsize=(15,5))
```

We can plot the fit with `plot_fit()`:

```
model.plot_fit(figsize=(15,5))
```

And plot predictions of future conditional volatility with `plot_predict()`:

```
model.plot_predict(h=10)
```

If we had wanted predictions in dataframe form, we could have used `predict()`: instead.

We can view how well we predicted using in-sample rolling prediction with `plot_predict_is()`:

```
model.plot_predict_is(h=50,figsize=(15,5))
```

We can also estimate a Beta-t-EGARCH model with leverage through `add_leverage()`:

```
model.add_leverage()
x = model.fit()
x.summary()

EGARCH(1,1)
=========================================␣
↪=================================================
Dependent Variable: JPM Returns          Method: MLE
Start Date: 2006-01-05 00:00:00          Log Likelihood: 6688.2732
End Date: 2016-03-10 00:00:00            AIC: -13364.5465
Number of observations: 2562             BIC: -13329.4552
============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========␣
↪=========================
Vol Constant              -0.0586    0.0219     -2.6753   0.0075   (-0.1015 | -0.0157)
p(1)                      0.9934
q(1)                      0.0781
Leverage Term             0.0578     0.0012     49.8546   0.0      (0.0555 | 0.0601)
v                         6.3724
```

(continues on next page)

(continued from previous page)

```
Returns Constant          0.0005     0.0        160.6585 0.0       (0.0005 | 0.0005)
==========================================================================================
```

We have a small leverage effect for the time series:

5.5.4 Class Description

class EGARCH(*data*, *p*, *q*, *target*)

Beta-t-EGARCH Models

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
p	int	The number of autoregressive lags σ^2
q	int	The number of ARCH terms ϵ^2
target	string or int	Which column of DataFrame/array to use.

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

add_leverage()

Adds a leverage term to the model, meaning volatility can respond differently to the sign of the news; see Harvey and Succarrat (2013). Conditional volatility will now follow:

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j} + \kappa \left(\text{sgn}\left(-\epsilon_{t-1}\right)\left(u_{t-1}+1\right)\right)$$

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z (*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc (*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.5.5 References

Black, F. (1976) Studies of stock price volatility changes. In: Proceedings of the 1976 Meetings of the American Statistical Association. pp. 171–181.

Harvey, A.C. & Chakravarty, T. (2008) Beta-t-(E)GARCH. Cambridge Working Papers in Economics 0840, Faculty of Economics, University of Cambridge, 2008. [p137]

Harvey, A.C. & Sucarrat, G. (2013) EGARCH models with fat tails, skewness and leverage. Computational Statistics and Data Analysis, Forthcoming, 2013. URL http://dx.doi.org/10.1016/j.csda.2013.09. 022. [p138, 139, 140, 143]

Nelson, D. B. (1991), 'Conditional heteroskedasticity in asset returns: A new approach', Econometrica 59, 347—370.

5.6 Beta-t-EGARCH in-mean models

5.6.1 Introduction

Beta-t-EGARCH in-mean models extend $Beta\text{-}t\text{-}EGARCH(p,q)$ models by allowing returns to depend upon a past conditional volatility component. The $Beta\text{-}t\text{-}EGARCH(p,q)$ in-mean model includes this effect ϕ as follows:

$$y_t = \mu + \phi \exp\left(\lambda_{t|t-1}/2\right) + \exp\left(\lambda_{t|t-1}/2\right)\epsilon_t$$

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j}$$

$$\epsilon_t \sim t_\nu$$

5.6.2 Developer Note

- This model type has yet to be Cythonized so performance can be slow.

5.6.3 Example

First let us load some financial time series data from Yahoo Finance:

```
import numpy as np
import pyflux as pf
import pandas as pd
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

jpm = DataReader('JPM',  'yahoo', datetime(2006,1,1), datetime(2016,3,10))
returns = pd.DataFrame(np.diff(np.log(jpm['Adj Close'].values)))
returns.index = jpm.index.values[1:jpm.index.values.shape[0]]
returns.columns = ['JPM Returns']

plt.figure(figsize=(15,5));
plt.plot(returns.index,returns);
plt.ylabel('Returns');
plt.title('JPM Returns');
```

One way to visualize the underlying volatility of the series is to plot the absolute returns $|y|$:

```
plt.figure(figsize=(15,5))
plt.plot(returns.index, np.abs(returns))
plt.ylabel('Absolute Returns')
plt.title('JP Morgan Absolute Returns');
```

There appears to be some evidence of volatility clustering over this period. Let's fit a $Beta\text{-}t\text{-}EGARCH - M(1,1)$ model using a BBVI estimate z^{BBVI}:

```
model = pf.EGARCHM(returns,p=1,q=1)
x = model.fit('BBVI', record_elbo=True, iterations=1000, map_start=False)
x.summary()

EGARCHM(1,1)
=========================================
 ↪=================================================
Dependent Variable: AAPL Returns         Method: BBVI
Start Date: 2006-01-05 00:00:00          Unnormalized Log Posterior: 7016.148
End Date: 2016-11-11 00:00:00            AIC: -14020.2959822
Number of observations: 2734             BIC: -13984.8148561
=============================================================================================
Latent Variable              Median              Mean                95% Credibility
 ↪Interval
============================ =================== ===================
 ↪==========================
Vol Constant                 -0.2842             -0.2841             (-0.3474 | -0.2207)
p(1)                         0.9624              0.9624              (0.9579 | 0.9665)
q(1)                         0.1889              0.1889              (0.1784 | 0.2)
v                            8.689               8.6902              (8.0781 | 9.3552)
Returns Constant             0.0001              0.0001              (-0.0094 | 0.0093)
GARCH-M                      0.1087              0.1085              (0.0226 | 0.1942)
=============================================================================================
```

We can plot the ELBO through BBVI by calling `plot_elbo()`: on the results object:

```
x.plot_elbo(figsize=(15,7))
```

As we can see, the ELBO converges after around 200 iterations. We can plot the model fit through `plot_fit()`:

```
model.plot_fit(figsize=(15,7))
```

And plot predictions of future conditional volatility with `plot_predict()`:

```
model.predict(h=10)
```

We can plot samples from the posterior predictive density through `plot_sample()`:

```
model.plot_sample(figsize=(15, 7))
```

And we can do posterior predictive checks on discrepancies of interest:

```
from scipy.stats import kurtosis
model.plot_ppc(T=kurtosis,figsize=(15, 7))
model.plot_ppc(T=np.std,figsize=(15, 7))
```

Here it appears our generated samples generate kurtosis that is slightly lower than the data, and a standard deviation that is slightly higher, but we are not too off in both checks.

5.6.4 Class Description

class EGARCHM(*data*, *p*, *q*, *target*)
Beta-t-EGARCH in-mean Models

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
p	int	The number of autoregressive lags σ^2
q	int	The number of ARCH terms ϵ^2
target	string or int	Which column of DataFrame/array to use.

Attributes

latent_variables
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

add_leverage()
Adds a leverage term to the model, meaning volatility can respond differently to the sign of the news; see Harvey and Succarrat (2013). Conditional volatility will now follow:

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j} + \kappa \left(\text{sgn}\left(-\epsilon_{t-1}\right)\left(u_{t-1}+1\right)\right)$$

adjust_prior(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc (*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.6.5 References

Black, F. (1976) Studies of stock price volatility changes. In: Proceedings of the 1976 Meetings of the American Statistical Association. pp. 171–181.

Harvey, A.C. & Chakravarty, T. (2008) Beta-t-(E)GARCH. Cambridge Working Papers in Economics 0840, Faculty of Economics, University of Cambridge, 2008. [p137]

Harvey, A.C. & Sucarrat, G. (2013) EGARCH models with fat tails, skewness and leverage. Computational Statistics and Data Analysis, Forthcoming, 2013. URL http://dx.doi.org/10.1016/j.csda.2013.09. 022. [p138, 139, 140, 143]

Nelson, D. B. (1991), 'Conditional heteroskedasticity in asset returns: A new approach', Econometrica 59, 347—370.

5.7 Beta-t-EGARCH in-mean regression models

5.7.1 Introduction

We can expand the Beta-t-EGARCH in-mean model to include exogenous regressors in both the returns and conditional volatility equation:

$$y_t = \mu + \sum_{k=1}^{m} \phi_m X_{m,t} + \exp\left(\lambda_{t|t-1}/2\right)\epsilon_t$$

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j \left(\frac{(\nu+1)\, y_{t-j}^2}{\nu \exp\left(\lambda_{t-j|t-j-1}\right) + y_{t-j}^2} - 1 \right) + \sum_{k=1}^{m} \gamma_m X_{m,t}$$

$$\epsilon_t \sim t_\nu$$

5.7.2 Developer Note

- This model type has yet to be Cythonized so performance can be slow.

5.7.3 Example

First let us load some financial time series data from Yahoo Finance:

```
import numpy as np
import pyflux as pf
import pandas as pd
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

a = DataReader('JPM',  'yahoo', datetime(2000,1,1), datetime(2016,3,10))
a_returns = pd.DataFrame(np.diff(np.log(a['Adj Close'].values)))
a_returns.index = a.index.values[1:a.index.values.shape[0]]
a_returns.columns = ["JPM Returns"]

spy = DataReader('SPY',  'yahoo', datetime(2000,1,1), datetime(2016,3,10))
spy_returns = pd.DataFrame(np.diff(np.log(spy['Adj Close'].values)))
spy_returns.index = spy.index.values[1:spy.index.values.shape[0]]
spy_returns.columns = ['S&P500 Returns']

one_mon = DataReader('DGS1MO', 'fred',datetime(2000,1,1), datetime(2016,3,10))
one_day = np.log(1+one_mon)/365
```

(continues on next page)

(continued from previous page)

```
returns = pd.concat([one_day,a_returns,spy_returns],axis=1).dropna()
excess_m = returns["JPM Returns"].values - returns['DGS1MO'].values
excess_spy = returns["S&P500 Returns"].values - returns['DGS1MO'].values
final_returns = pd.DataFrame(np.transpose([excess_m,excess_spy, returns['DGS1MO'].
↪values]))
final_returns.columns=["JPM","SP500","Rf"]
final_returns.index = returns.index

plt.figure(figsize=(15,5))
plt.title("Excess Returns")
x = plt.plot(final_returns);
plt.legend(iter(x), final_returns.columns);
```

Let's fit an EGARCH-M model to JPM's excess returns series, with a risk-free rate regressor:

```
modelx = pf.EGARCHMReg(p=1,q=1,data=final_returns,formula='JPM ~ Rf')
results = modelx.fit()
results.summary()

EGARCHMReg(1,1)
=======================================␣
↪==============================================
Dependent Variable: JPM                  Method: MLE
Start Date: 2001-08-01 00:00:00          Log Likelihood: 9621.2996
End Date: 2016-03-10 00:00:00            AIC: -19226.5993
Number of observations: 3645             BIC: -19176.9904
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========␣
↪=========================
p(1)                      0.9936
q(1)                      0.0935
v                         6.5414
GARCH-M                   -0.0199    0.023      -0.8657   0.3866   (-0.0649 | 0.0251)
Vol Beta 1                -0.0545    0.0007     -77.9261  0.0      (-0.0559 | -0.0531)
Vol Beta Rf               -0.0346    1.1161     -0.031    0.9753   (-2.2222 | 2.153)
Returns Beta 1            0.0011     0.0011     1.0218    0.3069   (-0.001 | 0.0032)
Returns Beta Rf           -1.1324    0.0941     -12.0398  0.0      (-1.3167 | -0.948)
=============================================================================================
```

Let's plot the latent variables with *plot_z()*:

```
modelx.plot_z([5,7],figsize=(15,5))
```

For this stock, the risk-free rate has a negative effect on excess returns. For the effects of returns on volatility, we are far more uncertain. We can plot the fit with `plot_fit()`:

```
modelx.plot_fit(figsize=(15,5))
```

5.7.4 Class Description

class EGARCHMReg(*data*, *formula*, *p*, *q*)

Long Memory Beta-t-EGARCH Models

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
formula	string	Patsy notation specifying the regression
p	int	The number of autoregressive lags σ^2
q	int	The number of ARCH terms ϵ^2

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

add_leverage()

Adds a leverage term to the model, meaning volatility can respond differently to the sign of the news; see Harvey and Succarrat (2013). Conditional volatility will now follow:

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j} + \kappa \left(\text{sgn}\left(-\epsilon_{t-1}\right)\left(u_{t-1}+1\right)\right)$$

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)
Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)
Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)
Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc (*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.7.5 References

Black, F. (1976) Studies of stock price volatility changes. In: Proceedings of the 1976 Meetings of the American Statistical Association. pp. 171–181.

Fernandez, C., & Steel, M. F. J. (1998a). On Bayesian Modeling of Fat Tails and Skewness. Journal of the American Statistical Association, 93, 359–371.

Harvey, A.C. & Chakravarty, T. (2008) Beta-t-(E)GARCH. Cambridge Working Papers in Economics 0840, Faculty of Economics, University of Cambridge, 2008. [p137]

Harvey, A.C. & Sucarrat, G. (2013) EGARCH models with fat tails, skewness and leverage. Computational Statistics and Data Analysis, Forthcoming, 2013. URL http://dx.doi.org/10.1016/j.csda.2013.09. 022. [p138, 139, 140, 143]

Mandelbrot, B.B. (1963) The variation of certain speculative prices. Journal of Business, XXXVI (1963). pp. 392–417

Nelson, D. B. (1991) Conditional heteroskedasticity in asset returns: A new approach. Econometrica 59, 347—370.

5.8 Beta-t-EGARCH long memory models

5.8.1 Introduction

Introducing a long-term and a short-term component into the Beta-t-EGARCH framework allows for the conditional volatility series to exhibit long memory, which is a feature of many financial time series, as first discussed by Mandelbrot in the 1960s:

$$y_t = \mu + \exp\left(\lambda_{t|t-1}/2\right)\epsilon_t$$

$$\lambda_{t|t-1} = \omega + \lambda_{1,t|t-1} + \lambda_{2,t|t-1}$$

$$\lambda_{1,t|t-1} = \sum_{i=1}^{p}\alpha_{1,i}\lambda_{1,t-i} + \sum_{j=1}^{q}\beta_{1,j}u_{t-j}$$

$$\lambda_{2,t|t-1} = \sum_{i=1}^{p}\alpha_{2,i}\lambda_{2,t-i} + \sum_{j=1}^{q}\beta_{2,j}u_{t-j}$$

$$\epsilon_t \sim t_\nu$$

We require $\alpha_1 \neq \alpha_2$ for identifiability.

5.8.2 Developer Note

- This model type has yet to be Cythonized so performance can be slow.

5.8.3 Example

First let us load some financial time series data from Yahoo Finance:

```
import numpy as np
import pyflux as pf
import pandas as pd
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

jpm = DataReader('JPM',  'yahoo', datetime(2006,1,1), datetime(2016,3,10))
returns = pd.DataFrame(np.diff(np.log(jpm['Adj Close'].values)))
returns.index = jpm.index.values[1:jpm.index.values.shape[0]]
returns.columns = ['JPM Returns']
```

Let's fit a long-memory $Beta\ t\ EGARCH(1,1)$ model using a point mass estimate z^{MLE}:

```
model = pf.LMEGARCH(returns,p=1,q=1)
x = model.fit()
x.summary()

LMEGARCH(1,1)
=========================================
 ↪=============================================
Dependent Variable: JPM Returns          Method: MLE
Start Date: 2006-01-05 00:00:00          Log Likelihood: 6660.3439
End Date: 2016-03-10 00:00:00            AIC: -13306.6879
Number of observations: 2562             BIC: -13265.748
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ======== ========
 ↪=========================
Vol Constant              -9.263     0.6113     -15.1536 0.0      (-10.4611 | -8.0649)
Component 1 p(1)          0.2491
Component 1 q(1)          0.0476
Component 2 p(1)          1.0
Component 2 q(1)          0.0935
v                         6.095
Returns Constant          0.0008     0.0386     0.0195   0.9844   (-0.075 | 0.0765)
=============================================================================================
```

The standard errors are not shown for transformed variables. You can pass through a `transformed=False` argument to `summary` to obtain this information for untransformed variables.

Let's plot the fit with `plot_fit()`:

```
model.plot_fit(figsize=(15,5))
```

And plot predictions of future conditional volatility with `plot_predict()`:

```
model.plot_predict(h=10)
```

5.8.4 Class Description

class LMGARCHM(*data*, *p*, *q*, *target*)

Long Memory Beta-t-EGARCH Models

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
p	int	The number of autoregressive lags σ^2
q	int	The number of ARCH terms ϵ^2
target	string or int	Which column of DataFrame/array to use.

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

add_leverage()

Adds a leverage term to the model, meaning volatility can respond differently to the sign of the news; see Harvey and Succarrat (2013). Conditional volatility will now follow:

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j} + \kappa \left(\text{sgn}\left(-\epsilon_{t-1}\right)\left(u_{t-1}+1\right)\right)$$

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty.

Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is(*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample(*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.8.5 References

Black, F. (1976) Studies of stock price volatility changes. In: Proceedings of the 1976 Meetings of the American Statistical Association. pp. 171–181.

Fernandez, C., & Steel, M. F. J. (1998a). On Bayesian Modeling of Fat Tails and Skewness. Journal of the American Statistical Association, 93, 359–371.

Harvey, A.C. & Chakravarty, T. (2008) Beta-t-(E)GARCH. Cambridge Working Papers in Economics 0840, Faculty of Economics, University of Cambridge, 2008. [p137]

Harvey, A.C. & Sucarrat, G. (2013) EGARCH models with fat tails, skewness and leverage. Computational Statistics and Data Analysis, Forthcoming, 2013. URL http://dx.doi.org/10.1016/j.csda.2013.09. 022. [p138, 139, 140, 143]

Mandelbrot, B.B. (1963) The variation of certain speculative prices. Journal of Business, XXXVI (1963). pp. 392–417

Nelson, D. B. (1991) Conditional heteroskedasticity in asset returns: A new approach. Econometrica 59, 347—370.

5.9 Beta Skew-t GARCH models

5.9.1 Introduction

Beta Skew-t EGARCH models were proposed by Harvey and Chakravarty (2008). They extend on GARCH models through the use of a Skew-t conditional score to drive the conditional variance. This formulation allows for increased robustness to outliers. The basic formulation follows that of a Beta-t-EGARCH model. The Skew-t distribution employed originates from Fernandez and Steel (1998).

$$f\left(\epsilon_t \mid \gamma\right) = \frac{2}{\gamma + \gamma^{-1}} \left[f\left(\frac{\epsilon_t}{\gamma}\right) I_{[0,\infty]}\left(\epsilon_t\right) + f\left(\epsilon_t \gamma\right) I_{(-\infty,0)}\left(\epsilon_t\right) \right]$$

The γ latent variable represents the degree of skewness; for $\gamma = 1$, there is no skewness, for $\gamma > 1$ there is positive skewness, and for $\gamma < 1$ there is negative skewness.

5.9.2 Developer Note

- This model type has yet to be Cythonized so performance can be slow.

5.9.3 Example

First let us load some financial time series data from Yahoo Finance:

```
import numpy as np
import pyflux as pf
import pandas as pd
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

jpm = DataReader('JPM',  'yahoo', datetime(2006,1,1), datetime(2016,3,10))
returns = pd.DataFrame(np.diff(np.log(jpm['Adj Close'].values)))
returns.index = jpm.index.values[1:jpm.index.values.shape[0]]
returns.columns = ['JPM Returns']

plt.figure(figsize=(15,5));
plt.plot(returns.index,returns);
plt.ylabel('Returns');
plt.title('JPM Returns');
```

One way to visualize the underlying volatility of the series is to plot the absolute returns $\mid y \mid$:

```
plt.figure(figsize=(15,5))
plt.plot(returns.index, np.abs(returns))
plt.ylabel('Absolute Returns')
plt.title('JP Morgan Absolute Returns');
```

There appears to be some evidence of volatility clustering over this period. Let's fit a $Beta\ Skew-t\ EGARCH(1,1)$ model using a point mass estimate z^{MLE}:

```
skewt_model = pf.SEGARCH(p=1, q=1, data=returns, target='JPM Returns')
x = skewt_model.fit()
x.summary()
```

(continues on next page)

(continued from previous page)

```
SEGARCH(1,1)
=========================================
↪==============================================
Dependent Variable: JPM Returns          Method: MLE
Start Date: 2006-01-05 00:00:00          Log Likelihood: 6664.2692
End Date: 2016-03-10 00:00:00            AIC: -13316.5384
Number of observations: 2562             BIC: -13281.4472
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========
↪========================
Vol Constant              -0.0586    0.0249     -2.3589   0.0183   (-0.1073 | -0.0099)
p(1)                      0.9932
q(1)                      0.104
Skewness                  0.9858
v                         6.0465
Returns Constant          0.0015     0.0057     0.271     0.7864   (-0.0096 | 0.0127)
=============================================================================================
```

The standard errors are not shown for transformed variables. You can pass through a `transformed=False` argument to `summary` to obtain this information for untransformed variables.

We can plot the skewness latent variable γ with *plot_z()*:

```
skewt_model.plot_z([3],figsize=(15,5))
```

So the series is slightly negatively skewed – which is consistent with the direction of skewness for most financial time series. We can plot the fit with `plot_fit()`:

```
skewt_model.plot_fit(figsize=(15,5))
```

And plot predictions of future conditional volatility with `plot_predict()`:

```
model.plot_predict(h=10)
```

If we had wanted predictions in dataframe form, we could have used `predict()`: instead.

We can also estimate a Beta-t-EGARCH model with leverage through `add_leverage()`:

```
skewt_model = pf.SEGARCH(p=1,q=1,data=returns,target='JPM Returns')
skewt_model.add_leverage()
x = skewt_model.fit()
x.summary()

SEGARCH(1,1)
=========================================
↪==============================================
Dependent Variable: JPM Returns          Method: MLE
Start Date: 2006-01-05 00:00:00          Log Likelihood: 6684.9381
End Date: 2016-03-10 00:00:00            AIC: -13355.8762
Number of observations: 2562             BIC: -13314.9364
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========
↪========================
Vol Constant              -0.1203    0.0152     -7.898    0.0      (-0.1501 | -0.0904)
```

(continues on next page)

(continued from previous page)

```
p(1)                        0.9857
q(1)                        0.1097
Leverage Term               0.0713      0.0095      7.5284    0.0       (0.0527 | 0.0899)
Skewness                    0.9984
v                           5.9741
Returns Constant            0.0004      0.0001      6.9425    0.0       (0.0003 | 0.0006)
==========================================================================================
```

We have a small leverage effect for the time series. We can plot the fit:

```
skewt_model.plot_fit(figsize=(15,5))
```

And we can plot ahead with the new model:

```
skewt_model.plot_predict(h=30,figsize=(15,5))
```

5.9.4 Class Description

class SEGARCH(*data*, *p*, *q*, *target*)

Beta Skew-t EGARCH Models

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
p	int	The number of autoregressive lags σ^2
q	int	The number of ARCH terms ϵ^2
target	string or int	Which column of DataFrame/array to use.

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

add_leverage()

Adds a leverage term to the model, meaning volatility can respond differently to the sign of the news; see Harvey and Succarrat (2013). Conditional volatility will now follow:

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j} + \kappa \left(\text{sgn}\left(-\epsilon_{t-1}\right)\left(u_{t-1}+1\right)\right)$$

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample (*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z (*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc (*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample(*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.9.5 References

Black, F. (1976) Studies of stock price volatility changes. In: Proceedings of the 1976 Meetings of the American Statistical Association. pp. 171–181.

Fernandez, C., & Steel, M. F. J. (1998a). On Bayesian Modeling of Fat Tails and Skewness. Journal of the American Statistical Association, 93, 359–371.

Harvey, A.C. & Chakravarty, T. (2008) Beta-t-(E)GARCH. Cambridge Working Papers in Economics 0840, Faculty of Economics, University of Cambridge, 2008. [p137]

Harvey, A.C. & Sucarrat, G. (2013) EGARCH models with fat tails, skewness and leverage. Computational Statistics and Data Analysis, Forthcoming, 2013. URL http://dx.doi.org/10.1016/j.csda.2013.09. 022. [p138, 139, 140, 143]

Nelson, D. B. (1991), 'Conditional heteroskedasticity in asset returns: A new approach', Econometrica 59, 347—370.

5.10 Beta Skew-t in-mean GARCH models

5.10.1 Introduction

This model type is the same as the Beta t EGARCH model, but uses a Skew t distribution. See the documentation on Beta Skew-t EGARCH models for information on this distribution and the intuition behind the additional skewness parameter.

5.10.2 Developer Note

- This model type has yet to be Cythonized so performance can be slow.

5.10.3 Example

First let us load some financial time series data from Yahoo Finance:

```
import numpy as np
import pyflux as pf
import pandas as pd
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

aapl = DataReader('AAPL',  'yahoo', datetime(2006,1,1), datetime(2016,3,10))
returns = pd.DataFrame(np.diff(np.log(aapl['Adj Close'].values)))
```

(continues on next page)

(continued from previous page)

```
returns.index = aapl.index.values[1:aapl.index.values.shape[0]]
returns.columns = ['AAPL Returns']
```

Let's fit a Beta Skew-t in-mean $EGARCH(1,1)$ model using a point mass estimate z^{MLE}:

```
skewt_model = pf.SEGARCHM(p=1, q=1, data=returns, target='AAPL Returns')
x = skewt_model.fit()
x.summary()

SEGARCHM(1,1)
=========================================
=================================================
Dependent Variable: AAPL Returns         Method: MLE
Start Date: 2006-01-05 00:00:00          Log Likelihood: 6547.0586
End Date: 2016-03-10 00:00:00            AIC: -13080.1172
Number of observations: 2562             BIC: -13039.1774
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ======== ========
=========================
Vol Constant              -0.2917    0.0074     -39.3318 0.0      (-0.3063 | -0.2772)
p(1)                      0.9653
q(1)                      0.1354
Skewness                  0.9201
v                         6.7153
Returns Constant          0.0028     0.0001     19.4147  0.0      (0.0025 | 0.0031)
GARCH-M                   0.362      0.1235     2.9323   0.0034   (0.12 | 0.604)
=============================================================================================
```

The standard errors are not shown for transformed variables. You can pass through a `transformed=False` argument to `summary` to obtain this information for untransformed variables.

We can plot the Skew-t GARCH-M latent variable with *plot_z()*:

```
skewt_model.plot_z([6],figsize=(15,5))
```

So we have a risk-premium associated with the stock. Let's plot the fit with `plot_fit()`:

```
skewt_model.plot_fit(figsize=(15,5))
```

And plot predictions of future conditional volatility with `plot_predict()`:

```
model.plot_predict(h=10)
```

5.10.4 Class Description

class SEGARCHM(*data*, *p*, *q*, *target*)

Beta Skew-t EGARCH in-mean Models

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
p	int	The number of autoregressive lags σ^2
q	int	The number of ARCH terms ϵ^2
target	string or int	Which column of DataFrame/array to use.

Attributes

latent_variables
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

add_leverage()
Adds a leverage term to the model, meaning volatility can respond differently to the sign of the news; see Harvey and Succarrat (2013). Conditional volatility will now follow:

$$\lambda_{t|t-1} = \alpha_0 + \sum_{i=1}^{p} \alpha_i \lambda_{t-i} + \sum_{j=1}^{q} \beta_j u_{t-j} + \kappa \left(\text{sgn}\left(-\epsilon_{t-1}\right)\left(u_{t-1}+1\right)\right)$$

adjust_prior(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)
Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)
Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)
Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.10.5 References

Black, F. (1976) Studies of stock price volatility changes. In: Proceedings of the 1976 Meetings of the American Statistical Association. pp. 171–181.

Fernandez, C., & Steel, M. F. J. (1998a). On Bayesian Modeling of Fat Tails and Skewness. Journal of the American Statistical Association, 93, 359–371.

Harvey, A.C. & Chakravarty, T. (2008) Beta-t-(E)GARCH. Cambridge Working Papers in Economics 0840, Faculty of Economics, University of Cambridge, 2008. [p137]

Harvey, A.C. & Sucarrat, G. (2013) EGARCH models with fat tails, skewness and leverage. Computational Statistics and Data Analysis, Forthcoming, 2013. URL http://dx.doi.org/10.1016/j.csda.2013.09. 022. [p138, 139, 140, 143]

Nelson, D. B. (1991), 'Conditional heteroskedasticity in asset returns: A new approach', Econometrica 59, 347—370.

5.11 GARCH models

5.11.1 Introduction

Generalized autoregressive conditional heteroskedasticity (GARCH) models aim to model the conditional volatility of a time series. Let r_t be the dependent variable, for example the returns of a stock in time t. We can model this series as:

$$r_t = \mu + \sigma_t \epsilon_t$$

Here mu is the expected value of r_t, σ_t is the standard deviation of r_t in time t, and ϵ_t is an error term for time t.

GARCH models are motivated by the desire to model σ_t conditional on past information. A primitive model might be a rolling standard deviation - e.g. a 30 day window - or an exponentially weighted standard deviation. A windowed model imposes an arbitrary cutoff which does not seem desirable. An EWMA is slightly more attractive, but how to select the weighting parameter λ is not immediate.

ARCH/GARCH models are an alterative model which allow for parameters to be estimated in a likelihood-based model. The basic driver of the model is a weighted average of past squared residuals. These lagged squared residuals are known as ARCH terms. Bollerslev (1986) extended the model by including lagged conditional volatility terms, creating GARCH models. Below is the formulation of a GARCH model:

$$y_t \sim N\left(\mu, \sigma_t^2\right)$$

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

We need to impose constraints on this model to ensure the volatility is over 1, in particular $\omega, \alpha, \beta > 0$. If we want to ensure stationarity, we also need to ensure $\alpha + \beta < 1$.

Once we have estimated parameters for the model, we can perform retrospective analysis on volatility, as well as make forecasts for future conditional volatility.

5.11.2 Example

First let us load some financial time series data from Yahoo Finance:

```
import numpy as np
import pyflux as pf
import pandas as pd
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

jpm = DataReader('JPM',  'yahoo', datetime(2006,1,1), datetime(2016,3,10))
returns = pd.DataFrame(np.diff(np.log(jpm['Adj Close'].values)))
returns.index = jpm.index.values[1:jpm.index.values.shape[0]]
returns.columns = ['JPM Returns']

plt.figure(figsize=(15,5));
plt.plot(returns.index,returns);
plt.ylabel('Returns');
plt.title('JPM Returns');
```

One way to visualize the underlying volatility of the series is to plot the absolute returns $| y |$:

```
plt.figure(figsize=(15,5))
plt.plot(returns.index, np.abs(returns))
plt.ylabel('Absolute Returns')
plt.title('JP Morgan Absolute Returns');
```

There appears to be some evidence of volatility clustering over this period. Let's fit a GARCH(1,1) model using a point mass estimate z^{MLE}:

```
model = pf.GARCH(returns,p=1,q=1)
x = model.fit()
x.summary()

GARCH(1,1)
=========================================
=============================================
Dependent Variable: JPM Returns          Method: MLE
Start Date: 2006-01-05 00:00:00          Log Likelihood: 6594.7911
End Date: 2016-03-10 00:00:00            AIC: -13181.5822
Number of observations: 2562             BIC: -13158.188
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========
=========================
Vol Constant              0.0
q(1)                      0.0933
p(1)                      0.9013
Returns Constant          0.0009     0.0065     0.1359    0.8919   (-0.0119 | 0.0137)
=============================================================================================
```

The standard errors are not shown for transformed variables. You can pass through a `transformed=False` argument to `summary` to obtain this information for untransformed variables.

We can plot the GARCH latent variables with *plot_z()*:

```
model.plot_z(figsize=(15,5))
```

We can plot the fit with `plot_fit()`:

And plot predictions of future conditional volatility with `plot_predict()`:

```
model.plot_predict(h=10)
```

If we had wanted predictions in DataFrame form, we could have used `predict()`:.

We can view how well we predicted using in-sample rolling prediction with `plot_predict_is()`:

```
model.plot_predict_is(h=50,figsize=(15,5))
```

5.11.3 Class Description

class GARCH(*data*, *p*, *q*, *target*)

Generalized Autoregressive Conditional Heteroskedasticity Models (GARCH)

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
p	int	The number of autoregressive lags σ^2
q	int	The number of ARCH terms ϵ^2
target	string or int	Which column of DataFrame/array to use.

Attributes

`latent_variables`
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

`adjust_prior`(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

`fit`(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

`plot_fit`(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

`plot_ppc`(*T*, *nsims*)
Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)
Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)
Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)
Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)
Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)
Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)
Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.11.4 References

Bollerslev, T. (1986). Generalized Autoregressive Conditional Heteroskedasticity. Journal of Econometrics. April, 31:3, pp. 307–27.

Engle, R.F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. Econometrica. 50(4), 987-1007.

5.12 GAS models

5.12.1 Introduction

Generalized Autoregressive Score (GAS) models are a recent class of observation-driven time-series model for non-normal data, introduced by Creal et al. (2013) and Harvey (2013). For a conditional observation density $p\left(y_t \mid x_t\right)$ with an observation y_t and a latent time-varying parameter x_t, we assume the parameter x_t follows the recursion:

$$x_t = \mu + \sum_{i=1}^{p} \phi_i x_{t-i} + \sum_{j=1}^{q} \alpha_j S\left(x_{j-1}\right) \frac{\partial \log p\left(y_{t-j} \mid x_{t-j}\right)}{\partial x_{t-j}}$$

For example, for a Poisson distribution density, where the default scaling is $\exp\left(x_j\right)$, the time-varying parameter follows:

$$x_t = \mu + \sum_{i=1}^{p} \phi_i x_{t-i} + \sum_{j=1}^{q} \alpha_j \left(\frac{y_{t-j}}{\exp\left(x_{t-j}\right)} - 1 \right)$$

These types of model can be viewed as approximations to parameter-driven state space models, and are often competitive in predictive performance. See **GAS State Space models** for a more general class of models that extend beyond the simple autoregressive form. The simple GAS models considered here in this notebook can be viewed as an approximation to non-linear ARIMA processes.

5.12.2 Example

We demonstrate an example below for count data. The data below records if a country somewhere in the world experiences a banking crisis in a given year.

```
import numpy as np
import pyflux as pf
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

data = pd.read_csv("https://vincentarelbundock.github.io/Rdatasets/csv/Ecdat/
↪bankingCrises.csv")
numpy_data = np.sum(data.iloc[:,2:73].values,axis=1)
numpy_data[np.isnan(numpy_data)] = 0
financial_crises = pd.DataFrame(numpy_data)
financial_crises.index = data.year
financial_crises.columns = ["Number of banking crises"]

plt.figure(figsize=(15,5))
plt.plot(financial_crises)
plt.ylabel("Count")
plt.xlabel("Year")
plt.title("Number of banking crises across the world")
plt.show()
```

Here we specify an arbitrary $GAS(2,0,2)$ model with a `Poisson()` family:

```
model = pf.GAS(ar=2, sc=2, data=financial_crises, family=pf.Poisson())
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit("MLE")
x.summary()

PoissonGAS (2,0,2)
=========================================
↪=============================================
Dependent Variable: No of crises         Method: MLE
Start Date: 1802                         Log Likelihood: -497.8648
End Date: 2010                           AIC: 1005.7297
Number of observations: 209              BIC: 1022.4413
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ======== ========
↪=========================
Constant                  0.0        0.0267     0.0       1.0      (-0.0524 | 0.0524)
AR(1)                     0.4502     0.0552     8.1544    0.0      (0.342 | 0.5584)
AR(2)                     0.4595     0.0782     5.8776    0.0      (0.3063 | 0.6128)
SC(1)                     0.2144     0.0241     8.8929    0.0      (0.1671 | 0.2616)
SC(2)                     0.0571     0.0042     13.5323   0.0      (0.0488 | 0.0654)
=============================================================================================
```

We can plot the latent variables z^{MLE}: using the *plot_z()*: method:

```
model.plot_z(figsize=(15,5))
```

We can plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

We can get an idea of the performance of our model by using rolling in-sample prediction through the `plot_predict_is()`: method:

```
model.plot_predict_is(h=20, fit_once=True, figsize=(15,5))
```

If we want to plot predictions, we can use the `plot_predict()`: method:

```
model.plot_predict(h=10, past_values=30, figsize=(15,5))
```

If we want the predictions in a DataFrame form, then we can just use the `predict()`: method.

5.12.3 Class Description

class GAS (*data*, *ar*, *sc*, *integ*, *target*, *family*)

Generalized Autoregressive Score Models (GAS).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
ar	int	The number of autoregressive lags
sc	int	The number of score function lags
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty.

Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is (*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample (*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z (*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc (*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)
Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.12.4 References

Creal, D; Koopman, S.J.; Lucas, A. (2013). Generalized Autoregressive Score Models with Applications. Journal of Applied Econometrics, 28(5), 777–795. doi:10.1002/jae.1279.

Harvey, A.C. (2013). Dynamic Models for Volatility and Heavy Tails: With Applications to Financial and Economic Time Series. Cambridge University Press.

5.13 GAS local level models

5.13.1 Introduction

The principle behind score-driven models is that the linear update $y_t - \theta_t$, that the Kalman filter relies upon, can be robustified by replacing it with the conditional score of a non-normal distribution. For this reason, any class of traditional state space model has a score-driven equivalent.

For example, consider a local level model in this framework:

$$p\left(y_t \mid \theta_t\right)$$

$$\theta_t = \theta_{t-1} + \eta H_{t-1}^{-1} S_{t-1}$$

Here the underlying parameter θ_t resembles a random walk, while the score S_{t-1} drives the equation. The first term H_{t-1} is a scaling term that can incorporate second-order information, as per a Newton update. The term η is a learning rate or scaling term, and is the latent variable which is estimated in the model.

5.13.2 Example

We will use data on the number of goals scored by soccer teams Nottingham Forest and Derby in their head-to-head matches from the beginning of their competitive history. We are interested to know whether these games have become more or less high scoring over time.

```
import numpy as np
import pandas as pd
import pyflux as pf
import matplotlib.pyplot as plt
%matplotlib inline

eastmidlandsderby = pd.read_csv('http://www.pyflux.com/notebooks/eastmidlandsderby.csv
↪')
total_goals = pd.DataFrame(eastmidlandsderby['Forest'] + eastmidlandsderby['Derby'])
total_goals.columns = ['Total Goals']
plt.figure(figsize=(15,5))
plt.title("Total Goals in the East Midlands Derby")
plt.xlabel("Games Played");
plt.ylabel("Total Goals");
plt.plot(total_goals);
```

Here can fit a GAS Local Level model with a `Poisson()` family:

```
model = pf.GASLLEV(data=total_goals, family=pf.Poisson())
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit("MLE")
x.summary()

Poisson GAS LLM
=========================================␣
↪================================================
Dependent Variable: Total Goals          Method: MLE
Start Date: 1                            Log Likelihood: -198.7993
End Date: 96                             AIC: 399.5985
Number of observations: 96               BIC: 402.1629
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ======== ========␣
↪=========================
SC(1)                     0.1929     0.0468     4.1252   0.0      (0.1013 | 0.2846)
=============================================================================================
```

We can plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

If we want to plot predictions, we can use the `plot_predict()`: method:

```
model.plot_predict(h=10, past_values=30, figsize=(15,5))
```

If we want the predictions in a DataFrame form, then we can just use the `predict()`: method.

5.13.3 Class Description

class GASLLEV(*data*, *integ*, *target*, *family*)

GAS Local Level Models.

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)
Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)
Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)
Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc (*T*, *nsims*)
Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict (*h*, *intervals=False*)
Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)
Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.13.4 References

Creal, D; Koopman, S.J.; Lucas, A. (2013). Generalized Autoregressive Score Models with Applications. Journal of Applied Econometrics, 28(5), 777–795. doi:10.1002/jae.1279.

Harvey, A.C. (2013). Dynamic Models for Volatility and Heavy Tails: With Applications to Financial and Economic Time Series. Cambridge University Press.

5.14 GAS local linear trend models

5.14.1 Introduction

The principle behind score-driven models is that the linear update $y_t - \theta_t$, that the Kalman filter relies upon, can be robustified by replacing it with the conditional score of a non-normal distribution. For this reason, any class of traditional state space model has a score-driven equivalent.

For example, consider a local linear model in this framework:

$$p\left(y_t \mid \mu_t\right)$$

$$\mu_t = \mu_{t-1} + \beta_{t-1} + \eta_1 H_{t-1}^{-1} S_{t-1}$$

$$\beta_t = \beta_{t-1} + \eta_2 H_{t-1}^{-1} S_{t-1}$$

Here η represents the two learning rates or scaling terms, and are the latent variables which are estimated in the model.

5.14.2 Example

We will construct a local linear trend model for US GDP using a skew t-distribution. Here is the data:

```
growthdata = pd.read_csv('http://www.pyflux.com/notebooks/GDPC1.csv')
USgrowth = pd.DataFrame(np.log(growthdata['VALUE']))
USgrowth.index = pd.to_datetime(growthdata['DATE'])
USgrowth.columns = ['Logged US Real GDP']
plt.figure(figsize=(15,5))
plt.plot(USgrowth.index, USgrowth)
plt.ylabel('Real GDP')
plt.title('US Logged Real GDP');
```

Here can fit a GAS Local Linear Trend model with a `Skewt()` family:

```
model = pf.GASLLT(data=USgrowth-np.mean(USgrowth),family=pf.Skewt())
```

Next we estimate the latent variables. For this example we will use a BBVI estimate z^{BBVI}:

```
x = model.fit('BBVI', iterations=20000, record_elbo=True)
10% done : ELBO is 70.1403732024, p(y,z) is 82.4582067151, q(z) is 12.3178335126
20% done : ELBO is 71.5399641383, p(y,z) is 84.3269580596, q(z) is 12.7869939213
30% done : ELBO is 95.3663747496, p(y,z) is 108.551290696, q(z) is 13.1849159469
40% done : ELBO is 124.357073241, p(y,z) is 138.132000673, q(z) is 13.7749274322
50% done : ELBO is 144.111819073, p(y,z) is 158.386802182, q(z) is 14.274983109
60% done : ELBO is 164.792526642, p(y,z) is 179.422645151, q(z) is 14.6301185085
70% done : ELBO is 178.18148403, p(y,z) is 193.190633108, q(z) is 15.0091490782
80% done : ELBO is 206.095112618, p(y,z) is 221.579871841, q(z) is 15.4847592232
90% done : ELBO is 210.854594358, p(y,z) is 226.705793141, q(z) is 15.8511987823
100% done : ELBO is 226.965067448, p(y,z) is 243.29536546, q(z) is 16.3302980111

Final model ELBO is 224.286972026
```

We can plot the ELBO with `plot_elbo()`: on the results object:

```
x.plot_elbo(figsize=(15,7))
```

We can plot the latent variables with *plot_z()*:

```
model.plot_z([0,1,3])
```

```
model.plot_z([2,4])
```

The states are stored as an attribute `states` in the results object. Let's plot the trend state:

```
plt.figure(figsize=(15,5))
plt.title("Local Trend for US GDP")
plt.ylabel("Trend")
plt.plot(USgrowth.index[21:],x.states[1][20:]);
```

This reflects the underlying growth potential of the US economy.

We can also calculate the average growth rate for a forward forecast:

```
print("Average growth rate for this period is")
print(str(round(100*np.mean(np.exp(np.diff(model.predict(h=4)['Logged US Real GDP'].
↪values)) - 1),3)) + "%")

Average growth rate for this period is
0.504%
```

5.14.3 Class Description

class GASLLT(*data*, *integ*, *target*, *family*)

GAS Local Linear Trend Models.

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is(*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample(*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.14.4 References

Creal, D; Koopman, S.J.; Lucas, A. (2013). Generalized Autoregressive Score Models with Applications. Journal of Applied Econometrics, 28(5), 777–795. doi:10.1002/jae.1279.

Harvey, A.C. (2013). Dynamic Models for Volatility and Heavy Tails: With Applications to Financial and Economic Time Series. Cambridge University Press.

5.15 GAS ranking models

5.15.1 Introduction

GAS ranking models can be used for pairwise comparisons or competitive activities. The $GASRank$ model of Taylor, R. (2016) models a point difference between two competitors. The point difference is assumed to follow a particular distribution. For example, suppose that the point difference μ_t is normally distributed, then we can model the point difference as the location parameter μ:

$$\mu_t = \delta + \alpha_{t,i} - \alpha_{t,j}$$

Where δ_t is a 'home advantage' latent variable, i and j refer to home and away competitors, and α contains the team power rankings. The power rankings are modelled as random walk processes between each match:

$$\alpha_{k,i} = \alpha_{k-1,i} + \eta U_{k-1,i}$$

$$\alpha_{k,j} = \alpha_{k-1,j} - \eta U_{k-1,j}$$

Where k is the game index, η is a learning rate or scaling parameter to be estimated.

The model can be extended to a two component model where each competitor has two aspects to their 'team'. For example, we might model an NFL team along with the Quarterback power rankings in the same game:

$$\mu_t = \delta + \alpha_{t,i} - \alpha_{t,j} + \gamma_{t,i} - \gamma_{t,j}$$

Here γ represents the power ranking of the second component. The secondary component power rankings are modelled as random walk processes between each match:

$$\gamma_{k,i} = \gamma_{k-1,i} + \eta U_{2k-1,i}$$

$$\gamma_{k,j} = \gamma_{k-1,j} - \eta U_{2k-1,j}$$

5.15.2 Developer Note

- This model type has yet to be cythonized, so performance can be slow.

5.15.3 Example

We will model the point difference in NFL games with a simple model. Here is the data:

```
import pyflux as pf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv("nfl_data_new.csv")
data["PointsDiff"] = data["HomeScore"] - data["AwayScore"]
data.columns
Index(['Unnamed: 0', 'AFirstDowns', 'AFumbles0', 'AFumbles1', 'AIntReturns0',
    'AIntReturns1', 'AKickoffReturns0', 'AKickoffReturns1', 'ANetPassYards',
    'APassesAttempted', 'APassesCompleted', 'APassesIntercepted',
    'APenalties0', 'APenalties1', 'APossession', 'APuntReturns0',
    'APuntReturns1', 'APunts0', 'APunts1', 'AQB', 'ARushes0', 'ARushes1',
    'ASacked0', 'ASacked1', 'AwayScore', 'AwayTeam', 'Date', 'HFirstDowns',
    'HFumbles0', 'HFumbles1', 'HIntReturns0', 'HIntReturns1',
    'HKickoffReturns0', 'HKickoffReturns1', 'HNetPassYards',
    'HPassesAttempted', 'HPassesCompleted', 'HPassesIntercepted',
    'HPenalties0', 'HPenalties1', 'HPossession', 'HPuntReturns0',
    'HPuntReturns1', 'HPunts0', 'HPunts1', 'HQB', 'HRushes0', 'HRushes1',
    'HSacked0', 'HSacked1', 'HomeScore', 'HomeTeam', 'Postseason',
    'PointsDiff'],
   dtype='object')
```

We can plot the point difference to get an idea of potentially suitable distributions:

```
data = pd.read_csv("nfl_data_new.csv")
data["PointsDiff"] = data["HomeScore"] - data["AwayScore"]
plt.figure(figsize=(15,7))
plt.ylabel("Frequency")
plt.xlabel("Points Difference")
plt.hist(data["PointsDiff"],bins=20);
```

We will use a `pf.Normal()` families, although we could try a family with heavier tails also. We setup the $GASRank$ model, referring to the appropriate columns in our DataFrame:

```
model = pf.GASRank(data=data,team_1="HomeTeam", team_2="AwayTeam",
                   score_diff="PointsDiff", family=pf.Normal())
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit()
x.summary()

NormalGAS Rank
=========================================
 ↪================================================
Dependent Variable: PointsDiff           Method: MLE
Start Date: 0                            Log Likelihood: -10825.1703
End Date: 2667                           AIC: 21656.3406
Number of observations: 2668             BIC: 21674.0079
==========================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========
 ↪=========================
```

(continues on next page)

(continued from previous page)

```
Constant                  2.2405     0.2547     8.795    0.0       (1.7412 | 2.7398)
Ability Scale             0.0637     0.0058     10.9582  0.0       (0.0523 | 0.0751)
Normal Scale              13.9918
==============================================================================================
```

Once we have fit the model we can plot the power rankings of the teams in our DataFrame over their competitive history using `plot_abilities()`:

```
model.plot_abilities(["Denver Broncos", "Green Bay Packers", "New England Patriots",
                      "Carolina Panthers"],figsize=(15,8))
```

```
model.plot_abilities(["San Francisco 49ers", "Oakland Raiders", "San Diego Chargers"],
                       figsize=(15,8))
```

We can predict the point difference between two competitors in the future using `predict()`:

```
model.predict("Denver Broncos","Carolina Panthers",neutral=True)
array(-4.886816685966575)
```

Our DataFrame also has information on quarterbacks. Let's extend our model with a second component by including quarterbacks in the model:

```
model.add_second_component("HQB","AQB")
x = model.fit()
x.summary()

NormalGAS Rank
=========================================
↪=================================================
Dependent Variable: PointsDiff           Method: MLE
Start Date: 0                            Log Likelihood: -10799.4544
End Date: 2667                           AIC: 21606.9087
Number of observations: 2668             BIC: 21630.4651
==============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|     95% C.I.
========================= ========== ========== ========= ========
↪=========================
Constant                  2.2419     0.2516     8.9118    0.0       (1.7488 | 2.735)
Ability Scale 1           0.0186     0.0062     2.9904    0.0028    (0.0064 | 0.0307)
Ability Scale 2           0.0523     0.0076     6.8492    0.0       (0.0373 | 0.0673)
Normal Scale              13.8576
==============================================================================================
```

We can plot the power rankings of the QBs in our DataFrame over their competitive history using `plot_abilities()`:

```
model.plot_abilities(["Cam Newton", "Peyton Manning"],1,figsize=(15,8))
```

We can predict the point difference between two competitors in the future using `predict()`:

```
model.predict("Denver Broncos","Carolina Panthers","Peyton Manning","Cam Newton",
↪neutral=True)
array(-7.33759714587138)
```

And some more power rankings for fan interest...

```
model.plot_abilities(["Aaron Rodgers", "Tom Brady", "Russell Wilson"],1,figsize=(15,
↪8))
```

```
model.plot_abilities(["Peyton Manning","Michael Vick", "David Carr", "Carson Palmer"
                      ,"Eli Manning","Alex Smith","JaMarcus Russell","Matthew Stafford"
                      ,"Sam Bradford","Cam Newton","Andrew Luck","Jameis Winston"],1,
                      figsize=(15,8))
```

5.15.4 Class Description

class GASRank (*data*, *team_1*, *team_2*, *family*, *score_diff*)

Generalized Autoregressive Score Ranking Models (GASRank).

Parameter	Type	Description
data	pd.dataframe	Containing the competitive data
team_1	string	Column name for home team names
team_2	string	Column name for away team names
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`
score_diff	string	Column name for the point difference

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

add_second_component (*team_1*, *team_2*)

Adds a second component to the model

Parameter	Type	Description
team_1	string	Column name for team 1 second component
team_2	string	Column name for team 2 second component

Returns : void - changes model to a second component model

adjust_prior (*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit (*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_abilities(*team_ids*)

Plots power rankings of the model components. Optional arguments include *figsize*, the dimensions of the figure to plot.

Parameter	Type	Description
team_ids	list	Of strings (team names) or indices

For a two component model, arguments are:

Parameter	Type	Description
team_ids	list	Of strings (team names) or indices
component_id	int	0 for component 1, 1 for component 2

Returns : void - shows a matplotlib plot

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

predict(*team_1*, *team_2*, *neutral=False*)

Returns predicted point differences. For a one component model, arguments are:

Parameter	Type	Description
team_1	string or int	If string, team name, else team index
team_2	string or int	If string, team name, else team index
neutral	boolean	If True, disables home advantage

For a two component model, arguments are:

Parameter	Type	Description
team_1	string or int	If string, team name, else team index
team_2	string or int	If string, team name, else team index
team1b	string or int	If string, team 1, player 2 name
team2b	string or int	If string, team 2, player 2 name
neutral	boolean	If True, disables home advantage

Returns : np.ndarray - point difference predictions

5.15.5 References

Creal, D; Koopman, S.J.; Lucas, A. (2013). Generalized Autoregressive Score Models with Applications. Journal of Applied Econometrics, 28(5), 777–795. doi:10.1002/jae.1279.

Harvey, A.C. (2013). Dynamic Models for Volatility and Heavy Tails: With Applications to Financial and Economic Time Series. Cambridge University Press.

Taylor, R. (2016). A Tour of Time Series Analysis (and a model for predicting NFL games). https://github.com/RJT1990/PyData2016-SanFrancisco

5.16 GAS regression models

5.16.1 Introduction

The principle behind score-driven models is that the linear update $y_t - \theta_t$, that the Kalman filter relies upon, can be robustified by replacing it with the conditional score of a non-normal distribution. For this reason, any class of traditional state space model has a score-driven equivalent.

For example, consider a dynamic regression model in this framework:

$$p\left(y_t \mid \theta_t\right)$$

$$\theta_t = \boldsymbol{x}_t^{'}\boldsymbol{\beta}_t$$

$$\boldsymbol{\beta}_t = \boldsymbol{\beta}_{t-1} + \boldsymbol{\eta} H_{t-1}^{-1} S_{t-1}$$

Here η represents the learning rates or scaling terms, and are the latent variables which are estimated in the model.

5.16.2 Example

We will use a dynamic t regression to extract a dynamic β for a stock. Using t-distributed errors is more robust than a normality assumption, that could be obtained with a Kalman filter. The β captures the amount of systematic risk in the stock - i.e. the stock's relationship with the market.

```
from pandas_datareader import DataReader
from datetime import datetime

a = DataReader('AMZN',  'yahoo', datetime(2012,1,1), datetime(2016,6,1))
a_returns = pd.DataFrame(np.diff(np.log(a['Adj Close'].values)))
a_returns.index = a.index.values[1:a.index.values.shape[0]]
a_returns.columns = ["Amazon Returns"]

spy = DataReader('SPY',  'yahoo', datetime(2012,1,1), datetime(2016,6,1))
spy_returns = pd.DataFrame(np.diff(np.log(spy['Adj Close'].values)))
spy_returns.index = spy.index.values[1:spy.index.values.shape[0]]
spy_returns.columns = ['S&P500 Returns']

one_mon = DataReader('DGS1MO', 'fred',datetime(2012,1,1), datetime(2016,6,1))
one_day = np.log(1+one_mon)/365
```

(continues on next page)

(continued from previous page)

```
returns = pd.concat([one_day,a_returns,spy_returns],axis=1).dropna()
excess_m = returns["Amazon Returns"].values - returns['DGS1MO'].values
excess_spy = returns["S&P500 Returns"].values - returns['DGS1MO'].values
final_returns = pd.DataFrame(np.transpose([excess_m,excess_spy, returns['DGS1MO'].
↪values]))
final_returns.columns=["Amazon","SP500","Risk-free rate"]
final_returns.index = returns.index

plt.figure(figsize=(15,5))
plt.title("Excess Returns")
x = plt.plot(final_returns);
plt.legend(iter(x), final_returns.columns);
```

We can fit a GAS Regression model with a `t()` family:

```
model = pf.GASReg('Amazon ~ SP500', data=final_returns, family=pf.t())
```

Next we estimate the latent variables. For this example we will use a Maximum Likelihood estimate z^{MLE}:

```
x = model3.fit()
x.summary()

t GAS Regression
=========================================␣
↪================================================
Dependent Variable: Amazon                Method: MLE
Start Date: 2012-01-04 00:00:00           Log Likelihood: 3158.435
End Date: 2016-06-01 00:00:00             AIC: -6308.87
Number of observations: 1101              BIC: -6288.8541
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ======== ========␣
↪=========================
Scale 1                   0.0
Scale SP500               0.0474
t Scale                   0.0095
v                         2.8518
=============================================================================================
```

We can plot the fit with `plot_fit()`:

```
model.plot_fit(intervals=False,figsize=(15,15))
```

One of the advantages of using a GASRegression rather than a Kalman filtered Dynamic Linear Regression is that the GASRegression with t errors is more robust to outliers. We do not produce the whole analysis here, but for the same data, the filtered estimates are compared below:

5.16.3 Class Description

class GASReg(*data*, *formula*, *target*, *family*)

Generalized Autoregressive Score Regression Models (GASReg).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
formula	string	Patsy notation specifying the regression
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

`latent_variables`
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

`adjust_prior`(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

`fit`(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

`plot_fit`(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

`plot_ppc`(*T*, *nsims*)
Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *oos_data*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the h first rows from *oos_data* and take the values for the exogenous variables that you asked for in the patsy formula.

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *oos_data*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
intervals	boolean	Whether to return prediction intervals

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the 5 first rows from *oos_data* and take the values for the exogenous variables that you specified as exogenous variables in the patsy formula.

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is(*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample(*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.16.4 References

Creal, D; Koopman, S.J.; Lucas, A. (2013). Generalized Autoregressive Score Models with Applications. Journal of Applied Econometrics, 28(5), 777–795. doi:10.1002/jae.1279.

Harvey, A.C. (2013). Dynamic Models for Volatility and Heavy Tails: With Applications to Financial and Economic Time Series. Cambridge University Press.

5.17 GASX models

5.17.1 Introduction

GASX models extend GAS models by including exogenous factors X. For a conditional observation density $p\left(y_t \mid \theta_t\right)$ with an observation y_t and a latent time-varying parameter θ_t, we assume the parameter θ_t follows the recursion:

$$\theta_t = \mu + \sum_{k=1}^{K} \beta_k X_{t,k} + \sum_{i=1}^{p} \phi_i \theta_{t-i} + \sum_{j=1}^{q} \alpha_j S\left(x_{j-1}\right) \frac{\partial \log p\left(y_{t-j} \mid \theta_{t-j}\right)}{\partial \theta_{t-j}}$$

For example, for the Poisson family, where the default scaling is $\exp\left(\theta_t\right)$, the time-varying latent variable follows:

$$\theta_t = \mu + \sum_{k=1}^{K} \beta_k X_{t,k} + \sum_{i=1}^{p} \phi_i \theta_{t-i} + \sum_{j=1}^{q} \alpha_j \left(\frac{y_{t-j}}{\exp\left(\theta_{t-j}\right)} - 1 \right)$$

The model can be viewed as an approximation to a non-linear ARIMAX model.

5.17.2 Example

Below we estimate the β for a stock – the systematic (market) component of returns – using a heavy tailed distribution and some short-term autoregressive effects. First let's load some data:

```
from pandas_datareader.data import DataReader
from datetime import datetime

a = DataReader('AMZN',  'yahoo', datetime(2012,1,1), datetime(2016,6,1))
a_returns = pd.DataFrame(np.diff(np.log(a['Adj Close'].values)))
a_returns.index = a.index.values[1:a.index.values.shape[0]]
a_returns.columns = ["Amazon Returns"]

spy = DataReader('SPY',  'yahoo', datetime(2012,1,1), datetime(2016,6,1))
spy_returns = pd.DataFrame(np.diff(np.log(spy['Adj Close'].values)))
spy_returns.index = spy.index.values[1:spy.index.values.shape[0]]
spy_returns.columns = ['S&P500 Returns']

one_mon = DataReader('DGS1MO', 'fred',datetime(2012,1,1), datetime(2016,6,1))
one_day = np.log(1+one_mon)/365

returns = pd.concat([one_day,a_returns,spy_returns],axis=1).dropna()
excess_m = returns["Amazon Returns"].values - returns['DGS1MO'].values
excess_spy = returns["S&P500 Returns"].values - returns['DGS1MO'].values
final_returns = pd.DataFrame(np.transpose([excess_m,excess_spy, returns['DGS1MO'].
↪values]))
final_returns.columns=["Amazon","SP500","Risk-free rate"]
```

(continues on next page)

(continued from previous page)

```
final_returns.index = returns.index

plt.figure(figsize=(15,5))
plt.title("Excess Returns")
x = plt.plot(final_returns);
plt.legend(iter(x), final_returns.columns);
```

Below we estimate a point mass estimate z^{MLE} of the latent variables for a $GASX(1,1)$ model:

```
model = pf.GASX(formula="Amazon~SP500",data=final_returns,ar=1,sc=1,family=pf.
↪GASSkewt())
x = model.fit()
x.summary()

Skewt GASX(1,0,1)
=========================================␣
↪==============================================
Dependent Variable: Amazon                Method: MLE
Start Date: 2012-01-05 00:00:00           Log Likelihood: 3165.9237
End Date: 2016-06-01 00:00:00             AIC: -6317.8474
Number of observations: 1100              BIC: -6282.8259
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|     95% C.I.
========================= ========== ========== ========= ========␣
↪=========================
AR(1)                     0.0807     0.0202     3.9956    0.0001    (0.0411 | 0.1203)
SC(1)                     -0.0       0.0187     -0.0001   0.9999    (-0.0367 | 0.0367)
Beta 1                    -0.0005    0.0249     -0.0184   0.9853    (-0.0493 | 0.0484)
Beta SP500                1.2683     0.0426     29.7473   0.0       (1.1848 | 1.3519)
Skewness                  1.017
Skewt Scale               0.0093
v                         2.7505
=============================================================================================
WARNING: Skew t distribution is not well-suited for MLE or MAP inference
Workaround 1: Use a t-distribution instead for MLE/MAP
Workaround 2: Use M-H or BBVI inference for Skew t distribution
```

The results table warns us about using the Skew t distribution. This choice of family can sometimes be unstable, so we may want to opt for a t-distribution instead. But in this case, we seem to have obtained sensible results. We can plot the constant and the GAS latent variables by referencing their indices with *plot_z()*:

```
model.plot_z(indices=[0,1,2])
```

Similarly we can plot β:

```
model.plot_z(indices=[3])
```

Our β_{AMZN} estimate is above 1.0 (fairly strong systematic risk). Let us plot the model fit and the systematic component of returns with `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

5.17.3 Class Description

class GASX(*data*, *formula*, *ar*, *sc*, *integ*, *target*, *family*)

Generalized Autoregressive Score Exogenous Variable Models (GASX).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
formula	string	Patsy notation specifying the regression
ar	int	The number of autoregressive lags
sc	int	The number of score function lags
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *oos_data*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the h first rows from *oos_data* and take the values for the exogenous variables that you asked for in the patsy formula.

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *oos_data*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
intervals	boolean	Whether to return prediction intervals

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the 5 first rows from *oos_data* and take the values for the exogenous variables that you specified as exogenous variables in the patsy formula.

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is(*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample(*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

5.17.4 References

Creal, D; Koopman, S.J.; Lucas, A. (2013). Generalized Autoregressive Score Models with Applications. Journal of Applied Econometrics, 28(5), 777–795. doi:10.1002/jae.1279.

Harvey, A.C. (2013). Dynamic Models for Volatility and Heavy Tails: With Applications to Financial and Economic Time Series. Cambridge University Press.

5.18 GP-NARX models

5.18.1 Example

```
import numpy as np
import pandas as pd
import pyflux as pf

USgrowth = #somequarterlyGDPgrowthdatahere
model = pf.GPNARX(USgrowth, ar=4, kernel_type='OU')
```

5.18.2 Class Arguments

class GPNARX(*data*, *ar*, *kernel_type*, *integ*, *target*)

data
pd.DataFrame or array-like : the time-series data

ar
int : the number of autoregressive terms

kernel_type
string : the type of kernel; one of ['SE','RQ','OU','Periodic','ARD']

integ
int : Specifies how many time to difference the time series.

target
string (data is DataFrame) or int (data is np.array) : which column to use as the time series. If None, the first column will be chosen as the data.

5.18.3 Class Methods

adjust_prior(*index*, *prior*)
Adjusts the priors of the model. **index** can be an int or a list. **prior** is a prior object, such as Normal(0,3).

Here is example usage for `adjust_prior()`:

```
import pyflux as pf

# model = ... (specify a model)
model.list_priors()
model.adjust_prior(2,pf.Normal(0,1))
```

fit(*method*, ***kwargs*)

Estimates latent variables for the model. Returns a Results object. **method** is an inference/estimation option; see Bayesian Inference and Classical Inference sections for options. If no **method** is provided then a default will be used.

Optional arguments are specific to the **method** you choose - see the documentation for these methods for more detail.

Here is example usage for `fit()`:

```
import pyflux as pf

# model = ... (specify a model)
model.fit("M-H",nsims=20000)
```

plot_fit(***kwargs*)

Graphs the fit of the model.

Optional arguments include **figsize** - the dimensions of the figure to plot.

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty. **indices** is a list referring to the latent variable indices that you want ot plot. Figsize specifies how big the plot will be.

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model. **h** is an int of how many steps ahead to predict. **past_values** is an int of how many past values of the series to plot. **intervals** is a bool on whether to include confidence/credibility intervals or not.

Optional arguments include **figsize** - the dimensions of the figure to plot.

plot_predict_is(*h*, *fit_once*, ***kwargs*)

Plots in-sample rolling predictions for the model. **h** is an int of how many previous steps to simulate performance on. **fit_once** is a boolean specifying whether to fit the model once at the beginning of the period (True), or whether to fit after every step (False).

Optional arguments include **figsize** - the dimensions of the figure to plot.

predict(*h*)

Returns DataFrame of model predictions. **h** is an int of how many steps ahead to predict.

predict_is(*h*, *fit_once*)

Returns DataFrame of in-sample rolling predictions for the model. **h** is an int of how many previous steps to simulate performance on. **fit_once** is a boolean specifying whether to fit the model once at the beginning of the period (True), or whether to fit after every step (False).

5.19 Gaussian Local Level models

5.19.1 Introduction

Gaussian state space models - often called structural time series or unobserved component models - provide a way to decompose a time series into several distinct components. These components can be extracted in closed form using the Kalman filter if the errors are jointly Gaussian, and parameters can be estimated via the prediction error decomposition and Maximum Likelihood.

One classic univariate structural time series model is the **local level model**. We can write this as a combination of a time-varying level and an irregular term:

$$y_t = \mu_t + \epsilon_t$$

$$\mu_t = \mu_{t-1} + \eta_t$$

$$\epsilon_t \sim N\left(0, \sigma_\epsilon^2\right)$$

$$\eta_t \sim N\left(0, \sigma_\eta^2\right)$$

5.19.2 Example

We will use data on the number of goals scored by soccer teams Nottingham Forest and Derby in their head-to-head matches from the beginning of their competitive history. We are interested to know whether these games have become more or less high scoring over time.

```
import numpy as np
import pyflux as pf
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

nile = pd.read_csv('https://vincentarelbundock.github.io/Rdatasets/csv/datasets/Nile.
↪csv')
nile.index = pd.to_datetime(nile['time'].values,format='%Y')
plt.figure(figsize=(15,5))
plt.plot(nile.index,nile['Nile'])
plt.ylabel('Discharge Volume')
plt.title('Nile River Discharge');
plt.show()
```

Here define a Local Level model as follows:

```
model = pf.LLEV(data=nile, target='Nile')
```

We can also use the higher-level wrapper which allows us to specify the family, although if we pick a non-Gaussian family then the model will be estimated in a different way (not through the Kalman filter):

```
model = pf.LocalLevel(data=nile, target='Nile', family=pf.Normal())
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit()
x.summary()

LLEV
=========================================
↪=================================================
Dependent Variable: Nile                 Method: MLE
Start Date: 1871-01-01 00:00:00          Log Likelihood: -641.5238
End Date: 1970-01-01 00:00:00            AIC: 1287.0476
Number of observations: 100              BIC: 1292.258
=============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ======== ========
↪=========================
Sigma^2 irregular         15098.5722
Sigma^2 level             1469.11317
=============================================================================================
```

We can plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

The model adapts to the lower level at the beginning of the 20th century.

We can use the Durbin and Koopman (2002) simulation smoother to simulate draws from the local level state, using `simulation_smoother()`:

```
plt.figure(figsize=(15,5))
for i in range(10):
    plt.plot(model.index, model.simulation_smoother(
            model.latent_variables.get_z_values())[0][0:model.index.shape[0]])
plt.show()
```

If we want to plot rolling in-sample predictions, we can use the `plot_predict_is()`: method:

```
model.plot_predict_is(h=20,figsize=(15,5))
```

We can view out-of-sample predictions using `plot_predict()`:

```
model.plot_predict(h=5,figsize=(15,5))
```

If we want the predictions in a DataFrame form, then we can just use the `predict()`: method.

5.19.3 Class Description

class LLEV(*data*, *integ*, *target*)

Local Level Models.

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.

Attributes

latent_variables
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)
Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty.

Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)

Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

simulation_smoother (*beta*)

Returns np.ndarray of draws of the data from the Durbin and Koopman (2002) simulation smoother.

Parameter	Type	Description
beta	np.array	np.array of latent variables

Recommended just to use model.latent_variables.get_z_values() for the beta input, if you have already fit a model.

Returns : np.ndarray - samples from simulation smoother

5.19.4 References

Durbin, J. and Koopman, S. J. (2002). A simple and efficient simulation smoother for state space time series analysis. Biometrika, 89(3):603–615.

Harvey, A. C. (1989). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge.

5.20 Gaussian Local Linear Trend models

5.20.1 Introduction

Gaussian state space models - often called structural time series or unobserved component models - provide a way to decompose a time series into several distinct components. These components can be extracted in closed form using the Kalman filter if the errors are jointly Gaussian, and parameters can be estimated via the prediction error decomposition and Maximum Likelihood.

One classic univariate structural time series model is the **local linear trend model**. We can write this as a combination of a time-varying level, time-varying trend and an irregular term:

$$y_t = \mu_t + \epsilon_t$$

$$\mu_t = \beta_{t-1} + \mu_{t-1} + \eta_t$$

$$\beta_t = \beta_{t-1} + \zeta_t$$

$$\epsilon_t \sim N\left(0, \sigma_\epsilon^2\right)$$

$$\eta_t \sim N\left(0, \sigma_\eta^2\right)$$

$$\zeta_t \sim N\left(0, \sigma_\zeta^2\right)$$

5.20.2 Example

We will use data on logged US Real GDP.

```
growthdata = pd.read_csv('http://www.pyflux.com/notebooks/GDPC1.csv')
USgrowth = pd.DataFrame(np.log(growthdata['VALUE']))
USgrowth.index = pd.to_datetime(growthdata['DATE'])
USgrowth.columns = ['Logged US Real GDP']
plt.figure(figsize=(15,5))
plt.plot(USgrowth.index, USgrowth)
plt.ylabel('Real GDP')
plt.title('US Real GDP');
```

Here define a Local Linear Trend model as follows:

```
model = pf.LLT(data=USgrowth)
```

We can also use the higher-level wrapper which allows us to specify the family, although if we pick a non-Gaussian family then the model will be estimated in a different way (not through the Kalman filter):

```
model = pf.LocalTrend(data=USgrowth, family=pf.Normal())
```

Next we estimate the latent variables. For this example we will use a maximum likelihood point mass estimate z^{MLE}:

```
x = model.fit()
x.summary()

LLT
==========================================
==============================================
Dependent Variable: Logged US Real GDP   Method: MLE
```

(continues on next page)

(continued from previous page)

```
Start Date: 1947-01-01 00:00:00          Log Likelihood: 877.3334
End Date: 2015-10-01 00:00:00            AIC: -1748.6667
Number of observations: 276              BIC: -1737.8055
============================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ======== ========␣
↪=========================
Sigma^2 irregular         3.306e-07
Sigma^2 level             5.41832e-0
Sigma^2 trend             1.55224e-0
============================================================================================
```

We can plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

The trend picks out the underlying local growth rate in the series. Together with the local level this constitutes the smoothed series. We can use the simulation smoother to simulate draws from the states, using py:func:*simulation_smoother*:. We draw from the local trend state:

```
plt.figure(figsize=(15,5))
for i in range(10):
    plt.plot(model.index,model.simulation_smoother(
            model.latent_variables.get_z_values())[1][0:model.index.shape[0]])
plt.show()
```

If we want to plot rolling in-sample predictions, we can use the `plot_predict_is()`: method:

```
model.plot_predict_is(h=15,figsize=(15,5))
```

We can view out-of-sample predictions using `plot_predict()`:

```
model.plot_predict(h=20, past_values=17*4, figsize=(15,6))
```

If we want the predictions in a DataFrame form, then we can just use the `predict()`: method.

5.20.3 Class Description

class LLT (*data*, *integ*, *target*)

Local Linear Trend Models.

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc(*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_sample(*nsims*, *plot_data=True*)

Plots samples from the posterior predictive density of the model. This method only works if you fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many samples to draw
plot_data	boolean	Whether to plot the real data as well

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

ppc(*T*, *nsims*)

Returns a p-value for a posterior predictive check. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: int - the p-value for the discrepancy test

predict(*h*, *intervals=False*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
intervals	boolean	Whether to return prediction intervals

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction inter-

vals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

sample (*nsims*)
Returns np.ndarray of draws of the data from the posterior predictive density. This method only works if you have fitted the model using Bayesian inference.

Parameter	Type	Description
nsims	int	How many posterior draws to take

Returns : np.ndarray - samples from the posterior predictive density.

simulation_smoother (*beta*)
Returns np.ndarray of draws of the data from the Durbin and Koopman (2002) simulation smoother.

Parameter	Type	Description
beta	np.array	np.array of latent variables

Recommended just to use model.latent_variables.get_z_values() for the beta input, if you have already fit a model.

Returns : np.ndarray - samples from simulation smoother

5.20.4 References

Durbin, J. and Koopman, S. J. (2002). A simple and efficient simulation smoother for state space time series analysis. Biometrika, 89(3):603–615.

Harvey, A. C. (1989). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge.

5.21 Non-Gaussian Dynamic Regression Models

5.21.1 Introduction

With Non-Gaussian state space models, we have the same basic setup as Gaussian state space models, but now a potentially non-Gaussian measurement density. That is we are interested in problems of the form:

$$p\left(y_t \mid z_t\right)$$

$$\theta_t = f\left(\alpha_t\right)$$

$$\alpha_t = \alpha_{t-1} + \eta_t$$

$$\eta_t \sim N(0, \Sigma)$$

Usually MCMC based schemes are the right way to tackle this problem. Currently PyFlux uses BBVI for speed, but the mean-field approximation means there can be some bias in the states (although the results are generally okay for prediction). In the future, PyFlux will use a more structured approximation.

The **Non-Gaussian dynamic regression model** has the same form as a dynamic linear regression model, but with a non-Gaussian measurement density.

5.21.2 Example

See the notebook at https://github.com/RJT1990/talks/blob/master/PyDataTimeSeriesTalk.ipynb and the example for non-Gaussian estimation of a beta coefficient for finance. The API is from an old version here, but shows a use of this model type.

5.21.3 Class Description

class NDynReg (*formula*, *data*, *family*)

Non-Gaussian Dynamic Regression models

Parameter	Type	Description
formula	string	Patsy notation specifying the regression
data	pd.DataFrame	Contains the univariate time series
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

latent_variables

A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior (*index*, *prior*)

Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit (*method*, ***kwargs*)

Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit (***kwargs*)

Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_ppc (*T*, *nsims*)

Plots a histogram for a posterior predictive check with a discrepancy measure of the user's choosing. This method only works if you have fitted using Bayesian inference.

Parameter	Type	Description
T	function	Discrepancy, e.g. `np.mean` or `np.max`
nsims	int	How many simulations for the PPC

Returns: void - shows a matplotlib plot

plot_predict (*h*, *oos_data*, *past_values*, *intervals*, ***kwargs*)

Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the h first rows from *oos_data* and take the values for the exogenous variables that you asked for in the patsy formula.

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is (*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_z (*indices*, *figsize*)
Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

predict (*h*, *oos_data*)
Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead
oos_data	pd.DataFrame	Exogenous variables in a frame for h steps

To be clear, the *oos_data* argument should be a DataFrame in the same format as the initial dataframe used to initialize the model instance. The reason is that to predict future values, you need to specify assumptions about exogenous variables for the future. For example, if you predict *h* steps ahead, the method will take the 5 first rows from *oos_data* and take the values for the exogenous variables that you specified as exogenous variables in the patsy formula.

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)
Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

5.21.4 References

Harvey, A. C. (1989). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge.

Ranganath, R., Gerrish, S., and Blei, D. M. (2014). Black box variational inference. In Artificial Intelligence and Statistics.

5.22 Non-Gaussian Local Level Models

5.22.1 Introduction

With Non-Gaussian state space models, we have the same basic setup as Gaussian state space models, but now a potentially non-Gaussian measurement density. That is we are interested in problems of the form:

$$p\left(y_t \mid z_t\right)$$

$$\theta_t = f\left(\alpha_t\right)$$

$$\alpha_t = \alpha_{t-1} + \eta_t$$

$$\eta_t \sim N\left(0, \Sigma\right)$$

Usually MCMC based schemes are the right way to tackle this problem. Currently PyFlux uses BBVI for speed, but the mean-field approximation means there can be some bias in the states (although the results are generally okay for prediction). In the future, PyFlux will use a more structured approximation.

The **Non-Gaussian local level model** has the same form as a Gaussian local level model, but with a non-Gaussian measurement density.

5.22.2 Example

For fun, and since it's topical, we'll apply a Poisson local level model to count data on the number of goals the football team Leicester have scored since they rejoined the Premier League. Each index represents a match they have played. This is a short dataset, but it shows the principle behind the model.

```
import numpy as np
import pyflux as pf
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

leicester = pd.read_csv('http://www.pyflux.com/notebooks/leicester_goals_scored.csv')
leicester.columns= ["Time","Goals","Season2"]
plt.figure(figsize=(15,5))
plt.plot(leicester["Goals"])
plt.ylabel('Goals Scored')
plt.title('Leicester Goals Since Joining EPL');
plt.show()
```

We can fit a Poisson local level model as follows:

```
model = pf.NLLEV(data=leicester, target='Goals', family=pf.Poisson())
```

We can also use the higher-level wrapper which allows us to specify the family. If you pick a Normal distribution, then the Kalman filter will be used:

```
model = pf.LocalLevel(data=leicester, target='Goals', family=pf.Poisson())
```

Next we estimate the latent variables through a BBVI estimate z^{BBVI}:

```
x = model.fit(iterations=5000)
x.summary()

10% done : ELBO is -107.599165657
20% done : ELBO is -127.571498111
30% done : ELBO is -136.25857363
40% done : ELBO is -137.626516299
50% done : ELBO is -137.539662707
60% done : ELBO is -137.321490055
70% done : ELBO is -137.518451697
80% done : ELBO is -137.311382466
90% done : ELBO is -136.3580387
100% done : ELBO is -137.346927749

Final model ELBO is -135.76799195

Poisson Local Level Model
=======================================␣
↪==============================================
Dependent Variable: Goals                Method: BBVI
Start Date: 0                            Unnormalized Log Posterior: -56.8409
End Date: 74                             AIC: 115.681720125
Number of observations: 75               BIC: 117.999208239
=============================================================================================
Latent Variable           Median             Mean               95% Credibility␣
↪Interval
========================= ================== ==================␣
↪==========================
Sigma^2 level             0.0406             0.0406             (0.0353 | 0.0467)
=============================================================================================
```

We can plot the evolution parameter with *plot_z()*:

```
model.plot_z()
```

Next we will plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

The sharp changes at the beginning reflect the diffuse initialization; together with high initial uncertainty, this leads to stronger updates towards the beginning of the series. We can predict forward using plot_predict:

We can get an idea of the performance of our model by prediction through the `plot_predict()`: method:

```
model.plot_predict(h=5,figsize=(15,5))
```

If we just want the predictions themselves, we can use the `predict()`: method.

5.22.3 Class Description

class NLLEV(*data*, *ar*, *integ*, *target*, *family*)

Non-Gaussian Local Level Models (NLLEV).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

`latent_variables`
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

`adjust_prior`(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

`fit`(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

`plot_fit`(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

`plot_predict`(*h*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty.

Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

predict(*h*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is(*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

5.22.4 References

Harvey, A. C. (1989). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge.

Ranganath, R., Gerrish, S., and Blei, D. M. (2014). Black box variational inference. In Artificial Intelligence and Statistics.

5.23 Non-Gaussian Local Linear Trend Models

5.23.1 Introduction

With Non-Gaussian state space models, we have the same basic setup as Gaussian state space models, but now a potentially non-Gaussian measurement density. That is we are interested in problems of the form:

$$p\left(y_t \mid z_t\right)$$

$$\theta_t = f\left(\alpha_t\right)$$

$$\alpha_t = \alpha_{t-1} + \eta_t$$

$$\eta_t \sim N\left(0, \Sigma\right)$$

Usually MCMC based schemes are the right way to tackle this problem. Currently PyFlux uses BBVI for speed, but the mean-field approximation means there can be some bias in the states (although the results are generally okay for prediction). In the future, PyFlux will use a more structured approximation.

The **Non-Gaussian local linear trend model** has the same form as a Gaussian local linear trend model, but with a non-Gaussian measurement density.

5.23.2 Example

For fun, and since it's topical, we'll apply a Poisson local level model to count data on the number of goals the football team Leicester have scored since they rejoined the Premier League. Each index represents a match they have played. This is a short dataset, but it shows the principle behind the model.

```
import numpy as np
import pyflux as pf
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

leicester = pd.read_csv('http://www.pyflux.com/notebooks/leicester_goals_scored.csv')
leicester.columns= ["Time","Goals","Season2"]
plt.figure(figsize=(15,5))
plt.plot(leicester["Goals"])
plt.ylabel('Goals Scored')
plt.title('Leicester Goals Since Joining EPL');
plt.show()
```

We can fit a Poisson local linear trend model as follows:

```
model = pf.NLLT(data=leicester, target='Goals', family=pf.Poisson())
```

We can also use the higher-level wrapper which allows us to specify the family. If you pick a Normal distribution, then the Kalman filter will be used:

```
model = pf.LocalTrend(data=leicester, target='Goals', family=pf.Poisson())
```

Next we estimate the latent variables through a BBVI estimate z^{BBVI}:

```
x = model.fit(iterations=5000)
x.summary()

10% done : ELBO is -27837.965202
20% done : ELBO is -10667.1947315
30% done : ELBO is -5150.42573307
40% done : ELBO is -2567.54029949
50% done : ELBO is -1291.29282788
60% done : ELBO is -578.99494029
70% done : ELBO is -251.124996408
80% done : ELBO is -100.355592594
90% done : ELBO is -49.3752685727
100% done : ELBO is -13.9899801048

Final model ELBO is 46.2333499244
Poisson Local Linear Trend Model
=============================================================
↪=============================================
Dependent Variable: Goals                               Method: BBVI
Start Date: 0                                           Unnormalized Log Posterior:
↪235.942
End Date: 74                                            AIC: -467.884097447
Number of observations: 75                              BIC: -463.24912122
=============================================================================================================
Latent Variable                          Median             Mean               95%
↪Credibility Interval
======================================== ================== ==================
↪==========================
Sigma^2 level                            0.1738             0.1739             (0.
↪1539 | 0.197)
Sigma^2 trend                            0.0                0.0                (0.0 |
↪0.0)
=============================================================================================================
```

We can plot the evolution parameter with *plot_z()*:

```
model.plot_z([0])
model.plot_z([1])
```

Next we will plot the in-sample fit using `plot_fit()`:

```
model.plot_fit(figsize=(15,10))
```

5.23.3 Class Description

class NLLT (*data*, *ar*, *integ*, *target*, *family*)

Non-Gaussian Local Linear Trend Models (NLLT).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
family	pf.Family instance	The distribution for the time series, e.g `pf.Normal()`

Attributes

`latent_variables`
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

`adjust_prior`(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

`fit`(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

`plot_fit`(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

`plot_predict`(*h*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty.

Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is(*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_z(*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

predict(*h*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead

Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty. Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : pd.DataFrame - the model predictions

predict_is(*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

5.23.4 References

Harvey, A. C. (1989). Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge.

Ranganath, R., Gerrish, S., and Blei, D. M. (2014). Black box variational inference. In Artificial Intelligence and Statistics.

5.24 VAR models

5.24.1 Introduction

Vector autoregressions (VARs) were introduced in the econometrics literature in the 1980s to allow for (linear) dependencies among multiple variables. For a K x 1 vector y_t we can specify a VAR(p) model as:

$$y_t = c + A_1 y_{t-1} + ... + A_p y_{t-p} + e_t$$

These models can be estimated quickly through OLS. But with a large number of dependent variables, the number of parameters to be estimated can grow very quickly. See the notebook on **Bayesian VARs** for an alternative way to approach these types of model.

5.24.2 Example

We'll run an VAR model for US banking sector stocks.

```
import numpy as np
import pyflux as pf
from pandas_datareader import DataReader
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline

ibm = DataReader(['JPM','GS','BAC','C','WFC','MS'],  'yahoo', datetime(2012,1,1),␣
↪datetime(2016,6,28))
opening_prices = np.log(ibm['Open'])
plt.figure(figsize=(15,5));
plt.plot(opening_prices.index,opening_prices);
plt.legend(opening_prices.columns.values,loc=3);
plt.title("Logged opening price");
```

Here we specify an arbitrary VAR(2) model, which we fit via OLS:

```
model = pf.VAR(data=opening_prices, lags=2, integ=1)
```

Next we estimate the latent variables. For this example we will use an OLS estimate z^{OLS}:

```
x = model.fit()
x.summary()

VAR(2)
=========================================␣
↪================================================
Dependent Variable: Differenced BAC      Method: OLS
```

(continues on next page)

(continued from previous page)

```
Start Date: 2012-01-05 00:00:00          Log Likelihood: 21547.2578
End Date: 2016-06-28 00:00:00            AIC: -42896.5156
Number of observations: 1126             BIC: -42398.8994
==========================================================================================
Latent Variable           Estimate   Std Error  z         P>|z|    95% C.I.
========================= ========== ========== ========= ========␣
↪=========================
Diff BAC Constant         0.0007     0.0006     1.2007    0.2299   (-0.0004 | 0.0018)
Diff BAC AR(1)            -0.0525    0.0005     -97.5672  0.0      (-0.0535 | -0.0514)
Diff C to Diff BAC AR(1)  -0.0616    0.0004     -143.365  0.0      (-0.0625 | -0.0608)
Diff GS to Diff BAC AR(1) 0.0595     0.0004     132.6638  0.0      (0.0587 | 0.0604)
Diff JPM to Diff BAC AR(1)0.0296     0.0006     49.3563   0.0      (0.0284 | 0.0308)
Diff MS to Diff BAC AR(1) -0.0231    0.0004     -62.6218  0.0      (-0.0239 | -0.0224)
Diff WFC to Diff BAC AR(1)-0.0417    0.0598     -0.6968   0.4859   (-0.159 | 0.0756)
Diff BAC AR(2)            0.1171     0.0555     2.1087    0.035    (0.0083 | 0.226)
Diff C to Diff BAC AR(2)  -0.1266    0.0444     -2.8528   0.0043   (-0.2136 | -0.0396)
Diff GS to Diff BAC AR(2) 0.1698     0.0464     3.6618    0.0003   (0.0789 | 0.2606)
Diff JPM to Diff BAC AR(2)-0.0959    0.062      -1.5472   0.1218   (-0.2174 | 0.0256)
Diff MS to Diff BAC AR(2) -0.0213    0.0381     -0.557    0.5775   (-0.096 | 0.0535)
Diff WFC to Diff BAC AR(2)-0.001     0.0701     -0.0149   0.9881   (-0.1384 | 0.1363)
Diff C Constant           0.0003     0.065      0.0047    0.9962   (-0.1272 | 0.1278)
Diff C AR(1)              0.0193     0.052      0.3706    0.7109   (-0.0826 | 0.1211)
Diff BAC to Diff C AR(1)  -0.0576    0.0543     -1.0613   0.2886   (-0.164 | 0.0488)
Diff GS to Diff C AR(1)   0.0579     0.0726     0.7979    0.4249   (-0.0844 | 0.2002)
Diff JPM to Diff C AR(1)  0.0831     0.0447     1.8595    0.063    (-0.0045 | 0.1706)
Diff MS to Diff C AR(1)   -0.037     0.0794     -0.4657   0.6414   (-0.1925 | 0.1186)
Diff WFC to Diff C AR(1)  -0.1785    0.0737     -2.4235   0.0154   (-0.3229 | -0.0341)
Diff C AR(2)              0.1612     0.0589     2.7379    0.0062   (0.0458 | 0.2765)
Diff BAC to Diff C AR(2)  -0.1021    0.0615     -1.6598   0.0969   (-0.2226 | 0.0185)
Diff GS to Diff C AR(2)   0.1109     0.0822     1.3483    0.1776   (-0.0503 | 0.272)
Diff JPM to Diff C AR(2)  -0.0453    0.0506     -0.8946   0.371    (-0.1444 | 0.0539)
Diff MS to Diff C AR(2)   0.0127     0.0775     0.1643    0.8695   (-0.1391 | 0.1646)
Diff WFC to Diff C AR(2)  -0.1313    0.0719     -1.8261   0.0678   (-0.2723 | 0.0096)
Diff GS Constant          0.0003     0.0575     0.006     0.9952   (-0.1123 | 0.113)
Diff GS AR(1)             -0.016     0.06       -0.266    0.7903   (-0.1336 | 0.1017)
Diff BAC to Diff GS AR(1) 0.0051     0.0803     0.0633    0.9495   (-0.1523 | 0.1624)
Diff C to Diff GS AR(1)   -0.0785    0.0494     -1.5891   0.112    (-0.1753 | 0.0183)
Diff JPM to Diff GS AR(1) 0.0507     0.0575     0.8814    0.3781   (-0.062 | 0.1633)
Diff MS to Diff GS AR(1)  0.0425     0.0534     0.7961    0.4259   (-0.0621 | 0.1471)
Diff WFC to Diff GS AR(1) -0.0613    0.0426     -1.4376   0.1505   (-0.1449 | 0.0223)
Diff GS AR(2)             0.0865     0.0445     1.9422    0.0521   (-0.0008 | 0.1738)
Diff BAC to Diff GS AR(2) -0.1896    0.0596     -3.1832   0.0015   (-0.3064 | -0.0729)
Diff C to Diff GS AR(2)   0.0423     0.0367     1.1553    0.248    (-0.0295 | 0.1142)
Diff JPM to Diff GS AR(2) 0.0667     0.0769     0.8664    0.3863   (-0.0841 | 0.2174)
Diff MS to Diff GS AR(2)  0.0433     0.0714     0.6067    0.5441   (-0.0966 | 0.1833)
Diff WFC to Diff GS AR(2) -0.0362    0.0571     -0.6347   0.5256   (-0.1481 | 0.0756)
Diff JPM Constant         0.0005     0.0596     0.0082    0.9934   (-0.1163 | 0.1173)
Diff JPM AR(1)            -0.0304    0.0797     -0.3813   0.703    (-0.1866 | 0.1258)
Diff BAC to Diff JPM AR(1)-0.0281    0.049      -0.5738   0.5661   (-0.1243 | 0.068)
Diff C to Diff JPM AR(1)  0.0695     0.0594     1.1698    0.2421   (-0.047 | 0.186)
Diff GS to Diff JPM AR(1) -0.0106    0.0552     -0.1924   0.8474   (-0.1187 | 0.0975)
Diff MS to Diff JPM AR(1) -0.0338    0.0441     -0.7675   0.4428   (-0.1202 | 0.0526)
Diff WFC to Diff JPM AR(1)-0.0725    0.046      -1.5744   0.1154   (-0.1627 | 0.0178)
Diff JPM AR(2)            0.096      0.0616     1.559     0.119    (-0.0247 | 0.2167)
Diff BAC to Diff JPM AR(2)-0.1246    0.0379     -3.2883   0.001    (-0.1989 | -0.0503)
Diff C to Diff JPM AR(2)  0.0229     0.0696     0.3284    0.7426   (-0.1136 | 0.1593)
Diff GS to Diff JPM AR(2) -0.0084    0.0646     -0.1301   0.8965   (-0.1351 | 0.1182)
```

(continues on next page)

(continued from previous page)

```
Diff MS to Diff JPM AR(2) 0.0319     0.0516     0.6182   0.5364   (-0.0693 | 0.1332)
Diff WFC to Diff JPM AR(2)-0.0117    0.0539     -0.2161  0.8289   (-0.1174 | 0.0941)
Diff MS Constant          0.0004     0.0721     0.005    0.996    (-0.141 | 0.1417)
Diff MS AR(1)             0.0249     0.0444     0.5605   0.5752   (-0.0621 | 0.1119)
Diff BAC to Diff MS AR(1) 0.0456     0.0783     0.5833   0.5597   (-0.1077 | 0.199)
Diff C to Diff MS AR(1)   0.0083     0.0726     0.1148   0.9086   (-0.134 | 0.1507)
Diff GS to Diff MS AR(1)  0.1319     0.0581     2.2717   0.0231   (0.0181 | 0.2457)
Diff JPM to Diff MS AR(1) -0.1771    0.0606     -2.9213  0.0035   (-0.296 | -0.0583)
Diff WFC to Diff MS AR(1) -0.151     0.0811     -1.8629  0.0625   (-0.31 | 0.0079)
Diff MS AR(2)             0.1512     0.0499     3.0308   0.0024   (0.0534 | 0.249)
Diff BAC to Diff MS AR(2) -0.2173    0.0772     -2.8157  0.0049   (-0.3686 | -0.066)
Diff C to Diff MS AR(2)   0.1827     0.0716     2.5499   0.0108   (0.0423 | 0.3231)
Diff GS to Diff MS AR(2)  -0.0107    0.0573     -0.1873  0.8514   (-0.1229 | 0.1015)
Diff JPM to Diff MS AR(2) 0.0004     0.0598     0.0066   0.9947   (-0.1168 | 0.1176)
Diff WFC to Diff MS AR(2) -0.0697    0.08       -0.8711  0.3837   (-0.2264 | 0.0871)
Diff WFC Constant         0.0005     0.0492     0.0095   0.9924   (-0.096 | 0.0969)
Diff WFC AR(1)            0.0092     0.0574     0.1611   0.872    (-0.1032 | 0.1217)
Diff BAC to Diff WFC AR(1)-0.0059    0.0532     -0.1113  0.9114   (-0.1103 | 0.0984)
Diff C to Diff WFC AR(1)  0.0062     0.0425     0.1448   0.8848   (-0.0772 | 0.0896)
Diff GS to Diff WFC AR(1) 0.0525     0.0444     1.1811   0.2376   (-0.0346 | 0.1396)
Diff JPM to Diff WFC AR(1)-0.0047    0.0594     -0.0792  0.9368   (-0.1212 | 0.1118)
Diff MS to Diff WFC AR(1) -0.1996    0.0366     -5.4578  0.0      (-0.2713 | -0.1279)
Diff WFC AR(2)            0.0291     0.0773     0.3759   0.707    (-0.1225 | 0.1806)
Diff BAC to Diff WFC AR(2)-0.0509    0.0718     -0.7087  0.4785   (-0.1915 | 0.0898)
Diff C to Diff WFC AR(2)  0.0255     0.0574     0.4444   0.6567   (-0.0869 | 0.1379)
Diff GS to Diff WFC AR(2) 0.0235     0.0599     0.3922   0.6949   (-0.0939 | 0.1409)
Diff JPM to Diff WFC AR(2)0.015      0.0801     0.1878   0.851    (-0.142 | 0.1721)
Diff MS to Diff WFC AR(2) -0.0556    0.0493     -1.1276  0.2595   (-0.1522 | 0.041)
==============================================================================================
```

We can plot latent variables with *plot_z()*: method:

```
model.plot_z(list(range(0,6)),figsize=(15,5))
```

We can plot the in-sample fit with `plot_fit()`:

```
model.plot_fit(figsize=(15,5))
```

We can make forward predictions with our model using `plot_predict()`:

```
model.plot_predict(past_values=19, h=5, figsize=(15,5))
```

How does our model perform? We can get a sense by performing a rolling in-sample prediction – `plot_predict_is()`: for plotted graphs:

```
model.plot_predict_is(h=30, figsize=((15,5)))
```

5.24.3 Class Description

class VAR(*data*, *lags*, *integ*, *target*, *use_ols_covariance*)

Vector Autoregression Models (VAR).

Parameter	Type	Description
data	pd.DataFrame or np.ndarray	Contains the univariate time series
lags	int	The number of autoregressive lags
integ	int	How many times to difference the data (default: 0)
target	string or int	Which column of DataFrame/array to use.
use_ols_covariance	boolean	Whether to use fixed OLS covariance

Attributes

latent_variables
A pf.LatentVariables() object containing information on the model latent variables, prior settings. any fitted values, starting values, and other latent variable information. When a model is fitted, this is where the latent variables are updated/stored. Please see the documentation on Latent Variables for information on attributes within this object, as well as methods for accessing the latent variable information.

Methods

adjust_prior(*index*, *prior*)
Adjusts the priors for the model latent variables. The latent variables and their indices can be viewed by printing the `latent_variables` attribute attached to the model instance.

Parameter	Type	Description
index	int	Index of the latent variable to change
prior	pf.Family instance	Prior distribution, e.g. `pf.Normal()`

Returns: void - changes the model `latent_variables` attribute

fit(*method*, ***kwargs*)
Estimates latent variables for the model. User chooses an inference option and the method returns a results object, as well as updating the model's `latent_variables` attribute.

Parameter	Type	Description
method	str	Inference option: e.g. 'M-H' or 'MLE'

See Bayesian Inference and Classical Inference sections of the documentation for the full list of inference options. Optional parameters can be entered that are relevant to the particular mode of inference chosen.

Returns: pf.Results instance with information for the estimated latent variables

plot_fit(***kwargs*)
Plots the fit of the model against the data. Optional arguments include *figsize*, the dimensions of the figure to plot.

Returns : void - shows a matplotlib plot

plot_predict(*h*, *past_values*, *intervals*, ***kwargs*)
Plots predictions of the model, along with intervals.

Parameter	Type	Description
h	int	How many steps to forecast ahead
past_values	int	How many past datapoints to plot
intervals	boolean	Whether to plot intervals or not

Optional arguments include *figsize* - the dimensions of the figure to plot. Please note that if you use Maximum Likelihood or Variational Inference, the intervals shown will not reflect latent variable uncertainty.

Only Metropolis-Hastings will give you fully Bayesian prediction intervals. Bayesian intervals with variational inference are not shown because of the limitation of mean-field inference in not accounting for posterior correlations.

Returns : void - shows a matplotlib plot

plot_predict_is (*h*, *fit_once*, *fit_method*, ***kwargs*)

Plots in-sample rolling predictions for the model. This means that the user pretends a last subsection of data is out-of-sample, and forecasts after each period and assesses how well they did. The user can choose whether to fit parameters once at the beginning or every time step.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Optional arguments include *figsize* - the dimensions of the figure to plot. **h** is an int of how many previous steps to simulate performance on.

Returns : void - shows a matplotlib plot

plot_z (*indices*, *figsize*)

Returns a plot of the latent variables and their associated uncertainty.

Parameter	Type	Description
indices	int or list	Which latent variable indices to plot
figsize	tuple	Size of the matplotlib figure

Returns : void - shows a matplotlib plot

predict (*h*)

Returns a DataFrame of model predictions.

Parameter	Type	Description
h	int	How many steps to forecast ahead

Returns : pd.DataFrame - the model predictions

predict_is (*h*, *fit_once*, *fit_method*)

Returns DataFrame of in-sample rolling predictions for the model.

Parameter	Type	Description
h	int	How many previous timesteps to use
fit_once	boolean	Whether to fit once, or every timestep
fit_method	str	Which inference option, e.g. 'MLE'

Returns : pd.DataFrame - the model predictions

simulation_smoother (*beta*)

Returns np.ndarray of draws of the data from the Durbin and Koopman (2002) simulation smoother.

Parameter	Type	Description
beta	np.array	np.array of latent variables

Recommended just to use model.latent_variables.get_z_values() for the beta input, if you have already fit a model.

Returns : np.ndarray - samples from simulation smoother

5.24.4 References

Lütkepohl, H. & Kraetzig, M. (2004). Applied Time Series Econometrics. Cambridge University Press, Cambridge.

CHAPTER 6

Inference Guide

6.1 Bayesian Inference

PyFlux supports Bayesian inference for all the model types on offer.

6.1.1 Interface

To view the current priors, you should print the model's latent variable object. For example:

```
import pyflux as pf

# model = ... (specify a model)
print(model.z)
```

This will outline the current prior assumptions for each latent variable, as well as the variational approximate distribution that is assumed (if you are performing variational inference). To adjust priors, simply use the following method on your model object:

adjust_prior(*index*, *prior*)
Adjusts the priors of the model. **index** can be an int or a list. **prior** is a prior object, such as *Normal*.

Here is example usage for `adjust_prior()`:

```
import pyflux as pf

# model = ... (specify a model)
model.list_priors()
model.adjust_prior(2, pf.Normal(0,1))
```

6.1.2 Methods

There are a number of Bayesian inference options using the `fit()` method. These can be chosen with the method argument.

Black-Box Variational Inference

Performs Black Box Variational Inference. Currently the fixed assumption is mean-field variational inference with normal approximate distributions. The gradient used in this implementation is the score function gradient. By default we use 24 samples for the gradient which is quite intense (other implementations use 2-8 samples). For your application, less samples may be as effective and quicker. One of the limitations of the implementation right now is BBVI here does not support using mini-batches of data. It is not clear yet how mini-batches would work with model types that have an underlying sequence of latent states - if it is shown to be effective, then this option will be included in future.

```
model.fit(method='BBVI', iterations='10000', optimizer='ADAM')
```

- *batch_size* : (default : 24) number of Monte Carlo samples for the gradient
- *iterations* : (default : 3000) number of iterations to run
- *optimizer* : (default: RMSProp) RMSProp or ADAM (stochastic optimizers)
- *map_start*: (default: True) if True, starts latent variables using a MAP/PML estimate
- *mini_batch*: (default: None) if an int, then will sample mini-batches from the data of the size selected (e.g. 32). This option does not work for some model types.
- *learning_rate*: (default: 0.001) the learning rate for the optimizer
- *record_elbo* : (default: False) if True, will record ELBO during optimization, which can is stored as `x.elbo_records` for a `BBVIResults()` object `x`.

Laplace Approximation

Performs a Laplace approximation on the posterior.

```
model.fit(method='Laplace')
```

Metropolis-Hastings

Performs Metropolis-Hastings MCMC. Currently uses 'one long chain' which is not ideal, but works okay for most of the models available. This method applies a warm-up period of half the number of simulations and applies thinning by removing every other sample to reduce correlation.

```
model.fit(method='M-H')
```

- *map_start* : (default: True) whether to initialize starting values and the covariance matrix using MAP estimates and the Inverse Hessian
- *nsims* : number of simulations for the chain

Penalized Maximum Likelihood

Provides a Maximum a posteriori (MAP) estimate. This estimate is not completely Bayesian as it is based on a 0/1 loss rather than a squared or absolute loss. It can be considered a form of modal approximation, when taken together with the Inverse Hessian matrix.

```
model.fit(method='PML')
```

- *preopt_search* : (default : True) if True will use a preoptimization stage to find good starting values (if the model type has no available preoptimization method, this argument will be ignored). Turning this off will speed up optimization at the risk of obtaining an inferior solution.

6.2 Classical Inference

PyFlux supports classical methods of inference. These can be considered as point mass approximations to the full posterior.

6.2.1 Methods

There are a number of classical inference options using the `fit()` method. These can be chosen with the method option.

Maximum Likelihood

Performs Maximum Likelihood estimation.

```
model.fit(method='MLE')
```

- *preopt_search* : (default : True) if True will use a preoptimization stage to find good starting values (if the model type has no available preoptimization method, this argument will be ignored). Turning this off will speed up optimization at the risk of obtaining an inferior solution.

Ordinary Least Squares

Performs Ordinary Least Squares estimation.

```
model.fit(method='OLS')
```

Penalized Maximum Likelihood

From a frequentist perspective, PML can be viewed as a type of regularization on the coefficients.

```
model.fit(method='PML')
```

- *preopt_search* : (default : True) if True will use a preoptimization stage to find good starting values (if the model type has no available preoptimization method, this argument will be ignored). Turning this off will speed up optimization at the risk of obtaining an inferior solution.

6.3 Families

6.3.1 Introduction

PyFlux uses a unified family API that can be used for specifying model measurement densities as well as the priors on latent variables in the model. This guide shows the various distributions available and their uses.

6.3.2 Family Guidebook

class Cauchy(*loc*, *scale*, *transform*)

Cauchy Family

Parameter	Type	Description
loc	float	Location parameter for Cauchy family
scale	float	Scale for Cauchy family
transform	string	Whether to transform lmd (e.g. 'exp')

This class can be used for priors and model measurement densities. For example:

```
model = pf.ARIMA(ar=1,ma=0,data=my_data, family=pf.Cauchy())
```

```
model.adjust_prior(0, pf.Cauchy(0,1))
```

class Exponential(*lmd*, *transform*)
Exponential Family

Parameter	Type	Description
lmd	float	Rate parameter for the Exponential family
transform	string	Whether to transform lmd (e.g. 'exp')

This class can be used for model measurement densities. For example:

```
model = pf.ARIMA(ar=1,ma=0,data=my_data, family=pf.Exponential())
```

class Flat(*lmd*, *transform*)
Flat Family

Parameter	Type	Description
transform	string	Whether to transform the parameter

This class can be used as an non-informative prior distribution.

```
model.adjust_prior(0, pf.Flat())
```

class InverseGamma(*alpha*, *beta*, *transform*)
Inverse Gamma Family

Parameter	Type	Description
alpha	float	Alpha parameter for the IGamma family
beta	float	Beta parameter for the IGamma family
transform	string	Whether to transform the parameter

This class can be used as a prior distribution.

```
model.adjust_prior(0, pf.InverseGamma(1,1))
```

class InverseWishart(*v*, *Psi*, *transform*)
Inverse Wishart Family

Parameter	Type	Description
v	float	v parameter for the family
Psi	float	Psi covariance matrix for the family
transform	string	Whether to transform the parameter

This class can be used as a prior distribution.

```
my_covariance_prior = np.eye(3)
model.adjust_prior(0, pf.InverseWishart(3, my_covariance_prior))
```

class Laplace(*loc*, *scale*, *transform*)

Laplace Family

Parameter	Type	Description
loc	float	Location parameter for Laplace family
scale	float	Scale for Laplace family
transform	string	Whether to transform loc (e.g. 'exp')

This class can be used for priors and model measurement densities. For example:

```
model = pf.ARIMA(ar=1,ma=0,data=my_data, family=pf.Laplace())
```

```
model.adjust_prior(0, pf.Laplace(0,1))
```

class Normal(*mu*, *sigma*, *transform*)

Normal Family

Parameter	Type	Description
mu	float	Location parameter for Normal family
sigma	float	Standard deviation for Normal family
transform	string	Whether to transform mu (e.g. 'exp')

This class can be used for priors and model measurement densities. For example:

```
model = pf.ARIMA(ar=1,ma=0,data=my_data, family=pf.Normal())
```

```
model.adjust_prior(0, pf.Normal(0,1))
```

class Poisson(*lmd*, *transform*)

Poisson Family

Parameter	Type	Description
lmd	float	Rate parameter for the Poisson family
transform	string	Whether to transform mu (e.g. 'exp')

This class can be used for model measurement densities. For example:

```
model = pf.ARIMA(ar=1,ma=0,data=my_data, family=pf.Poisson())
```

class t(*loc*, *scale*, *df*, *transform*)

Student-t Family

Parameter	Type	Description
loc	float	Location parameter for t family
scale	float	Standard deviation for t family
df	float	Degrees of freedom for t family
transform	string	Whether to transform mu (e.g. 'exp')

This class can be used for priors and model measurement densities. For example:

```
model = pf.ARIMA(ar=1,ma=0,data=my_data, family=pf.t())
```

```
model.adjust_prior(0, pf.t(0, 1, 3))
```

class Skewt (*loc*, *scale*, *df*, *gamma*, *transform*)

Skewed Student-t Family

Parameter	Type	Description
loc	float	Location parameter for t family
scale	float	Standard deviation for t family
df	float	Degrees of freedom for t family
gamma	float	Skewness parameter for t family
transform	string	Whether to transform mu (e.g. 'exp')

This class can be used for priors and model measurement densities. For example:

```
model = pf.ARIMA(ar=1,ma=0,data=my_data, family=pf.Skewt())
```

```
model.adjust_prior(0, pf.Skewt(0, 1, 3, 0.9))
```

class TruncatedNormal (*mu*, *sigma*, *lower*, *upper*, *transform*)

Truncated Normal Family

Parameter	Type	Description
mu	float	Location parameter for TNormal family
sigma	float	Standard deviation for TNormal family
lower	float	Lower limit for the truncation
upper	float	Upper limit for the truncation
transform	string	Whether to transform mu (e.g. 'exp')

This class can be used as a prior. For example:

```
model.adjust_prior(0, pf.TruncatedNormal(0, 1, lower=0.0, upper=1.0))
```

CHAPTER 7

Acknowledgements by the Author

I am grateful to my employer ALPIMA for being supportive of my efforts to develop this library in my spare time. I am also grateful to those in the PyData community who have given helpful comments and feedback for development.

Ross Taylor, @rosstaylor90

Index

S

T

V