QBUS6840: Tutorial 12 – State-Space Models

Objectives

- Define and apply State-Space Models
- Compare State-Space Models with other models
- Develop Python skills

This tutorial is adopted from the example in the following materials
http://www.chadfulton.com/files/fulton_statsmodels_2017_v1.pdf
https://jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/
http://www.statsmodels.org/stable/examples/notebooks/generated/statespace_local_linear_trend.html

In this tutorial, we will use `statsmodels` package to do state-space modeling.

Some models are implemented in `statsmodel` such as ARIMA models we have used. But the others you need to define the models by yourself. Do not panic, the `statsmodels` package offers functionalities for us to do prediction, filtering and smoothing for state-space models.

To define and use a model not in `statsmodel` or other packages, we can follow a standard procedure to define any customized model.

1. Class in python

Before starting working on our own model, let us have a look on a concept in python - class.

In python, class provide an approach for you to bundle data and functionality together. Defining a new class creates a new *type* of object, allowing new *instances* of the specific type to be declared. Each class instance can have multiple attributes maintaining its state. Class instances can also have functions (defined by its class) for manipulating its data.

Below is an example of a simple class:

```
class Customer(object):

    """A customer of Bank with an account. Customers have
the following properties(Attributes):

        name: A string representing the customer's name.
        balance: A float tracking the current balance of
the customer's account.
    """

    def __init__(self, name, balance=0.0):
```

```
        #Return a Customer object whose name is *name*
and starting balance is *balance*.
        self.name = name
        self.balance = balance

    def withdraw(self, amount):
        #Return the balance remaining after withdrawing
*amount* dollars.
        if amount > self.balance:
            raise RuntimeError('Amount greater than
available balance.')
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        #Return the balance remaining after depositing
*amount* dollars.
        self.balance += amount
        return self.balance
```

You can run the below code to see how to use this class.

```
john = Customer("john",1000)
print (john.deposit(5000))
print (john.withdraw(2000))
```

2. Define your own class

Now with basic understanding on class, let us define a LocalLevel class for your Exponential Model.

```
class LocalLevel(sm.tsa.statespace.MLEModel):
    # statsmodel defines the basic MLEModel with all the
functionalities we need
    def __init__(self, endog):
        # The dimension of the states, this is 1 because
state is
        # mu_t, see the slide
        k_states = 1

        # The dimension of the noise terms, i.e., xi_t
        k_posdef = 1

        # Initialize the statespace
        # Note endog is the argument for the time series
data
        # we will use approximate-diffusing method to
initialize u_{1|0} and P_{1|0}, see the slide
        super(LocalLevel, self).__init__(
            endog, k_states=k_states, k_posdef=k_posdef,
```

```
            initialization='approximate_diffuse',
            loglikelihood_burn=10
        )

        # Initialize the matrices
        self.ssm['design'] = np.array([1])     #for Z_t
        self.ssm['transition'] = np.array([1]) #for T_t
        self.ssm['selection'] = np.array([1])  #for R_t

    # We give names to parameters sigma^2_e, sigma^2_eta
and sigma^2_xi
    @property
    def param_names(self):
        return ['sigma2.measurement', 'sigma2.level']

    # initial value for the parameters
    @property
    def start_params(self):
        return [np.std(self.endog)]*2

    # Because our parameters are variances which are
greater than zero (a condition)
    # these transforms are used to transform the
parameters to an unconditional
    # space for unconstrained optimisation
    def transform_params(self, unconstrained):
        return unconstrained**2

    def untransform_params(self, constrained):
        return constrained**0.5

    # We must define this functions
    def update(self, params, *args, **kwargs):
        params = super(LocalLevel, self).update(params,
*args, **kwargs)

        # Observation covariance  i.e., H_t
        self.ssm['obs_cov',0,0] = params[0]

        # State covariance   i.e., Q_t
        self.ssm['state_cov', 0, 0] = params[1]
```

3. Setup your defined model

Load and plot the Australian Visitors data

```
visitors = pd.read_csv('AustralianVisitors.csv')

y = visitors['No of Visitors']
y.index = pd.date_range('01/01/1991', '31/12/2016',
```

```
freq='MS')
```

Setup the model

```
model_1 = LocalLevel(y)
```

4. Model fitting and forecast

Fit your model and print summary

```
res = model_1.fit(disp=False)
print(res.summary())
```

Finally, we can do post-estimation prediction and forecasting.

```
predict = res.get_prediction()
forecast = res.get_forecast('2018')
```

Plot the results

```
fig, ax = plt.subplots(figsize=(10,4))
y.plot(ax=ax, style='k.', label='Observations')
predict.predicted_mean.plot(ax=ax, label='One-step-ahead
Prediction')

forecast.predicted_mean.plot(ax=ax, style='r',
label='Forecast')

legend = ax.legend(loc='lower left')
```

Now you have defined your own Exponential Model and used it for forecasting. Following let us try another model.

5. Univariate Holt's Linear Trend Model

Similarly, you can define Univariate Holt's Linear Trend Model as following.

As an example, below presents the code for a full implementation of the Local Linear Trend model.

This model has the form:

$$\begin{aligned} y_t &= \mu_t + \varepsilon_t & \varepsilon_t &\sim N(0, {\sigma_\varepsilon}^2) \\ \mu_{t+1} &= \mu_t + \nu_t + \xi_t & \xi_t &\sim N(0, {\sigma_\xi}^2) \\ \nu_{t+1} &= \nu_t + \zeta_t & \zeta_t &\sim N(0, {\sigma_\zeta}^2) \end{aligned}$$

Therefore, this can be cast into state space form as:

$$y_t = [1 \;\; 0] \begin{bmatrix} \mu_t \\ \nu_t \end{bmatrix} + \varepsilon_t$$

$$\begin{bmatrix} \mu_{t+1} \\ \nu_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mu_t \\ \nu_t \end{bmatrix} + \begin{bmatrix} \xi_t \\ \zeta_t \end{bmatrix}$$

Notice that much of the state space representation is composed of known values; in fact, the only parts in which parameters to be estimated appear are in the variance / covariance matrices:

$$H_t = [\sigma_\varepsilon^2]$$

$$Q_t = \begin{bmatrix} \sigma_\xi^2 & 0 \\ 0 & \sigma_\zeta^2 \end{bmatrix}$$

```
class LocalLinearTrend(sm.tsa.statespace.MLEModel):
    # statsmodel defines the basic MLEModel with all the
functionalities we need
    def __init__(self, endog):
        # The dimension of the states, this is 2 because
states are
        # alpha_t and nu_t, see the slide
        k_states = 2

        # int – The dimension of a guaranteed positive
definite covariance matrix describing the shocks in the
measurement equation.
        k_posdef = 2

        # Initialize the state space
        # Note endog is the argument for the time series
data
        # we will use approximate-diffusing method to
initialize u_{1|0} and P_{1|0}, see the slide
        super(LocalLinearTrend, self).__init__(
            endog, k_states=k_states, k_posdef=k_posdef,
            initialization='approximate_diffuse',
            loglikelihood_burn=k_states
        )

        # Initialize the matrices
        self.ssm['design'] = np.array([1, 0])    #for Z_t
        self.ssm['transition'] = np.array([[1, 1],
                                           [0, 1]])  #for T_t
        self.ssm['selection'] = np.eye(k_states) #for R_t

        # Cache some indices, for state cov Q_t the
parameters are at diagonal
        self._state_cov_idx = ('state_cov',) +
np.diag_indices(k_posdef)

    # We give names to parameters sigma^2_e, sigma^2_eta
and sigma^2_zeta
    @property
    def param_names(self):
        return ['sigma2.measurement', 'sigma2.level',
'sigma2.trend']
```

```
    # initial value for the parameters
    @property
    def start_params(self):
        return [np.std(self.endog)]*3

    # Because our parameters are variances which are
greater than zero (a condition)
    # these transforms are used to transform the
parameters to an unconditional
    # space for unconstrained optimisation
    def transform_params(self, unconstrained):
        return unconstrained**2

    def untransform_params(self, constrained):
        return constrained**0.5

    # We must define this functions
    def update(self, params, *args, **kwargs):
        params = super(LocalLinearTrend,
self).update(params, *args, **kwargs)

        # Observation covariance  i.e., H_t
        self.ssm['obs_cov',0,0] = params[0]

        # State covariance   i.e., Q_t
        self.ssm[self._state_cov_idx] = params[1:]
```

Load the data and Setup the model

```
# We load the time series data
df2 = pd.read_csv('OxCodeAll.csv')
# Preparing index for visualisation
df2.index = pd.date_range(start='%d-01-01' % df2.date[0],
end='%d-01-01' % df2.iloc[-1, 0], freq='AS')

# Log transform
df2['lff'] = np.log(df2['ff'])

# Setup the model
model2 = LocalLinearTrend(df2['lff'])
```

Fit your new model and print summary

```
res2 = model2.fit(disp=False)
print(res2.summary())
```

Then do post-estimation prediction and forecasting, and plot the results.

```
predict = res2.get_prediction()
```

```
forecast = res2.get_forecast('2014')

fig, ax = plt.subplots(figsize=(10,4))

# Plot the results
df2['lff'].plot(ax=ax, style='k.', label='Observations')
predict.predicted_mean.plot(ax=ax, label='One-step-ahead
Prediction')
predict_ci = predict.conf_int(alpha=0.05)
predict_index = np.arange(len(predict_ci))
ax.fill_between(predict_index[2:], predict_ci.iloc[2:, 0],
predict_ci.iloc[2:, 1], alpha=0.1)

forecast.predicted_mean.plot(ax=ax, style='r',
label='Forecast')
forecast_ci = forecast.conf_int()
forecast_index = np.arange(len(predict_ci),
len(predict_ci) + len(forecast_ci))
ax.fill_between(forecast_index, forecast_ci.iloc[:, 0],
forecast_ci.iloc[:, 1], alpha=0.1)

# Cleanup the image
ax.set_ylim((4, 8));
legend = ax.legend(loc='lower left');
```

Now you have defined and used the second model. However, this is not the end, you may try define other state space models and use them for forecasting, and compare with the models above.

6. ARMA(1,1) Process

An ARMA(1,1) model can be written in state-space form as

$$y_t = [1 \;\; \theta_1]\begin{bmatrix}\mu_{1t}\\ \mu_{2t}\end{bmatrix}$$
$$\begin{bmatrix}\mu_{1t+1}\\ \mu_{2t}+1\end{bmatrix} = \begin{bmatrix}\phi_1 & 0\\ 1 & 0\end{bmatrix}\begin{bmatrix}\mu_{1t}\\ \mu_{2t}\end{bmatrix} + \begin{bmatrix}1\\ 0\end{bmatrix}\xi_t$$

Where

$$\xi_t \sim N(0, \sigma^2)$$

Therefore, we only have 3 unknown parameters in this model: $\phi_1, \theta_1, \sigma^2$

Step1: Define parameters for an AR(1)

```
from scipy.signal import lfilter
nobs = int(1e3)
true_phi = 0.5
true_sigma = 1**0.5
```

Step2: Simulate a time series

```
np.random.seed(1234)
disturbances = np.random.normal(0, true_sigma,
size=(nobs,))
endog = lfilter([1], np.r_[1, -true_phi], disturbances)
```

Step3: Construct the model for an ARMA(1,1)

```
class ARMA11(sm.tsa.statespace.MLEModel):
    def __init__(self, endog):
        # Initialize the state space model
        super(ARMA11, self).__init__(endog, k_states=2,
k_posdef=1,

initialization='stationary')

        # Setup the fixed components of the state space
representation
        self['design'] = [1., 0]
        self['transition'] = [[0, 0],
                              [1., 0]]
        self['selection', 0, 0] = 1.

    # Describe how parameters enter the model
    def update(self, params, transformed=True, **kwargs):
        params = super(ARMA11, self).update(params,
transformed, **kwargs)

        self['design', 0, 1] = params[0]
        self['transition', 0, 0] = params[1]
        self['state_cov', 0, 0] = params[2]

    # Specify start parameters and parameter names
    @property
    def start_params(self):
        return [0.,0.,1]  # these are very simple
```

Step4: Create and fit the model

```
mod = ARMA11(endog)
res = mod.fit()
print(res.summary())
```

Step5: Example of SARIMA(1,1,1)x(0,1,1)$_4$

Whereas the previous example showed an ad-hoc creation and estimation of a specific model, the power of object-oriented programming in Python can be leveraged to create generic and reusable estimation classes.

For example, for the common class of (Seasonal) Autoregressive Integrated

Moving Average models (optionally with exogenous regressors), an SARIMAX class has been written to automate the creation and estimation of those types of models. For example, an SARIMA(1,1,1)×(0,1,1,4) model of GDP can be specified and estimated as (an added bonus is that we can download the GDP data on-the-fly from FRED using Pandas):

You may need to install the `pandas_datareader` before you run the following code.

In your Terminal,

```
pip install pandas_datareader
```

or `pip3` if you are python3 user.

```
pip3 install pandas_datareader
```

After the installation, you can run the following code:

```
mod = ARMA11(endog)
res = mod.fit()
print(res.summary())

import statsmodels.api as sm
from pandas_datareader.data import DataReader

gdp = DataReader('GDPC1', 'fred', start='1959', end='12-
31-2014')

# Create the model, here an SARIMA(1,1,1) x (0,1,1,4)
model
mod = sm.tsa.SARIMAX(gdp, order=(1,1,1),
seasonal_order=(0,1,1,4))

# Fit the model via maximum likelihood
res = mod.fit()
print(res.summary())
```