Estimating time series models by state space methods in Python - Statsmodels

September 12, 2018 - Securities and Exchange Commission

Chad Fulton†. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Federal Reserve Board

† The views expressed are solely the responsibility of the author and should not be interpreted as reflecting the views of the Board of Governors of the Federal Reserve System, or anyone else in the Federal Reserve System.
Python

- General purpose programming language: *it can do a lot*
- High level: *it is easy to write*
- Heavily used for scientific computing: *lots of resources*
Interest over time

Python    R  MATLAB

Average

Jan 1, 2004   Jan 1, 2010   Jan 1, 2016

Housekeeping

For the rest of this presentation I am using Python 3.6 with:

```python
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
```
Scientific python ecosystem

**numpy** - "Numeric python" - arrays and matrices

```python
X = np.random.normal(size=(500, 2))
eps = np.random.normal(size=500)
beta = np.array([2, -2])

y = X.dot(beta) + eps

beta_hat = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
print(beta_hat)

array([[ 2.0287782 , -2.03871258]])
```
Scientific python ecosystem

**pandas** - working with data

dta = pd.read_csv('fredmd_2018-09.csv', skiprows=[1])
dta.index = pd.DatetimeIndex(
    start='1959-01', periods=len(dta), freq='MS')
print(dta.loc['2018-01':'2018-06', 'FEDFUNDS':'TB6MS'])

<table>
<thead>
<tr>
<th></th>
<th>FEDFUNDS</th>
<th>CP3Mx</th>
<th>TB3MS</th>
<th>TB6MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-01</td>
<td>1.41</td>
<td>1.63</td>
<td>1.41</td>
<td>1.59</td>
</tr>
<tr>
<td>2018-02-01</td>
<td>1.42</td>
<td>1.78</td>
<td>1.57</td>
<td>1.75</td>
</tr>
<tr>
<td>2018-03-01</td>
<td>1.51</td>
<td>2.08</td>
<td>1.70</td>
<td>1.87</td>
</tr>
<tr>
<td>2018-04-01</td>
<td>1.69</td>
<td>2.20</td>
<td>1.76</td>
<td>1.93</td>
</tr>
<tr>
<td>2018-05-01</td>
<td>1.70</td>
<td>2.16</td>
<td>1.86</td>
<td>2.02</td>
</tr>
<tr>
<td>2018-06-01</td>
<td>1.82</td>
<td>2.19</td>
<td>1.90</td>
<td>2.06</td>
</tr>
</tbody>
</table>
Scientific python ecosystem

**statsmodels** - "Statistical models" - highlights include:

- Linear regression: OLS, GLS, WLS, Quantile, Recursive
- Generalized linear models
- Time-series:
  - Exponential smoothing, SARIMAX, Unobserved components
  - VARMAX, Dynamic Factors
  - Markov-switching
  - Full state space model framework
- Hypothesis testing
Statsmodels
Where

- **Project website**: https://www.statsmodels.org/
- **Github**: https://github.com/statsmodels/statsmodels
- **Mailing list**: https://groups.google.com/forum/#!forum/pystatsmodels
How

Typical workflow:

1. Create a model:

```python
model = sm.OLS(endog, exog)
```

2. Estimate the parameters of the model, via `fit`

```python
results = model.fit()
```

3. Print a text summary of the results

```python
print(results.summary())
```
Note on variable naming

- **endog** is the "left-hand-side variable"
- **exog** are explanatory "right-hand-side variables"

This convention is followed throughout Statsmodels.
from pandas_datareader.data import DataReader

cpia = DataReader('CPIAPPNS', 'fred', start='1990')
cpia.plot(title='CPI - Apparel', figsize=(8, 3))
mod = sm.tsa.UnobservedComponents(cpi, 'local level', seasonal=12)
res = mod.fit()
print(res.summary())
res.plot_components( observed=False, figsize=(8, 6));

Predicted vs observed

Level component

Seasonal component

Note: The first 12 observations are not shown, due to approximate diffuse initialization.
State space models in Statsmodels
Resources

• **Working paper**: "Estimating time series models by state space methods in Python - Statsmodels" (Fulton, 2017)

• **My website**: http://www.chadfulton.com/topics.html

• **Statsmodels documentation**: https://www.statsmodels.org/dev/statespace.html

• **Mailing list**: https://groups.google.com/forum/#!forum/pystatsmodels
State space models

\[
y_t = d_t + Z_t \alpha_t + \varepsilon_t \\
\alpha_{t+1} = c_t + T_t \alpha_t + R_t \eta_t
\]

\[
\varepsilon_t \sim N(0, H_t) \\
\eta_t \sim N(0, Q_t)
\]

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_t$</td>
<td>'obs_intercept'</td>
</tr>
<tr>
<td></td>
<td>Observation intercept</td>
</tr>
<tr>
<td>$Z_t$</td>
<td>'design'</td>
</tr>
<tr>
<td></td>
<td>Design matrix</td>
</tr>
<tr>
<td>$H_t$</td>
<td>'obs_cov'</td>
</tr>
<tr>
<td></td>
<td>Observation disturbance covariance matrix</td>
</tr>
<tr>
<td>$c_t$</td>
<td>'state_intercept'</td>
</tr>
<tr>
<td></td>
<td>State intercept</td>
</tr>
<tr>
<td>$T_t$</td>
<td>'transition'</td>
</tr>
<tr>
<td></td>
<td>Transition matrix</td>
</tr>
<tr>
<td>$R_t$</td>
<td>'selection'</td>
</tr>
<tr>
<td></td>
<td>Selection matrix</td>
</tr>
<tr>
<td>$Q_t$</td>
<td>'state_cov'</td>
</tr>
<tr>
<td></td>
<td>State disturbance covariance matrix</td>
</tr>
</tbody>
</table>
Why

Many basic time series models fall under the state space framework:

- ARIMA (or, more generally, SARIMAX)
- Unobserved components models (e.g. local level)
- VAR (or, more generally, VARMAX)
- Dynamic factor models
Many models of interest to macroeconomists can be estimated via the state space framework:

- DSGE models (linearized + Gaussian)
- Time-varying parameters models (e.g. TVP-VAR models)
- Regime-switching models (e.g. Markov switching dynamic factor models)
State space model:

$$y_t = d_t + Z_t \alpha_t + \varepsilon_t \quad \varepsilon_t \sim N(0, H_t)$$

$$\alpha_{t+1} = c_t + T_t \alpha_t + R_t \eta_t \quad \eta_t \sim N(0, Q_t)$$

$$\alpha_1 \sim N(a_1, P_1)$$

AR(1) model:

$$y_t = \nu + \phi y_{t-1} + \eta_t, \quad \eta_t \sim N(0, \sigma^2)$$

In state space form ($c_t = H_t = 0, Z_t = R_t = 1, c_t = \nu, T_t = \phi, Q_t = \sigma^2$):

$$y_t = \alpha_t$$

$$\alpha_{t+1} = \nu + \phi \alpha_t + \eta_t \quad \eta_t \sim N(0, \sigma^2)$$
class AR1(sm.tsa.statespace.MLEModel):
    _start_params = [0., 0., 1.]
    _param_names = ['nu', 'phi', 'sigma']

    def __init__(self, endog):
        super().__init__(endog, k_states=1,
                         initialization='stationary')

        self['design', 0, 0] = 1  # Set Z_t = 1
        self['selection', 0, 0] = 1  # Set R_t = 1

    def update(self, params, **kwargs):
        params = super().update(params, **kwargs)

        self['state_intercept', 0, 0] = params[0]  # c_t = nu
        self['transition', 0, 0] = params[1]  # T_t = phi
        self['state_cov', 0, 0] = params[2]**2  # Q_t = sigma^2
Get some data:

```python
from pandas_datareader.data import DataReader

cpi = DataReader('CPIAUCSL', 'fred')
inf = (cpi - cpi.shift(1)) / cpi.shift(1) * 100
inf.plot(title='CPI Inflation - monthly', figsize=(10, 5));
```
Construct the model:

```python
model = AR1(inf)
```

Estimate the parameters of the model:

```python
results = model.fit()
```

Print the output:

```python
print(results.summary())
```
Statespace Model Results

Model: AR1  Log Likelihood -228.424
Date: Wed, 12 Sep 2018  AIC 462.848
Time: 01:06:06  BIC 470.752
Sample: 01-01-2010 - 07-01-2018  HQIC 466.049
Covariance Type: opg

==============================================================================
coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
nu      1.0830      0.250      4.338      0.000       0.594       1.572
phi     0.3735      0.069      5.377      0.000       0.237       0.510
sigma   2.2700      0.141     16.104      0.000       1.994       2.546
==============================================================================

Ljung-Box (Q): 41.44  Jarque-Bera (JB): 4.00
Prob(Q): 0.41  Prob(JB): 0.14
Heteroskedasticity (H): 1.18  Skew: -0.34
Prob(H) (two-sided): 0.63  Kurtosis: 3.69

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step)
Out-of-sample forecasts:

```
forecasts = results.forecasts('2020')
```
Evaluate model fit:

```python
results.plot_diagnostics(figsize=(8, 8))
```
Can handle missing data:

```python
inf_missing = inf.copy()
inf_missing.loc['2014-06':'2015-06'] = np.nan
model = AR1(inf_missing)
# ...
```

**AR(1) forecast of CPI Inflation (w/ missing)**
```python
class AR1(sm.tsa.statespace.MLEModel):
    _start_params = [0., 0., 1.]
    _param_names = ['nu', 'phi', 'sigma']

def __init__(self, endog):
    super().__init__(endog, k_states=1,
                     initialization='stationary')

    self['design', 0, 0] = 1  # Set Z_t = 1
    self['selection', 0, 0] = 1  # Set R_t = 1

def update(self, params, **kwargs):
    params = super().update(params, **kwargs)

    self['state_intercept', 0, 0] = params[0]  # c_t = nu
    self['transition', 0, 0] = params[1]  # T_t = phi
    self['state_cov', 0, 0] = params[2]**2  # Q_t = sigma^2
```
As a child of `sm.tsa.statespace.MLEModel`, our `AR1` class inherits the following methods (among others):

- **loglike**: evaluate the loglikelihood of the model at a given set of parameters
  - Returns a number

- **smooth**: perform full Kalman filtering and smoothing at a given set of parameters
  - Returns a `Results` object

- **fit**: find parameters that maximize the likelihood estimation
  - Returns a `Results` object
Details: Results attributes

All results objects inherit the following attributes (among others):

- **params**: the parameters used to create the `Results` object (may not be MLE if `smooth` was used)
- **bse**: the standard errors of those parameter estimates
- **llf**: the loglikelihood at those parameters
- **fittedvalues**: the one-step-ahead predictions of the model
- **resid**: the one-step-ahead forecast errors
- **aic**, **bic**, **hqic**: information criteria for model selection
Details: filter / smoother attributes

All results objects contain almost all of the Kalman filter / smoother output described by Durbin and Koopman (2012). Among others, these include:

- `filtered_state, smoothed_state` : the filtered or smoothed estimates of the underlying state vector

- `filtered_state_cov, smoothed_state_cov` : the covariance of the filtered or smoothed estimates of the underlying state vector

- `standardized_forecasts_error` : the standardized one-step-ahead forecast errors
Details: Results methods

All results objects inherit the following methods (among others):

- **summary**: produce a text summary table
- **predict, get_prediction**: in-sample prediction (only point values or with confidence intervals)
- **forecast, get_forecast**: out-of-sample forecasting (only point values or with confidence intervals)
- **impulse_responses**: compute impulse response functions
- **simulate**: simulate a new time series
- **simulate**: simulate a new time series
Other major state space features:

- Filtered and smoothed estimates of the state vector
  - Smoothed lag-one autocovariance (useful for DFM)
- Simulation smoother
- Exact diffuse initialization
- Univariate treatment of multivariate series
- Collapsing large observation vectors
- Simulation of time series data
Other major features:

- **Fast**: underlying filter, smoother, and simulation smoother are compiled (Cython)

- **Documented**: generated API documentation, example notebooks, working paper

- **Tested**: nearly 2000 unit tests (for state space alone) that run continuously
class AR1(sm.tsa.statespace.MLEModel):
    _start_params = [0., 0., 1.]
    _param_names = ['nu', 'phi', 'sigma']

    def __init__(self, endog):
        super().__init__(endog, k_states=1,
                         initialization='stationary')

        self['design', 0, 0] = 1  # Set Z_t = 1
        self['selection', 0, 0] = 1  # Set R_t = 1

    def update(self, params, **kwargs):
        params = super().update(params, **kwargs)

        self['state_intercept', 0, 0] = params[0]  # c_t = nu
        self['transition', 0, 0] = params[1]  # T_t = phi
        self['state_cov', 0, 0] = params[2]**2  # Q_t = sigma^2
Details

There are two required methods of any model:

- `__init__`: initialize the model
- `update`: update the parameters in the system matrices
Details: **__init__**: 

```python
def __init__(self, endog):
    super().__init__(endog, k_states=1,
                     initialization='stationary')

    self['design', 0, 0] = 1  # Set Z_t = 1
    self['selection', 0, 0] = 1  # Set R_t = 1
```

- Initialize the base state space model class (the `super` call)
- Initialize fixed elements of system matrices (e.g. Z_t = 1)
- Initialize the first state in the model (e.g. initialization='stationary')
def update(self, params, **kwargs):
    params = super().update(params, **kwargs)

    self['state_intercept', 0, 0] = params[0]  # c_t = nu
    self['transition', 0, 0] = params[1]       # T_t = phi
    self['state_cov', 0, 0] = params[2]**2      # Q_t = sigma^2

- Basic parameter handling, e.g. transformations (the super call)
- Map parameter values into system matrices (e.g. T_t = params[1])
Details: maximum likelihood estimation

The **fit** method performs maximum likelihood estimation, and usually does not need to be defined in a class like `AR1`.

- Numerically maximizes the likelihood function
- Requires **starting parameters** (e.g. using `_start_params`, above, but can be more complex)
- The optimization method (like BFGS, Nelder-Mead, Powell, etc.) can be selected (e.g. `fit(method="powell")`)
- Optimization parameters can be tuned (e.g. `fit(maxiter=1000)`)

Often times, we want to impose restrictions on the estimated parameters.

- For example, we may want to require that \(-1 < \phi < 1\).

In the Statsmodels state space package, restrictions are implemented using parameter transformations.

1. The optimizer selects over an unconstrained parameter space.

2. The unconstrained parameter is transformed into a constrained parameter that is valid for the model.

3. The constrained parameter is placed into the state space system matrix.
Example: parameter restrictions

```python
def transform_params(self, unconstrained):
    constrained = unconstrained.copy()

    # Require: -1 < phi < 1
    tmp = unconstrained[1]
    constrained[1] = tmp / (1 + np.abs(tmp))

    # Require: sigma2 > 0

    return constrained
```

Note: but is important to also define the inverse transformation in `untransform_params`.
Built-in parameter restrictions

Restrictions to induce stationarity for AR(p), MA(q), and VAR(p), VMA(q) are a little tedious to write (as are their inverses), so we have them built-in.

In `sm.tsa.statespace.tools`:

- `constrain_stationary_univariate`, `unconstrain_stationary_univariate`
- `constrain_stationary_multivariate`, `constrain_stationary_multivariate`
def transform_params(self, unconstrained):
    constrained = unconstrained.copy()

    # Require: -1 < phi < 1
    constrained[1] = constrain_stationary_univariate(unconstrained[1])

    # Require: sigma2 > 0

    return constrained

def untransform_params(self, constrained):
    unconstrained = constrained.copy()

    # Reverse: -1 < phi < 1
    unconstrained[1] = unconstrain_stationary_univariate(constrained)

    # Reverse: sigma2 > 0
    unconstrained[2] = constrained[2]**0.5

    return unconstrained
Starting parameters for maximum likelihood estimation can be specified in three ways:

1. `_start_params` class attribute

2. `start_params` class property

3. Can be overridden in call to `fit`

```python
@property
def start_params(self):
    y = self.endog[1:]
    X = np.c_[np.ones(self.nobs - 1), self.endog[:-1]]
    nu, phi = np.linalg.pinv(X).dot(y)
    sigma = np.std(y)
    return np.r_[nu, phi, sigma]
```

```python
res = mod.fit(start_params=[1, 2, 3])
```
Built-in state space models

- SARIMAX
- Unobserved components
- VARMAX
- Dynamic factors
- Recursive least squares
What's next?

We'd love to get more feedback

- Bug reports
- Feature requests
- Use cases
- Questions on the mailing list
What's next?

We'd love to get more developers.

- Example: so far we only have basic support for VAR analysis (SVAR, FEVD, IRFs, etc.)

- Example: missing many statistical tests (e.g. Canova-Hansen)

- Example: would be great to get better documentation, more unit tests