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

September 12, 2018 - Securities and Exchange Commission

Chad Fulton† . . . . Federal Reserve Board

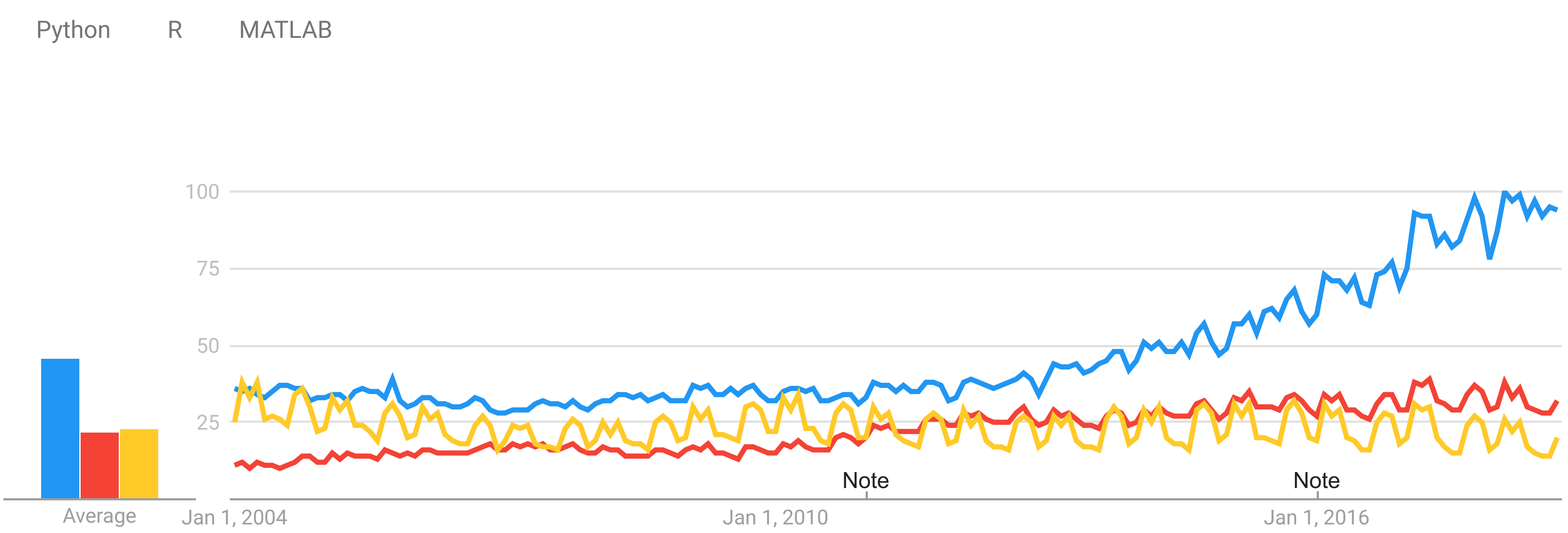
† The views expressed are solely the responsibility of the author and should not be interpreted as reflecting the views of the Board of Governors of the Federal Reserve System, or anyone else in the Federal Reserve System.

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

United States. 1/1/04 - 9/12/18. Web Search.



United States. 1/1/04 - 9/12/18. Web Search.

Housekeeping

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

```
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

```
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'])
```

```
            FEDFUNDS  CP3Mx  TB3MS  TB6MS
2018-01-01      1.41   1.63   1.41   1.59
2018-02-01      1.42   1.78   1.57   1.75
2018-03-01      1.51   2.08   1.70   1.87
2018-04-01      1.69   2.20   1.76   1.93
2018-05-01      1.70   2.16   1.86   2.02
2018-06-01      1.82   2.19   1.90   2.06
```

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:

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

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

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

3. Print a text summary of the results

```
print(results.summary())
```

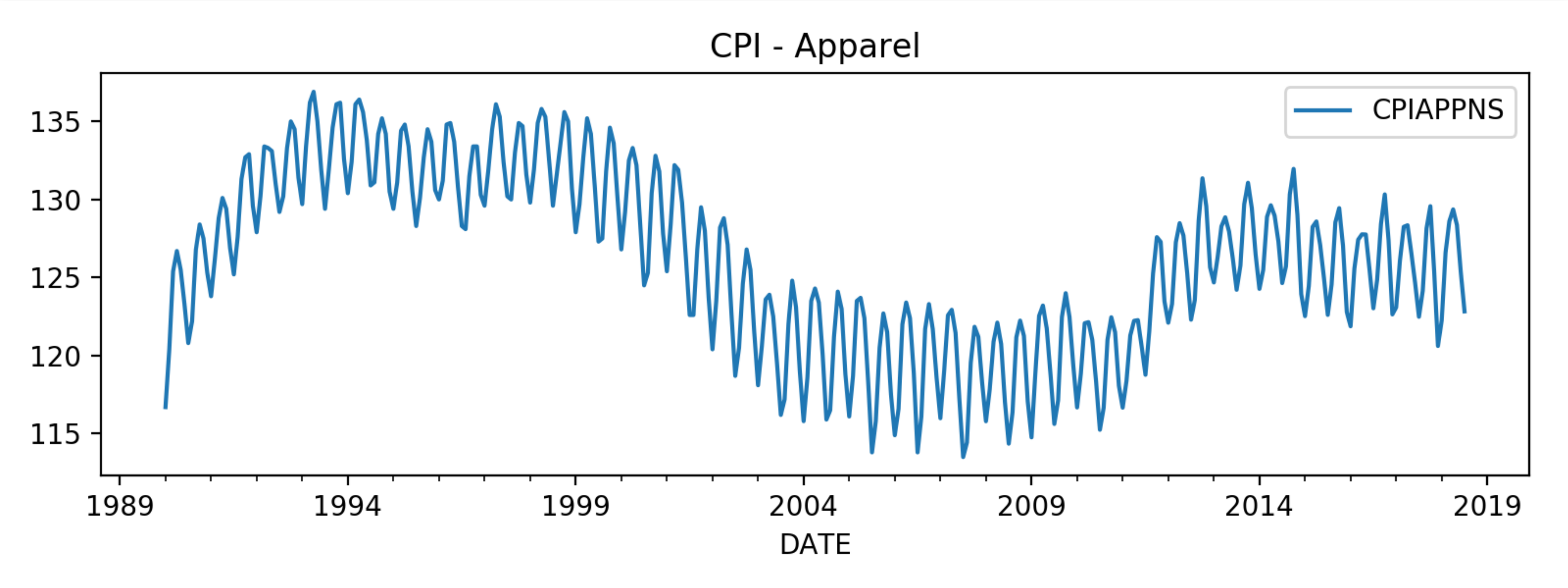
Note on variable naming

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

This convention is followed throughout Statsmodels.

Example: seasonal adjustment

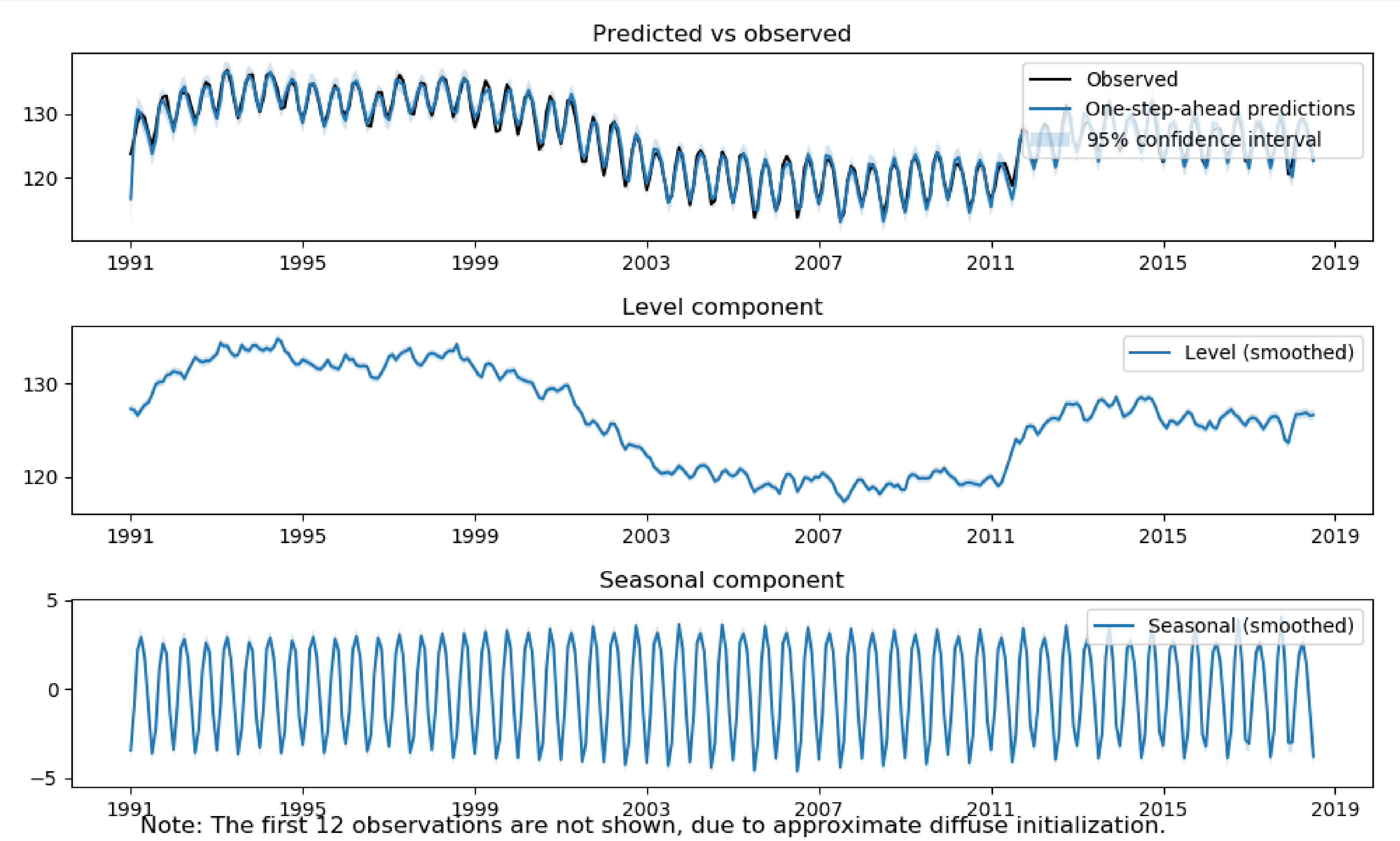
```
from pandas_datareader.data import DataReader

cpia = DataReader('CPIAPPNS', 'fred', start='1990')
cpia.plot(title='CPI - Apparel', figsize=(8, 3))
```



```python
mod = sm.tsa.UnobservedComponents(cpia, 'local level', seasonal=12)
res = mod.fit()
print(res.summary())
```

```
                        Unobserved Components Results
=====================================================================================
Dep. Variable:                         CPIAPPNS   No. Observations:              343
Model:                              local level   Log Likelihood            -408.642
                   + stochastic seasonal(12)   AIC                        823.285
Date:                          Wed, 12 Sep 2018   BIC                        834.691
Time:                                  10:15:50   HQIC                       827.834
Sample:                              01-01-1990
                                   - 07-01-2018
Covariance Type:                            opg
====================================================================================
                       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
sigma2.irregular   2.45e-11      0.031   7.81e-10      1.000      -0.062       0.062
sigma2.level         0.4213      0.046      9.189      0.000       0.331       0.511
sigma2.seasonal      0.0237      0.007      3.562      0.000       0.011       0.037
===================================================================================
Ljung-Box (Q):                      287.50   Jarque-Bera (JB):                  1.63
Prob(Q):                              0.00   Prob(JB):                          0.44
Heteroskedasticity (H):               1.36   Skew:                              0.17
Prob(H) (two-sided):                  0.11   Kurtosis:                          3.03
===================================================================================
```

```python
res.plot_components(observed=False, figsize=(8, 6));
```



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

What

State space models

$$
\begin{aligned}
y_t &= d_t + Z_t \alpha_t + \varepsilon_t \qquad & \varepsilon_t &\sim N(0, H_t) \\
\alpha_{t+1} &= c_t + T_t \alpha_t + R_t \eta_t \qquad & \eta_t &\sim N(0, Q_t)
\end{aligned}
$$

Time Series Analysis by State Space Methods: Second Edition.
Durbin, James, and Siem Jan Koopman. 2012.
Oxford University Press.

	Attribute	Description
d_t	`'obs_intercept'`	Observation intercept
Z_t	`'design'`	Design matrix
H_t	`'obs_cov'`	Observation disturbance covariance matrix
c_t	`'state_intercept'`	State intercept
T_t	`'transition'`	Transition matrix
R_t	`'selection'`	Selection matrix
Q_t	`'state_cov'`	State disturbance covariance matrix

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

Why

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)

How

State space model:

$$
\begin{aligned}
y_t &= d_t + Z_t \alpha_t + \varepsilon_t & \varepsilon_t &\sim N(0, H_t) \\
\alpha_{t+1} &= c_t + T_t \alpha_t + R_t \eta_t & \eta_t &\sim N(0, Q_t) \\
\alpha_1 &\sim N(a_1, P_1)
\end{aligned}
$$

AR(1) model:

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

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

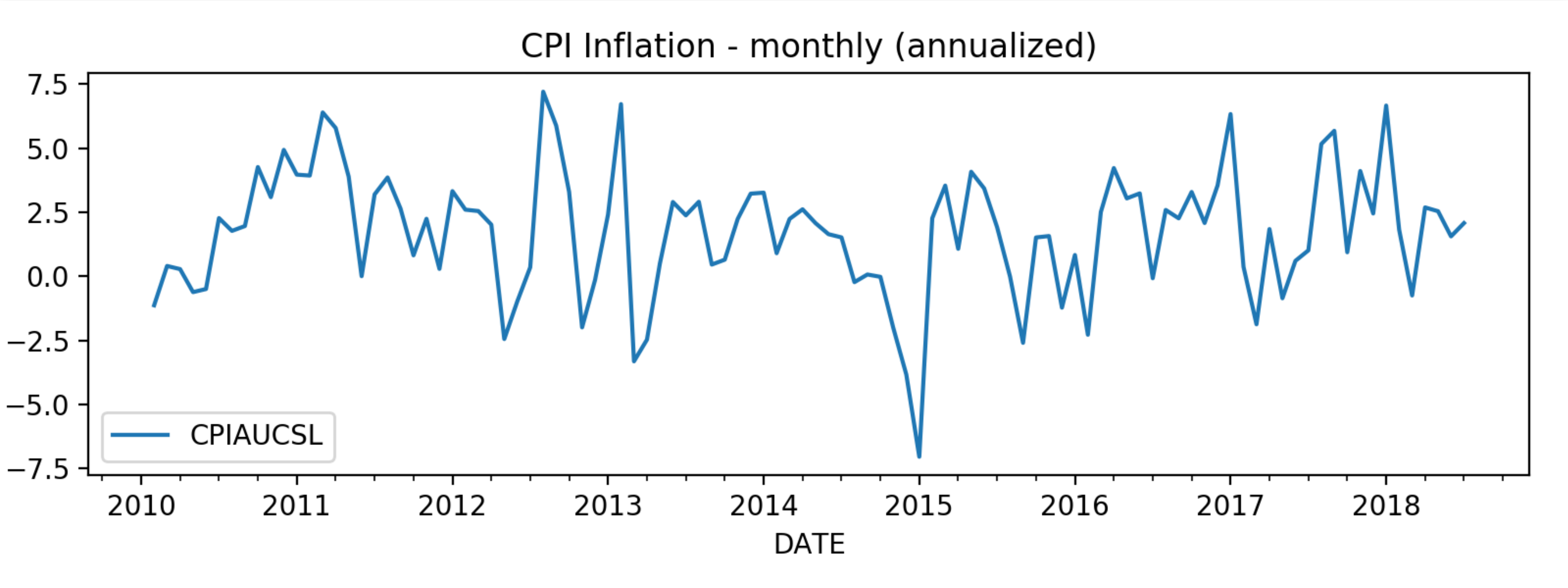
$$
\begin{aligned}
y_t &= \alpha_t \\
\alpha_{t+1} &= \nu + \phi \alpha_t + \eta_t & \eta_t &\sim N(0, \sigma^2)
\end{aligned}
$$

How: AR(1) in 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
```

Get some data:

```
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:

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

Estimate the parameters of the model:

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

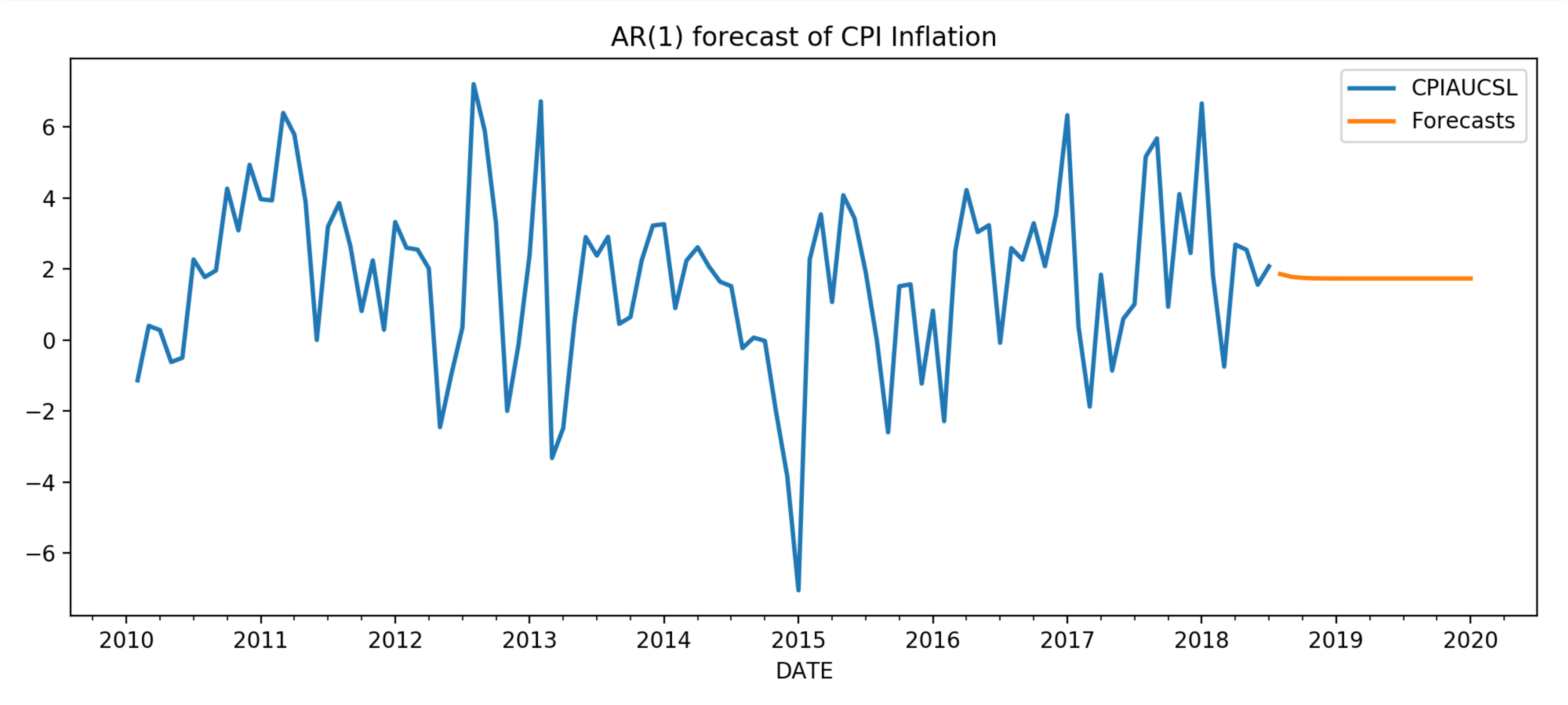
Print the output:

```
print(results.summary())
```

```
                           Statespace Model Results
==============================================================================
Dep. Variable:                CPIAUCSL   No. Observations:                  103
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   HQIC                           466.049
                          - 07-01-2018
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)
```

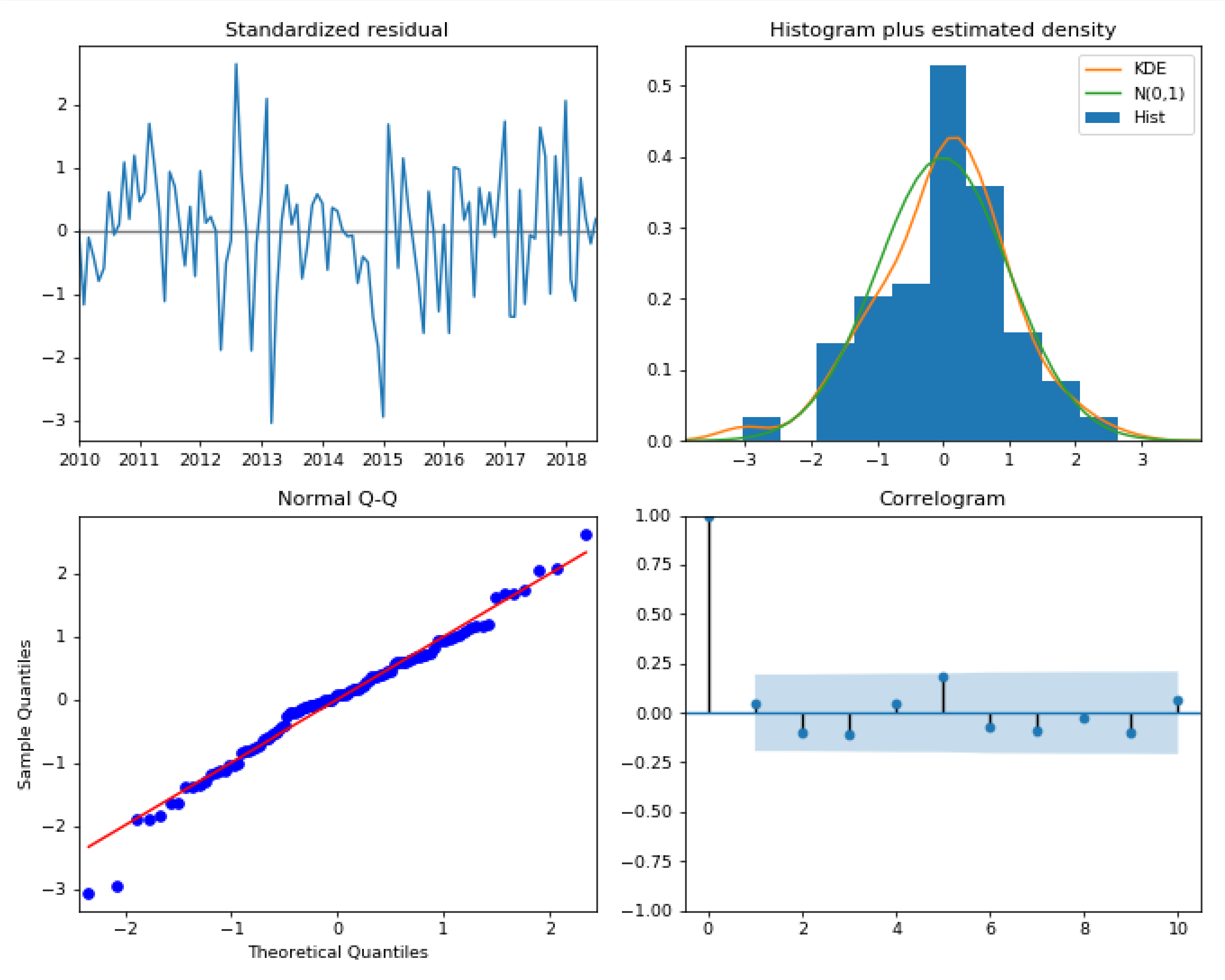
Out-of-sample forecasts:

```
forecasts = results.forecasts('2020')
```



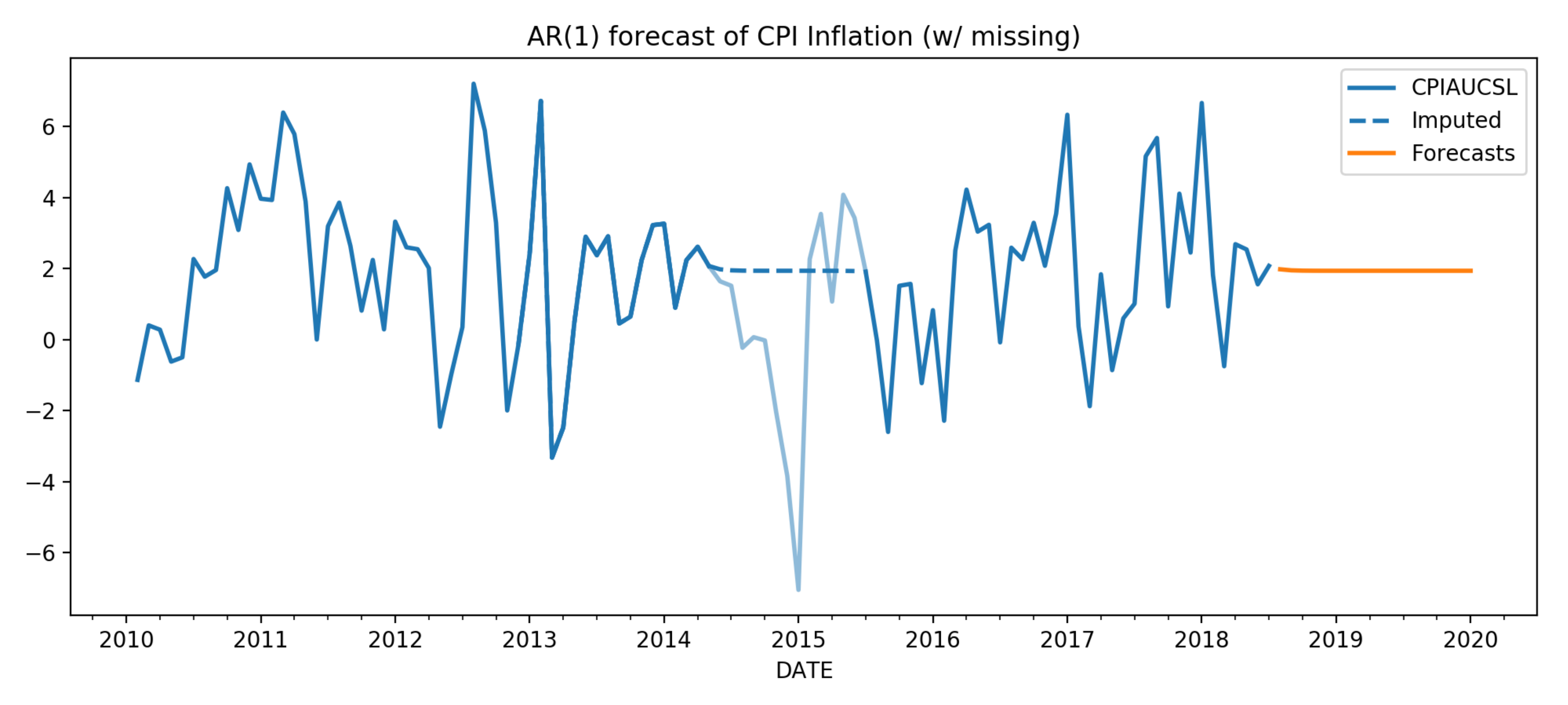
Evalaute model fit:

```
results.plot_diagnostics(figsize=(8, 8))
```



Can handle missing data:

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

model = AR1(inf_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
```

Details: Model

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

```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
```

Details

There are two required methods of any model:

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

Details: `__init__`:

```
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'`)

Details: **update**:

```
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)`)

Details: parameter restrictions

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

```
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
    constrained[2] = unconstrained[2]**2

    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`

Example: parameter restrictions

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

    # Require: -1 < phi < 1
    constrained[1] = constrain_stationary_univariate(unconstrained[1
    # Require: sigma2 > 0
    constrained[2] = unconstrained[2]**2

    return constrained

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

    # Reverse: -1 < phi < 1
    unconstrained[1] = unconstrain_stationary_univariate(constrainec
    # Reverse: sigma2 > 0
    unconstrained[2] = constrained[2]**0.5

    return unconstrained
```

Details: starting parameters

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

1. `_start_params` class attribute
2. `start_params` class property

```
@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]
```

3. Can be overridden in call to `fit`

```
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