Time Series Forecasting with Python – Cheat Sheet

Data Science with Marco

Time series analysis

ACF plot

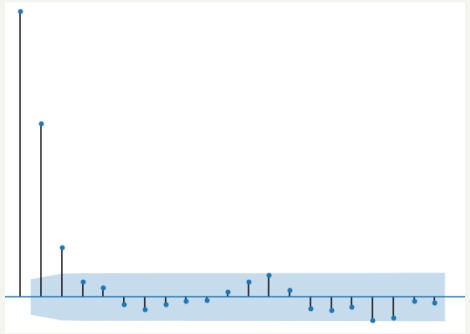
The autocorrelation function (ACF) plot shows the autocorrelation coefficients as a function of the lag.

- Use it to determine the order *q* of a stationary MA(q) process
- A stationary MA(q) process has significant coefficients up until lag *q*

```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(data)
```

Output for a MA(2) process (i.e., q = 2):



PACF plot

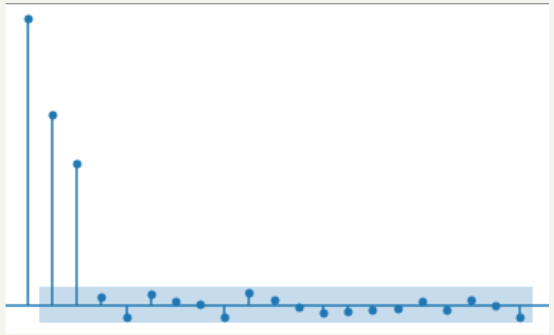
The partial autocorrelation function (PACF) plot shows the partial autocorrelation coefficients as a function of the lag.

- Use it to determine the order *p* of a stationary AR(p) process
- A stationary AR(p) process has significant coefficients up until lag *p*

```
from statsmodels.graphics.tsaplots import plot_pacf

plot_pacf(data)
```

Output for an AR(2) process (i.e., p = 2):



Time series decomposition

Separate the series into 3 components: trend, seasonality, and residuals

- Trend: long-term changes in the series
- Seasonality: periodical variations in the series
- Residuals: what is not explained by trend and seasonality

```
from statsmodels.tsa.seasonal import STL

decomp = STL(data, period=m).fit()

fig, (ax1, ax2, ax3, ax4) =
plt.subplots(nrows=4, ncols=1, sharex=True)

ax1.plot(decomp.observed)
ax1.set_ylabel('Observed')

ax2.plot(decomp.trend)
ax2.set_ylabel('Trend')

ax3.plot(decomp.seasonal)
ax3.set_ylabel('Seasonal')

ax4.plot(decomp.resid)
ax4.set_ylabel('Residuals')
```

Note: m is the frequency of data (i.e., how many observations per season)

Time Series Forecasting with Python – Cheat Sheet

Data Science with Marco

Statistical tests

ADF test – Test for stationarity

A series is stationary it its **mean**, **variance**, and **autocorrelation** are constant over time. Test for stationarity with augmented Dickey-Fuller (ADF) test.

- Null hypothesis: a unit root is present (i.e., the series is not stationary)
- We want a p-value < 0.05

```
from statsmodels.tsa.stattools import adfuller

_, p-value = adfuller(data)
```

Note: to make a series stationary, use differencing.

- *n = 1*: difference between consecutive timesteps
- *n = 4*: difference between values 4 timesteps apart

Differencing removes *n* data points.

```
import numpy as np

diff_data = np.diff(data, n=1)
```

Ljung-Box test – Residuals analysis

Used to determine if the autocorrelation of a group of data is significantly different from 0. Use it on the residuals to check if they are independent.

- Null hypothesis: the data is independently distributed (i.e., there is no autocorrelation)
- We want a p-value > 0.05

```
from statsmodels.stats.diagnostic import acorr_ljungbox

residuals = model.resid

_, pvalue = acorr_ljungbox(residuals, np.arange(1, h, 1))

print(pvalue)
```

Note: print the p-values up to *h* lags, where *h* is the length of your forecast horizon

Granger causality – Multivariate forecasting

Determine if one time series is useful in predicting the other one.
Use to validate the VAR model. If Granger causality test fails, then the VAR model is invalid.

- Null hypothesis: $y_{2,t}$ does not Granger-causes $y_{1,t}$
- Works for predictive causality
- Tests causality in one direction only (i.e., must run the test twice)
- We want a p-value < 0.05

```
from statsmodels.tsa.stattools import grangercausalitytests

# Note how we test the causality in both directions
# by running the test twice

granger_1 = grangercausalitytests(data[['t1','t2']])
granger_2 = grangercausalitytests(data[['t2','t1']])
```

Note: see how we run the test twice to test causality in both directions

Time Series Forecasting with Python – Cheat Sheet

Data Science with Marco

Forecasting – Statistical models

Moving average model – MA(q)

The moving average model: the current value depends on the mean of the series, the current error term, and past error terms.

- Denoted as MA(q) where *q* is the order
- Use ACF plot to find *q*
- Assumes stationarity. Use only on stationary data

Equation

$$y_t = \mu + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \cdots + \theta_q\epsilon_{t-q}$$

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(data, order=(0,0,q))
res = model.fit()
predictions = res.get_prediction(start_index, end_index)
```

Autoregressive model – AR(p)

The autoregressive model is a regression against itself. This means that the present value depends on past values.

- Denoted as AR(p) where *p* is the order
- Use PACF to find *p*
- Assumes stationarity. Use only on stationary data

Equation

$$y_t = C + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \epsilon_t$$

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(data, order=(p,0,0))
res = model.fit()
predictions = res.get_prediction(start_index, end_index)
```

ARMA(p,q)

The autoregressive moving average model (ARMA) is the combination of the autoregressive model AR(p), and the moving average model MA(q).

- Denoted as ARMA(p,q) where *p* is the order of the autoregressive portion, and *q* is the order of the moving average portion
- Cannot use ACF or PACF to find the order *p*, and *q*. Must try different (p,q) value and select the model with the lowest **AIC** (Akaike's Information Criterion)
- Assumes stationarity. Use only on stationary data.

Equation

$$y_t = C + \phi_1 y_{t-1} + \cdots \phi_p y_{t-p} + \theta_1\epsilon_{t-1} + \cdots \theta_q\epsilon_{t-q} + \epsilon_t$$

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(data, order=(p,0,q))
res = model.fit()
predictions = res.get_prediction(start_index, end_index)
```

Time Series Forecasting with Python – Cheat Sheet

Data Science with Marco

Forecasting – Statistical models

ARIMA(p,d,q)

The autoregressive integrated moving average (ARIMA) model is the combination of the autoregressive model AR(p), and the moving average model MA(q), but in terms of the differenced series.

- Denoted as ARMA(p,d,q), where *p* is the order of the autoregressive portion, *d* is the order of integration, and *q* is the order of the moving average portion
- Can use on non-stationary data

Equation

$$y'_t = C + \phi_1 y'_{t-1} + \cdots \phi_p y'_{t-p} + \theta_1 \epsilon_{t-1} + \cdots \theta_q \epsilon_{t-q} + \epsilon_t$$

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(data, order=(p,d,q))
res = model.fit()
predictions = res.get_prediction(start_index, end_index)
```

Note: the order of integration *d* is simply the number of time a series was differenced to become stationary.

SARIMA(p,d,q)(P,D,Q)$_m$

The seasonal autoregressive integrated moving average (SARIMA) model includes a seasonal component on top of the ARIMA model.

- Denoted as SARIMA(p,d,q)(P,D,Q)m. Here, p, d, and q have the same meaning as in the ARIMA model.
- *P* is the seasonal order of the autoregressive portion
- *D* is the seasonal order of integration
- *Q* is the seasonal order of the moving average portion
- *m* is the frequency of the data (i.e., the number of data points in one season)

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(data, order=(p,d,q),seasonal_order
(P,D,Q,m))

res = model.fit()
predictions = res.get_prediction(start_index, end_index)
```

SARIMAX

SARIMAX is the most general model. It combines seasonality, a moving average portion, an autoregressive portion, and exogenous variables.

- Can use external variables to forecast a series

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

model = SARIMAX(target, exog, order=(p,d,q),
seasonal_order=(P,D,Q,m))

res = model.fit()
predictions = res.get_prediction(start_index, end_index)
```

Caveat: SARIMAX predicts the next timestep. If your horizon is longer than one timestep, then you must forecast your exogenous variables too, which can amplify the error in your model

Time Series Forecasting with Python – Cheat Sheet

Data Science with Marco

Forecasting – Statistical models

VARMAX

The vector autoregressive moving average with exogenous variables (VARMAX) model is used for multivariate forecasting (i.e., predicting two time series at the same time)

- Assumes Granger-causality. Must use the Granger-causality test. If the test fails, the VARMAX model cannot be used.

```
from statsmodels.tsa.statespace.varmax import VARMAX

model = VARMAX(target, exog, order=(p,q))
res = model.fit(disp=False)
predictions = res.get_prediction(start_index, end_index)
```

BATS and TBATS

BATS and TBATS are used when the series has more than one seasonal period. This can happen when we have high frequency data, such as daily data.

- When there is more than one seasonal period, SARIMA cannot be used. Use BATS or TBATS.
- BATS: Box-Cox transformation, ARMA errors, Trend and Seasonal components
- TBATS: Trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend and Seasonal components.

```
from sktime.forecasting.bats import BATS

forecaster = BATS(sp=[period_1, period_2])
forecaster.fit(data)

predictions = forecaster.predict(horizon)

############ Same syntax for TBATS ###################

from sktime.forecasting.tbats import TBATS

forecaster = TBATS(sp=[period_1, period_2])
forecaster.fit(data)

predictions = forecaster.predict(horizon)
```

Exponential smoothing

Exponential smoothing uses past values to predict the future, but the weights decay exponentially as the values go further back in time.

- Simple exponential smoothing: returns flat forecasts
- Double exponential smoothing: adds a trend component. Forecasts are a straight line (increasing or decreasing)
- Triple exponential smoothing: adds a seasonal component
- Trend can be “additive” or “exponential”
- Seasonality can be “additive” or “multiplicative”

```
from statsmodels.tsa.holtwinters import
ExponentialSmoothing

model = ExponentialSmoothing(
  data,
  trend='add',          # or "mul"
  seasonal='add',       # or "mul"
  seasonal_periods=m,
  initialization_method='estimated').fit()

predictions = model.forecast(horizon)
```

Time Series Forecasting with Python – Cheat Sheet

Data Science with Marco

Forecasting – Deep learning models

Deep neural network (DNN)

A deep neural network stacks fully connected layers and can model non-linear relationship in the time series if the activation function is non-linear.

- Start with a simple model with few hidden layers. Experiment training for more epochs before adding layers

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

dnn = Sequential([
  Dense(units=64, activation='relu'),
  Dense(units=64, activation='relu'),
  Dense(units=1)
])

dnn.compile(
  loss='mean_squared_error',
  optimizer='adam',
  metrics=['mean_absolute_error']
)
```

Long short-term memory - LSTM

An LSTM is great at processing sequences of data, such as text and time series.

Its architecture allows for past information to still be used for later predictions

- You can stack many LSTM layers in your model
- You can try combining an LSTM with a CNN
- An LSTM is longer to train since the data is processed in sequence

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, LSTM

lstm_model = Sequential([
  LSTM(32, return_sequences=True),
  Dense(units=1)
])

lstm_model.compile(loss='mean_squared_error',
                   optimizer='adam',
                   metrics=['mean_absolute_error']
)
```

Convolutional neural network - CNN

A CNN can act as a filter for our time series, due to the convolution operation which reduces the feature space.

- A CNN trains faster than an LSTM
- Can be combined with an LSTM. Place the CNN layer before the LSTM

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Conv1D

cnn_model = Sequential([
  Conv1D(filters=32,
         kernel_size=(KERNEL_WIDTH,),
         activation='relu'),
  Dense(units=32, activation='relu'),
  Dense(units=1)
])

cnn_model.compile(loss='mean_squared_error',
                  optimizer='adam',
                  metrics=['mean_absolute_error']
)
```

Time Series Forecasting with Python – Cheat Sheet

Data Science with Marco

Forecasting – Deep learning models

Autoregressive deep learning model

An autoregressive deep learning model feeds its predictions back into the model to make further predictions. That way, we generate a sequence of predictions, one forecast at a time.

- Can be used with any architecture, whether it's a DNN, LSTM or CNN
- If early predictions are bad, the errors will magnify as more predictions are made.

Autoregressive deep learning model (cont.)

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, LSTMCell, RNN

class AutoRegressive(Model):

  def __init__(self, units, out_steps):
    super().__init__()
    self.out_steps = out_steps
    self.units = units
    self.lstm_cell = LSTMCell(units)
    self.lstm_rnn = RNN(self.lstm_cell,
                        return_state=True)
    self.dense = Dense(train_df.shape[1])

    def warmup(self, inputs):
      x, *state = self.lstm_rnn(inputs)
      prediction = self.dense(x)

      return prediction, state

    def call(self, inputs, training=None):
      predictions = []
      prediction, state = self.warmup(inputs)

      predictions.append(prediction)

      for n in range(1, self.out_steps):
        x = prediction
        x, state = self.lstm_cell(x,
                                  states=state,
                                  training=training)

        prediction = self.dense(x)
        predictions.append(prediction)

        predictions = tf.stack(predictions)
        predictions = tf.transpose(predictions,[1, 0, 2])

        return predictions

AR_LSTM = AutoRegressive(units=32, out_steps=24)
```