Chapter 24

ARIMA Model for Forecasting

A popular and widely used statistical method for time series forecasting is the ARIMA model. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a class of model that captures a suite of different standard temporal structures in time series data. In this tutorial, you will discover how to develop an ARIMA model for time series data with Python. After completing this tutorial, you will know:

- About the ARIMA model the parameters used and assumptions made by the model.
- How to fit an ARIMA model to data and use it to make forecasts.
- How to configure the ARIMA model on your time series problem.

Let's get started.

24.1 Autoregressive Integrated Moving Average Model

An ARIMA model is a class of statistical models for analyzing and forecasting time series data. It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- **AR**: *Autoregression*. A model that uses the dependent relationship between an observation and some number of lagged observations.
- **I**: *Integrated*. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- **MA**: *Moving Average*. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of `ARIMA(p,d,q)` where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used.

The parameters of the ARIMA model are defined as follows:

- p: The number of lag observations included in the model, also called the lag order.
- d: The number of times that the raw observations are differenced, also called the degree of differencing.
- q: The size of the moving average window, also called the order of moving average.

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model. A value of 0 can be used for a parameter, which indicates to not use that element of the model. This way, the ARIMA model can be configured to perform the function of an ARMA model, and even a simple AR, I, or MA model.

Adopting an ARIMA model for a time series assumes that the underlying process that generated the observations is an ARIMA process. This may seem obvious, but helps to motivate the need to confirm the assumptions of the model in the raw observations and in the residual errors of forecasts from the model. Next, let's take a look at how we can use the ARIMA model in Python. We will start with loading a simple univariate time series.

24.2 Shampoo Sales Dataset

In this lesson, we will use the Shampoo Sales dataset as an example. This dataset describes the monthly number of sales of shampoo over a 3 year period. You can learn more about the dataset in Appendix A.1. Place the dataset in your current working directory with the filename `shampoo-sales.csv`. Below is an example of loading the Shampoo Sales dataset with Pandas with a custom function to parse the date-time field.

```
# load and plot dataset
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
    squeeze=True, date_parser=parser)
# summarize first few rows
print(series.head())
# line plot
series.plot()
pyplot.show()
```

Listing 24.1: Load and plot the Shampoo Sales dataset.

Running the example prints the first 5 rows of the dataset.

```
Month
1901-01-01   266.0
1901-02-01   145.9
1901-03-01   183.1
1901-04-01   119.3
1901-05-01   180.3
```

Listing 24.2: Example output of the first 5 rows of the Shampoo Sales dataset.

The data is also plotted as a time series with the month along the x-axis and sales figures on the y-axis.

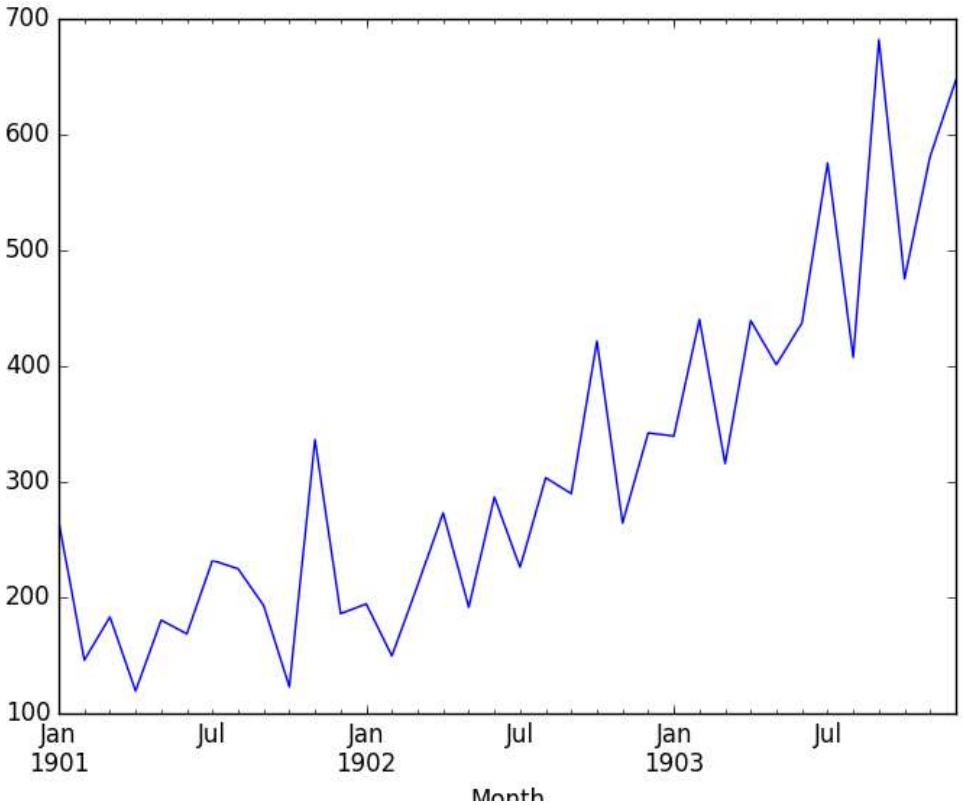

Figure 24.1: Line plot of the Shampoo Sales dataset.

We can see that the Shampoo Sales dataset has a clear trend. This suggests that the time series is not stationary and will require differencing to make it stationary, at least a difference order of 1. Let's also take a quick look at an autocorrelation plot of the time series. This is also built-in to Pandas. The example below plots the autocorrelation for a large number of lags in the time series.

```
# autocorrelation plot of time series
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from pandas.tools.plotting import autocorrelation_plot
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
    squeeze=True, date_parser=parser)
```

```
# autocorrelation plot
autocorrelation_plot(series)
pyplot.show()
```

Listing 24.3: Autocorrelation plot of the Shampoo Sales dataset.

Running the example, we can see that there is a positive correlation with the first 10-to-12 lags that is perhaps significant for the first 5 lags. A good starting point for the AR parameter of the model may be 5.

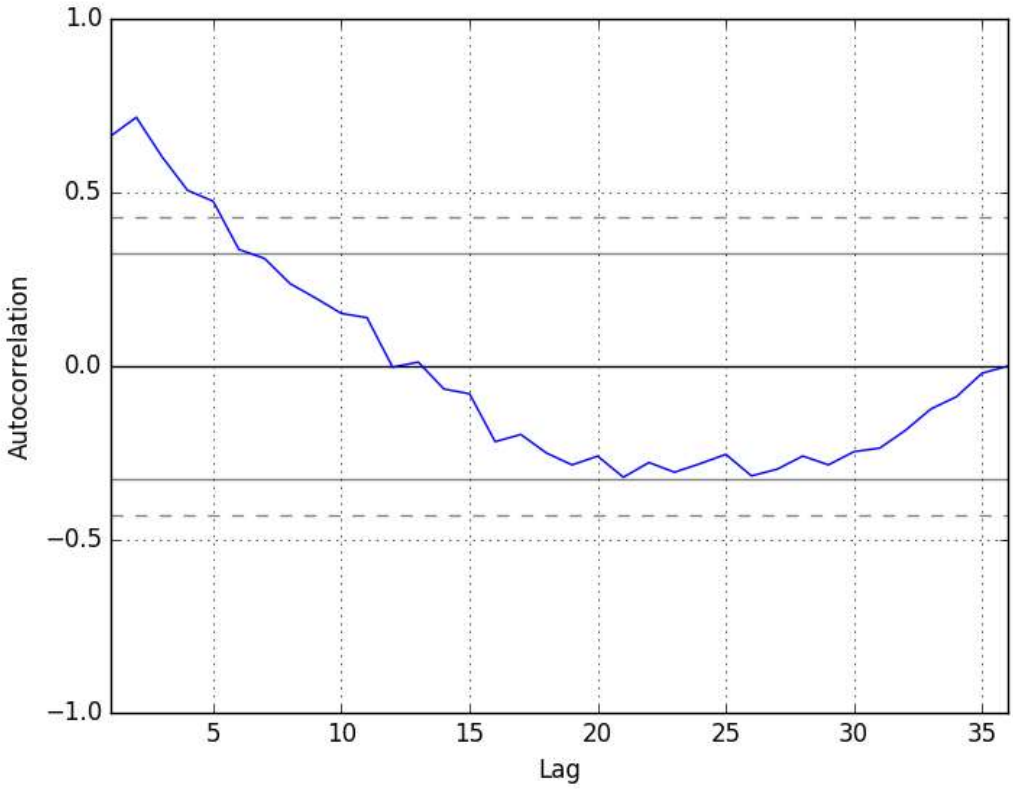

Figure 24.2: Autocorrelation plot of the Shampoo Sales dataset.

24.3 ARIMA with Python

The Statsmodels library provides the capability to fit an ARIMA model. An ARIMA model can be created using the Statsmodels library as follows:

1. Define the model by calling `ARIMA()`[1] and passing in the `p`, `d`, and `q` parameters.
2. The model is prepared on the training data by calling the `fit()` function[2].

[1] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMA.html

[2] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMA.fit.html

3. Predictions can be made by calling the `predict()` function[3] and specifying the index of the time or times to be predicted.

Let's start off with something simple. We will fit an ARIMA model to the entire Shampoo Sales dataset and review the residual errors. First, we fit an `ARIMA(5,1,0)` model. This sets the lag value to 5 for autoregression, uses a difference order of 1 to make the time series stationary, and uses a moving average model of 0. When fitting the model, a lot of debug information is provided about the fit of the linear regression model. We can turn this off by setting the `disp` argument to 0.

```
# fit an ARIMA model and plot residual errors
from pandas import read_csv
from pandas import datetime
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
    squeeze=True, date_parser=parser)
# fit model
model = ARIMA(series, order=(5,1,0))
model_fit = model.fit(disp=0)
# summary of fit model
print(model_fit.summary())
# line plot of residuals
residuals = DataFrame(model_fit.resid)
residuals.plot()
pyplot.show()
# density plot of residuals
residuals.plot(kind='kde')
pyplot.show()
# summary stats of residuals
print(residuals.describe())
```

Listing 24.4: ARIMA model of the Shampoo Sales dataset.

Running the example prints a summary of the fit model. This summarizes the coefficient values used as well as the skill of the fit on the on the in-sample observations.

```
                             ARIMA Model Results
==============================================================================
Dep. Variable:             D.Sales  No. Observations:                   35
Model:                ARIMA(5, 1, 0)  Log Likelihood                -196.170
Method:                       css-mle  S.D. of innovations             64.241
Date:                Fri, 20 Jan 2017  AIC                            406.340
Time:                        15:01:24  BIC                            417.227
Sample:                     02-01-1901  HQIC                           410.098
                          - 12-01-1903
==============================================================================
                 coef    std err          z      P>|z|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
```

[3] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMA.predict.html

```
const          12.0649     3.652     3.304     0.003       4.908   19.222
ar.L1.D.Sales  -1.1082     0.183    -6.063     0.000      -1.466   -0.750
ar.L2.D.Sales  -0.6203     0.282    -2.203     0.036      -1.172   -0.068
ar.L3.D.Sales  -0.3606     0.295    -1.222     0.231      -0.939    0.218
ar.L4.D.Sales  -0.1252     0.280    -0.447     0.658      -0.674    0.424
ar.L5.D.Sales   0.1289     0.191     0.673     0.506      -0.246    0.504
                                    Roots
=============================================================================
                 Real          Imaginary         Modulus        Frequency
-----------------------------------------------------------------------------
AR.1          -1.0617          -0.5064j          1.1763          -0.4292
AR.2          -1.0617          +0.5064j          1.1763           0.4292
AR.3           0.0816          -1.3804j          1.3828          -0.2406
AR.4           0.0816          +1.3804j          1.3828           0.2406
AR.5           2.9315          -0.0000j          2.9315          -0.0000
-----------------------------------------------------------------------------
                0
count   35.000000
mean    -5.495213
std     68.132882
min   -133.296597
25%    -42.477935
50%     -7.186584
75%     24.748357
max    133.237980
```

Listing 24.5: Example output of the ARIMA model coefficients summary.

First, we get a line plot of the residual errors, suggesting that there may still be some trend information not captured by the model.

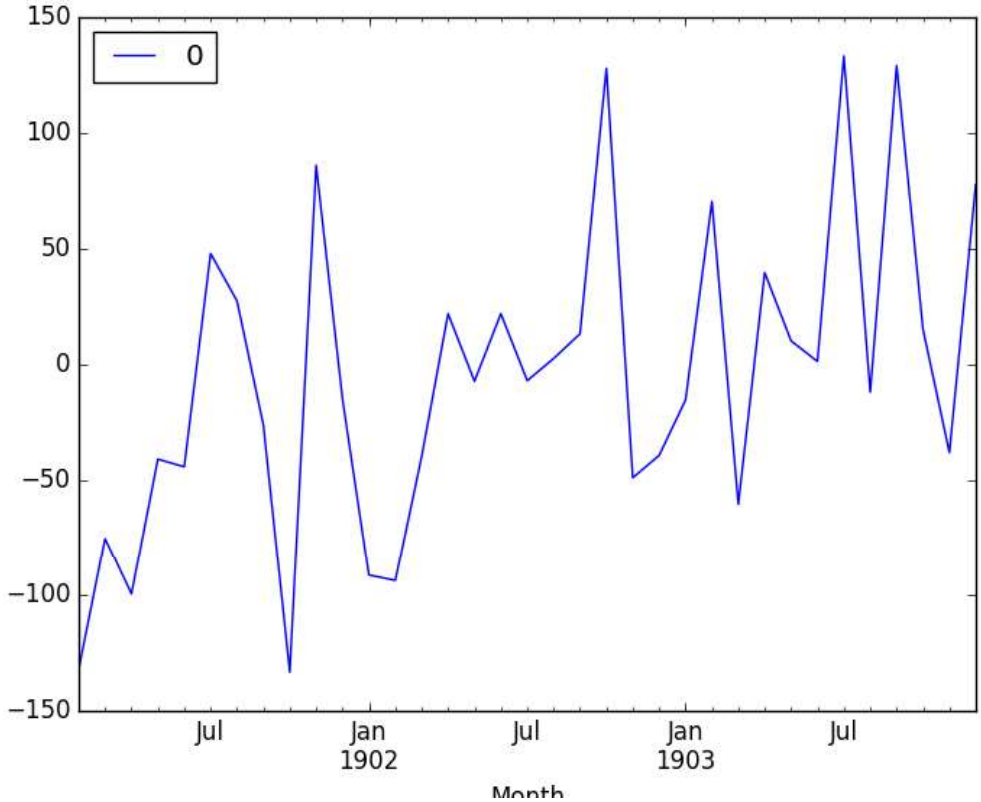

Figure 24.3: Line plot of ARIMA forecast residual errors on the Shampoo Sales dataset.

Next, we get a density plot of the residual error values, suggesting the errors are Gaussian, but may not be centered on zero.

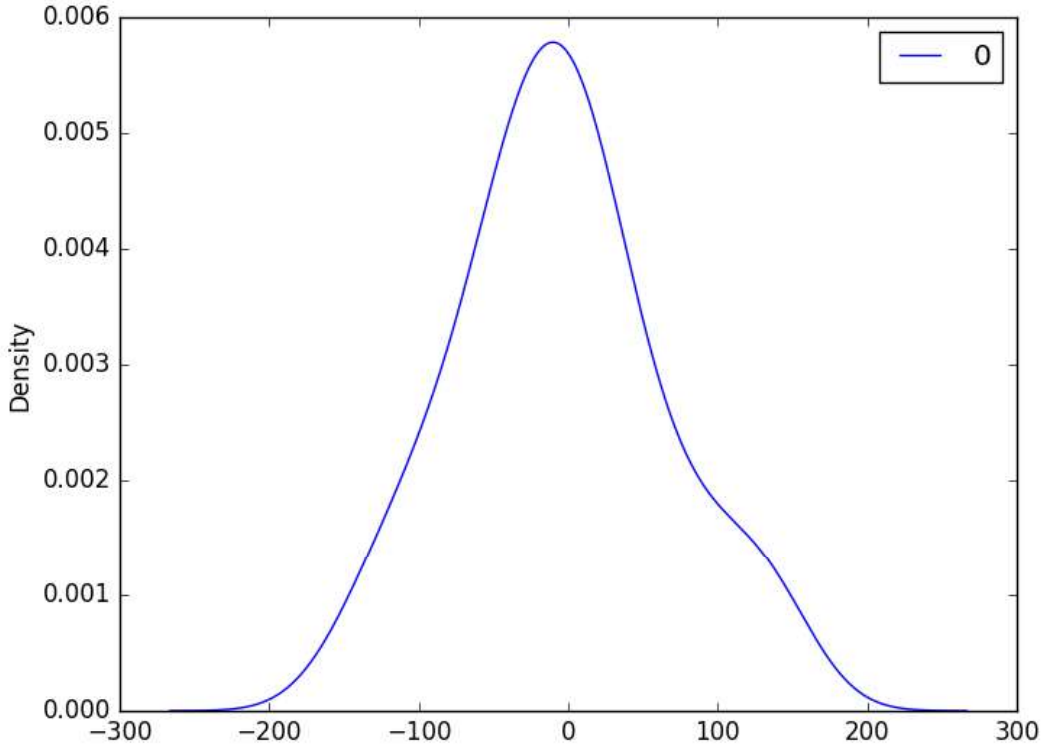

Figure 24.4: Density plot of ARIMA forecast residual errors on the Shampoo Sales dataset.

The distribution of the residual errors is displayed. The results show that indeed there is a bias in the prediction (a non-zero mean in the residuals).

```
count   35.000000
mean    -5.495213
std     68.132882
min   -133.296597
25%    -42.477935
50%     -7.186584
75%     24.748357
max    133.237980
```

Listing 24.6: Example output of the ARIMA model summary statistics of forecast error residuals.

Note, that although above we used the entire dataset for time series analysis, ideally we would perform this analysis on just the training dataset when developing a predictive model. Next, let's look at how we can use the ARIMA model to make forecasts.

24.4 Rolling Forecast ARIMA Model

The ARIMA model can be used to forecast future time steps. We can use the `predict()` function on the `ARIMAResults` object[4] to make predictions. It accepts the index of the time steps to make predictions as arguments. These indexes are relative to the start of the training dataset used to make predictions.

If we used 100 observations in the training dataset to fit the model, then the index of the next time step for making a prediction would be specified to the prediction function as `start=101, end=101`. This would return an array with one element containing the prediction. We also would prefer the forecasted values to be in the original scale, in case we performed any differencing ($d > 0$ when configuring the model). This can be specified by setting the `typ` argument to the value `'levels'`: `typ='levels'`.

Alternately, we can avoid all of these specifications by using the `forecast()` function[5] which performs a one-step forecast using the model. We can split the training dataset into train and test sets, use the train set to fit the model, and generate a prediction for each element on the test set.

A rolling forecast is required given the dependence on observations in prior time steps for differencing and the AR model. A crude way to perform this rolling forecast is to re-create the ARIMA model after each new observation is received. We manually keep track of all observations in a list called history that is seeded with the training data and to which new observations are appended each iteration. Putting this all together, below is an example of a rolling forecast with the ARIMA model in Python.

```
# evaluate an ARIMA model using a walk-forward validation
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
    squeeze=True, date_parser=parser)
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
  model = ARIMA(history, order=(5,1,0))
  model_fit = model.fit(disp=0)
  output = model_fit.forecast()
  yhat = output[0]
```

[4] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMAResults.html

[5] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMAResults.forecast.html

```
  predictions.append(yhat)
  obs = test[t]
  history.append(obs)
  print('predicted=%f, expected=%f' % (yhat, obs))
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot forecasts against actual outcomes
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 24.7: ARIMA model with rolling forecasts on the Shampoo Sales dataset.

Running the example prints the prediction and expected value each iteration. We can also calculate a final root mean squared error score (RMSE) for the predictions, providing a point of comparison for other ARIMA configurations.

```
...
predicted=434.915566, expected=407.600000
predicted=507.923407, expected=682.000000
predicted=435.483082, expected=475.300000
predicted=652.743772, expected=581.300000
predicted=546.343485, expected=646.900000
Test RMSE: 83.417
```

Listing 24.8: Example output of the ARIMA model with rolling forecasts on the Shampoo Sales dataset.

A line plot is created showing the expected values compared to the rolling forecast predictions. We can see the values show some trend and are in the correct scale.

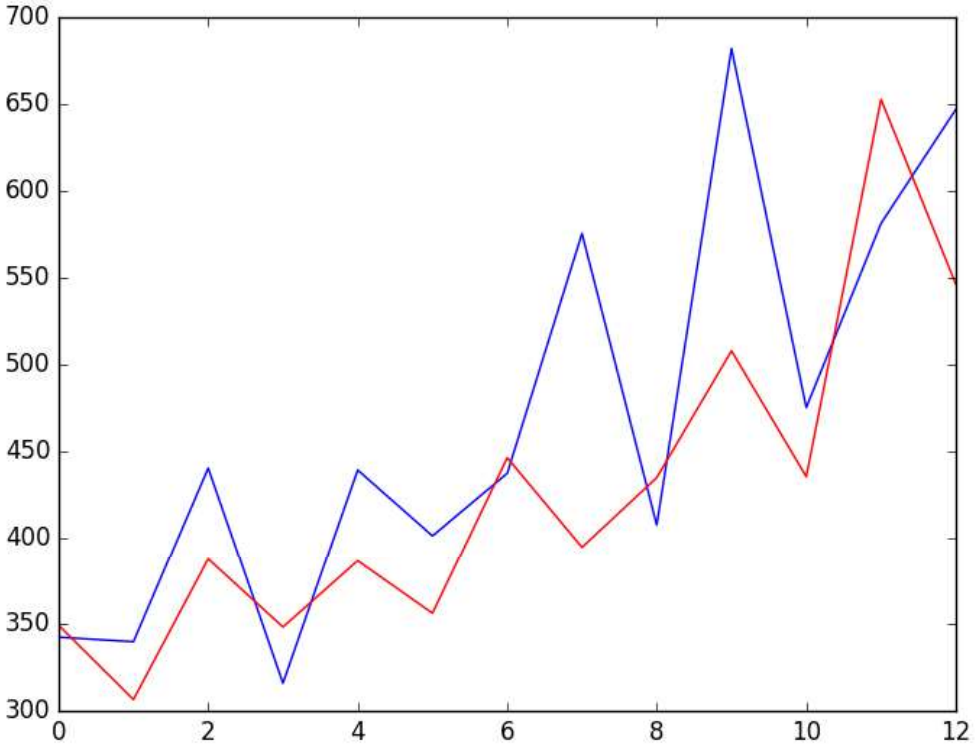

Figure 24.5: Line plot of expected values (blue) and rolling forecast (red) with an ARIMA model on the Shampoo Sales dataset.

The model could use further tuning of the `p`, `d`, and maybe even the `q` parameters.

24.5 Summary

In this tutorial, you discovered how to develop an ARIMA model for time series forecasting in Python. Specifically, you learned:

- About the ARIMA model, how it can be configured, and assumptions made by the model.
- How to perform a quick time series analysis using the ARIMA model.
- How to use an ARIMA model to forecast out of sample predictions.

24.5.1 Next

In the next lesson you will discover how to differentiate ACF and PACF plots and how to read them.

Chapter 25

Autocorrelation and Partial Autocorrelation

Autocorrelation and partial autocorrelation plots are heavily used in time series analysis and forecasting. These are plots that graphically summarize the strength of a relationship with an observation in a time series with observations at prior time steps. The difference between autocorrelation and partial autocorrelation can be difficult and confusing for beginners to time series forecasting. In this tutorial, you will discover how to calculate and plot autocorrelation and partial correlation plots with Python. After completing this tutorial, you will know:

- How to plot and review the autocorrelation function for a time series.
- How to plot and review the partial autocorrelation function for a time series.
- The difference between autocorrelation and partial autocorrelation functions for time series analysis.

Let's get started.

25.1 Minimum Daily Temperatures Dataset

In this lesson, we will use the Minimum Daily Temperatures dataset as an example. This dataset describes the minimum daily temperatures over 10 years (1981-1990) in the city Melbourne, Australia. You can learn more about the dataset in Appendix A.2. Place the dataset in your current working directory with the filename `daily-minimum-temperatures.csv`.

25.2 Correlation and Autocorrelation

Statistical correlation summarizes the strength of the relationship between two variables. We can assume the distribution of each variable fits a Gaussian (bell curve) distribution. If this is the case, we can use the Pearson's correlation coefficient to summarize the correlation between the variables. The Pearson's correlation coefficient is a number between -1 and 1 that describes a negative or positive correlation respectively. A value of zero indicates no correlation.

We can calculate the correlation for time series observations with observations with previous time steps, called lags. Because the correlation of the time series observations is calculated with

values of the same series at previous times, this is called a serial correlation, or an autocorrelation. A plot of the autocorrelation of a time series by lag is called the AutoCorrelation Function, or the acronym ACF. This plot is sometimes called a correlogram or an autocorrelation plot.

Below is an example of calculating and plotting the autocorrelation plot for the Minimum Daily Temperatures using the `plot_acf()` function[1] from the Statsmodels library.

```
# ACF plot of time series
from pandas import Series
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
plot_acf(series)
pyplot.show()
```

Listing 25.1: Create an ACF plot on the Minimum Daily Temperatures dataset.

Running the example creates a 2D plot showing the lag value along the x-axis and the correlation on the y-axis between -1 and 1. Confidence intervals are drawn as a cone. By default, this is set to a 95% confidence interval, suggesting that correlation values outside of this cone are very likely a correlation and not a statistical fluke.

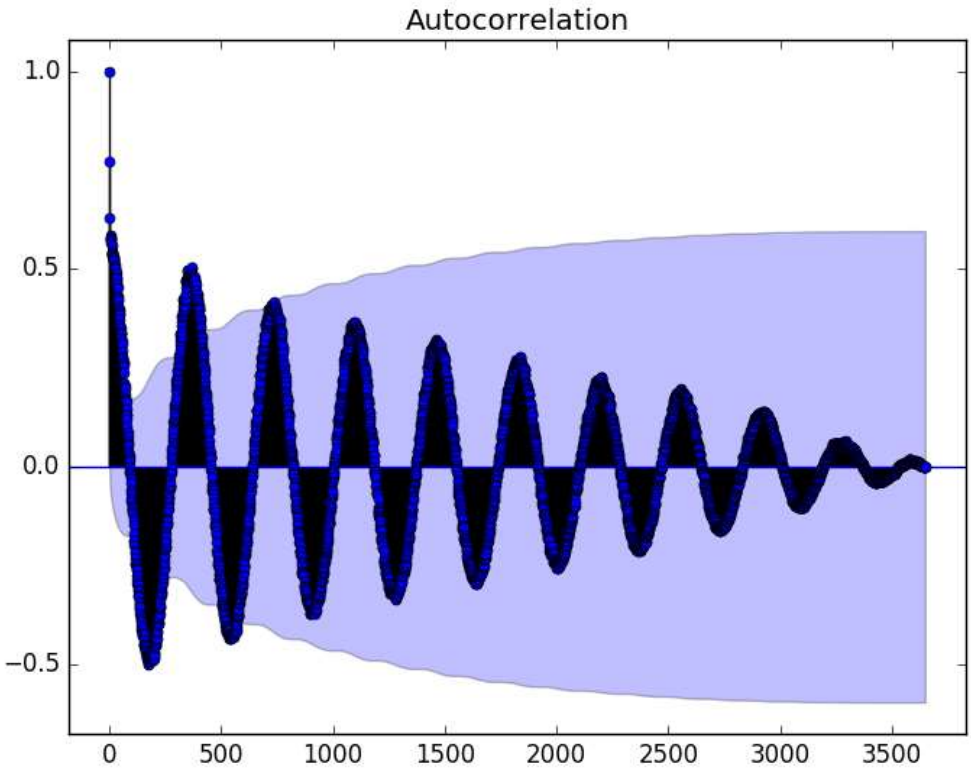

Figure 25.1: ACF plot on the Minimum Daily Temperatures dataset.

[1] http://statsmodels.sourceforge.net/devel/generated/statsmodels.graphics.tsaplots.plot_acf.html

By default, all lag values are printed, which makes the plot noisy. We can limit the number of lags on the x-axis to 50 by setting the `lags` argument.

```
# zoomed-in ACF plot of time series
from pandas import Series
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
plot_acf(series, lags=50)
pyplot.show()
```

Listing 25.2: Create an ACF plot on the Minimum Daily Temperatures dataset.

The resulting plot is easier to read.

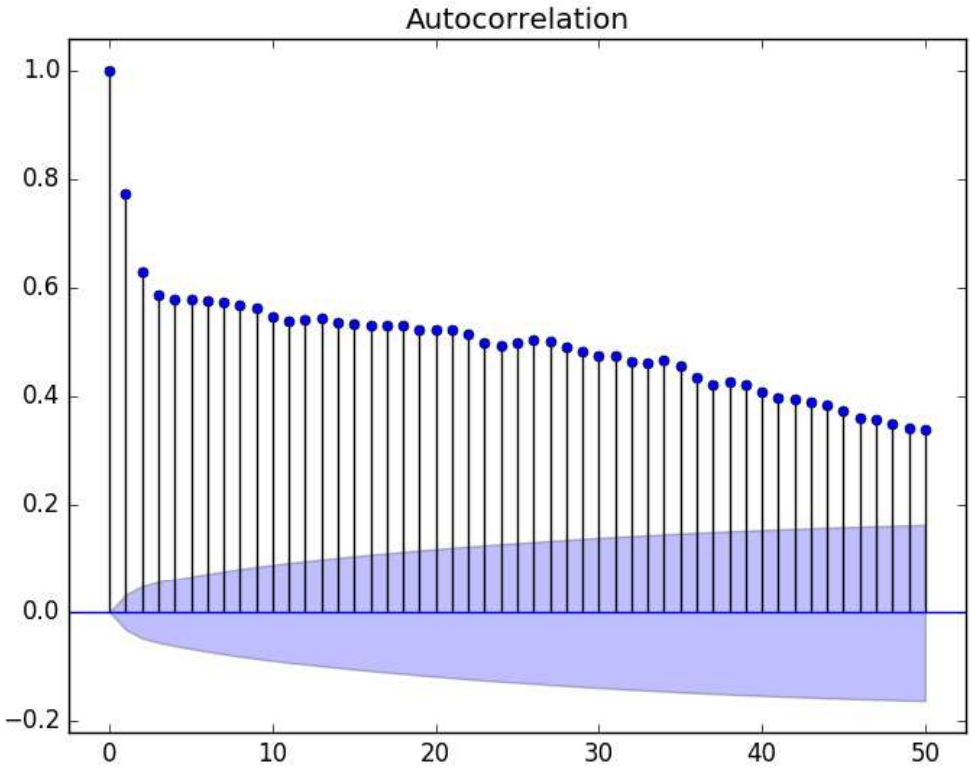

Figure 25.2: ACF plot with lags=50 on the Minimum Daily Temperatures dataset.

25.3 Partial Autocorrelation Function

A partial autocorrelation is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.

> The partial autocorrelation at lag k is the correlation that results after removing the effect of any correlations due to the terms at shorter lags.