Navigation

Machine Learning Mastery
Making Developers Awesome at Machine Learning

Click to Take the FREE Time Series Crash-Course

Search...

How to Create an ARIMA Model for Time Series Forecasting in Python

by **Jason Brownlee** on November 18, 2023 in **Time Series** **834**

Share Tweet Share

A popular and widely used statistical method for time series forecasting is the ARIMA model.

ARIMA stands for AutoRegressive Integrated Moving Average and represents a cornerstone in time series forecasting. It is a statistical method that has gained immense popularity due to its efficacy in handling various standard temporal structures present in time series data.

In this tutorial, you will discover how to develop an ARIMA model for time series forecasting in 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.

Kick-start your project with my new book Time Series Forecasting With Python, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

- **Updated Apr/2019**: Updated the link to dataset.
- **Updated Sep/2019**: Updated examples to use latest API.
- **Updated Dec/2020**: Updated examples to use latest API.
- **Updated Nov/2023**: #####

Autoregressive Integrated Moving Average Model

The ARIMA (AutoRegressive Integrated Moving Average) model stands as a statistical powerhouse 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.

Let’s decode the essence of ARIMA:

- **AR** (Autoregression): This emphasizes the dependent relationship between an observation and its preceding or ‘lagged’ observations.
- **I** (Integrated): To achieve a stationary time series, one that doesn’t exhibit trend or seasonality, differencing is applied. It typically involves subtracting an observation from its preceding observation.
- **MA** (Moving Average): This component zeroes in on the relationship between an observation and the residual error from a moving average model based on lagged observations.

Each of these components is explicitly specified in the model as a parameter. A standard notation is used for 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 lag order, representing the number of lag observations incorporated in the model.
- **d**: Degree of differencing, denoting the number of times raw observations undergo differencing.
- **q**: Order of moving average, indicating the size of the moving average window.

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model.

Interestingly, any of these parameters can be set to 0. Such configurations enable the ARIMA model to mimic the functions of simpler models like ARMA, AR, I, or MA.

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 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.

Shampoo Sales Dataset

The Shampoo Sales dataset provides a snapshot of monthly shampoo sales spanning three years, resulting in 36 observations. Each observation is a sales count. The genesis of this dataset is attributed to Makridakis, Wheelwright, and Hyndman (1998).

Getting Started:

- Download the dataset
- Save it to your current working directory with the filename “shampoo-sales.csv”.

Loading and Visualizing the Dataset:

Below is an example of loading the Shampoo Sales dataset with Pandas with a custom function to parse the date-time field. The dataset is baselined in an arbitrary year, in this case 1900.

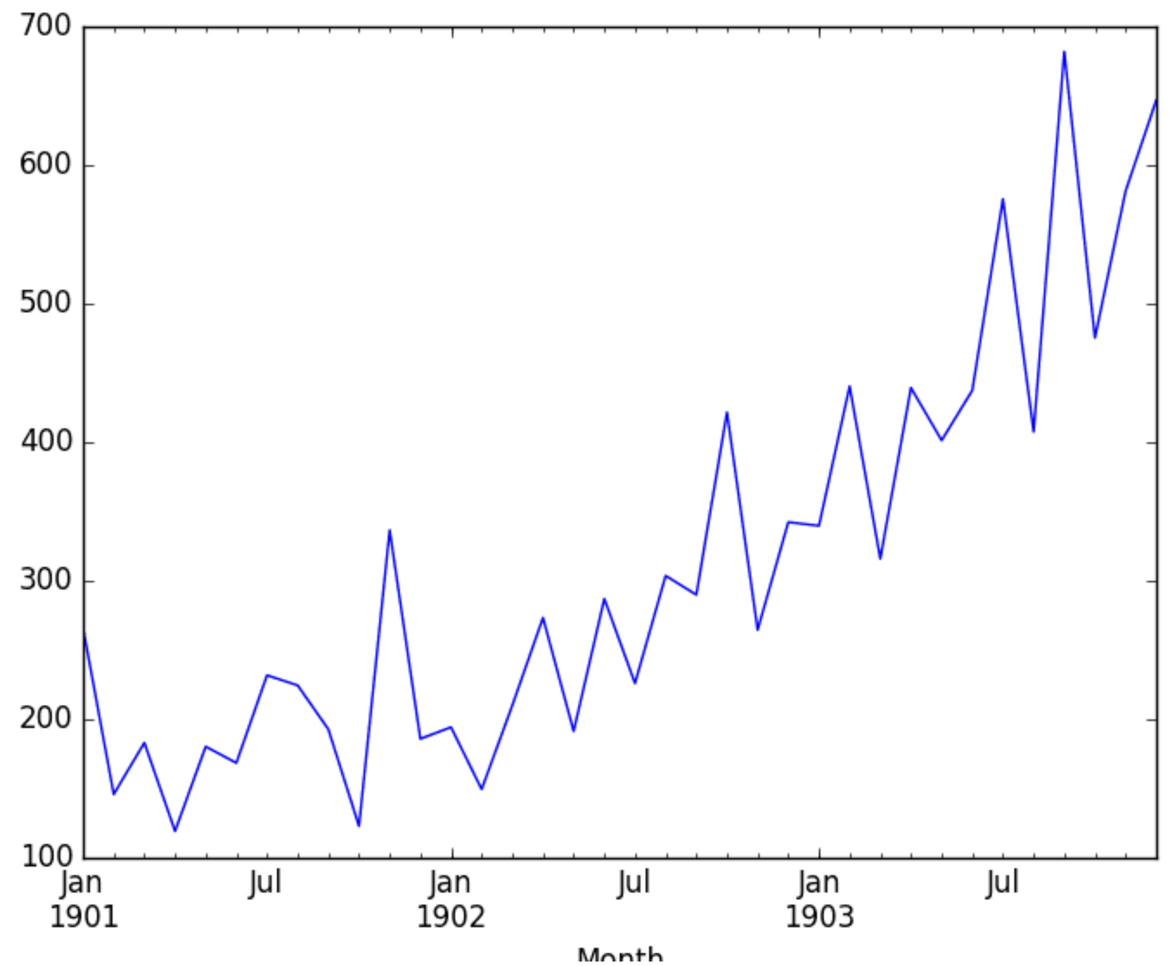
```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

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,
print(series.head())
series.plot()
pyplot.show()
```

When executed, this code snippet will display the initial five dataset entries:

```
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
Name: Sales, dtype: float64
```



Shampoo Sales Dataset Plot

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

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.

Pandas offers a built-in capability to plot autocorrelations. The following example showcases the autocorrelation for an extensive set of time series lags:

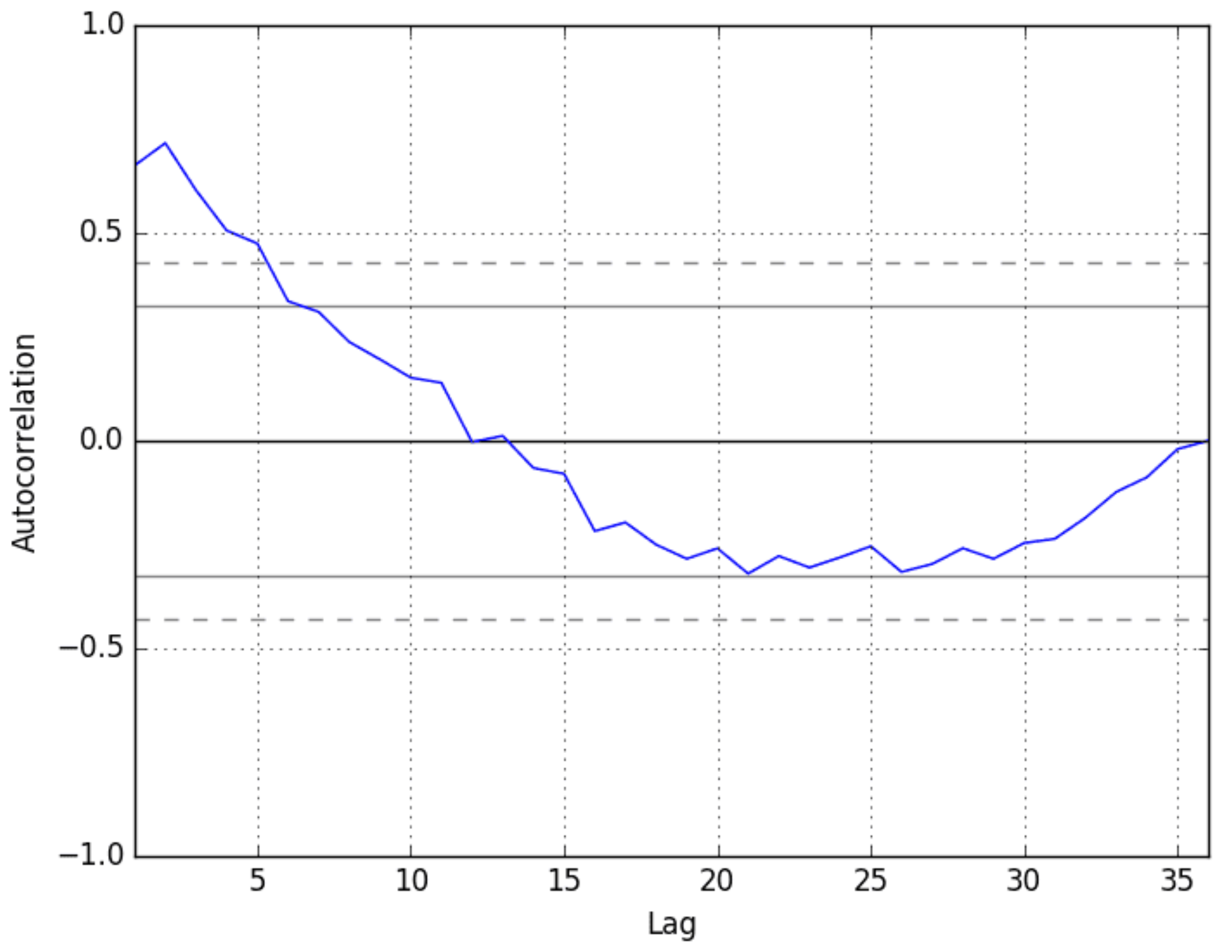
```
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from pandas.plotting import autocorrelation_plot

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,
autocorrelation_plot(series)
pyplot.show()
```

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.

This provides a hint: initiating the AR parameter of our model with a value of 5 could be a beneficial starting point.



Autocorrelation Plot of Shampoo Sales Data

ARIMA with Python

The statsmodels library stands as a vital tool for those looking to harness the power of ARIMA for time series forecasting in Python.

Building an ARIMA Model: A Step-by-Step Guide:

1. **Model Definition**: Initialize the ARIMA model by invoking ARIMA() and specifying the p, d, and q parameters.
2. **Model Training**: Train the model on your dataset using the fit() method.
3. **Making Predictions**: Generate forecasts by utilizing the predict() function and designating the desired time index or indices.

Let’s start with something simple. We will fit an ARIMA model to the entire Shampoo Sales dataset and review the residual errors.

We’ll employ the ARIMA(5,1,0) configuration:

- 5 lags for autoregression (AR)
- 1st order differencing (I)
- No moving average term (MA)

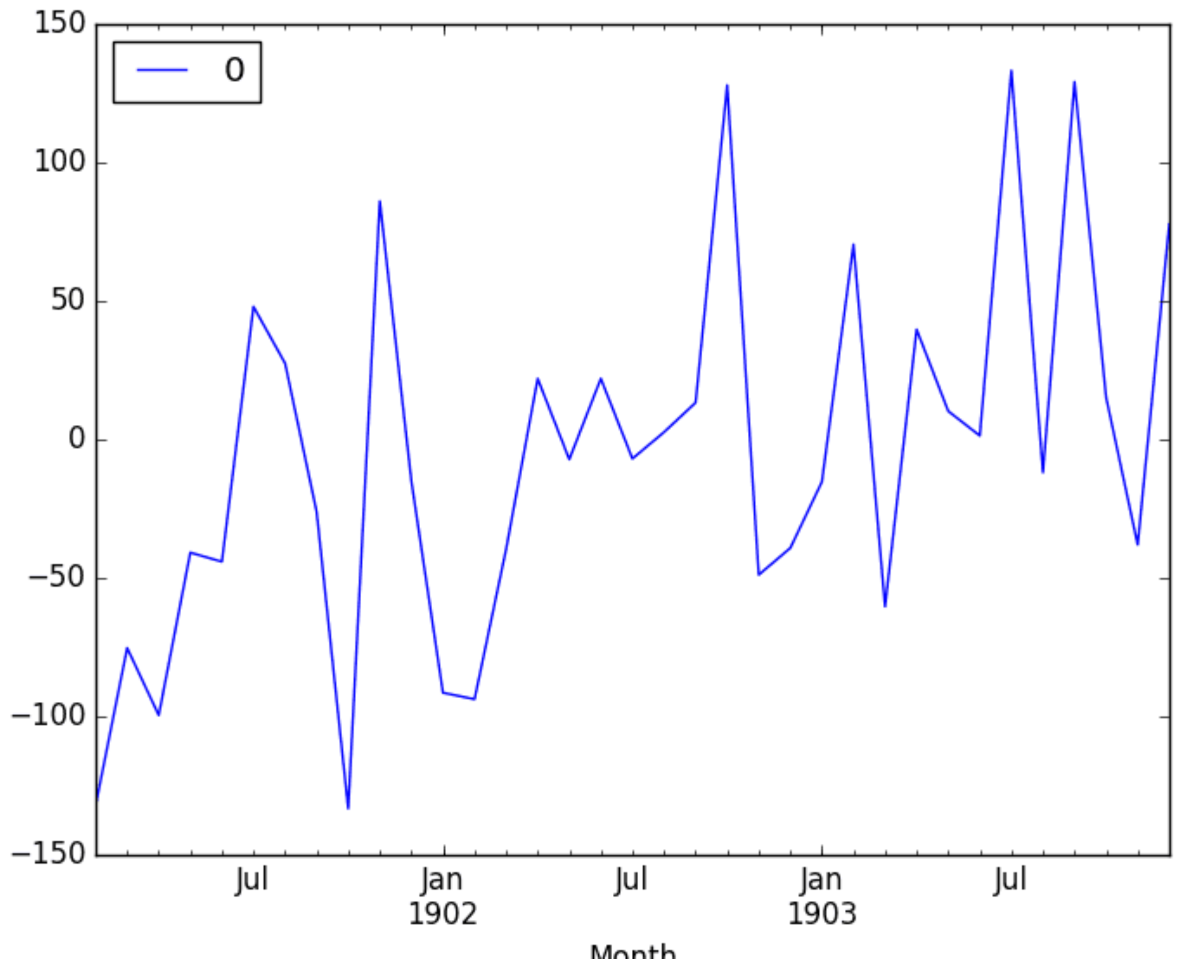
```
# fit an ARIMA model and plot residual errors
from pandas import datetime
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima.model import ARIMA
from matplotlib import pyplot
# load dataset
def parser(x):
 return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True, squeeze=True
series.index = series.index.to_period('M')
# fit model
model = ARIMA(series, order=(5,1,0))
model_fit = model.fit()
# 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())
```

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.

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  Sales   No. Observations:                   36
Model:                 ARIMA(5, 1, 0)   Log Likelihood                -198.485
Date:                Thu, 10 Dec 2020   AIC                            408.969
Time:                        09:15:01   BIC                            418.301
Sample:                    01-31-1901   HQIC                           412.191
                         - 12-31-1903
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.9014      0.247     -3.647      0.000      -1.386      -0.417
ar.L2         -0.2284      0.268     -0.851      0.395      -0.754       0.298
ar.L3          0.0747      0.291      0.256      0.798      -0.497       0.646
ar.L4          0.2519      0.340      0.742      0.458      -0.414       0.918
ar.L5          0.3344      0.210      1.593      0.111      -0.077       0.746
sigma2      4728.9608   1316.021      3.593      0.000    2149.607    7308.314
===================================================================================
Ljung-Box (L1) (Q):                   0.61   Jarque-Bera (JB):                 0.96
```

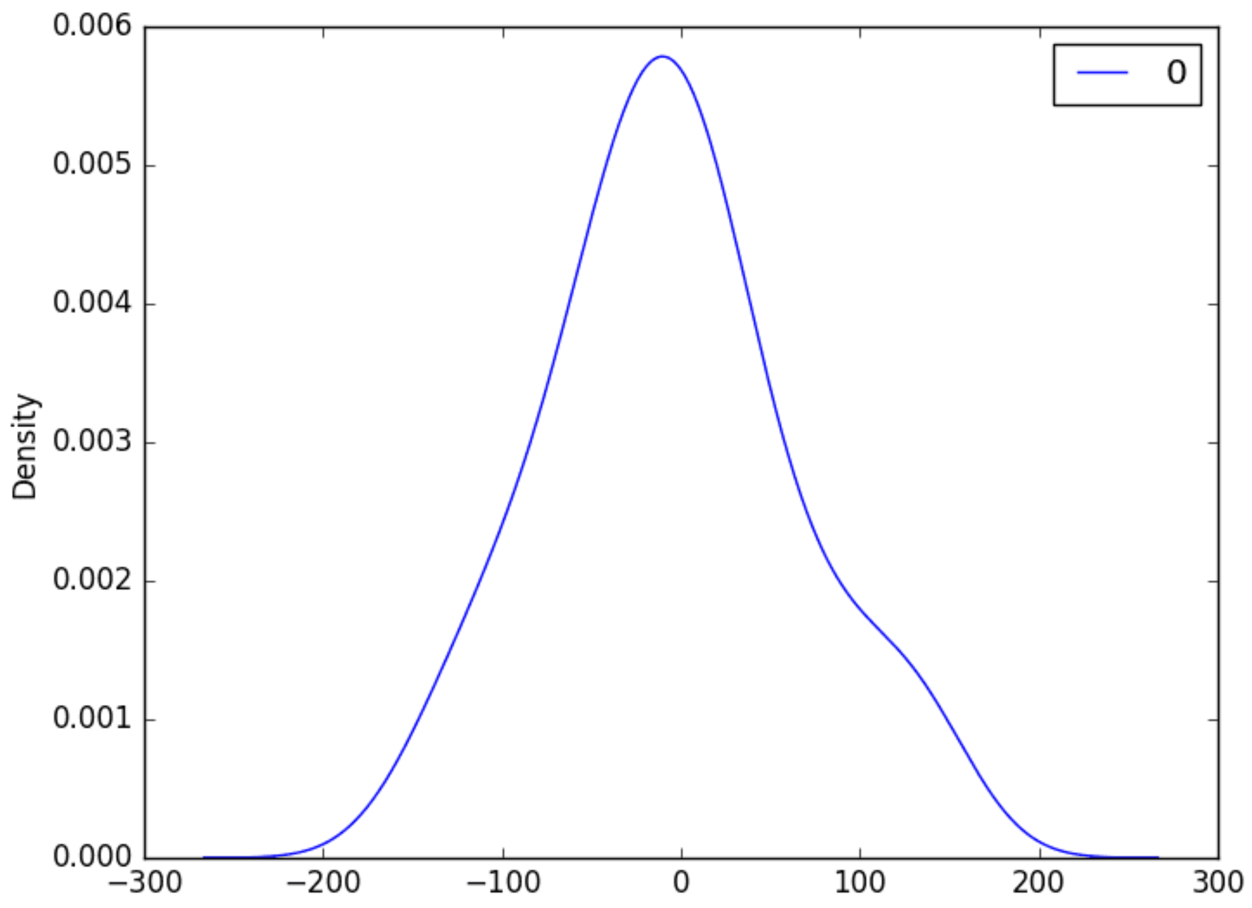
```
Prob(Q):                              0.44   Prob(JB):                         0.62
Heteroskedasticity (H):               1.07   Skew:                             0.28
Prob(H) (two-sided):                  0.90   Kurtosis:                         2.41
===================================================================================
```

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



ARMA Fit Residual Error Line Plot

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



ARMA Fit Residual Error Density Plot

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   36.000000
mean    21.936144
std     80.774430
min   -122.292030
25%    -35.040859
50%     13.147219
75%     68.848286
max    266.000000
```

Note, that although 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.

Rolling Forecast ARIMA Model

The ARIMA model can be used to forecast future time steps.

The ARIMA model is adept at forecasting future time points. In a rolling forecast, the model is often retrained as new data becomes available, allowing for more accurate and adaptive predictions.

We can use the predict() function on the ARIMAResults object 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.

How to Forecast with ARIMA:

1. Use the predict() function on the ARIMAResults object. This function requires the index of the time steps for which predictions are needed.
2. To revert any differencing and return predictions in the original scale, set the typ argument to 'levels'.
3. For a simpler one-step forecast, employ the forecast() function.

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.

```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, index_col=0, parse_dates=True, squeeze=True
series.index = series.index.to_period('M')
# 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()
 output = model_fit.forecast()
 yhat = output[0]
 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()
```

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.

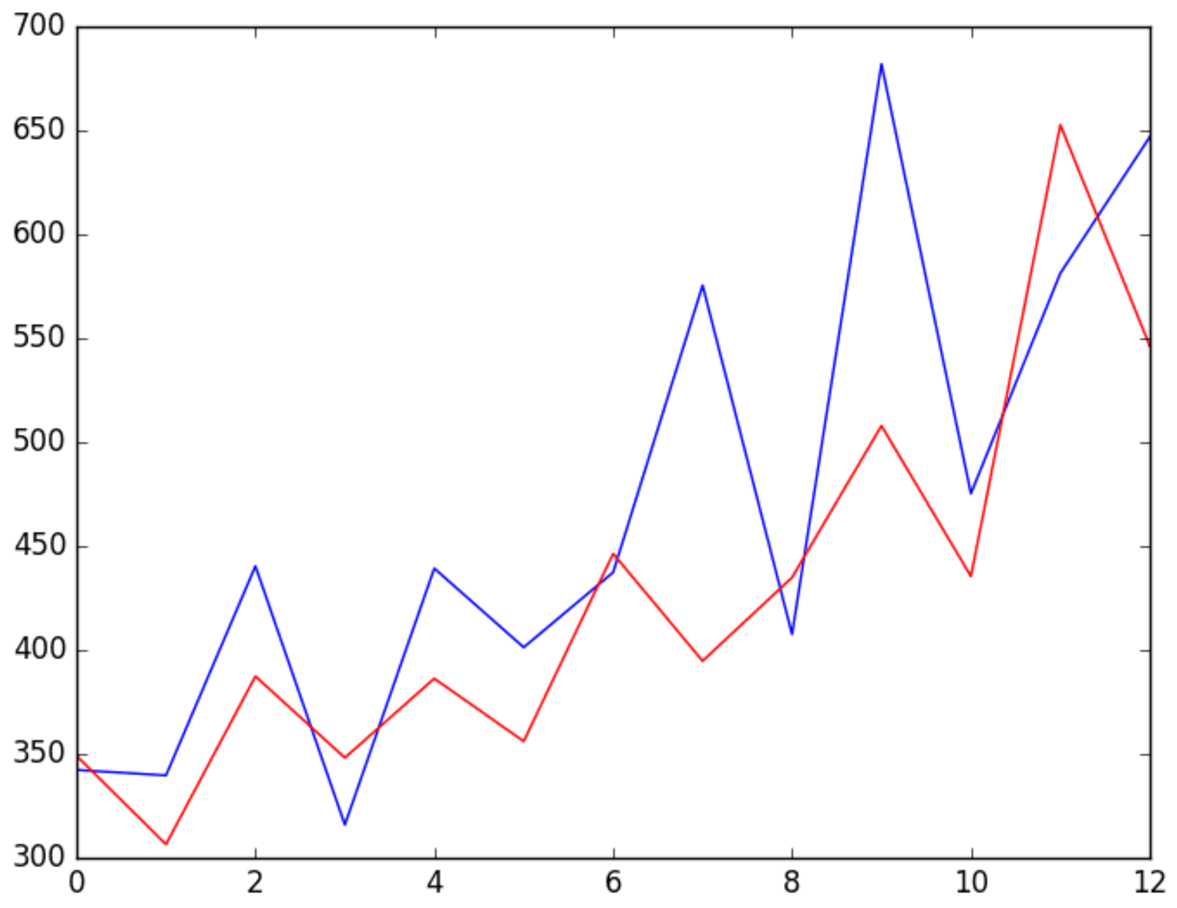
Putting this all together, below is an example of a rolling forecast with the ARIMA model in Python.

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=343.272180, expected=342.300000
predicted=293.329674, expected=339.700000
predicted=368.668956, expected=440.400000
predicted=335.044741, expected=315.900000
predicted=363.220221, expected=439.300000
predicted=357.645324, expected=401.300000
predicted=443.047835, expected=437.400000
predicted=378.365674, expected=575.500000
predicted=459.415021, expected=407.600000
predicted=526.890876, expected=682.000000
predicted=457.231275, expected=475.300000
predicted=672.914944, expected=581.300000
predicted=531.541449, expected=646.900000
Test RMSE: 89.021
```

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



ARIMA Rolling Forecast Line Plot

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

Configuring an ARIMA Model

ARIMA is often configured using the classical Box-Jenkins Methodology. This process employs a meticulous blend of time series analysis and diagnostics to pinpoint the most fitting parameters for the ARIMA model.

The Box-Jenkins Methodology: A Three-Step Process:

1. **Model Identification**: Begin with visual tools like plots and leverage summary statistics. These aids help recognize trends, seasonality, and autoregressive elements. The goal here is to gauge the extent of differencing required and to determine the optimal lag size.
2. **Parameter Estimation**: This step involves a fitting procedure tailored to derive the coefficients integral to the regression model.
3. **Model Checking**: Armed with plots and statistical tests delve into the residual errors. This analysis illuminates the temporal structure that the model might have missed.

The process is repeated until either a desirable level of fit is achieved on the in-sample or out-of-sample observations (e.g. training or test datasets).

The process was described in the classic 1970 textbook on the topic titled Time Series Analysis: Forecasting and Control by George Box and Gwilym Jenkins. An updated 5th edition is now available if you are interested in going deeper into this type of model and methodology.

Given that the model can be fit efficiently on modest-sized time series datasets, grid searching parameters of the model can be a valuable approach.

For an example of how to grid search the hyperparameters of the ARIMA model, see the tutorial:

- How to Grid Search ARIMA Model Hyperparameters with Python

Summary

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

Specifically, you learned:

- **ARIMA Model Overview**: Uncovered the foundational aspects of the ARIMA model, its configuration nuances, and the key assumptions it operates on.
- **Quick Time Series Analysis**: Explored a swift yet comprehensive analysis of time series data using the ARIMA model.
- **Out-of-Sample Forecasting with ARIMA**: Delved into harnessing the ARIMA model for making predictions beyond the sample data.

Do you have any questions about ARIMA, or about this tutorial?

Ask your questions in the comments below and I will do my best to answer.