Table of Contents

[Avoiding Too Much Differencing 1](#_Toc124447229)

[Augmented Dickey-Fuller Test (ADF) 1](#_Toc124447230)

[ARIMA for Differencing 3](#_Toc124447231)

[Grid Searching ARIMA Parameters 5](#_Toc124447232)

[Auto-ARIMA 7](#_Toc124447233)

[References 9](#_Toc124447234)

[ARIMAX Models with Exogenous Variables 9](#_Toc124447235)

[SARIMA Introduction 12](#_Toc124447236)

[Seasonal ARIMA Elements 12](#_Toc124447237)

[1. ARIMA Elements 12](#_Toc124447238)

[2. Seasonal Elements 12](#_Toc124447239)

[SARIMAX Models 16](#_Toc124447240)

[Day Ahead Predictions with Exogenous Variables 17](#_Toc124447241)

[Back-Test Simulations 19](#_Toc124447242)

[Simulations for Other Data Domains 21](#_Toc124447243)

### Avoiding Too Much Differencing

Differencing can be performed multiple times. However, it is important not to over-difference the data. If you cannot decide which differenced series to choose then choose the series with the least amount of variance.

### Augmented Dickey-Fuller Test (ADF)

The ADF test is often used to determine if a series is stationary. The null hypothesis and alternate hypothesis are shown below.

: The time series is non-stationary.

: The time series is stationary.

If the p-value of the test is less than 0.05 then you reject the null hypothesis and differencing is not needed.

Example 1: ADF Test

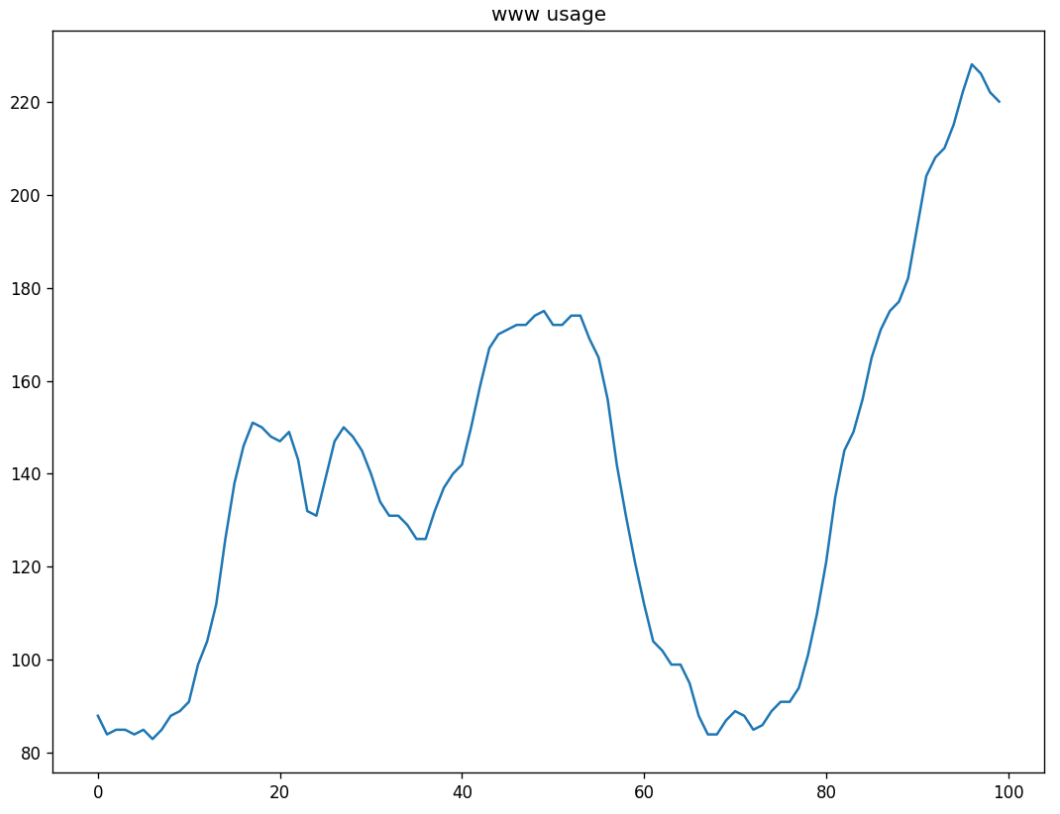
This example shows how to perform the ADF test to determine if the wwwusage data is stationary. Figure 1 shows the www usage data plotted over time. The result of the ADF test is:

ADF Statistic: -2.464240

p-value: 0.124419

The p-value is insignificant so the data is not stationary so more differencing is needed.

Figure 1: www usage data plotted over time



Here is the full example:

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  import numpy as np, pandas as pd  from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf  import matplotlib.pyplot as plt  plt.rcParams.update({'figure.figsize':(9,7), 'figure.dpi':120})  # Import data  df = pd.read\_csv(  "https://raw.githubusercontent.com/selva86/datasets/master/wwwusage.csv", \  names=['value'], header=0)  print(df)  df.value.plot()  plt.title("www usage")  plt.show()  from statsmodels.tsa.stattools import adfuller  result = adfuller(df.value.dropna())  print('ADF Statistic: %f' % result[0])  print('p-value: %f' % result[1]) |

Exercise 1 (8 marks)

Starting with Example 1, difference the data.

dfDifferenced = df.diff()

After differencing the data, plot the data. Show your new plot here.

|  |
| --- |
|  |

Then perform an ADF test. Show the p-value that is obtained by your ADF test above. Indicate if additional differencing is needed.

|  |
| --- |
|  |

Perform differencing a second time and show the new plot of the second order differenced data here.

|  |
| --- |
|  |

# ARIMA for Differencing

Today we will introduce the full ARIMA model. The I in ARIMA stands for integrated and refers to the order differencing, d, performed on the time series observations before predictions are made in the linear regression model.

Differencing is used to ensure that all patterns are removed from the data. In effect, the differencing will make the data stationary. Stationary data has constant random variance.

When making manual predictions, we must perform this differencing of the dataset prior to calling the predict() function. Below is a function that implements differencing of the entire dataset.

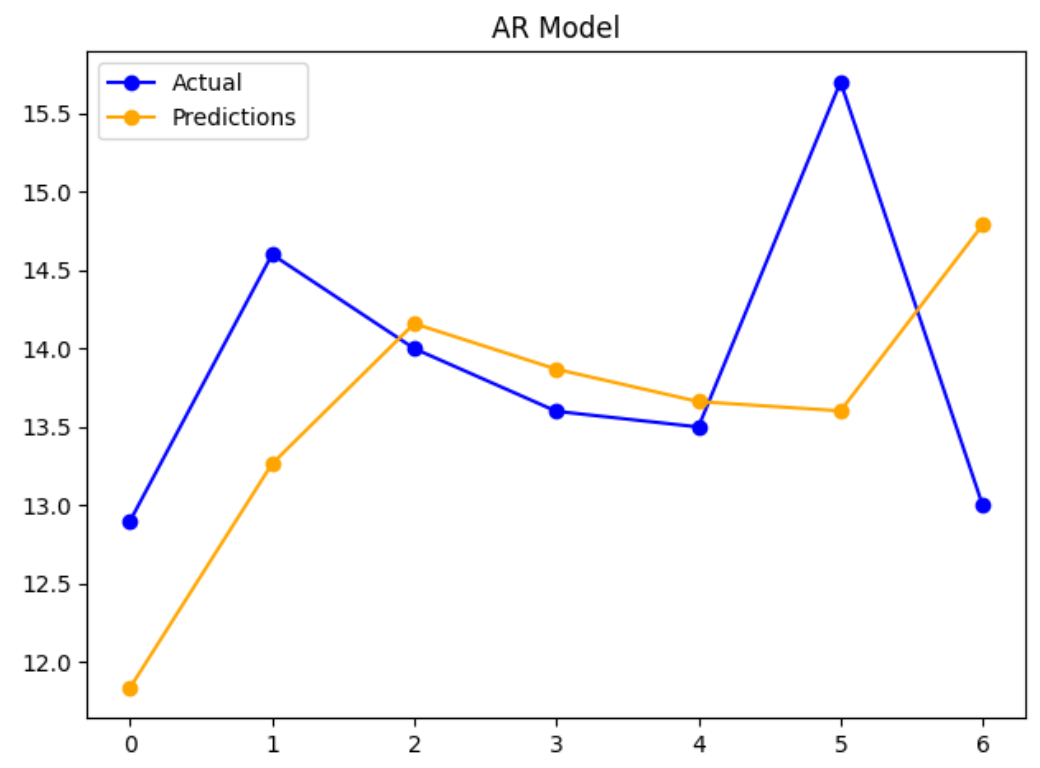
Example 2: Adding Differencing to the ARIMA model

This model implements day ahead predictions with differencing.

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  from pandas import read\_csv  import matplotlib.pyplot as plt  import statsmodels.tsa.arima.model as sma  from sklearn.metrics import mean\_squared\_error  from math import sqrt  import numpy as np  PATH = "/Users/pm/Desktop/DayDocs/data/"  series = read\_csv(PATH + 'daily-min-temperatures.csv',  header=0, index\_col=0)  # Split the data set so the test set is 7.  NUM\_TEST\_DAYS = 7  X = series.values  size = len(X) - NUM\_TEST\_DAYS  train, test = X[0:size], X[size:]  def difference(dataset):  diff = list()  for i in range(1, len(dataset)):  value = dataset[i] - dataset[i - 1]  diff.append(value)  return np.array(diff)  # Create a list with the training array.  history = [x for x in train]  predictions = []  # predict() receives the model coefficients and all past data (t-1, t-2, t-2) etc.  def predict(coef, history):  yhat = 0.0  for i in range(1, len(coef) + 1):  # Make the prediction (yhat)  # This multiplies L1coeff\*L1  # and L2coeff\*L2 if it exists  # and L3coeff\*L3 if it exists  yhat += coef[i - 1] \* history[-i]  return yhat # Return the prediction.  for t in range(len(test)):  print("History length: " + str(len(history)))  #################################################################  # Model building and prediction section.  model = sma.ARIMA(history, order=(1, 1, 1)).fit()  ar\_coef, ma\_coef = model.arparams, model.maparams  resid = model.resid # Error (difference between actual and predicted)  diff = difference(history) # Differenced data.  yhat = history[-1] + predict(ar\_coef, diff) + predict(ma\_coef, resid)  #################################################################  predictions.append(yhat) # Store the prediction in a list.  obs = test[t] # Get the actual current value.  history.append(obs) # Append the actual current value to the training data.  # Actual values will be used as t-1, t-2 etc next iteration.  print('>predicted=%.3f, expected=%.3f' % (yhat, obs))  rmse = sqrt(mean\_squared\_error(test, predictions))  print('Test RMSE: %.3f' % rmse)  plt.plot(test, label='Actual', marker='o', color='blue')  plt.plot(predictions, label='Predictions', marker='o', color='orange')  plt.legend()  plt.title("AR Model")  plt.show() |

The RMSE of 1.232 is the best fitting yet compared to the ARMA model (RMSE=1.4050) and the other models which perform more poorly.

RMSE: 1.232



## Grid Searching ARIMA Parameters

Example 3: Grid searching ARIMA p,d,q Parameters

It is possible to optimize ARIMA parameters through a grid search. This article discusses a novel way to iterate through all possible ARIMA combinations with different values for p, d and q. For more information see:

<https://machinelearningmastery.com/grid-search-arima-hyperparameters-with-python/>

You may want to try this routine but you are not expected to run it. This grid search is painfully slow. If you do run it though you will notice that it prints out an MSE rating for each of the combinations.

The first leading model candidate is ARIMA(0,1,2). The parameters for the first model candidate are p=0, d=1 and q=2. In other words, p=0 implies there is no (AR) autoregressive component. The parameter d=1 implies first order differencing is required. q=2 implies that two lags of the (MA) moving average which are used. Remember that the moving average component is referring to residual error in this case and is not to be confused with weighted moving average.

ARIMA(0, 0, 0) MSE=15.818

ARIMA(0, 0, 1) MSE=8.882

ARIMA(0, 0, 2) MSE=7.471

ARIMA(0, 1, 0) MSE=6.829

ARIMA(0, 1, 1) MSE=6.325

ARIMA(0, 1, 2) MSE=5.461

ARIMA(0, 2, 0) MSE=16.157

ARIMA(0, 2, 1) MSE=6.835

ARIMA(0, 2, 2) MSE=6.331

ARIMA(1, 0, 0) MSE=6.091

ARIMA(1, 0, 1) MSE=6.016

ARIMA(1, 0, 2) MSE=5.434

ARIMA(1, 1, 0) MSE=6.604

You are not expected to run this code but in case you want or need to here is the code:

|  |
| --- |
| from pandas import read\_csv  import matplotlib.pyplot as plt  from statsmodels.tsa.arima\_model import ARIMA  from sklearn.metrics import mean\_squared\_error  from math import sqrt  import warnings  warnings.filterwarnings("ignore")  PATH = "/Users/pm/Desktop/DayDocs/data/"  series = read\_csv(PATH + 'daily-min-temperatures.csv', header=0, index\_col=0)  # Evaluate an ARIMA model for a given order (p,d,q).  def evaluate\_arima\_model(X, arima\_order):  # Prepare training dataset.  train\_size = int(len(X) \* 0.66)  train, test = X[0:train\_size], X[train\_size:]  history = [x for x in train]    # Make predictions.  predictions = list()    for t in range(len(test)):  model = ARIMA(history, order=arima\_order)  model\_fit = model.fit(disp=0)  yhat = model\_fit.forecast()[0]  predictions.append(yhat)  history.append(test[t])    # Calculate out of sample error,  error = mean\_squared\_error(test, predictions)  return error  # Evaluate combinations of p, d and q values for an ARIMA model.  def evaluate\_models(dataset, p\_values, d\_values, q\_values):  dataset = dataset.astype('float32')  best\_score, best\_cfg = float("inf"), None  for p in p\_values:  for d in d\_values:  for q in q\_values:  order = (p, d, q)  try:  mse = evaluate\_arima\_model(dataset, order)  if mse < best\_score:  best\_score, best\_cfg = mse, order  print('ARIMA%s MSE=%.3f' % (order, mse))  except:  continue  print('Best ARIMA%s MSE=%.3f' % (best\_cfg, best\_score))  # Set parameter ranges.  p\_values = [0, 1, 2, 4, 6, 8, 10]  d\_values = range(0, 3)  q\_values = range(0, 3)  # Evaluate performance.  evaluate\_models(series.values, p\_values, d\_values, q\_values) |

## Auto-ARIMA

An auto\_arima() function can find an optimal p,d,q combination more quickly than the grid search in Example 3. We will look more at auto\_arima() during the lesson that follows this lab. For now, here is an example to get started with the function.

Example 4: Optimizing p,d,q parameters with auto\_arima()

This example shows how to automate the process to find a reasonably optimal solution for making predictions with the minimum daily temperatures data set. This example searches for the optimal p, d, q combination at each iteration through the loop with the most recent observations from the data set. Starting with Example 2, add a reference to the required package;

|  |
| --- |
| import pmdarima as pm |

Note: If you have trouble with pmdarima uninstall and reinstall with pip.

Note:

If you receive this error;

ERROR: Could not install packages due to an OSError: [WinError 5] Access is denied:

On windows try a pip install when running PyCharm as administrator.

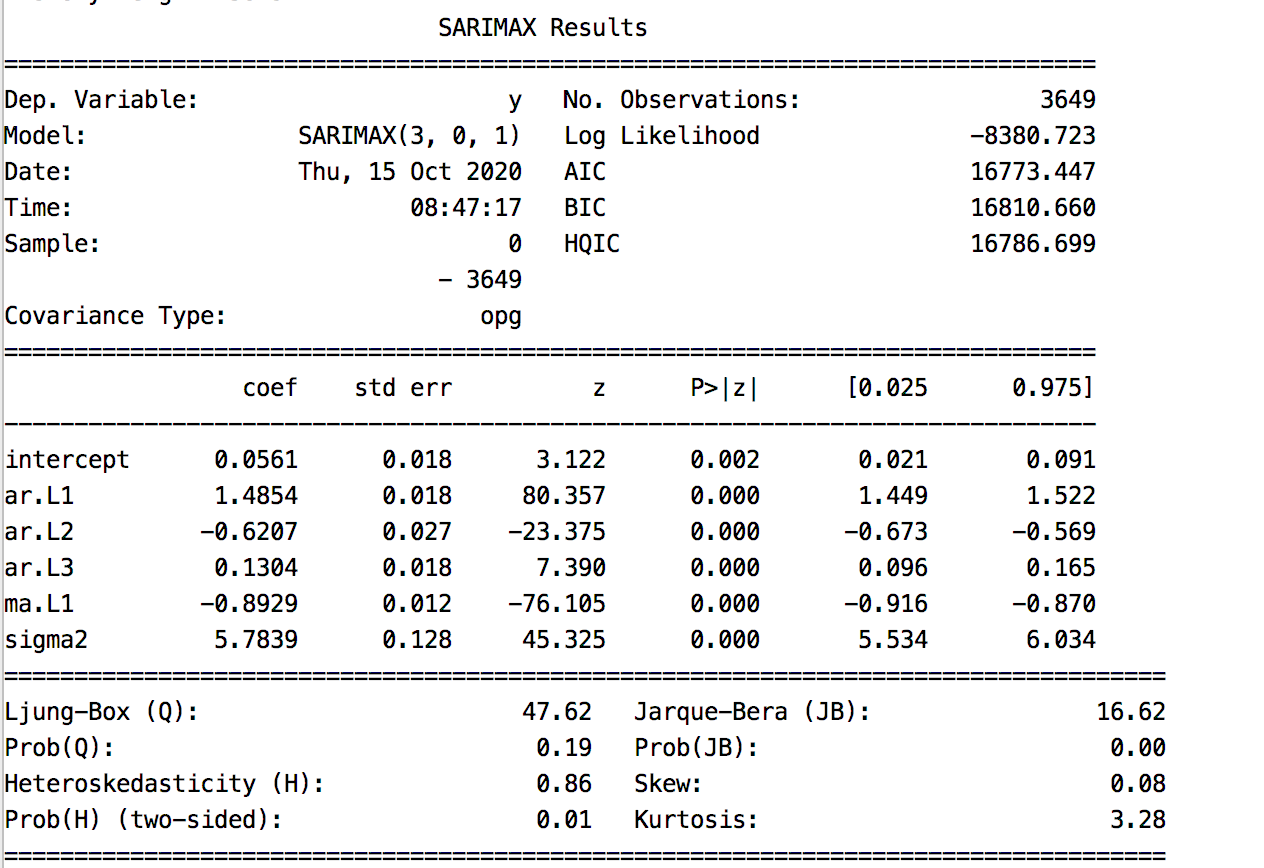
OR try running the install and example 8 in Spyder.

Next, swap out the model building and prediction section with this version.

|  |
| --- |
| #################################################################  # Model building and prediction section.  # Build day ahead model.  model = pm.auto\_arima(history, start\_p=1, start\_q=1,  test='adf',  max\_p=3, max\_q=3, m=0,  start\_P=0, seasonal=False,  error\_action='ignore',  suppress\_warnings=True,  stepwise=True)  print(model.summary())  fc, confint = model.predict(n\_periods=1,  return\_conf\_int=True)  yhat = fc[0]  ################################################################# |

The results are decent. The RMSE is 1.246. This is very good performance when compared to all other models but ARIMA(p=1, d=1,q=1) model was able to achieve an RMSE of 1.232.

Model summaries are displayed during each iteration. The routine happened to choose an ARIMA(p=3, d=0, q=1) model at each iteration. All coefficients shown in these models are statistically significant.



To practice applying the material discussed in this document, this next set of exercises will use the **daily-total-female-births.csv** file as a data source.

Exercise 2 (1 mark)

After loading the **daily-total-female-births.csv** data source into a data frame, print the data frame with the head() instruction to see what is in it. Show your contents here:

|  |
| --- |
|  |

Exercise 3 (1 mark)

Print the data frame with the describe() function to observe the mean, standard deviation and percentiles. Show the output here:

|  |
| --- |
|  |

Exercise 4 (4 marks)

Using the auto\_arima() function while looping through the test data, find the best fitting combination of p, d and q lag values. Show a screenshot of the model summary for the last combination here:

|  |
| --- |
|  |

Show the RMSE here:

|  |
| --- |
|  |

Exercise 5 (1 mark)

For model summary in the last iteration, what is the optimal p, d, q combination for the ARIMA model given 7 test days? Are the coefficients statistically significant?

|  |
| --- |
|  |

## References

<https://machinelearningmastery.com/sarima-for-time-series-forecasting-in-python/>

<https://machinelearningmastery.com/how-to-grid-search-sarima-model-hyperparameters-for-time-series-forecasting-in-python/>

<https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/>

<https://alkaline-ml.com/pmdarima/tips_and_tricks.html>

# ARIMAX Models with Exogenous Variables

ARIMAX models are ARIMA models which contain exogenous variables. An **exogenous variable** is an alternate feature and is different than the lagged feature.

Example 5: Temp Preditction with Exogenous

This example uses a lagged closing price as an exogenous variable to make predictions for opening stock prices. When running this example, the RMSE is 5.734. The exogenous variable appears to detract from the results when compared to an RMSE of 3.288 when no exogenous variable is used. Remove the highlighted code to remove the exogenous variable. Still, hopefully you recognize the potential for being able to supply additional exogenous variables with your ARIMA models. This article shows an example where an exogenous variable does make an improvement <https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/>

|  |  |
| --- | --- |
| With exogenous variable.  RMSE: 5.734 | Without exogenous variable.  RMSE: 3.288 |

Here is the code:

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  import numpy as np  import matplotlib.pyplot as plt  import pandas as pd  from pandas\_datareader import data as pdr  import yfinance as yfin # Work around until  # pandas\_datareader is fixed.  import datetime  import pmdarima as pm  def getStock(stk, ttlDays):  numDays = int(ttlDays)  # Only gets up until day before during  # trading hours  dt = datetime.date.today()  # For some reason, must add 1 day to get current stock prices  # during trade hours. (Prices are about 15 min behind actual prices.)  dtNow = dt + datetime.timedelta(days=1)  dtNowStr = dtNow.strftime("%Y-%m-%d")  dtPast = dt + datetime.timedelta(days=-numDays)  dtPastStr = dtPast.strftime("%Y-%m-%d")  yfin.pdr\_override()  df = pdr.get\_data\_yahoo(stk, start=dtPastStr, end=dtNowStr)  return df  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line.  pd.set\_option('display.width', 1000)  dfStock = getStock('AAPL', 1200)  print(dfStock)  TOTAL\_DAYS = 5  # Build feature set with backshifted closing prices.  dfStock['Close\_t\_1'] = dfStock['Close'].shift(1)  dfStock = dfStock.dropna()  dfX = dfStock[['Open', 'Close\_t\_1']]  size = len(dfX) - TOTAL\_DAYS  train, test = dfX[0:size], dfX[size:]  # Create training set and copy of the training set.  train.tail(TOTAL\_DAYS)  history = train.copy()  predictions = []  # Iterate to make predictions for the evaluation set.  for i in range(0, len(test)):  lenOpen = len(history[['Close\_t\_1']])  print("\n\nModel " + str(i))  print(history.shape)  model = pm.auto\_arima(history[['Open']],  exogenous=history[['Close\_t\_1']],  start\_p=1, start\_q=1,  test='adf', # Use adftest to find optimal 'd'  max\_p=3, max\_q=3, # Set maximum p and q.  d=None, # Let model determine 'd'.  trace=True,  error\_action='ignore',  suppress\_warnings=True)  fc, confint = model.predict(n\_periods=1,  exogenous=np.array(  history.iloc[lenOpen-1]['Close\_t\_1']).reshape(1,-1),  return\_conf\_int=True)  predictions.append(fc)  open = test.iloc[i]['Open']  close\_t\_1 = test.iloc[i]['Close\_t\_1']  history = history.append({"Open":open, "Close\_t\_1":close\_t\_1},  ignore\_index=True)  plt.plot(test.index, test['Open'], marker='o',  label='Actual', color='blue')  plt.plot(test.index, predictions, marker='o',  label='Predicted', color='orange')  plt.legend()  plt.xticks(rotation=70)  plt.show()  from sklearn.metrics import mean\_squared\_error  from math import sqrt  rmse = sqrt(mean\_squared\_error(test['Open'], predictions))  print('Test RMSE: %.3f' % rmse) |

SARIMA Introduction  
Seasonal Autoregressive Integrated Moving Average, SARIMA or Seasonal ARIMA, is an extension of ARIMA model that explicitly supports univariate time series data with a seasonal component.

SARIMA models add three new hyperparameters to specify the autoregression (AR), differencing (I) and moving average (MA) for the seasonal component of the series. An additional parameter is also used to specify the period of the seasonality.

## Seasonal ARIMA Elements

The SARIMA model contains two sets of elements.

### ARIMA Elements

The same ARIMA model elements are included as before in the first set:

p: Trend autoregression order.

d: Trend difference order.

q: Trend moving average order.

### Seasonal Elements

In the second set, there are four seasonal elements that are not part of ARIMA model. These capitalized parameters include;

P: Seasonal autoregressive order.

D: Seasonal difference order.

Q: Seasonal moving average order.

m: The number of time steps for a single seasonal period.

Together, the notation for an SARIMA model is specified as:

SARIMA(p,d,q)(P,D,Q)m

For example, here is a model summary for a monthly model:

SARIMA(3,1,0)(1,1,0)12

Time periods for ***m*** include:

1 – daily

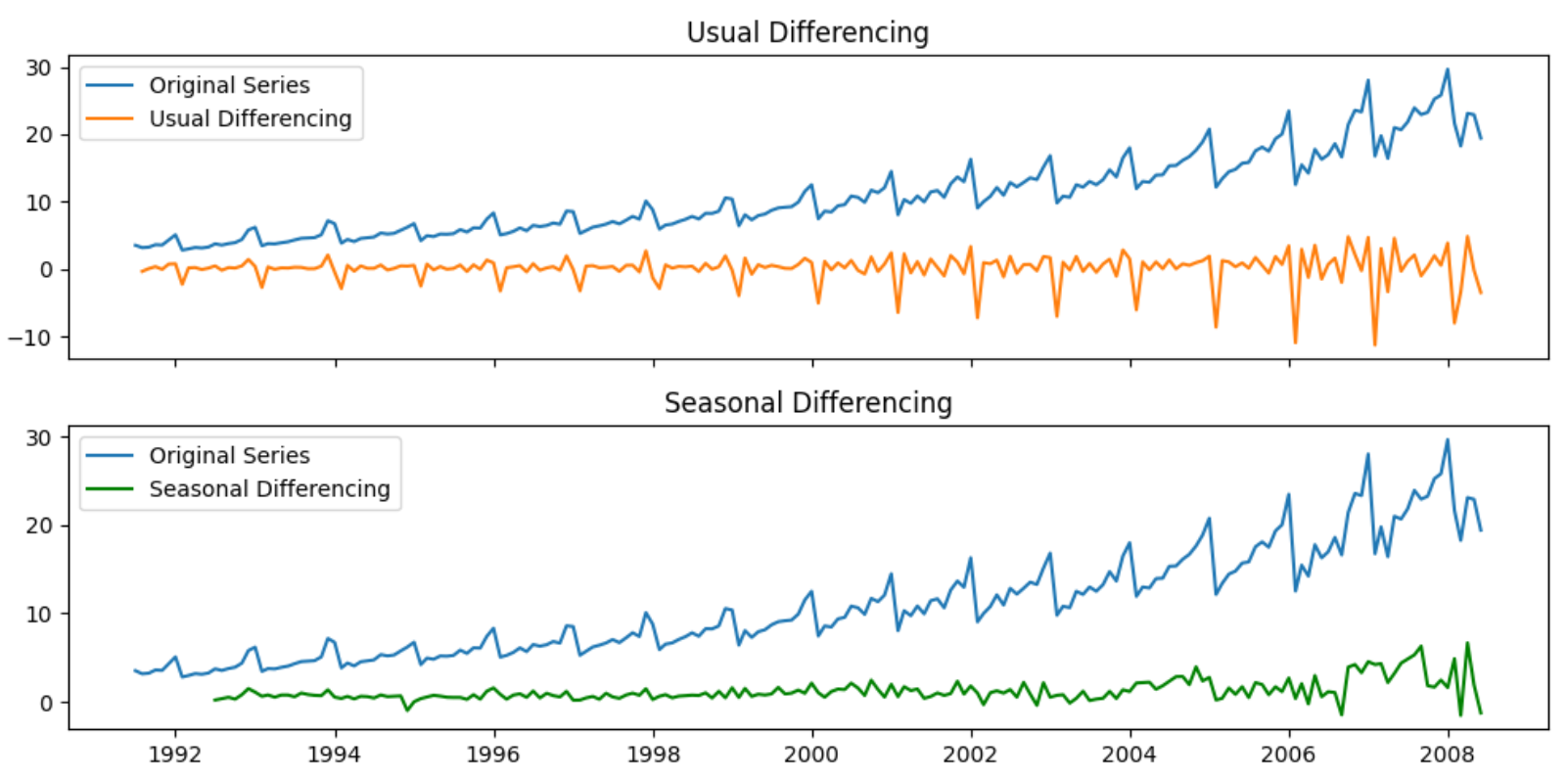
12 – monthly

4 – quarterly

52 – weekly

Example 6: Observing Seasonal Differencing

This example visually explores differences between ways to treat seasonal patterns. As you can see, the seasonal spikes remain intact after applying usual differencing (lag 1). Seasonal differencing in the plot at the bottom shows that the pattern is removed.



Here is the code which loads the drugSales.csv, performs two separate differencing techniques and visualizes the outcome. The data in the graph at the top is differenced with the instruction:

data[:].diff(1)

The data in the graph at the bottom is differenced with the instruction:

data[:].diff(12)

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  import pandas as pd  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE\_NAME = "drugSales.csv"  import matplotlib.pyplot as plt  # Import  data = pd.read\_csv(PATH + FILE\_NAME, parse\_dates=['date'], index\_col='date')  # Plot  fig, axes = plt.subplots(2, 1, figsize=(10,5), dpi=100, sharex=True)  # Usual Differencing  axes[0].plot(data[:], label='Original Series')  axes[0].plot(data[:].diff(1), label='Usual Differencing')  axes[0].set\_title('Usual Differencing')  axes[0].legend(loc='upper left', fontsize=10)  # Seasonal Differencing  axes[1].plot(data[:], label='Original Series')  axes[1].plot(data[:].diff(12), label='Seasonal Differencing', color='green')  axes[1].set\_title('Seasonal Differencing')  plt.legend(loc='upper left', fontsize=10)  plt.show() |

Example 7: Build SARIMA Model

In this example, we recognize the monthly seasonal effect on our data so we will build a SARIMA model. The best model chosen appears to be SARIMAX(3, 0, 0)x(0, 1, 1, 12). This model has an AIC of 516.995 and all p-values are significant.

|  |
| --- |
| Fit ARIMA(2,0,2)x(1,1,2,12) [intercept=True]; AIC=525.207, BIC=554.525, Time=4.698 seconds  Fit ARIMA(1,0,0)x(1,1,2,12) [intercept=True]; AIC=593.876, BIC=613.421, Time=1.616 seconds  Fit ARIMA(1,0,2)x(1,1,2,12) [intercept=True]; AIC=524.213, BIC=550.273, Time=4.700 seconds  Fit ARIMA(3,0,0)x(1,1,2,12) [intercept=True]; AIC=516.995, BIC=543.055, Time=3.125 seconds  Near non-invertible roots for order (3, 0, 0)(1, 1, 2, 12); setting score to inf (at least one inverse root too close to the border of the unit circle: 0.993)  Fit ARIMA(3,0,2)x(1,1,2,12) [intercept=True]; AIC=518.565, BIC=551.140, Time=4.788 seconds  Near non-invertible roots for order (3, 0, 2)(1, 1, 2, 12); setting score to inf (at least one inverse root too close to the border of the unit circle: 0.990)  Total fit time: 48.123 seconds  SARIMAX Results  ===============================================================================================  Dep. Variable: y No. Observations: 204  Model: SARIMAX(3, 0, 0)x(1, 1, [1, 2], 12) Log Likelihood -250.498  Date: Wed, 14 Oct 2020 AIC 516.995  Time: 20:52:50 BIC 543.055  Sample: 0 HQIC 527.550  - 204  Covariance Type: opg  ==============================================================================  coef std err z P>|z| [0.025 0.975]  ------------------------------------------------------------------------------  intercept 0.0205 0.027 0.748 0.454 -0.033 0.074  ar.L1 0.0397 0.048 0.825 0.409 -0.055 0.134  ar.L2 0.4330 0.039 11.152 0.000 0.357 0.509  ar.L3 0.4347 0.056 7.782 0.000 0.325 0.544  ar.S.L12 0.8393 0.121 6.965 0.000 0.603 1.075  ma.S.L12 -1.6430 0.160 -10.285 0.000 -1.956 -1.330  ma.S.L24 0.8395 0.126 6.646 0.000 0.592 1.087  sigma2 0.6953 0.081 8.580 0.000 0.536 0.854  ===================================================================================  Ljung-Box (Q): 63.30 Jarque-Bera (JB): 118.33  Prob(Q): 0.01 Prob(JB): 0.00  Heteroskedasticity (H): 13.06 Skew: 0.44  Prob(H) (two-sided): 0.00 Kurtosis: 6.74  ===================================================================================  Warnings:  [1] Covariance matrix calculated using the outer product of gradients (complex-step) |

Note that the following parameters are set when initializing the auto\_arima() search for the seasonal model:

start\_P=0, # Set the starting seasonal autoregessive value to 0

seasonal=True, # Allow searching for the seasonal component parameters

d=None, # Let the routine find the best difference for ARIMA

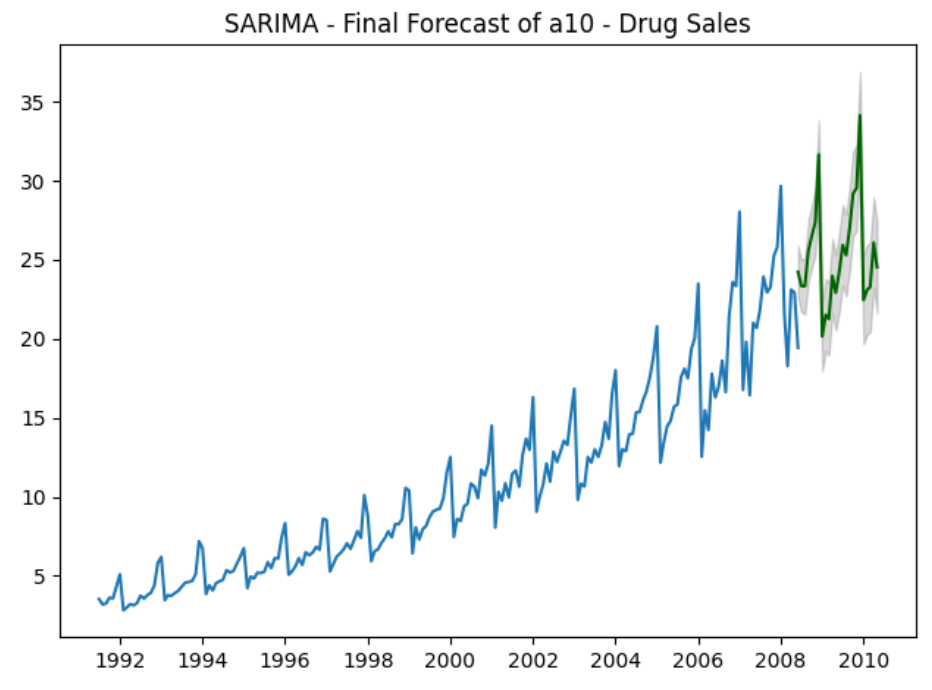
D=12, # Set the starting seasonal differencing to D=1 by default

Here is the code which uses the auto\_arima() function to find the model.

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  import pandas as pd  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE\_NAME = "drugSales.csv"  import matplotlib.pyplot as plt  # Import  data = pd.read\_csv(PATH + FILE\_NAME, parse\_dates=['date'], index\_col='date')  import pmdarima as pm  # Seasonal - fit stepwise auto-ARIMA  smodel = pm.auto\_arima(data, start\_p=1, start\_q=1,  test='adf',  max\_p=3, max\_q=3, m=12,  start\_P=0, seasonal=True,  d=None, D=1, trace=True,  error\_action='ignore',  suppress\_warnings=True,  stepwise=True)  print(smodel.summary()) |

Example 8: Making Predictions and Plotting the Confidence Interval

This example shows how to use the SARIMAX model to make predictions over 24 months. Confidence intervals were extracted from the predictions as well.



|  |
| --- |
| # Forecast  NUM\_TIMESTEPS = 24  fitted, confint = smodel.predict(n\_periods=NUM\_TIMESTEPS, return\_conf\_int=True)  index\_of\_fc = pd.date\_range(data.index[-1], periods = n\_periods, freq='MS')  # make series for plotting purpose  fitted\_series = pd.Series(fitted, index=index\_of\_fc)  lower\_series = pd.Series(confint[:, 0], index=index\_of\_fc)  upper\_series = pd.Series(confint[:, 1], index=index\_of\_fc)  # Plot  plt.plot(data)  plt.plot(fitted\_series, color='darkgreen')  plt.fill\_between(lower\_series.index,  lower\_series,  upper\_series,  color='k', alpha=.15)  plt.title("SARIMA - Final Forecast of a10 - Drug Sales")  plt.show() |

# SARIMAX Models

Similar to the ARIMAX model, you can also include an exogenous variable array to supplement the SARIMAX model. An example of how to build the SARIMAX model can be found at <https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/>

Note there is an error in the code in the SARIMAX example at the site:

exogenous=np.tile(seasonal\_index.value, 2).reshape(-1,1),

Should really be like this:

exogenous=np.tile(seasonal\_index.seasonal, 2).reshape(-1,1),

Exercise 6 (3 marks)

Run the following code to load the energy production data set.

|  |
| --- |
| import pandas as pd  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE\_NAME = "Energy\_Production.csv"  import matplotlib.pyplot as plt  # Import  data = pd.read\_csv(PATH + FILE\_NAME, index\_col=0)  data.index = pd.to\_datetime(data.index) |

Show the first few rows of data frame here:

|  |
| --- |
|  |

Use the describe() function on the data frame and show the results here:

|  |
| --- |
|  |

Exercise 7 (4 marks)

Using code like the code in Example 6 to visualize the effects of seasonal differencing. Use a monthly period of 12. Show the plots here:

|  |
| --- |
|  |

Exercise 8 (4 marks)

Run the auto\_arima() function for the Energy\_Production.csv data set to discover the optimal SARIMA p,d,q values. You can assume m=12. Show your ARIMA p,d,q values and your seasonal component p,d,q parameters.

**Note**: If you do receive a memory error you can truncate the data set to   
a smaller size before implementing the auto\_arima.  
  
This code truncates the data set to 300 rows.  
#data = data[0: 300]

|  |
| --- |
|  |

Exercise 9 (4 marks)

Plot the predictions for 24 periods and actual values together on a graph. Show your results here:

|  |
| --- |
|  |

Day Ahead Predictions with Exogenous Variables

*Example 9: Day Ahead Prediction with Two Exogenous Features*

This code is making day-ahead predictions for the opening price of stock. The model uses the closing price (**Close\_t\_1**) and volume (**Volume\_t\_1**) from the day before to predict the (**Open**) price.

Currently, the ways that we could evaluate the model are limited. In this case, an RMSE for the past 15 days is calculated. However, performing an evaluation with such a small amount of data is not a robust approach.

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  import numpy as np  import matplotlib.pyplot as plt  import pandas as pd  from pandas\_datareader import data as pdr  import yfinance as yfin # Work around until  # pandas\_datareader is fixed.  import datetime  import pmdarima as pm  def getStock(stk, ttlDays):  numDays = int(ttlDays)  # Only gets up until day before during  # trading hours  dt = datetime.date.today()  # For some reason, must add 1 day to get current stock prices  # during trade hours. (Prices are about 15 min behind actual prices.)  dtNow = dt + datetime.timedelta(days=1)  dtNowStr = dtNow.strftime("%Y-%m-%d")  dtPast = dt + datetime.timedelta(days=-numDays)  dtPastStr = dtPast.strftime("%Y-%m-%d")  yfin.pdr\_override()  df = pdr.get\_data\_yahoo(stk, start=dtPastStr, end=dtNowStr)  return df  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line.  pd.set\_option('display.width', 1000)  dfStock = getStock('AAPL', 1200)  print(dfStock)  TOTAL\_DAYS = 15  # Build feature set with backshifted closing prices.  dfStock['Close\_t\_1'] = dfStock['Close'].shift(1)  dfStock['Volume\_t\_1']= dfStock['Volume'].shift(1)  dfStock = dfStock.dropna()  dataDf = dfStock[['Open', 'Close\_t\_1', 'Volume\_t\_1']]  splitRowNum = len(dataDf) - TOTAL\_DAYS  trainDf, testDf = dataDf[0:splitRowNum], dataDf[splitRowNum:]  # Create training set and copy of the training set.  trainDf.tail(TOTAL\_DAYS)  history = trainDf.copy()  predictions = []  FEATURE\_LIST = ['Close\_t\_1', 'Volume\_t\_1']  # Iterate to make predictions for the evaluation set.  for i in range(0, len(testDf)):  # Find the best model with most recent features.  model = pm.auto\_arima(history[['Open']],  exogenous=history[FEATURE\_LIST],  start\_p=1, start\_q=1,  test='adf', # Use adftest to find optimal 'd'  max\_p=3, max\_q=3, # Set maximum p and q.  d=None, # Let model determine 'd'.  seasonal=False, # No Seasonality.  trace=True,  error\_action='ignore',  suppress\_warnings=True,  stepwise=True)  lenHistory = len(history)  subDf = history[FEATURE\_LIST]  featureArray = np.array(subDf)  lastFeatureRow = featureArray[lenHistory - 1]  # Make prediction with most recent data prior to prediction.  pred, confint = model.predict(n\_periods=1,  exogenous=lastFeatureRow.reshape(1,-1),  return\_conf\_int=True)  predictions.append(pred)  # Extract next row in the test set.  open = testDf.iloc[i]['Open'] # Open price morning.  close\_t\_1 = testDf.iloc[i]['Close\_t\_1'] # Close price day before.  volume\_t\_1 = testDf.iloc[i]['Volume\_t\_1'] # Volume day before.  # Add most recently available training data.  history = history.append({  "Open":open,  "Close\_t\_1":close\_t\_1,  "Volume\_t\_1":volume\_t\_1},  ignore\_index=True)  plt.plot(testDf.index, testDf['Open'], marker='o',  label='Actual', color='blue')  plt.plot(testDf.index, predictions, marker='o',  label='Predicted', color='orange')  plt.legend()  plt.xticks(rotation=70)  plt.show()  from sklearn.metrics import mean\_squared\_error  from math import sqrt  rmse = sqrt(mean\_squared\_error(testDf['Open'], predictions))  print('testDf RMSE: %.3f' % rmse) |

*Exercise 10 (Not for marks)*

Why would it not be appropriate to include Open t-1, Open t-2 or other lags for the Open price in the exogenous variable?

|  |
| --- |
|  |

*Exercise 11 (Not for marks)*

Can other prices and volumes for other stocks or trade indexes be included as exogenous variable features?

|  |
| --- |
|  |

Back-Test Simulations

Since the ability to obtain current data is limited, we can develop a back-testing simulation to help evaluate our model. Back-testing allows us to evaluate our model’s ability to achieve specific goals with a larger data set. However, be cautious with back-test simulations since external factors outside the model which affect the outcome will also change over time also.

*Example 10: Back-Testing Simulation*

The strategy being back-tested in this example involves buying stock at the end of the day and selling at the next open when a higher price is expected at the opening. To begin, start with the code for Example 9.

The simulation tries to predict the gains over time. To observe how the routine would have performed during the past 45 days, change the duration of the back-test by setting TOTAL\_DAYS to 45.

|  |
| --- |
| TOTAL\_DAYS = 45 |

Next, place the following code before the loop in Example 9. The code starts with a balance of $20,000. The function implements a routine to purchase stock at the close of the day, sell the stock at the next opening and update the cash balance.

|  |
| --- |
| cashBalance = 20000  def udpateCashBalance(cashBalance, purchasePrice, sellPrice):  qty = int(cashBalance / purchasePrice)  cost = purchasePrice \* qty  cashBalance -= cost  revenue = sellPrice \* qty  cashBalance += revenue  return cashBalance |

Next, at the end of the loop, add this code to update the cash balance whenever the opening price is predicted to be higher than the closing price.

|  |
| --- |
| if(pred > close\_t\_1):  cashBalance = udpateCashBalance(cashBalance, close\_t\_1, open) |

At the end of the time series, show the final cash balance.

|  |
| --- |
| print("The final cash balance is: " + str(cashBalance)) |

After running the simulation, the output appears as follows (but without the formatting):

The final cash balance is: **$20,353.72**

If you ran the trial again without including volume (at the time this document was written) you would notice that the final cash balance would be the same. If you ran the trial again and included the close\_t\_2 within the exogenous feature set you would notice the final cash balance would be:

**$20,360.16**

Here are some other use cases that you try (if you had time):

* Set the TOTAL\_DAYS to include one full year.
* Include in stock prices from a stock from the same sector within the exogenous variable feature set.

Simulations for Other Data Domains

With stocks the back-test would often evaluate the success of a trade strategy. Other knowledge domains may demand creative adjustments to provide a worthwhile simulation to evaluate the model’s ability to achieve specific goals. For example, when trying to simulate predictions of sales for Walmart, you could examine how close total accumulated sales predictions match the actual values over a fixed duration.