Table of Contents

[Autocorrelation 1](#_Toc124362690)

[ACF vs. PACF Plots 1](#_Toc124362691)

[Autoregressive Models 4](#_Toc124362692)

[Optimizing the Model 7](#_Toc124362693)

[Predicting the Future with Auto-regressive Models 10](#_Toc124362694)

[Moving Average Components 12](#_Toc124362695)

[ARMA Model 12](#_Toc124362696)

[A Problem with Multi-Period Forecasts 15](#_Toc124362697)

[Walk-Forward Models 16](#_Toc124362698)

[Stationarity 17](#_Toc124362699)

[Differencing 18](#_Toc124362700)

# Autocorrelation

Autocorrelation refers to the correlation of a current observation with previous time steps. Partial autocorrelation represents the correlation amount specifically between each lag and the current time step.

### ACF vs. PACF Plots

The partial correlation function (PACF) provides a clearer view of the correlation of each lag with the current step since it removes all other correlations for prior lags. Figure 2 shows the Autocorrelation Function (ACF) and Partial autocorrelation (PACF) plots for sunspot activity.

The correlation of a variable value at the first step is always 1 because this is the correlation of the current variable value with itself. The blue shaded area represents the 95% confidence interval (see Figure 2). The fact that the first two lags of the PACF plot in Figure 2 appear outside the confidence interval means that the correlation between these lags are statistically significant.

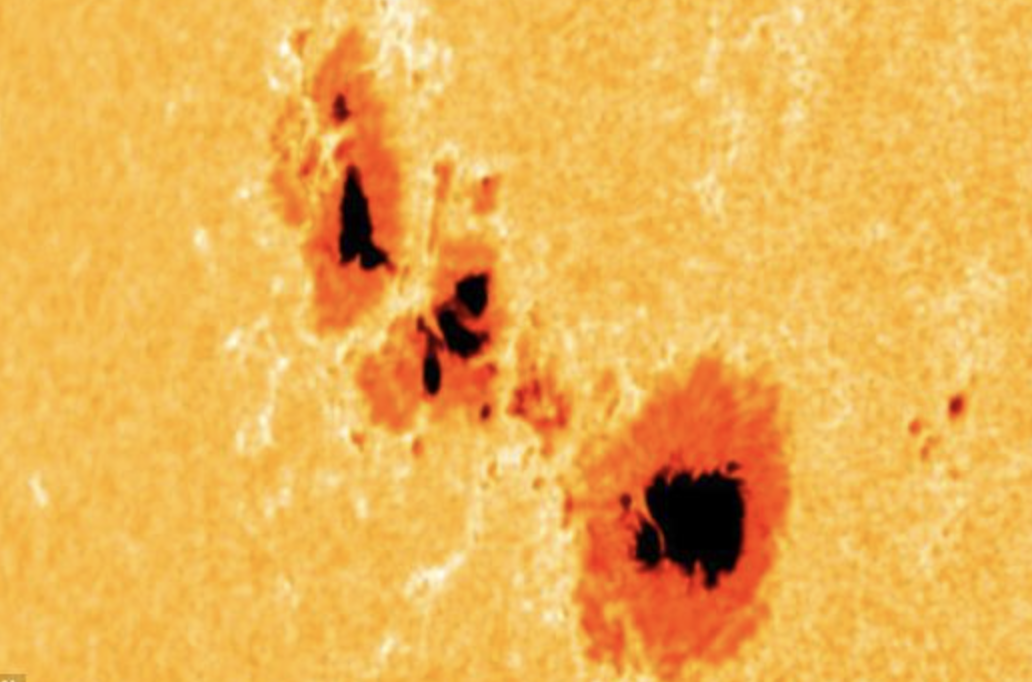
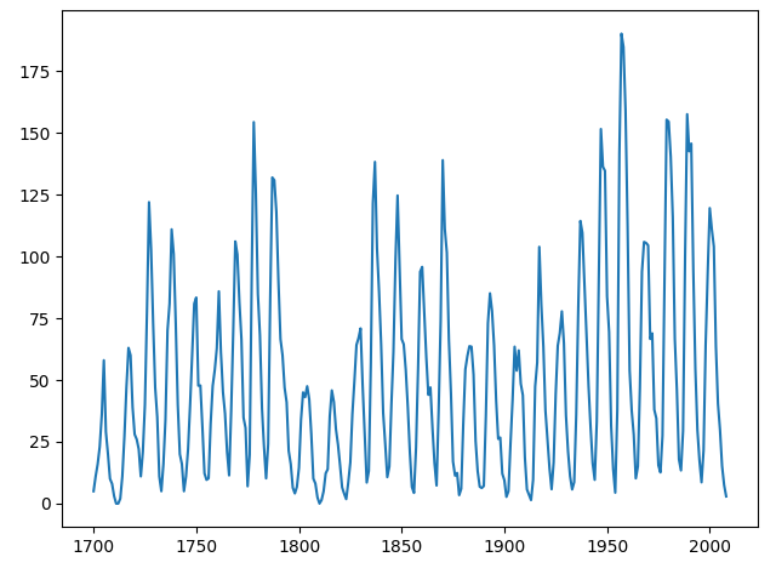
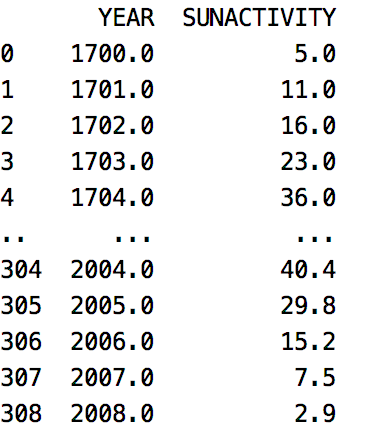
Example : Autocorrelation of Sunspot Activity

This example shows how to model yearly sunspot activity since the 1700’s.

Sunspots are temporary phenomena on the Sun's photosphere that appear as spots darker than the surrounding areas. They are regions of reduced surface temperature caused by concentrations of magnetic field flux. Their number varies according to the approximately 11-year solar cycle. Individual sunspots or groups of sunspots may last anywhere from a few days to a few months, but eventually decay. Sunspots expand and contract as they move across the surface of the Sun, with diameters ranging from 16 km (10 mi) to 160,000 km (100,000 mi). Larger sunspots can be visible from Earth without the aid of a telescope. They may travel at relative speeds of a few hundred meters per second when they first emerge.

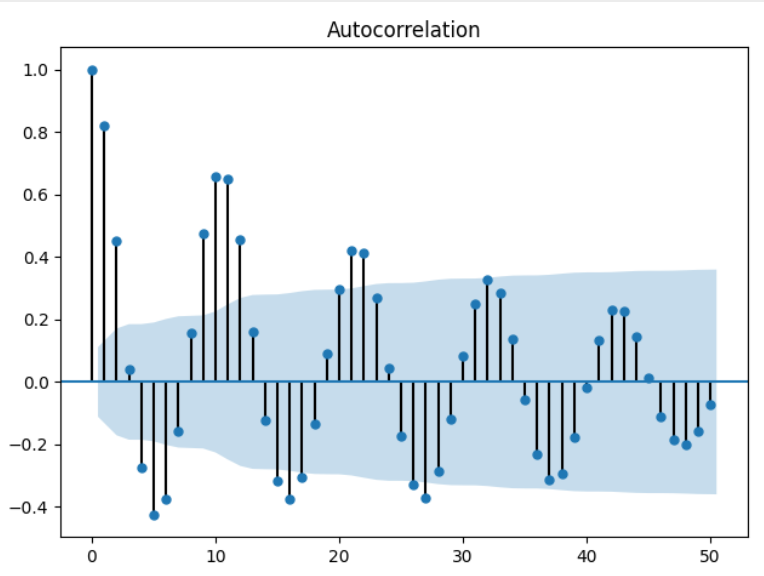
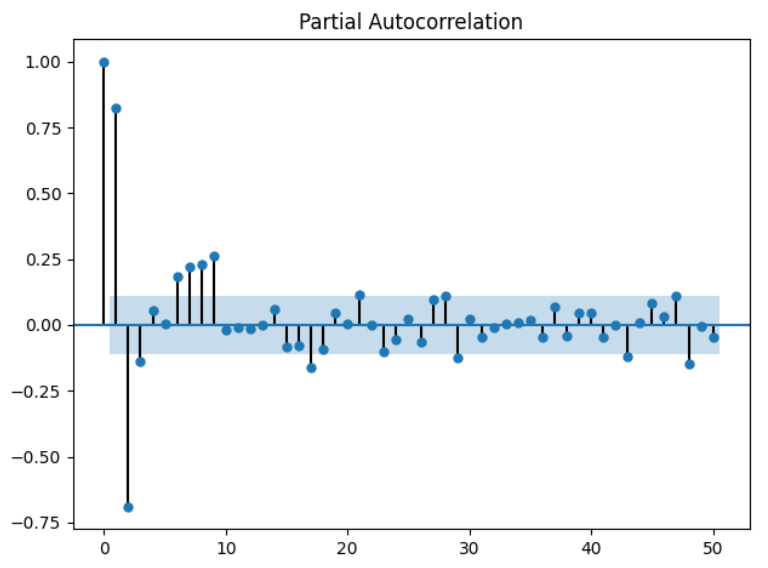
The chart and graph below present the sunspot activity. A picture of sunspots is at the right.

Figure : Sunspot Activity



The auto-correlation plot shows that the current time step is correlated to the previous time steps. The partial auto correlation function shows that the initial lag is correlated with the first 2-time lags and arguably others up until about lag 9.

Figure : Autocorrelation partial autocorrelation for sun spot data

Here is the code that presents the sun spot data:

|  |
| --- |
| import matplotlib.pyplot as plt  import statsmodels.api as sm  dta = sm.datasets.sunspots.load\_pandas().data  print(dta)  plt.plot(dta['YEAR'], dta['SUNACTIVITY'])  plt.show()  # Show autocorrelation function.  # General correlation of lags with past lags.  from statsmodels.graphics.tsaplots import plot\_acf  plot\_acf(dta['SUNACTIVITY'], lags=50)  plt.show()  # Show partial-autocorrelation function.  # Shows correlation of 1st lag with past lags.  from statsmodels.graphics.tsaplots import plot\_pacf  plot\_pacf(dta['SUNACTIVITY'], lags=50)  plt.show()  print(dta) |

Exercise (10 marks)

Using the back-shifting technique from lab week 2, generate back-shifted columns and perform OLS regression using the different lags to predict sunspot activity. Use the last 10 samples in the data set for testing. Show your model summary here and ensure that it includes only significant predictors.

|  |
| --- |
|  |

Show the RMSE here:

|  |
| --- |
| Root Mean Squared Error: 12.506442807794723 |

Plot the actual test values versus predicted values here.

|  |
| --- |
|  |

Show your complete program here:

import matplotlib.pyplot as plt  
import statsmodels.api as sm  
import pandas as pd  
from statsmodels.graphics.tsaplots import plot\_acf  
from statsmodels.graphics.tsaplots import plot\_pacf  
import warnings  
  
# CONFIGURATION SECTION  
NUM\_DAYS = 1200  
NUM\_TIME\_STEPS = 2  
  
TEST\_DAYS = 10  
  
warnings.filterwarnings("ignore")  
  
# 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)  
  
  
dta = sm.datasets.sunspots.load\_pandas().data  
print(dta)  
plt.plot(dta['YEAR'], dta['SUNACTIVITY'])  
plt.show()  
  
# Show autocorrelation function.  
# General correlation of lags with past lags.  
  
plot\_acf(dta['SUNACTIVITY'], lags=11)  
plt.show()  
  
# Show partial-autocorrelation function.  
# Shows correlation of 1st lag with past lags.  
  
plot\_pacf(dta['SUNACTIVITY'], lags=11)  
plt.show()  
  
print(dta)  
  
  
# Creates time shifted columns for as many time steps needed.  
def backShiftColumns(df, originalColName, numTimeSteps):  
 dfNew = df[[originalColName]]  
  
 for i in range(1, numTimeSteps + 1):  
 newColName = originalColName[0] + 't-' + str(i)  
 dfNew[newColName] = dfNew[originalColName].shift(periods=i)  
 return dfNew  
  
  
def prepareSunDf(columns):  
 df = dta  
  
 # Create data frame with back shift columns for all features of interest.  
 mergedDf = pd.DataFrame()  
 for i in range(0, len(columns)):  
 backShiftedDf = backShiftColumns(df, columns[i], NUM\_TIME\_STEPS)  
 if (i == 0):  
 mergedDf = backShiftedDf  
 else:  
 mergedDf = mergedDf.merge(backShiftedDf, left\_index=True,  
 right\_index=True)  
  
 newColumns = list(mergedDf.keys())  
  
 # Append stock symbol to column names.  
 for i in range(0, len(newColumns)):  
 mergedDf.rename(columns={newColumns[i]: "\_" + newColumns[i]}, inplace=True)  
  
 return mergedDf  
  
  
columns = ['YEAR', 'SUNACTIVITY']  
mergedDf = prepareSunDf(columns)  
mergedDf = mergedDf.dropna()  
print(mergedDf)  
  
  
  
import seaborn as sns  
  
corr = mergedDf.corr()  
plt.figure(figsize=(4, 4))  
ax = sns.heatmap(corr[['\_SUNACTIVITY']],  
 linewidth=0.5, vmin=-1,  
 vmax=1, cmap="YlGnBu")  
plt.show()  
  
  
# xfeatures = ['\_YEAR', '\_Yt-1', '\_Yt-2', '\_St-1', '\_St-2']  
xfeatures = ['\_St-1', '\_St-2']  
X = mergedDf[xfeatures]  
y = mergedDf[['\_SUNACTIVITY']]  
  
# Add intercept for OLS regression.  
import statsmodels.api as sm  
X = sm.add\_constant(X)  
  
# Split into test and train sets. The test data must be  
# the latest data range.  
lenData = len(X)  
X\_train = X[0:lenData-TEST\_DAYS]  
y\_train = y[0:lenData-TEST\_DAYS]  
X\_test = X[lenData-TEST\_DAYS:]  
y\_test = y[lenData-TEST\_DAYS:]  
  
# Model and make predictions.  
model = sm.OLS(y\_train, X\_train).fit()  
print(model.summary())  
predictions = model.predict(X\_test)  
  
# Show RMSE and plot the data.  
from sklearn import metrics  
import numpy as np  
print('Root Mean Squared Error:',  
 np.sqrt(metrics.mean\_squared\_error(y\_test, predictions)))  
  
plt.plot(y\_test, label='Actual', marker='o')  
plt.plot(predictions, label='Predicted', marker='o')  
plt.xticks(rotation=45)  
plt.legend(loc='best')  
plt.show()

How do the ACF and PACF plots suggest that a model to predict sunspot activity based on past sunspot activity is possible?

|  |
| --- |
| The first 3 lags are significant as they fall outside of the confidence interval shading area as seen in the plot. |

Exercise (3 marks)

Run the following code to generate ACF and PACF plot for opening prices of Microsoft stock.

|  |
| --- |
| from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf  from pandas\_datareader import data as pdr  import yfinance as yfin # Work around until  # pandas\_datareader is fixed.  import datetime  import matplotlib.pyplot as plt  import pandas as pd  import warnings  warnings.filterwarnings("ignore")  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  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  NUM\_DAYS = 70  df = getStock('MSFT', NUM\_DAYS)  print(df)  # Plot ACF for stock.  from statsmodels.graphics.tsaplots import plot\_acf  plot\_acf(df['Open'])  plot\_pacf(df['Open'])  plt.show() |

What do the ACF and PACF plots tell us about the opening prices related to the opening price lags at past time steps?

|  |
| --- |
| The first lag is significant and the ~13-16 day range is significant as seen in the plot. |

Is the most current timestep at the left or right of the ACF and PACF?

|  |
| --- |
| The most current timestep is on the left of the ACF and PACF. |

Exercise (1 mark)

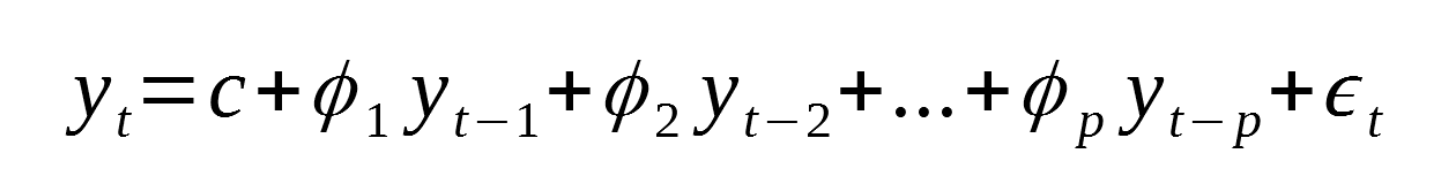
Why are the y-xis in Figure 2 ranges between -1 and 1?

|  |
| --- |
| The y-axis represents the correlation. 1 represents perfect correlation, 0 no correlation, and -1 perfect negative correlation. |

# Autoregressive Models

Autoregression models are time series models that use observations from previous time steps as input to a regression equation to predict the value at a future time step.

Where phi is a coefficient found by the model and y is an observation at a point in time the model becomes:



Where:

* p: is the order
* c: is a constant
* epsilon: noise

The statsmodel library helps to automate the generation of an autoregressive model for a single variable series.

|  |
| --- |
| model = AutoReg(train, lags=NUM\_TIME\_STEPS)  model\_fit = model.fit() |

When the model is fit it is possible to obtain the coefficients:

|  |
| --- |
| print('Coefficients: %s' % **model\_fit.params**) |

We can then use the model to make a forecast for the same number of time steps:

|  |
| --- |
| predictions = model\_fit.predict(start=len(train), end=NUM\_TEST\_DAYS, dynamic=False) |

Example : Building Autoregression Models

This example shows how to build an autoregressive model to predict sunspot activity for the most recent 10 years in the data set. We will use the ARIMA class to build our AR model. The ARIMA class offers several features that we will use very soon. For this case though, we are building an autoregression model which performs regression on the last two time steps. The number of time steps is assigned with the first argument of three in the order attribute. Will discuss the other arguments later but now set the second and third parameters of the order attribute to zero so we can have a pure AR model.

|  |
| --- |
| arma\_mod20 = ARIMA(dta['SUNACTIVITY'], order=(2, 0, 0)).fit() |

There are several ways to evaluate the model fit. We can examine the RMSE, AIC and BIC.

|  |
| --- |
| rmse = np.sqrt(mean\_squared\_error(dfTest['SUNACTIVITY'].values,  np.array(predictions)))  print('Test RMSE: %.3f' % rmse)  print('Model AIC %.3f' % model.aic)  print('Model BIC %.3f' % model.bic) |

We can also view a plot of the actual versus the predicted values (see ).

Table : Evaluating the AR Model

|  |  |
| --- | --- |
| \*\*\* Evaluating ARMA(2,0,0)  Coefficients: const 49.728505  ar.L1 1.386171  ar.L2 -0.687217  sigma2 278.758455  dtype: float64  Test RMSE: 25.145  Model AIC 2542.401  Model BIC 2557.203 |  |

Here is the code which builds the model and presents the results for evaluation of the sunspot activity predictions.

|  |
| --- |
| import pandas as pd  import matplotlib.pyplot as plt  import statsmodels.api as sm  from statsmodels.tsa.arima.model import ARIMA  from sklearn.metrics import mean\_squared\_error  import numpy as np  import warnings  warnings.filterwarnings("ignore")  dta = sm.datasets.sunspots.load\_pandas().data  dta.index = pd.Index(sm.tsa.datetools.dates\_from\_range('1700', '2008'), freq='Y')  # Show autocorrelation function.  # General correlation of lags with past lags.  from statsmodels.graphics.tsaplots import plot\_acf  plot\_acf(dta['SUNACTIVITY'], lags=50)  plt.show()  # Split the data.  NUM\_TEST\_YEARS = 10  lenData = len(dta)  dfTrain = dta.iloc[0:lenData - NUM\_TEST\_YEARS, :]  dfTest = dta.iloc[lenData-NUM\_TEST\_YEARS:,:]  def buildModelAndMakePredictions(AR\_time\_steps, dfTrain, dfTest):  # This week we will use the ARIMA model.  model = ARIMA(dfTrain['SUNACTIVITY'], order=(AR\_time\_steps, 0, 0),  freq='Y').fit()  print("\n\*\*\* Evaluating ARMA(" + str(AR\_time\_steps) + ",0,0)")  print('Coefficients: %s' %model.params)  # Strings which can be converted to time stamps are passed in.  # For this case the entire time range for the test set is represented.  predictions = model.predict('1999-12-31', '2008-12-31', dynamic=True)  rmse = np.sqrt(mean\_squared\_error(dfTest['SUNACTIVITY'].values,  np.array(predictions)))  print('Test RMSE: %.3f' % rmse)  print('Model AIC %.3f' % model.aic)  print('Model BIC %.3f' % model.bic)  return model, predictions  print(dfTest)  arma\_mod20, predictionsARMA\_20 = buildModelAndMakePredictions(2, dfTrain, dfTest)  plt.plot(dfTest.index, dfTest['SUNACTIVITY'],  label='Actual Values', color='blue')  plt.plot(dfTest.index, predictionsARMA\_20,  label='Predicted Values AR(20)', color='orange')  plt.legend(loc='best')  plt.show() |

Example : Comparing Predictions

In this example we will compare an AR(3) model with the AR(2) model. The AR(3) model does slightly better.

|  |  |  |
| --- | --- | --- |
| \*\*\* Evaluating ARMA(2,0,0)  Coefficients: const 49.728505  ar.L1 1.386171  ar.L2 -0.687217  sigma2 278.758455  dtype: float64  Test RMSE: 25.145  Model AIC 2542.401  Model BIC 2557.203 | \*\*\* Evaluating ARMA(3,0,0)  Coefficients: const 49.767980  ar.L1 1.303286  ar.L2 -0.520229  ar.L3 -0.120298  sigma2 274.716644  dtype: float64  Test RMSE: 22.480  Model AIC 2540.073  Model BIC 2558.575 |  |

To build this example, add this code just after where the AR(2) model is declared.

|  |
| --- |
| arma\_mod30, predictionsARMA\_30 = buildModelAndMakePredictions(3, dfTrain, dfTest) |

Then place this code where the other lines are plotted.

|  |
| --- |
| plt.plot(dfTest.index, predictionsARMA\_30,  label=**'Predicted Values AR(30)'**, color=**'brown'**) |

## Optimizing the Model

As you can see in , choosing different time step values for the autoregression model can lead to different results. We can actually automate the process of selecting the best time step.

Example : Selecting the time step order

This example shows how to find the best time step for our AR model. For this example, we are displaying the BIC (Bayesian Information Criterion) scores for AR models which include lag counts that range between 0 and 10.

|  |
| --- |
| dict\_values([ 0  0 3174.049905  1 2830.369176  2 2637.569703  3 2638.070335  4 2642.878700  5 2648.610908  6 2645.594430  7 2635.308015  8 2625.737548  9 2611.689366  10 2617.421383, (9, 0)]) |

Adding this code to will generate and display the information criterion or **ic** statistic for the past 10-time steps. For this model, AR(9) appears to be the best fit since the BIC score is the lowest. However, AR(2) offers a simpler and competitive option.

|  |
| --- |
| print("\n\*\*\* Comparing AR Model Time Steps")  import warnings  warnings.filterwarnings("ignore")  res = sm.tsa.arma\_order\_select\_ic(dta['SUNACTIVITY'], max\_ar=10, max\_ma=0, ic='bic')  print(res.values()) |

Exercise : (3 marks)

Plot the results for your AR(9) model with the AR(3), AR(2) predictions and the actual values. Show the graph here:

|  |
| --- |
|  |

Exercise : (3 marks)

Compare the AIC, BIC and RMSE values for each of the three AR models in a table. Show your table here. Please make it easy to read.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | |  | AIC | BIC | RMSE | | AR(2) | 2542.401 | 2557.203 | 25.145 | | AR(3) | 2540.073 | 2558.575 | 22.480 | | AR(9) | 2492.359 | 2533.064 | 14.885 | |

Exercise : (2 marks)

Explain in your own words why the AR(9) model is better than the AR(3) and AR(2) models.

|  |
| --- |
| The AIC, BIC, and RMSE is the lowest for AR(9). |

Exercise : (3 marks)

Using the equation definition for autoregressive models. Show the equation for the AR(3) model using the model coefficients here. (Refer to to obtain the model coefficients)

|  |
| --- |
|  |

Example : Australian Temperatures Autocorrelation

We will start with the minimum daily temperatures in Melbourne Australia. The ACF and PACF plots for daily temperature are displayed in Figure 3.

Figure : ACF and PACF plots for Melbourne Daily Temperatures

|  |
| --- |
|  |

Here is the code that loads and plots the data.

|  |
| --- |
| from pandas import read\_csv  import matplotlib.pyplot as plt  from statsmodels.tsa.ar\_model import AutoReg  from sklearn.metrics import mean\_squared\_error  from math import sqrt  import warnings  warnings.filterwarnings("ignore")  # Load the data.  PATH = "/Users/pm/Desktop/DayDocs/data/"  series = read\_csv(PATH + 'daily-min-temperatures.csv',  header=0, index\_col=0, parse\_dates=True, squeeze=True)  # Plot ACF.  from statsmodels.graphics.tsaplots import plot\_acf  plot\_acf(series, lags=20)  plt.show()  # Plot PACF.  from statsmodels.graphics.tsaplots import plot\_pacf  plot\_pacf(series, lags=20)  plt.show() |

Example : Building a 3-Day Model

This example shows how to build a 3-day model to predict minimum daily temperatures for Melbourne Australia. This technique is really useful because it helps us to focus our efforts to optimize our 3-day forecast. To build this example, add this code to the end of the code in Example 5.

|  |
| --- |
| NUM\_TEST\_DAYS = 3  # Split dataset into test and train.  X = series.values  lenData = len(X)  train = X[0:lenData-NUM\_TEST\_DAYS]  test = X[lenData-NUM\_TEST\_DAYS:]  # Train.  model = AutoReg(train, lags=3)  model\_fit = model.fit()  print('Coefficients: %s' % model\_fit.params)  print(model\_fit.summary())  # Make predictions.  predictions = model\_fit.predict(start=len(train),  end=len(train)+len(test)-1,  dynamic=False)  for i in range(len(predictions)):  print('predicted=%f, expected=%f' % (predictions[i], test[i]))  rmse = sqrt(mean\_squared\_error(test, predictions))  print('Test RMSE: %.3f' % rmse)  # Plot results.  plt.plot(test, marker='o', label='actual')  plt.plot(predictions, color='brown', linewidth=4,  marker='o', label='predicted')  plt.legend()  plt.show() |

Now we have a 3-day model with some degree of confidence that it will predict within a specific range. Three days may not be that impressive. However, the AR model gives us a chance to build and evaluate a 5-day or a 7-day model while helping to ensure the predictions within that range are reasonably accurate.

Remember to proceed with caution when using the AR model. The AR model does not account for random variance and unforeseen shocks to the environment. It can also be adversely affected by cyclical, seasonal or long-term trends. Other variables may also affect the AR model performance.

Exercise (2 marks)

Approximately how many lags does the PACF plot from Example 5 suggest should be used in building the model?

|  |
| --- |
| 9 lags |

Exercise (4 marks)

Try running your code for Example 6 with the number of time steps that you listed in Exercise 8. Next, view the model summary output with your new step count. Show your model summary here:

|  |
| --- |
|  |

Which coefficients are statistically significant?

|  |
| --- |
| . |

Is the RMSE higher with the new set of coefficients when compared to using data from only three time steps?

|  |
| --- |
| . |

### Predicting the Future with Auto-regressive Models

Updating these types of forecasts can be challenging. You might consider updating the model coefficients on a regular basis.

To make predictions with an AR model we can extract the weights from the auto-regressive process. Then we can feed in the latest observation, make a prediction and then update the time steps. This process can be repeated iteratively.

Example : Forecasting into the Future

This example shows how to make predictions in the future with the weights that are returned by the auto-regressive process. Since the model is designed with a 3-day range I am only using the model for a 3-day range. To build this example, add this code onto the code from Example 6.

|  |
| --- |
| # Use model coefficients from autoregression to make a prediction.  def makePrediction(t\_1, t\_2, t\_3):  intercept = 1.88820768  t1Coeff = 0.70018223  t2Coeff = - 0.05949822  t3Coeff = 0.19010829  prediction = intercept + t1Coeff\*t\_1\  + t2Coeff\*t\_2\  + t3Coeff\*t\_3  return prediction  testLen = len(test)  t\_1 = test[testLen-1]  t\_2 = test[testLen-2]  t\_3 = test[testLen-3]  futurePredictions = []  for i in range(0, NUM\_TEST\_DAYS):  prediction = makePrediction(t\_1, t\_2, t\_3)  futurePredictions.append(prediction)  t\_3 = t\_2  t\_2 = t\_1  t\_1 = prediction  print("Here is a one week temperature forecast: ")  print(futurePredictions) |

Exercise (5 marks)

Change the model so it becomes a 7-day model. Update the makePredictions() function. Show your updated makePredictions() function here:

|  |
| --- |
| # Use model coefficients from autoregression to make a prediction. def makePrediction(t\_1, t\_2, t\_3, t\_4, t\_5, t\_6, t\_7):  intercept = 1.88820768  t1Coeff = 0.70018223  t2Coeff = - 0.05949822  t3Coeff = 0.19010829  t4Coeff = 1  t5Coeff = 1  t6Coeff = 1  t7Coeff = 1   prediction = intercept + t1Coeff \* t\_1 + t2Coeff \* t\_2 + t3Coeff \* t\_3 + t4Coeff \* t\_4 + t5Coeff + t\_5 + \  t6Coeff \* t\_6 + t7Coeff \* t\_7   return prediction |

Using the final day in the data set as your first input, show your future prediction output here:

|  |
| --- |
| . |

# Moving Average Components

**Note:** A moving average model (MA Model) \*is not to be confused with the moving average\*.

Moving average components within an autoregressive model refer to unexplained variance at each time step. The MA(q) component is a linear combination of past error terms.

In a moving average model (MA model), depends only on the lagged forecast errors where εt is white noise and q is the number of time steps.

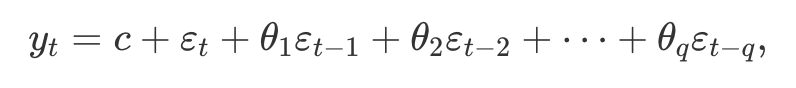
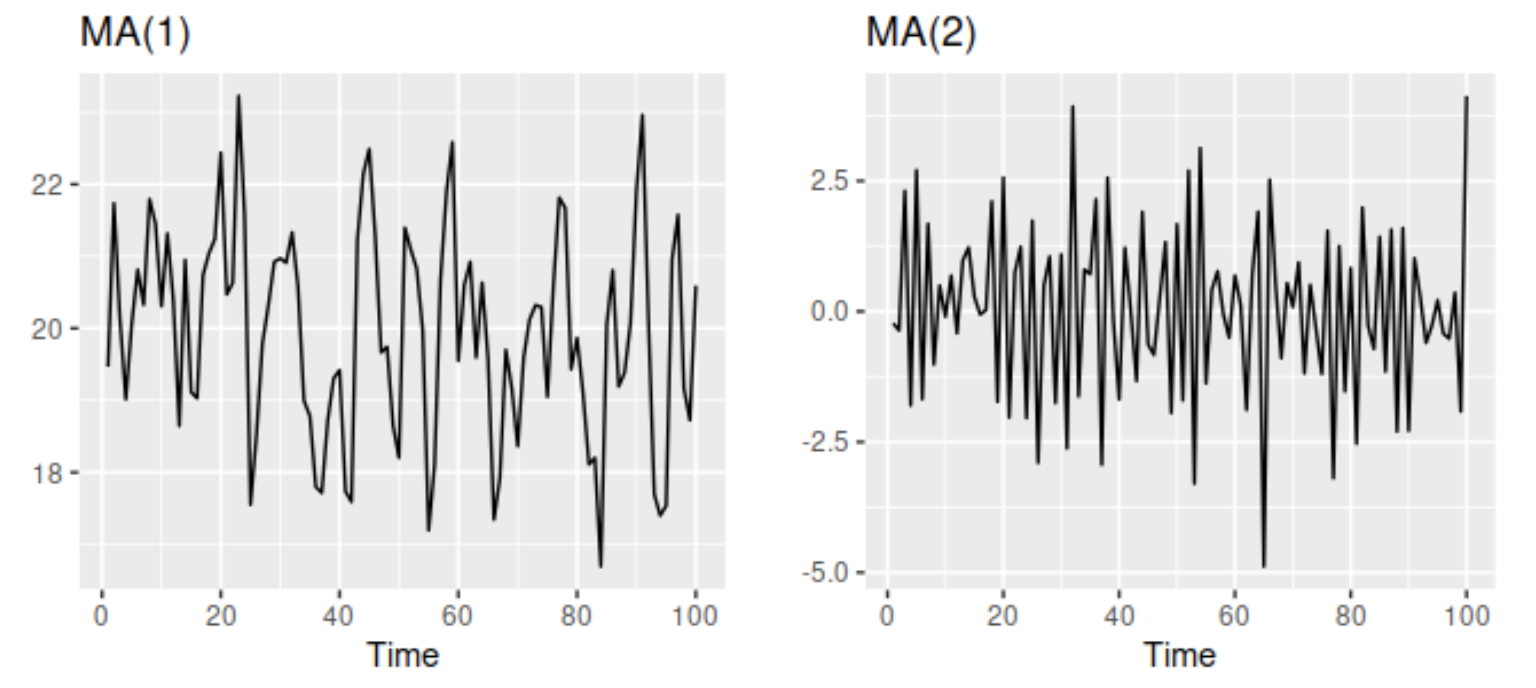
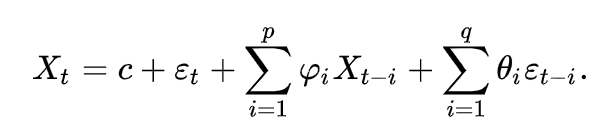


Figure 4: Unexplained shocks at time steps 1 and 2.



## ARMA Model

AR and MA models can sometimes be combined to generate better predictions than are possible with only an AR model. Together, the AR(p) and MA(q) components create an **ARMA** model.

 +

Example 8: Plotting ARMA Predictions

By including a moving average component in addition to an auto-regressive component, we ideally can improve the quality of our predictions. Over a period of 5 days it appears that including the moving average and a in our model improves the predictive result compared to not including it.

|  |
| --- |
| ar ma rmse  6 1 1 33.174186  7 1 2 33.346990  4 0 4 33.748635  3 0 3 33.783283  8 1 3 33.905250  1 0 1 33.986710  2 0 2 34.088609  0 0 0 34.102631  5 1 0 35.686172  …  15 3 0 45.457091  14 2 4 52.583664  19 3 4 59.351976  24 4 4 59.982582 |

Here is the code which grid searches the models based on different lag totals for auto regressive moving average (residual error) components.

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  import statsmodels.api as sm  import statsmodels.tsa.arima.model as sma  import matplotlib.pyplot as plt  import pandas as pd  from sklearn.metrics import mean\_squared\_error  import numpy as np  def getData():  df = sm.datasets.sunspots.load\_pandas().data[['SUNACTIVITY']]  df.index = pd.date\_range(start='1700', end='2009', freq='A')  TEST\_SZ = 5  train = df[0:len(df)-TEST\_SZ]  test = df[len(df)-TEST\_SZ:]  return train, test  def buildModel(df, ar, i, ma):  model = sma.ARIMA(df['SUNACTIVITY'], order=(ar, i, ma)).fit()  return model  def predictAndEvaluate(model, test, title):  print("\n\*\*\*" + title)  print(model.summary())  predictions = model.predict(start='2010', end='2014')  mse = mean\_squared\_error(predictions, test)  rmse = np.sqrt(mse)  print("RMSE: " + str(rmse))  return rmse  train, test = getData()  modelStats = []  for ar in range(0, 5):  for ma in range(0, 5):  model = buildModel(train, ar, 0, ma)  title = str(ar) + "\_0\_" + str(ma)  rmse = predictAndEvaluate(model, test, title)  modelStats.append({"ar":ar, "ma":ma, "rmse":rmse})  dfSolutions = pd.DataFrame(data=modelStats)  dfSolutions = dfSolutions.sort\_values(by=['rmse'])  print(dfSolutions) |

Exercise 11 (10 marks)

Use the following code to find the best ARMA model to find an optimal AR and MA combination for Microsoft stock Open prices. Here is some code to start with:

|  |
| --- |
| import warnings  warnings.filterwarnings("ignore")  import numpy as np  from scipy import stats  import pandas as pd  import matplotlib.pyplot as plt  import statsmodels.api as sm  from statsmodels.tsa.arima.model import ARIMA  dta = sm.datasets.sunspots.load\_pandas().data  import datetime  from pandas\_datareader import data as pdr  import yfinance as yfin # Work around until  # pandas\_datareader is fixed.    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  stkName = 'MSFT'  dfStock = getStock(stkName, 400)  # Split the data.  NUM\_TEST\_DAYS = 5  lenData = len(dfStock)  dfTrain = dfStock.iloc[0:lenData - NUM\_TEST\_DAYS, :]  dfTest = dfStock.iloc[lenData-NUM\_TEST\_DAYS:,:]  plt.plot(dfStock.index, dfStock['Open'])  plt.show() |

Hints:

The only possible feature to build the ARMA model with is opening price. No other features are needed.

An easier way to make the prediction is by setting the start to the beginning and end of the test set like so:

start = len(dfTrain)  
 end = start + len(dfTest) -1  
 predictions = model.predict(start=start, end=end, dynamic=True)

Example 1 shows the main body of code that is needed to build the ARMA solution.

State the ARMA model here:

|  |
| --- |
|  |

In English, interpret the model coefficients of the ARMA model:

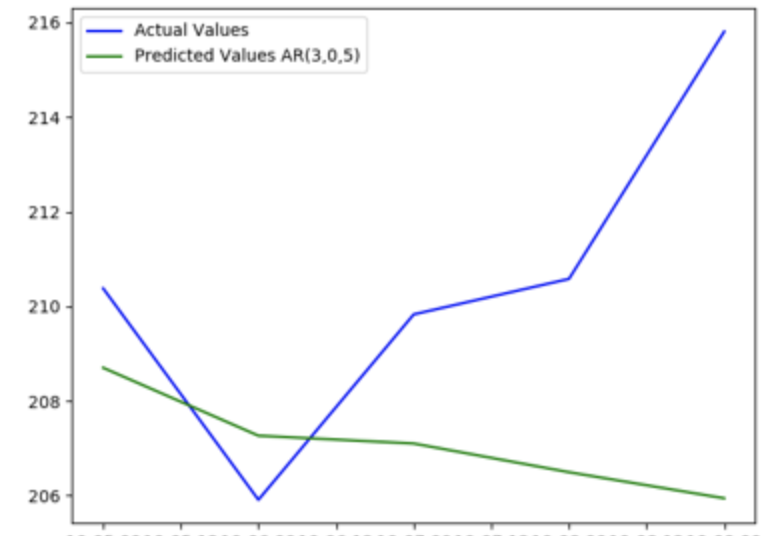
|  |
| --- |
| Ar.L1 autoregressive term with a lag of 1  Ar.L2 autogressive term with lag of 2  Ma.L1 moving average terms with lag of 1  Sigma1 variance of residual values |

Show a plot of the actual versus predicted values for 5 days here:

|  |
| --- |
|  |

## A Problem with Multi-Period Forecasts

A problem you may encounter when making multi-day forecasts is the results are flat.

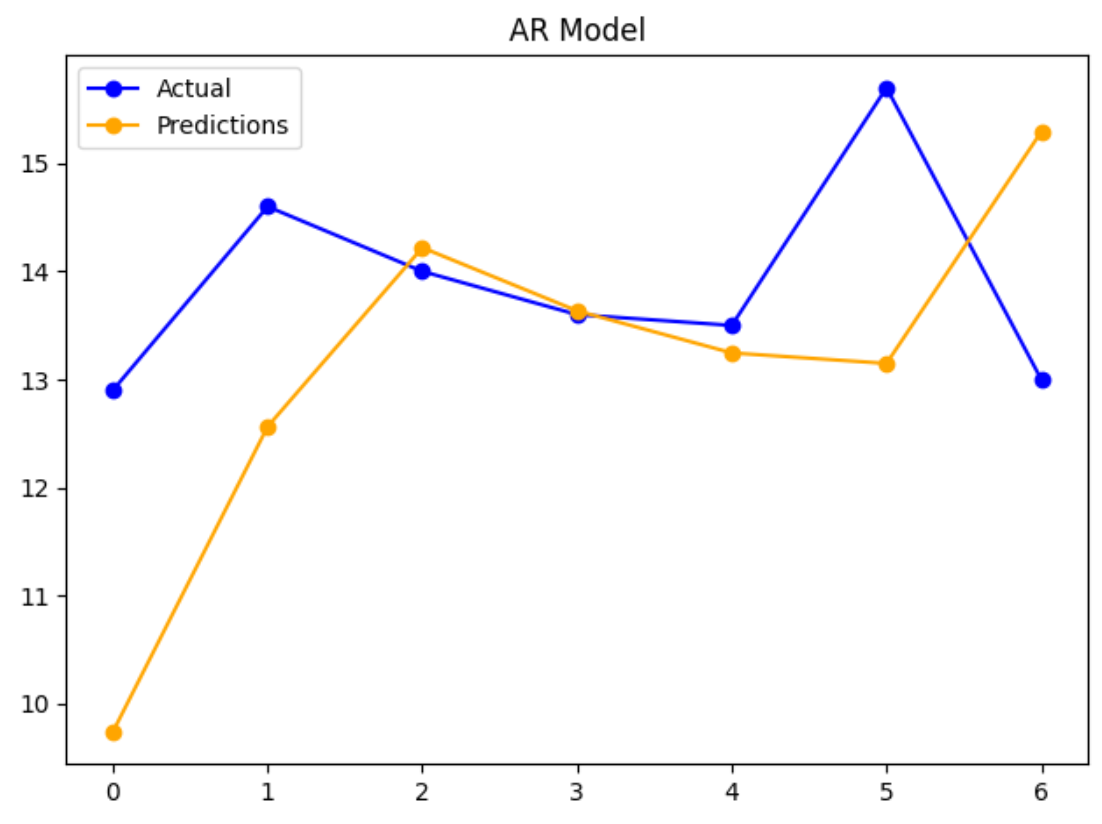


Further-ahead-predictions inherently become increasingly less accurate over time. The results are usually smoothed and they will not represent the fluctuations in volatility over time. Because of this issue, the examples today will focus on making predictions with the latest observations that are one period ahead. For today we will use the number of days as the period.

# Walk-Forward Models

Example : Day Ahead Prediction

This example shows how to build an AR model with a walk-forward prediction that is always one day ahead. At every iteration for each day, the observations are updated with the most recently available data before making a new prediction.



RMSE: 1.928

We are going to re-use these routines in several examples. The documentation for this code is in the comments.

This example implements a day ahead prediction

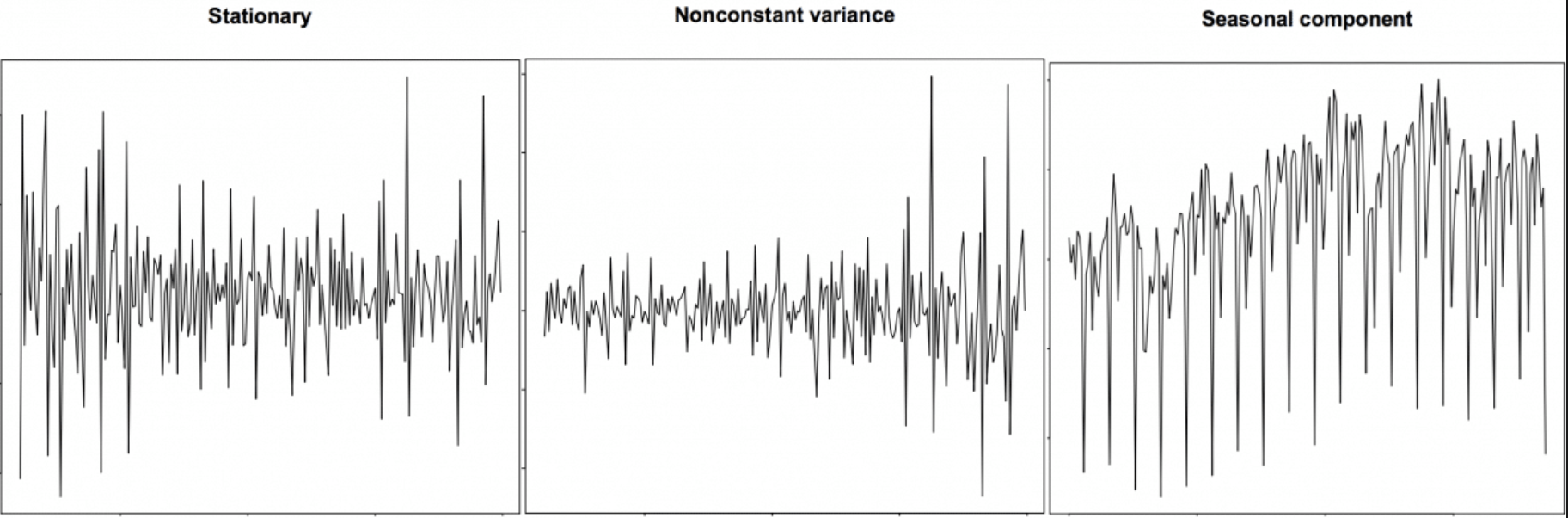
|  |
| --- |
| from pandas import read\_csv  import matplotlib.pyplot as plt  import statsmodels.tsa.arima.model as sma  from sklearn.metrics import mean\_squared\_error  from math import sqrt  PATH = "/Users/pm/Desktop/DayDocs/data/"  series = read\_csv(PATH + 'daily-min-temperatures.csv', header=0, 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:]  # 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, 0, 0)).fit()  print("Model parameters: " + str(model.arparams))  # Get the ar\_modle parameters.  ar\_coef = model.arparams  # Make the prediction.  yhat = predict(ar\_coef, history)  #################################################################  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 history list.  # 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() |

# Stationarity

Stationarity refers to constant random variance. The left image in exhibits stationarity because the variance is constant and random. The middle image in Figure 5 on the other hand shows increasing variance that appears to be heteroskedastic. The variance on the right of Figure 5 appears to be somewhat cyclical.

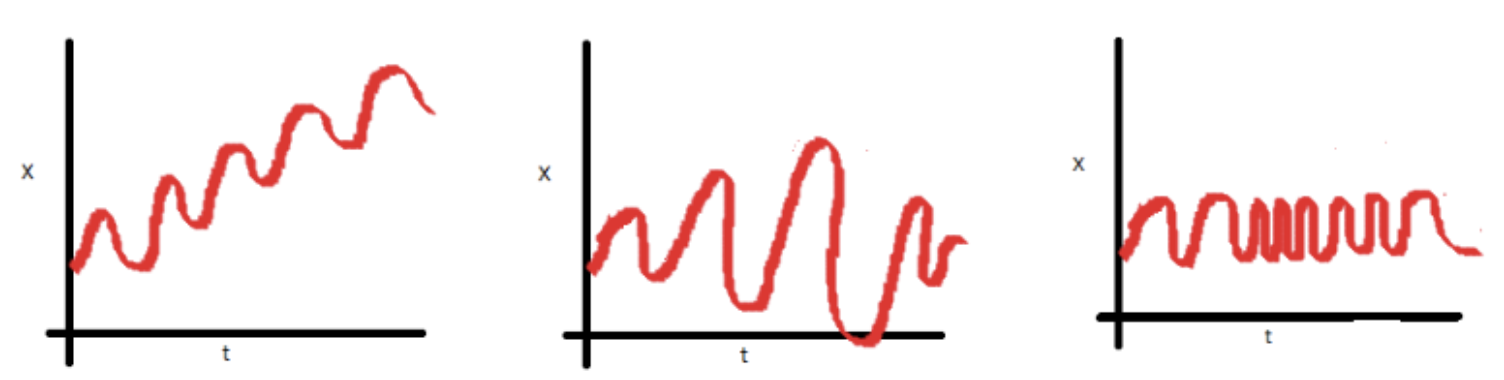
Many time series algorithms require stationary data so it is important to understand how to detect non-stationarity and how to transform it into a stationary set. For example, when trying to understand risk of markets it is important to remove trend, seasonal and cyclical data. There is also a need for stationary data when trying to understand volatility such as random consumption of hydro power which is not explained by trend, cyclical or seasonal factors.

Figure 5: Stationary versus Non-stationary Variance



Non-stationary data can contain trend and seasonal data. A stationary series is one where trend, seasonal and cyclical fluctuations have been removed. The result leads to a mean, variance and covariance that do not vary with time. Figure 6 shows a stationary series on the right and non-stationary series in the middle and on the left.

Figure 6: Trend on the left, uneven variance in the middle, and stationarity on the right.



# Differencing

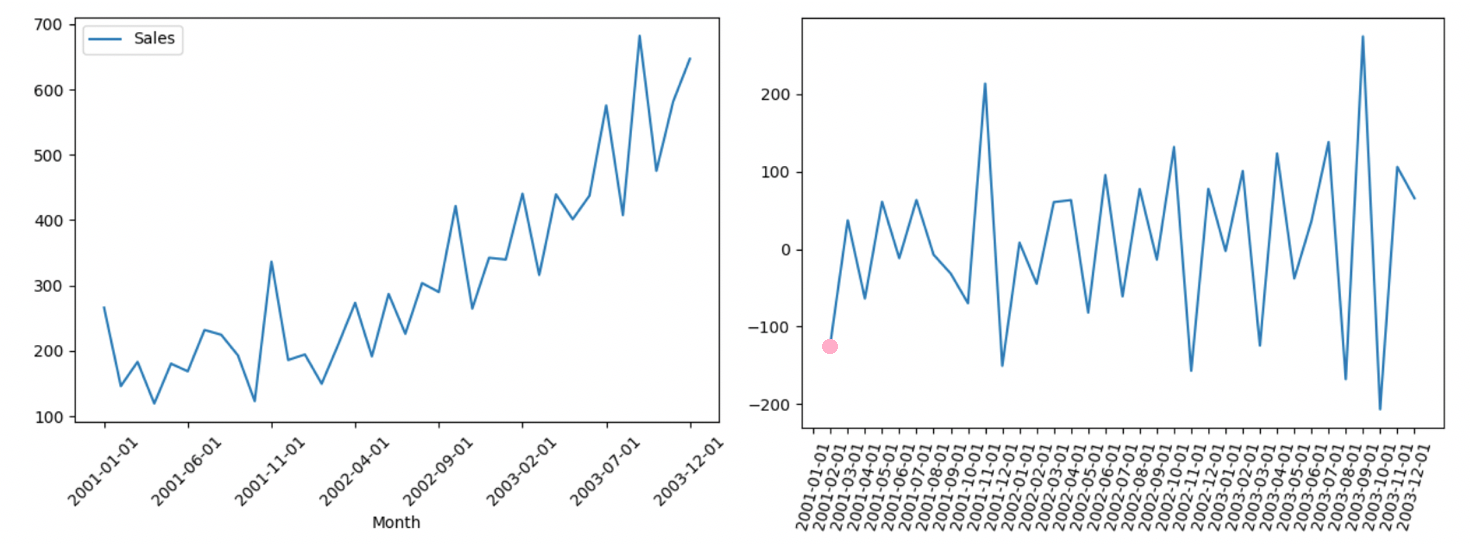
Differencing is a popular transformation that helps to stabilize the mean of a time series. Differencing removes changes in the level of a time series, and therefore eliminates (or reduces) trend and seasonality. Differencing is done to eliminate series dependence over time. This dependence includes trend and seasonality.

|  |
| --- |
| diff = df.diff()  plt.plot(diff)  plt.xticks(rotation=75)  plt.show() |

Example 10: Differencing

This example shows how to transform this undifferenced data set into a differenced data set. The graph on the right of Figure 7 shows non-stationary data. The graph on the right of Figure 7 shows the same data set after differencing.

Figure 7: Transforming Undifferenced Data on the Left to Differenced Data on the Right



Here is the full code sample:

|  |
| --- |
| import pandas as pd  import datetime  import matplotlib.pyplot as plt  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = 'shampoo.csv'  df = pd.read\_csv(PATH + FILE, index\_col=0)  df.info()  # Plot data before differencing.  df.plot()  plt.xticks(rotation=45)  plt.show()  # Perform differencing.  dfDifferenced = df.diff()  # Plot data after differencing.  plt.plot(dfDifferenced)  plt.xticks(rotation=75)  plt.show() |

Example 11: Manually Performing Differencing

This example shows how to manually perform the same differencing routine that was implemented in Example 10. In this case:

Differenced sales = - = 145.9 – 266 = -120.1.

Table 2: Sales Differencing Calculations

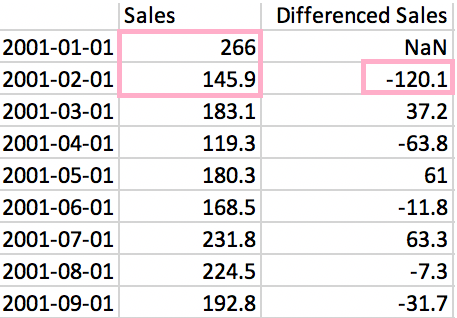


Table 2 shows how the differenced value is calculated for February 1, 2001. This differenced value is also denoted by the pink dot in Figure 7.

Exercise 12 (1 mark)

Starting with Example 11, show the calculations by hand that are needed to calculate the differenced value of $37.20 for March 1st 2001.

|  |
| --- |
| Differenced sales = - = 183.1 - 145.9 = 37.2 |