Table of Contents

[References 1](#_Toc123725431)

[Time Series Decomposition 1](#_Toc123725432)

[Trend 2](#_Toc123725433)

[Seasonality 2](#_Toc123725434)

[Cyclic Behaviour 2](#_Toc123725435)

[Classical Decomposition of a Time Series 3](#_Toc123725436)

[Multiplicative Decomposition 3](#_Toc123725437)

[Additive Decomposition 5](#_Toc123725438)

[Comparing Additive and Multiplicative Decomposition 6](#_Toc123725439)

[Seasonal Detail 7](#_Toc123725440)

[De-Trending 9](#_Toc123725441)

[De-Seasonalizing Data 10](#_Toc123725442)

[Rolling Moving Averages 11](#_Toc123725443)

[Trailing Mean 11](#_Toc123725444)

[Centered Moving Average 11](#_Toc123725445)

[Weighted Moving Average (Rolling Mean) 13](#_Toc123725446)

[Using Moving Averages to Buy and Sell Stock 13](#_Toc123725447)

[Exponentially Weighted Moving Average (Exponential Smoothing) 15](#_Toc123725448)

# References

Some of the content in this document is based on the following articles.

<https://otexts.com/fpp2/classical-decomposition.html>

<https://www.machinelearningplus.com/time-series/time-series-analysis-python/>

<https://machinelearningmastery.com/moving-average-smoothing-for-time-series-forecasting-python/>

<https://machinelearningmastery.com/autoregression-models-time-series-forecasting-python/>

# Time Series Decomposition

Over the next few weeks we will be building univariate time series models. Time series data will often exhibit trend, seasonal and cyclic behaviour. However, recurrent components cannot always be modelled as easily with the types of models that we will be building.  With that in mind, today we will break down time series data into recurrent and error components to better understand how to analyze and prepare the data for modelling.

The statsmodels library provides an implementation of the naive, or classical, decomposition method in a function called seasonal\_decompose(). It requires that you specify whether the model is additive or multiplicative.

**Caution:**

Healthy skepticism is needed when using automated decomposition methods. The patterns revealed by decomposition almost always are not actually realized with such clarity. It is important to treat decomposition as a potentially useful analysis tool while also considering exploring the data in many alternate ways.

### Trend

A trend is observed when there is an increasing or decreasing slope in the time series.

### Seasonality

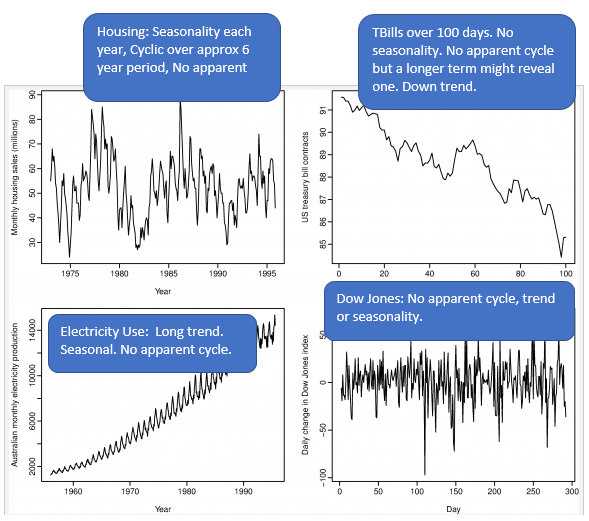
When seasonality is observed there is a distinct repeated pattern observed between regular calendar intervals that occur due to seasonal factors.

### Cyclic Behaviour

Cyclic behaviour occurs when a rise and fall pattern occurs at fixed calendar-based intervals. Care should be taken to not confuse ‘cyclic’ effect with ‘seasonal’ effect. Unlike the seasonality, cyclic effects are typically influenced by the business and other socio-economic factors.

Figure 1 illustrates four different combinations of cyclic, seasonal and trend behaviour.

Figure 1: Varying Combinations of Cyclic, Seasonal, and Trend



## Classical Decomposition of a Time Series

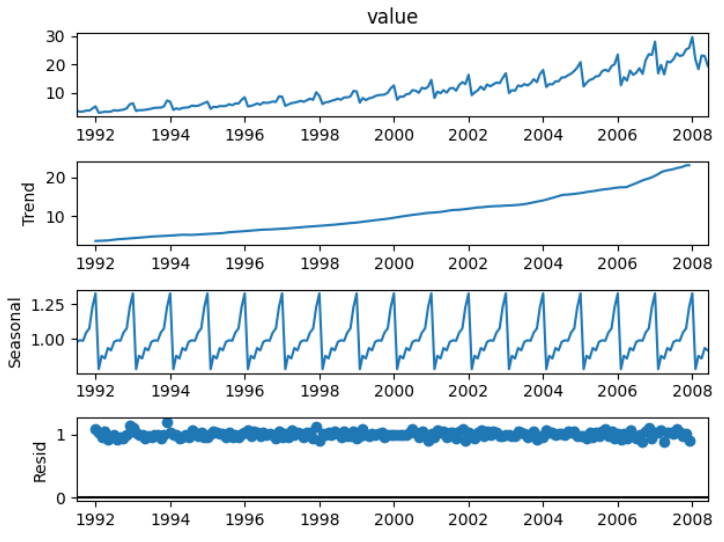
There are two ways to decompose a time series:

1. Multiplicative Decomposition.
2. Additive decomposition.

### Multiplicative Decomposition

This section shows how to decompose time series data into separate components by using multiplicative decomposition. Figure 2 shows visualizations of the separate components.

Figure 2: Multiplicative Decomposition of Drug Sales



Example 1: Multiplicative Decomposition Visualization

This code is used to plot the multiplicative series that is shown in Figure 2.

|  |
| --- |
| from statsmodels.tsa.seasonal import seasonal\_decompose  import pandas as pd  import matplotlib.pyplot as plt  # Import data.  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "drugSales.csv"  df = pd.read\_csv(PATH + FILE, parse\_dates=['date'], index\_col='date')  type(df.index)  # Perform decomposition using multiplicative decomposition.  tseries = seasonal\_decompose(df['value'], model='multiplicative', extrapolate\_trend="freq")  tseries.plot()  plt.show() |

Example 2: Numeric Multiplicative Decomposition

This example helps to explain how the different time steps are represented through multiplicative decomposition. To build this example add this code to the end of Example 1.

|  |
| --- |
| # Extract the Components ----  # Actual Values = Product of (Seasonal \* Trend \* Resid)  dfComponents = pd.concat([tseries.seasonal, tseries.trend,  tseries.resid, tseries.observed], axis=1)  dfComponents.columns = ['seas', 'trend', 'resid', 'actual\_values']  print(dfComponents.head()) |

The output in Table 1identifies the different numeric components of the actual drug sales totals for each time step. All cell values in each row can be multiplied together to calculate the actual value.

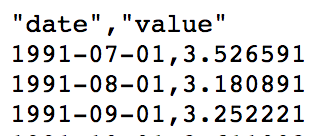
**0.987845\*3.060085\*1.166629 = 3*.526591***

Table 1: Seasonal, Trend, and Error Components for Multiplicative Decomposition

|  |
| --- |
| seas trend resid actual\_values  date  1991-07-01 0.987845 3.060085 1.166629 3.526591  1991-08-01 0.990481 3.124765 1.027745 **3.180891**  1991-09-01 0.987476 3.189445 1.032615 3.252221  1991-10-01 1.048329 3.254125 1.058513 3.611003  1991-11-01 1.074527 3.318805 0.999923 3.565869 |

Exercise 1 (2 marks)

Write a tiny Python program to multiply the seasonal, trend and residual (error) components to calculate and print the second value in the time series of **3.180891**. Show your program here:



|  |
| --- |
| from statsmodels.tsa.seasonal import seasonal\_decompose import pandas as pd import matplotlib.pyplot as plt  # Import data. PATH = 'C:\\datasets\\' FILE = "drugSales.csv" df = pd.read\_csv(PATH + FILE, parse\_dates=['date'], index\_col='date') type(df.index)  # Perform decomposition using multiplicative decomposition. tseries = seasonal\_decompose(df['value'], model='multiplicative', extrapolate\_trend="freq")  tseries.plot() plt.show()  # Extract the Components ---- # Actual Values = Product of (Seasonal \* Trend \* Resid) # dfComponents = pd.concat([tseries.seasonal, tseries.trend, # tseries.resid, tseries.observed], axis=1) # dfComponents.columns = ['seas', 'trend', 'resid', 'actual\_values'] # print(dfComponents.head())   dfComponents = pd.concat([tseries.observed], axis=1) dfComponents.columns = ['value'] print(dfComponents.head()) |

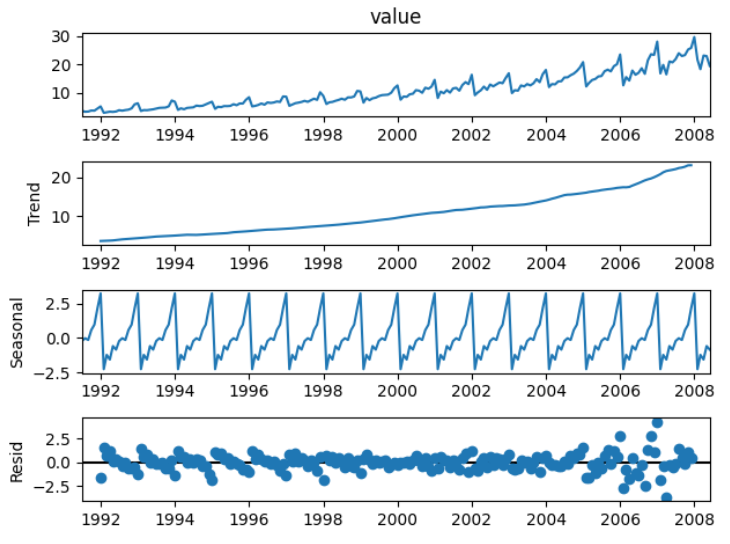
## Additive Decomposition

The process for implementing additive decomposition is almost identical to multiplicative decomposition except the seasonal, trend and error components are added together to form the

Example 3: Additive Decomposition

Here are the seasonal, trend and error components that are extracted using additive decomposition. Figure 3 shows the additive components.

Figure 3: Additive Decomposition Visualization



To build this example change the parameter in Example 1 to additive.

|  |
| --- |
| tseries = seasonal\_decompose(df['value'], model='additive', extrapolate\_trend="freq") |

The actual drug sales values are calculated by adding the seasonal, trend and error components together.

**-0.140765+3.060085+0.607271= 3*.526591***

|  |
| --- |
| seas trend resid actual\_values  date  1991-07-01 -0.140765 3.060085 0.607271 3.526591  1991-08-01 0.027747 3.124765 0.028379 3.180891  1991-09-01 -0.090361 3.189445 0.153137 3.252221  1991-10-01 0.602876 3.254125 -0.245998 3.611003  1991-11-01 0.970698 3.318805 -0.723634 3.565869 |

### Comparing Additive and Multiplicative Decomposition

The multiplicative and additive diagrams are similar. However, the error component in the additive decomposition graph at the right of Figure 4 shows some pattern remains. The remaining pattern in the residual component of additive decomposition suggests that multiplicative decomposition is slightly better at decomposing the series. Ideally, we want to make the residual component completely random.

Figure 4: Comparing Multiplicative and Additive Decomposition

|  |  |
| --- | --- |
| **Multiplicative Decomposition** | **Additive Decomposition** |
|  |  |

## Seasonal Detail

It is further possible to examine seasonal date in detail. After decomposing a series, you can easily isolate any of the different decomposed components:

trend = tseries.trend

seasonal = tseries.seasonal

error = tseries.resid

Example 4: Plotting Seasonal Detail

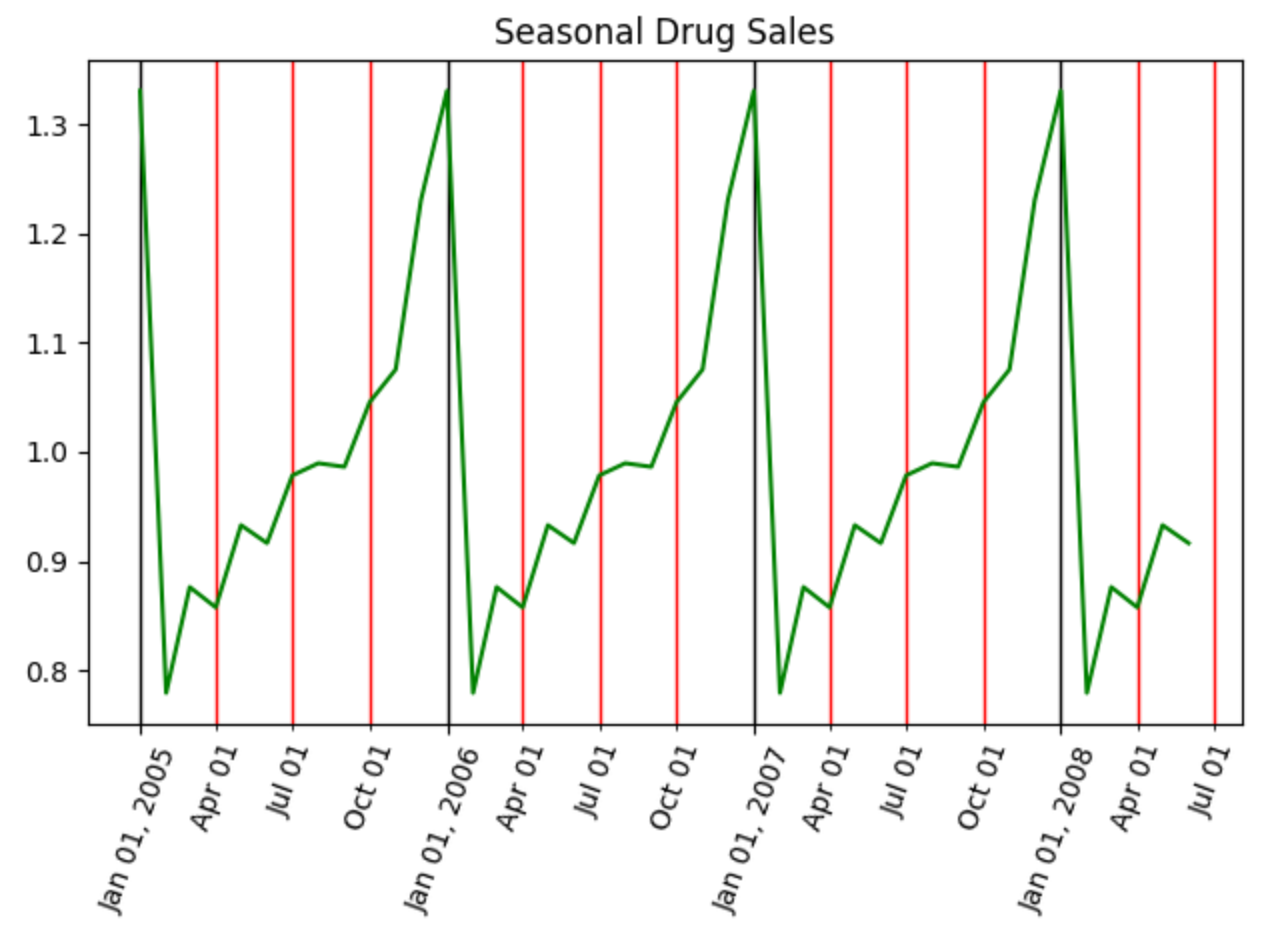
To avoid too much clutter, this example plots seasonal detail starting from January 2005:

start, end = '2005-01', '2009-12'

ax.plot(seasonal.loc[start:end], color='green')

Techniques from the lab are used to plot the seasonal data in Figure 5. It is interesting to note how drug sales spike in the December holiday season. Try not to read too much into this though since I believe the data is fictitious.

Figure 5: Seasonal Drug Sales Between January 1, 2005 to July 2008.



|  |
| --- |
| from statsmodels.tsa.seasonal import seasonal\_decompose  import pandas as pd  import matplotlib.pyplot as plt  import matplotlib.dates as mdates  # Import data.  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "drugSales.csv"  df = pd.read\_csv(PATH + FILE, parse\_dates=['date'], index\_col='date')  type(df.index)  fig, ax = plt.subplots()  # Perform decomposition using multiplicative decomposition.  tseries = seasonal\_decompose(df['value'], model='multiplicative',  extrapolate\_trend='freq')  trend = tseries.trend  seasonal = tseries.seasonal  # Set vertical major grid.  ax.xaxis.set\_major\_locator(mdates.YearLocator(day=1))  ax.xaxis.grid(True, which = 'major', linewidth = 1, color = 'black')  # Set vertical minor grid.  ax.xaxis.set\_minor\_locator(mdates.MonthLocator(bymonth=(1,4,7,10),bymonthday=1))  ax.xaxis.grid(True, which = 'minor', linewidth = 1, color = 'red')  start, end = '2005-01', '2009-12'  ax.plot(seasonal.loc[start:end], color='green')  plt.setp(ax.xaxis.get\_majorticklabels(), rotation=70)  plt.setp(ax.xaxis.get\_minorticklabels(), rotation=70)  ax.xaxis.set\_minor\_formatter(mdates.DateFormatter('%b %d'))  ax.xaxis.set\_major\_formatter(mdates.DateFormatter('%b %d, %Y'))  plt.title("Seasonal Drug Sales")  plt.show() |

Exercise 2 (3 marks)

Perform additive and multiplicative decomposition with the **AirPassengers.csv** file.Show graphs of additive and multiplicative components.

|  |
| --- |
| Multiplicative:    Additive: |

Which type of decomposition does a better job at breaking down all of the patterns and why?

|  |
| --- |
| Multiplicative is better, the residual values are ~1. In the additive, the residual is not consistent, range from -50 to 50. |

## De-Trending

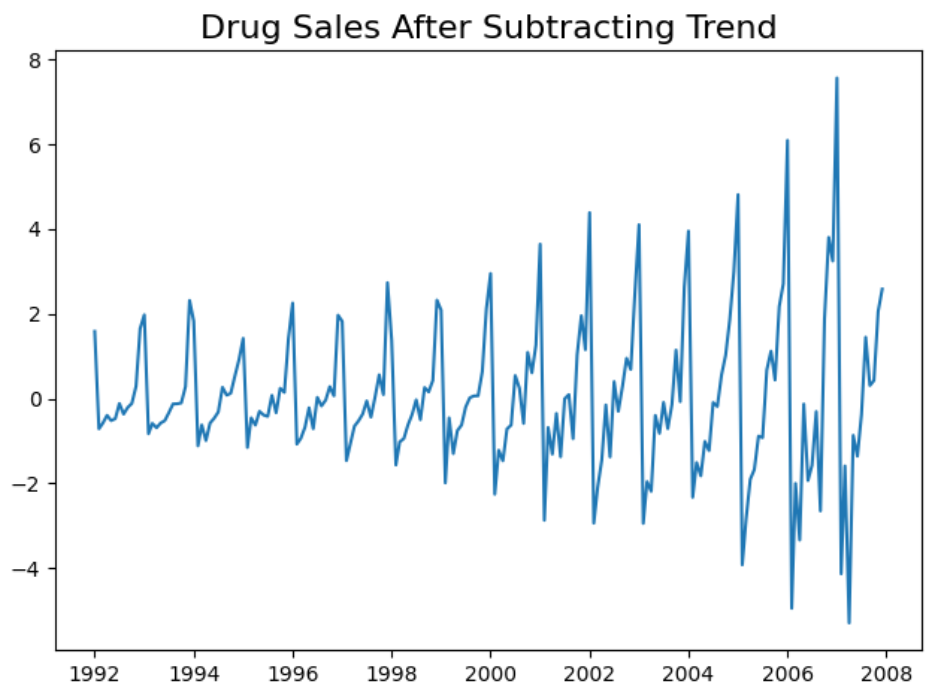
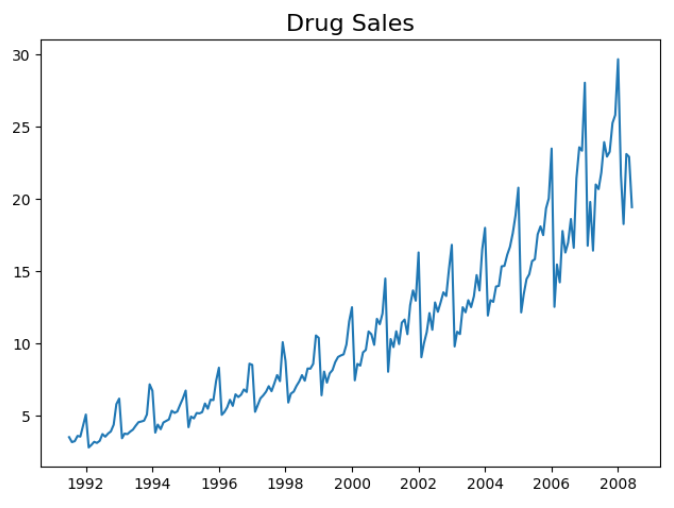
It is possible to remove the trend component from a series. Removing the trend may help to improve the results of a model but not necessarily. Removing a trend may also help to identify relationships between input and target variables. We can use several methods to remove the trend. These methods may include:

1. Subtracting the best fit line from the series.
2. Subtracting the trend component from the series.
3. Other alternatives.

Example 5: De-Trending by Removing Subtracting the Trend

Subtracting the trend component offers one of the cleanest ways to isolate the seasonal and error data for drug sales. This example shows how to isolate the drug sale data so it can be presented without the trend like the plot at the right of Figure 6.

Figure 6: Drug Sales with Trend and Without Trend Data



Here is the code which draws plots for drug sale data with and without the trend component.

|  |
| --- |
| from statsmodels.tsa.seasonal import seasonal\_decompose  import pandas as pd  import matplotlib.pyplot as plt  # Import Data  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "drugSales.csv"  df = pd.read\_csv(PATH + FILE, parse\_dates=['date'], index\_col='date')  tseries = seasonal\_decompose(df['value'], model='additive',  extrapolate\_trend='freq')  plt.plot(df['value'])  plt.title("Drug Sales", fontsize=16)  plt.show()  detrended = df['value'] - tseries.trend  plt.plot(detrended)  plt.title('Drug Sales After Subtracting Trend', fontsize=16)  plt.show() |

Exercise 3 (3 marks)

Plot the airline passenger data set output before and after the trend data has been removed.

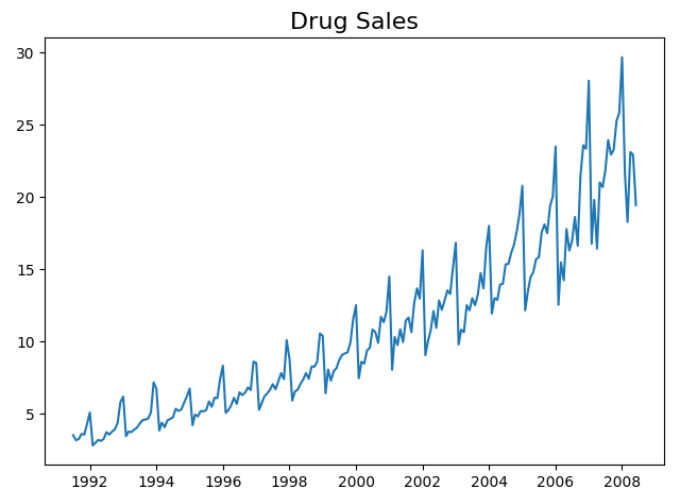
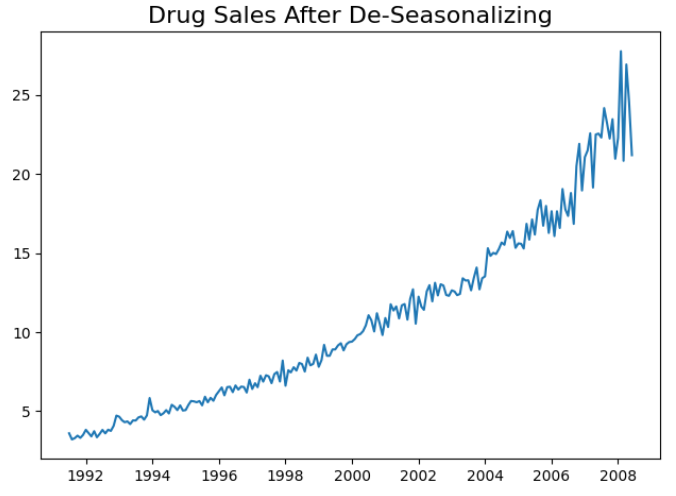
|  |
| --- |
|  |

## De-Seasonalizing Data

As you might expect there are several ways to remove seasonal information from a series. Removing the seasonal component may help to visualize the performance of inputs and outputs along with trend with random environmental shocks.

Example 6: De-Seasonalizing

This example shows how to remove seasonal data by dividing the drug sales data by the seasonal component vector. The result shows drug sale data without the seasonal fluctuation.

Here is the code:

|  |
| --- |
| from statsmodels.tsa.seasonal import seasonal\_decompose  import pandas as pd  import matplotlib.pyplot as plt  # Import Data  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = "drugSales.csv"  df = pd.read\_csv(PATH + FILE, parse\_dates=['date'], index\_col='date')  tseries = seasonal\_decompose(df['value'], model='multiplicative',  extrapolate\_trend='freq')  plt.plot(df['value'])  plt.title("Drug Sales", fontsize=16)  plt.show()  deseasonalized = df.value.values / tseries.seasonal  plt.plot(deseasonalized)  plt.title('Drug Sales After De-Seasonalizing', fontsize=16)  plt.show() |

Exercise 4 (3 marks)

Show the airline passenger data before and after seasonal data has been removed.

|  |
| --- |
|  |

# Rolling Moving Averages

The rolling moving average is a naïve and effective technique that can be used in time series forecasting. Rolling moving averages also help to reduce volatility when trying to understand trends, cycles and seasonal components.

The rolling moving average, also called the rolling mean, can be calculated in several ways.

### Trailing Mean

A trailing mean is calculated with an evenly weighted average of sample observations.

Trailing mean =

### Centered Moving Average

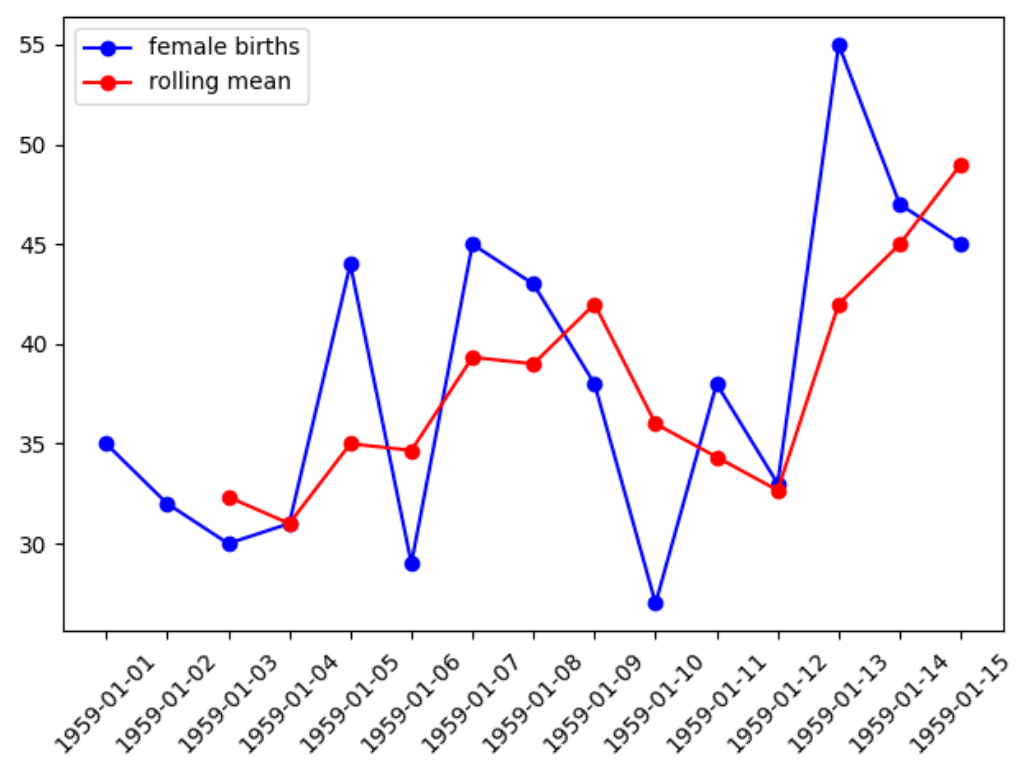
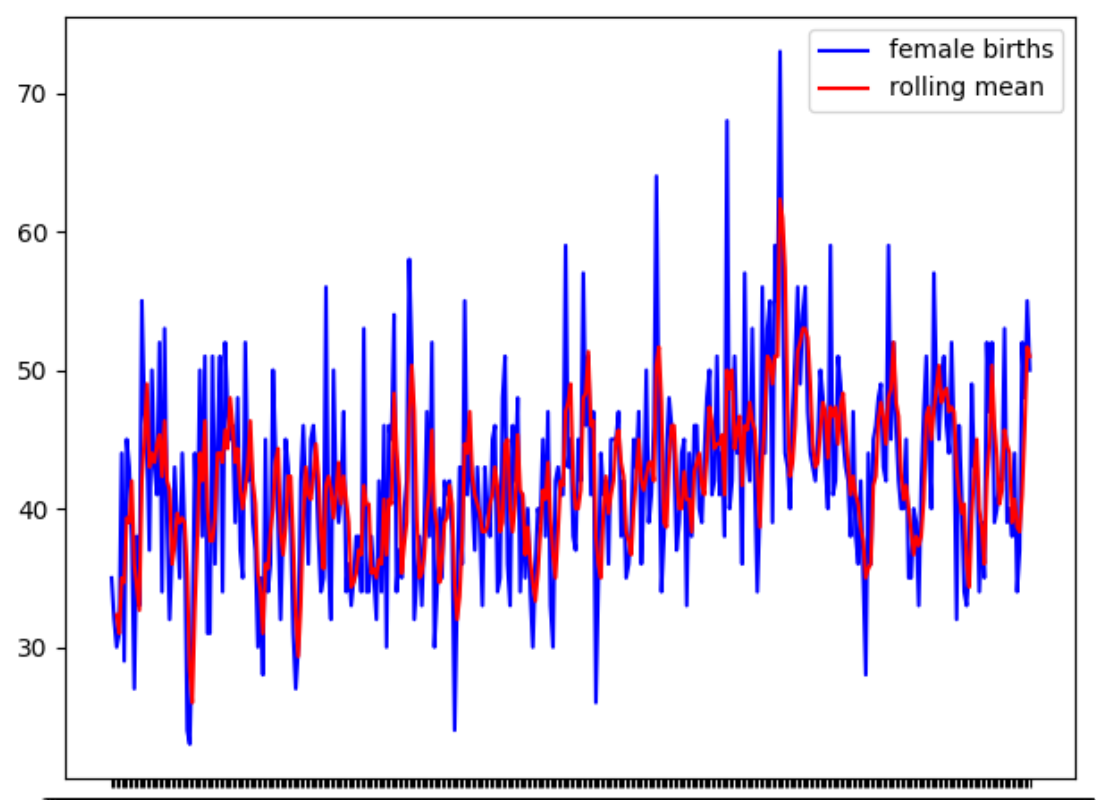
The centered moving average is calculated by averaging an equally weighted number of observations about the current time.

Centered moving average =

Example 7: Rolling Moving Average of Daily Female Births

This example shows how to calculate the trailing moving average of daily female births. Figure 7 shows plots with the rolling mean and actual values. Note that the rolling mean is typically less volatile but it still follows the same trend.

Figure 7: Rolling Mean and Actual Values for all Values and for the 1st 15 days



|  |
| --- |
| from pandas import read\_csv  import matplotlib.pyplot as plt  PATH = "/Users/pm/Desktop/DayDocs/2019\_2020/PythonForDataAnalytics/workingData/"  FILE = 'daily-total-female-births.csv'  series = read\_csv(PATH + FILE, header=0, index\_col=0)  print(series.head())  series.plot(rot=45)  plt.show()  # Calculate rolling moving average 3 steps back.  print("\n\*\*\* Rolling mean")  rolling = series.rolling(window=3)  rolling\_mean = rolling.mean()  print(rolling\_mean.head(5))  # Plot actual and rolling mean values.  plt.plot(series, color='blue', label='female births')  plt.plot(rolling\_mean, color='red', label='rolling mean')  plt.legend()  plt.show() |

Example 8:Manually Calculating the Trailing Moving Average

The moving average in Example 7 is calculated by taking the average of recent target values. Table 2 shows the manual calculations needed to obtain a rolling moving average that is based on the 3 most recent values.

Table 2: Rolling Moving Average Calculation

|  |  |  |
| --- | --- | --- |
| Original Series | obs(t) = (t-2 + t-1 + t)/3 | Rolling moving average. |
| 1959-01-01 35  1959-01-02 32  1959-01-03 30  1959-01-04 31  1959-01-05 44 | =(35+32+30)/3=32.33333  =(32+30+31)/3=31.00000 | 1959-01-01 NaN  1959-01-02 NaN  1959-01-03 32.333333  1959-01-04 31.000000  1959-01-05 35.000000 |

Exercise 5 (2 marks)

Perform the manual calculation to calculate the rolling moving average for January 5, 1959. Show your result here:

|  |
| --- |
| )/3 |

## Weighted Moving Average (Rolling Mean)

Weighted moving averages smooth the fluctuations but they lag behind the actual data.

Trailing mean =

The rolling mean can be calculated with the following code:

|  |
| --- |
| rolling\_mean = df['Close'].rolling(window=20).mean() |

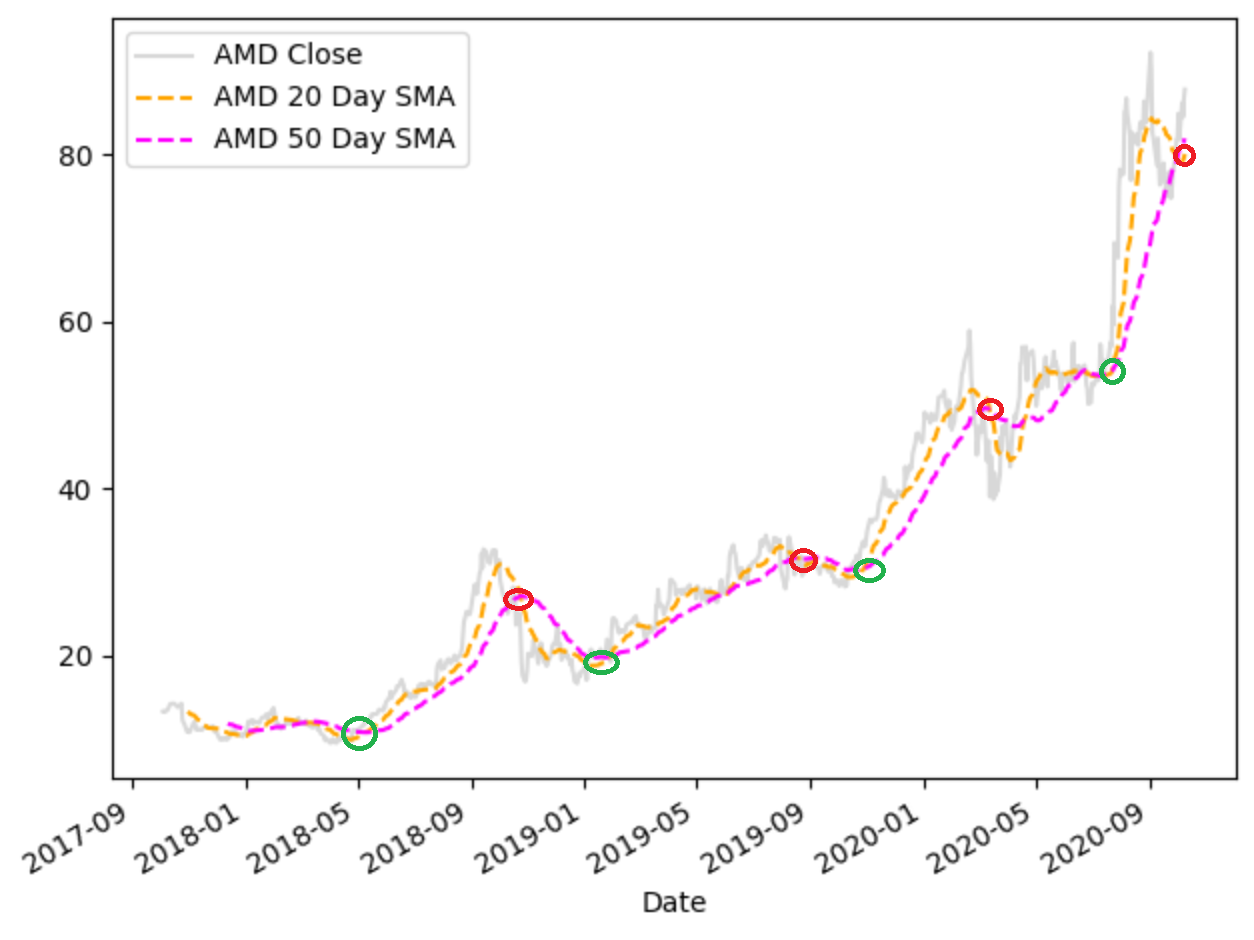
## Using Moving Averages to Buy and Sell Stock

A common stock trading strategy involves using moving averages to predict when to buy or sell. When the short term moving average cross above the long term moving average buy. When the short term moving average falls below the long term moving average sell (refer to Figure 8).

Example 9: Weighted Moving Average

This example shows how a 20-day versus a 50-day simple moving average for AMD stock closing prices. Both moving average plots are noticeably smoother than the actual time closing prices plot. The shorter moving average is a closer match to the actual data since it uses more current data.

Figure 8: Moving Average Trading Strategy



Here is the code:

|  |
| --- |
| 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  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  df = getStock('AMD', 1100)  print(df)  rolling\_mean = df['Close'].rolling(window=20).mean()  rolling\_mean2 = df['Close'].rolling(window=50).mean()  #plt.figure(figsize=(10,30))  df['Close'].plot(label='AMD Close ', color='gray', alpha=0.3)  rolling\_mean.plot(label='AMD 20 Day SMA', style='--', color='orange')  rolling\_mean2.plot(label='AMD 50 Day SMA', style='--',color='magenta')  plt.legend()  plt.show() |

Exercise 6 (1 mark)

The green circle at January 2019 indicates one of the following. Please highlight the correct answer: a) Buy b) Sell

The red circle near September 2019 indicates one of the following. Please highlight the correct answer: a) Buy b) Sell

Exercise 7 (1 mark)

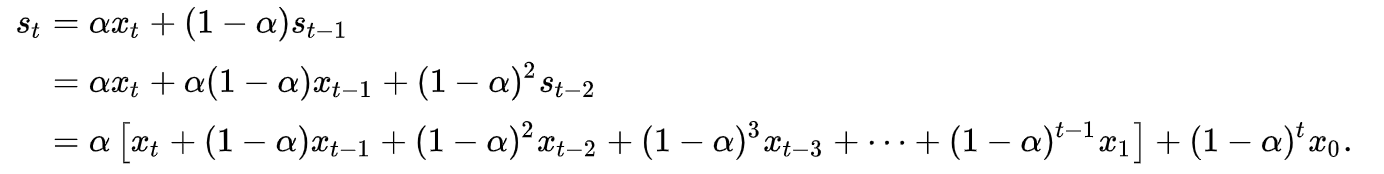
Indicate if the following is true or false.

A buy is indicated when the long-term moving average crosses above the short-term moving average during an upwards stock price trend.

1. True b) False

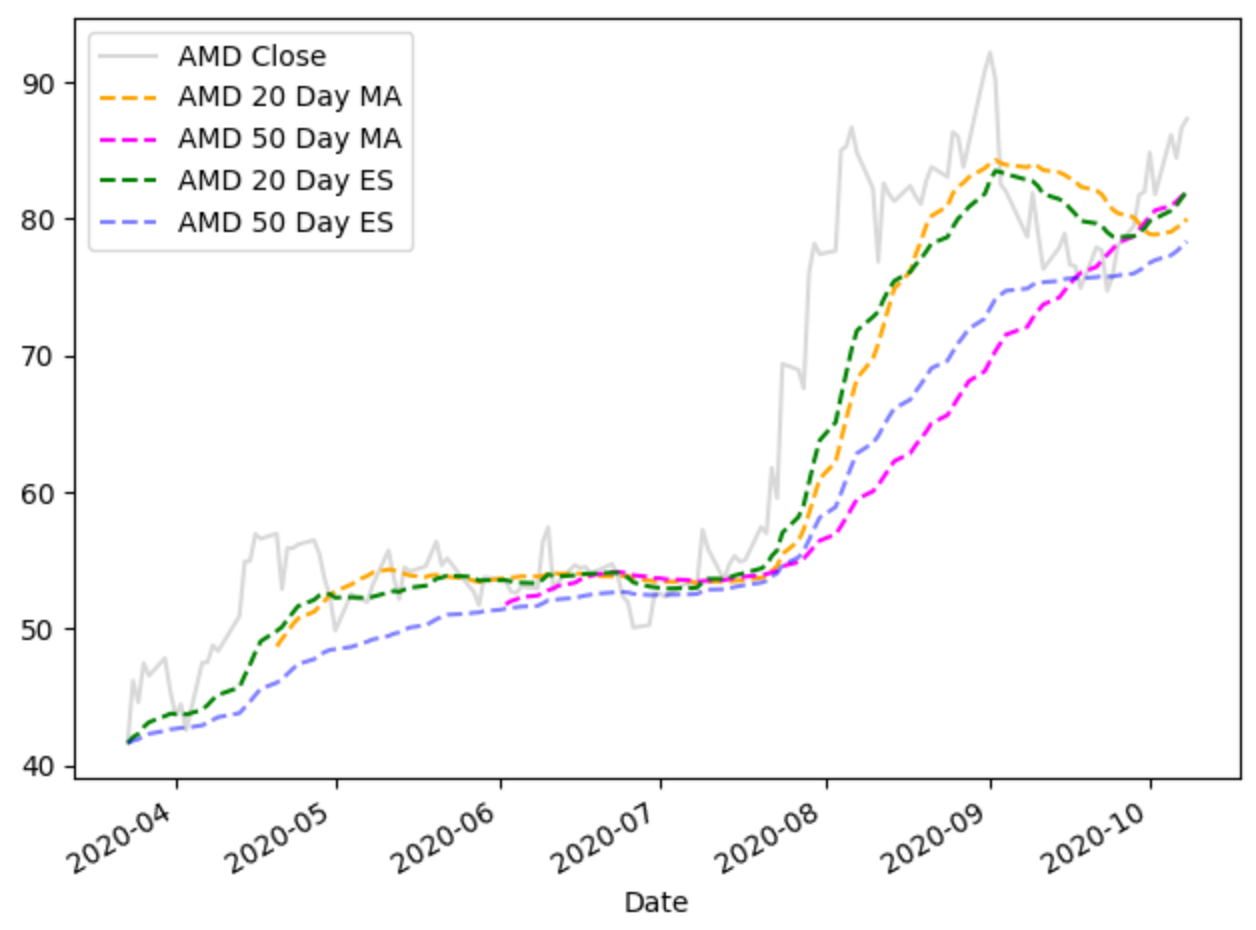
## Exponentially Weighted Moving Average (Exponential Smoothing)

Exponential smoothing provides a slightly better fit to the fluctuations in the time series because it places a higher weight on more recent time steps. Exponential smoothing lags behind the actual data less than the simple moving average.



The **50-day** exponential smoothing line (blue) is better able to represent the actual data fluctuations compared to the **50-day** moving average line (pink). The **20-day** exponential smoothing line (green) is better able to represent the actual data fluctuations compared to the **20-day** moving average line (yellow). Refer to Figure 9.

Figure 9: Moving Average Versus Exponential Smoothing



Exponential smoothing is implemented with the following code:

|  |
| --- |
| exp20 = df['Close'].ewm(span=20, adjust=False).mean() |

Example 10: Exponential Smoothing

This example implements the code that is used to calculate and present the moving averages and exponentially smoothed data that is plotted in Figure 9.

|  |
| --- |
| 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  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  df = getStock('AMD', 200)  # Calculating the moving averages.  rolling\_mean = df['Close'].rolling(window=20).mean()  rolling\_mean2 = df['Close'].rolling(window=50).mean()  # Calculate the exponentially smoothed series.  exp20 = df['Close'].ewm(span=20, adjust=False).mean()  exp50 = df['Close'].ewm(span=50, adjust=False).mean()  #plt.figure(figsize=(10,30))  df['Close'].plot(label='AMD Close ', color='gray', alpha=0.3)  rolling\_mean.plot(label='AMD 20 Day MA', style='--', color='orange')  rolling\_mean2.plot(label='AMD 50 Day MA', style='--',color='magenta')  exp20.plot(label='AMD 20 Day ES', style='--',color='green')  exp50.plot(label='AMD 50 Day ES', style='--',color='blue', alpha=0.5)  plt.legend()  plt.show() |

Exercise 8 (3 marks)

In the same graph, plot the 50-day moving average for Microsoft stock closing prices. (Microsoft uses stock symbol MSFT). Also plot the 50-day exponentially weighted moving average. Please label the graph properly and adjust the names of your variables as needed as well. Show your plot here:

|  |
| --- |
|  |

Please show your code here:

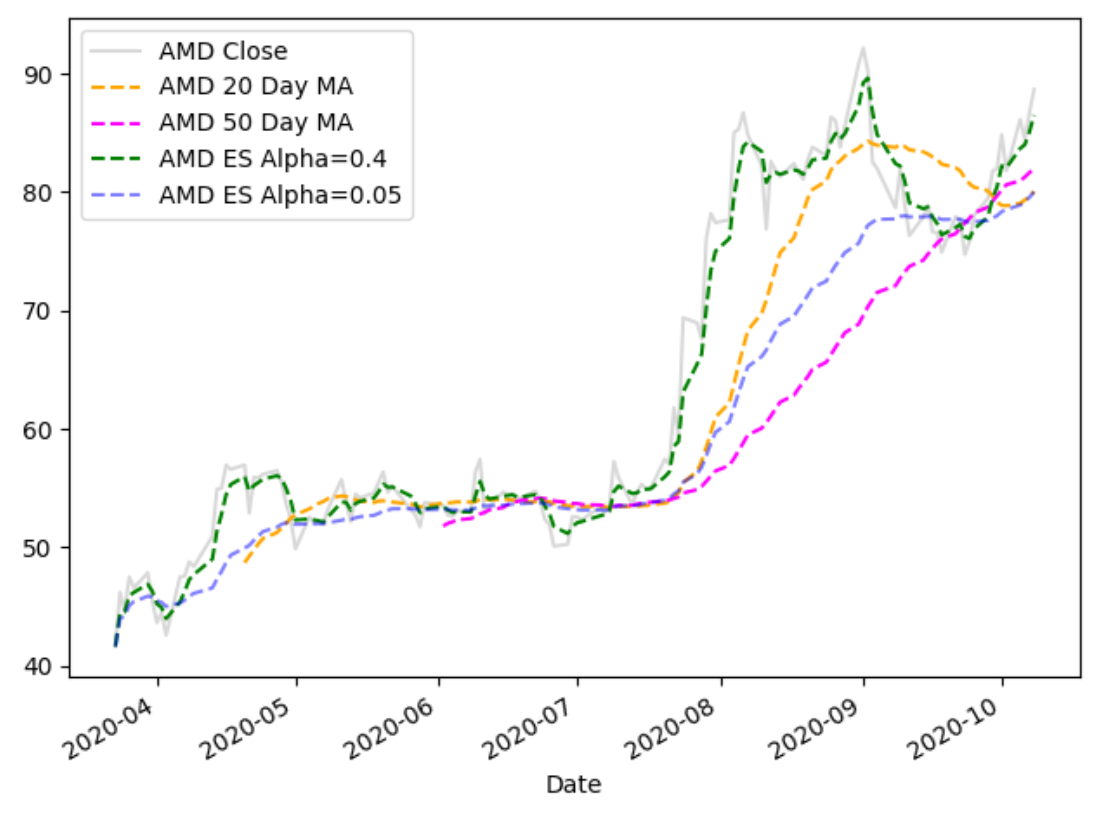
|  |
| --- |
| from pandas\_datareader import data as pdr import yfinance as yfin # Work around until  # pandas\_datareader is fixed. import datetime import matplotlib.pyplot as plt  def getStock(stk, ttlDays):  numDays = int(ttlDays)  # Only gets up until day before during  # trading hours  dt = datetime.date.today()  # 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  df = getStock('MSFT', 200)  # Calculating the moving averages. rolling\_mean = df['Close'].rolling(window=20).mean() rolling\_mean2 = df['Close'].rolling(window=50).mean()  # Calculate the exponentially smoothed series. exp20 = df['Close'].ewm(span=20, adjust=False).mean() exp50 = df['Close'].ewm(span=50, adjust=False).mean()  #plt.figure(figsize=(10,30)) df['Close'].plot(label='MSFT Close ', color='gray', alpha=0.3)  # magenta for rolling average smoothing rolling\_mean2.plot(label='MSFT 50 Day MA', style='--',color='magenta')  #blue is exponentional smoothing exp50.plot(label='MSFT 50 Day ES', style='--',color='blue', alpha=0.5) plt.legend() plt.show() |

Is the exponentially smoothed data more quick to respond to rises and falls than the simple moving average? Explain why or why not.

|  |
| --- |
| Exponential smoothing is closer to the actual data because it places higher weight on more recent data points. |

Example 11: Adjusting the Alpha Value

With exponential smoothing, instead of specifying days you can also set the alpha value to assign a weight to the first time-lag. Higher alpha values assign more importance to the more current time lags.



Starting with Example 10, replace the code which calculates the exponentially weighted moving average with this code:

|  |
| --- |
| exp20 = df['Close'].ewm(alpha=0.4).mean()  exp50 = df['Close'].ewm(alpha=0.05).mean() |

Exercise 9 (1 mark)

Based on your observation of Example 11, state if the values that are modified with an exponentially weighted moving average when using a larger alpha value are more responsive to fluctuations in the actual closing prices compared to the exponentially weighted data that uses a smaller alpha value. Clearly explain why or why not.

|  |
| --- |
| When using a larger alpha, the line is much closer to the actual data points than using a smaller alpha. Higher alpha values will increase the sensitivity and higher importance to more current time lags. |