Time Series Analysis using Python

Prelude

**Concepts you must know before doing this course**

* Usage of vector, matrix, and loop while solving problems
* Knowledge of null and alternate hypothesis as a concept
* Usage of simple linear regression to solve problems
* Knowledge of correlation between two variables as a concept

**Software requirement**

* Anaconda v4.4.0 or higher (Python v3.x)

**Demos and Datasets used in this course**

* [Click here](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0126051802648821761305_shared/web-hosted/assets/TimeSeriesAnalysisusingPython1612787183638.zip)to download the Demos and Datasets used in the course.

**At the end of this course, you will be able to:**

* Differentiate between time series data and non-time series data and pick an appropriate method for forecasting
* Identify different components of time series data to build a forecasting model
* Compare the additive and multiplicative decomposition models which will help in forecasting
* Forecast the values of time series data by using different techniques such as Moving Average, Exponential Smoothing, Seasonal Indexes & Autoregressive Integrated Moving Average Models

Introduction

Decision making is the key to success in the fast-paced world that we live in. We can use data and technology to analyze and forecast. This makes the task of decision making easier. For example:

* Prediction of a stock's price could help an investor make good decisions.
* Prediction of rainfall can help the farmers decide about the type of crops.

Such predictions are possible by the analysis of data using various methods. One such method for analyzing time based data is Time Series Analysis. It helps us get insights from the past data patterns. The insights can be used for quick decision making. Good decision making is key to corporate competitiveness.

In the above scenario, if we know the historical stock prices, we might find interesting patterns in this data by using some statistical techniques. We can predict the future stock price by using such patterns. In this course, we shall learn how we can make such predictions.

Scenarios

Business Scenario’s

Let us consider some business scenarios where data analysis can be advantageous.

**Scenario 1: Aviation**

A company called Cyrus Aviation needs to decide how many flights they should schedule in order to meet the demands of the varying number of customers they cater to.

Cyrus Aviation has the following data:

* number of passengers
* month and year of travel

Cyrus Aviation is looking for seasonal patterns in data. It wants to know about the lean periods where it can offer discounts.

**Scenario 2: Stock Market**

An investor wants to invest in a stock market by buying some shares of Cyrus Aviation, but he is not sure whether it will give him a significant profit. If he makes a wrong estimate, then he might incur a loss.

The investor is looking for patterns in the company’s performance, these patterns can be used to decide if the stock price will go up.

**Scenario 3: Real Estate**

Bob, a real estate builder wants to build houses in a particular area. However, he wants to analyze the demand because higher demand would earn him a profit but lower demand would result in a loss.

Bob is looking for a way to predict the demand, based on which he can decide the number of houses to be built.

In all these scenarios, analyzing the data will help the organizations in making better decisions regarding the future.

**Data Analysis**

In order to perform efficient data analysis, we need to be able to understand the type of data we are dealing with and interpret it accordingly. This will enhance the accuracy of prediction. Let us look at some more information that we may want to derive from Cyrus Aviation:

1. We may want to understand whether Cyrus Aviation has had an increasing customer base by analyzing the past data.
2. We may want to forecast the number of passengers taking the flight in a particular month such as January.
3. We may want to predict the month in which the least/maximum number of passengers will board the international flight.

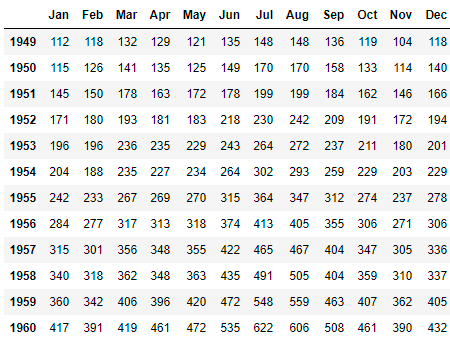
Cyrus Aviation can use this information to set up marketing strategies and vary flight schedules in such a way that maximizes their revenue. We can derive this information using many statistical analysis methods such as Association Analysis, Time Series Analysis and, Regression Analysis. The choice of method depends largely on the type of data. Let us understand the type of data on which Time Series Analysis can be applied.

Aviation Scenario

Let us consider Cyrus Aviation.

Cyrus Aviation, as discussed in scenario 1 has an AirPassengers dataset. AirPassengers is a monthly data which shows the number of passengers taking international flights from 1949 to 1960.

A snapshot of the AirPassengers data is given below.



Types of Data

Aviation Scenario

Let's try to analyze the AirPassengers [dataset](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0126051846164643841294_shared/web-hosted/assets/AirPassengers1612202105441.zip) using traditional methods to predict the number of people traveling in a given month. We shall find the average number of people traveling in a month(mean) from the year 1949 to 1959. Ideally, this should give us a good idea of how to schedule the flights for 1960.

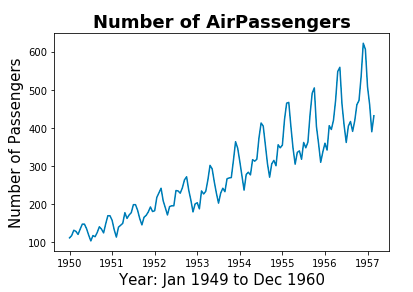
Let us now read the data and find the summary:

1. *# Importing necessary libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. import seaborn as sns
6. import warnings
7. warnings.filterwarnings("ignore")
8. *# In case if you are working in a notebook*
9. %matplotlib inline
10. *# Reading the data*
11. AirPassengers = pd.read\_csv('AirPassengers.csv', parse\_dates=[0])#Marking first column to datetype format
12. *# Summary of the data*
13. AirPassengers.describe().T

We can see that on an average of 280 passengers travel each month. If Cyrus Aviation were to schedule flights based on this analysis, they would run into a shortage of supply throughout 1960.

Let us now visualize the data:

1. *# Importing necessary libraries*
2. import matplotlib.pyplot as plt
3. *# Feching years*
4. year = AirPassengers['Travel date'].dt.year
5. *# Visualizing the time series by plotting number of passengers versus time*
6. fig, ax = plt.subplots()
7. ax.plot(AirPassengers.Passengers)
8. ax.set\_title('Number of AirPassengers', weight='bold', fontsize=18) *# Title*
9. ax.set\_ylabel('Number of Passengers', fontsize=15)
10. ax.set\_xlabel('Year: Jan 1949 to Dec 1960', fontsize=15)
11. ax.set\_xticklabels(np.unique(year))
12. fig.show()

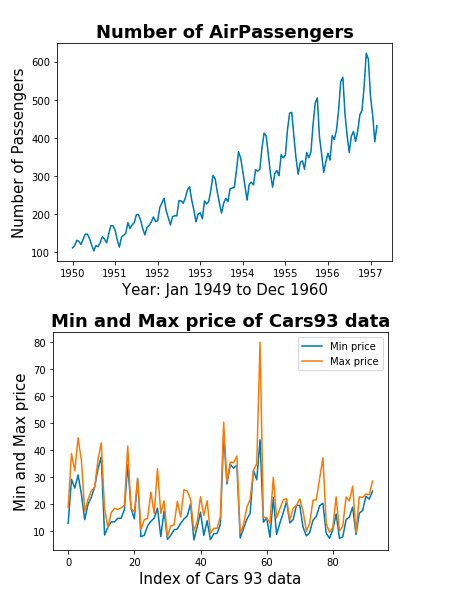


Our prediction is inaccurate, simply because we have not considered that this dataset may have some pattern that may help us come up with more specific predictions.

Let's study the properties of time-sequenced data. We will compare it with Cars93 [data](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0126051846164643841294_shared/web-hosted/assets/Cars931612202333548.zip) to understand the differences.

Comparing AirPassengers and Cars93 datasets

Since the Airpassengers data contains the number of passengers over time, it is known as Time Series Data. On the other hand, the cars dataset contains 27 parameters all presumed to be observed all at the same time, therefore the cars data is known as Identically and Independently Distributed data (IID).



As the two datasets are of different types, the prediction techniques used for Cars93 cannot be used for AirPassengers and vice versa.

The code of the above-shown output is as follows:

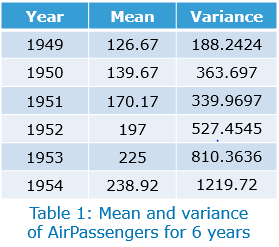
1. *# Importing necessary libraries*
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. *# Reading cars data*
5. cars93 = pd.read\_csv('Cars93.csv')
6. *# Comparing minimum and maximum price of car*
7. plt.plot(cars93["Min.Price"], label='Min price')
8. plt.plot(cars93["Max.Price"], label='Max price')
9. plt.legend()
10. plt.xlabel('Index of Cars 93 data', fontsize=15)
11. plt.ylabel('Min and Max price', fontsize=15)
12. plt.title('Min and Max price of Cars93 data', weight='bold', fontsize=18)
13. plt.show()

\* for simplicity we have considered only 2 parameters of Cars93

Time Series Data vs IID

Let's look at how Time Series data differs from IID data.

* In time series, data is dependent on the previous values.
* In time series data, mean and variance might change over a period of time, this is known as heteroscedasticity. If it does not vary, the data is said to have homoscedasticity.



We can observe from Table 1 that the mean of the AirPassengers(From January 1949 to December 1954) data is continuously increasing and its variance is also changing each year. The lowest value of the variance is 188.24 and the highest value is 1219.72. This is an example of heteroscedasticity.

Time series data may exhibit conditional heteroscedasticity, where the mean might be constant but its variance is varying.

* In time series data, one variable is observed at different time intervals, whereas in IID, different variables are assumed to be observed at the same time or in the same time interval.

For example, the data in AirPassengers is observed over different months, whereas in Cars93, the data is assumed to be observed at the same time.

**Conclusion**

We have now understood the difference between data captured at one point of time, IID data (e.g. Cars93) and data captured over a period of time, Time Series data (e.g. AirPassengers). Once we are able to identify which category a particular dataset belongs to, we can choose an appropriate method to analyze it and make accurate predictions.

Time Series Data

Convert Data to Time Series

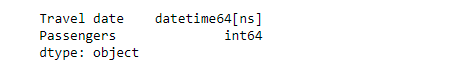
In order to create a time series object in python, we can use the methods provided in Pandas library. Many times, the date attribute may be present with object datatype, or you may need to create a dummy date with a certain frequency. Here, we will learn about implementing the basic operations on time data.

Consider reading the AirPassengers dataset, say, with no date time parsing during reading. Once read, we will learn how to change the data type of Date attribute as well as its frequency. Frequency is the number of observations to be considered at a time.

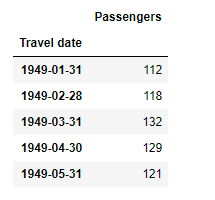
1. *# Importing necessary library*
2. import pandas as pd
3. *# Reading dataset without parsing*
4. AirPassengers = pd.read\_csv('AirPassengers.csv')
5. *# Analysing the datatype of columns*
6. AirPassengers.dtypes



1. *# Importing necessary library*
2. import pandas as pd
3. *# Changing the datatype of Travel date*
4. AirPassengers['Travel date'] = pd.to\_datetime(AirPassengers['Travel date'])
5. AirPassengers.dtypes



1. *# Making Travel date as Index and dropping current column to change the frequency*
2. AirPassengers.index = AirPassengers['Travel date']
3. AirPassengers.drop(['Travel date'], axis=1, inplace=True)
4. *# Changing frequency from daily to month end and getting the mean passenger for each month*
5. AirPassengers.resample('M').mean().head()
6. *# Note - M is one of the DateOffset object provided by Pandas.*



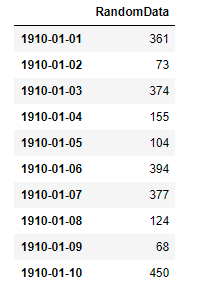
**Creating a date time range with a particular frequency:**

The below python script generates date from January 1st, 1910 to January 10th, 1910 and a random uniformly distributed numeric value.

1. *#importing necessary libraries*
2. import numpy as np
3. *# Generating 10 random values*
4. np.random.seed(42)
5. val = np.arange(500)
6. np.random.shuffle(val)
7. val = val[:10]
8. val



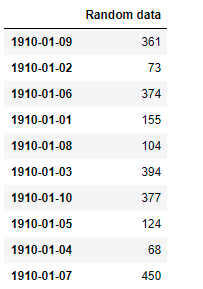
1. *# importing necessary library*
2. import pandas as pd
3. *#creating dataframe with dummy values*
4. dummy\_ = pd.DataFrame(val, index=pd.date\_range(start='1/1/1910', end='1/10/1910', freq='D'),
5. *# DateOffset 'D' refers to daily*
6. columns=['RandomData'])
7. dummy\_



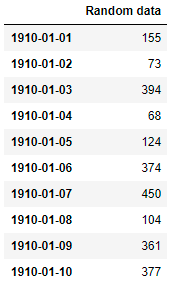
**Converting an unsorted date into a sorted date:**

Let us try to generate random dates between January 1st, 1910 and January 10th, 1910 and assign random data generated for the previous case and later try to rearrange the data in sorted date manner.

1. *#importing necessary libraries*
2. import numpy as np
3. import pandas as pd
4. *# Creation of unordered date*
5. np.random.seed(42)
6. date = pd.date\_range(start='1/1/1910', end='1/10/1910', freq='D')
7. *# generating unordered index*
8. idx\_ = np.arange(10)
9. np.random.shuffle(idx\_)
10. unordered\_data = pd.DataFrame(val, index=date[idx\_], columns=['Random data'])
11. unordered\_data



1. *# sorting dataframe based on date index*
2. unordered\_data.sort\_index(inplace=True)
3. unordered\_data



Times Series Data: Examples

Let us now have a look at a few examples of Time Series data:

* **Stock Exchange Data:** The dataset consists of the price at which the stock market opens and closes on a given day. It also contains the highest price and lowest price of a share on that day.
* **Air Quality Data Set:**  Contains the responses of a gas multi-sensor device deployed on the field in an Italian city. Average of the hourly responses are recorded along with gas references from a certified analyzer.
* **Daily and Sports Activities Data Set:** The dataset comprises motion sensor data of 19 daily and sports activities each performed by 8 subjects in their own style for 5 minutes. Five Xsens MTx units are used on the torso, arms, and legs.
* **Online Retail Data Set:** This is a transnational data set that contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail.

All the above examples have a time component associated with them. You can experiment with the datasets available on the UCI Machine learning repository to understand Time Series data better.

Time Series: Definition

Now that we can identify time series data, let us see some formal definitions.

**Time series**is defined as 'a series of data points listed (or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time.'

Time series data is represented by:

**yt= y(t-1)+ εt**

where 'yt' is the observed value at time 't'

'εt' is the error at time 't'

**Time series analysis**is defined as 'methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data. Time series forecasting is the use of a model to predict future values based on previously observed values.' (Source: Wikipedia)

Time series data must be a stationary series to perform time series analysis. Let us understand what a stationary series is.

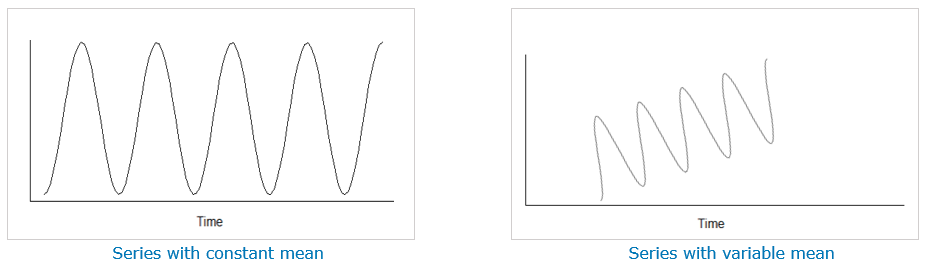
A stationary series exhibits the following characteristics:

* mean is constant
* variance is constant
* co-variance is constant

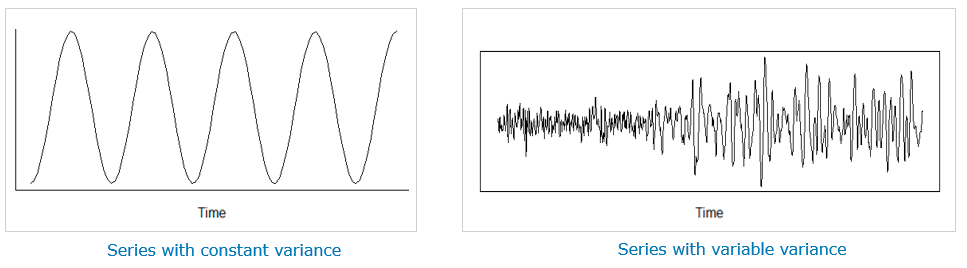
Performing time series analysis on a non-stationary series will result in inaccurate predictions. Let us see few examples of stationary and non-stationary series.

Stationary Series

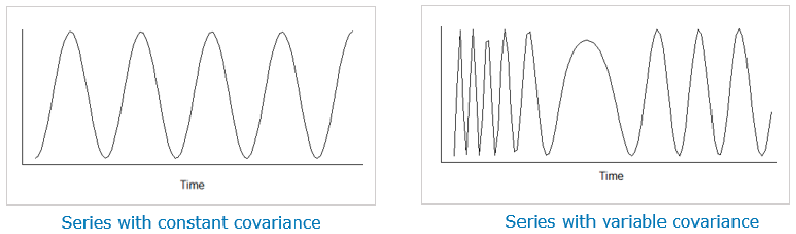
In the first figure, the mean of the series is parallel to the x-axis whereas in the second the mean is upward sloping and increasing with respect to time. Therefore, the first figure is a stationary time series, on the other hand, the second figure is non-stationary.



In the third figure, the summation of the difference between data point and mean is constant whereas in the fourth figure the difference between the data point and mean does not stabilize for a long time so that summation could not be computed. Therefore the third figure is a stationary time series whereas the fourth figure is non-stationary.



In the fifth figure, the covariance between data point observed at a time 't' and data point observed at a time 't+m' is constant whereas, in the sixth figure, it varies with time as the spread is very close in the starting whereas it is quite far in the last. Therefore fifth is a stationary series and sixth is non-stationary.



**Conclusion**

We have learnt the difference between the stationary and non-stationary series. If the given series is non-stationary then we need to convert it to stationary in order to perform time series analysis. We shall learn how to convert non-stationary data into stationary data in the later part of this course.

Time Series Analysis – Exercise

**Problem Statement:**

Create a time series data on a monthly basis starting from April 2007 to March 2012 considering random numbers as values with seed value as 42.

## Patterns in Time Series Data

Now that we know that the AirPassengers data is a time series data, let us try to make a prediction using this data. For instance, let's understand whether Cyrus Aviation has had an increasing customer base over time by analyzing the past data.

Analyzing the past data can be done by keeping in mind the following factors:

* **Trend**: shows a long term increase or decrease in data
* **Seasonal**: when the data is affected by seasonal factors
* **Cyclic**: data shows fluctuations across any period of time
* **Random**: If the above 3 components are missing in a data

The time series data may be a combination of one or more of these factors. Understanding these factors will help analysts identify patterns in the data. These components are extracted by using the statsmodels package in python, which is explained later in this course.

## Trends

A long term increase or decrease found in data is known as a trend. A trend may change its direction from upward to downward and vice versa. The direction of the trend can help us draw conclusions from the data.

In the AirPassengers dataset, the graph shows that between 1949 and 1960 the number of passengers was increasing. This is an upward trend.

## 

The code of the above-shown output is as follows:

1. *#Importing libraries*
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. *# Loading statsmodel method to perform decomposition*
5. from statsmodels.tsa.seasonal import seasonal\_decompose
6. *# Another way to load a time series data by defining a date format*
7. dateparse = lambda dates: pd.datetime.strptime(dates, "%m/%d/%Y")
8. AirPassenger = pd.read\_csv('AirPassengers.csv', index\_col='Travel date', date\_parser=dateparse)
9. *# Visualizing trend of Airpassengers data*
10. plt.plot(seasonal\_decompose(AirPassengers).trend)
11. plt.xlabel('Year', fontsize=15)
12. plt.ylabel('Number of Passengers', fontsize=15)
13. plt.title('Upward Trend in AirPassengers data', weight='bold', fontsize=18)
14. plt.show()

Thus, we can establish that the customer base for Cyrus Aviation was steadily increasing.

Let's use a small sample of Google stock [dataset](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0126051850322984961303_shared/web-hosted/assets/Googlestock1612785067248.zip) to visualize a mixed trend i.e. a data having both upward as well as downward trends.

## 

The code of the above-shown output is as follows:

1. *#Importing libraries*
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. *# Reading file*
5. dateparse = lambda dates: pd.datetime.strptime(dates, '%m/%d/%Y')
6. GOOG\_stock = pd.read\_csv('Google\_stock.csv', index\_col='Date', date\_parser=dateparse)
7. *# Visualizing upward and downward trends*
8. plt.figure(figsize=(12, 6))
9. ax1 = plt.subplot2grid((1, 2), (0,0))
10. ax1.plot(GOOG\_stock.index.day, *# Labeling only days*
11. GOOG\_stock.Volume)
12. ax1.set\_xlabel('Days', fontsize=15)
13. ax1.set\_ylabel('Volume', fontsize=15)
14. ax1.set\_title('GOOG Stock between July 7th, 2006 \n and July 28th, 2006', weight='bold', fontsize=18)
15. ax2 = plt.subplot2grid((1, 2), (0,1))
16. ax2.plot(GOOG\_stock.index.day, seasonal\_decompose(GOOG\_stock.Volume).trend)
17. ax2.set\_xlabel('Days', fontsize=15)
18. ax2.set\_ylabel('Volume', fontsize=15)
19. ax2.set\_title('Days 12-20: Upward Trend \n Days 20-26: Downward Trend', weight='bold', fontsize=18)
20. plt.tight\_layout()

A season may be identified as a monthly or quarterly period in a year.

AirPassengers: The graph shows that the peak period of flight booking is around the months of July and August. Also, it can be observed that these patterns repeat every year.

## 

It can be concluded that the AirPassenger data has seasonality as well as an upward trend (based on the result from the previous page).

Python script for the seasonal data from January 1949 to December 1960 and seasonal data for 1949 is given below.

1. *#importing libraries*
2. import matplotlib.pyplot as plt
3. *# Visualizing seasonality of Airpassengers data*
4. plt.plot(seasonal\_decompose(AirPassengers).seasonal)
5. plt.xlabel('Jan 1949 to Dec 1960', fontsize=15)
6. plt.ylabel('Number of Passengers', fontsize=15)
7. plt.title('Seasonal', weight='bold', fontsize=18)
8. plt.show()
9. *# Visualizing seasonality of Airpassengers data for year 1949*
10. plt.figure(figsize=(10, 4))
11. plt.plot(seasonal\_decompose(AirPassengers).seasonal.iloc[1:12,:])
12. plt.xlabel('January 1949 to December 1949', fontsize=15)
13. plt.ylabel('Number of Passengers', fontsize=15)
14. plt.title('Seasonal monthly data for 1949', weight='bold', fontsize=18)
15. plt.show()

## 

## 

## Cyclic

Cyclic variations can be identified over a large period of time. Unlike seasonal variations which occur within a year, a cycle may be spread across any period of time. Also, the end of an ongoing cycle is uncertain.

Example: As AirPassengers does not exhibit cyclic variations, let's analyze the hsales [dataset.](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0126051850322984961303_shared/web-hosted/assets/hsales1612247147997.zip)

## 

The code of the above-shown output is as follows:

1. *# Importing libraries*
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. *# Reading data*
5. dateparse = lambda dates: pd.datetime.strptime(dates, '%m/%d/%Y')
6. hsales = pd.read\_csv('hsales.csv', index\_col='date', date\_parser=dateparse)
7. *# Visualizing house sales data*
8. plt.plot(hsales)
9. plt.xlabel('January 1973 to Novermber 1995', fontsize=15)
10. plt.ylabel('Monthly housing sales', fontsize=15)
11. plt.title('house sale data', weight='bold', fontsize=18)
12. plt.show()

In the figure below, red circles show the change of trend from the downward direction to the upward direction, and green circles show the change of trend from the upward direction to downward along with black boxes which depict an upward trend, and the yellow boxes which depict a downward trend.

Capturing cyclic variations is a challenging process, as it occurs over a vast period of time. For example in this graph, we are not sure when in the future the trend will change its direction from downward to upward or vice-versa.

## 

1. *# Importing libraries*
2. import matplotlib.pyplot as plt
3. *# Visualizing monthly sales of new one-family houses*
4. plt.plot(seasonal\_decompose(hsales).trend)
5. plt.xlabel('January 1973 to Novermber 1995', fontsize=15)
6. plt.ylabel('Monthly housing sales', fontsize=15)
7. plt.title('Monthly sales of new one-family houses \n sold in the USA since 1973', weight='bold', fontsize=18)
8. plt.show()

## 

## The data exhibits seasonality within a year but it also has a cyclic behavior of around 7 years.

## Random

When irregular fluctuations occur in the time series data, they can be categorized under Random factors.

For example, let's consider the hsales dataset

If the three components of a time series graph are missing i.e. trends, seasonality, or cyclic then the time series data is termed as random. Time series data does not need to have all the components.

When the other components are removed from the hsales data, we can observe that the remainder appears to be random. Therefore, we can conclude that the hsales data does not have any definite trend, but it has a seasonality component along with a random component.

## 

The code of the above shown output is as follows:

1. *# Importing libraries*
2. import matplotlib.pyplot as plt
3. *# Visualizing residuals of house sales data*
4. plt.plot(seasonal\_decompose(hsales).resid)
5. plt.xlabel('January 1973 to Novermber 1995', fontsize=15)
6. plt.ylabel('Monthly housing sales', fontsize=15)
7. plt.title('house sales data residuals', weight='bold', fontsize=18)
8. plt.show()

# Conclusion

Trend, seasonal, cyclic, and random components help us understand the behavior of time series data. These components help us build a forecasting model. You will learn these forecasting models in the later part of this course.

## Patterns in time series – Exercise

**Problem Statement:**

Plot the trend, seasonality, and residuals of the number of appliances present against the date feature for the [Appliance](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_012717716855816192252_shared/web-hosted/assets/Appliance1612768250128.zip)dataset.

Decomposition Models

We can improve the accuracy of our predictions by breaking down the effect that the factors have on the data. The four factors, trends, seasonal, cyclic, and random can be extracted with the help of the decomposition models. The decomposition models are used to find these four factors in time series data. The model helps us identify patterns in the data and make accurate predictions.

There are basically two decomposition models.

1. **Additive**model is used when seasonal variations are relatively constant over time
2. **Multiplicative**model is used when seasonal variations increase or decrease over time

Additive model

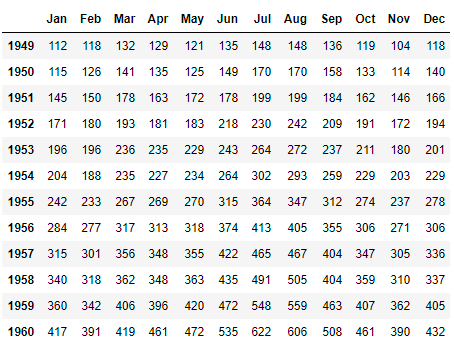
Additive model can be used to understand the seasonal effects on the data when the variations are relatively constant. The time series data may be decomposed as follows:

 Time Series Data = (Seasonal) + (Trends) + (Random)

Before proceeding with the decomposition of the data, we need to define the seasonal span by setting the frequency.

For example, if the data is in terms of years then you can set the year-end frequency by dateoffset 'A'. Similarly, for quarter end you can use 'Q'.

Example: AirPassengers



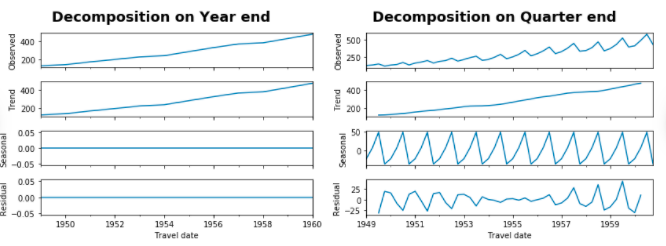
The code of the above shown output is as follows:

1. *# Importing libraries*
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. *# Load fresh data*
5. dateparse = lambda dates: pd.datetime.strptime(dates, '%m/%d/%Y')
6. AirPassenger = pd.read\_csv('AirPassengers.csv', index\_col='Travel date', date\_parser=dateparse)
7. *# Loading statsmodel method to perform decomposition*
8. from statsmodels.tsa.seasonal import seasonal\_decompose
9. *# Getting data corresponding to each month*
10. cols = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
11. AP\_reshaped = pd.DataFrame(AirPassenger.values.reshape(-1, 12),
12. columns=cols, *# Month*
13. index=range(1949, 1961)) *# Year*
14. AP\_reshaped

The seasonal\_decompose command helps to compute the trend, seasonal, and random values of AirPassengers data. The below python script shows the decomposition of AirPassengers data on yearly basis by using additive model.

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. *# Extracting the seasonal values*
5. seasonal\_data = np.round(seasonal\_decompose(AirPassenger, model='additive').seasonal, 2)
6. *# Getting output corresponding to each month*
7. AP\_season = pd.DataFrame(seasonal\_data.values.reshape(-1, 12),
8. columns=cols, *# Month*
9. index=range(1949, 1961)) *# Year*
10. AP\_season

Figures below show the decomposition of data taken on an annual basis (Frequency='A') and a quarterly basis (Frequency='Q').



Both the figures depict an upward trend.

The code of the above shown output is as follows:

1. *# Importing libraries*
2. import matplotlib.pyplot as plt
3. *# Decomposition of Airpassengers data on year end taking mean of monthly values*
4. year\_end = AirPassenger.resample('A').mean() *# Here, DateOffset 'A' represents Year end*
5. seasonal\_decompose(year\_end, model='additive').plot()
6. plt.suptitle('Decomposition on Year end', weight='bold', fontsize=18, y=1.05)
7. plt.show()
8. *# Importing libraries*
9. import matplotlib.pyplot as plt
10. *# Decomposition of Airpassengers data on quarter end taking mean of monthly values*
11. quarter\_end = AirPassenger.resample('Q').mean() *# Here, DateOffset 'Q' represents Quarter end*
12. seasonal\_decompose(quarter\_end, model='additive').plot()
13. plt.suptitle('Decomposition on Quarter end', weight='bold', fontsize=18, y=1.05)
14. plt.show()

Deseasonalized Data

The seasonality factor makes it difficult for us to identify whether the data is depicting an upward trend or a downward trend. Therefore, we need to remove the seasonality factor from our data. The process of removing the seasonality factor from the data is known as de-seasonalization and the resultant values are known as seasonally adjusted values.

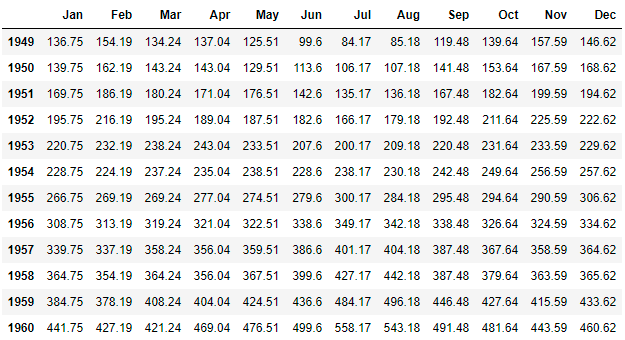
Let's calculate the seasonally adjusted values in the AirPassengers data using the below formula:

  Seasonally adjusted values = Time Series Data - (Seasonal) = (Trends) + (Random)

The number of passengers taking the international flight in December 1959 is 405 and the seasonal effect for December is -28.619949. So, the seasonally adjusted values can be computed as follows:

seasonally adjusted value = (405 - (-28.619949)) = 433.619949

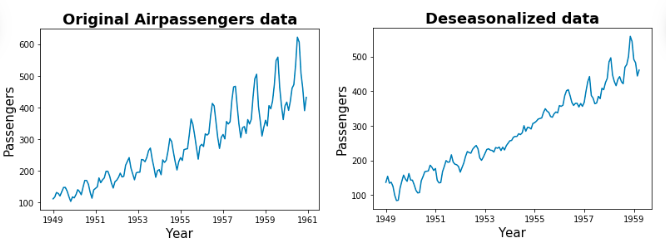
Similarly, the seasonally adjusted value of the AirPassengers data for the yearly basis is computed below.



The code of the above shown output is as follows:

1. *# Deseasonalized data*
2. AP\_deseasonalized = AP\_reshaped - AP\_season
3. AP\_deseasonalized

Given figures show the AirPassengers data and the seasonally adjusted value of AirPassengers data.



These deseasonalized values will be used further in forecasting, which is explained later in this course.

The code of the above shown output is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. *# Melting the data by forming a 1D data to proceed with visualization*
6. plt.plot(AirPassenger)
7. plt.xlabel('Year', fontsize=15)
8. plt.ylabel('Passengers', fontsize=15)
9. plt.title('Original Airpassengers data', weight='bold', fontsize=18)
10. plt.show()
11. *# Melting the data by forming it to a 1D data to proceed with visualization*
12. plt.plot(pd.melt(AP\_deseasonalized.T).value)
13. plt.xticks(np.linspace(0, 140, 6), np.unique(pd.melt(AP\_deseasonalized.T).variable)[::2])
14. plt.xlabel('Year', fontsize=15)
15. plt.ylabel('Passengers', fontsize=15)
16. plt.title('Deseasonalized data', weight='bold', fontsize=18)
17. plt.show()

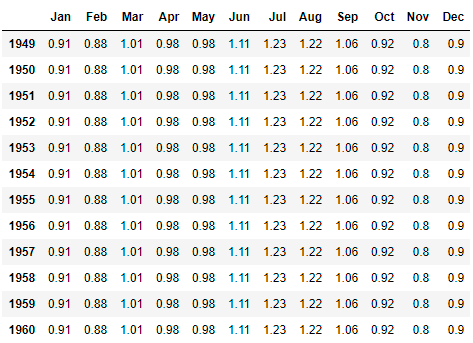
Multiplicative Model

As discussed in additive model we need to compute the seasonally adjusted value for the yearly AirPassengers data.

It is useful when the seasonal variations change over a period of time.

Time Series Data = (Seasonal) \* (Trends) \* (Random)

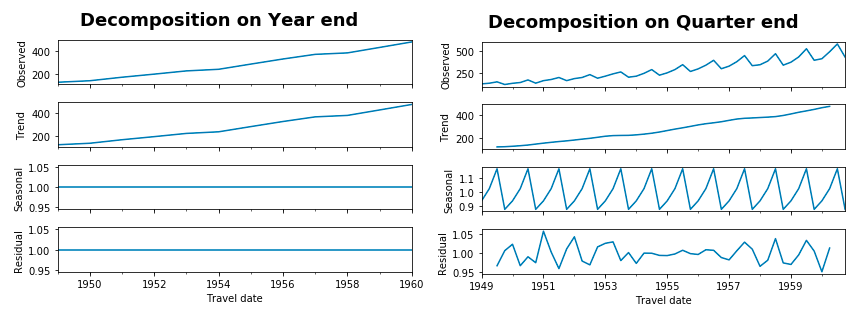
Example: AirPassengers



The code of the above shown output is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. *# Extracting the seasonal values*
5. seasonal\_data\_mult = np.round(seasonal\_decompose(AirPassenger, model='multiplicative').seasonal, 2)
6. *# Getting output corresponding to each month*
7. AP\_season\_mult = pd.DataFrame(seasonal\_data\_mult.values.reshape(-1, 12),
8. columns=cols, *# Month*
9. index=range(1949, 1961)) *# Year*
10. AP\_season\_mult

Below graph shows the yearly and the quarterly decomposition of the AirPassengers. The trend of the quarterly decomposed data has some variations.



The code of the above shown output is as follows:

1. *# Importing libraries*
2. import matplotlib.pyplot as plt
3. *# Decomposition of Airpassengers data on year end taking mean of monthly values*
4. year\_end = AirPassenger.resample('A').mean() *# Here, DateOffset 'A' represents Year end*
5. seasonal\_decompose(year\_end, model='multiplicative').plot()
6. plt.suptitle('Decomposition on Year end', weight='bold', fontsize=18, y=1.05)
7. plt.show()
8. *# Importing libraries*
9. import matplotlib.pyplot as plt
10. *# Decomposition of Airpassengers data on quarter end taking mean of monthly values*
11. quarter\_end = AirPassenger.resample('Q').mean() *# Here, DateOffset 'Q' represents Quarter end*
12. seasonal\_decompose(quarter\_end, model='multiplicative').plot()
13. plt.suptitle('Decomposition on Quarter end', weight='bold', fontsize=18, y=1.05)
14. plt.show()

Deseasonalized Data

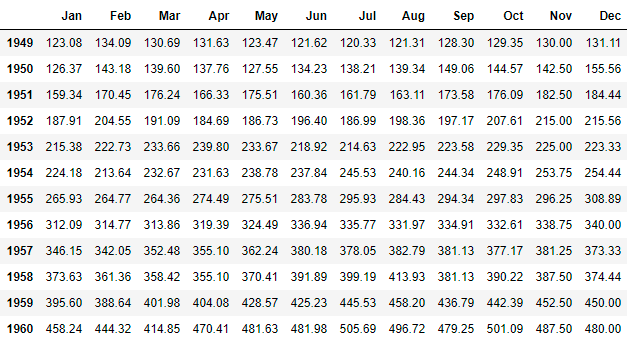
As discussed in the additive model we need to compute the seasonally adjusted value.

The number of passengers taking the international flight in the month of December 1959 is 405 and the seasonal effect for December is 0.9. So, the seasonally adjusted value is calculated by:

Time Series Data = (Seasonal) \* (Trends) \* (Random)

Time Series Data/(Seasonal) = (Trends) \* (Random)

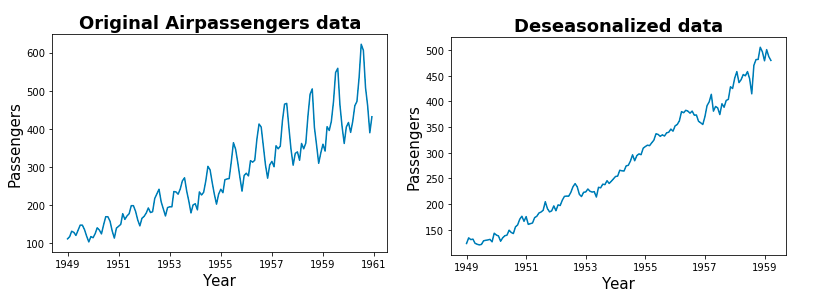
So, the seasonally adjusted value is (405/0.9), i.e. 450.5886.



The code of the above shown output is as follows:

1. *# Importing libraries*
2. import numpy as np
3. *# Multiplicative Deseasonalized data*
4. AP\_deseasonalized\_mult = np.round(AP\_reshaped / AP\_season\_mult, 2)
5. AP\_deseasonalized\_mult

Below figures show the AirPassengers data and the seasonally adjusted value of AirPassengers data.



The code of the above shown output is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. *# Melting the data by forming a 1D data to proceed with visualization*
6. plt.plot(AirPassenger)
7. plt.xlabel('Year', fontsize=15)
8. plt.ylabel('Passengers', fontsize=15)
9. plt.title('Original Airpassengers data', weight='bold', fontsize=18)
10. plt.show()
11. *# Melting the data by forming a 1D data to proceed with visualization*
12. plt.plot(pd.melt(AP\_deseasonalized\_mult.T).value)
13. plt.xticks(np.linspace(0, 140, 6), np.unique(pd.melt(AP\_deseasonalized.T).variable)[::2])
14. plt.xlabel('Year', fontsize=15)
15. plt.ylabel('Passengers', fontsize=15)
16. plt.title('Deseasonalized data', weight='bold', fontsize=18)
17. plt.show()

# Conclusion

We observe that there are more variations in the trend of the additive model as compared to the multiplicative model. By looking at these observations we may conclude that deseasonalized AirPassengers data might give us a clearer trend using the multiplicative model.

These deseasonalized values obtained using multiplicative model will be used for forecasting, which is explained later in this course.

**Decomposition model - Exercise**

## Problem Statement:

Plot the deseasonalized values of AirPassengers data by assigning its frequency to the beginning of the business year. Using both the models (additive as well as multiplicative). The DateOffset to be used is 'BAS'.

## Smoothing Time Series

## We have data from Jan 1949 to December 1960. We want to find out how many people will take the flight in January 1961, this type of forecasting is known as short term forecasting. Here, we want to forecast a value that immediately follows the given data.

## 

## Moving Averages

Here we discuss a smoothing technique moving average, which is used to determine the average of the observed values at each time period.

* Each observation is assigned some weight in order to obtain the weighted average.

**MA =   ( α(0)y(t) + α(1)y(t -1) + … + α(n-1) y(t -n)) / n**

  'α' is the weight assigned to the data

  'yt' is the last day data(latest data)

  'n' is the time period for which we want to calculate the moving average.

 In **simple moving average** each data is assigned equal weights, i.e. α(0) = α(1) = ... = α(n-1)**=**1 or any constant.

**SMA =   ( y(t) + y(t -1) + … + y(t -n)) / n**

Example: The table below shows the simple moving average on a quarterly basis for the AirPassengers from January 1949 to December 1949. Column 3 shows a 4 period moving average and column 4 shows the 8 period moving average.

## 

## 

In the above figures, the blue line represents the actual number of air passengers and the orange line represents the moving average of yearly AirPassengers. We can observe from the 4 period moving average figure that the curve has lots of variations as compared to the latter one.

The code of the above shown output is as follows:

1. *# Importing libraries*
2. import matplotlib.pyplot as plt
3. *# 4 period moving average*
4. plt.plot(AirPassenger, label='Original')
5. plt.plot(AirPassenger.rolling(4, center=False).mean(), label='Moving Average')
6. plt.legend()
7. plt.xlabel('Year', fontsize=15)
8. plt.ylabel('Passengers', fontsize=15)
9. plt.title('Moving Average with period=4', weight='bold', fontsize=18)
10. plt.show()
11. *# 8 period moving average*
12. plt.plot(AirPassenger, label='Original')
13. plt.plot(AirPassenger.rolling(8, center=False).mean(), label='Moving Average')
14. plt.legend()
15. plt.xlabel('Year', fontsize=15)
16. plt.ylabel('Passengers', fontsize=15)
17. plt.title('Moving Average with period=8', weight='bold', fontsize=18)
18. plt.show()

## 

Among the above plots, the first one showcases that there is a sudden fall (abrupt shift) in the series. If there is an abrupt shift (sudden rise and fall) in the series then 'n' is set to a lower value, this is known as under smoothing. In under smoothing, there will be a lot of variations that will disguise the trend. For example, in the 4 period moving average figure, the trend is slightly disguised, due to the variations.

We can observe from another given figure, that there is a gradual increase in the series with fluctuations. If there are fluctuations (irregular rise and fall) in the series then 'n' is set to a large value, this is known as over smoothing. In over smoothing, some of the interesting patterns might be lost. For example, in the 8 period moving average figure, we may not be able to observe some interesting patterns, present in this dataset. Over smoothing might suppress these variations.

The value of 'n' should be set in a manner where both under smoothing and over smoothing can be avoided.

# Conclusion

The smoothed value of December 1949 based on 4 period moving average is 119; this value is considered as the forecasted value for January 1950.

## Centered Moving Average

## As we have seen earlier the trend line is asymmetric in simple moving average. For n period moving average, the first n-1 terms are missing as observed in the previous page plots. We can make the trend line symmetric by placing the average in the middle of the time series, this is known as centered moving average.  This works well for odd period moving average than even period moving average. The table below shows the centered moving average for the year 1949.

## 

The below figures show the 2, 3, 4, and 5 period moving average for the year 1949.

## 

We can observe that 3rd (green) and 5th (purple) period centered moving average are symmetric, on the other hand, 2nd (yellow) and 4th (red) period centered moving average are asymmetric.

The code of the above shown output is as follows:

1. *# Importing libraries*
2. import matplotlib.pyplot as plt
3. *# Centered Moving Averages*
4. plt.figure(figsize=(10, 5))
5. plt.plot(AirPassenger.iloc[1:12], label='Original')
6. plt.plot(AirPassenger.iloc[1:12].rolling(2, center=True).mean(), '--', label='CMA Period=2')
7. plt.plot(AirPassenger.iloc[1:12].rolling(3, center=True).mean(), '--', label='CMA Period=3')
8. plt.plot(AirPassenger.iloc[1:12].rolling(4, center=True).mean(), '--', label='CMA Period=4')
9. plt.plot(AirPassenger.iloc[1:12].rolling(5, center=True).mean(), '--', label='CMA Period=5')
10. plt.legend()
11. plt.xlabel('Year', fontsize=15)
12. plt.ylabel('Passengers', fontsize=15)
13. plt.title('Centered Moving Average with different periods', weight='bold', fontsize=18)
14. plt.show()

# Conclusion

Simple and centered moving average, however, treats the last n observations equally. All observations before that are ignored. In some scenarios, all of the past data must be given gradual weightage. Maximum weightage should be given to the most recent data while minimum weightage should be given to the least recent data or vice-versa. In between maximum and minimum, there is a gradual change in the weightage which are assigned to the observations. This type of smoothing is called exponential smoothing.

# **Exponential Smoothing**

Exponential smoothing works on the original data in order to identify trend patterns.

Exponential smoothing equation:

**S*t*= αy*(t)*+ (1 - α) S*(t-1)***

**S*t*= αy*(t)*+ α(1-α) y*(t-1)*+ α(1-α)*2* y*(t-2)*+ …+(1-α)*(t-1)* y*(1)***(Expanded)

  S*t* is an exponentially smoothed time series at 't', where 't' > 0.

  'y*(t)'*denotes the latest observation in the series at period 't' and 'y*1*'is the first observation.

  'α' is a smoothing constant (alpha).

alpha is a constant between 0 and 1. Usually, alpha is not set to 0 or 1, because if we set alpha to 0 then the smoothed series will depend only on the latest estimation and if we set it to 1 then the smoothed series will depend only on the latest observation.

Exponential smoothing uses the latest observed observation (y*(t)*) and the latest estimation (S*(t-1)*) to predict future observation using (S*(t)*).

Example: The table below shows the exponential smoothening for the AirPassengers data from January 1949 to December 1949.

## 

## Here in this example, we have set alpha to 0.2. Exponential smoothing for March takes the estimated value of March (113.2) and the observed value of March (132). The exponentially smoothed value for December 1949 is 123.40; this value is considered as the forecasted value for January 1950.

1. *# Importing libraries*
2. import numpy as np
3. *# Exponential smoothing is calculated and rounded off to 2 decimal values*
4. np.round(AirPassenger.ewm(alpha=0.2,adjust=False).mean().head(),2)

## 

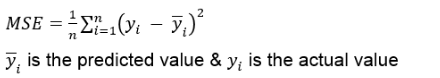
## In this example, we set alpha=0.2. Let us understand how to estimate the optimal value of alpha.

## Estimation of the Smoothing Constant

In unforeseen situations such as natural calamities and economy crashes, there may be a sudden shift in the observed data. By setting alpha to a larger value we give more weight to these observations and less weight to the forecasted values. If we set alpha to a lower value, more weight is given to the recent forecasted value. If there is a sudden change in the series i.e. if y(t) observed to be very high or low, then this change will not be captured in the smoothing. The accuracy of the forecast depends on choosing a proper value of alpha.

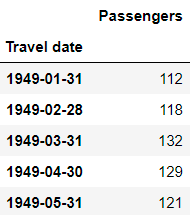
Let's estimate alpha value which can help us predict the number of people who will take the flight in January 1960.

We compare the forecasted values and the observed values for different values of alpha. The value of alpha that gives us the minimum forecasted error, is then chosen. Estimation of alpha is done on the basis of minimum MSE (Mean Square Error) as given below.

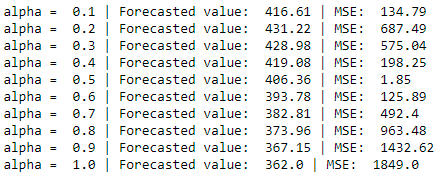


Below, we have computed the MSE with different alpha values for December 1959(405). Alpha may take an infinite number of values between 0 and 1. We have shown a sample of 10 alpha values and forecasted values for December 1959.

1. *# Importing libraries*
2. import pandas as pd
3. *# Setting frequency*
4. AirPassenger = pd.read\_csv('AirPassengers.csv', index\_col='Travel date', date\_parser=dateparse)
5. AirPassenger = AirPassenger.resample('M').mean()
6. AirPassenger.head()



1. *# Importing libraries*
2. import numpy as np
3. *# Simple exponential smoothing*
4. *# Finding alpha value for the forecasted value and mean squared error*
5. from statsmodels.tsa.holtwinters import SimpleExpSmoothing
6. from sklearn.metrics import mean\_squared\_error
7. for i in range(1, 11):
8. model = SimpleExpSmoothing(AirPassenger.iloc[:131]).fit(smoothing\_level=i/10, optimized=False)
9. forecasted\_val = np.round(model.forecast(1)[0], 2)
10. print('alpha = ', i/10, '| Forecasted value: ', forecasted\_val,
11. '| MSE: ', np.round(mean\_squared\_error(np.array(AirPassenger.iloc[131]), np.array([forecasted\_val])), 2))



We can observe from the above results that alpha 0.1 has an MSE value of 134.79 and for alpha 1 it is 1849. We can choose the value of alpha which has minimum MSE. In this example, we can set alpha to 0.5, since it has the minimum MSE value of 1.85 for the AirPassengers data. But this value of alpha may not be optimal - in this example, we have considered alpha up to 1 decimal place. Since alpha can take infinite values like 0.52442 or 0.5342, we are not sure if 0.5 is optimal. To find an optimal value of alpha we can use SciPy library to get optimal results.

1. *# Importing libraries*
2. import numpy as np
3. import scipy as sp
4. *# Function to find optimum value of alpha*
5. def optimum\_alpha(x):
6. model = SimpleExpSmoothing(AirPassenger.iloc[:131]).fit(smoothing\_level=x, optimized = False)
7. forecasted\_val = np.round(model.forecast(1)[0], 2)
8. mse = np.round(mean\_squared\_error(np.array(AirPassenger.iloc[131]), np.array([forecasted\_val])), 2)
9. print('alpha: ', np.round(x[0], 5), 'MSE: ', mse)
10. return mse
11. optimum\_alpha\_result = sp.optimize.fmin(optimum\_alpha, x0=1)
12. if optimum\_alpha\_result < 0:
13. optimum\_alpha\_result = 0.001 *# Least value, you can perform further optimization to improve it*
14. optimum\_alpha\_result

 Alpha value suggested by SciPy is 0.5109375 and MSE for this optimal value is 0, which is smaller than 1.85 (alpha=0.5).

Now, we can build the forecast value using this optimal alpha value.

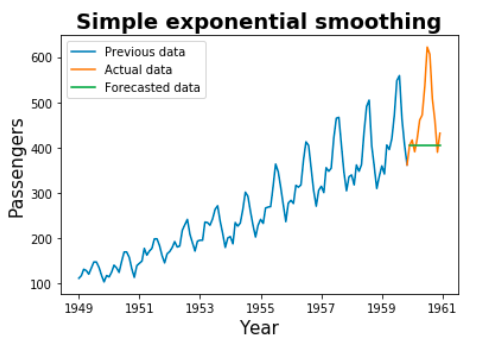
1. *# Importing libraries*
2. import numpy as np
3. *# Creating model using optimum alpha value*
4. model = SimpleExpSmoothing(AirPassenger.iloc[:131]).fit(smoothing\_level=optimum\_alpha\_result, optimized = False)
5. forecasted\_val = np.round(model.forecast(1)[0], 2)
6. mse = np.round(mean\_squared\_error(np.array(AirPassenger.iloc[131]), np.array([forecasted\_val])), 2)
7. print('Alpha: ', optimum\_alpha\_result[0],
8. '\nActual value: ', AirPassenger.iloc[131][0],
9. '\nForecasted value: ', np.round(forecasted\_val).astype(int),
10. '\nMean Squared Error: ', mse)

Exponential Smoothing

Let us forecast the values for the next year using the optimal model.

1. *# Importing libraries*
2. import numpy as np
3. *# Forecasting values for next year*
4. forecasted\_data=np.round(model.forecast(13)).astype(int)
5. forecasted\_data

The output gives the point forecast. Below figure shows the short term prediction by using simple exponential smoothing.



The code of the above shown graph is as follows:

1. *# Importing libraries*
2. import matplotlib.pyplot as plt
3. *# Visualizing the forecasted value*
4. plt.plot(AirPassenger.iloc[:131], label='Previous data')
5. plt.plot(AirPassenger.iloc[130:], label='Actual data')
6. plt.plot(forecasted\_data, label='Forecasted data')
7. plt.xlabel('Year', fontsize=15)
8. plt.ylabel('Passengers', fontsize=15)
9. plt.title('Simple exponential smoothing', weight='bold', fontsize=18)
10. plt.legend()
11. plt.show()

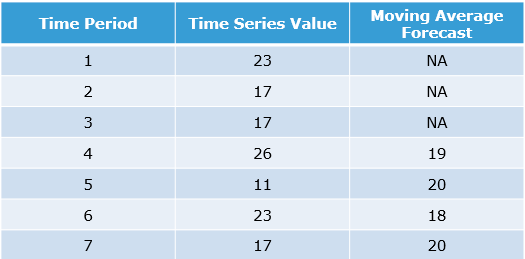
# Conclusion

Smoothing techniques can be utilized for short term predictions. In order to perform long term predictions, like forecasting values for 1 or 2 years, we need to use advanced methods, which we will see later in this course.

# **Moving Average - Exercise**

## Problem Statement:

For the following time series, you are given the moving average forecast. Compute the mean squared error.



Forecasting

## Models for Forecasting

## Forecasting

Using short term forecasting, we were able to forecast only next month's data. Now, let us try to forecast more than one value.

Commonly used models for forecasting are as follows:

* Seasonal Indexing
* Autoregressive
* Moving Average
* Autoregressive moving average
* Autoregressive integrated moving average

Let's look at these models in detail.

## Seasonal indexing

Seasonal indexing model can be used when we have to predict values based on seasonal data. Seasonality is the pattern that occurs within a year. We can recall that AirPassengers data had a seasonality component associated with it.

Considering the AirPassengers Data till the year 1959(yearly data: frequency = 12) assuming it to be a multiplicative model as we have seen earlier. We shall predict the future value for the year 1960 using seasonal indexing.

## 

Below are the steps to forecast the value using seasonal indexing:

**Step 1:** Calculate the average number of passengers traveling each year.

## 

1. *# Excluding year 1960*
2. AP = AP\_reshaped.iloc[:11, :].copy()
3. *# Finding average yearly*
4. AP\_yearly\_mean = AP.mean(axis=1)
5. AP\_yearly\_mean

**Step 2:** Divide the monthly data by the average of the corresponding year. The resulting data is shown below.

## 

1. *# Dividing with mean values*
2. AP\_month\_avg = AP.div(AP\_yearly\_mean.values, axis=0)
3. AP\_month\_avg

**Step 3:** Compute the average across each year for the values of each month from step 2 in order to obtain the Seasonal Index

## 

1. *# Finding monthly average*
2. AP\_SI = AP\_month\_avg.mean(axis=0)
3. AP\_SI

The seasonal index(SI) can be used to calculate the increase or decrease in the predicted values using the formula:

**(1-SI)\*100.**

For example, let's estimate the change in predicted value for January and July 1960

Change in predicted value for January 1960 = ((1-0.8598)\*100) = 14.02

Change in predicted value for July 1960 = ((1-1.230005)\*100) = -23.30

Therefore the number of Passengers traveling in January 1960 will be 14.02% less than the average seasonal index while in the month of July it will be 23.30% more.

**Step 4:** Divide the AirPassenger data by the seasonal index to obtain the data that does not have a seasonal factor as we have assumed it to be multiplicative model. If the decomposition of the data is of type additive model then we need to subtract it from the seasonal index, this is known as deseasonalized data.

## 

1. *# Dividing by seasonal index, assuming multiplicative model*
2. AP\_deseasonalized = AP.div(AP\_SI)
3. AP\_deseasonalized

**Step 5:** Use linear regression to forecast the trend values for each month of 1960. (intercept=93.839; slope= 2.533)

## 

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. *# Reading data*
5. AP\_1D = pd.DataFrame(AP\_deseasonalized.values.reshape(-1, 1), columns=['value'])
6. *# Building Linear Regression model and predicting trend values*
7. from sklearn.linear\_model import LinearRegression
8. model = LinearRegression()
9. model.fit(np.arange(1, 133).reshape(-1, 1), AP\_1D.value.values.reshape(-1, 1) )
10. pred = pd.DataFrame(model.predict(np.arange(133, 145).reshape(-1, 1)), columns=['Trend values'], index=cols)
11. pred['Regression equations'] = pred['Trend values'].apply(lambda x: str(np.round(model.coef\_[0][0], 2)) + ' \* '+ str(np.round(x, 2)) + ' + ' + str(np.round(model.intercept\_[0], 2)))
12. pred

**Step 6:** Calculate the monthly occupancy for the year 1960 by multiplying the seasonality index with the trend value.

## 

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. *# Creating dataframe with seasonal index, trend values, monthly occupancy, and error in order to compare*
5. out = pd.DataFrame(AP\_SI, columns=['Seasonal Index'])
6. out.index.name = 1960
7. out['Trend values'] = pred['Trend values']
8. out['Monthly occupancy'] = np.round(AP\_SI \* pred['Trend values']).astype(int)
9. out['Error'] = AP\_reshaped.loc[1960] - AP\_SI \* pred['Trend values']
10. out

# Conclusion

We can observe from the above results that in August, the number of passengers boarding the international flight will be maximum, and in the month of November, it will be minimum.

Sum of the seasonal index is always equal to the seasonal span (Eg: for yearly data it is 12, quarterly data it is 4 and monthly data it is 1).

Mean of the seasonal index is always 1.

# **Forecasting**

The autoregressive and moving average models are used as stand alone models as well as in a combination depending on the type of data we are working with.

In order to use autoregressive, moving average, or autoregressive moving average models, we need to ensure that the series is stationary. Recall that we have learnt the properties of a stationary series:

* mean is constant
* variance is constant
* co-variance is constant

Now, let us understand how to convert a series into a stationary series and also to test the stationarity of a series.

# **Stationary Time Series**

A series is said to be "stationary" if its mean and variance is constant over time and the series should not have any trend.

A series that is non-stationary can be made stationary after differencing.

A series that is stationary after being differenced by 1 is said to be Integrated of order 1 and is denoted by I(1)

In general, a series that is stationary after being differenced 'd' times is said to be Integrated of order d, denoted by I(d)

A series that is not differenced is said to be I(0)

Below are the steps to compute Difference:

* Diff(1) = AirPassengers[n] – AirPassengers[n-1]
* Diff(2) = Diff(1)[n] – Diff(1)[n-1]

Lag is the delay in the series, below are the steps to compute lag:

* Lag(1) = AirPassengers[n] – AirPassengers[n-1]
* Lag(2) = AirPassengers[n] – AirPassengers[n-2]

Here AirPassengers[n] is the observed value of a time series data at a given point of time and 'n' is greater than 1. The below figure shows the diff and lag of the original data.

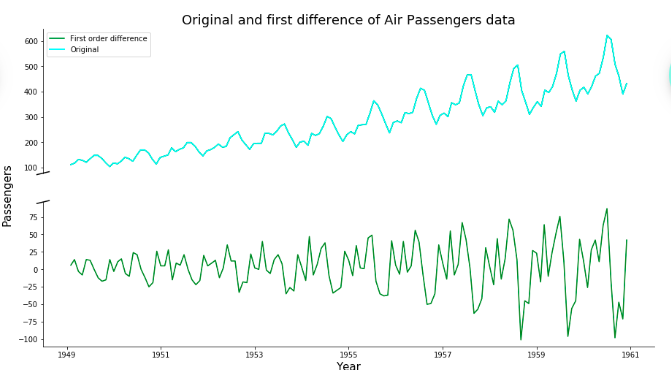
## 

The code of the above shown output is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. *# Taking year 1949 data*
5. st = pd.DataFrame(AirPassenger.iloc[:12].T.values[0], columns=['Year 1949'], index=cols)
6. *# Lag 1*
7. st['lag1'] = AirPassenger.iloc[:12].diff(1).T.values[0]
8. *# Lag 2*
9. st['lag2'] = AirPassenger.iloc[:12].diff(2).T.values[0]
10. *# Difference with order 1*
11. st['diffOrd1'] = np.concatenate(([np.nan], np.diff(AirPassenger.iloc[:12].T.values[0], 1)))
12. *# Difference with order 2*
13. st['diffOrd2'] = np.concatenate(([np.nan, np.nan], np.diff(AirPassenger.iloc[:12].T.values[0], 2)))
14. st

Lag(1) and diff(1) are always the same.

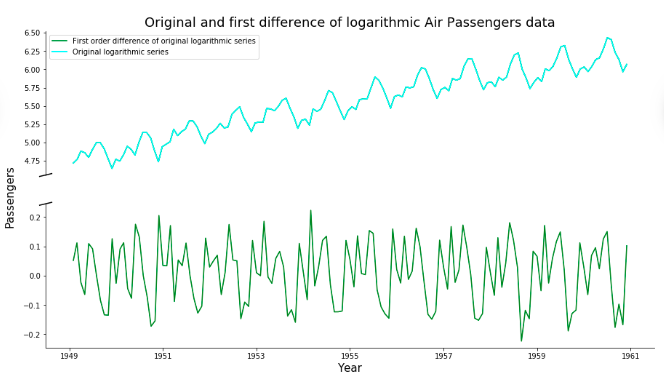
Below figure shows the differenced (I(1)) series of the AirPassengers (Δ AirPassengers) along with the original series. Lag and diff help us to remove the trend. In Δ AirPassengers, the trend is removed and the mean is almost constant, but it still has unequal variances.



The code of the above shown graph is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. *# First order difference values*
6. diffOrd1 = pd.DataFrame(np.concatenate(([np.nan], np.diff(AirPassenger.T.values[0], 1))), index=AirPassenger.index)
7. plt.figure(1, figsize=(15, 8))
8. ax1 = plt.subplot2grid((2, 1), (0, 0))
9. *# Dummy plot, used to create legend label for second plot*
10. ax1.plot(AirPassenger, label="First order difference", c='g')
11. ax1.plot(AirPassenger, label='Original', c='cyan')
12. ax1.set\_title('Original and first difference of Air Passengers data', fontsize=18)
13. *# Removing bottom, top and right plot lines*
14. ax1.spines['bottom'].set\_visible(False)
15. ax1.spines['top'].set\_visible(False)
16. ax1.spines['right'].set\_visible(False)
17. *# Removing ticks and labels of x-axis*
18. ax1.tick\_params(axis='x', which='both', bottom=False, labelbottom=False)
19. *# Adding a diagonal mark on y-axis*
20. d = .01 *# how big to make the diagonal lines in axes coordinates*
21. kwargs = dict(transform=ax1.transAxes, color='k', clip\_on=False)
22. ax1.plot((-d,d),(-d,+d), \*\*kwargs)
23. plt.legend()
24. *# Generating second plot*
25. ax2 = plt.subplot2grid((2, 1), (1, 0))
26. ax2.plot(diffOrd1, c='g')
27. ax2.set\_xlabel('Year', fontsize=15)
28. ax2.set\_ylabel('Passengers', fontsize=15)
29. ax2.spines['top'].set\_visible(False)
30. ax2.spines['right'].set\_visible(False)
31. kwargs = dict(transform=ax2.transAxes, color='k', clip\_on=False)
32. ax2.plot((-d,d),(1-d,1+d), \*\*kwargs)
33. *# Adding a common y-label*
34. ax2.yaxis.set\_label\_coords(-0.05, 1.05)
35. plt.show()

We may use *log*transformation in order to have an equal variance for the AirPassengers data. We can observe from the below figures that the log(AirPassengers) data has almost uniform variance and the Δlog(AirPassengers) has no trend. Mean and variance is almost uniform.



The code of the above shown graph is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. *# First order difference values of logarithmic series*
6. diffOrd1 = pd.DataFrame(np.concatenate(([np.nan], np.diff(np.log(AirPassenger).T.values[0], 1))), index=AirPassenger.index)
7. plt.figure(1, figsize=(15, 8))
8. ax1 = plt.subplot2grid((2, 1), (0, 0))
9. *# Dummy plot, used to create legend label for second plot*
10. ax1.plot(np.log(AirPassenger), label="First order difference of original logarithmic series", c='g')
11. ax1.plot(np.log(AirPassenger), label='Original logarithmic series', c='cyan')
12. ax1.set\_title('Original and first difference of logarithmic Air Passengers data', fontsize=18)
13. *# Removing bottom, top and right plot lines*
14. ax1.spines['bottom'].set\_visible(False)
15. ax1.spines['top'].set\_visible(False)
16. ax1.spines['right'].set\_visible(False)
17. *# Removing ticks and labels of x-axis*
18. ax1.tick\_params(axis='x', which='both', bottom=False, labelbottom=False)
19. *# Adding a diagonal mark on y-axis*
20. d = .01 *# how big to make the diagonal lines in axes coordinates*
21. kwargs = dict(transform=ax1.transAxes, color='k', clip\_on=False)
22. ax1.plot((-d,d),(-d,+d), \*\*kwargs)
23. plt.legend()
24. *# Generating second plot*
25. ax2 = plt.subplot2grid((2, 1), (1, 0))
26. ax2.plot(diffOrd1, c='g')
27. ax2.set\_xlabel('Year', fontsize=15)
28. ax2.set\_ylabel('Passengers', fontsize=15)
29. ax2.spines['top'].set\_visible(False)
30. ax2.spines['right'].set\_visible(False)
31. kwargs = dict(transform=ax2.transAxes, color='k', clip\_on=False)
32. ax2.plot((-d,d),(1-d,1+d), \*\*kwargs)
33. *# Adding a common y-label*
34. ax2.yaxis.set\_label\_coords(-0.05, 1.05)
35. plt.show()

# Conclusion

The plot of the series helps us to draw some conclusions, whether a particular series is stationary or not. We may also use Dickey-Fuller test to check the stationarity of a series.

ADF Test for Stationarity

We can check the stationarity of the series by using the Dickey-Fuller test. It uses hypothesis testing for checking the stationarity of a series as mentioned below.

Hnull  : series is non-stationary

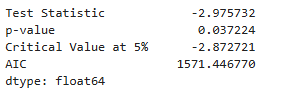
Halt : series is stationary

If the null hypothesis is rejected, then the series is assumed to be stationary, where the p-value will be less than 0.05.

If the null hypothesis is not rejected, then the series is assumed to be non-stationary. We need to difference the variable and repeat the dickey-fuller test to see if the differenced variable is stationary.

**Example 1**:  To check whether the series hsales is stationary or not, we can use adfuller method in python.

1. *# Importing libraries*
2. import pandas as pd
3. *# Load fresh data*
4. dateparse = lambda dates: pd.datetime.strptime(dates, '%m/%d/%Y')
5. hsales = pd.read\_csv('hsales.csv', index\_col='date', date\_parser=dateparse)
6. *# Function for Dickey-Fuller test*
7. from statsmodels.tsa.stattools import adfuller
8. def dffunc(ts):
9. dftest = adfuller(ts, autolag='AIC')
10. dfoutput = pd.Series([dftest[0],dftest[1], dftest[4]["5%"],dftest[5]], index=['Test Statistic','p-value', 'Critical Value at 5%','AIC'])
11. return(dfoutput)
12. *# Passing values as 1D numpy array*
13. dffunc(hsales.T.values[0])



The series is stationary since the p-values (0.037224) is less than 0.05.

**Example 2**:  To check whether the first differenced series of log(AirPassengers) is stationary or not.

1. *# Importing libraries*
2. import numpy as np
3. *# Dickey-Fuller test of first order difference of log transformed Air passengers data*
4. dffunc(np.diff(np.log(AirPassenger).T.values[0], 1))

The series is still not stationary since the p-values (0.071) is greater than 0.05.

Let us take the second difference of log(AirPassengers) series and check the stationarity once again.

1. *# Importing libraries*
2. import numpy as np
3. *# Dickey-Fuller test of second order difference of log transformed Air passengers data*
4. dffunc(np.diff(np.log(AirPassenger).T.values[0], 2))

This time the p-value(7.41e-13) is much smaller than 0.05 and hence the series has become stationary.

# Conclusion

Thus, we can utilize Dickey-Fuller test to quickly analyze if a series is stationary or not without visualization.

Auto Regressive (AR) Model

Autoregressive model works only on stationary data and is used to examine the relationship between different values of the same variable. It is a linear model that can be used to predict future values based on past and present values.

It is represented as AR(p), where p denotes the order or the number of observations considered.

AR(p) :  **y(t) =** **β(0) + β(1)y(t-1) + β(2)y(t-2) + β(3)y(t-3) + … + β(p)y(t-p)+ ε(t)**

Where 'β' is the coefficient & 'ε' is an error.

For p=1 and p=3

AR(1)  :  y(t) = β0 + β1y(t-1)+ ε(t)

AR(3)  :  y(t) = β0 + β1y(t-1) + β2y(t-2) + β3y(t-3)+ ε(t)

# Conclusion

Autoregressive model helps us to predict the future. There are many ways to identify an appropriate order of the model. Partial auto correlation function (PACF) plot is one of them, which we will see later in this course. It is also possible to predict the future values by using the error terms i.e., moving average.

Moving Average (MA) Model

The moving average model uses the past errors that result from the regression model to predict the future values.

The future values are calculated by multiplying the past errors with their corresponding coefficients.

Here ε*(t)* represents the error at time 't'. It is independent and identically distributed data.

MA(q) : **y*(t)* = β*(0)* + α*(0)*ε*(t)* + α*(1)*ε*(t-1)* + α*(2)*ε*(t-2)* + α*(3)*ε*(t-3)*+ … + α*(q)*ε*(t-q)***

Here, q denotes the order of the moving average.

Moving average is represented as MA(q), where q is the order of the model, which indicates how many previous errors we consider to predict the present data

MA(1)  :  y*(t)* = β*0* + α*0*ε*(t)* + α*1*ε*(t-1)*

MA(3)  :  y*(t)* = β*0* + α*0*ε*(t)* + α*1*ε*(t-1)* + α*2*ε*(t-2)* + α*3*ε*(t-3)*

Where 'β0' and 'α' are the coefficient & 'ε' is an error term with mean zero and a constant variance.

We can determine the order of the moving average model by observing the auto correlation function(ACF) plot, which we will see in the later part of this course. We might also use a combination of autoregressive and moving average for model building which will lead us to more accurate prediction, but it all depends on the data.

ARIMA Model

Autoregressive Integrated Moving Average which is also known as Box-Jenkins models which may include autoregressive, moving average, and differencing.

It is referred to as AR if it uses only autoregressive model, MA if it uses moving average. Differencing order refers to successive first difference.

 It is used for forecasting trend and assumes that data are correlated with the past data values.

* Non-seasonal ARIMA(p,d,q)
  + p: Autoregressive Order
  + d: Integration Order
  + q: Moving Average Order
* Seasonal ARIMA(p,d,q)x(P,D,Q)s
  + P=number of seasonal autoregressive (SAR) terms,
  + D=number of seasonal differences,
  + Q=number of seasonal moving average (SMA) terms
  + S = seasonality period

Here p,d,q are the non-seasonal terms and, P, D, Q are the seasonal terms.

When both p and q are not equal to 0 and d=0, then a combination of AR and MA can be used. This is known as the ARMA model.

* ARMA(p,q)
  + p: Autoregressive Order
  + q: Moving Average Order

The model is referred as:

**y(t) =** **AR(p) + MA(q)**

**y(t) =** **β(0) +** (**β(1)y(t-1) + β(2)y(t-2) + β(3)y(t-3) + … + β(p)y(t-p)) + (α(0)ε(t) + α(1)ε(t-1) + α(2)ε(t-2) + α(3)ε(t-3)+ … + α(q)ε(t-q))**

For ARMA(1,1) and ARMA(3,3):

ARMA(1,1):  y(t) = β(0) + (β(1)y(t-1)) + (α(0)ε(t) + α(1)ε(t-1))

ARMA(3,3):  y(t) = β(0) + (β(1)y(t-1) + β(2)y(t-2) + β(3)y(t-3)) + (α(0)ε(t) + α(1)ε(t-1) + α(2)ε(t-2) + α(3)ε(t-3))

p and q can be determined by observing the auto correlation and partial auto correlation plot.

## Diagnostic Checking

Different models can be obtained for various combinations of AR and MA individually and collectively.

The best model is obtained by following the diagnostic testing procedure. Below are the two measures for the goodness of fit. The measure trade-off between model fit and complexity of the model.

**1. Akaike Information Criterion(AIC)**

AIC = -2ln(L) + 2k

where L is the value of likelihood function

k is the number of estimated parameters

**2. Bayesian Information Criterion(BIC)**

BIC = -2ln(L) + ln(N)k

where L is the value of likelihood function

N is the number of observations

k is the number of estimated parameters

# Conclusion

The model with the lowest value of the above criterion(AIC and BIC) is chosen as the best model.

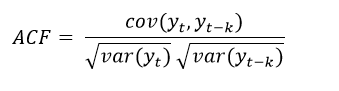
## ACF and PACF

## Autocorrelation and Partial Autocorrelation Function

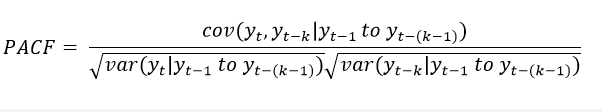
Before we consider the other two models: ARMA and ARIMA, we need to evaluate the values of p, d, and q. This will help us select the appropriate ARIMA(p,d,q) model.

The auto correlation function(ACF) and partial auto correlation function(PACF) are used to estimate the parameters of ARIMA(p,d,q).

* Auto correlation function is a simple correlation between current observation (Yk) and the observation p periods from the current one (Y(t-k))



* Partial auto correlations are used to measure the degree of association between Yt and Y(t-k) when the effects at other time lags 1,2,3,.., (k-1) are removed.



**ACF**

The auto correlation function plot (ACF plot) helps us to estimate the parameter q of the ARIMA model(p,d,q). In the ACF plot, the spikes at different lag are considered to be insignificant if it lies between the dashed lines.

The auto correlation function plot (ACF plot) helps us to estimate the parameter q of the ARIMA model(p,d,q). In the ACF plot, the spikes at different lag are considered to be insignificant if it lies between the dashed lines.

The ACF plot of hsales data is shown below. From the plot, we can make the following observations:

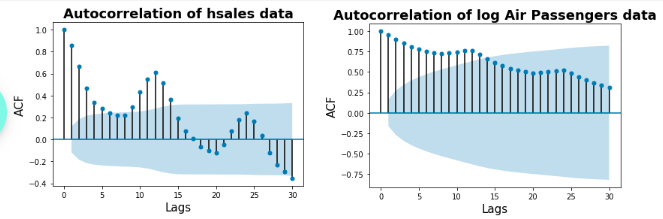
* The difference between the ACF value between lags is not significant.
* The ACF values decay towards 0

This signifies that the hsales data might not have MA(q) and that the series is stationary.

However, when we observe the ACF plot of log(AirPassengers), we see that:

* The difference between the ACF value between lags is not significant
* The ACF values are not decaying towards 0, indicating that the series is not stationary

As the series is non-stationary, we need to make the series stationary by differencing it.

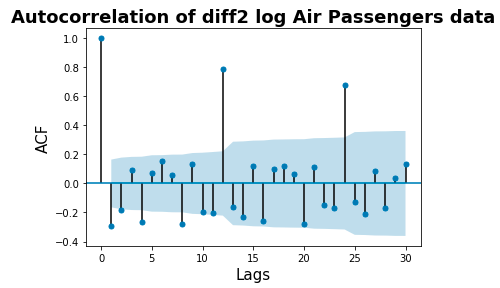


The code of the above shown graph is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from statsmodels.graphics.tsaplots import plot\_acf
5. *# ACF plot of hsales*
6. plot\_acf(hsales, lags=30)
7. plt.xlabel('Lags', fontsize=15)
8. plt.ylabel('ACF', fontsize=15)
9. plt.title('Autocorrelation of hsales data', fontsize=18, weight='bold')
10. plt.show()
11. *# ACF plot of log(Airpassengers)*
12. plot\_acf(np.log(AirPassenger), lags=30)
13. plt.xlabel('Lags', fontsize=15)
14. plt.ylabel('ACF', fontsize=15)
15. plt.title('Autocorrelation of log Air Passengers data', fontsize=18, weight='bold')
16. plt.show()

Now, let us consider the Δ2log(AirPassengers) data which is stationary. We can observe from the figure that:

* There is a significant difference between the ACF values between adjacent lags
* The ACF values abruptly go towards 0 after lag 2

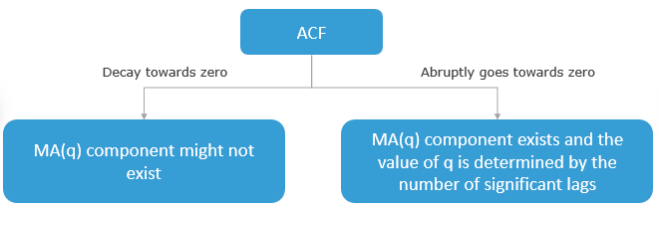


The code of the above shown graph is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from statsmodels.graphics.tsaplots import plot\_acf
5. *# ACF plot of diff2(log(Airpassengers))*
6. plot\_acf(np.diff(np.log(AirPassenger).T.values[0], 2), lags=30)
7. plt.xlabel('Lags', fontsize=15)
8. plt.ylabel('ACF', fontsize=15)
9. plt.title('Autocorrelation of diff2 log Air Passengers data', fontsize=18, weight='bold')
10. plt.show()

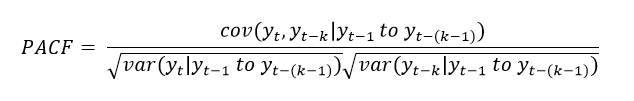
The abrupt movement of the spike towards 0 signifies that Δ2log(AirPassengers) data might have MA(q) component. The lag after which the value of ACF is between the shaded area is chosen as the value of q. Here we can observe that q = 2.

If a model is only MA(q) model then the spikes in the ACF plot goes abruptly towards zero (i.e the difference between the first and the second lag is significant) and in PACF spikes decay towards zero after.



Partial Autocorrelation Function

Partial auto correlations are used to measure the degree of association between Yt and Y(t-k) when the effects at other time lags 1,2,3,.., (k-1) are removed.



From the PACF plot of hsales data, we can make the following observations:

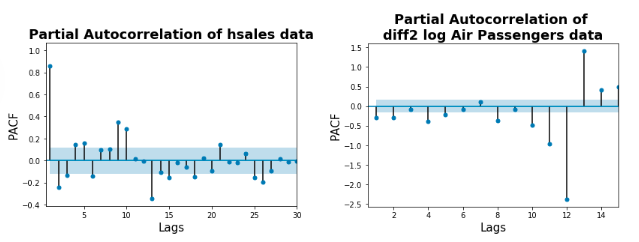
* There is a significant difference between the PACF values between adjacent lags.
* The PACF values abruptly go towards 0 after lag 6.

The abrupt movement of the spike towards 0 signifies hsales data might have AR(p) component. The lag after which the value of ACF is between the shaded area is chosen as the value of p. Here we can observe that p=6.

From the PACF plot of Δ2log(AirPassengers) data, we can make the following observations:

* There is a significant difference between the PACF values between adjacent lags.
* The PACF values abruptly go towards 0 after lag 2.

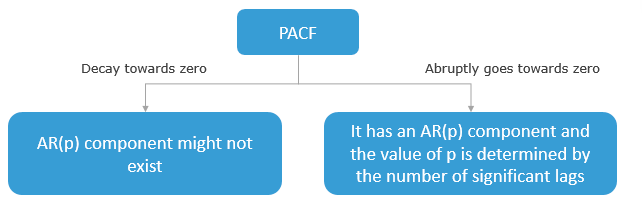
The abrupt movement of the spike towards 0 signifies Δ2log(AirPassengers) data might have AR(p) component. The lag after which the value of PACF is between the shaded area is chosen as the value of p. Here we can observe that p=2.



The code of the above shown graph is as follows:

1. *# Importing libraries*
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from statsmodels.graphics.tsaplots import plot\_pacf
5. *# PACF plot of hsales*
6. plot\_pacf(hsales, lags=30)
7. plt.xlabel('Lags', fontsize=15)
8. plt.ylabel('PACF', fontsize=15)
9. plt.xlim(0.5, 30)
10. plt.title('Partial Autocorrelation of hsales data', fontsize=18, weight='bold')
11. plt.show()
12. *# PACF plot of diff2(log(Airpassengers))*
13. plot\_pacf(np.diff(np.log(AirPassenger).T.values[0], 2), lags=15)
14. plt.xlabel('Lags', fontsize=15)
15. plt.ylabel('PACF', fontsize=15)
16. plt.xlim(0.5, 15)
17. plt.title('Partial Autocorrelation of \ndiff2 log Air Passengers data', fontsize=18, weight='bold')
18. plt.show()

If a model is only AR(p) model then the spikes in the PACF plot goes abruptly towards zero and in ACF spikes decay towards zero.



Estimating the ARIMA parameters

Seasonal ARIMA procedure

Steps involved in Seasonal ARIMA (SARIMA) modeling:

1. If a series has seasonality and no trend, then we need to take a difference after lag 's', where 's' is the seasonality span. For instance, take the 12th difference for monthly data with seasonality.
2. If a series has a trend and no seasonality, then take the 1st difference.
3. If a series has both trend and seasonality, then we need to apply a seasonal difference to the data and then re-evaluate the trend.  If the series still has a trend, then we need to take the 1st difference of seasonal differenced series.
4. If the series has a strong and consistent seasonality pattern, then we need to use seasonal differencing (D). But, the value of (d+D) should not be more than 2.
5. Determine the values of p and q as discussed earlier
6. In only seasonal autoregressive (SAR(1)) there are spikes in ACF at lag s,2s,3s, etc, while in PACF it cuts off after lag s.
7. In only seasonal moving average (SMA(1)) there are spikes in PACF at lag s,2s,3s, etc, while in ACF it cuts off after lag s.
8. If the autocorrelation at the seasonal period is positive consider adding a SAR term to the model, on the other hand, if the autocorrelation at the seasonal period is negative consider adding an SMA term to the model. Usually order of SAR and SMA is not more than 1. We should avoid mixing both SAR and SMA term.

Seasonal ARIMA model building and evaluation

Here we try to estimate the parameters for AirPassengers data and later build and evaluate the SARIMA model.

# Parameters estimation

We have already figured out the values of p, and q which are 2, and 2. Also, we have differentiated the series to make it stationary, hence the value of d = 1.

Next, let us figure out the values corresponding to P, Q, and D. From the ACF and PACF plots of Δ2log(AirPassengers) we can observe a spike in lag (seasonal, S = 12) 12 and therefore can estimate that the values of P, as well as Q, is 1. Furthermore, the original series has a stable seasonal pattern over time, and therefore D = 1.

# Model building

Let us build the model using SARIMA function available under Statsmodels package. We will split the data into 2 parts one for training and the other used for forecasting/testing.

# Model evaluation

We use RMSE to find the performance of the current model.

1. *# Importing libraries*
2. import numpy as np
3. *#finding the mean squared error*
4. from sklearn.metrics import mean\_squared\_error
5. np.sqrt(mean\_squared\_error(AirPassenger.iloc[k:], np.round(np.exp(model\_fit.forecast(144-k)))))



# Conclusion

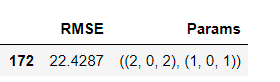
As we can observe from the figure that the model has forecasted 4 years of data quite well with an RMSE of 23.

Finding the parameters manually can be challenging sometimes, therefore next you will learn about grid search which creates different models on a range of parameters and gets the best based on AIC/BIC value.

 Grid search

To find the optimum values of p, d, q, and P, D, Q, we can iterate over a given range of values and find the RMSE of each model. Later, the model with the least RMSE can be considered as the best model.

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. from statsmodels.tsa.statespace.sarimax import SARIMAX
5. *# Iterating over parameters*
6. rmse = []
7. params = []
8. for p in range(3):
9. for d in range(2):
10. for q in range(3):
11. for P in range(3):
12. for D in range(2):
13. for Q in range(3):
14. try:
15. model = SARIMAX(np.log(AirPassenger.iloc[:k]),
16. order=(p, d, q),
17. seasonal\_order=(P, D, Q, 12),
18. enforce\_stationarity=False,
19. enforce\_invertibility=False)
20. model\_fit = model.fit(disp=False)
21. rmse.append(np.sqrt(mean\_squared\_error(AirPassenger.iloc[k:],
22. np.round(np.exp(model\_fit.forecast(144-k))))))
23. params.append(((p, d, q), (P, D, Q)))
24. except:
25. pass
27. *# Storing RMSE and parameters and sorting the dataframe based on RMSE in ascending order*
28. res = pd.DataFrame([rmse, params]).T.sort\_values([0])
29. res.columns = ['RMSE', 'Params']
30. res.head(1)



As we can observe that the parameters are similar to what we have found out manually.

We can also picture all the values of RMSE as shown:

1. *# Importing libraries*
2. import numpy as np
3. import matplotlib.pyplot as plt
4. *# plotting RMSE values across all possible parameter combinations*
5. plt.figure(figsize=(15, 5))
6. plt.plot(rmse, label='All RMSEs')
7. plt.scatter(np.arange(210, 211), res.iloc[0, 0], s=70, c='g', label='Chosen RMSE')
8. plt.legend()
9. plt.xlabel('Iterations', fontsize=15)
10. plt.ylabel('RMSE', fontsize=15)
11. plt.title('RMSE values across all possible parameter combinations', weight='bold', fontsize=18)
12. plt.show()

**Seasonal indexing - Exercise**

**Problem Statement**

For the dataset [hsales](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_012717731760349184262_shared/web-hosted/assets/hsales1612768380259.zip" \t "_blank), you need to forecast the values of year 1996 using seasonal indexing method.

References:

* Forecasting from otexts
* Wikipedia
* Time series analysis course from National Institute of Standards and Technology
* Stat 510 course from PennState
* Duke University

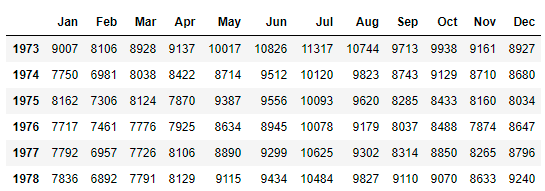
Capstone project

**Problem Statement**

Dataset : [usdeaths](https://infyspringboard.us.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_0126051855394242561300_shared/web-hosted/assets/usdeaths1612787077779.zip" \t "_blank)

'usdeaths' is a monthly data which shows the number of accidental deaths in the USA from January 1973 to December 1978.

A reshaped version of the dataset is shown below:



1. Use exponential smoothing to predict the accidental deaths in the US for January 1978, based on the historical data from January 1973 to December 1977. Plot the predicted data along with the actual and the past data.
2. Use ARIMA to predict the monthly deaths in the US for the year 1978, based on the historical data from January 1973 to December 1977. Also, plot the forecasted data along with the actual and the past data.
3. Find the confidence interval of the forecasted data by ARIMA model. Also, plot the forecasted data along with the confidence intervals.