Time Series Forecasting - I

Types of Forecasting

Forecasting means predicting some values that might occur in future. For example, stock market traders often tend to forecast the stock prices based on the current stock prices or on the trend followed in the past. Similarly, the sales department of a company plans to forecast the sales for this quarter based on various factors affecting the company's profitability. And there are various examples of forecasting that are present in the industry. Forecasting values in future based on the present or previous time's values is called time series forecasting. And the data used in such forecasting is called a time series data.

Before we speak much about the time series data and it's forecasting techniques, let's understand in general what are the types of forecasting.

There are two types of forecasting. They are:

1. Qualitative forecasting
2. Quantitative forecasting

The following are the differences between the quantitative and qualitative forecasting

Quantitative Forecasting	**Qualitative Forecasting**
Based on the data and any repeating historical patterns in the data.	This procedure is taken when data is not available and historical patterns do not repeat. This is based on expert decisions.
Captures complex patterns which humans cannot identify.	May not capture complex patterns.
No bias	Bias
E.g. Time series forecasting	E.g. Delphi method

One of the quantitative forecasting methods is **time series forecasting**.

It is important to understand the components of a time series data and they are explained as follows:

Time Series Data: Any data that has a time component involved in it is termed as a time-series data. For example, the number of orders made on a food ordering app per day is an example of time-series data.

Time Series Analysis: Performing analysis on a time-series data to find useful insights and patterns in termed as time series analysis. Let's take a food ordering app example. The app might have the data for every day logged in per hour. They might notice that in this data, the number of orders is significantly higher in, say, the 1-2 PM time slot but is significantly lower in the 3-4 PM time slot. This information might be useful for them as they would then be able to estimate the number of delivery boys required at a particular time of the day. Hence, time series analysis is indispensable while working with any time series data.

Time Series Forecasting: Time series forecasting is basically looking at the past data to make predictions into the future. Say that the food ordering app wants to predict the number of orders per day for the next month in order to plan the resources better. For this, they will look at tons of past data and use it in order to forecast accurately.

In the given module, we are using the following problem statement to predict forecasts on a time series data of an airline. The problem statement is given below.

The Air Passenger Traffic Forecasting Problem: An airline company has the data on the number of passengers that have travelled with them on a particular route for the past few years. Using this data, they want to see if they can forecast the number of passengers for the next twelve months.

Making this forecast could be quite beneficial to the company as it would help them take some crucial decisions like -

1. What capacity aircraft should they use?
2. When should they fly?
3. How many air hostesses and pilots do they need?
4. How much food should they stock in their inventory?

Basic Steps for Forecasting

For any forecasting problem, there are some basic steps that need to be in place. Let us look at the basic steps mentioned below. And these steps are similar to any data science problem.

The basic steps involved in any forecasting problem are:

1. Define the problem
2. Collect the data
3. Analyze the data
4. Build and evaluate the forecast model

The one thing to keep in mind before moving forward is that there are some caveats associated with a time series forecasting. These caveats revolve around the steps followed while defining the problem.

The Granularity Rule: The more aggregate your forecasts are, the more accurate you are in your predictions simply because aggregated data has lesser variance and hence, lesser noise. As a thought experiment, suppose you work at ABC, an online entertainment streaming service, and you want to predict the number of views for a few newly launched TV shows in Mumbai for the next one year. Now, would you be more accurate in your predictions if you predicted at the city-level or if you go at an area-level? Obviously, accurately predicting the views from each area might be difficult but when you sum up the number of views for each area and present your final predictions at a city-level, your predictions might be surprisingly accurate. This is because, for some areas, you might have predicted lower views than the actual whereas, for some, the number of predicted views might be higher. And when you sum all of these up, the noise and variance cancel each other out, leaving you with a good prediction. Hence, you should not make predictions at the very granular levels.

The Frequency Rule: This rule tells you to keep updating your forecasts regularly to capture any new information that comes in the data so that you can account for any recent jumps or changes in any attribute. Let's continue with the ABC, an online entertainment streaming service, an example where the problem is to predict the number of views for a newly launched TV show in Mumbai for the next year. Now, if you keep the frequency too low, you might not be able to capture accurately the new information coming in. For example, say, your frequency for updating the forecasts is 3 months. Now, due to the COVID-19 pandemic, the residents may be locked in their homes for around 2-3 months during which the number of views will significantly increase. Now, if the frequency of your forecast is only 3 months, you will not be able to capture the increase in views which may incur significant losses and lead to mismanagement.

The Horizon Rule: When you have the horizon planned for a large number of months into the future, you are more likely to be accurate in the earlier months as compared to the later ones. Let's again go back to ABC, an online entertainment streaming service, example. Suppose that Netflix made a prediction for the number of views for the next 6 months in December 2019. Now, it may have been quite accurate for the first two months, but due to the unforeseen COVID-19 situation, the actual number of views in the next couple of months would have been significantly higher than predicted because of everyone staying at home. The farther ahead we go into the future, the more uncertain we are about the forecasts.

Collect the Data

After we have identified the problem statement and studied it, we need to work on collecting or gathering the required data for the business problem.

There are three important characteristics that every time series data must exhibit in order for us to make a good forecast.

1. **Relevant**: The time-series data should be relevant for the set objective that we want to achieve.
2. **Accurate**: The data should be accurate in terms of capturing the timestamps and capturing the observation correctly.
3. **Long enough**: The data should be long enough to forecast. This is because it is important to identify all the patterns in the past and forecast which patterns repeat in the future.

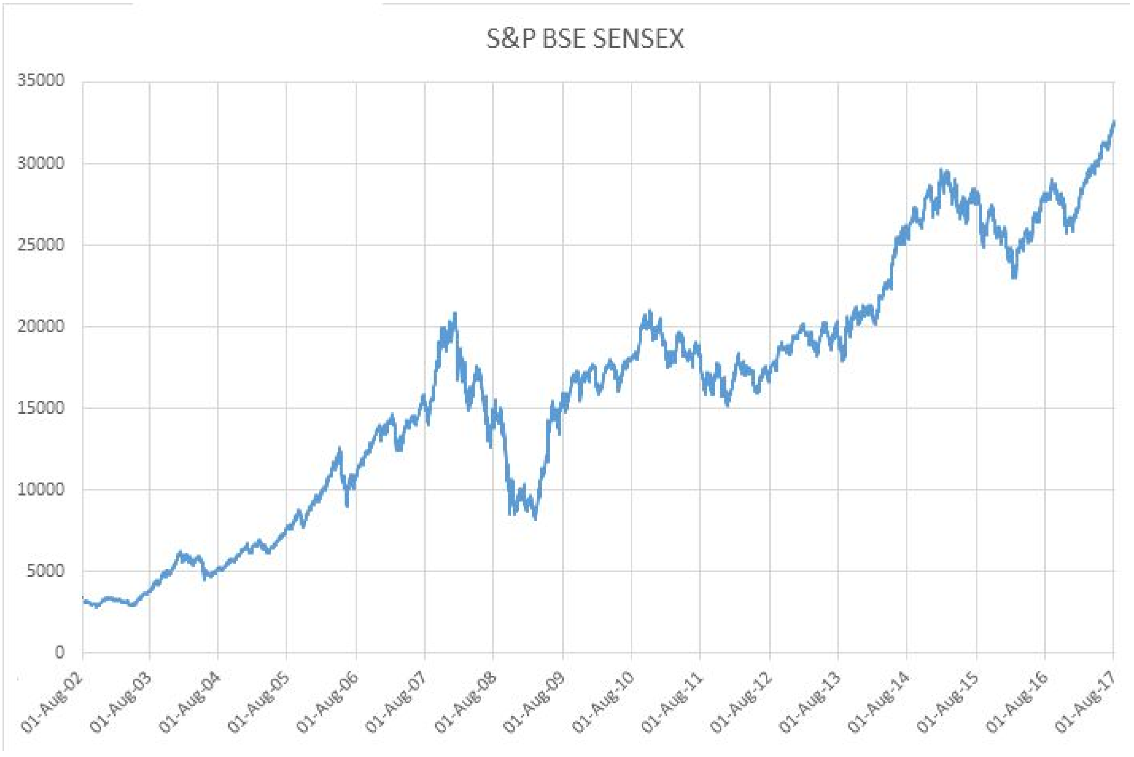
There are various types of data sources to get a time-series data and they are:

1. **Private enterprise data**: E.g. financial information about the quarterly results of any private organisation.
2. **Public data**: E.g. the government publishes the economic indicators such as GDP, consumer price index etc.
3. **System/Sensor data**: E.g. Logs generated by the servers during their 24/7 working hours.

Analyze the Data

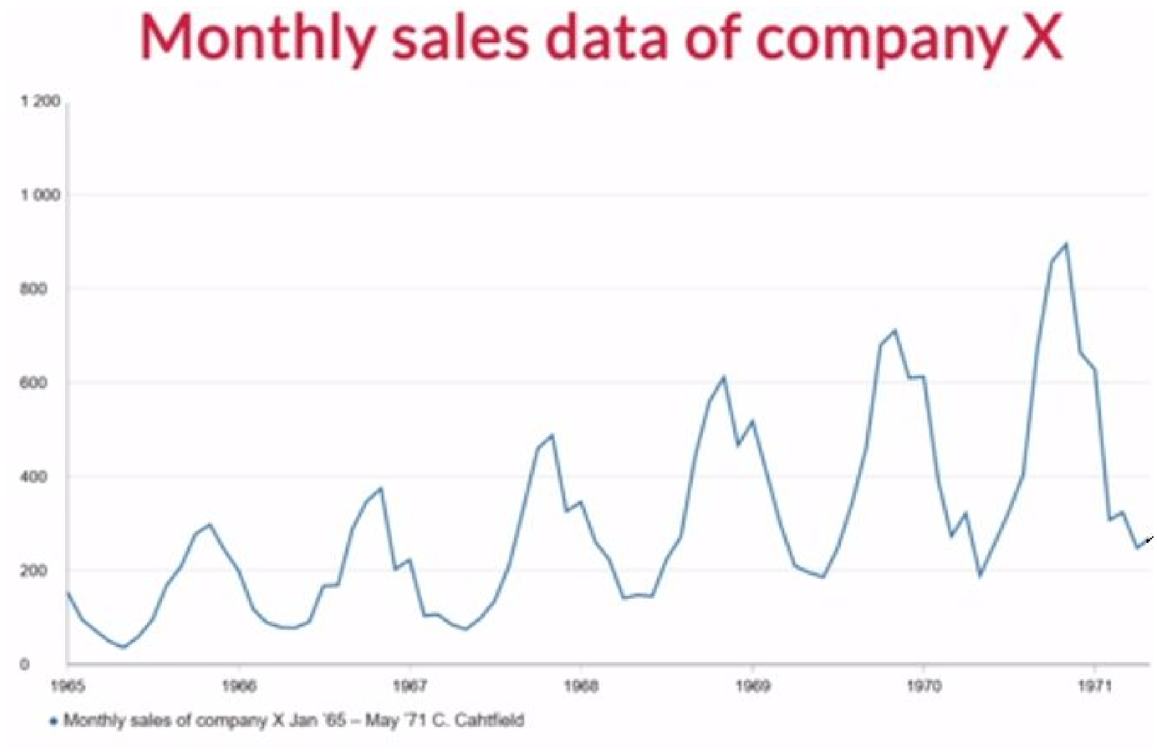
After defining the problem statement and collecting the data comes the next step of analysing the data and extracting meaningful insights from it.

1. **Level**: This is the baseline of a time series. This gives the baseline to which we add the different other components.
2. **Trend**: Over a long term, this gives an indication of whether the time series moves lower or higher. For example, in the following Sensex graph you can clearly observe that with time, the overall value is increasing i.e. this particular time series data has an increasing trend.

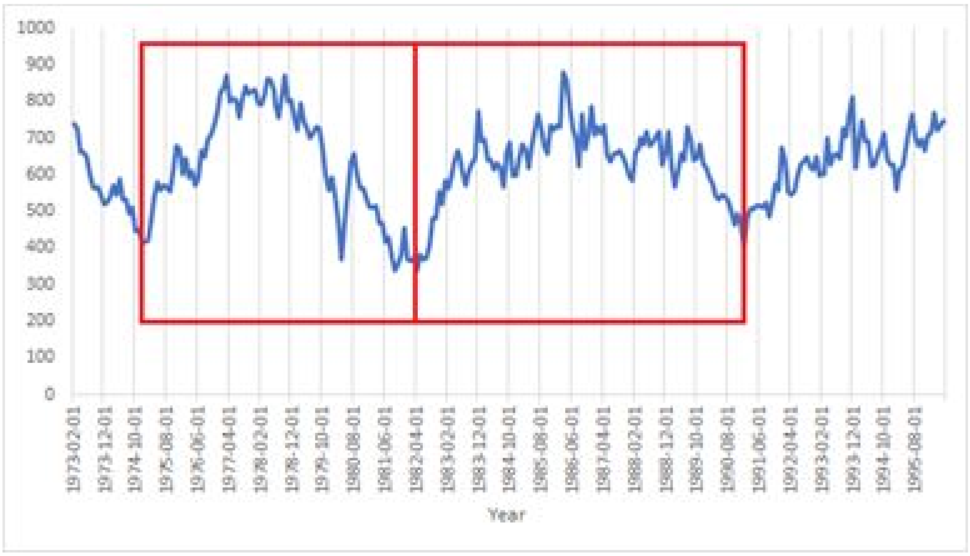


3. **Seasonality:** It is a pattern in a time-series data that repeats itself after a given period of time. For example, in the following graph 'Monthly sales data of company X', you can clearly observe that a fixed pattern is repeating every year.

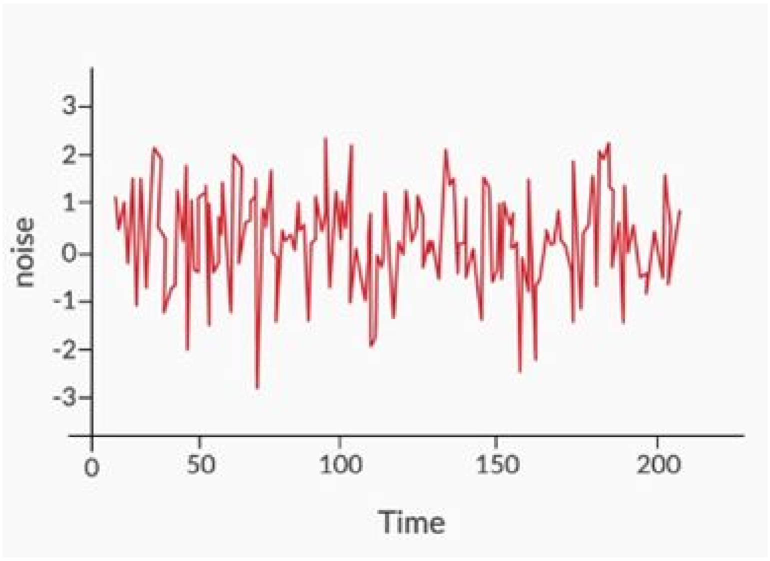
   The simplest example to explain this could be, say, the sales of winter wear in India. In winter, during months like November-January, you would expect these sales to be very high whereas for the other months, the sales might be low. This shows a seasonality pattern and proves to be very useful when making forecasts.



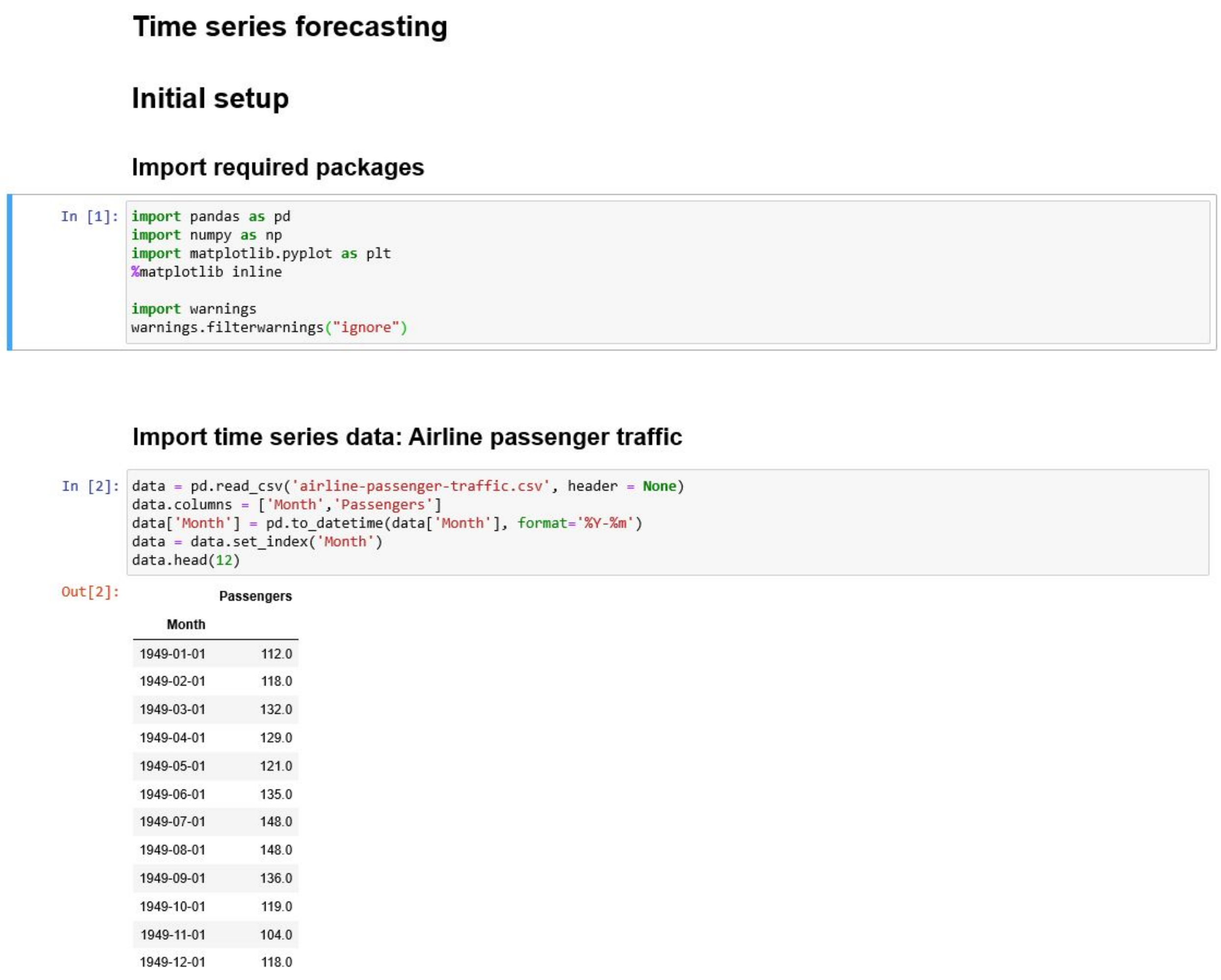
4. **Cyclicity:** It is also a repeating pattern in data that repeats itself *aperiodically*. We don't get into the more details of this component as it is out of the scope of this module.



5. **Noise:** Noise is the completely random fluctuation present in the time series data and we cannot use this component to forecast into the future. This is that component of the time series data that no one can explain and is completely random. Because of noise present in the data, it becomes difficult to forecast sometimes the accurate values.



For the airline passenger traffic dataset shared, we will start with the initial setup of the time series data. The same has been explained in the code shared below



Time series forecasting

Initial setup

Import required packages

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")
```

Import time series data: Airline passenger traffic

In [2]:
```
data = pd.read_csv('airline-passenger-traffic.csv', header = None)
data.columns = ['Month','Passengers']
data['Month'] = pd.to_datetime(data['Month'], format='%Y-%m')
data = data.set_index('Month')
data.head(12)
```

Out[2]:

	Passengers
Month	
1949-01-01	112.0
1949-02-01	118.0
1949-03-01	132.0
1949-04-01	129.0
1949-05-01	121.0
1949-06-01	135.0
1949-07-01	148.0
1949-08-01	148.0
1949-09-01	136.0
1949-10-01	119.0
1949-11-01	104.0
1949-12-01	118.0

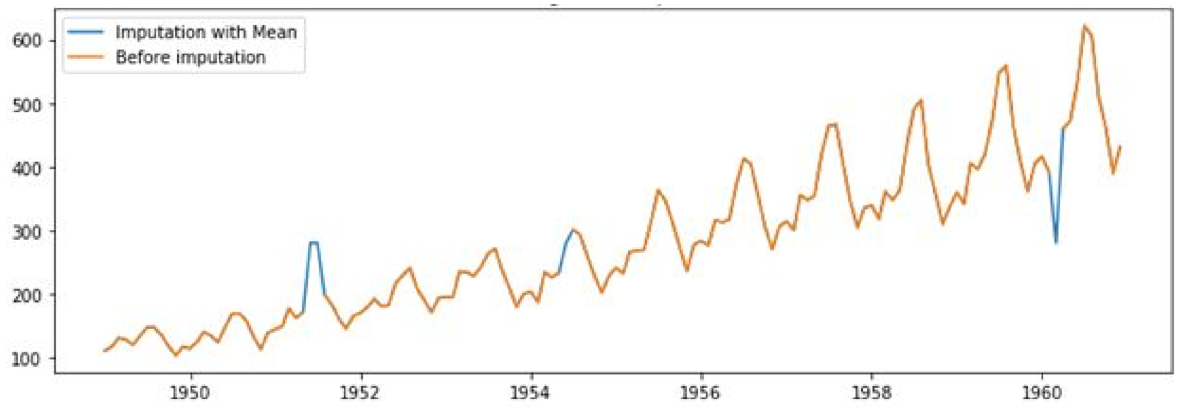
The above images show the basic packages required to import the time series data and how to read the data for the attributes present in it.

Handling Missing Values

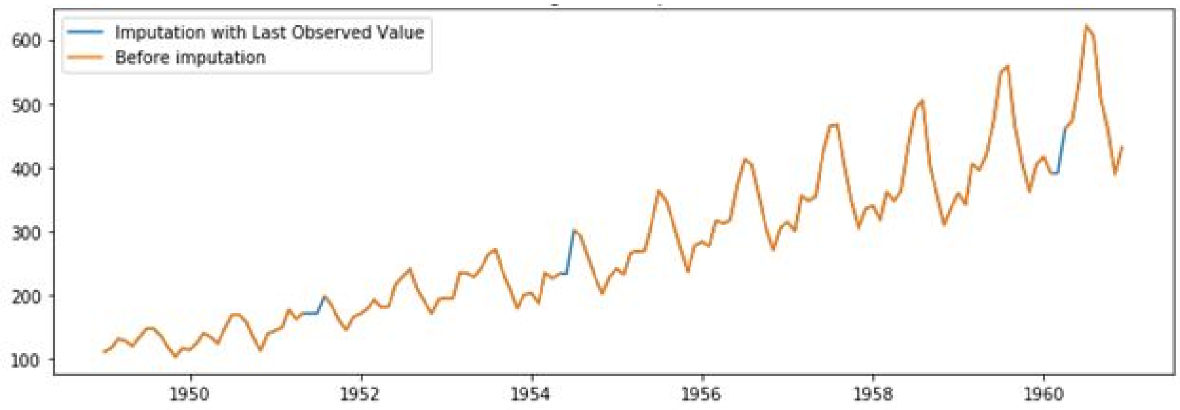
It is important before we even start building the time series models, we should check the dataset for any missing values. This will help us predict or forecast the values accurately as it keeps in check all the past observations in check. Also, without missing values and analysing the raw dataset, it is difficult to build forecasting models on the time series data.

Following are the methods of handling missing values in any time series data.

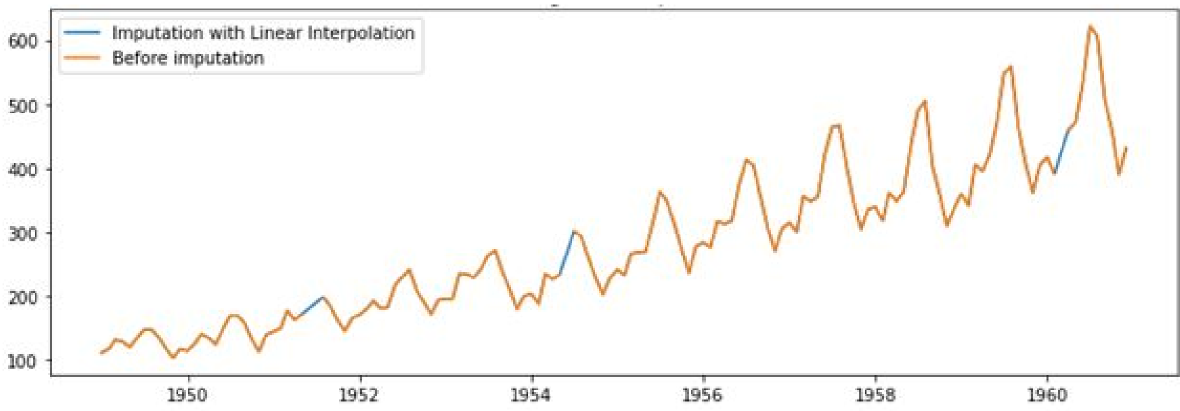
1. **Mean Imputation:** Imputing the missing values with the overall mean of the time series data. In the following graph, the blue curves indicate the values imputed using the mean imputation method done for the time series data represented in orange.



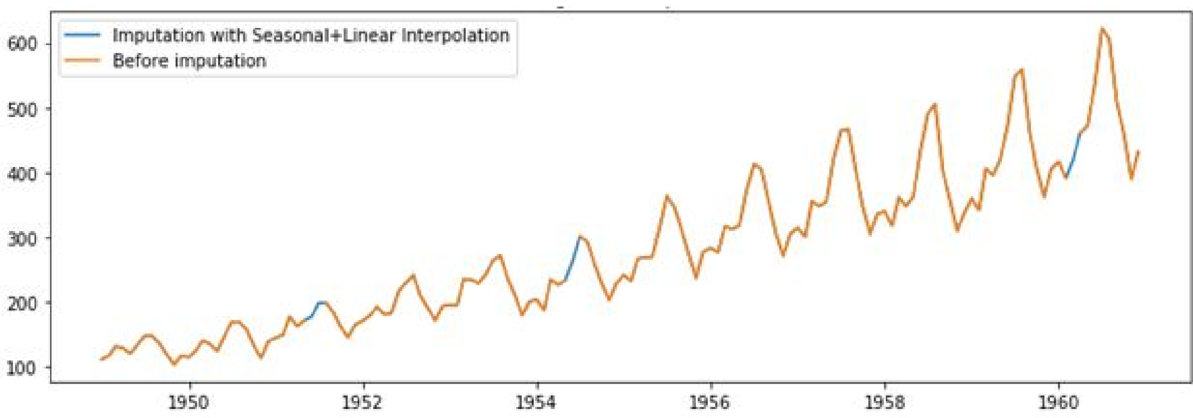
2. **Last observation carried forward:** We impute the missing values with its previous value in the data. In the following graph, the blue curves indicate the values imputed using the above method for the time series data represented in orange.



3. **Linear interpolation:** You draw a straight line joining the next and previous points of the missing values in the data. In the following graph, the blue curves indicate the values imputed using the linear interpolation method done for the time series data represented in orange.



4. **Seasonal + Linear interpolation:** This method is best applicable for the data with trend and seasonality. Here, the missing value is imputed with the average of the corresponding data point in the previous seasonal period and the next seasonal period of the missing value.

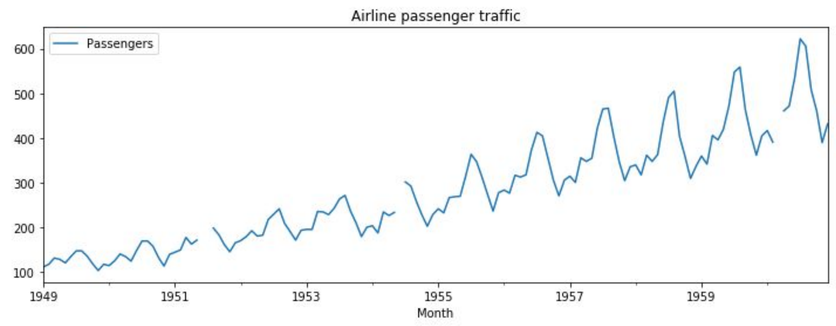


The above methods of imputing missing values are used in the airline passenger dataset shared and below are the snippets from the code. Before that, below shown is the airline passenger data with some missing values in it

Time series analysis

Plot time series data

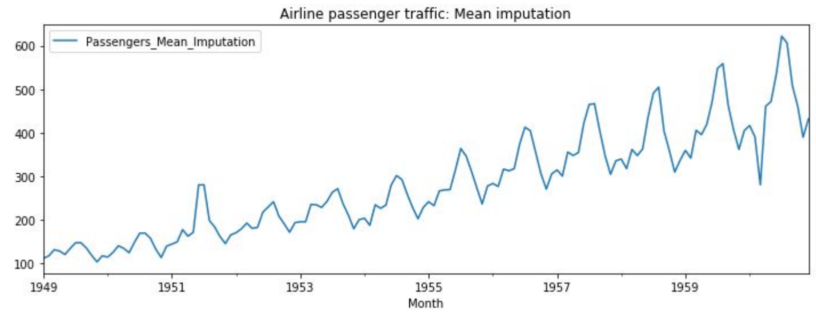
```
In [3]: data.plot(figsize=(12, 4))
        plt.legend(loc='best')
        plt.title('Airline passenger traffic')
        plt.show(block=False)
```



Missing value treatment

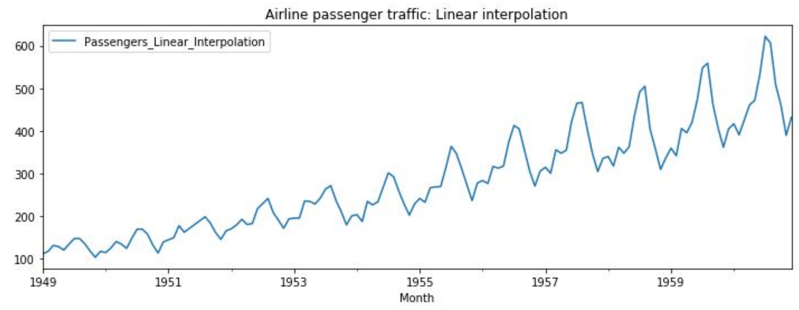
Mean imputation

```
In [4]: data = data.assign(Passengers_Mean_Imputation=data.Passengers.fillna(data.Passengers.mean()))
        data[['Passengers_Mean_Imputation']].plot(figsize=(12, 4))
        plt.legend(loc='best')
        plt.title('Airline passenger traffic: Mean imputation')
        plt.show(block=False)
```



Linear interpolation

```
In [5]: data = data.assign(Passengers_Linear_Interpolation=data.Passengers.interpolate(method='linear'))
        data[['Passengers_Linear_Interpolation']].plot(figsize=(12, 4))
        plt.legend(loc='best')
        plt.title('Airline passenger traffic: Linear interpolation')
        plt.show(block=False)
```



As you can see from the above plots, the linear interpolation method works better for the airline passenger dataset.

Use linear interpolation to impute missing values

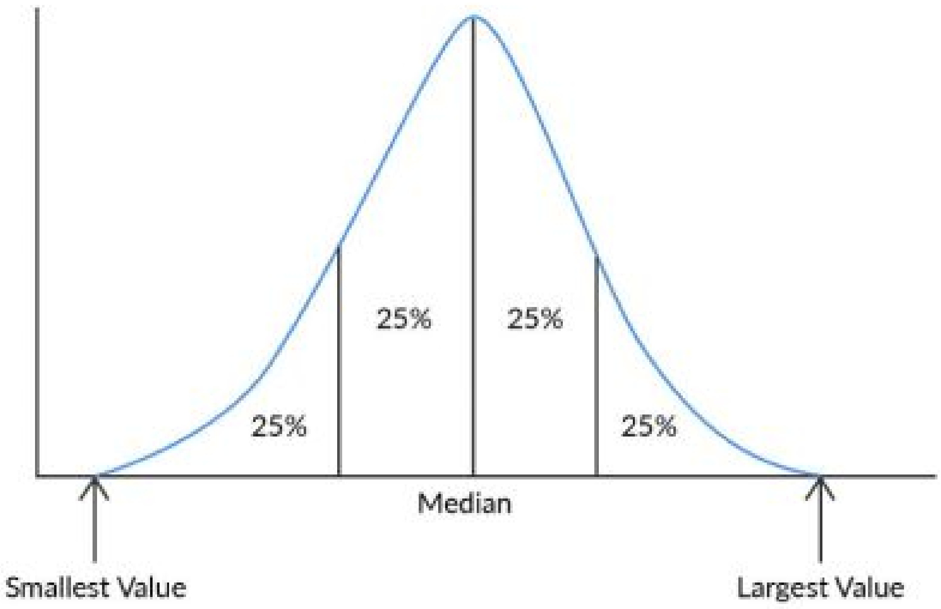
```
In [6]: data['Passengers'] = data['Passengers_Linear_Interpolation']
        data.drop(columns=['Passengers_Mean_Imputation','Passengers_Linear_Interpolation'],inplace=True)
```

Handling Outliers

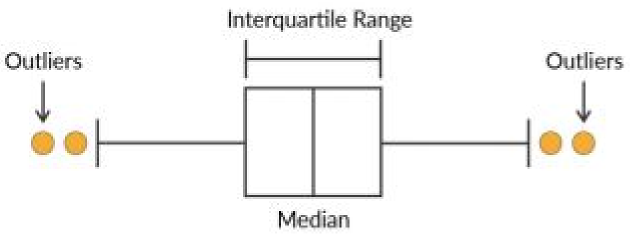
Apart from checking for missing values and imputing them if found any, it is also important to check the dataset for outliers as well before building the models. This will help us analyse the forecast plots more accurately and treat some outliers in case any value is present in the time-series data.

The following methods of detecting outliers in any time series data are helpful:

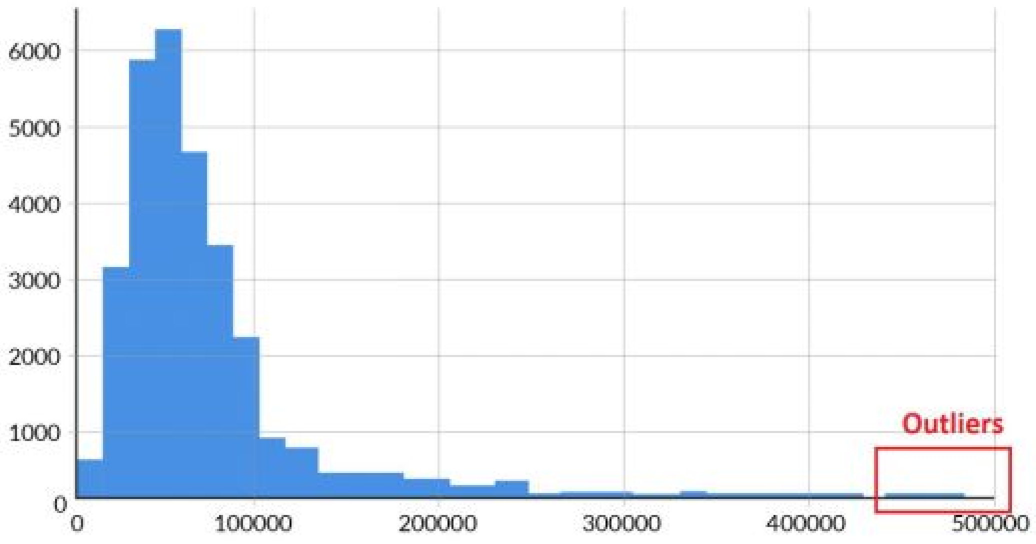
1. **Extreme value analysis:** Remove the smallest and largest values in the dataset.



2. **Box Plot:** The points lying on either side of the whiskers are considered to be outliers as shown in the image. The length of these whiskers is subjective and can be defined by you according to the problem.



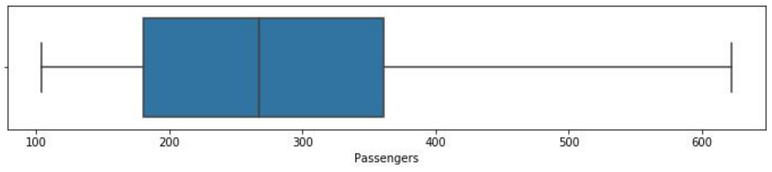
3. **Histogram:** Simply plotting a histogram can also reveal the outliers - basically the extreme values with low frequencies visible in the plot.



The airline passenger dataset is checked for outliers using the code shared below.

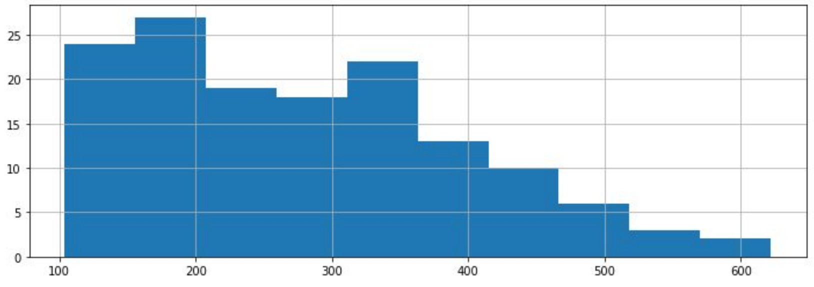
Box plot and interquartile range

```
In [7]: import seaborn as sns
        fig = plt.subplots(figsize=(12, 2))
        ax = sns.boxplot(x=data['Passengers'],whis=1.5)
```



Histogram plot

```
In [8]: fig = data.Passengers.hist(figsize = (12,4))
```



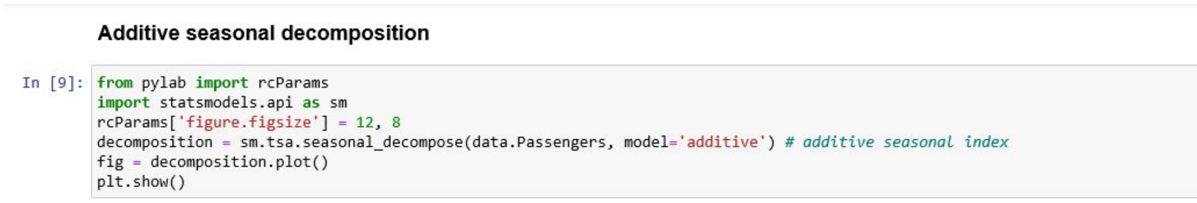
And there are no outliers in the dataset and hence there is no outlier treatment needed in this case.

A time-series data mostly has two major components - trend and seasonality. The data can be decomposed to extract these two components individually.

There are two ways in which the time series data can be decomposed that is additive seasonal decomposition and multiplicative seasonal decomposition.

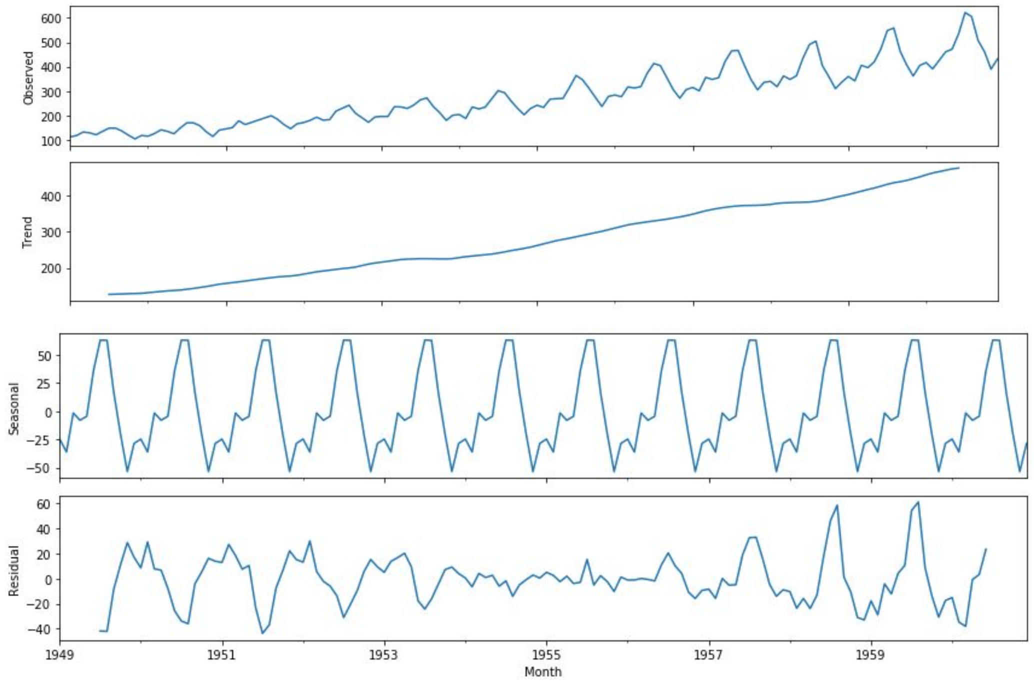
- **Additive Seasonal Decomposition** - the individual components in any time series data can be added to get the time-series data. This decomposition is used when the magnitude of the seasonal pattern in the data does not directly correlate with the value of the series, the additive seasonal decomposition may be a better choice to split the time series so that the residual does not have any pattern.

  For the airline passenger dataset, we used the additive decomposition in the following manner.



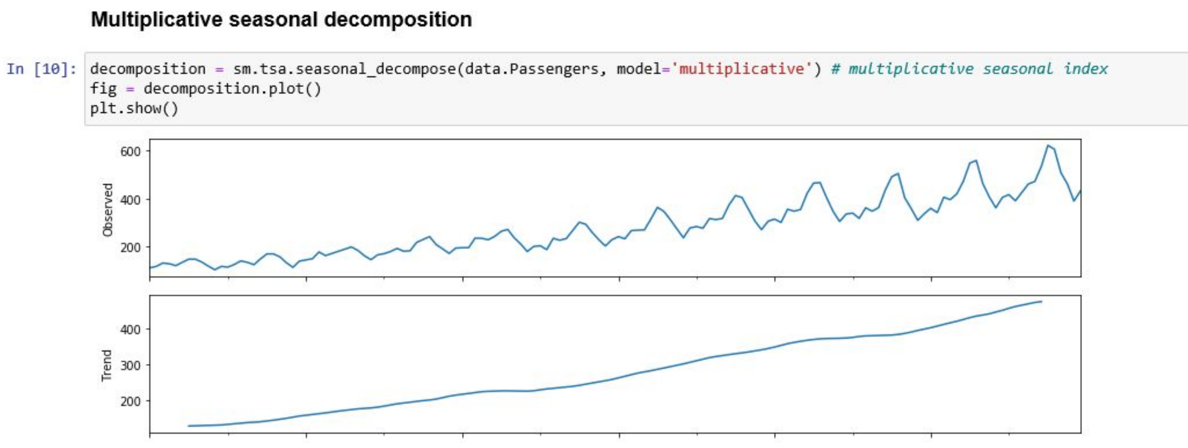
Additive seasonal decomposition

In [9]:
```
from pylab import rcParams
import statsmodels.api as sm
rcParams['figure.figsize'] = 12, 8
decomposition = sm.tsa.seasonal_decompose(data.Passengers, model='additive') # additive seasonal index
fig = decomposition.plot()
plt.show()
```



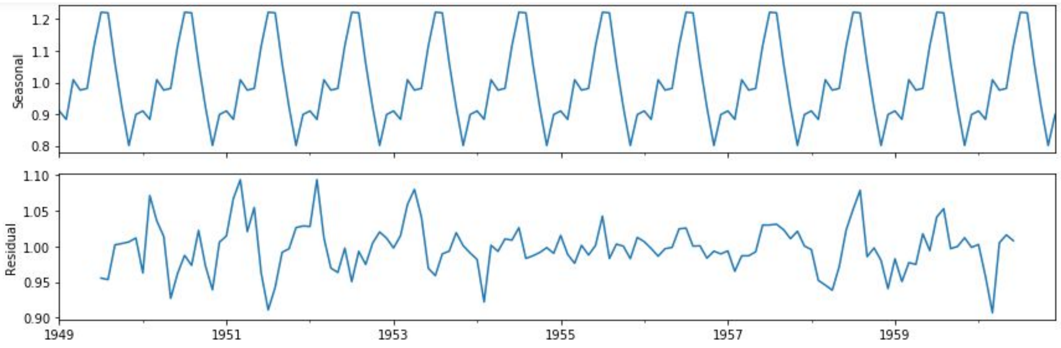
- **Multiplicative Seasonal Decomposition** - the individual components in any time series data can be multiplied to get the time-series data. This decomposition is used when the magnitude of the seasonal pattern in the data increases with an increase in data values and decreases with a decrease in the data values, the multiplicative seasonal decomposition may be a better choice.

  For the airline passenger dataset, we used the multiplicative decomposition in the following manner.

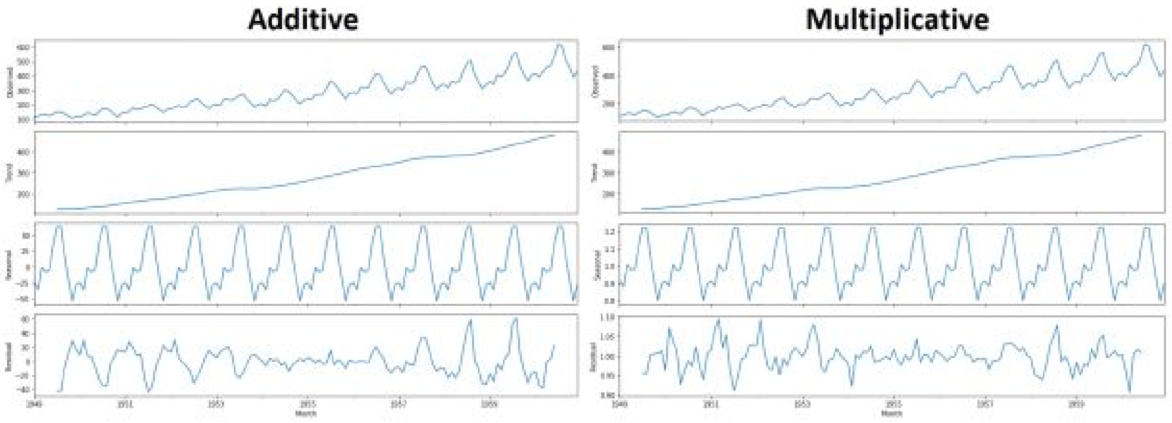


Multiplicative seasonal decomposition

```
In [10]: decomposition = sm.tsa.seasonal_decompose(data.Passengers, model='multiplicative') # multiplicative seasonal index
         fig = decomposition.plot()
         plt.show()
```



The below images show an instance of using the additive seasonal and multiplicative seasonal decomposition used on a time series data.



Smoothing Techniques

Let us first understand how forecasting for a time series data is different from simple regression methods and there are various reasons why this cannot work.

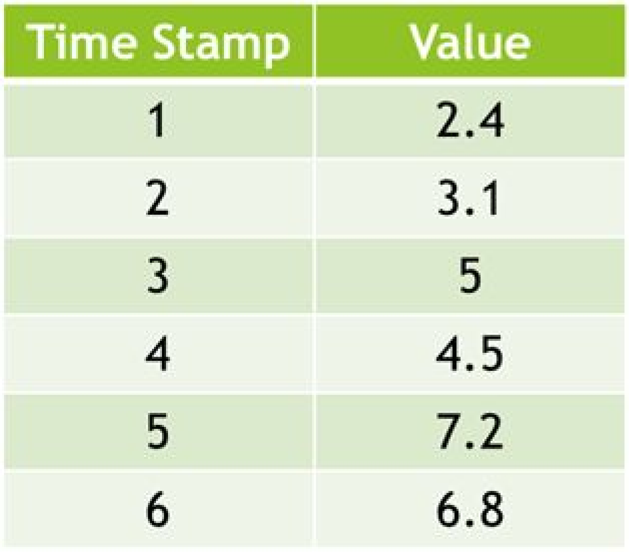
Regression vs Time Series

Time series is a series of time-stamped values. In other words, it is a sequence of values with time values attached to it. Here, the order attached to the values is very important which is not the case with a normal regression model.

Using time series analysis, you can forecast:

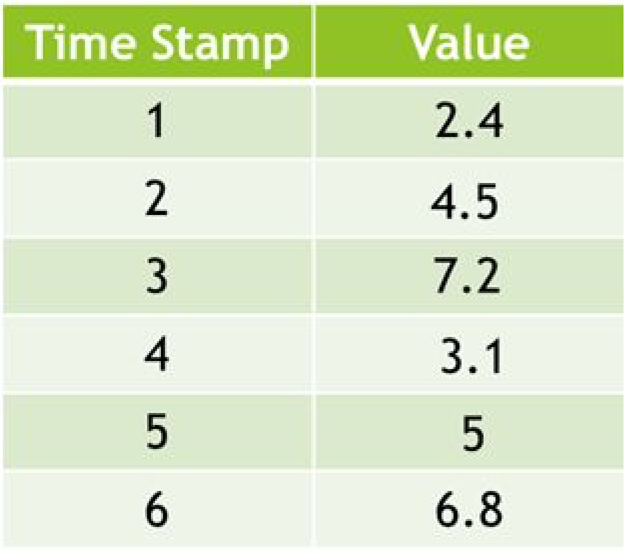
1. the value of the stock market index for a future month, or
2. the value of the literacy rate for a future census
3. the population of a nation in the coming year

You can’t just use regression or advanced regression to make a forecast, taking time as an independent variable. This will not work due to various reasons. One reason is that in a time series, the sequence is important. For example, let’s take the data provided below.



Time Stamp	Value
1	2.4
2	3.1
3	5
4	4.5
5	7.2
6	6.8

Using regression or advanced regression, let’s say you predict the value for timestamp 7. Now let’s say you shuffle the data around like this.



Time Stamp	Value
1	2.4
2	4.5
3	7.2
4	3.1
5	5
6	6.8

Even though we shuffled the data in the value's column, linear regression works on the linear relationship between the variables and thus the above data will also give you the same prediction for timestamp 7 if you use regression on the second table. However, a time series analysis will give you different forecasts for the original data and for the shuffled one.

Why does this happen?

This happens because while forecasting using time series, your model predicts not only on the basis of the values given but also on the basis of the sequence in which the values are given. Hence, the sequence is very important in a time series analysis and should not be played around with.

So, the two most important differences between time series and regression are:

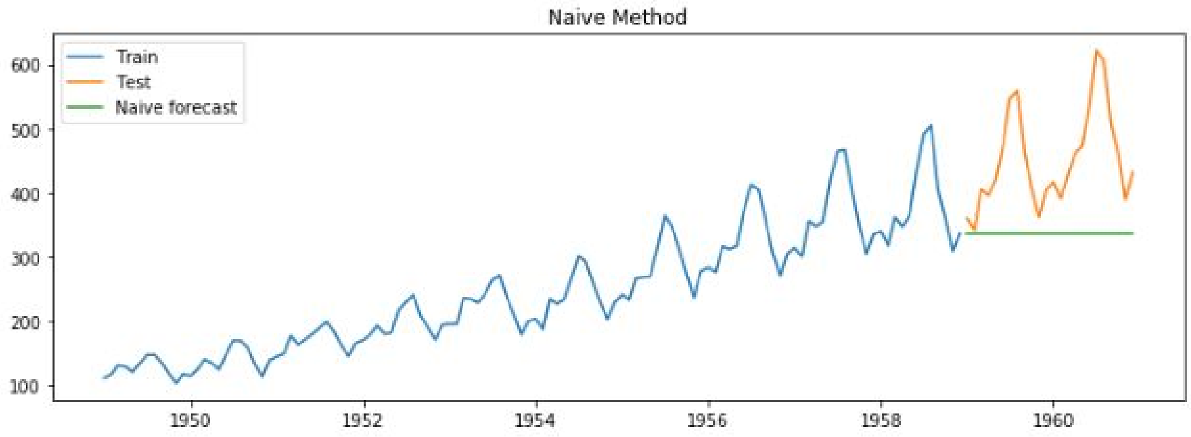
1. Time series have a strong temporal (time-based) dependence — each of these data sets essentially consists of a series of time-stamped observations i.e. each observation is tied to a specific time instance. Thus, unlike regression, the order of the data is important in a time series.
2. In a time series, you are not concerned with the causal relationship between the response and explanatory variables. The cause behind the changes in the response variable is very much a black box.

For example, let's say you want to predict what the value of the stock market index will be next month. You will not look at why the stock market index increases in value or if it's because of an increase in GDP or there are some changes in any sector or some other factor. You will only look at the sequence of values for the past months and predict for the next month, based on that sequence.

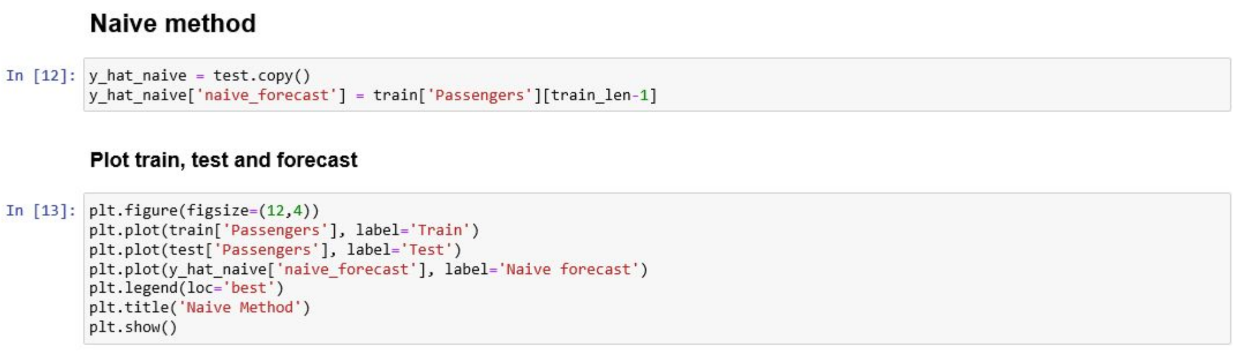
Basic Forecast Models

The basic forecasting methods are as follows

- Naive method
    - Forecast = Last month's sales



The application of the Naive method in python for the airline passenger dataset is shown below.

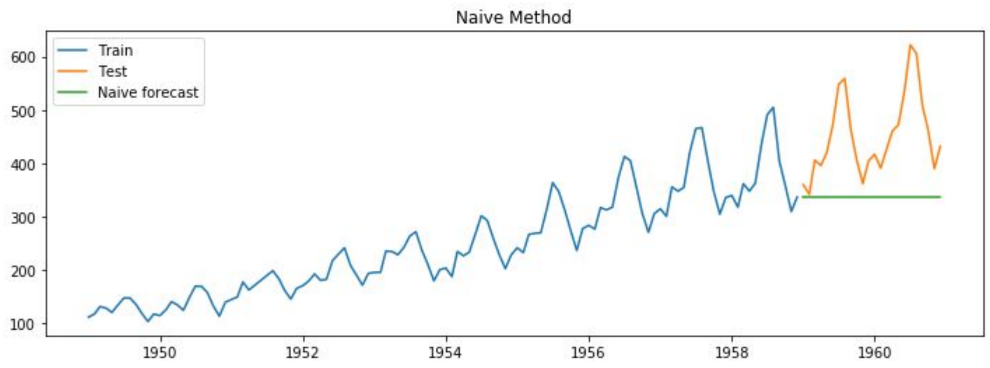


Naive method

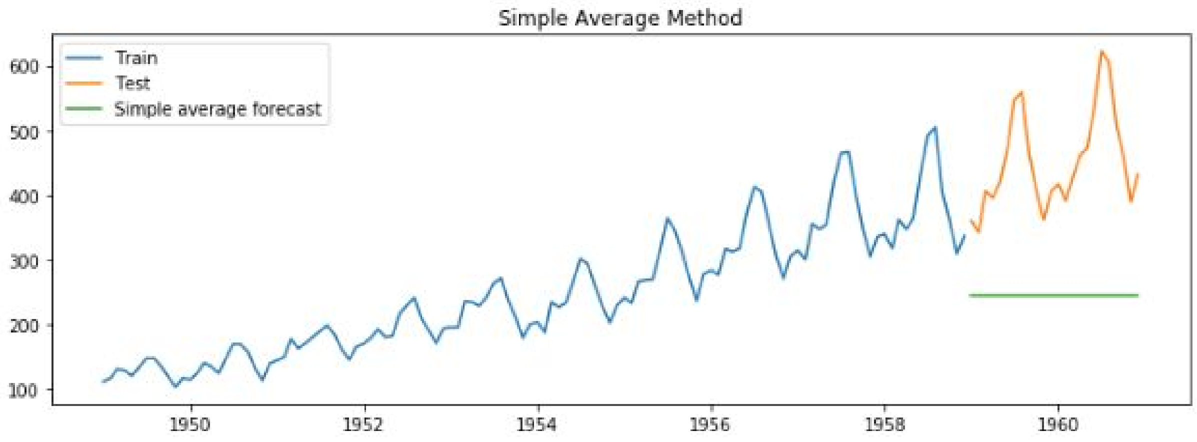
```
In [12]: y_hat_naive = test.copy()
         y_hat_naive['naive_forecast'] = train['Passengers'][train_len-1]
```

Plot train, test and forecast

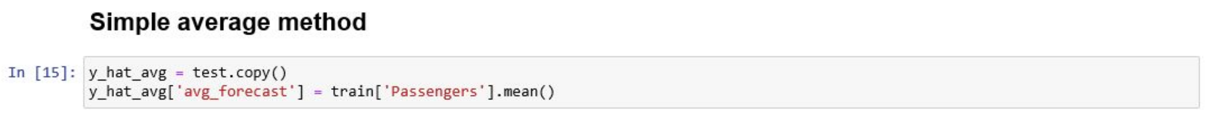
```
In [13]: plt.figure(figsize=(12,4))
         plt.plot(train['Passengers'], label='Train')
         plt.plot(test['Passengers'], label='Test')
         plt.plot(y_hat_naive['naive_forecast'], label='Naive forecast')
         plt.legend(loc='best')
         plt.title('Naive Method')
         plt.show()
```



- Simple average method
    - Forecast = Average of all past months' sales



The application of the Simple average method in python for the airline passenger dataset is shown below.

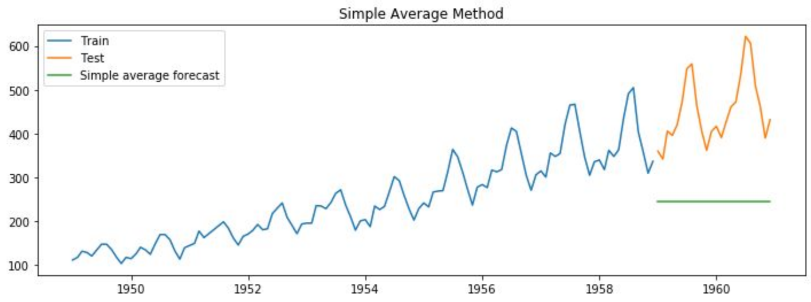


Simple average method

```
In [15]: y_hat_avg = test.copy()
         y_hat_avg['avg_forecast'] = train['Passengers'].mean()
```

Plot train, test and forecast

```
In [16]: plt.figure(figsize=(12,4))
         plt.plot(train['Passengers'], label='Train')
         plt.plot(test['Passengers'], label='Test')
         plt.plot(y_hat_avg['avg_forecast'], label='Simple average forecast')
         plt.legend(loc='best')
         plt.title('Simple Average Method')
         plt.show()
```



Error Measures

- **Mean Forecast Error (MFE):** In this naive method, you simply subtract the actual values of the dependent variable, i.e., 'y' with the forecasted values of 'y'. This can be represented using the equation below.

$$MFE = \frac{1}{n}\sum_{i=1}^{n}(y_{actual} - \hat{y}_{forecast})$$

- **Mean Absolute Error (MAE):** Since MFE might cancel out a lot of overestimated and underestimated forecasts, hence measuring the mean absolute error or MAE makes more sense as in this method, you take the absolute values of the difference between the actual and forecasted values.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y_{actual} - \hat{y}_{forecast}\right|$$

- **Mean Absolute Percentage Error (MAPE):** The problem with MAE is that even if you get an error value, you have nothing to compare it against. For example, if the MAE that you get is 1.5, you cannot tell just on the basis of this number whether you have made a good forecast or not. If the actual values are in single digits, this error of 1.5 is obviously high but if the actual values are, say in the order of thousands, an error of 1.5 indicates a good forecast. So in order to capture how the forecast is doing based upon the actual values, you evaluate mean absolute error where you take the mean absolute error (MAE) as the percentage of the actual values of 'y'.

$$MAPE = \frac{100}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y_i}}{y_i} \right|$$

- **Mean Squared Error (MSE):** The idea behind mean squared error is the same as mean absolute error, i.e., you want to capture the absolute deviations so that the negative and positive deviations do not cancel each other out. In order to achieve this, you simply square the error values, sum them up and take their average. This is known as mean squared error or MSE which can be represented using the equation below.

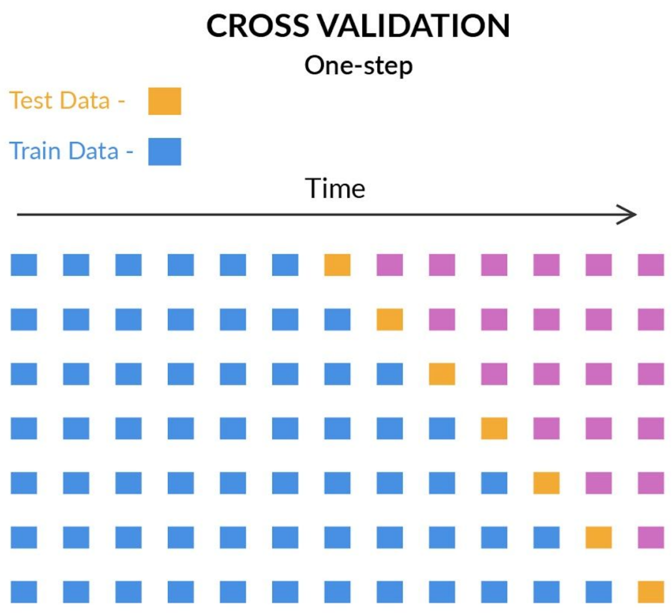
$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_{actual} - \hat{y}_{predicted})^2$$

- **Root Mean Squared Error (RMSE):** Since the error term you get from MSE is not in the same dimension as the target variable 'y' (it is squared), you deploy a metric known as RMSE wherein you take the square root of the MSE value obtained.

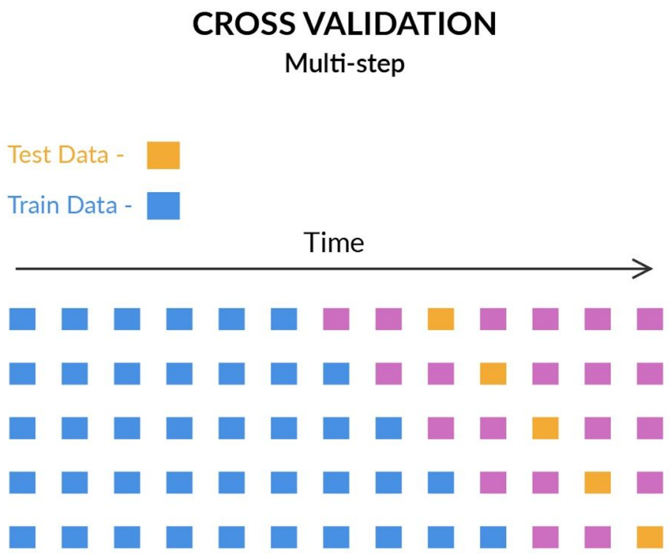
$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_{actual} - \hat{y}_{predicted})^2}$$

Let us now also understand about cross validation for any time-series data. Basically cross-validation in time series is not done in the same way you might have done for any of the classical machine learning algorithms. This caveat stems from the fact that order matters in time series. So while building a forecast model, the test dataset is always on the right-hand side of the train dataset. Let's understand the two types of validation you learnt just now.

- **One-Step Validation**
  - The testing set is just one step ahead to the training set. So suppose out of 15 data points, you decided to keep the first 10 of them as 'train' and the next 5 of them as 'test'. Now, the data points in the test set will be taken one-by-one starting from the left since you need all the previous values to predict the future values. So firstly, you take the 11th point to be the test set. Once this value is successfully forecasted, you move on to the 12th point which is now your new test point and so on until you forecast for all the 5 points. This idea is represented in the image below. Here the blue squares represent the train data, the yellow squares represent the test data, and the purple ones are the future values for which the initial test points need to be predicted first. In the image below, you have 7 test data points, hence it takes 7 iterations (or forecasts) to fully predict the test set.



- **Multi-Step Validation**
    - This is the same as one-step validation, the only difference being that you do not consider a few points to the immediate right of the last training datapoint but rather skip a few of these points to make forecasts well into the future. This can be seen in the image below.

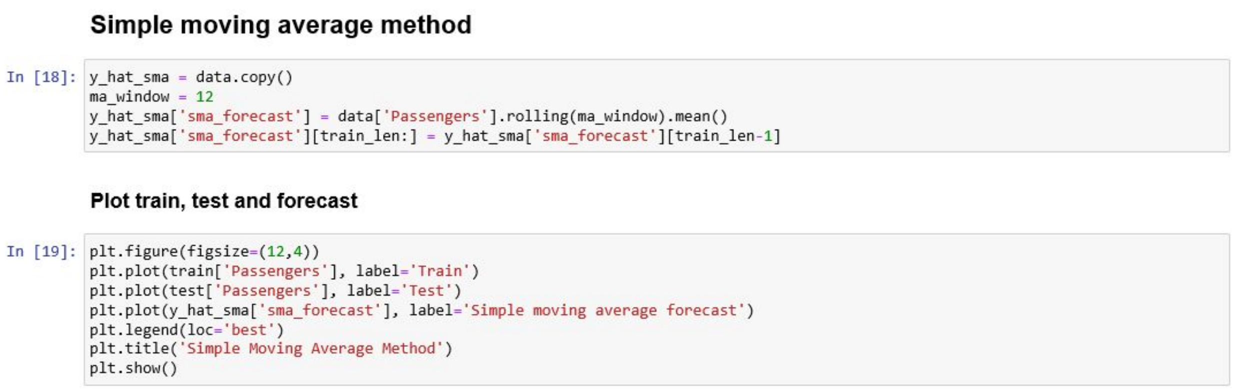


Simple Moving Average Method

Considering that the last observation in the time series has more impact on the future rather than the first observation, in the simple moving average method, we take the average of only the last few observations to forecast the future.

In this method, the forecasts are calculated using the average of the time-series data in the moving window considered. The window of the past observations in the time series data keeps moving, and hence the average values keep changing. This helps in forecasting values at every step in the dataset. This is helpful, because in any time series data, the recent observations have more influence on the forecasts than the previous observations. Thus a smaller window size generally forecasts closer to the actual values as it is able to take the variations in the time series data of the recent values. The higher the window size, these variations get distributed and as a result, the forecasts may not be very close to the actual values.

The simple moving average method modelled in python for the airline passenger traffic dataset is shown below

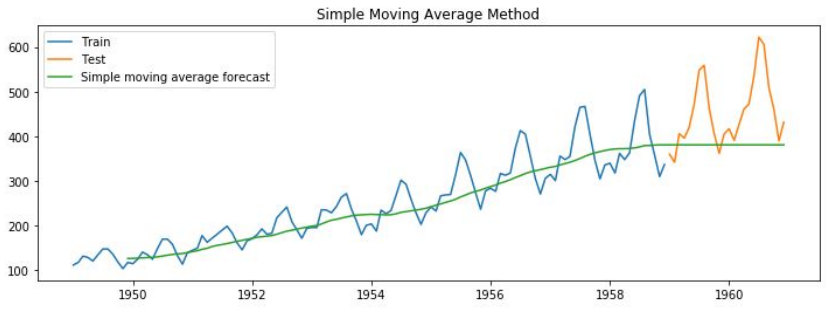


Simple moving average method

```
In [18]: y_hat_sma = data.copy()
         ma_window = 12
         y_hat_sma['sma_forecast'] = data['Passengers'].rolling(ma_window).mean()
         y_hat_sma['sma_forecast'][train_len:] = y_hat_sma['sma_forecast'][train_len-1]
```

Plot train, test and forecast

```
In [19]: plt.figure(figsize=(12,4))
         plt.plot(train['Passengers'], label='Train')
         plt.plot(test['Passengers'], label='Test')
         plt.plot(y_hat_sma['sma_forecast'], label='Simple moving average forecast')
         plt.legend(loc='best')
         plt.title('Simple Moving Average Method')
         plt.show()
```



Calculate RMSE and MAPE

In [20]:
```
rmse = np.sqrt(mean_squared_error(test['Passengers'], y_hat_sma['sma_forecast'][train_len:])).round(2)
mape = np.round(np.mean(np.abs(test['Passengers']-y_hat_sma['sma_forecast'][train_len:])/test['Passengers'])*100,2)

tempResults = pd.DataFrame({'Method':['Simple moving average forecast'], 'RMSE': [rmse],'MAPE': [mape] })
results = pd.concat([results, tempResults])
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[20]:

	Method	RMSE	MAPE
0	Naive method	137.51	23.63
0	Simple average method	219.69	44.28
0	Simple moving average forecast	103.33	15.54

Simple Exponential Smoothing Technique

In the weighted average technique, the underlying idea is that each observation influencing y_{t+1} is assigned a specific weight. More recent observations get more weight, whereas the previous observations get less weight. Suppose you consider a time series data of the previous 12 months and are forecasting $\mathbf{y_{t+1}}$. Then, the weighted moving average will be calculated as follows:

$$y_{t+1} = \frac{a.y_1 + b.y_2 + c.y_3 + ... + k.y_{12}}{(a + b + c + ... + k)}$$

such that **a < b < c < .....< k**, where k is the largest weight assigned to the most recent data point i.e. $\mathbf{y_{12}}$.

Now, you also know by now that any time-series data primarily consists of the following three components:

1. Level
2. Trend
3. Seasonality

We started with the simple exponential smoothing technique.This helps us forecast the level in the time series data. The forecast observation data, $\mathbf{y_{t+1}}$, is a function of the level component that is denoted by $\mathbf{l_t}$. Here, the level component is written as follows:

$$l_t = \alpha.y_t + (1 - \alpha).l_{t-1}$$

The most recent value it takes a weight of **α**, also known as the level smoothing parameter, whereas the previous observation's level component takes the value of **1−α**.

The values of **α** lie between **0** and **1**.

You will notice that each level term for $\mathbf{y_t, y_{t,-1}, y_{t,-2}}$ and so on can be written as follows:

$$\begin{aligned} l_t &= \alpha.y_t + (1 - \alpha).l_{t-1} \\ l_{t-1} &= \alpha.y_{t-1} + (1 - \alpha).l_{t-2} \\ l_{t-2} &= \alpha.y_{t-2} + (1 - \alpha).l_{t-3} \end{aligned}$$

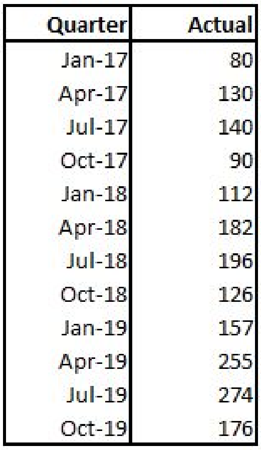
and so on.

Once you replace all the level terms in the forecast equation, you will obtain the following:

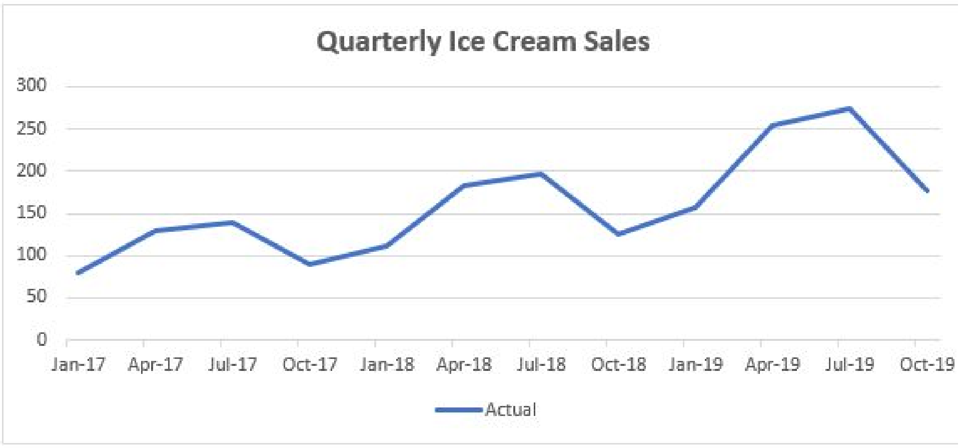
$$\hat{y}_{t+1} = \alpha.y_t + \alpha.(1-\alpha.)y_{t-1} + \alpha.(1-\alpha.)^2 y_{t-2}$$

Here, notice that the value of the weight assigned to every observation decreases in an exponential manner such as **$(1-\alpha) < (1-\alpha)^2 < (1-\alpha)^3$**. Hence, this technique is called an exponential smoothing technique.

For example, look at the time series plot below for a quarterly ice cream sales data.



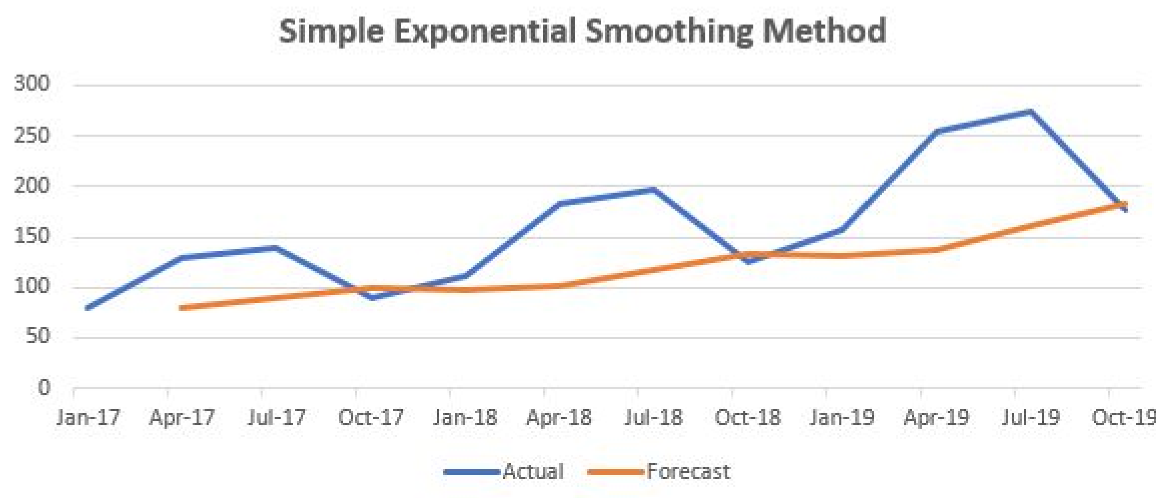
Quarter	Actual
Jan-17	80
Apr-17	130
Jul-17	140
Oct-17	90
Jan-18	112
Apr-18	182
Jul-18	196
Oct-18	126
Jan-19	157
Apr-19	255
Jul-19	274
Oct-19	176



Using the Simple Exponential Smoothing Technique and applying the equations mentioned above by using the level smoothing parameter as 0.2, we can get the forecast values along with the plot as below.

alpha	0.2

Quarter	Actual	Level l(t)	Forecast
Jan-17	80	80	
Apr-17	130	90	80
Jul-17	140	100	90
Oct-17	90	98	100
Jan-18	112	101	98
Apr-18	182	117	101
Jul-18	196	133	117
Oct-18	126	131	133
Jan-19	157	137	131
Apr-19	255	160	137
Jul-19	274	183	160
Oct-19	176	182	183



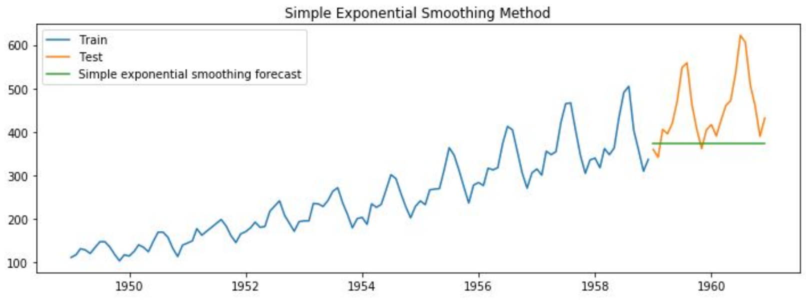
The simple exponential smoothing method for the airline passenger dataset is shown below.

Simple exponential smoothing

In [1]:
```
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
model = SimpleExpSmoothing(train['Passengers'])
model_fit = model.fit(smoothing_level=0.2,optimized=False)
model_fit.params
y_hat_ses = test.copy()
y_hat_ses['ses_forecast'] = model_fit.forecast(24)
```

Plot train, test and forecast

In [22]:
```
plt.figure(figsize=(12,4))
plt.plot(train['Passengers'], label='Train')
plt.plot(test['Passengers'], label='Test')
plt.plot(y_hat_ses['ses_forecast'], label='Simple exponential smoothing forecast')
plt.legend(loc='best')
plt.title('Simple Exponential Smoothing Method')
plt.show()
```



In [23]:
```
rmse = np.sqrt(mean_squared_error(test['Passengers'], y_hat_ses['ses_forecast'])).round(2)
mape = np.round(np.mean(np.abs(test['Passengers']-y_hat_ses['ses_forecast'])/test['Passengers'])*100,2)

tempResults = pd.DataFrame({'Method':['Simple exponential smoothing forecast'], 'RMSE': [rmse],'MAPE': [mape] })
results = pd.concat([results, tempResults])
results
```

Out[23]:

	Method	RMSE	MAPE
0	Naive method	137.51	23.63
0	Simple average method	219.69	44.28
0	Simple moving average forecast	103.33	15.54
0	Simple exponential smoothing forecast	107.65	16.49

Holt's Exponential Smoothing Technique

Taking a step further, Holt's exponential smoothing technique forecasts the level and trend of the time series data. Now the forecast equation is a function of both level and trend, that is,

$$\hat{y}_{t+1} = l_t + b_t$$

Here l_t is the level component, and b_t is the trend component.

Here, the trend component is calculated as follows:

$$\hat{b}_t = \beta(l_t - l_{t-1}) + (1 - \beta).b_{t-1}$$

Here β is the smoothing parameter for trend. In the second equation given above, the difference in the level components in the recent observation shows the trend of the recent value, which is assigned a weight of β , whereas the trend values of the previous observations are assigned a weight of 1−β .

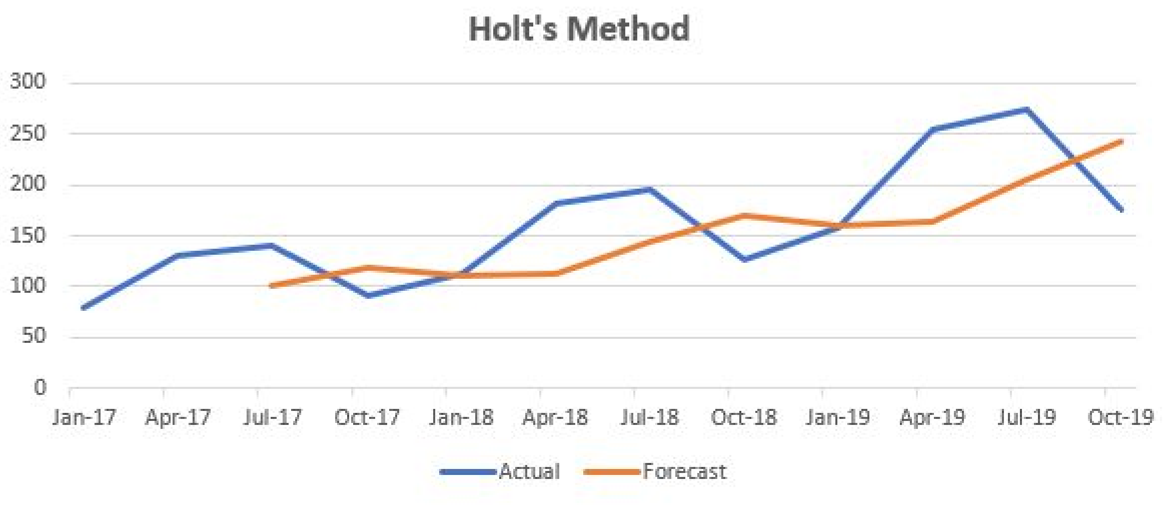
The equation for the level component remains the same with minor addition of the previous value's trend component in the calculation of the previous value's level component.

$$l_t = \alpha.y_t + (1 - \alpha).(l_{t-1} + b_{t-1})$$

For the same quarterly ice cream sales example used above, upon using the Holt's exponential smoothing technique, and using the level and trend parameters as 0.2 each, we get the forecast values and the plot as below.

alpha	0.2
beta	0.2

Quarter	Actual	Level l(t)	Trend b(t)	Forecast
Jan-17	80	80		
Apr-17	130	90	10	
Jul-17	140	108	10	100
Oct-17	90	112	-2	118
Jan-18	112	111	3	110
Apr-18	182	127	16	114
Jul-18	196	154	16	143
Oct-18	126	161	-1	170
Jan-19	157	159	5	160
Apr-19	255	182	24	164
Jul-19	274	220	23	206
Oct-19	176	229	-1	242



Holt's exponential smoothing method for the airline passenger dataset is shown below.

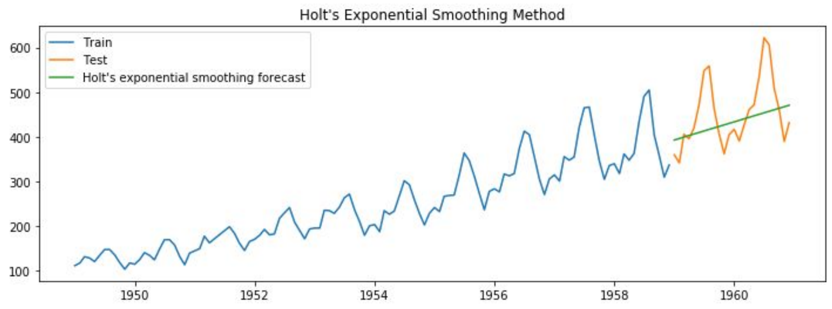
Holt's method with trend

In [24]:
```
from statsmodels.tsa.holtwinters import ExponentialSmoothing
model = ExponentialSmoothing(np.asarray(train['Passengers']) ,seasonal_periods=12 ,trend='additive', seasonal=None)
model_fit = model.fit(smoothing_level=0.2, smoothing_slope=0.01, optimized=False)
print(model_fit.params)
y_hat_holt = test.copy()
y_hat_holt['holt_forecast'] = model_fit.forecast(len(test))
```

```
{'smoothing_level': 0.2, 'smoothing_slope': 0.01, 'smoothing_seasonal': None, 'damping_slope': nan, 'initial_level': 112.0, 'in
itial_slope': 6.0, 'initial_seasons': array([], dtype=float64), 'use_boxcox': False, 'lamda': None, 'remove_bias': False}
```

Plot train, test and forecast

In [25]:
```
plt.figure(figsize=(12,4))
plt.plot( train['Passengers'], label='Train')
plt.plot(test['Passengers'], label='Test')
plt.plot(y_hat_holt['holt_forecast'], label='Holt\'s exponential smoothing forecast')
plt.legend(loc='best')
plt.title('Holt\'s Exponential Smoothing Method')
plt.show()
```



Calculate RSME and MAPE

In [26]:
```
rmse = np.sqrt(mean_squared_error(test['Passengers'], y_hat_holt['holt_forecast'])).round(2)
mape = np.round(np.mean(np.abs(test['Passengers']-y_hat_holt['holt_forecast'])/test['Passengers'])*100,2)

tempResults = pd.DataFrame({'Method':['Holt\'s exponential smoothing method'], 'RMSE': [rmse],'MAPE': [mape] })
results = pd.concat([results, tempResults])
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[26]:

	Method	RMSE	MAPE
0	Naive method	137.51	23.63
0	Simple average method	219.69	44.28
0	Simple moving average forecast	103.33	15.54
0	Simple exponential smoothing forecast	107.65	16.49
0	Holt's exponential smoothing method	71.94	11.11

Holt-Winters' Exponential Smoothing Technique

The Holt-Winters' smoothing technique forecasts the level, trend as well as the seasonality for a time series data.. The forecast equation now has seasonal components, including level and trend i.e.

$$y_{t+1} = l_t + b_t + S_{t+1-m}$$

Here, m is the number of times a season repeats during a period. The seasonal component is calculated using the following equation:

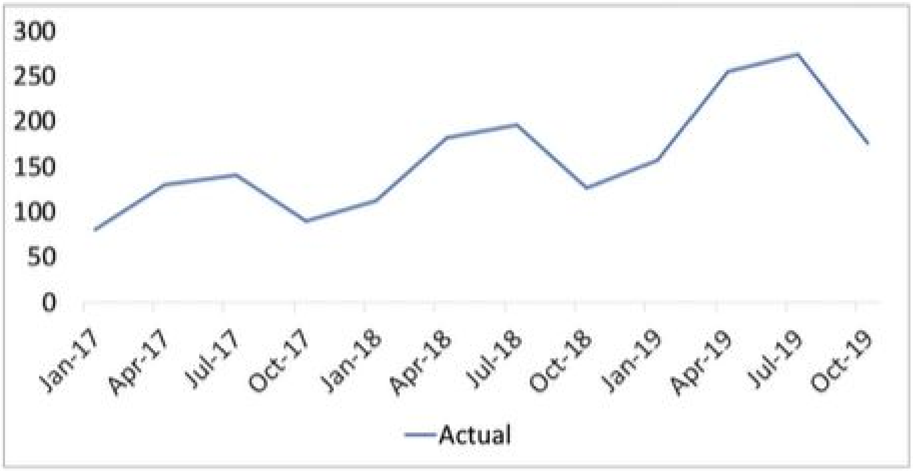
$$S_t = \gamma.(y_t - l_{t-1} - b_{t-1}) + (1-\gamma).s_{t-m}$$

Where γ is the weight assigned to the seasonal component of the recent observations. Note that the term y_t - l_{t-1} - b_{t-1} is the best guess of the seasonal component of the recent observation that is obtained after subtracting the level and the trend components from yt. The weight of γ is assigned to this component, whereas 1-γ is the weight assigned to the seasonal component of the recent observations.

The trend and the level equations respectively are as follows:

$$\hat{b}_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$
$$l_t = \alpha.(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1})$$

There are two methods of performing the Holt-Winters' smoothing techniques: additive and multiplicative methods. In a time-series data, if the seasonality is not a function of the level component or the difference between subsequent troughs of the time series data does not increase as you progress in the graph, then the Holt-Winters' additive method works best. You use this method for the quarterly ice cream sales example because as you can observe in the image below, the difference between the troughs does not increase.

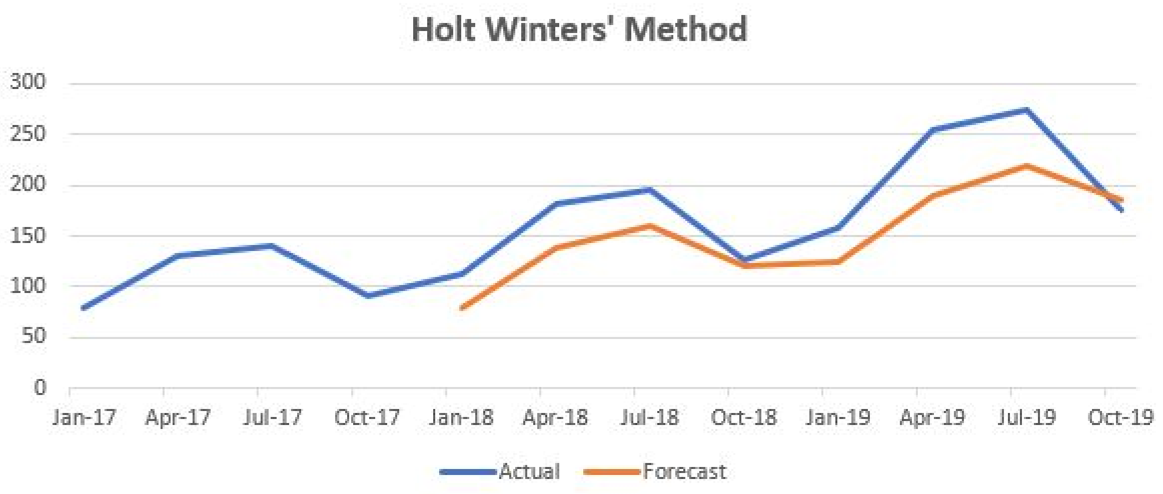


But suppose seasonality is a function of the level and the difference between the troughs of the time series data increases as you progress in the graph, then you use the multiplicative method.

For the same quarterly ice-cream sales example used in the previous methods, using the Holts' method and setting the level, trend and seasonality parameter as 0.2 each, we get the forecast values and plots as shown below.

alpha	0.2
beta	0.2
gamma	0.2

Quarter	Actual	Level l(t)	Trend b(t)	Season s(t)	Forecast
Jan-17	80			-30	
Apr-17	130			20	
Jul-17	140			30	
Oct-17	90	110		-20	
Jan-18	112	116	1	-24	80
Apr-18	182	127	3	29	138
Jul-18	196	137	5	37	160
Oct-18	126	142	5	-19	121
Jan-19	157	154	6	-17	123
Apr-19	255	173	9	42	189
Jul-19	274	193	11	48	219
Oct-19	176	202	11	-21	185



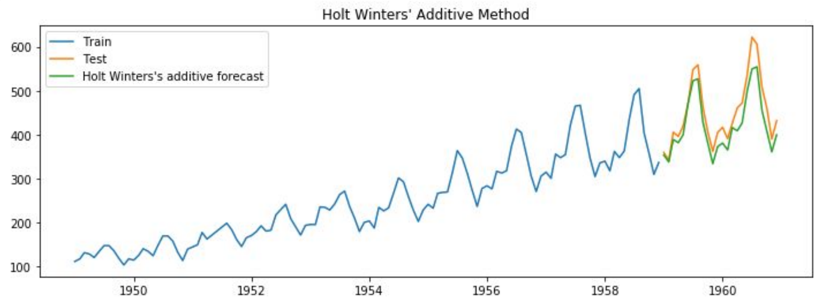
Holt-Winters' additive method for the airline passenger dataset is shown below.

Holt Winters' additive method with trend and seasonality

```
In [27]: y_hat_hwa = test.copy()
         model = ExponentialSmoothing(np.asarray(train['Passengers']) ,seasonal_periods=12 ,trend='add', seasonal='add')
         model_fit = model.fit(optimized=True)
         print(model_fit.params)
         y_hat_hwa['hw_forecast'] = model_fit.forecast(24)
```

Plot train, test and forecast

```
In [28]: plt.figure(figsize=(12,4))
         plt.plot( train['Passengers'], label='Train')
         plt.plot(test['Passengers'], label='Test')
         plt.plot(y_hat_hwa['hw_forecast'], label='Holt Winters\'s additive forecast')
         plt.legend(loc='best')
         plt.title('Holt Winters\' Additive Method')
         plt.show()
```



Calculate RMSE and MAPE

```
In [29]: rmse = np.sqrt(mean_squared_error(test['Passengers'], y_hat_hwa['hw_forecast'])).round(2)
         mape = np.round(np.mean(np.abs(test['Passengers']-y_hat_hwa['hw_forecast'])/test['Passengers'])*100,2)

         tempResults = pd.DataFrame({'Method':['Holt Winters\' additive method'], 'RMSE': [rmse],'MAPE': [mape] })
         results = pd.concat([results, tempResults])
         results = results[['Method', 'RMSE', 'MAPE']]
         results
```

Out[29]:

	Method	RMSE	MAPE
0	Naive method	137.51	23.63
0	Simple average method	219.69	44.28
0	Simple moving average forecast	103.33	15.54
0	Simple exponential smoothing forecast	107.65	16.49
0	Holt's exponential smoothing method	71.94	11.11
0	Holt Winters' additive method	35.10	6.53

Holt-Winters' multiplicative method for the airline passenger dataset is shown below.

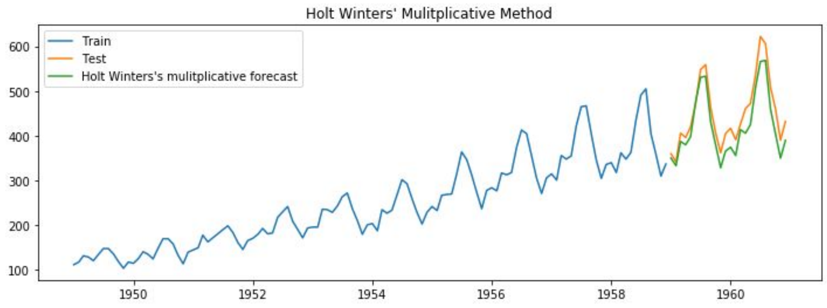
Holt Winter's multiplicative method with trend and seasonality

In [30]:
```
y_hat_hwm = test.copy()
model = ExponentialSmoothing(np.asarray(train['Passengers']) ,seasonal_periods=12 ,trend='add', seasonal='mul')
model_fit = model.fit(optimized=True)
print(model_fit.params)
y_hat_hwm['hw_forecast'] = model_fit.forecast(24)
```

```
{'smoothing_level': 0.38484064840698556, 'smoothing_slope': 0.035386453247783, 'smoothing_seasonal': 0.6151593186319729, 'dampi
ng_slope': nan, 'initial_level': 212.39830870038878, 'initial_slope': 1.1057967107993691, 'initial_seasons': array([0.51755725,
0.54595018, 0.60261532, 0.56966257, 0.52017629,
       0.5722879 , 0.62749981, 0.62520353, 0.56963344, 0.49468435,
       0.43937027, 0.51362925]), 'use_boxcox': False, 'lamda': None, 'remove_bias': False}
```

Plot train, test and forecast

In [31]:
```
plt.figure(figsize=(12,4))
plt.plot( train['Passengers'], label='Train')
plt.plot(test['Passengers'], label='Test')
plt.plot(y_hat_hwm['hw_forecast'], label='Holt Winters\'s mulitplicative forecast')
plt.legend(loc='best')
plt.title('Holt Winters\' Mulitplicative Method')
plt.show()
```



Calculate RMSE and MAPE

In [32]:
```
rmse = np.sqrt(mean_squared_error(test['Passengers'], y_hat_hwm['hw_forecast'])).round(2)
mape = np.round(np.mean(np.abs(test['Passengers']-y_hat_hwm['hw_forecast'])/test['Passengers'])*100,2)

tempResults = pd.DataFrame({'Method':['Holt Winters\' multiplicative method'], 'RMSE': [rmse],'MAPE': [mape] })
results = pd.concat([results, tempResults])
results = results[['Method', 'RMSE', 'MAPE']]
results
```

Out[32]:

	Method	RMSE	MAPE
0	Naive method	137.51	23.63
0	Simple average method	219.69	44.28
0	Simple moving average forecast	103.33	15.54
0	Simple exponential smoothing forecast	107.65	16.49
0	Holt's exponential smoothing method	71.94	11.11
0	Holt Winters' additive method	35.10	6.53
0	Holt Winters' multiplicative method	34.83	6.91