Subject Name: Machine Learning Laboratory **Semester: 6**

Subject Code: IS6LB2 **L-P-C: 0-3-1.5**

Course Objectives:

Sl. No	Course Objectives
1	Make use of Data sets in implementing the machine learning algorithms.
2	Implement the machine learning concepts and algorithms in any suitable language of choice.
3	Understand and present the key algorithms and theory that form the core of machine learning.
4	Analyze the performance of Machine Learning techniques which varies with the number of training examples presented.

Lab Cycles	Description
I	1. Write a Python program to load iris data set and apply Naïve-Bayes algorithm for classification of Iris flowers. 2. Write a Python program to extract social_network_ads.csv file. Apply k-Nearest Neighbor technique to identify the users who purchased the item or not. 3. Write a Python program to load whether data set and apply a perceptron learning algorithm to determine whether the rain occurs tomorrow or not.
II	4. Implement the Backpropagation algorithm in Python to classify iris data set. 5. Consider a Mall_Customers data set which is the data of customers who visit the mall and spend there. In the given dataset, we have Customer_Id, Gender, Age, Annual Income ($), and Spending Score (which is the calculated value of how much a customer has spent in the mall, the more the value, the more he has spent). From this dataset, calculate some patterns using k-Means clustering method. 6. Consider a dataset that has two variables: salary (dependent variable) and experience (Independent variable). Build a simple Linear-Regression model in Python to do the following: • Find out if there is any correlation between these two variables. • Find the best fit line for the dataset. • Show how the dependent variable is changing by changing the independent variable.
III	7. Implement Support Vector Machine algorithm in Python for any suitable data set available. 8. Consider the User Database which contains information about UserID, Gender, Age, EstimatedSalary, and Purchased. Apply Logistic Regression in Python to predict whether a user will purchase the company's newly launched product or not. 9. Implement Polynomial Regression model in Python for any suitable data set available.

Pattern for practical exam conduction:

In the examination each student picks one question out of 10 questions from the above question bank.

Course Outcomes:

Sl. No.	Descriptions
1	Understand the implementation procedures for the machine learning algorithms.
2	Design Python programs for various Learning algorithms.
3	Apply appropriate data sets to the Machine Learning algorithms.
4	Identify and apply Machine Learning algorithms to solve real world problems.

1. Write a Python program to load iris data set and apply Naïve-Bayes algorithm for classification of Iris flowers.

Overview of Naive Bayes Classification:

Naive Bayes is one such algorithm in classification that can never be overlooked upon due to its special characteristic of being "naive". It makes the assumption that features of a measurement are independent of each other.

For example, an animal may be considered as a cat if it has cat eyes, whiskers and a long tail. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this animal is a cat and that is why it is known as 'Naive'.

According to Bayes Theorem, the various features are mutually independent. For two independent events, P(A,B) = P(A)P(B). This assumption of Bayes Theorem is probably never encountered in practice, hence it accounts for the "naive" part in Naive Bayes. Bayes' Theorem is stated as: P(a|b) = (P(b|a) * P(a)) / P(b). Where P(a|b) is the probability of a given b.
Let us understand this algorithm with a simple example. The Student will be a pass if he wears a "red" color dress on the exam day. We can solve it using above discussed method of posterior probability.

By Bayes Theorem, P(Pass| Red) = P(Red| Pass) * P(Pass) / P (Red).
From the values, let us assume P (Red|Pass) = 3/9 = 0.33, P(Red) = 5/14 = 0.36, P(Pass)= 9/14 = 0.64. Now, P (Pass| Red) = 0.33 * 0.64 / 0.36 = 0.60, which has higher probability.
In this way, Naive Bayes uses a similar method to predict the probability of different class based on various attributes.

Problem Analysis:

To implement the Naive Bayes Classification, we shall use a very famous Iris Flower Dataset that consists of 3 classes of flowers. In this, there are 4 independent variables namely the, sepal_length, sepal_width, petal_length and petal_width. The dependent variable is the species which we will predict using the four independent features of the flowers.

There are 3 classes of species namely *setosa*, *versicolor* and the *virginica*. This dataset was originally introduced in 1936 by Ronald Fisher. Using the various features of the flower (independent variables), we have to classify a given flower using Naive Bayes Classification model.

Step 1: Importing the Libraries

As always, the first step will always include importing the libraries which are the NumPy, Pandas and the Matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Step 2: Importing the dataset

In this step, we shall import the Iris Flower dataset which is stored in my github repository as IrisDataset.csv and save it to the variable dataset. After this, we assign the 4 independent variables to *X* and the dependent variable 'species' to *Y*. The first 5 rows of the dataset are displayed.

```
dataset = pd.read_csv('https://raw.githubusercontent.com/mk-
gurucharan/Classification/master/IrisDataset.csv')

X = dataset.iloc[:,:4].values
y = dataset['species'].values

dataset.head(5)

>>
sepal_length  sepal_width  petal_length  petal_width   species
5.1           3.5          1.4           0.2           setosa
4.9           3.0          1.4           0.2           setosa
4.7           3.2          1.3           0.2           setosa
4.6           3.1          1.5           0.2           setosa
5.0           3.6          1.4           0.2           setosa
```

Step 3: Splitting the dataset into the Training set and Test set

Once we have obtained our data set, we have to split the data into the training set and the test set. In this data set, there are 150 rows with 50 rows of each of the 3 classes. As each class is given in a continuous order, we need to randomly split the dataset. Here, we have the test_size=0.2, which means that *20%* of the dataset will be used for testing purpose as the *test set* and the remaining *80%* will be used as the *training set* for training the Naive Bayes classification model.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

Step 4: Feature Scaling

The dataset is scaled down to a smaller range using the Feature Scaling option. In this, both the X_train and X_test values are scaled down to smaller values to improve the speed of the program.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Step 5: Training the Naive Bayes Classification model on the Training Set

In this step, we introduce the class GaussianNB that is used from the sklearn.naive_bayes library. Here, we have used a Gaussian model, there are several other models such as Bernoulli, Categorical and Multinomial. Here, we assign the GaussianNB class to the variable classifier and fit the X_train and y_train values to it for training purpose.

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

Step 6: Predicting the Test set results

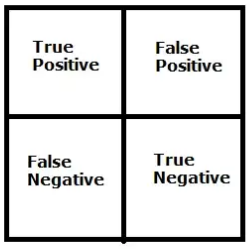
Once the model is trained, we use the the classifier.predict() to predict the values for the Test set and the values predicted are stored to the variable y_pred.

```
y_pred = classifier.predict(X_test)
y_pred
```

Step 7: Confusion Matrix and Accuracy

This is a step that is mostly used in classification techniques. In this, we see the Accuracy of the trained model and plot the confusion matrix.

The confusion matrix is a table that is used to show the number of correct and incorrect predictions on a classification problem when the real values of the Test Set are known. It is of the format



True Positive	False Positive
False Negative	True Negative

The True values are the number of correct predictions made.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

from sklearn.metrics import accuracy_score
print ("Accuracy : ", accuracy_score(y_test, y_pred))
cm

>>Accuracy :  0.9666666666666667

>>array([[14,  0,  0],
       [ 0,  7,  0],
       [ 0,  1,  8]])
```

From the above confusion matrix, we infer that, out of 30 test set data, 29 were correctly classified and only 1 was incorrectly classified. This gives us a high accuracy of 96.67%.

Step 8: Comparing the Real Values with Predicted Values

In this step, a Pandas DataFrame is created to compare the classified values of both the original Test set (*y_test*) and the predicted results (*y_pred*).

```
df = pd.DataFrame({'Real Values':y_test, 'Predicted Values':y_pred})
df

>>
Real Values   Predicted Values
setosa        setosa
setosa        setosa
virginica     virginica
versicolor    versicolor
setosa        setosa
setosa        setosa
...  ...   ...  ...  ...
virginica     versicolor
virginica     virginica
setosa        setosa
setosa        setosa
versicolor    versicolor
versicolor    versicolor
```

This step is an additional step which is not much informative as the Confusion matrix and is mainly used in regression to check the accuracy of the predicted value.
As you can see, there is one incorrect prediction that has predicted *versicolor* instead of *virginica*.

Conclusion

Thus in this story, we have successfully been able to build a *Naive Bayes Classification* Model that is able to classify a flower depending upon 4 characteristic features. This model can be implemented and tested with several other classification datasets that are available on the net.

2. Write a Python program to extract social_network_ads.csv file. Apply k-Nearest Neighbor technique to identify the users who purchased the item or not.

KNN (K Nearest Neighbors) algorithm is a supervised Machine Learning classification algorithm. It is one of the simplest and widely used classification algorithms in which a new data point is classified based on similarity in the specific group of neighboring data points. This gives a competitive result.

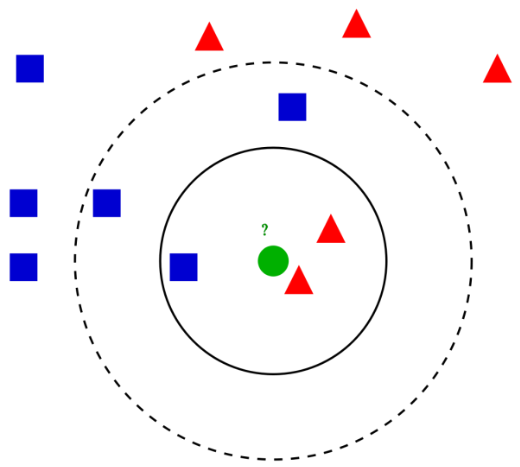
Working:

For a given data point in the set, the algorithms find the distances between this and all other K numbers of data point in the dataset close to the initial point and votes for that category that has the most frequency. Usually, Euclidean distance is taking as a measure of distance. Thus the end resultant model is just the labeled data placed in a space. This algorithm is popularly known for various applications like genetics, forecasting, etc. The algorithm is best when more features are present.

KNN reducing over fitting is a fact. On the other hand, there is a need to choose the best value for K. So now how do we choose K? Generally we use the Square root of the number of samples in the dataset as value for K. An optimal value has to be found out since lower value may lead to overfitting and higher value may require high computational complication in distance. So using an error plot may help. Another method is the elbow method. You can prefer to take root else can also follow the elbow method.

Different steps of K-NN for classifying a new data point:

Step 1: Select the value of K neighbors (say k=5)
Step 2: Find the K (5) nearest data point for our new data point based on Euclidean distance.
Step 3: Among these K data points count the data points in each category.
Step 4: Assign the new data point to the category that has the most neighbors of the new data point.



Example:

Consider an example problem for getting a clear intuition on the K -Nearest Neighbor classification. We are using the Social network ad dataset. The dataset contains the details of users in a social networking site to find whether a user buys a product by clicking the ad on the site based on their salary, age, and gender.

Importing essential libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sklearn
```

Importing of the dataset and slicing it into independent and dependent variables:

```
dataset = pd.read_csv('Social_Network_Ads.csv')
X = dataset.iloc[:, [1, 2, 3]].values
y = dataset.iloc[:, -1].values
```

Since the dataset containing character variables, need to encode it using LabelEncoder.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X[:,0] = le.fit_transform(X[:,0])
```

Split the dataset into train and test set. Providing the test size as 0.20, that means training sample contains 320 training set and test sample contains 80 tests set.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state = 0)
```

Next, feature scaling is done to the training and test set of independent variables for reducing the size to smaller values.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Build and train the K Nearest Neighbor model with the training set.

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p
= 2)
classifier.fit(X_train, y_train)
```

Three different parameters are used in the model creation. n_neighbors is setting as 5, which means 5 neighborhood points are required for classifying a given point. The distance metric used is Minkowski. Equation for the same is given below.

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$$

As per the equation, we need to select the p-value.
p = 1 , Manhattan Distance
p = 2 , Euclidean Distance
p = infinity , Cheybchev Distance

In this example, we are choosing the p value as 2. Machine Learning model is created, now we have to predict the output for the test set.

```
y_pred = classifier.predict(X_test)
```

Comparing true and predicted value:

```
y_test
>>
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
       1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,
       0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1,
       1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1], dtype=int64)
```

```
y_pred
>>
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
       1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,
       0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1,
       1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1], dtype=int64)
```

Evaluating the model using the confusion matrix and accuracy score by comparing the predicted and actual test values.

```
from sklearn.metrics import confusion_matrix,accuracy_score
cm = confusion_matrix(y_test, y_pred)
ac = accuracy_score(y_test,y_pred)
```

Confusion matrix :

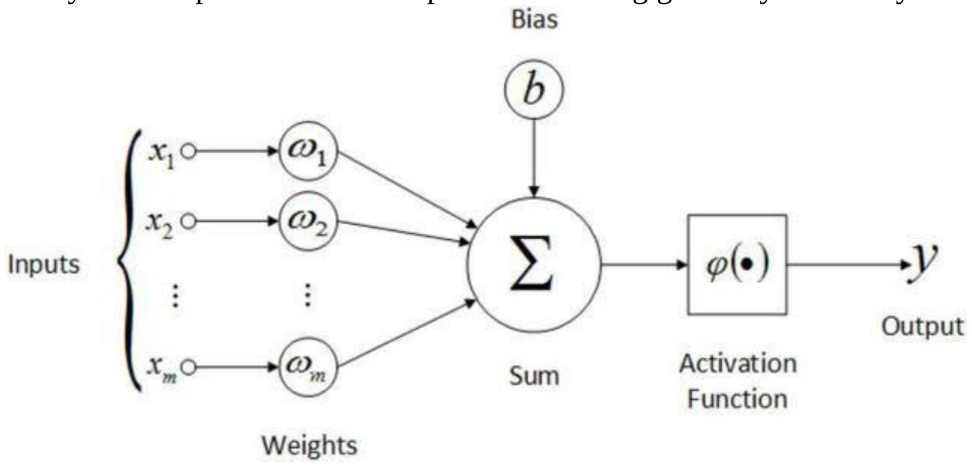
```
cm
>>

[[64  4]

 [ 3 29]]
```

```
ac
>>
0.95
```

3. Write a Python program to load whether data set and apply a perceptron learning algorithm to determine whether the rain occurs tomorrow or not.

[Perceptron Learning with iris data set]

Artificial Neural Networks (ANNs) are the new trend for all data scientists. From classical machine learning techniques, it is now shifted towards deep learning. Neural networks mimic the human brain which passes information through neurons. Perceptron is the first neural network to be created. It was designed by Frank Rosenblatt in 1957. Perceptron is a single layer neural network. This is the only neural network without any hidden layer. Perceptron is used in supervised learning generally for binary classification.



The above picture is of a perceptron where inputs are acted upon by weights and summed to bias and lastly passes through an activation function to give the final output.

Python code to implement perceptron learning algorithm:

```
import numpy as np
from sklearn.datasets import load_iris

iris = load_iris()

iris.target_names
```

OUTPUT:

```
    array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

We will merge the classes 'versicolor' and 'virginica' into one class. This means that only two classes are left. So we can differentiate with the classifier between

- Iris setosa
- not Iris setosa, or in other words either 'viriginica' od 'versicolor'

We accomplish this with the following command:

```
targets = (iris.target==0).astype(np.int8)
print(targets)
```

OUTPUT:

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
 0 0]
```

We split the data into a learn and a testset:

```
from sklearn.model_selection import train_test_split
datasets = train_test_split(iris.data,
                            targets,
                            test_size=0.2)

train_data, test_data, train_labels, test_labels = datasets
```

Now, we create a Perceptron instance and fit the training data:

```
from sklearn.linear_model import Perceptron
p = Perceptron(random_state=42,
              max_iter=10,
              tol=0.001)
p.fit(train_data, train_labels)
```

OUTPUT:

```
Perceptron(max_iter=10, random_state=42)
```

Now, we are ready for predictions and we will look at some randomly chosen random X values:

```
import random


sample = random.sample(range(len(train_data)), 10)
for i in sample:
    print(i, p.predict([train_data[i]]))
```

OUTPUT:

```
102 [0]
86 [0]
89 [0]
16 [0]
108 [0]
87 [1]
98 [1]
82 [0]
39 [0]
```

```
   118 [0]
```

```
from sklearn.metrics import classification_report

print(classification_report(p.predict(train_data), train_labels))
```

OUTPUT:

```
                 precision    recall  f1-score   support

             0       1.00      1.00      1.00        79
             1       1.00      1.00      1.00        41

      accuracy                           1.00       120
     macro avg       1.00      1.00      1.00       120
  weighted avg       1.00      1.00      1.00       120
```

```
from sklearn.metrics import classification_report

print(classification_report(p.predict(test_data), test_labels))
```

OUTPUT:

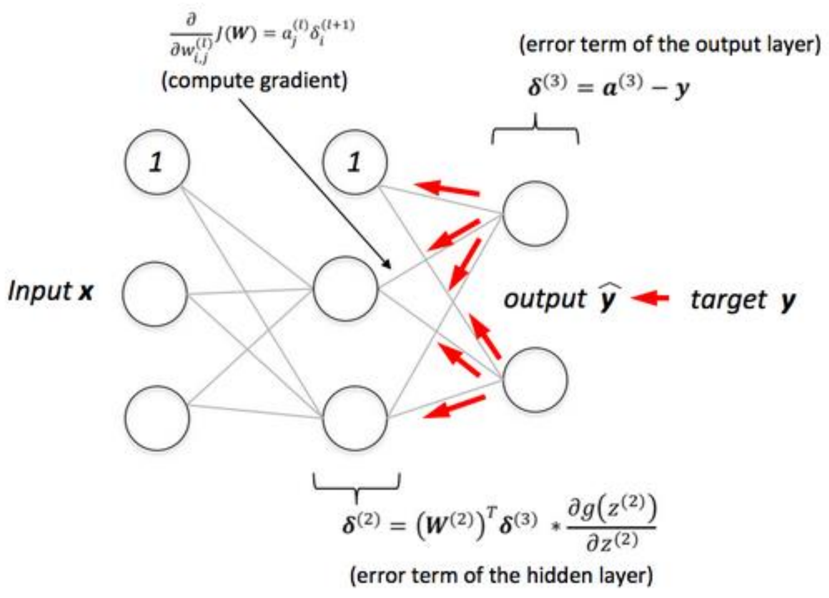
```
                 precision    recall  f1-score   support

             0       1.00      1.00      1.00        21
             1       1.00      1.00      1.00         9

      accuracy                           1.00        30
     macro avg       1.00      1.00      1.00        30
  weighted avg       1.00      1.00      1.00        30
```

4. Implement the Backpropagation algorithm in Python to classify iris data set.

Backpropagation Neural Network (BPN) is used to improve the accuracy of neural network and make them capable of self-learning. Backpropagation means “backward propagation of errors”. Here error is spread into the reverse direction in order to achieve better performance.

Backpropagation is an algorithm for supervised learning of artificial neural networks that uses the gradient descent method to minimize the cost function. It searches for optimal weights that optimize the mean-squared distance between the predicted and actual labels.

BPN was discovered by Rumelhart, Williams & Honton in 1986. The core concept of BPN is to backpropagate or spread the error from units of output layer to internal hidden layers in order to tune the weights to ensure lower error rates. It is considered a practice of fine-tuning the weights of neural networks in each iteration. Proper tuning of the weights will make a sure minimum loss and this will make a more robust, and generalizable trained neural network.



$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
1
1
Input **x**
output $\hat{y}$
target **y**
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

BPN learns in an iterative manner. In each iteration, it compares training examples with the actual target label. Target label can be a class label or continuous value. The backpropagation algorithm works in the following steps:

Initialize Network: BPN randomly initializes the weights.
Forward Propagate: After initialization, we will propagate into the forward direction. In this phase, we will compute the output and calculate the error from the target output.
Back Propagate Error: For each observation, weights are modified in order to reduce the error in a technique called the delta rule or gradient descent. It modifies weights in a "backward" direction to all the hidden layers.

Import Libraries:

```
#Import Libraries
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Load Dataset:

```
# Load dataset
data = load_iris()

# Get features and target
X=data.data
y=data.target
```

Prepare Dataset:

```
# Get dummy variable
y = pd.get_dummies(y).values
```

```
y[:3]
```

```
>>
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0]], dtype=uint8)
```

Split train and test set:

```
#Split data into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=20,
random_state=4)
```

Initialize Hyper parameters and Weights:

```
# Initialize variables
learning_rate = 0.1
iterations = 5000
N = y_train.size

# number of input features
input_size = 4

# number of hidden layers neurons
hidden_size = 2

# number of neurons at the output layer
output_size = 3

results = pd.DataFrame(columns=["mse", "accuracy"])
```

Initialize the weights for hidden and output layers with random values.

```
# Initialize weights
np.random.seed(10)

# initializing weight for the hidden layer
W1 = np.random.normal(scale=0.5, size=(input_size, hidden_size))

# initializing weight for the output layer
W2 = np.random.normal(scale=0.5, size=(hidden_size , output_size))
```

Helper Functions:

Create helper functions such as sigmoid, mean_square_error, and accuracy.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def mean_squared_error(y_pred, y_true):
    return ((y_pred - y_true)**2).sum() / (2*y_pred.size)
```

```
def accuracy(y_pred, y_true):
    acc = y_pred.argmax(axis=1) == y_true.argmax(axis=1)
    return acc.mean()
```

Backpropagation Neural Network:

In this phase, we are creating BPN in three steps feedforward propagation, error calculation and backpropagation phase. To do this, we are creating a for loop for given number of iterations that execute the three steps (feedforward propagation, error calculation and backpropagation phase) and update the weights in each iteration.

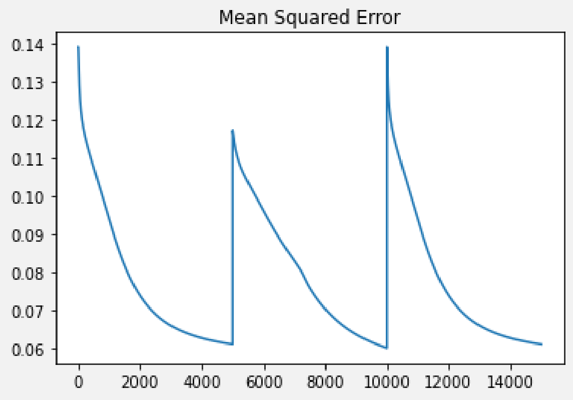
```
for itr in range(iterations):

    # feedforward propagation
    # on hidden layer
    Z1 = np.dot(x_train, W1)
    A1 = sigmoid(Z1)

    # on output layer
    Z2 = np.dot(A1, W2)
    A2 = sigmoid(Z2)


    # Calculating error
    mse = mean_squared_error(A2, y_train)
    acc = accuracy(A2, y_train)
    results=results.append({"mse":mse, "accuracy":acc},ignore_index=True )

    # backpropagation
    E1 = A2 - y_train
    dW1 = E1 * A2 * (1 - A2)
    E2 = np.dot(dW1, W2.T)
    dW2 = E2 * A1 * (1 - A1)
```

```
    # weight updates
    W2_update = np.dot(A1.T, dW1) / N
    W1_update = np.dot(x_train.T, dW2) / N

    W2 = W2 - learning_rate * W2_update
    W1 = W1 - learning_rate * W1_update
```

Plot MSE and Accuracy:

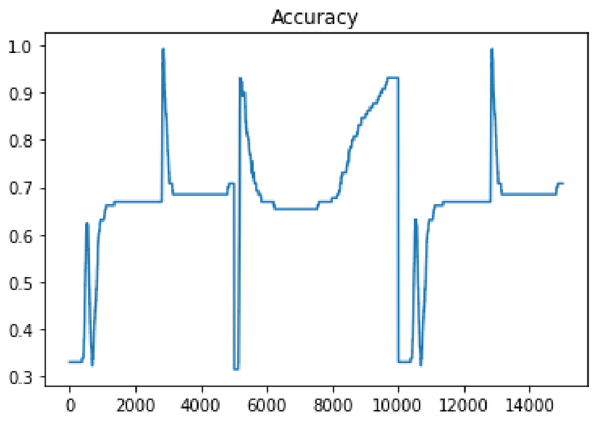
Let’s plot mean squared error in each iteration using pandas plot() function.

```
results.mse.plot(title="Mean Squared Error")
```



Lets plot accuracy in each iteration using pandas plot() function.

```
results.accuracy.plot(title="Accuracy")
```



Predict for Test Data and Evaluate the Performance:

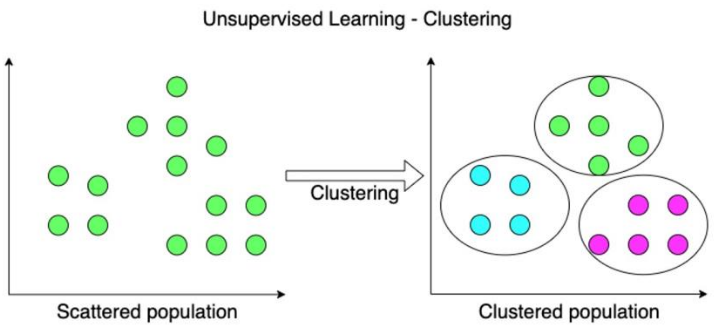
Let’s make prediction for the test data and assess the performance of Backpropagation neural network.

```
# feedforward
Z1 = np.dot(x_test, W1)
A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2)
A2 = sigmoid(Z2)

acc = accuracy(A2, y_test)
print("Accuracy: {}".format(acc))

>>
Accuracy: 0.8
```

5. Consider a Mall_Customers data set which is the data of customers who visit the mall and spend there. In the given dataset, we have Customer_Id, Gender, Age, Annual Income ($), and Spending Score (which is the calculated value of how much a customer has spent in the mall, the more the

value, the more he has spent). From this dataset, calculate some patterns using k-Means clustering method.

Clustering is a type of unsupervised machine learning in which the algorithm processes our data and divided them into "clusters". Clustering is based on the principle that items within the same cluster must be similar to each other. The data is grouped in such a way that related elements are close to each other.



K-Means Clustering:

K-Means clustering is an unsupervised machine learning algorithm that divides the given data into the given number of clusters. Here, the "K" is the given number of predefined clusters, that need to be created.
It is a centroid based algorithm in which each cluster is associated with a centroid. The main idea is to reduce the distance between the data points and their respective cluster centroid.
The algorithm takes raw unlabeled data as an input and divides the dataset into clusters and the process is repeated until the best clusters are found.

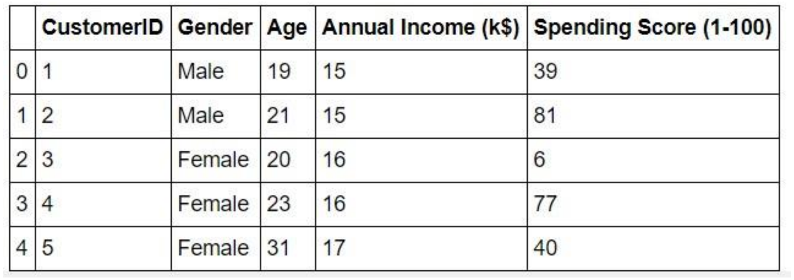
K-Means is very easy and simple to implement. It is highly scalable, can be applied to both small and large datasets. There is, however, a problem with choosing the number of clusters or K. Also, with the increase in dimensions, stability decreases. But, overall K Means is a simple and robust algorithm that makes clustering very easy.

The data set includes the following features:
1. Customer ID
2. Customer Gender
3. Customer Age
4. Annual Income of the customer (in Thousand Dollars)
5. Spending score of the customer (based on customer behavior and spending nature)

Implementation:

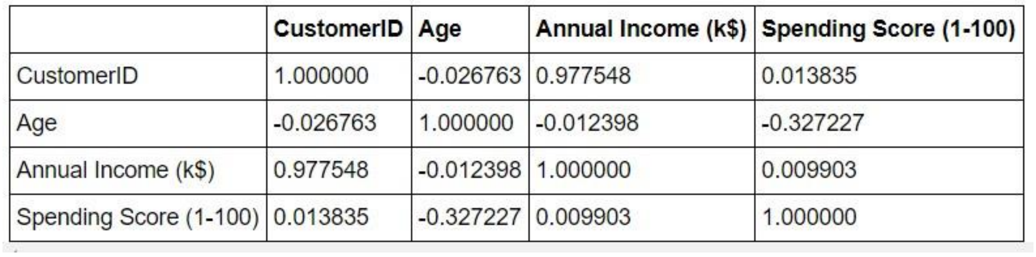
```
#Importing the necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from mpl_toolkits.mplot3d import Axes3D
```

```
%matplotlib inline
#Reading the excel file
data=pd.read_excel("Mall_Customers.xlsx")
data.head()
```



	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

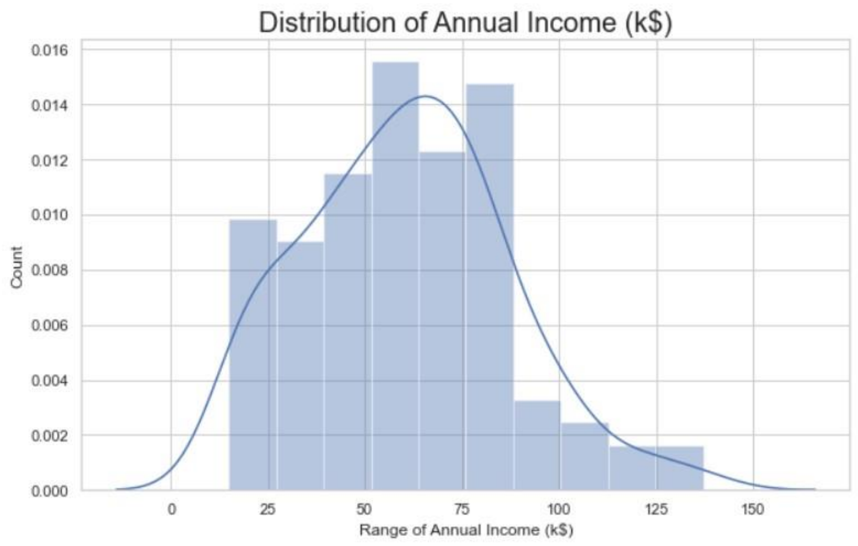
```
data.corr()
```



	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
CustomerID	1.000000	-0.026763	0.977548	0.013835
Age	-0.026763	1.000000	-0.012398	-0.327227
Annual Income (k$)	0.977548	-0.012398	1.000000	0.009903
Spending Score (1-100)	0.013835	-0.327227	0.009903	1.000000

Annual Income Distribution:

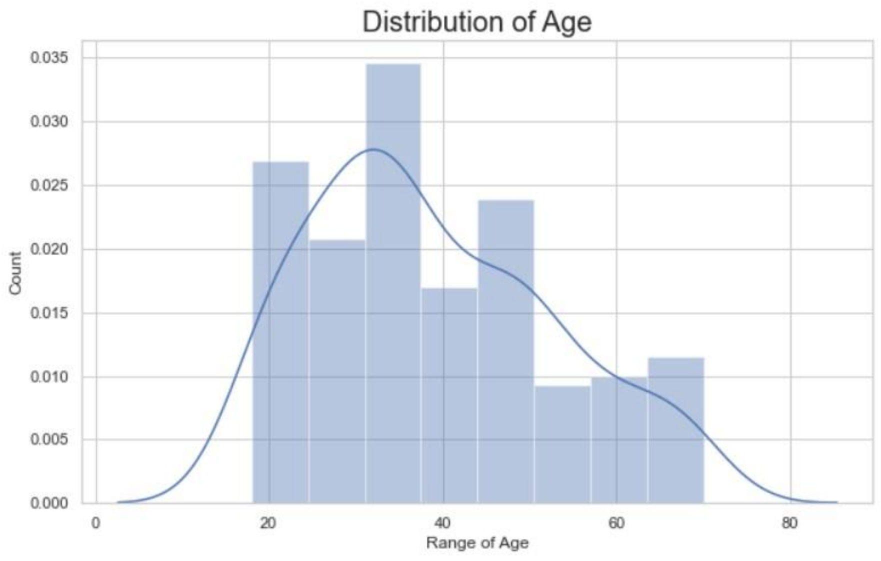
```
#Distribution of Annnual Income
plt.figure(figsize=(10, 6))
sns.set(style = 'whitegrid')
sns.distplot(data['Annual Income (k$)'])
plt.title('Distribution of Annual Income (k$)', fontsize = 20)
plt.xlabel('Range of Annual Income (k$)')
plt.ylabel('Count')
```



Most of the annual income falls between 50K to 85K.

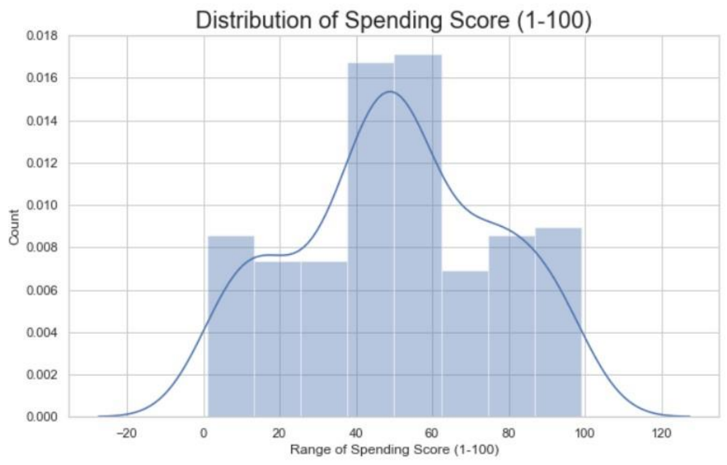
Age Distribution:

```
#Distribution of age
plt.figure(figsize=(10, 6))
sns.set(style = 'whitegrid')
sns.distplot(data['Age'])
plt.title('Distribution of Age', fontsize = 20)
plt.xlabel('Range of Age')
plt.ylabel('Count')
```



There are customers of a wide variety of ages.

Spending Score Distribution:

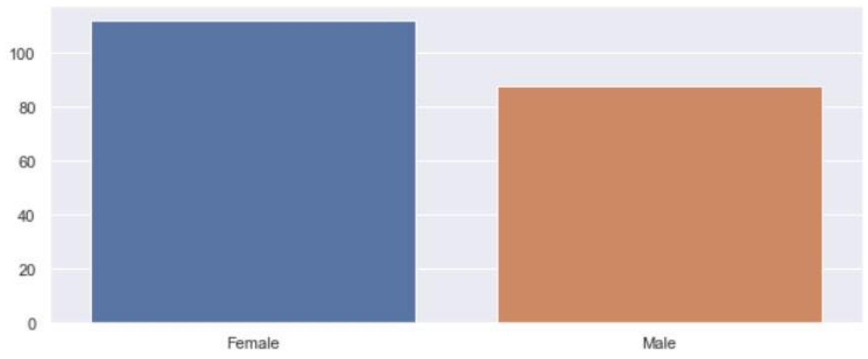


```
#Distribution of spending score
plt.figure(figsize=(10, 6))
sns.set(style = 'whitegrid')
sns.distplot(data['Spending Score (1-100)'])
plt.title('Distribution of Spending Score (1-100)', fontsize = 20)
plt.xlabel('Range of Spending Score (1-100)')
plt.ylabel('Count')
```

The maximum spending score is in the range of 40 to 60.

Gender Analysis:

```
genders = data.Gender.value_counts()
sns.set_style("darkgrid")
plt.figure(figsize=(10,4))
sns.barplot(x=genders.index, y=genders.values)
plt.show()
```

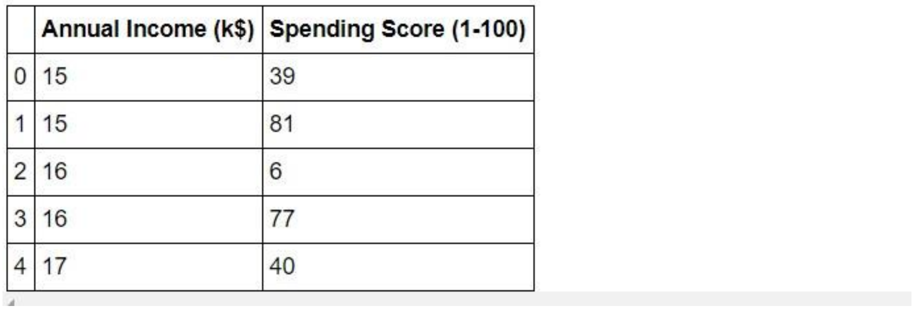


More female customers than male.

Clustering based on 2 features:

First, we work with two features only, annual income and spending score.

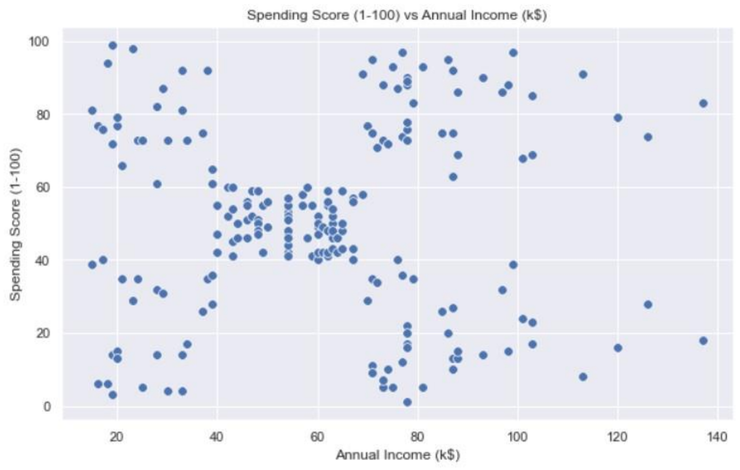
```
#We take just the Annual Income and Spending score
df1=data[["CustomerID","Gender","Age","Annual Income (k$)","Spending Score (1-100)"]]
X=df1[["Annual Income (k$)","Spending Score (1-100)"]]
#The input data
X.head()
```



	Annual Income (k$)	Spending Score (1-100)
0	15	39
1	15	81
2	16	6
3	16	77
4	17	40

```
#Scatterplot of the input data
plt.figure(figsize=(10,6))
sns.scatterplot(x = 'Annual Income (k$)',y = 'Spending Score (1-100)',  data = X  ,s = 60 )
```

```
plt.xlabel('Annual Income (k$)')

plt.ylabel('Spending Score (1-100)')

plt.title('Spending Score (1-100) vs Annual Income (k$)')

plt.show()
```



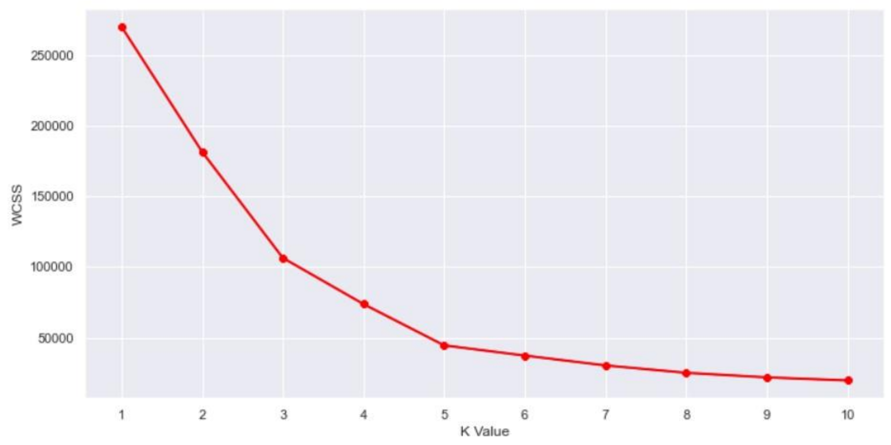
The data does seem to hold some patterns.

```
#Importing KMeans from sklearn

from sklearn.cluster import KMeans
```

Now we calculate the Within Cluster Sum of Squared Errors (WSS) for different values of k. Next, we choose the k for which WSS first starts to diminish. This value of K gives us the best number of clusters to make from the raw data.

```
wcss=[]

for i in range(1,11):

    km=KMeans(n_clusters=i)

    km.fit(X)

    wcss.append(km.inertia_)


#The elbow curve

plt.figure(figsize=(12,6))

plt.plot(range(1,11),wcss)

plt.plot(range(1,11),wcss, linewidth=2, color="red", marker ="8")

plt.xlabel("K Value")

plt.xticks(np.arange(1,11,1))

plt.ylabel("WCSS")

plt.show()
```

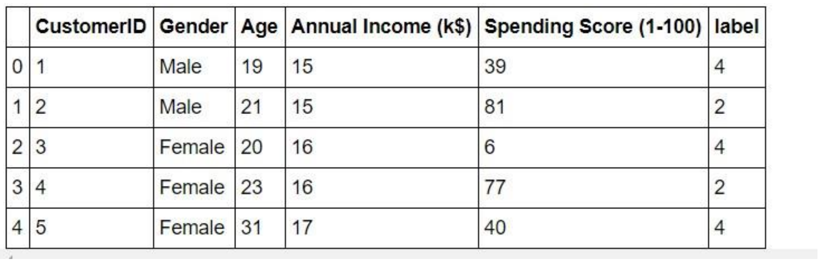
The plot:



This is known as the elbow graph, the x-axis being the number of clusters, the number of clusters is taken at the elbow joint point. This point is the point where making clusters is most relevant as here the value of WCSS suddenly stops decreasing. Here in the graph, after 5 the drop is minimal, so we take 5 to be the number of clusters.

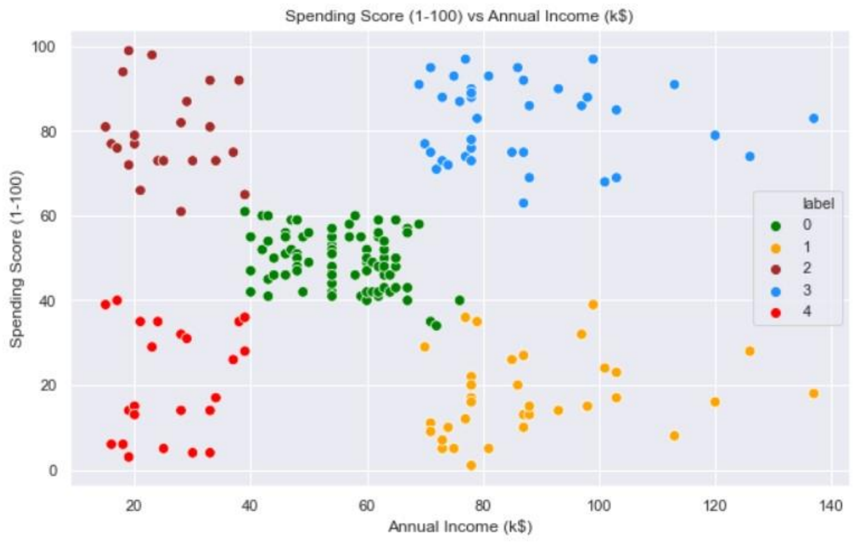
```
#Taking 5 clusters

km1=KMeans(n_clusters=5)

#Fitting the input data

km1.fit(X)

#predicting the labels of the input data

y=km1.predict(X)

#adding the labels to a column named label

df1["label"] = y

#The new dataframe with the clustering done

df1.head()
```

The labels added to the data.



	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	label
0	1	Male	19	15	39	4
1	2	Male	21	15	81	2
2	3	Female	20	16	6	4
3	4	Female	23	16	77	2
4	5	Female	31	17	40	4

```
plt.figure(figsize=(10,6))

sns.scatterplot(x = 'Annual Income (k$)',y = 'Spending Score (1-100)',hue="label",

                 palette=['green','orange','brown','dodgerblue','red'], legend='full',data =
df1  ,s = 60 )

plt.xlabel('Annual Income (k$)')

plt.ylabel('Spending Score (1-100)')

plt.title('Spending Score (1-100) vs Annual Income (k$)')

plt.show()
```



We can clearly see that 5 different clusters have been formed from the data. The red cluster is the customers with the least income and least spending score, similarly, the blue cluster is the customers with the most income and most spending score.

k-Means Clustering on the basis of 3D data: (optional)

6. Consider a dataset that has two variables: salary (dependent variable) and experience (Independent variable). Build a simple Linear-Regression model in Python to do the following:
i) Find out if there is any correlation between these two variables.
ii) Find the best fit line for the dataset.
iii) Show how the dependent variable is changing by changing the independent variable.

Step-1: Data Pre-processing:

Importing the three important libraries for loading the dataset, plotting the graphs, and creating the Simple Linear Regression model.

```
import numpy as nm

import matplotlib.pyplot as mtp

import pandas as pd

data_set= pd.read_csv('Salary_Data.csv')

data_set
```

The dataset has two variables: Salary and Experience. We need to extract the dependent and independent variables from the given dataset. The independent variable is years of experience, and the dependent variable is salary. Below is code for it:

```
x= data_set.iloc[:, :-1].values

y= data_set.iloc[:, 1].values

x

y
```

Next, we will split both variables into the test set and training set. We have 30 observations, so we will take 20 observations for the training set and 10 observations for the test set. We are splitting our dataset so that we can train our model using a training dataset and then test the model using a test dataset. The code for this is given below:

```
# Splitting the dataset into training and test set.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 1/3, random_state=0)
x_train
x_test
y_train
y_test
```

Step-2: Fitting the Simple Linear Regression to the Training Set:

```
#Fitting the Simple Linear Regression model to the training dataset
from sklearn.linear_model import LinearRegression
regressor= LinearRegression()
regressor.fit(x_train, y_train)
```

In the above code, we have used a **fit()** method to fit our Simple Linear Regression object to the training set. In the fit() function, we have passed the x_train and y_train, which is our training dataset for the dependent and an independent variable. We have fitted our regressor object to the training set so that the model can easily learn the correlations between the predictor and target variables. After executing the above lines of code, we will get the below output.

Output:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Step: 3. Prediction of test set result:

Now, the model is ready to predict the output for the new observations. In this step, we will provide the test dataset (new observations) to the model to check whether it can predict the correct output or not.

We will create a prediction vector **y_pred**, and **x_pred**, which will contain predictions of test dataset, and prediction of training set respectively.

```
#Prediction of Test and Training set result
y_pred= regressor.predict(x_test)
x_pred= regressor.predict(x_train)
```

On executing the above lines of code, two variables named y_pred and x_pred will generate in the variable explorer options that contain salary predictions for the training set and test set.

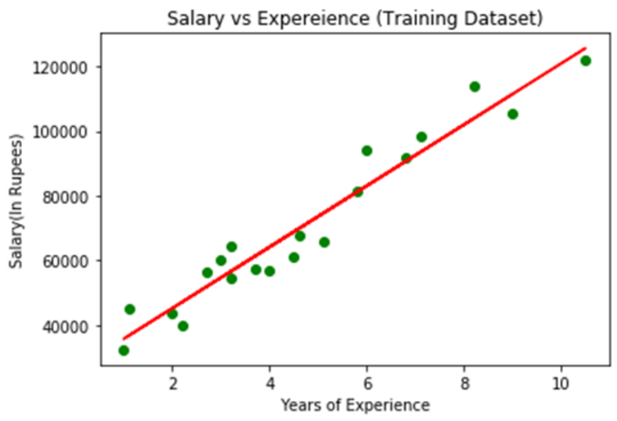
```
y_pred
x_pred
```

We can check the result by comparing values: y_pred and y_test. We can also analyse how good our model is performing.

Step: 4. Visualizing the Training set results:

```
mtp.scatter(x_train, y_train, color="green")
mtp.plot(x_train, x_pred, color="red")
mtp.title("Salary vs Experience (Training Dataset)")

mtp.xlabel("Years of Experience")

mtp.ylabel("Salary(In Rupees)")

mtp.show()
```

Output:



In the above plot, we can see the real values observations in green dots and predicted values are covered by the red regression line. The regression line shows a correlation between the dependent and independent variable.

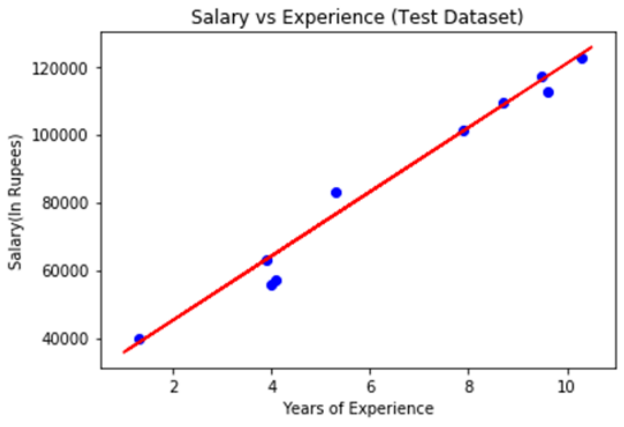
The good fit of the line can be observed by calculating the difference between actual values and predicted values. But as we can see in the above plot, most of the observations are close to the regression line, hence our model is good for the training set.

Step: 5. Visualizing the Test set results:

In the previous step, we have visualized the performance of our model on the training set. Now, we will do the same for the Test set. The complete code will remain the same as the above code, except in this, we will use x_test, and y_test instead of x_train and y_train.

```
#visualizing the Test set results

mtp.scatter(x_test, y_test, color="blue")

mtp.plot(x_train, x_pred, color="red")

mtp.title("Salary vs Experience (Test Dataset)")

mtp.xlabel("Years of Experience")

mtp.ylabel("Salary(In Rupees)")

mtp.show()
```

Output:



In the above plot, there are observations given by the blue color, and prediction is given by the red regression line. As we can see, most of the observations are close to the regression line, hence we can say our Simple Linear Regression is a good model and able to make good predictions.

7. Implement Support Vector Machine algorithm in Python for any suitable data set available.

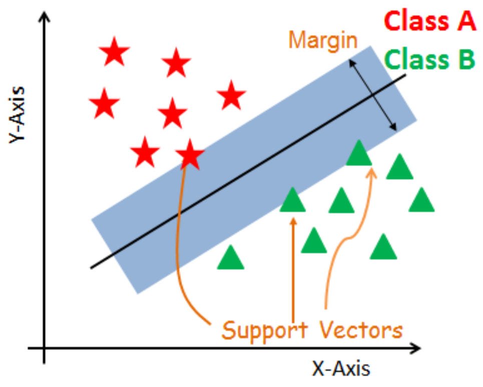
Support Vector Machines with Scikit-learn:

SVM offers very high accuracy compared to other classifiers such as logistic regression, and decision trees. It is known for its kernel trick to handle nonlinear input spaces. It is used in a variety of applications such as face detection, intrusion detection, classification of emails, news articles and web pages, classification of genes, and handwriting recognition.

SVM is an exciting algorithm and the concepts are relatively simple. The classifier separates data points using a hyperplane with the largest amount of margin. That's why an SVM classifier is also known as a discriminative classifier. SVM finds an optimal hyperplane which helps in classifying new data points.

Support Vector Machines

Generally, Support Vector Machines is considered to be a classification approach, it but can be employed in both types of classification and regression problems. It can easily handle multiple continuous and categorical variables. SVM constructs a hyperplane in multidimensional space to separate different classes. SVM generates optimal hyperplane in an iterative manner, which is used to minimize an error. The core idea of SVM is to find a maximum marginal hyperplane(MMH) that best divides the dataset into classes.



Support Vectors

Support vectors are the data points, which are closest to the hyperplane. These points will define the separating line better by calculating margins. These points are more relevant to the construction of the classifier.

Hyperplane

A hyperplane is a decision plane which separates between a set of objects having different class memberships.

Margin

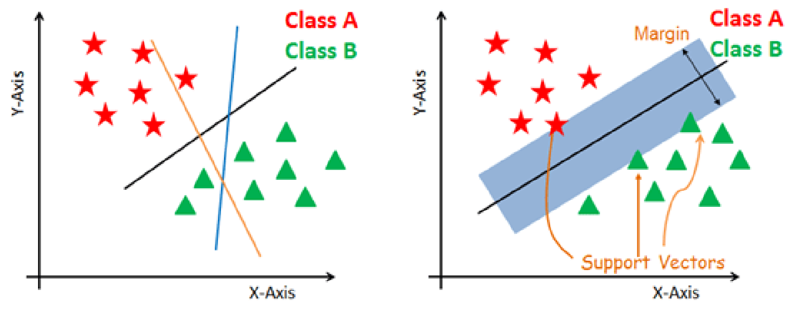
A margin is a gap between the two lines on the closest class points. This is calculated as the perpendicular distance from the line to support vectors or closest points. If the margin is larger in between the classes, then it is considered a good margin, a smaller margin is a bad margin.

How does SVM work?

The main objective is to segregate the given dataset in the best possible way. The distance between the either nearest points is known as the margin. The objective is to select a hyperplane with the maximum possible margin between support vectors in the given dataset. SVM searches for the maximum marginal hyperplane in the following steps:

1. Generate hyperplanes which segregates the classes in the best way. Left-hand side figure showing three hyperplanes black, blue and orange. Here, the blue and orange have higher classification error, but the black is separating the two classes correctly.

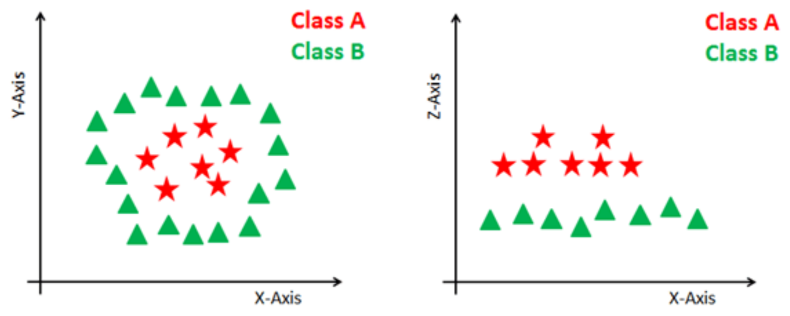
2. Select the right hyperplane with the maximum segregation from the either nearest data points as shown in the right-hand side figure.



Dealing with non-linear and inseparable planes

Some problems can’t be solved using linear hyperplane, as shown in the figure below (left-hand side).

In such situation, SVM uses a kernel trick to transform the input space to a higher dimensional space as shown on the right. The data points are plotted on the x-axis and z-axis (Z is the squared sum of both x and y: z=x^2=y^2). Now you can easily segregate these points using linear separation.



SVM Kernels

The SVM algorithm is implemented in practice using a kernel. A kernel transforms an input data space into the required form. SVM uses a technique called the kernel trick. Here, the kernel takes a low-dimensional input space and transforms it into a higher dimensional space. In other words, you can say that it converts nonseparable problem to separable problems by adding more dimension to it. It is most useful in non-linear separation problem. Kernel trick helps you to build a more accurate classifier.

Linear Kernel A linear kernel can be used as normal dot product any two given observations. The product between two vectors is the sum of the multiplication of each pair of input values.

K(x, xi) = sum(x * xi)

Polynomial Kernel A polynomial kernel is a more generalized form of the linear kernel. The polynomial kernel can distinguish curved or nonlinear input space.

K(x,xi) = 1 + sum(x * xi)^d

Where d is the degree of the polynomial. d=1 is similar to the linear transformation. The degree needs to be manually specified in the learning algorithm.

Radial Basis Function Kernel The Radial basis function kernel is a popular kernel function commonly used in support vector machine classification. RBF can map an input space in infinite dimensional space.

K(x,xi) = exp(-gamma * sum((x – xi^2))

Here gamma is a parameter, which ranges from 0 to 1. A higher value of gamma will perfectly fit the training dataset, which causes over-fitting. Gamma=0.1 is considered to be a good default value. The value of gamma needs to be manually specified in the learning algorithm.

Classifier Building in Scikit-learn

In the model the building part, we can use the cancer dataset, which is a very famous multi-class classification problem. This dataset is computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

The dataset comprises 30 features (mean radius, mean texture, mean perimeter, mean area, mean smoothness, mean compactness, mean concavity, mean concave points, mean symmetry, mean fractal dimension, radius error, texture error, perimeter error, area error, smoothness error, compactness error, concavity error, concave points error, symmetry error, fractal dimension error, worst radius, worst texture, worst perimeter, worst area, worst smoothness, worst compactness, worst concavity, worst concave points, worst symmetry, and worst fractal dimension) and a target (type of cancer).

This data has two types of cancer classes: malignant (harmful) and benign (not harmful). Here, you can build a model to classify the type of cancer. The dataset is available in the scikit-learn library or you can also download it from the UCI Machine Learning Library.

Loading Data

```
#Import scikit-learn dataset library

from sklearn import datasets

#Load dataset

cancer = datasets.load_breast_cancer()
```

Exploring Data

After you have loaded the dataset, you might want to know a little bit more about it. You can check feature and target names.

```
# print the names of the 13 features
print("Features: ", cancer.feature_names)
# print the label type of cancer('malignant' 'benign')
print("Labels: ", cancer.target_names)
Features:  ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
Labels:  ['malignant' 'benign']
```

We can check the shape of the dataset using shape.

```
# print data(feature)shape
cancer.data.shape
(569, 30)
```

Let's check top 5 records of the feature set.

```
# print the cancer data features (top 5 records)
print(cancer.data[0:5])
[[1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-01 3.001e-01
  1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
  6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
  1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
  4.601e-01 1.189e-01]
 [2.057e+01 1.777e+01 1.329e+02 1.326e+03 8.474e-02 7.864e-02 8.690e-02
  7.017e-02 1.812e-01 5.667e-02 5.435e-01 7.339e-01 3.398e+00 7.408e+01
  5.225e-03 1.308e-02 1.860e-02 1.340e-02 1.389e-02 3.532e-03 2.499e+01
  2.341e+01 1.588e+02 1.956e+03 1.238e-01 1.866e-01 2.416e-01 1.860e-01
  2.750e-01 8.902e-02]
 [1.969e+01 2.125e+01 1.300e+02 1.203e+03 1.096e-01 1.599e-01 1.974e-01
  1.279e-01 2.069e-01 5.999e-02 7.456e-01 7.869e-01 4.585e+00 9.403e+01
  6.150e-03 4.006e-02 3.832e-02 2.058e-02 2.250e-02 4.571e-03 2.357e+01
  2.553e+01 1.525e+02 1.709e+03 1.444e-01 4.245e-01 4.504e-01 2.430e-01
  3.613e-01 8.758e-02]
 [1.142e+01 2.038e+01 7.758e+01 3.861e+02 1.425e-01 2.839e-01 2.414e-01
  1.052e-01 2.597e-01 9.744e-02 4.956e-01 1.156e+00 3.445e+00 2.723e+01
  9.110e-03 7.458e-02 5.661e-02 1.867e-02 5.963e-02 9.208e-03 1.491e+01
  2.650e+01 9.887e+01 5.677e+02 2.098e-01 8.663e-01 6.869e-01 2.575e-01
  6.638e-01 1.730e-01]
 [2.029e+01 1.434e+01 1.351e+02 1.297e+03 1.003e-01 1.328e-01 1.980e-01
  1.043e-01 1.809e-01 5.883e-02 7.572e-01 7.813e-01 5.438e+00 9.444e+01
  1.149e-02 2.461e-02 5.688e-02 1.885e-02 1.756e-02 5.115e-03 2.254e+01
  1.667e+01 1.522e+02 1.575e+03 1.374e-01 2.050e-01 4.000e-01 1.625e-01
  2.364e-01 7.678e-02]]
```

Let's take a look at the target set.

```
# print the cancer labels (0:malignant, 1:benign)
print(cancer.target)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 0 1 0 0
 1 0 1 0 0 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 1
 1 1 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1
 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 1 0
 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0 1 1 0 0 1 1
 1 0 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1
 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 0 1 1
 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0
 0 1 0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1
 1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0 1 1 1 1 1 0 1 1
 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1
 1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 1 0 0
 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 0 0 0 0 0 0 1]
```

Splitting Data

To understand model performance, dividing the dataset into a training set and a test set is a good strategy. Split the dataset by using the function train_test_split(). you need to pass 3 parameters features, target, and test_set size. Additionally, you can use random_state to select records randomly.

```
# Import train_test_split function
from sklearn.model_selection import train_test_split
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, test_size=0.3,random_state=109) # 70% training and 30% test
```

Generating Model

Let's build support vector machine model. First, import the SVM module and create support vector classifier object by passing argument kernel as the linear kernel in SVC() function.

Then, fit your model on train set using fit() and perform prediction on the test set using predict().

```
#Import svm model
from sklearn import svm
#Create a svm Classifier
clf = svm.SVC(kernel='linear') # Linear Kernel
#Train the model using the training sets
clf.fit(X_train, y_train)
#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

Evaluating the Model

Let's estimate how accurately the classifier or model can predict the breast cancer of patients. Accuracy can be computed by comparing actual test set values and predicted values.

```
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
# Model Accuracy: how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9649122807017544
```

We got a classification rate of 96.49%, considered as very good accuracy. For further evaluation, we can also check precision and recall of model.

```
# Model Precision: what percentage of positive tuples are labeled as such?
print("Precision:",metrics.precision_score(y_test, y_pred))
# Model Recall: what percentage of positive tuples are labelled as such?
print("Recall:",metrics.recall_score(y_test, y_pred))
```

```
Precision: 0.9811320754716981
```

Recall: 0.9629629629629629

We got a precision of 98% and recall of 96%, which are considered as very good values.

Tuning Hyperparameters:

Kernel: The main function of the kernel is to transform the given dataset input data into the required form. There are various types of functions such as linear, polynomial, and radial basis function (RBF). Polynomial and RBF are useful for non-linear hyperplane. Polynomial and RBF kernels compute the separation line in the higher dimension. In some of the applications, it is suggested to use a more complex kernel to separate the classes that are curved or nonlinear. This transformation can lead to more accurate classifiers.

Regularization: Regularization parameter in python's Scikit-learn C parameter used to maintain regularization. Here C is the penalty parameter, which represents misclassification or error term. The misclassification or error term tells the SVM optimization how much error is bearable. This is how you can control the trade-off between decision boundary and misclassification term. A smaller value of C creates a small-margin hyperplane and a larger value of C creates a larger-margin hyperplane.

Gamma: A lower value of Gamma will loosely fit the training dataset, whereas a higher value of gamma will exactly fit the training dataset, which causes over-fitting. In other words, you can say a low value of gamma considers only nearby points in calculating the separation line, while the a value of gamma considers all the data points in the calculation of the separation line.

Advantages:

SVM Classifiers offer good accuracy and perform faster prediction compared to Naïve Bayes algorithm. They also use less memory because they use a subset of training points in the decision phase. SVM works well with a clear margin of separation and with high dimensional space.

Disadvantages:

SVM is not suitable for large datasets because of its high training time and it also takes more time in training compared to Naïve Bayes. It works poorly with overlapping classes and is also sensitive to the type of kernel used.

8. Consider the User Database which contains information about UserID, Gender, Age, EstimatedSalary, and Purchased. Apply Logistic Regression in Python to predict whether a user will purchase the company’s newly launched product or not.

Logistic regression is basically a supervised classification algorithm. In a classification problem, the target variable (or output), y, can take only discrete values for a given set of features (or inputs), X. Contrary to popular belief, logistic regression is a regression model. The model builds a regression model to predict the probability that a given data entry belongs to the category numbered as “1”. Logistic regression models the data using the sigmoid function.

Logistic regression becomes a classification technique only when a decision threshold is brought into the picture. The setting of the threshold value is a very important aspect of Logistic regression and is dependent on the classification problem itself.

Import Libraries:

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
```

Read and explore the data;

```
dataset = pd.read_csv("User_Data.csv")
```

Now, to predict whether a user will purchase the product or not, one needs to find out the relationship between Age and Estimated Salary. Here User ID and Gender are not important factors for finding out this.

```
# input

x = dataset.iloc[:, [2, 3]].values

# output

y = dataset.iloc[:, 4].values
```

Splitting The Dataset: Train and Test dataset

Splitting the dataset to train and test. 75% of data is used for training the model and 25% of it is used to test the performance of our model.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size = 0.25, random_state = 0)
```

Now, it is very important to perform feature scaling here because Age and Estimated Salary values lie in different ranges. If we don't scale the features then the Estimated Salary feature will dominate the Age feature when the model finds the nearest neighbor to a data point in the data space.

```
from sklearn.preprocessing import StandardScaler

sc_x = StandardScaler()

xtrain = sc_x.fit_transform(xtrain)

xtest = sc_x.transform(xtest)

print (xtrain[0:10, :])
```

Output:

```
[[ 0.58164944 -0.88670699]

 [-0.60673761  1.46173768]

 [-0.01254409 -0.5677824 ]

 [-0.60673761  1.89663484]

 [ 1.37390747 -1.40858358]

 [ 1.47293972  0.99784738]

 [ 0.08648817 -0.79972756]

 [-0.01254409 -0.24885782]

 [-0.21060859 -0.5677824 ]

 [-0.21060859 -0.19087153]]
```

Here once see that Age and Estimated salary features values are scaled and now there in the -1 to 1. Hence, each feature will contribute equally to decision making i.e. finalizing the hypothesis.

Finally, we are training our Logistic Regression model.

Train the Model

```
from sklearn.linear_model import LogisticRegression

classifier = LogisticRegression(random_state = 0)

classifier.fit(xtrain, ytrain)
```

After training the model, it is time to use it to do predictions on testing data.

```
y_pred = classifier.predict(xtest)
```

Evaluation Metrics

Metrics are used to check the model performance on predicted values and actual values.

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(ytest, y_pred)

print ("Confusion Matrix : \n", cm)
```

Output:

```
Confusion Matrix :

 [[65  3]

 [ 8 24]]
```

```
Out of 100 :
True Positive + True Negative = 65 + 24
False Positive + False Negative = 3 + 8
Performance measure – Accuracy
```

```
from sklearn.metrics import accuracy_score

print ("Accuracy : ", accuracy_score(ytest, y_pred))
```

Output:

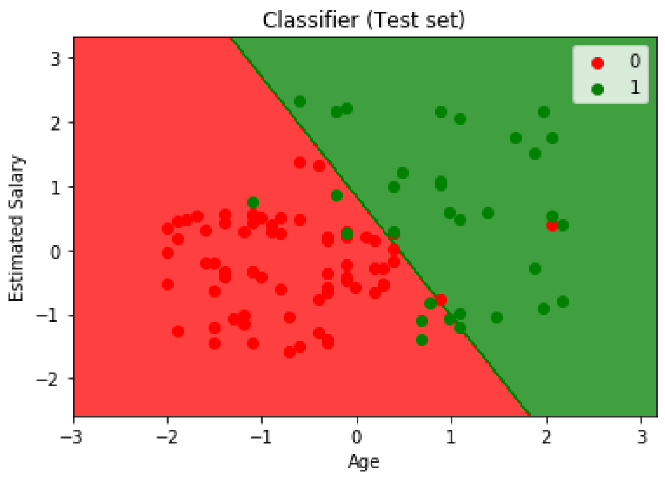
```
Accuracy :  0.89
```

Visualizing the performance of our model.

```
from matplotlib.colors import ListedColormap

X_set, y_set = xtest, ytest

X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,

                               stop = X_set[:, 0].max() + 1, step = 0.01),

                     np.arange(start = X_set[:, 1].min() - 1,

                               stop = X_set[:, 1].max() + 1, step = 0.01))
```

```
plt.contourf(X1, X2, classifier.predict(
             np.array([X1.ravel(), X2.ravel()]).T).reshape(
             X1.shape), alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)

plt.title('Classifier (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

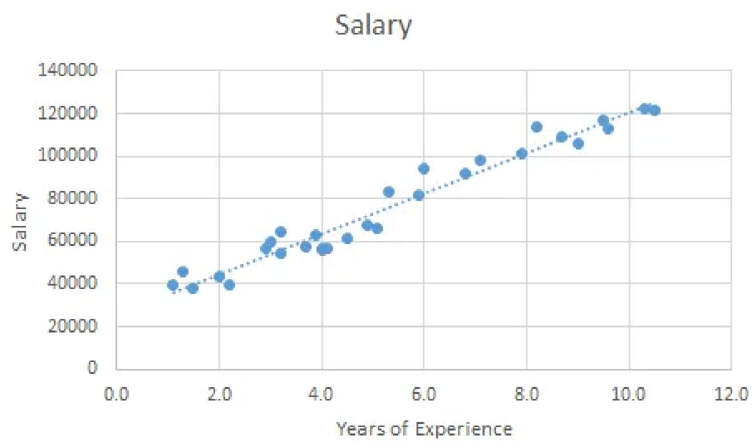
Output:



Analyzing the performance measures – accuracy and confusion matrix and the graph, we can clearly say that our model is performing really well.

9. Implement Polynomial Regression model in Python for any suitable data set available.

Linear Regression is applied for the data set that their values are **linear** as below example:

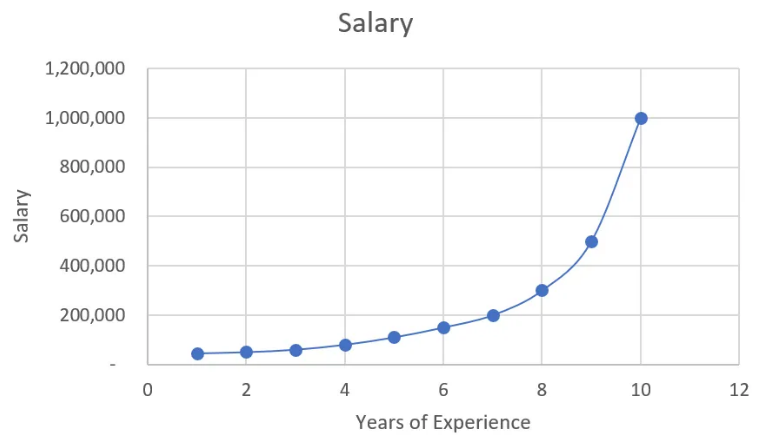


And real life is not that simple, especially when you observe from many different companies in different industries. Salary of 1 YE teacher is different from 1 YE engineer; even 1 YE civil engineer is different from mechanical engineer; and if you compare 2 mechanical engineers from 2 different companies, their salary mostly different as well. So how can we predict the salary of a candidate?

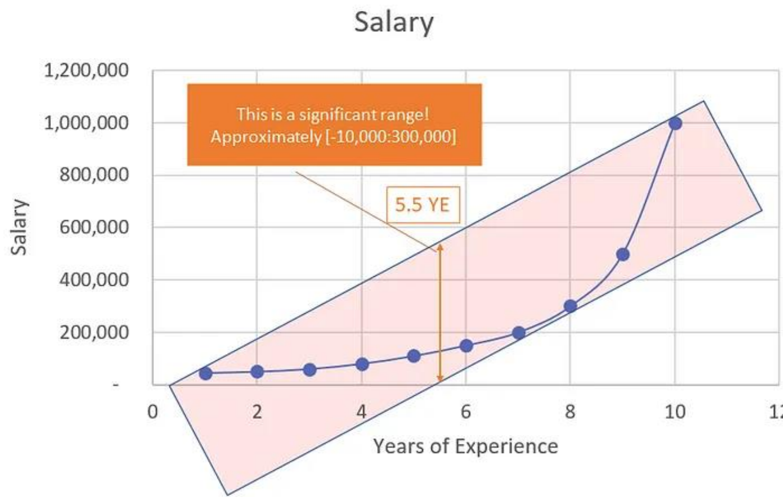
We will use another data set to represent the Polynomial shape.

Years of Experience	Salary
1	45,000
2	50,000
3	60,000
4	80,000
5	110,000
6	150,000
7	200,000
8	300,000
9	500,000
10	1,000,000

To get an overview of the increment of salary, let’s visualize the data set into a chart:



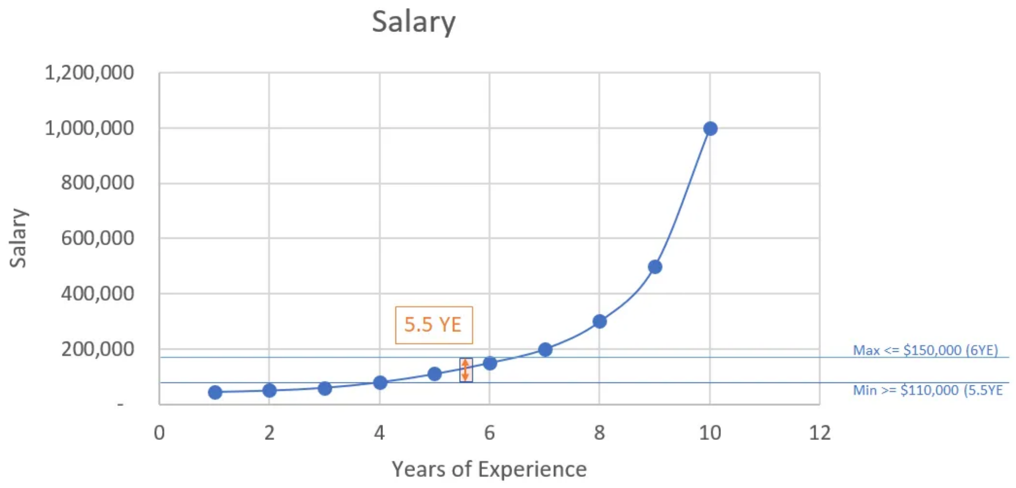
Let’s think about one candidate. He has 5 YE. What if we use the Linear Regression in this example?



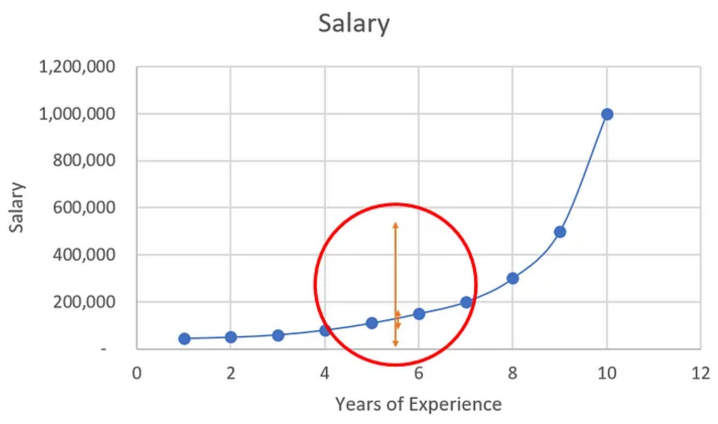
According to the picture above, the salary range of our candidate could be approximately *from minus $10,000 to $300,000*. Why? Look, the salary observations in this scenario are not linear. **They are in a curved shape!** That's why applying Linear Regression in this scenario is not giving the right value. It's time for **Polynomial Regression**.

Why Polynomial Regression?
Because it's much much more accurate!

We are already know the salary of 5 YE is $110,000 and 6 YE is $150,000. It means the salary of 5.5 YE should be between them! And this is how the best value should be:



Let's compare the gaps between using Linear and Polynomial.



It’s almost **7.75 times** more accurate than using Linear Regression!

We can calculate by using the Mean value. Because 5.5 is the average of 5 and 6, so the salary could be calculated as:

(150,000 + 110,000) / 2 = **$130,000**

But it’s not the highest accuracy rate and too manual! Let’s apply the Machine Learning for more accuracy and flexible calculation.

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd


# Importing the dataset

dataset = pd.read_csv('./Sample Data/PART 2. REGRESSION - Polynomial Regression - Polynomial_Regression/Polynomial_Regression/Position_Salaries.csv')

X = dataset.iloc[:, 1:2].values

y = dataset.iloc[:, 2].values


# Splitting the dataset into the Training set and Test set

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)


"""

# Scaling

from sklearn.preprocessing import StandardScaler

sc_X = StandardScaler()

X_train = sc_X.fit_transform(X_train)
```

```
X_test = sc_X.transform(X_test)
"""

# Fitting Linear Regression to the dataset - optional
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)

# Visualizing the Linear Regression results - optional

def viz_linear():
    plt.scatter(X, y, color='red')
    plt.plot(X, lin_reg.predict(X), color='blue')
    plt.title('Truth or Bluff (Linear Regression)')
    plt.xlabel('Position level')
    plt.ylabel('Salary')
    plt.show()
    return
viz_linear()

# Fitting Polynomial Regression to the dataset
from sklearn.preprocessing import PolynomialFeatures
poly_reg = PolynomialFeatures(degree=4)
X_poly = poly_reg.fit_transform(X)
pol_reg = LinearRegression()
pol_reg.fit(X_poly, y)

# Visualizing the Polymonial Regression results
def viz_polymonial():
    plt.scatter(X, y, color='red')
    plt.plot(X, pol_reg.predict(poly_reg.fit_transform(X)), color='blue')
    plt.title('Truth or Bluff (Linear Regression)')
    plt.xlabel('Position level')
    plt.ylabel('Salary')
    plt.show()
    return
viz_polymonial()
```

```
# Additional feature
# Making the plot line (Blue one) more smooth
def viz_polymonial_smooth():
    X_grid = np.arange(min(X), max(X), 0.1)
    X_grid = X_grid.reshape(len(X_grid), 1) #Why do we need to reshape?
(https://www.tutorialspoint.com/numpy/numpy_reshape.htm)
    # Visualizing the Polymonial Regression results
    plt.scatter(X, y, color='red')
    plt.plot(X_grid, pol_reg.predict(poly_reg.fit_transform(X_grid)), color='blue')
    plt.title('Truth or Bluff (Linear Regression)')
    plt.xlabel('Position level')
    plt.ylabel('Salary')
    plt.show()
    return
viz_polymonial_smooth()

# Predicting a new result with Linear Regression
lin_reg.predict([[5.5]])
#output should be 249500

# Predicting a new result with Polymonial Regression
pol_reg.predict(poly_reg.fit_transform([[5.5]]))
#output should be 132148.43750003
```