# 1. BASIC PROGRAMS USING NUMPY, PANDAS AND SCIPY

**Aim –** To install python environment and understand basic python libraries
**Objective –**
• To understand the basics of machine learning.
• To understand basic libraries in python and implement it.

**INSTALLATION OF PYTHON**
Once you've downloaded the Python for Windows installation file you should be prompted to run it. If not, open your Downloads folder and double-click the file. Now, follow the installation instructions on screen to install Python in the default location, as follows:
1. Click Install Now.
2. When asked whether to allow the program to make changes to your computer, choose Next
3. Click Close once installation finishes, and you should see a Python 3 entry in your Windows Start menu:

Programs involving Numpy, Pandas and Scipy libraries:

**Programs using Numpy:**

**1) Demonstrate broadcasting in NumPy with an example:**

```
import numpy as np
A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])
B = np.array([1, 2, 3])
print("Multiplication with broadcasting: ")
print(A * B)
print("... and now addition with broadcasting: ")
print(A + B)
```

**OUTPUT:**

```
Multiplication with broadcasting:
[[11 24 39]
 [21 44 69]
 [31 64 99]]
... and now addition with broadcasting:
[[12 14 16]
 [22 24 26]
 [32 34 36]]
```

**2) Demonstrate NumPy arithmetic and Matrices operations:**

#Arithmetic operations:

```
import numpy as np
a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])
add_ans = np.add(a, b)
print(add_ans)
sum=np.sum(add_ans)
print(sum)
sub_ans = np.subtract(a, b)
print(sub_ans)
mul_ans = np.multiply(a, b)
print(mul_ans)
div_ans = np.divide(a, b)
print(div_ans)
```

**OUTPUT:**

```
[  7  77  23 130]
237
[ 3 67  3 70]
[  10  360  130 3000]
[ 2.5        14.4         1.3         3.33333333]
```

#Matrices operations:

```
import numpy as np
# create two matrices
matrix1 = np.array([[1, 3], [5, 7]])
matrix2 = np.array([[2, 6], [4, 8]])
# Performing mod on two matrices
mod_ans = np.mod(matrix1, matrix2)
print("mod_ans : \n", mod_ans)
#Performing remainder on two matrices
rem_ans = np.remainder(matrix1,matrix2)
print("rem_ans : \n", rem_ans)
# Performing power of two matrices
pow_ans = np.power(matrix1, matrix2)
```

```
print("pow_ans : \n", pow_ans)
# calculate the dot product of the two matrices
result = np.dot(matrix1, matrix2)
print("result : \n", result)
#  get transpose of matrix1
result = np.transpose(matrix1)
print("result : \n", result)
```

**OUTPUT:**

```
mod_ans :
 [[1 3]
 [1 7]]
rem_ans :
 [[1 3]
 [1 7]]
pow_ans :
 [[      1     729]
 [    625 5764801]]
result :
 [[14 30]
 [38 86]]
result :
 [[1 5]
 [3 7]]
```

**3) Write a NumPy program to create a structured array from given student name, height, class and their data types and then sort the array on height:**

```
import numpy as np
data_type = [('name', 'S15'), ('class', int), ('height', float)]
students_details = [('James', 5, 48.5), ('Nail', 6, 52.5),('Paul', 5, 42.10), ('Pit', 5, 40.11)]
students = np.array(students_details, dtype=data_type)
print("Original array:")
print(students)
print("Sort by height:")
print(np.sort(students, order='height'))
```

**OUTPUT:**

```
Original array:
[(b'James', 5, 48.5 ) (b'Nail', 6, 52.5 ) (b'Paul', 5, 42.1 )
 (b'Pit', 5, 40.11)]
Sort by height:
[(b'Pit', 5, 40.11) (b'Paul', 5, 42.1 ) (b'James', 5, 48.5 )
 (b'Nail', 6, 52.5 )]
```

**4) Write a NumPy program to compute determinant of square array:**

```
import numpy as np
n_array = np.array([[50, 29], [30, 44]])
det = np.linalg.det(n_array)
print("\nDeterminant of given 2X2 matrix:")
print(int(det))
```

**OUTPUT:**

```
Determinant of given 2X2 matrix:
1330
```

**5) Demonstrate NumPy indexing and Slicing with an example:**

```
#Indexing: #Get a value at an index
import numpy as np
arr = np.arange(16)
print("arr : ", arr)
#Get a value at an index
print("Element at index 10 : ", arr[10])
#Get values in a range
print(arr[1:7])
```

**OUTPUT:**

```
arr :  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
Element at index 10 :  10
[1 2 3 4 5 6]
```

```
#Slicing: #Single dimension array
import numpy as np
a = np.arange(10)
b = a[2:7:2]
print(b)
# Multi dimensional array
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print(a)
print(a[1:])
print(a[1][2])
print(a[2])
```

**OUTPUT:**

```
[2 4 6]
[[1 2 3]
 [3 4 5]
 [4 5 6]]
[[3 4 5]
 [4 5 6]]
5
[4 5 6]
```

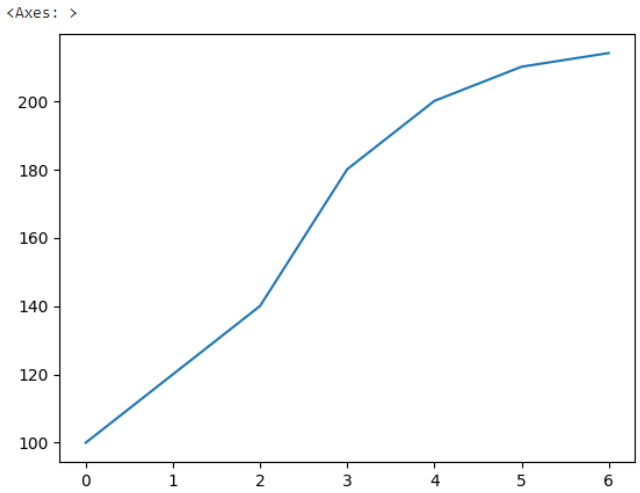
**Programs using Pandas:**

**1) Demonstrate line and Bar plotting in Pandas:**

#for line plot

```
import pandas as pd
data = [100, 120, 140, 180, 200, 210, 214]
s = pd.Series(data, index=range(len(data)))
s.plot()
```

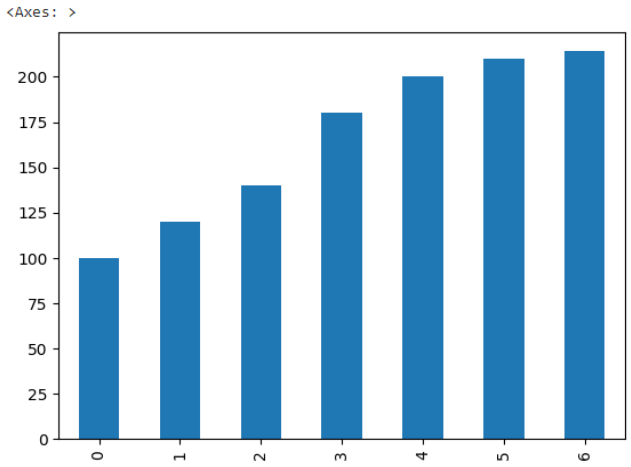
**OUTPUT:**




# For bar plot

```
import pandas as pd
data = [100, 120, 140, 180, 200, 210, 214]
s = pd.Series(data, index=range(len(data)))
s.plot(kind="bar")
```

**OUTPUT:**




**2) Demonstrate Series and Data Frame in Pandas:**

```
import pandas as pd
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
print(S)
# Creating a DataFrame from a dictionary
data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David'],
   'Age': [25, 30, 22, 28],
   'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
print(df)
```

**OUTPUT:**

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
      Name  Age           City
0    Alice   25       New York
1      Bob   30  San Francisco
2  Charlie   22    Los Angeles
3    David   28        Chicago
```

**Programs using Scipy:**

**1) Demonstrate how to solve linear equation in SciPy:**

```
import numpy as np
from scipy import linalg
# Creating the input array
a = np.array([[2, 1], [4,-5]])
# Creating the solution Array
b = np.array([[5], [7]])
# Solve the linear algebra to find values of x and y that satisfy the equations
x = linalg.solve(a, b)
# Printing the results
print(x)
```

**OUTPUT:**

```
[[2.28571429]
 [0.42857143]]
```

**2) Demonstrate how to find Determinant and inverse of matrices in SciPy:**

```
from scipy import linalg
import numpy as np
#Making the numpy array
mat = np.array([[1,2,3],[3,4,5],[7,8,6]])
#Passing the matrix  to the det function
mat1 = linalg.det(mat)
#printing the determinant
print(f'Determinant of the matrix\n{mat} \n is {mat1}')
#finding the inverse of the matrix using the inv() function
mat2 = linalg.inv(mat)
#printing the original and inverse matrices
print(f'Inverse of the matrix\n{mat} \n is \n{mat2}')
```

**OUTPUT:**

```
Determinant of the matrix
[[1 2 3]
 [3 4 5]
 [7 8 6]]
 is 6.0
Inverse of the matrix
[[1 2 3]
 [3 4 5]
 [7 8 6]]
 is
[[-2.66666667  2.         -0.33333333]
 [ 2.83333333 -2.5         0.66666667]
 [-0.66666667  1.         -0.33333333]]
```

# 2. LINEAR REGRESSION

**Aim –** To write a program on Linear regression
**Objective –** Applying linear regression to a dataset
**Theory –**
Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.
Linear regression fits a straight line or surface that minimizes the discrepancies between predicted and actual output values. There are simple linear regression calculators that use a “least squares” method to discover the best-fit line for a set of paired data.

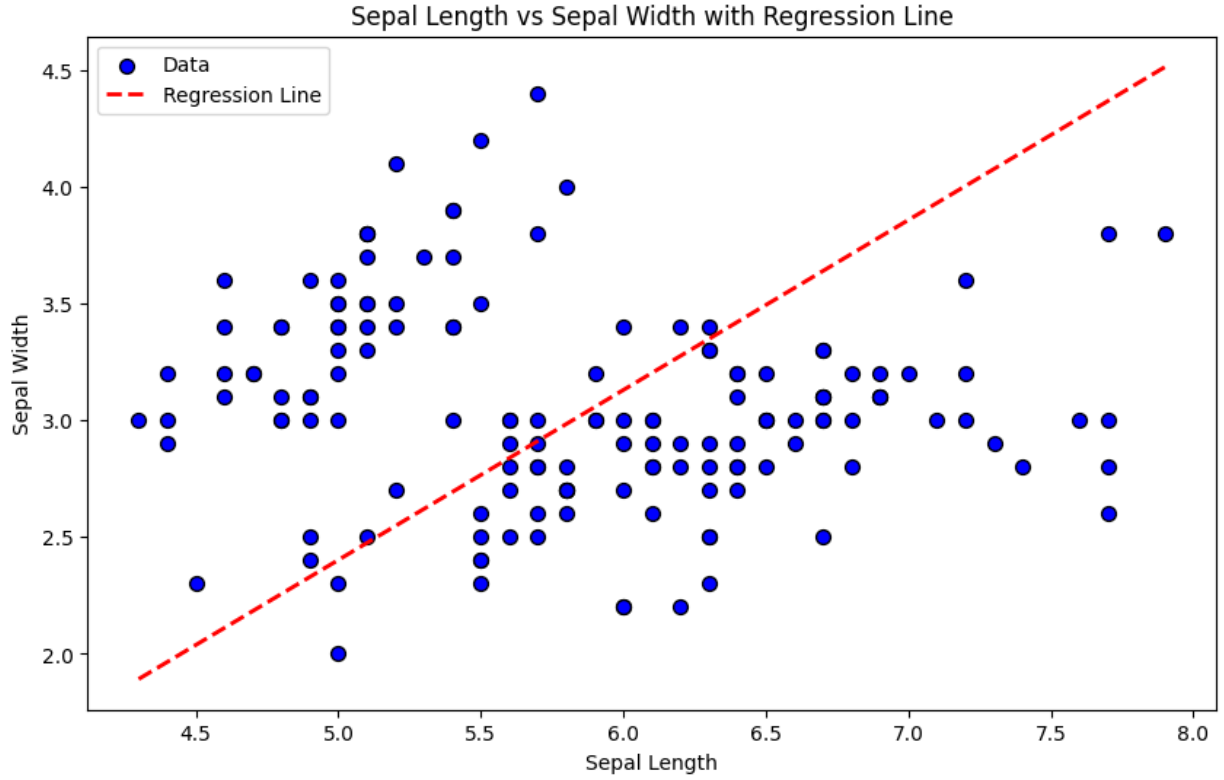
**Code –**

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Extract sepal length (feature 0) and sepal width (feature 1)
X_sepal = X[:, [0, 1]]  # Select sepal length and sepal width

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_sepal, y, test_size=0.25, random_state=42)

# Create and train a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Convert the predicted values to integers
y_pred = y_pred.round().astype(int)
```

```
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
# Plotting
plt.figure(figsize=(10, 6))
# Scatter plot of Sepal Length vs Sepal Width
plt.scatter(X_sepal[:, 0], X_sepal[:, 1], color='blue', edgecolor='k', s=50, label='Data')
# To plot the regression line
# Generate a range of values for sepal length (feature 0)
x_range = np.linspace(X_sepal[:, 0].min(), X_sepal[:, 0].max(), 100)
# Predict sepal width (feature 1) values using the linear model
y_range = model.coef_[0] * x_range + model.intercept_
# Plot the regression line
plt.plot(x_range, y_range, color='red', linestyle='--', linewidth=2, label='Regression Line')
# Set labels and title
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Sepal Length vs Sepal Width with Regression Line')
plt.legend()
plt.show()
```

**OUTPUT:**

```
Accuracy: 0.8421052631578947
```



# 3. LOGISTIC REGRESSION

**Aim** – To write a program on logistic regression

**Objective –** Applying logistic regression to a dataset

**Theory –**

Logistic regression is a supervised machine learning algorithm that accomplishes binary classification tasks by predicting the probability of an outcome, event, or observation. The model delivers a binary or dichotomous outcome limited to two possible outcomes: yes/no, 0/1, or true/false. Logistic regression is a supervised machine learning algorithm that accomplishes binary classification tasks by predicting the probability of an outcome, event, or observation. The model delivers a binary or dichotomous outcome limited to two possible outcomes: yes/no, 0/1, or true/false.

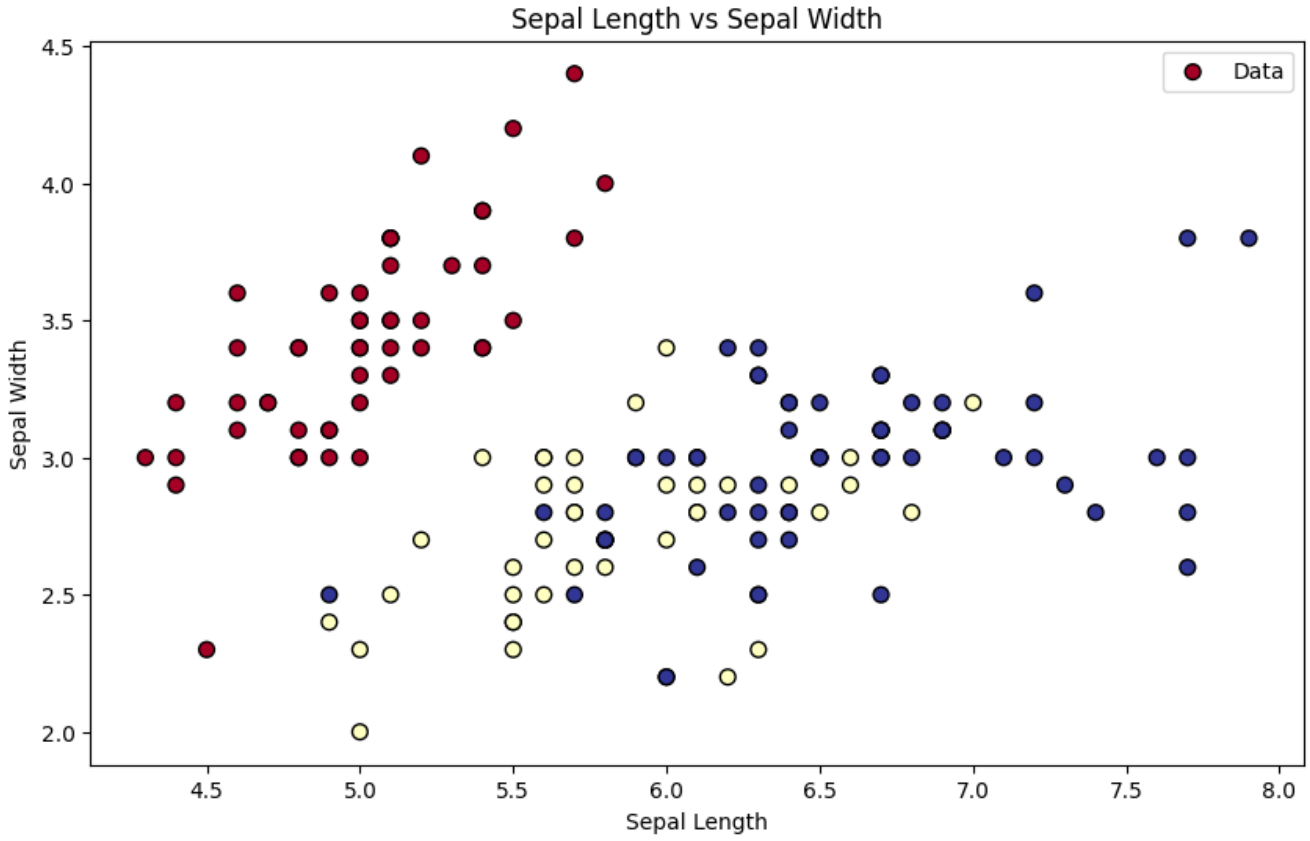
**Code –**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Extract sepal length (feature 0) and sepal width (feature 1)
X_sepal = X[:, [0, 1]]  # Select sepal length and sepal width

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_sepal, y, test_size=0.20, random_state=40)

# Create a LogisticRegression instance
logreg = LogisticRegression(max_iter=200)

# Fit the training data to the model
logreg.fit(X_train, y_train)

# Predict the labels for the test data
y_pred = logreg.predict(X_test)
```

```
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Plotting
plt.figure(figsize=(10, 6))

# Scatter plot of Sepal Length vs Sepal Width
plt.scatter(X_sepal[:, 0], X_sepal[:, 1], c=y, edgecolor='k', s=50, cmap=plt.cm.RdYlBu, marker='o',
label='Data')

# Set labels and title
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Sepal Length vs Sepal Width')
plt.legend()
plt.show()
```

**OUTPUT:**

```
Accuracy: 0.9333333333333333
```



# 4. DECISION TREE

**Aim** – To write a program on decision tree

**Objective –** To make a tree that clusters its points based on similarity factors

**Theory –**

A decision tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. It has a hierarchical, tree structure, which consists of a root node, branches, internal nodes and leaf nodes.

Decision tree learning employs a divide and conquer strategy by conducting a greedy search to identify the optimal split points within a tree. This process of splitting is then repeated in a top-down, recursive manner until all, or the majority of records have been classified under specific class labels. Whether or not all data points are classified as homogenous sets is largely dependent on the complexity of the decision tree. Smaller trees are more easily able to attain pure leaf nodes—i.e. data points in a single class.

**Algorithms** – • CART (Gini Index)

• ID3 (Entropy, Information Gain)

Step-1: Creating a root node

Entropy(Entropy of whole data-set)

Entropy (S) = (-p/p+n)*log2 (p/p+n) - (n/n+p)*log2 ((n/n+p))

p- p stand for number of positive examples

n- n stand for number of negative examples.

Step-2: For Every Attribute/Features

Average Information (AIG of a particular attribute)

I(Attribute) = Sum of {(pi+ni/p+n)*Entropy(Entropy of Attribute)}

pi- Here pi stand for number of positive examples in particular attribute.

ni- Here ni stand for number of negative examples in particular attribute.

Entropy (Attribute) - Entropy of Attribute calculated in same as we calculated for System (Whole Data-Set)

Step-3: Information Gain

Gain = Entropy(S) - I(Attribute)

1. If all examples are positive, Return the single-node tree ,with label=+
2. If all examples are Negative, Return the single-node tree,with label= -
3. If Attribute empty, Return the single-node tree

Step-4: Pick the Highest Gain Attribute

1. The attribute that has the most information gain has to create a group of all the its attributes and process them insame as which we have done for the parent (Root) node.
2. Again, the feature which has maximum information gain will become a node and this process will continueuntil we get the leaf node.

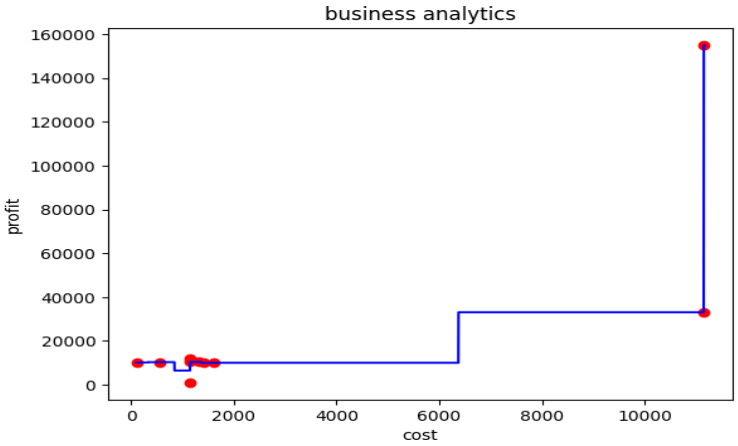
Step-5: Repeat Until we get final node (Leaf node).

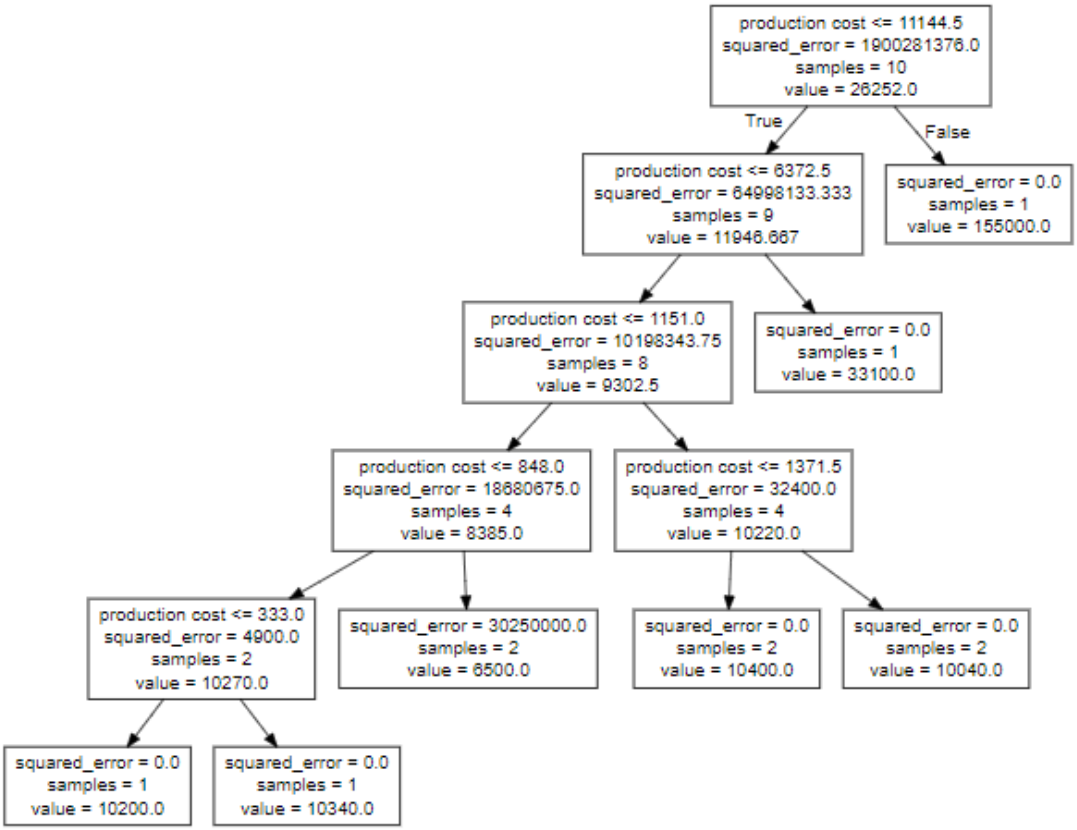
**Code –**

```
!pip install graphviz
!apt install libgraphviz-dev

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
dataset = np.array([
 ['famous engg works', 11155,155000],
 ['asset flip', 11134,33100],
 ['bmw', 555,10340],
 ['mjcet', 1332,10400],
 ['tax', 1141,12000],
 ['jeee', 1411,10040],
 ['yoyo', 111,10200],
 [' flip', 1141,1000],
 ['nessain', 1161,10400],
 ['asset', 1611,10040],
])
print(dataset)
x = dataset[:,1:2].astype(int)
y = dataset[:,2].astype(int)

from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor(random_state=0)
regressor.fit(x,y)
y_pred = regressor.predict([[111]])
print (y_pred)
x_grid = x_grid.reshape((len(x_grid),1))
plt.scatter(x,y,color ='red')
plt.plot(x_grid,regressor.predict(x_grid), color ='blue')
plt.title('business analytics')
plt.xlabel('cost')
plt.ylabel('profit')
plt.show()

from sklearn.tree import export_graphviz
export_graphviz(regressor ,out_file ='tree.dot',feature_names=['production cost'])
```

**OUTPUT:**

```
[['famous engg works' '11155' '155000']
 ['asset flip' '11134' '33100']
 ['bmw' '555' '10340']
 ['mjcet' '1332' '10400']
 ['tax' '1141' '12000']
 ['jeee' '1411' '10040']
 ['yoyo' '111' '10200']
 [' flip' '1141' '1000']
 ['nessain' '1161' '10400']
 ['asset' '1611' '10040']]
[10200.]
```



# 5. NAÏVE BAYES

**Aim –** To write a program on Naïve Bayes

**Objective –** To classify new instances based on their probability of belonging to different classes

**Theory –**

Naive Bayes is a probabilistic machine learning algorithm that is based on Bayes' theorem with the assumption of independence between features. It is commonly used for classification tasks and is particularly suited for text classification and spam filtering.

Bayes theorem is a formula that offers a conditional probability of an event A taking happening given another event B has previously happened. Its mathematical formula is as follows: –

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)}$$

Where,

A and B are two events

P(A|B) is the probability of event A provided event B has already happened.

P(B|A) is the probability of event B provided event A has already happened.

P(A) is the independent probability of A

P(B) is the independent probability of B

**Code –**

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics
iris = datasets.load_iris()
x = iris.data
y = iris.target
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 50)
naive_bayes = GaussianNB()
naive_bayes.fit(x_train, y_train)
y_pred = naive_bayes.predict(x_test)
acc = metrics.accuracy_score(y_test, y_pred)
print("Accuracy : ", acc)
new_instance = [[5.0, 3.5, 1.4, 2.2]]
predicted = naive_bayes.predict(new_instance)
print("Predicted class ", predicted)
```

**OUTPUT:**

```
Accuracy :  0.9555555555555556
Predicted class  [2]
```

# 6. K-NEAREST NEIGHBOUR

**Aim –** To write a program on K-Nearest Neighbor

**Objective –** To classify new data points based on their similarity to existing labelled data points

**Theory –**

The K-Nearest Neighbors (KNN) algorithm is a non-parametric and supervised machine learning algorithm. It is based on the principle that objects that are close to each other in a multi-dimensional space tend to belong to the same class or category. KNN is a lazy learning algorithm, meaning that it does not explicitly learn a model during the training phase. Instead, it stores the entire training dataset in memory to make predictions.

**Code –**

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

iris = datasets.load_iris()
x = iris.data
y = iris.target

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3)

knn = KNeighborsClassifier(n_neighbors = 3)

knn.fit(x_train, y_train)

y_pred = knn.predict(x_test)
acc = metrics.accuracy_score(y_test, y_pred)
print("Accuracy : ", acc)
new_instance = [[5.0, 3.5, 1.4, 2.2]]
predicted = knn.predict(new_instance)
print("Predicted class ", predicted)
```

**OUTPUT:**

```
Accuracy :  0.9777777777777777
Predicted class  [0]
```

# 7. SUPPORT VECTOR MACHINE

**Aim –** To write a program on Support Vector Machine

**Objective –** To find an optimal hyperplane that separates different classes in a high-dimensional feature space.

**Theory –**

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It is particularly effective in solving complex classification problems with a clear margin of separation between classes.

**Code –**

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score

# Load the diabetes dataset

diabetes_data = pd.read_csv('diabetes.csv')

# Split the dataset into features (X) and target variable (y)

X = diabetes_data.drop('Outcome', axis=1)

y = diabetes_data['Outcome']

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an SVM classifier

svm = SVC(kernel='linear')

# Train the classifier on the training data

svm.fit(X_train, y_train)

# Make predictions on the test data

y_pred = svm.predict(X_test)

# Calculate the accuracy of the classifier

accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

**OUTPUT:**

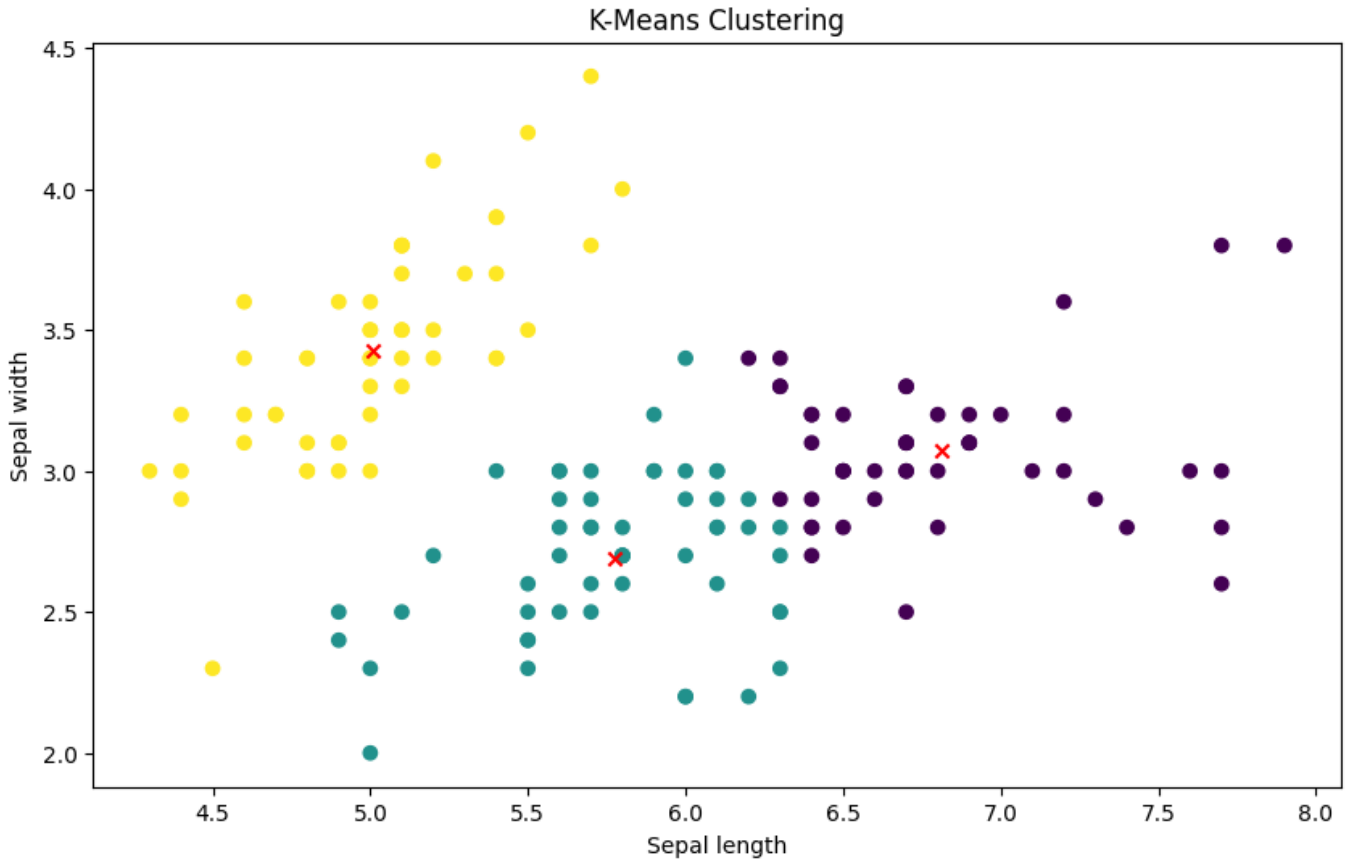
```
Accuracy: 0.7532467532467533
```

# 8. K-MEANS CLUSTERING

**Aim** – To demonstrate k-means clustering algorithm
**Objective –** Plotting of clusters using k-means clustering algorithm
**Theory –**
K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabelled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on.
It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabelled dataset on its own without the need for any training. It is a centroid-based algorithm, where each cluster is associated with a centroid.
The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.
**Code** –

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

# Load the Iris dataset
iris = load_iris()
X = iris.data[:, :2]

# We only take the first two features for visualization
kmeans = KMeans(n_clusters=3, random_state=42)
# Fit the algorithm to the data
kmeans.fit(X)

# Predict the cluster labels for the data points
labels = kmeans.labels_

# Get the coordinates of the cluster centers
centers = kmeans.cluster_centers_

# Visualize the clusters
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x')
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title('K-Means Clustering')
plt.show()
```

**OUTPUT:**


K-Means Clustering
Sepal width
Sepal length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

# 9. HIERARCHICAL CLUSTERING

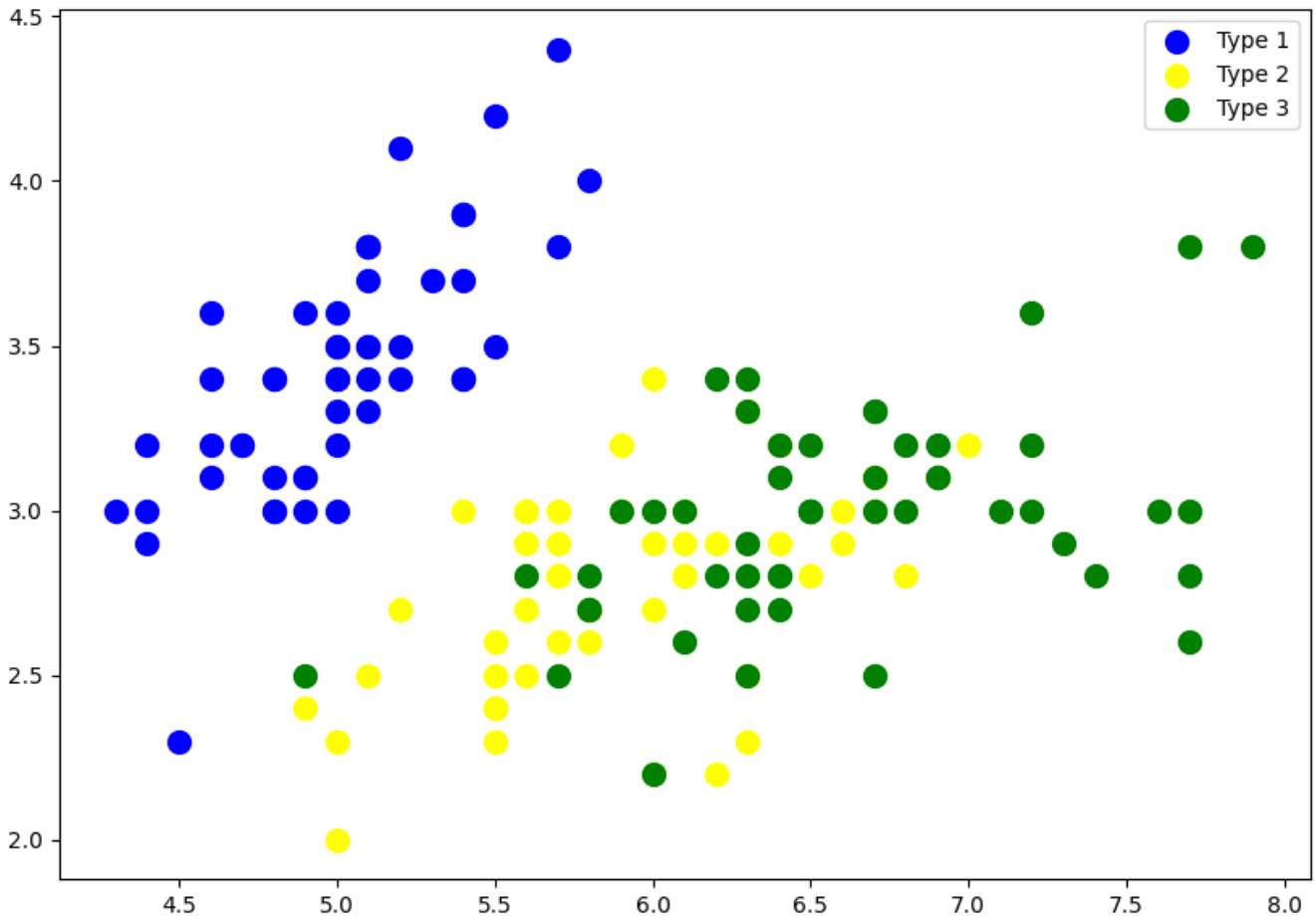
**Aim –** To demonstrate hierarchical clustering algorithm
**Objective –** Plotting of clusters using hierarchical clustering algorithm
**Theory –**
Hierarchical clustering is another unsupervised machine learning algorithm, which is used to group the unlabelled datasets into a cluster and known as hierarchical cluster analysis or HCA.
In this algorithm, we develop the hierarchy of clusters in the form of a tree, and this tree shaped structure is known as the dendrogram.
Sometimes the results of K-means clustering and hierarchical clustering may look similar, but they both differ depending on how they work. As there is no requirement to predetermine the number of clusters as we did in the K-Means algorithm.
The hierarchical clustering technique has two approaches:
1. Agglomerative: Agglomerative is a bottom-up approach, in which the algorithm starts with taking all data points as single clusters and merging them until one cluster is left.
2. Divisive: Divisive algorithm is the reverse of the agglomerative algorithm as it is a top-down approach.

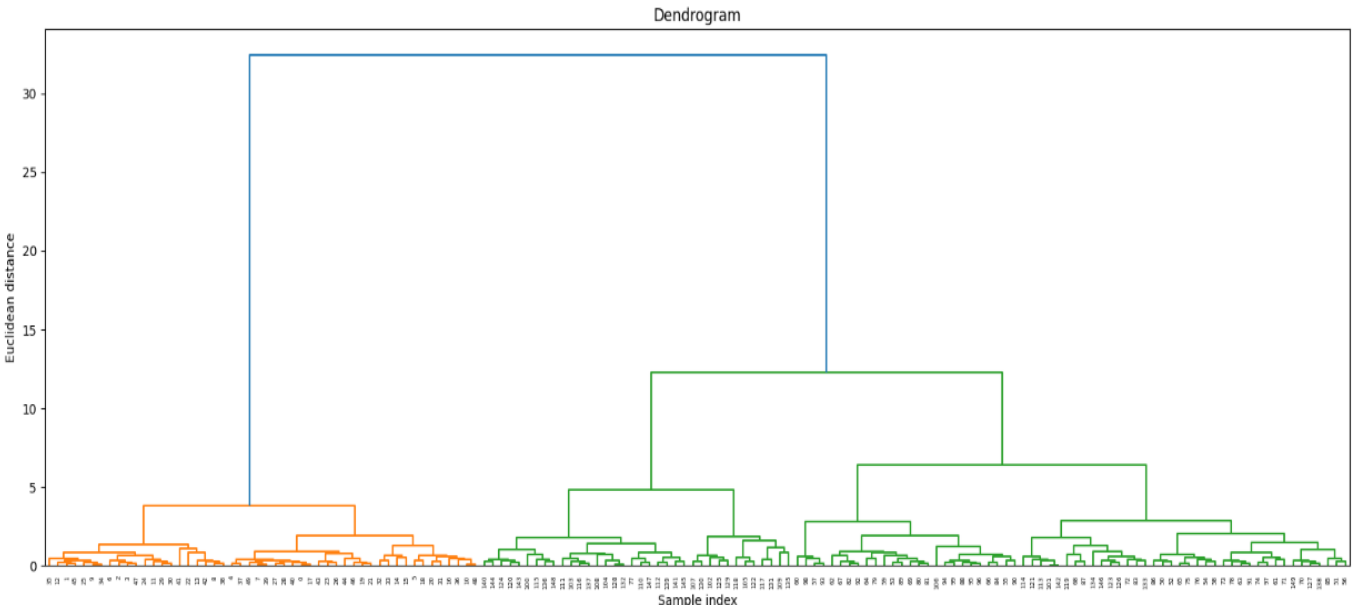
**Code –**

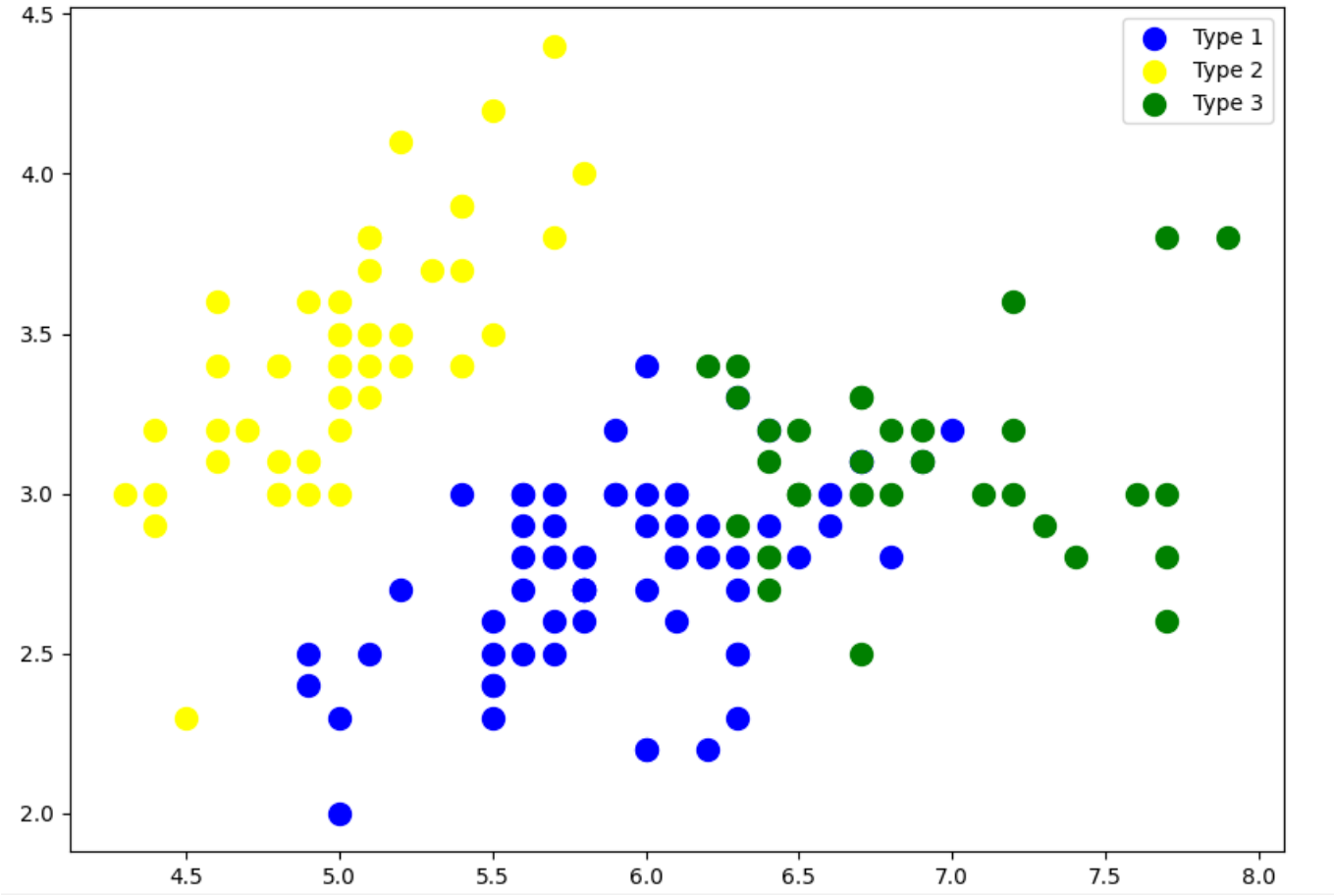
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
# Import iris data
iris = datasets.load_iris()
iris_data = pd.DataFrame(iris.data)
iris_data.columns = iris.feature_names
iris_data['flower_type'] = iris.target
iris_data.head()
iris_X = iris_data.iloc[:, [0, 1, 2,3]].values
iris_Y = iris_data.iloc[:,4].values
plt.figure(figsize=(10, 7))
plt.scatter(iris_X[iris_Y == 0, 0], iris_X[iris_Y == 0, 1], s=100, c='blue', label='Type 1')
plt.scatter(iris_X[iris_Y == 1, 0], iris_X[iris_Y == 1, 1], s=100, c='yellow', label='Type 2')
plt.scatter(iris_X[iris_Y == 2, 0], iris_X[iris_Y == 2, 1], s=100, c='green', label='Type 3')
plt.legend()
plt.show()

import scipy.cluster.hierarchy as sc
# Plot dendrogram
plt.figure(figsize=(20, 7))
```

```
plt.title("Dendrograms")
# Create dendrogram
sc.dendrogram(sc.linkage(iris_X, method='ward'))
plt.title('Dendrogram')
plt.xlabel('Sample index')
plt.ylabel('Euclidean distance')
from sklearn.cluster import AgglomerativeClustering
cluster = AgglomerativeClustering( n_clusters=3, affinity='euclidean', linkage='ward')
cluster.fit(iris_X)
labels = cluster.labels_
plt.figure(figsize=(10, 7))
plt.scatter(iris_X[labels == 0, 0], iris_X[labels == 0, 1], s = 100, c = 'blue', label = 'Type 1')
plt.scatter(iris_X[labels == 1, 0], iris_X[labels == 1, 1], s = 100, c = 'yellow', label = 'Type 2')
plt.scatter(iris_X[labels == 2, 0], iris_X[labels == 2, 1], s = 100, c = 'green', label = 'Type 3')
plt.legend()
plt.show()
```

**OUTPUT:**




Dendrogram
Euclidean distance
Sample index
0
5
10
15
20
25
30


Type 1
Type 2
Type 3
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

# 10. TEXT CLASSIFICATION USING PYTHON LIBRARIES (NEURAL NETWORKS)

**Aim –** To demonstrate Text classification using python Libraries
**Objective –** Classification of handwritten text using python libraries
**Theory –**
Text classification is a machine learning task where we assign predefined categories or labels to a given text based on its content. In this case, the aim is to classify handwritten digits using a Neural Network Classifier.
A Neural Network Classifier is a machine learning model inspired by the structure and functioning of the human brain. It consists of interconnected nodes (neurons) organized in layers. In the context of the code you provided, the classifier being used is a Multi-Layer Perceptron (MLP) Classifier, which is a type of feed forward neural network.

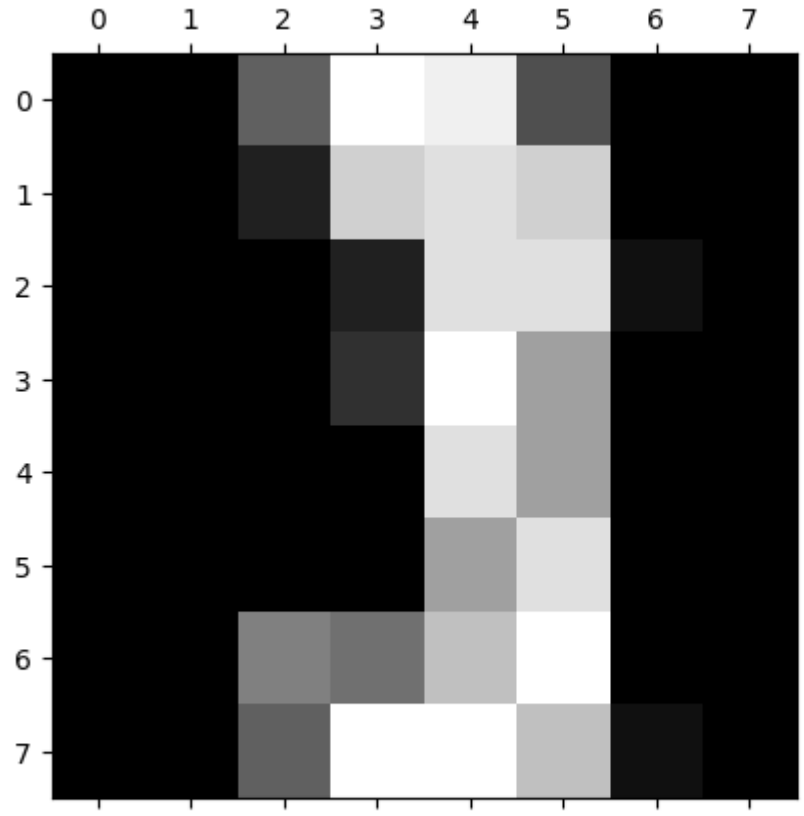
**Code –**

```
# Step 1  - Importing the necessary libraries
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import  accuracy_score

# Step 2 - Loading the dataset
dataset = load_digits()

# Step 3 - Splitting the data into test and train
x_train, x_test, y_train, y_test = train_test_split(dataset.data, dataset.target, test_size=0.20)

# Step 4 - Making the Neural Network Classifier
NN = MLPClassifier()

# Step 5
# Training the model on the training data and labels
NN.fit(x_train, y_train)

# Step 6
# Testing the model i.e. predicting the labels of the test data.
y_pred = NN.predict(x_test)
```

```
# Step 7 - Evaluating the results of the model
accuracy = accuracy_score(y_test,y_pred)*100


# Step 8 - Printing the Results
print("Accuracy for Neural Network is:",accuracy)

import matplotlib.pyplot as plt
plt.gray()
plt.matshow(dataset.images[1630])
plt.show()
print(dataset.data[1630])
```

**OUTPUT:**

```
Accuracy for Neural Network is: 96.94444444444444
<Figure size 640x480 with 0 Axes>
```




```
[ 0.  0.  6. 16. 15.  5.  0.  0.  0.  0.  2. 13. 14. 13.  0.  0.  0.  0.
  0.  2. 14. 14.  1.  0.  0.  0.  0.  3. 16. 10.  0.  0.  0.  0.  0.  0.
 14. 10.  0.  0.  0.  0.  0.  0. 10. 14.  0.  0.  0.  0.  8.  7. 12. 16.
  0.  0.  0.  0.  6. 16. 16. 12.  1.  0.]
```

# 11. CONVOLUTIONAL NEURAL NETWORKS (CNN)

**Aim-** To demonstrate how to apply convolutional layer, activation layer and pooling layer operation to extract the inside features of an image.

**Objective –**Applying convolutional layer, activation layer and pooling layer operation to extract the inside features of an image.

**Theory –**

Convolutional Neural Networks (CNNs) are a specialized type of deep neural network designed for processing and analysing grid-like data, such as images and other structured data. CNNs are particularly effective for tasks like image recognition, object detection, and image classification due to their ability to automatically learn hierarchical features directly from the data.

Convolution:

Convolution is a fundamental operation in CNNs. It involves sliding a small filter (also known as a kernel) over the input image to extract features. This operation helps identify patterns like edges, textures, and more complex structures.

Pooling (Down sampling):

Pooling layers reduce the spatial dimensions of the input data while retaining its important features. Max pooling and average pooling are common types of pooling layers. Pooling helps reduce the computational load and makes the network more robust to variations in the input.

Convolutional Layers:

These layers consist of a set of learnable filters. Each filter detects specific patterns or features in the input data. Convolutional layers capture spatial hierarchies of features, enabling the network to learn from local to global patterns.

Fully Connected (Dense) Layers:

These layers come after the convolutional and pooling layers. They're similar to the layers in traditional neural networks and are responsible for making the final predictions or classifications.

Back propagation and Gradient Descent:

Like other neural networks, CNNs are trained using back propagation and gradient descent. The network's weights and biases are updated iteratively to minimize a loss function, improving its ability to make accurate predictions.

**Code –**

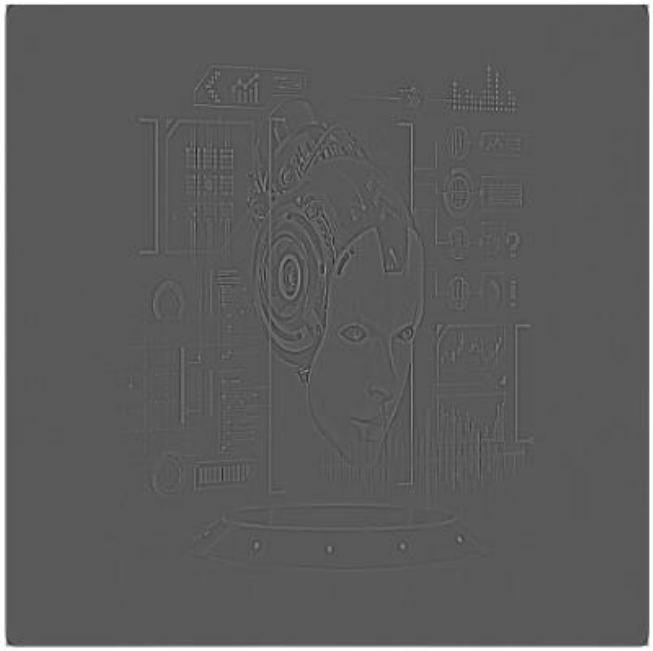
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Set the parameters for plotting
plt.rc('figure', autolayout=True)
```

```
plt.rc('image', cmap='magma')

# Define the kernel
kernel = tf.constant([[-1, -1, -1],
                      [-1, 8, -1],
                      [-1, -1, -1]], dtype=tf.float32)

# Load the image
image = tf.io.read_file('ml.jpg')
image = tf.io.decode_jpeg(image, channels=1)
image = tf.image.resize(image, size=[500, 500])

# Plot the original image
plt.figure(figsize=(5, 5))
plt.imshow(tf.squeeze(image).numpy(), cmap='gray')
plt.axis('off')
plt.title('Original Gray Scale Image')
plt.show()

# Reformat the image and kernel
image = tf.image.convert_image_dtype(image, dtype=tf.float32)
image = tf.expand_dims(image, axis=0)
kernel = tf.reshape(kernel, [*kernel.shape, 1, 1])

# Perform convolution
conv_fn = tf.nn.conv2d
image_filter = conv_fn(input=image, filters=kernel, strides=1, padding='SAME')

# Apply ReLU activation
relu_fn = tf.nn.relu
image_detect = relu_fn(image_filter)

# Apply max pooling
pooling_window_shape = (2, 2)
pooling_strides = (2, 2)
image_condense = tf.nn.max_pool2d(input=image_detect,
                                  ksize=pooling_window_shape,
                                  strides=pooling_strides,
                                  padding='SAME')

# Plot the results
```

```
plt.figure(figsize=(15, 5))

# Plot the convolved image
plt.subplot(1, 3, 1)
plt.imshow(tf.squeeze(image_filter).numpy(), cmap='gray')
plt.axis('off')
plt.title('Convolution')

# Plot the activated image
plt.subplot(1, 3, 2)
plt.imshow(tf.squeeze(image_detect).numpy(), cmap='gray')
plt.axis('off')
plt.title('Activation')

# Plot the pooled image
plt.subplot(1, 3, 3)
plt.imshow(tf.squeeze(image_condense).numpy(), cmap='gray')
plt.axis('off')
plt.title('Pooling')

plt.show()
```

**OUTPUT:**

Convolution



Activation

Pooling

# 12. RANDOM FOREST

**Aim** – To demonstrate random forest ensemble technique

**Objective –** To understand the working of random forest algorithm

**Theory –**

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model. As the name suggests, "Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output. The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

Since the random forest combines multiple trees to predict the class of the dataset, it is possible that some decision trees may predict the correct output, while others may not. But together, all the trees predict the correct output. Therefore, below are two assumptions for a better Random

Forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.

**The Working process can be explained in the below steps and diagram:**

Step-1: Select random K data points from the training set.

Step-2: Build the decision trees associated with the selected data points (Subsets).

Step-3: Choose the number N for decision trees that you want to build.

Step-4: Repeat Step 1 & 2.

Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

**Code –**

```
import pandas as pd

df = pd.read_csv("diabetes.csv")

df.head()

df.isnull().sum()

X = df.drop("Outcome",axis="columns")

y = df.Outcome

from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_scaled[:3]
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, stratify=y, random_state=10)
X_train.shape
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
scores = cross_val_score(RandomForestClassifier(n_estimators=50), X, y, cv=5)
scores.mean()
```

**OUTPUT:**

```
0.7605126899244545
```

# 13. BOOSTING

**Aim** – To demonstrate boosting ensemble technique
**Objective –** To understand the working of boosting algorithm
**Theory –**
Boosting is an ensemble learning technique that combines multiple weak or base learners to create a strong predictive model. It is designed to improve the overall accuracy of the model by sequentially training base learners, where each subsequent learner focuses on correcting the mistakes made by the previous ones.
The boosting process starts by training an initial base learner on the entire dataset. Then, it assigns higher weights to the misclassified instances, emphasizing their importance in subsequent iterations. In each iteration, a new base learner is trained on the modified dataset to give more attention to the previously misclassified instances. This iterative process continues until a predefined number of base learners are trained or a desired level of accuracy is achieved.
During the prediction phase, the base learners' predictions are combined using a weighted voting scheme, where the weights are determined based on their individual performance during training. The final prediction is typically obtained by taking a weighted majority vote or averaging the predictions of all base learners.
The most popular boosting algorithm is AdaBoost (Adaptive Boosting), which adjusts the weights of instances based on their difficulty to classify correctly. Another commonly used algorithm is Gradient Boosting, which minimizes a loss function by iteratively adding new base learners. Boosting has proven to be highly effective in handling complex classification and regression tasks, often outperforming individual base learners. It is known for its ability to reduce bias and variance, handle noisy data, and provide robust predictions. However, boosting can be sensitive to outliers and overfitting, so it's important to tune the model parameters carefully to achieve optimal performance.

**Code –**

```
import numpy as np
import pandas as pd
import xgboost
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

# Load the dataset
df = pd.read_csv("mushrooms.csv")

# Check column names
print("Columns in the DataFrame:", df.columns)

# Drop the 'veil-type' column if it exists
if "veil_type" in df.columns:
```

```
    df = df.drop("veil_type", axis=1)
else:
    print("'veil_type' column not found in the DataFrame")

# Display the first 6 rows of the DataFrame
df.head(6)

# Encode categorical features
label_encoder = LabelEncoder()
for column in df.columns:
    df[column] = label_encoder.fit_transform(df[column])

# Define features and target variable
X = df.loc[:, df.columns != 'type']  # Assuming 'type' is the target variable
Y = df['type']  # Set the target variable

# Split the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=100)

# Scale the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the AdaBoost model
adaboost = AdaBoostClassifier(n_estimators=50, learning_rate=0.2).fit(X_train, Y_train)

# Evaluate the model
score = adaboost.score(X_test, Y_test)
print("Model accuracy:", score)
```

**OUTPUT:**

```
Columns in the DataFrame: Index(['type', 'cap_shape', 'cap_surface', 'cap_color', 'bruises', 'odor',
       'gill_attachment', 'gill_spacing', 'gill_size', 'gill_color',
       'stalk_shape', 'stalk_root', 'stalk_surface_above_ring',
       'stalk_surface_below_ring', 'stalk_color_above_ring',
       'stalk_color_below_ring', 'veil_type', 'veil_color', 'ring_number',
       'ring_type', 'spore_print_color', 'population', 'habitat'],
      dtype='object')
Model accuracy: 0.9848236259228876
```

# 14. EVALUATION OF VARIOUS CLASSIFICATION ALGORITHMS PERFORMANCE ON A DATASET

**Aim** – To evaluate various classification algorithms performance on a dataset using various measures like True Positive Rate, False Positive Rate, Precision, Recall etc

**Objective –** Applying various measures to check the performance

**Theory –**

TRUE POSITIVE RATE

True Positive Rate (TPR), also known as Sensitivity, Recall, or Hit Rate, is a statistical metric used to evaluate the performance of a binary classification model. It measures the proportion of actual positive cases that are correctly identified as positive by the model.

TPR is calculated as the ratio of true positives (TP) to the sum of true positives and false negatives (FN)

TPR = TP / (TP + FN)

TPR is commonly used in medical diagnostics, machine learning, and other areas where correctly identifying positive instances is crucial. A higher TPR indicates better performance in correctly detecting positive cases, while a lower TPR suggests that the model is missing a significant number of positive instances.

FALSE POSITIVE RATE

False Positive Rate (FPR) is a statistical metric used to evaluate the performance of a binary classification model. It measures the proportion of actual negative cases that are incorrectly classified as positive by the model.

FPR is calculated as the ratio of false positives (FP) to the sum of false positives and true negatives (TN)

FPR = FP / (FP + TN)

FPR is particularly important when the cost or impact of false positives is high. For example, in medical testing, a high FPR could lead to unnecessary treatments or interventions for individuals who are actually healthy.

PRECISION

Precision is a statistical metric used to assess the performance of a binary classification model. It measures the proportion of correctly classified positive instances out of the total instances predicted as positive by the model. Precision is particularly relevant in situations where the cost or impact of false positives is high. It measures the model's ability to avoid false positives and make precise positive predictions.

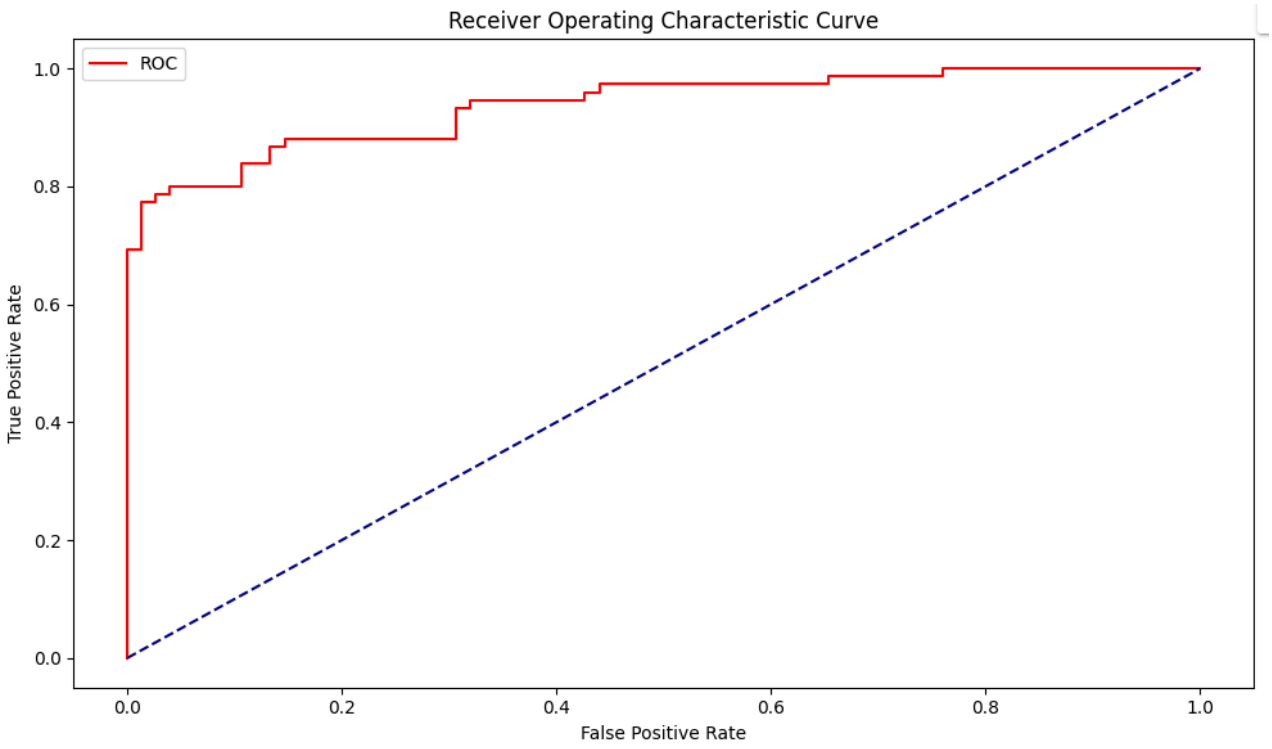
Precision is calculated as the ratio of true positives (TP) to the sum of true positives and false positives (FP)
Precision = TP / (TP + FP)
A high precision value suggests that the model has a low rate of false positives, meaning it is effective in identifying positive instances accurately.
RECALL
Recall, also known as Sensitivity or True Positive Rate, is a statistical metric used to evaluate the performance of a binary classification model. It measures the proportion of actual positive cases that are correctly identified as positive by the model. Recall is particularly important in situations where it is crucial to capture as many positive cases as possible, even if it results in a higher rate of false positives.
Recall is calculated as the ratio of true positives (TP) to the sum of true positives and false negatives (FN):
Recall = TP / (TP + FN)
A high recall value suggests that the model is effective in capturing positive instances and has a low rate of false negatives. However, a low recall value indicates a higher rate of false negatives, implying that the model is missing a significant number of positive instances.

**Code –**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from  sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
X, label = make_classification(n_samples=500, n_classes=2, weights=[1,1], random_state=100)
X_train, X_test, y_train, y_test = train_test_split(X, label, test_size=0.3, random_state=1)
model = LogisticRegression()
model.fit(X_train, y_train)
probs = model.predict_proba(X_test)
probs = probs[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, probs)
plt.figure(figsize = (10,6))
plt.plot(fpr, tpr, color='red', label='ROC')
plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic Curve')
plt.legend()
plt.show()
from sklearn import datasets
```

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
import matplotlib.pyplot as plt
data = datasets.load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:,:-1], df.iloc[:,-1], test_size=0.3,
random_state=42)
model = LogisticRegression()
model.fit(X_train, y_train)
pred = model.predict(X_test)
precision = precision_score(y_test, pred)
recall = recall_score(y_test, pred)
print('Precision: ',precision)
print('Recall: ',recall)
```

**OUTPUT:**




```
Precision:  0.963963963963964
Recall:  0.9907407407407407
```