

AIM: IMPLEMENT Naïve Bayesian Classifier with Different Datasets.**Program:**

```
# Importing necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Loading the dataset from CSV file

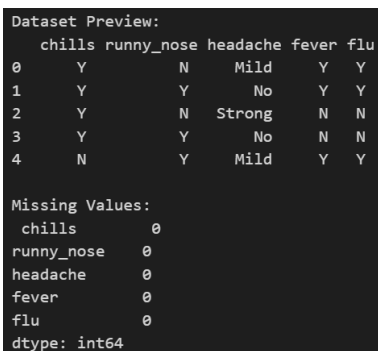
dataset = pd.read_csv("dataset.csv")

# Displaying first few rows

print("Dataset Preview:\n", dataset.head())

# Checking for missing values

print("\nMissing Values:\n", dataset.isnull().sum())
```

Output:

```
Dataset Preview:
  chills  runny_nose  headache  fever  flu
0      Y          N      Mild      Y    Y
1      Y          Y        No      Y    Y
2      Y          N   Strong      N    N
3      Y          Y        No      N    N
4      N          Y      Mild      Y    Y

Missing Values:
chills      0
runny_nose  0
headache    0
fever       0
flu         0
dtype: int64
```

```
# Encoding categorical variables manually

dataset.replace({"Y": 1, "N": 0, "Mild": 1, "No": 0, "Strong": 2}, inplace=True)

# Displaying the transformed dataset

print("\nEncoded Dataset:\n", dataset.head())
```

Output:

```
Encoded Dataset:
   chills  runny_nose  headache  fever  flu
0       1          0         1       1    1
1       1          1         0       1    1
2       1          0         2       0    0
3       1          1         0       0    0
4       0          1         1       1    1
C:\Users\G VENKATA SAI RAM\AppData\Local\Temp\ipykernel_18032\2867734653.py:2: FutureWarning:
  dataset.replace({"Y": 1, "N": 0, "Mild": 1, "No": 0, "Strong": 2}, inplace=True)
```

```
# Separating independent (X) and dependent (y) variables
```

```
X = dataset.iloc[:, :-1].values # All columns except 'flu'
```

```
y = dataset.iloc[:, -1].values # Target column 'flu'
```

```
# Splitting dataset into training (75%) and testing (25%) sets
```

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

```
# Checking training set distribution
```

```
print("\nTraining Set Class Distribution:\n", pd.Series(y_train).value_counts())
```

```
# Checking test set distribution
```

```
print("\nTest Set Class Distribution:\n", pd.Series(y_test).value_counts())
```

Output:

```
Training Set Class Distribution:
1     5
0     4
Name: count, dtype: int64

Test Set Class Distribution:
0     2
1     1
Name: count, dtype: int64
```

```
# Creating and training the Naive Bayes classifier
```

```
classifier = GaussianNB()
```

```
classifier.fit(X_train, y_train)
```

```
# Predicting test set results
```

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

```
# Printing predictions
```

```
print("Predicted Values:", y_pred)
```

```
Predicted Values: [1 0 0]
```

```
# Calculating accuracy
```

```
accuracy = accuracy_score(y_test, y_pred)
```

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

```
# Generating confusion matrix
```

```
cm = confusion_matrix(y_test, y_pred)
```

```
# Plotting confusion matrix
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
```

```
plt.title("Confusion Matrix")
```

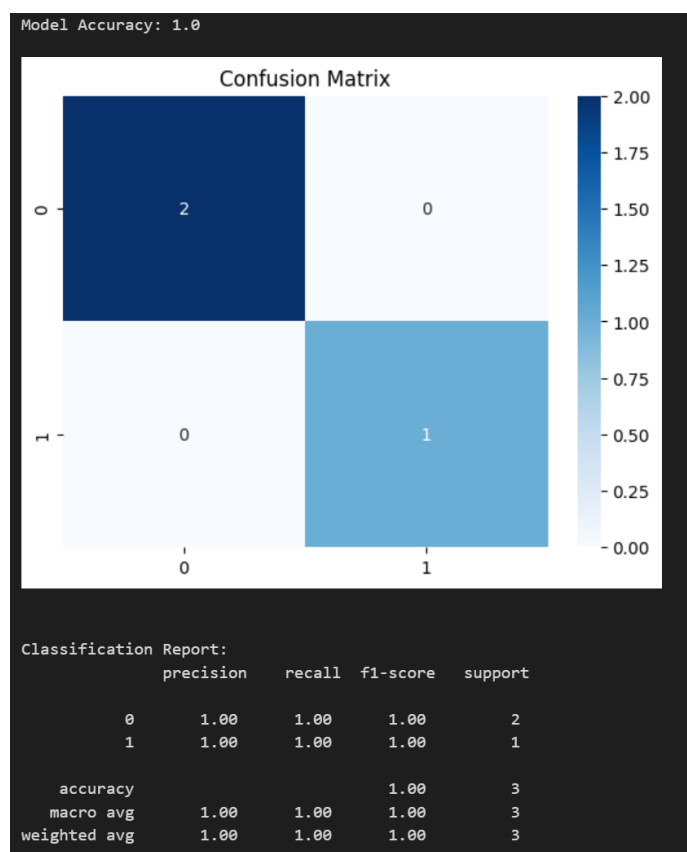
```
plt.savefig("confusion.png")
```

```
plt.show()
```

```
# Displaying classification report
```

```
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

Output:



Testing with the given input case (Y, N, Mild, Y) \rightarrow (1, 0, 1, 1)

test_input = np.array([[1, 0, 1, 1]]) # Example input converted to numerical format

Predicting outcome

prediction = classifier.predict(test_input)

Printing result

print("Predicted Flu Outcome:", "Yes" if prediction[0] == 1 else "No")

Output:

Predicted Flu Outcome: Yes

EXERCISE-6(A)

AIM: IMPLEMENT SUPPORT VECTOR MACHINE FOR A SAMPLE TRAINING DATA SET STORED AS A '.CSV' FILE. COMPUTE THE ACCURACY OF THE CLASSIFIER, CONSIDERING FEW TEST DATA SETS.

DESCRIPTION:

A support vector machine (SVM) is a machine learning algorithm that uses supervised learning models to solve complex classification, regression, and outlier detection problems by performing optimal data transformations that determine boundaries between data points based on predefined classes, labels, or outputs.

SVMs are widely adopted across disciplines such as healthcare, natural language processing, signal processing applications, and speech & image recognition fields.

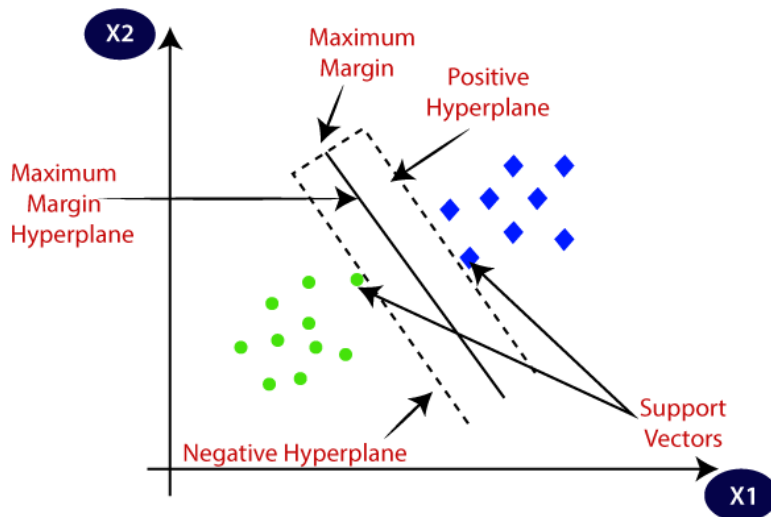
Technically, the primary objective of the SVM algorithm is to identify a hyperplane that distinguishably segregates the data points of different classes. The hyperplane is localized in such a manner that the largest margin separates the classes under consideration.

Types of Support Vector Machine (SVM) Algorithms:

- 1. Linear SVM:** When the data is perfectly linearly separable only then we can use Linear SVM. Perfectly linearly separable means that the *data points can be classified into 2 classes by using a single straight line(if 2D).*
- 2. Non-Linear SVM:** When the data is not linearly separable then we can use Non-Linear SVM, which means when the data points cannot be separated into 2 classes by using a straight line (if 2D) then we use some advanced techniques like kernel tricks to classify them. In most real-world applications we do not find linearly separable Datapoints hence we use kernel trick to solve them.

Important Terms:

- Support Vectors:** These are the points that are closest to the hyperplane. A separating line will be defined with the help of these data points.
- Margin:** it is the distance between the hyperplane and the observations closest to the hyperplane (support vectors). In SVM large margin is considered a good margin. There are two types of margins hard margin and soft margin.



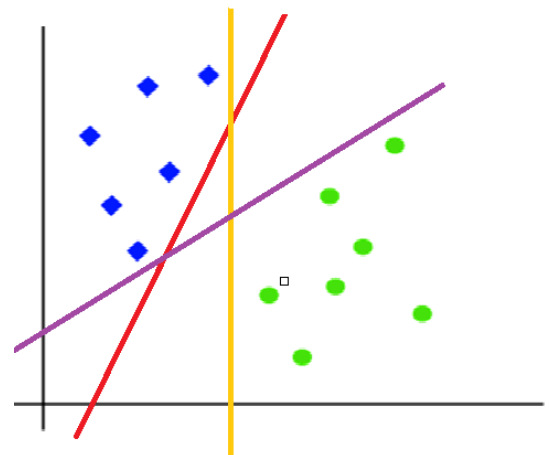
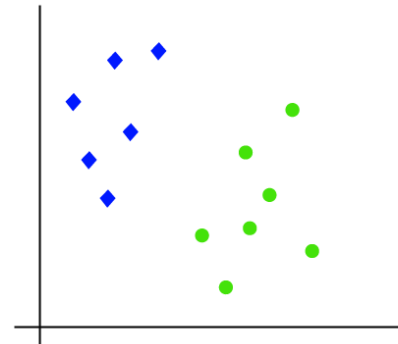
How does Support Vector Machine Works? The working of SVM using an example. Suppose we have a dataset that has two classes (green and blue). We want to classify that the new data point as either blue or green.

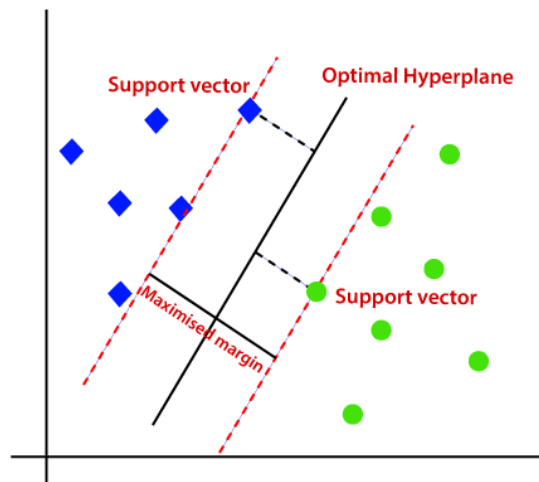
To **classify these points**, we can have **many decision boundaries**, but the question is which is the best and how do we find it?

NOTE: Since we are plotting the data points in a 2-dimensional graph we call this decision **boundary a straight line** but if we have more dimensions, we call this decision boundary **a “hyperplane”**.

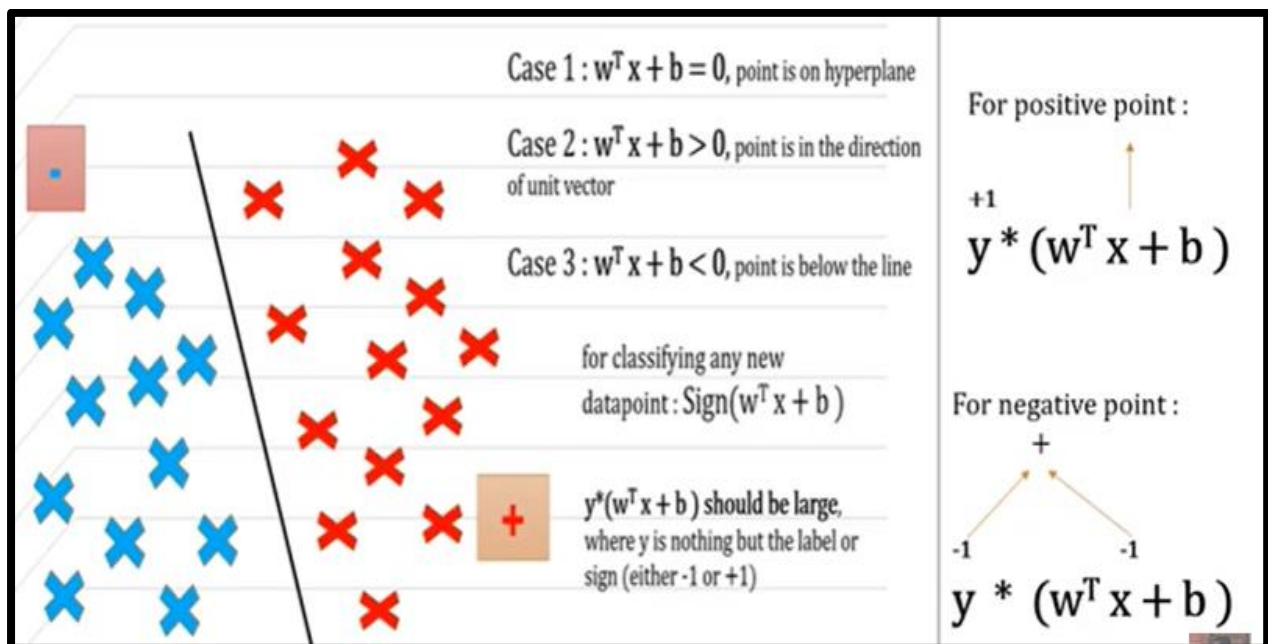
The **best hyperplane** is that plane that has the **maximum distance from both the classes**, and this is the main aim of SVM.

This is done by finding different hyperplanes which classify the labels in the best way then it will choose the **one which is farthest from the data points** or the one **which has a maximum margin**.





Mathematical Representation of Linear SVM:



Implementation of the Support Vector Machine involves below steps:

1. Install the Packages:

(a) **Numpy:** Numpy Python library is used for including any type of mathematical operation in the code. It is the fundamental package for scientific calculation in Python. It also supports to add large, multidimensional arrays and matrices. So, in Python, we can import it as:

import numpy as np

(b) **Matplotlib:** The second library is matplotlib, which is a Python 2D plotting library, and with this library, we need to import a sub-library pyplot. This library is used to plot any type of charts in Python for the code. It will be imported as below:

import matplotlib.pyplot as plt

Here we have used **plt** as a short name for this library.

c) Pandas: The last library is the Pandas library, which is one of the most famous Python libraries and used for **importing and managing the datasets**. It is an **open-source data manipulation and analysis library**. It will be imported as below:

import pandas as pd

Here, we have used **pd** as a short name for this library.

2. Importing the Dataset:

read_csv() function: Now to import the dataset, we will use **read_csv()** function of **pandas library**, which is used to read a csv file and performs various operations on it. Using this function, we can read a csv file locally as well as through an URL. **We can use read_csv function as below:**

For Eg: `dataset = pd.read_csv('Social_Network_Ads.csv')`

3. Separating Independent and Dependent Variables:

In machine learning, it is important to distinguish the matrix of features (independent variables) and dependent variables from dataset.

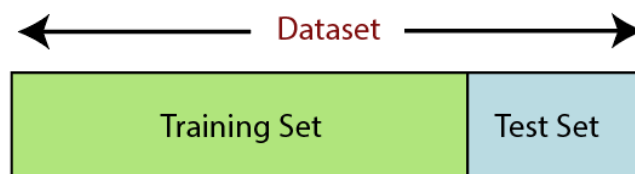
For Eg: In our dataset, there are **Two independent variables** that are **Age** and **Salary**, and **one is a dependent variable** which is **purchased**.

Seperating Independent and Dependent Variable

`X = dataset.iloc[:, :-1].values`

`y = dataset.iloc[:, -1].values`

4. Splitting dataset into training and test set: we divide our dataset into a training set and test set. This is one of the crucial steps of data pre-processing as by doing this, we can enhance the performance of our machine learning model.



Training Set: A subset of dataset to train the machine learning model, and we already know the output.

Test set: A subset of dataset to test the machine learning model, and by using the test set, model predicts the output.

For splitting the dataset, we will use the below lines of code:

```
# training and testing data
```

```
from sklearn.model_selection import train_test_split
```

```
# assign test data size 25%
```

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

Explanation: We set test_size=0.25, which means **25%** of the whole data set will be assigned to the **testing** part, and the remaining **75%** will be used for the model's **training**.

5. Feature Scaling: Feature scaling is the final step of data pre-processing in machine learning. It is a technique to standardize the independent variables of the dataset in a specific range. In feature scaling, we put in the same range and in the same scale so that no any variable dominate the other variable.

```
from sklearn.preprocessing import StandardScaler
```

```
sc = StandardScaler()
```

```
X_train = sc.fit_transform(X_train)
```

```
X_test = sc.transform(X_test)
```

6.Training model using SVM Classifier : Now, let's train our model using the Linear SVM Classifier

```
# import SVM classifier from SK Learn
```

```
from sklearn.svm import SVC
```

```
# create a Linear SVM Classifier
```

```
classifier = SVC(kernel = 'linear', random_state = 0)
```

```
# training the model
```

```
classifier.fit(X_train, y_train)
```

7.Test the model using SVM Classifier :

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

```
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

8. Find the Accuracy of the Model: Accuracy score in machine learning is an **evaluation metric** that **measures the number of correct predictions made by a model** in relation to **the total number of predictions made**. We calculate it by dividing the number of correct predictions by the total number of predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

```
# importing accuracy score
```

```
from sklearn.metrics import accuracy_score
```

```
# printing the accuracy of the model
```

```
print(accuracy_score(y_test,y_pred1))
```

Print the Confusion Matrix and Accuracy Score: The **confusion matrix** is one of the most popular and widely used performance measurement techniques for **classification models**.

Confusion Matrix as the name suggests gives us a **matrix as output** and describes the **complete performance of the model** and it also used to **determine the performance of the classification models for a given set of test data**.

```
from sklearn.metrics import confusion_matrix, accuracy_score
```

```
# passing actual and predicted values
```

```
cm = confusion_matrix(y_test, y_pred)
```

```
print(cm)
```

```
[[66  2]
 [ 8 24]]
```

```
# Print the Accuracy Score
```

```
accuracy_score(y_test, y_pred)
```

9. Visualizing the Clusters:

The last step is to visualize the clusters. To visualize the clusters will use scatter plot using matplotlib. Here we can use the **Colormap object generated** from a list of **colors**. This may be most useful when indexing directly into a **colormap**, but it can also be used to generate special colormaps for ordinary mapping.

Parameters: colorslist, array. Sequence of Matplotlib color specifications (color names or RGB(A) values).

```
from matplotlib.colors import ListedColormap
```

```
X_set, y_set = sc.inverse_transform(X_train), y_train
```

You use **meshgrid()** to convert the 1D vectors representing the axes into 2D arrays. You can then use those arrays in place of the x and y variables in the mathematical equation.

```
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop = X_set[:, 0].max() + 10, step = 0.25),  
np.arange(start = X_set[:, 1].min() - 1000, stop = X_set[:, 1].max() + 1000, step = 0.25))
```

The **contourf()** function in pyplot module of matplotlib library is used to **plot contours**. But contourf **draw filled contours**, while contourf draws **contour lines**.

```
plt.contourf(X1, X2, classifier.predict(sc.transform(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())
```

PROGRAM:

importing the libraries

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

importing the dataset

```
dataset = pd.read_csv('Social_Network_Ads.csv')
```

Seperating Independent and Dependent Variable

```
X = dataset.iloc[:, :-1].values  
y = dataset.iloc[:, -1].values
```

Splitting the Dataset into training and testing

```
from sklearn.model_selection import train_test_split
```

assign test data size 25%

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

Feature Scaling

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

Training the SVM Model on the Training Set

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
classifier.fit(X_train, y_train)
```

Predicting the New Result

```
print(classifier.predict(sc.transform([[30,87000]])))
```

Predicting the Test Result

```
y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1),
y_test.reshape(len(y_test),1)),1))
```

Making the Confusion Matrix

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

#Visualising the Training set results

```
from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop = X_set[:,
0].max() + 10, step = 0.25),
                    np.arange(start = X_set[:, 1].min() - 1000, stop = X_set[:, 1].max() +
1000, step = 0.25))
plt.contourf(X1, X2, classifier.predict(sc.transform(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('SVM (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

#Visualising the Test set results

```
from matplotlib.colors import ListedColormap
```

```
X_set, y_set = sc.inverse_transform(X_test), y_test
```

```
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop = X_set[:,
0].max() + 10, step = 0.25),
```

```
np.arange(start = X_set[:, 1].min() - 1000, stop = X_set[:, 1].max() +
1000, step = 0.25))
```

```
plt.contourf(X1, X2, classifier.predict(sc.transform(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)
```

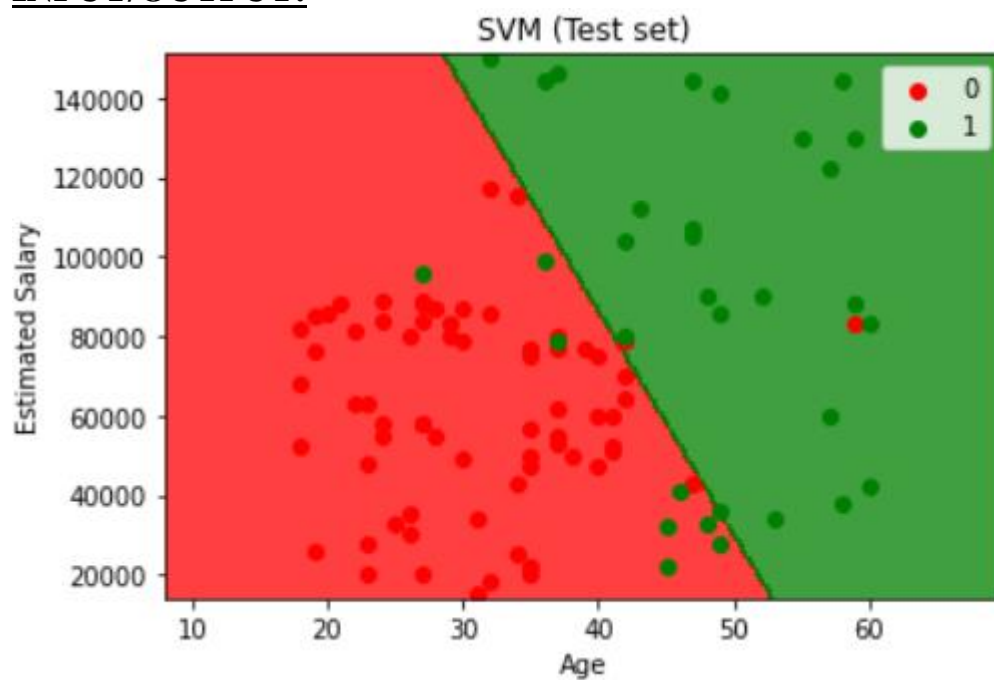
```
plt.title('SVM (Test set)')
```

```
plt.xlabel('Age')
```

```
plt.ylabel('Estimated Salary')
```

```
plt.legend()
```

```
plt.show()
```

INPUT/OUTPUT:**CONCLUSION:** Program is executed successfully without any error.

EXERCISE-6(B)

AIM: To implement Non-Linear SVM using Python and evaluate its performance on a dataset.

DESCRIPTION:**Non-Linear SVM:****1. Import Libraries:**

- numpy, matplotlib, and scikit-learn are used for data manipulation, model creation, and visualization.

2. Load Dataset:

- The Iris dataset is loaded using `datasets.load_iris()`. It has 150 samples and 4 features.

3. Preprocess the Data:

- Feature scaling is done using `StandardScaler` to ensure the features have a mean of 0 and a standard deviation of 1. This helps the SVM model to perform better.

4. Split Data:

- The dataset is split into training and testing sets (70% training, 30% testing) using `train_test_split()`.

5. Train Non-Linear SVM:

- A non-linear kernel (rbf - Radial Basis Function) is used to create a model that can handle non-linearly separable data.
- The C parameter controls the regularization, and gamma defines the influence of individual training points.

6. Evaluate the Model:

- The model's performance is evaluated on the test set using `classification_report()` and `confusion_matrix()` to check accuracy, precision, recall, and F1-score.

7. Visualize the Results:

- PCA (Principal Component Analysis) is used to reduce the feature space from 4 dimensions to 2 so that we can visualize the decision boundaries in 2D.
- The decision boundaries are plotted using a mesh grid.

PROGRAM:

```
# Import Libraries
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
from sklearn import datasets
```

```
from sklearn.svm import SVC
```

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.decomposition import PCA
```

```
from sklearn.metrics import classification_report, confusion_matrix
```

```
# Load Dataset
```

```
iris = datasets.load_iris()
```

```
X = iris.data
```

```
y = iris.target
```

```
# Preprocess the Data
```

```
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)
```

```
# Split Data into Training and Testing Sets
```



```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3,  
random_state=42)
```

```
# Train Non-Linear SVM Model
```

```
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale')
```

```
svm_model.fit(X_train, y_train)
```

```
# Evaluate the Model
```

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

```
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
# PCA for Visualization
```

```
pca = PCA(n_components=2)
```

```
X_pca = pca.fit_transform(X_scaled)
```

```
svm_model_pca = SVC(kernel='rbf', C=1.0, gamma='scale')
```

```
svm_model_pca.fit(X_pca, y)
```

```
# Plot Decision Boundaries
```

```
h = 0.02
```

```
x_min, x_max = X_pca[:, 0].min() - 1, X_pca[:, 0].max() + 1
```

```
y_min, y_max = X_pca[:, 1].min() - 1, X_pca[:, 1].max() + 1
```

```
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
```

```
                    np.arange(y_min, y_max, h))
```

```
Z = svm_model_pca.predict(np.c_[xx.ravel(), yy.ravel()])
```

```
Z = Z.reshape(xx.shape)
```

```
plt.contourf(xx, yy, Z, alpha=0.75, cmap=plt.cm.coolwarm)

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, edgecolors='k', marker='o',
            cmap=plt.cm.coolwarm)

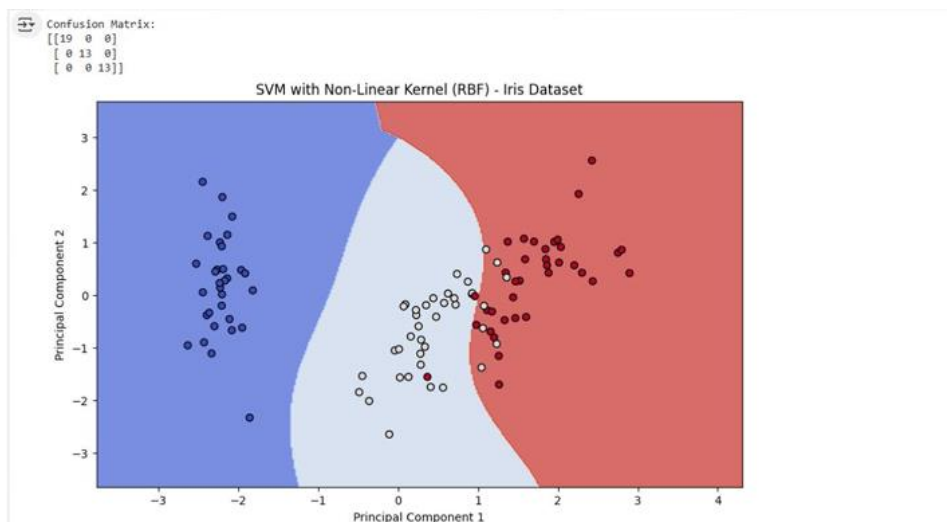
plt.title('SVM with Non-Linear Kernel (RBF) - Iris Dataset')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.show()
```

OUTPUT:



CONCLUSION:

- The Non-Linear SVM model was successfully implemented using the rbf kernel.
- The model was evaluated using classification metrics and visualized using PCA.
- The model handled non-linearly separable data effectively.

EXERCISE-7

AIM: IMPLEMENT K-NEAREST NEIGHBOURS CLASSIFICATION USING PYTHON.

DESCRIPTION:

K- Nearest Neighbours Algorithm (KNN) is one of the **supervised ML 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 neighbouring data points**. This gives a competitive result.

KNN Algorithm:

- 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 taken 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 and outperforms SVM in this case.
- **KNN reducing overfitting** 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**.

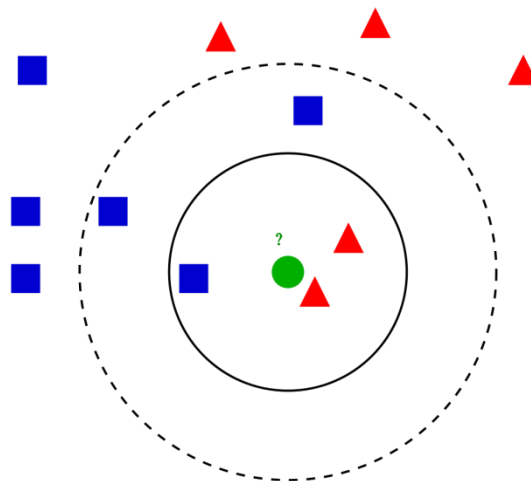
Let's dive deep into the different steps of K-NN for classifying a new data point

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 (which we discuss later)

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.



Implementation of the K- Nearest Neighbour Algorithm involves below steps:

4. Install the Packages:

(b) Numpy: Numpy Python library is used for including any type of **mathematical operation in the code**. It is the fundamental package for scientific calculation in Python. It also supports to **add large, multidimensional arrays and matrices**. So, in Python, we can import it as:

import numpy as np

(b) Matplotlib: The second library is matplotlib, which is a **Python 2D plotting library**, and with this library, we need to import a sub-library pyplot. This library is **used to plot any type of charts in Python** for the code. **It will be imported as below:**

import matplotlib.pyplot as plt

Here we have used **plt** as a short name for this library.

(c) Pandas: The last library is the Pandas library, which is one of the most famous Python libraries and used for **importing and managing the datasets**. It is an **open-source data manipulation and analysis library**. It will be imported as below:

import pandas as pd

Here, we have used **pd** as a short name for this library.

(d) Sklearn: . It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python.

import sklearn

5. Importing the Dataset:

read_csv() function: Now to import the dataset, we will use `read_csv()` function of **pandas library**, which is used to read a csv file and performs various operations on it. Using this function, we can read a csv file locally as well as through an URL. **We can use `read_csv` function as below:**

For Eg: `dataset = pd.read_csv('Social_Network_Ads.csv')`

3. Separating Independent and Dependent Variables:

In machine learning, it is important to distinguish the matrix of features (independent variables) and dependent variables from dataset.

For Eg:

Separating Independent and Dependent Variable

```
x = dataset.iloc[:, [1, 2, 3]].values
```

```
y = dataset.iloc[:, -1].values
```

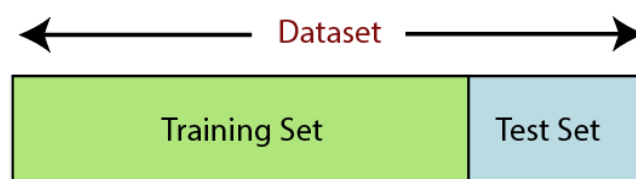
Since our **dataset containing character variables** we have to encode it using **LabelEncoder**

```
from sklearn.preprocessing import LabelEncoder
```

```
le = LabelEncoder()
```

```
x[:,0] = le.fit_transform(x[:,0])
```

4. Splitting dataset into training and test set: we divide our dataset into a training set and test set. This is one of the crucial steps of data pre-processing as by doing this, we can enhance the performance of our machine learning model.



Training Set: A subset of dataset to train the machine learning model, and we already know the output.

Test set: A subset of dataset to test the machine learning model, and by using the test set, model predicts the output.

```
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)
```

Explanation: We are performing a train test split on the dataset. We are providing the test size as 0.20, that means our training sample contains 320 training set and test sample contains 80 test set

5.Training model using KNN Classifier : We are using 3 parameters in the model creation. n_neighbors is setting as 5, which means 5 neighbourhood points are required for classifying a given point. The distance metric we are using is **Minkowski**, the equation for it is given below

```
from sklearn.neighbors import KNeighborsClassifier
```

```
classifier = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)
```

```
classifier.fit(X_train, y_train)
```

#Our Model is created, now we have to predict the output for the test set

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

6. Find the Accuracy of the Model: Accuracy score in machine learning is an **evaluation metric** that **measures the number of correct predictions made by a model** in relation to **the total number of predictions made**. We calculate it by dividing the number of correct predictions by the total number of predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

importing classification report

```
from sklearn.metrics import classification_report
```

printing the report

```
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.88	0.90	0.89	58
1	0.71	0.68	0.70	22
accuracy			0.84	80
macro avg	0.80	0.79	0.79	80
weighted avg	0.84	0.84	0.84	80

Print the Confusion Matrix: The **confusion matrix** is one of the most popular and widely used performance measurement techniques for **classification models**.

Confusion Matrix as the name suggests gives us a **matrix as output** and **describes the complete performance of the model** and it also used to **determine the performance of the classification models for a given set of test data**.

		ACTUAL VALUES	
		POSITIVE	NEGATIVE
PREDICTED VALUES	POSITIVE	TP	FP
	NEGATIVE	FN	TN

```
# importing the required modules
```

```
import seaborn as sns
```

```
from sklearn.metrics import confusion_matrix
```

```
# passing actual and predicted values
```

```
cm = confusion_matrix(y_test, y_pred)
```

```
# true write data values in each cell of the matrix
```

```
sns.heatmap(cm,annot=True)
```

```
plt.savefig('confusion.png')
```

Explanation: Heat Maps are most commonly used to display a more generalized view of numeric values as a graphical representation of data where the individual values contained in a matrix are represented as colors.

Print the Classification Report:

```
#evaluate our 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)*100
```

PROGRAM:

```
# importing the libraries
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
import pandas as pd
```

```
import sklearn
```

importing the dataset

```
dataset = pd.read_csv('Social_Network_Ads.csv')
```

Separating Independent and Dependent Variable

```
x = dataset.iloc[:, [1, 2, 3]].values
```

```
y = dataset.iloc[:, -1].values
```

#Since our dataset containing character variables we have to encode it using LabelEncoder

```
from sklearn.preprocessing import LabelEncoder
```

```
le = LabelEncoder()
```

```
x[:,0] = le.fit_transform(x[:,0])
```

#We are performing a train test split on the dataset. We are providing the test size as 0.20,**#that means our training sample contains 320 training set and test sample contains 80 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 = 0.20, random_state = 0)
```

Now we have to create 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)
```

#Our Model is created, now we have to predict the output for the test set

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

importing classification report

```
from sklearn.metrics import classification_report
```

printing the report

```
print(classification_report(y_test, y_pred))
```

importing the required modules

```
import seaborn as sns
```

```
from sklearn.metrics import confusion_matrix
```

passing actual and predicted values

```
cm = confusion_matrix(y_test, y_pred)
```

true write data values in each cell of the matrix

```
sns.heatmap(cm,annot=True)
```

```
plt.savefig('confusion.png')
```


#evaluate our model using the confusion matrix and accuracy score by comparing the predicted and actual test values

```
from sklearn.metrics import confusion_matrix, accuracy_score
```

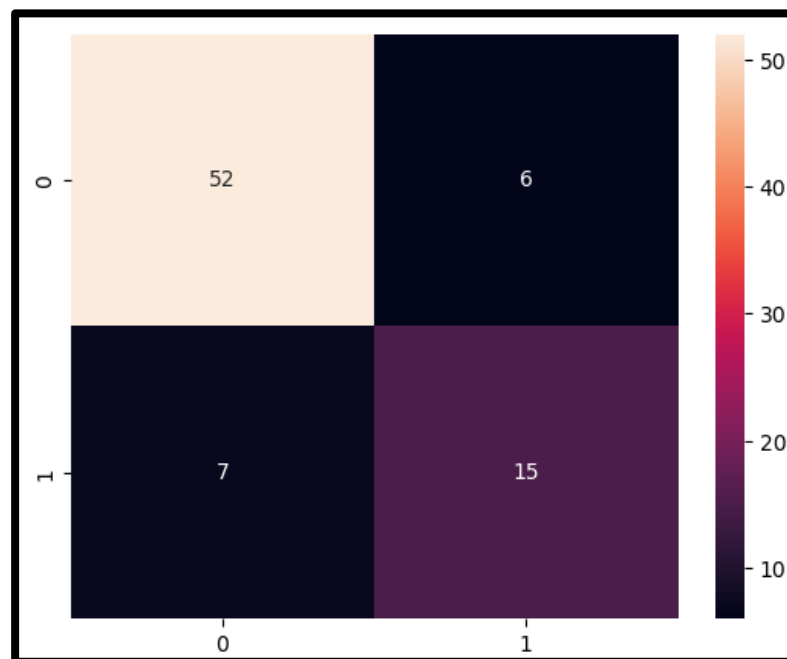
```
cm = confusion_matrix(y_test, y_pred)
```

```
ac = accuracy_score(y_test, y_pred) * 100
```

```
ac # Print Accuracy
```

INPUT / OUTPUT:

Confusion Matrix



Classification Report

	precision	recall	f1-score	support
0	0.88	0.90	0.89	58
1	0.71	0.68	0.70	22
accuracy			0.84	80
macro avg	0.80	0.79	0.79	80
weighted avg	0.84	0.84	0.84	80

CONCLUSION: Program is executed successfully without any error.

EXERCISE-8

AIM: Implementation of K-means Clustering algorithm.

DESCRIPTION:

K-Means Clustering Algorithm:

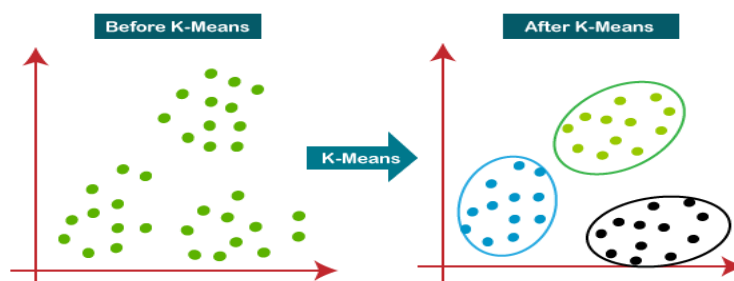
K-Means Clustering is an **unsupervised learning algorithm** that is used to solve the **clustering problems in machine learning**, which groups the **unlabelled dataset into different clusters**. Here **K** defines the number of **pre-defined clusters** that need to be created in the process, as if **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**.

The algorithm takes **the unlabelled dataset as input**, divides the dataset into **k-number of clusters**, and **repeats the process until it does not find the best clusters**. The value of k should be predetermined in this algorithm.

The **k-means clustering algorithm** mainly performs **two tasks**:

- Determines the **best value for K center points** or centroids by an iterative process.
- Assigns **each data point** to its **closest k-center**. Those data points which **are near to the particular k-center create a cluster**.



K-Means Algorithm:

The **working of the K-Means algorithm** is explained in the **below steps**:

Step-1: Select the number K to decide the number of clusters.

Step-2: Select random K points or centroids. (It can be other from the input dataset).

Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.

Step-4: Calculate the variance and place a new centroid of each cluster.

Step-5: Repeat the third steps, which mean reassign each data point to the new closest centroid of each cluster.

Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.

Step-7: The model is ready.

Python Implementation of K-means Clustering Algorithm:

We have a dataset of **Mall_Customers**, 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**, we need to **calculate some patterns**, as it is an **unsupervised method**, so we don't know what to calculate exactly.

The **steps to be followed for the implementation** are given below:

- Data Pre-processing
- Finding the optimal number of clusters using the elbow method
- Training the K-means algorithm on the training dataset
- Visualizing the clusters

1. Data Pre-processing:

(a) Importing Libraries: firstly, we will import the libraries for our model, which is part of data pre-processing. The code is given below:

```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
```

(b) Importing the Dataset: Next, we will import the dataset that we need to use. So here, we are using the **Mall_Customer_data.csv** dataset. It can be imported using the below code:

```
# Importing the dataset

dataset = pd.read_csv('D:\AI TOOLS(PVP-19)\AI TOOLS LAB\WEEK-3(ML using
USL)\Mall_Customers.csv');
```

(c) Extracting Independent Variables: Here we don't need any dependent variable for data pre-processing step as it is a clustering problem, and we have no idea about what to determine. So we will just add a line of code for the matrix of features.

```
x = dataset.iloc[:, [3, 4]].values
```

Step-2: Finding the optimal number of clusters using the elbow method

In the second step, we will try to find the **optimal number of clusters** for our clustering problem. So, as discussed above, here we are going to **use the elbow method** for this purpose. The Elbow method is one of the most popular ways to **find the optimal number of clusters**. This method uses the concept of WCSS value.

As we know, the **elbow method** uses the **WCSS concept to draw the plot by plotting WCSS values on the Y-axis and the number of clusters on the X-axis**. An ideal way to figure out the right number of clusters would be to calculate the **Within-Cluster-Sum-of-Squares (WCSS)**. WCSS is the sum of squares of the distances of each data point in all clusters to their respective centroids. So we are going to calculate the value for WCSS for different k values ranging from 1 to 10.

$$WCSS = \sum_{P_i \text{ in Cluster1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster2}} \text{distance}(P_i, C_2)^2 + \sum_{P_i \text{ in Cluster3}} \text{distance}(P_i, C_3)^2$$

In the above formula of WCSS,

$\sum_{P_i \text{ in Cluster1}} \text{distance}(P_i, C_1)^2$: It is the sum of the square of the distances between each data point and its centroid within a cluster1 and the same for the other two terms.

To measure the **distance between data points and centroid**, we can use any method such as **Euclidean distance or Manhattan distance**.

To find the optimal value of clusters, the elbow method follows the below steps:

- It executes the K-means clustering on a given dataset for different K values (ranges from 1-10).
- For each value of K, calculates the WCSS value.

Plots a curve between calculated WCSS values and the number of clusters K.

#finding optimal number of clusters using the elbow method

```
from sklearn.cluster import KMeans
```

```
wcss_list= [] #Initializing the list for the values of WCSS
```

#Using **for** loop **for** iterations from **1** to **10**.

for i in range(**1**, **11**):

 kmeans = KMeans(n_clusters=i, init='k-means++', random_state= **42**)

 kmeans.fit(x)

 wcss_list.append(kmeans.inertia_)

 mtp.plot(range(**1**, **11**), wcss_list)

 mtp.title('The Elbow Method Graph')

 mtp.xlabel('Number of clusters(k)')

 mtp.ylabel('wcss_list')

 mtp.show()

As we can see in the above code, we have used **the KMeans** class of sklearn. cluster library to form the clusters. Next, we have created the **wcss_list** variable to initialize an empty list, which is used to contain the value of wcss computed for different values of k ranging from 1 to 10.

After that, we have initialized the for loop for the iteration on a different value of k ranging from 1 to 10; since for loop in Python, exclude the outbound limit, so it is taken as 11 to include 10th value.

The rest part of the code is similar as we did in earlier topics, as we have fitted the model on a matrix of features and then plotted the graph between the number of clusters and WCSS.

Step- 3: Training the K-means algorithm on the training dataset:

As we have got the number of clusters, so we can **now train the model on the dataset**.

To train the model, we will use the same two lines of code as we have used in the above section, but here instead of using i, we will use 5, as we know there are 5 clusters that need to be formed. The code is given below:

#training the K-means model on a dataset

kmeans = KMeans(n_clusters=5, init='k-means++', random_state= 42)

y_predict= kmeans.fit_predict(x)

The first line is the same as above for creating the object of KMeans class.

In the second line of code, we have created the dependent variable **y_predict** to train the model.

By executing the above lines of code, we will get **the y_predict variable**. We can now compare the values of y_predict with our original dataset.

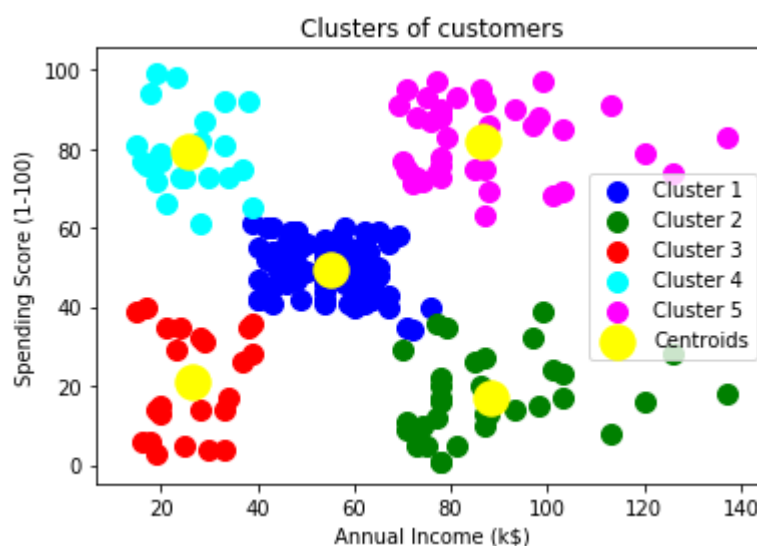
Step-4: Visualizing the Clusters:

The last step is to visualize the clusters. As we have 5 clusters for our model, so we will visualize each cluster one by one. To visualize the clusters will use scatter plot using `mtp.scatter()` function of `matplotlib`.

#visulaizing the clusters

```
mtp.scatter(x[y_predict == 0, 0], x[y_predict == 0, 1], s = 100, c = 'blue', label = 'Cluster 1') #for first cluster
mtp.scatter(x[y_predict == 1, 0], x[y_predict == 1, 1], s = 100, c = 'green', label = 'Cluster 2') #for second cluster
mtp.scatter(x[y_predict == 2, 0], x[y_predict == 2, 1], s = 100, c = 'red', label = 'Cluster 3') #for third cluster
mtp.scatter(x[y_predict == 3, 0], x[y_predict == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4') #for fourth cluster
mtp.scatter(x[y_predict == 4, 0], x[y_predict == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5') #for fifth cluster
mtp.scatter(kmeans.cluster_centers[:, 0], kmeans.cluster_centers[:, 1], s = 300, c = 'yellow', label = 'Centroid')
mtp.title('Clusters of customers')
mtp.xlabel('Annual Income (k$)')
mtp.ylabel('Spending Score (1-100)')
mtp.legend()
mtp.show()
```

In above lines of code, we have written code for each clusters, ranging from 1 to 5. The first coordinate of the `mtp.scatter`, i.e., `x[y_predict == 0, 0]` containing the x value for the showing the matrix of features values, and the `y_predict` is ranging from 0 to 1.



The output image is clearly showing the five different clusters with different colors. The clusters are formed between two parameters of the dataset; **Annual income of customer** and **Spending**. We can change the colors and labels as per the requirement or choice. We can also observe some points from the above patterns, which are given below:

- **Cluster1** shows the customers with average salary and average spending so we can categorize these customers as
- **Cluster2** shows the customer has a high income but low spending, so we can categorize them as careful.
- **Cluster3** shows the low income and also low spending so they can be categorized as sensible.
- **Cluster4** shows the customers with low income with very high spending so they can be categorized as careless.
- **Cluster5** shows the customers with high income and high spending so they can be categorized as target, and these customers can be the most profitable customers for the mall owner.

PROGRAM:

importing libraries

```
import numpy as nm
```

```
import matplotlib.pyplot as mtp
```

```
import pandas as pd
```

Importing the dataset

```
dataset = pd.read_csv('D:\AI TOOLS(PVP-19)\AI TOOLS LAB\WEEK-3( ML  
using USL)\Mall_Customers.csv')
```

```
print(dataset)
```

```
x = dataset.iloc[:, [3, 4]].values
```

```
print(x)
```

```
#finding optimal number of clusters using the elbow method
```

```
from sklearn.cluster import KMeans
```

```
wcss_list= [] #Initializing the list for the values of WCSS
```

```
#Using for loop for iterations from 1 to 10.
```

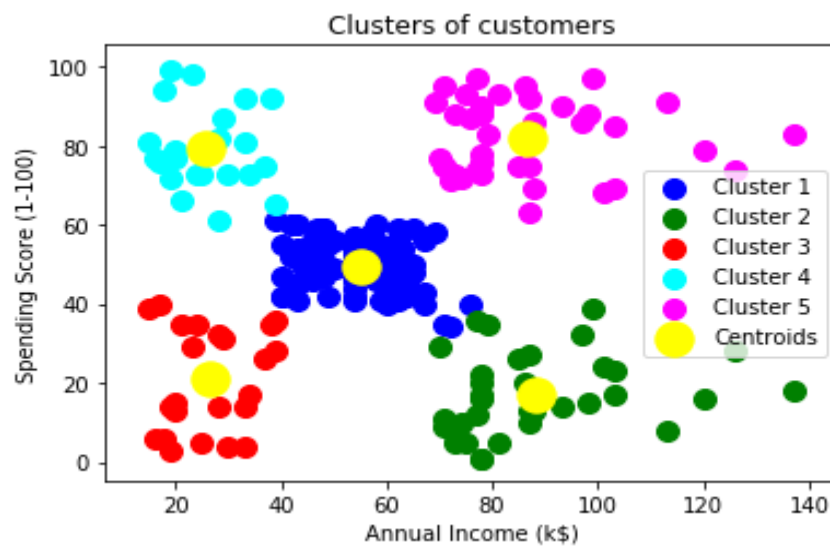
```
for i in range(1, 11):
```

```
kmeans = KMeans(n_clusters=i, init='k-means++', random_state= 42)
kmeans.fit(x)
wcss_list.append(kmeans.inertia_)
mtp.plot(range(1, 11), wcss_list)
mtp.title('The Elbow Method Graph')
mtp.xlabel('Number of clusters(k)')
mtp.ylabel('wcss_list')
mtp.show()
print(wcss_list)
#training the K-means model on a dataset
kmeans = KMeans(n_clusters=5, init='k-means++', random_state= 42)
y_predict= kmeans.fit_predict(x)
print(kmeans)
print(y_predict)
#visualizing the clusters
mtp.scatter(x[y_predict == 0, 0], x[y_predict == 0, 1], s = 100, c = 'blue', label =
'Cluster 1') #for first cluster
mtp.scatter(x[y_predict == 1, 0], x[y_predict == 1, 1], s = 100, c = 'green', label =
'Cluster 2') #for second cluster
mtp.scatter(x[y_predict == 2, 0], x[y_predict == 2, 1], s = 100, c = 'red', label =
'Cluster 3') #for third cluster
mtp.scatter(x[y_predict == 3, 0], x[y_predict == 3, 1], s = 100, c = 'cyan', label =
'Cluster 4') #for fourth cluster
mtp.scatter(x[y_predict == 4, 0], x[y_predict == 4, 1], s = 100, c = 'magenta', label =
'Cluster 5') #for fifth cluster
mtp.scatter(kmeans.cluster_centers_[0], kmeans.cluster_centers_[1], s = 300, c =
'yellow', label = 'Centroid')
mtp.title('Clusters of customers')
mtp.xlabel('Annual Income (k$)')
mtp.ylabel('Spending Score (1-100)')
```



```
mtp.legend()
```

```
mtp.show()
```

INPUT/OUTPUT:**CONCLUSION:** Program is executed successfully without any error.

EXERCISE-9

AIM: Implementation of DBSCAN Clustering Algorithm

DESCRIPTION:

DBSCAN Clustering: DBSCAN is a popular **density-based** data clustering algorithm.

- To cluster data points, this algorithm separates the high-density regions of the data from the low-density areas. Unlike the K-Means algorithm, the best thing with this algorithm is that we don't need to provide the number of clusters required prior.
- DBSCAN algorithm groups points based on distance measurement, usually the *Euclidean distance and the minimum number of points*. An essential property of this algorithm is that it helps us track down the outliers as the points in low-density regions; hence it is not sensitive to outliers as is the case of K-Means clustering.

DBSCAN algorithm works with two parameters.

These parameters are:

Epsilon (Eps): This is the least distance required for two points to be termed as a neighbor. This distance is known as Epsilon (Eps). Thus we consider Eps as a threshold for considering two points as neighbors, i.e., if the distance between two points is utmost Eps, then we consider the two points to be neighbors.

MinPoints: This refers to the minimum number of points needed to construct a cluster. We consider MinPoints as a threshold for considering a cluster as a cluster. A cluster is only recognized if the number of points is greater than or equal to the MinPts. We classify data points into three categories based on the two parameters above. So let us look at these categories.

Types of data points in a DBSCAN clustering

After the DBSCAN clustering is complete, we end up with three types of data points as follows:

1. **Core:** This is a point from which the two parameters above are fully defined, i.e., a point with at least *Minpoints* within the *Eps* distance from itself.
2. **Border:** This is any data point that is not a core point, but it has at least one *Core point* within *Eps* distance from itself.
3. **Noise:** This is a point with less than *Minpoints* within distance *Eps* from itself. Thus, it's not a *Core* or a *Border*.

DBSCAN algorithm:

Step 1: Initially, the algorithms start by selecting a point (x) randomly from the data set and finding all the neighbor points within *Eps* from it. If the number of *Eps-neighbours* is greater than or equal to **MinPoints**, we consider x a core point. Then, with its *Eps-neighbours*, x forms the first cluster. After creating the **first cluster**, we **examine all its member points and find their respective *Eps - neighbours***. If a member has at least ***MinPoints Eps-neighbours***, we expand the initial cluster by adding those ***Eps-neighbours*** to the cluster. This continues until there are no more points to add to this cluster.

Step 2: For any other core point not assigned to cluster, create a new cluster.

Step 3: To the core point cluster, find and assign all points that are recursively connected to it.

Step 4: Iterate through all unattended points in the dataset and assign them to the nearest cluster at *Eps* distance from themselves. If a point does not fit any available clusters, locate it as a noise point.

Python Implementation of DBSCAN Clustering Algorithm:

We have a dataset of **Mall_Customers_dataset.csv**, 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**, we need to **calculate some patterns**, as it is an **unsupervised method**, so we don't know what to calculate exactly.

The **steps to be followed for the implementation** are given below:

- Data Pre-processing
- extracting the above mentioned columns
- Compute data proximity from each other using Nearest Neighbours
- Sorting and plot the distances between the data points
- Implement DBSCAN Model
- Visualizing the clusters

1. Data Pre-processing:

(a) Importing Libraries: firstly, we will import the libraries for our model, which is part of data pre-processing. The code is given below:

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

(b) Importing the Dataset: Next, we will import the dataset that we need to use. So here, we are using the **Mall_Customer_data.csv** dataset. It can be imported using the below code:

```
# Importing the dataset

dataset = pd.read_csv('Mall_Customers_dataset.csv')

#we check if the dataset has any missing values.

data.isnull().any().any()
```

(c) Extracting Independent Variables: Here we don't need any dependent variable for data pre-processing step as it is a clustering problem, and we have no idea about what to determine. So we will just add a line of code for the matrix of features.

```
x= data.loc[:, ['Annual Income (k$)','Spending Score (1-100)']].values
```

Step-2: Before we apply the DBSCAN model, first, we need to obtain its two parameters.

1. **MinPoints:** We can obtain the minimum number of Points to be used to recognize a cluster, as follows:
 - If the dataset has two dimensions, use the min sample per cluster as 4.
 - If the data has more than two dimensions, the min sample per cluster should be:
 - **Min_sample(MinPoints) = 2 * Data dimension**

Since our data is two-dimensional, we shall use the default value of 4 as our MinPoint parameter.

2. **Epsilon (Eps):** To calculate the value of Eps, we shall calculate the distance between each data point to its closest neighbor using the Nearest Neighbours. After that, we sort them and finally plot them. From the plot, we identify the maximum value at the curvature of the graph. This value is our Eps.

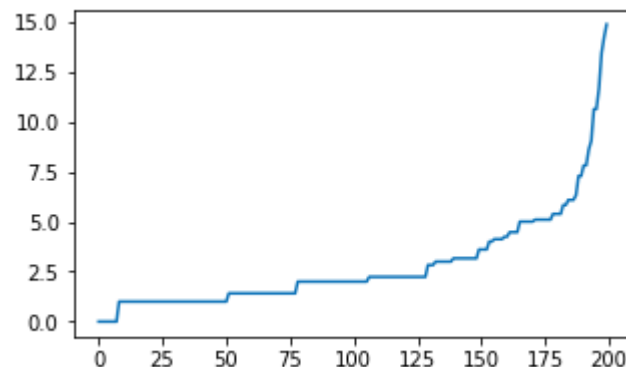
#Compute data proximity from each other using Nearest Neighbours

```
from sklearn.neighbors import NearestNeighbors # importing the library
neighb = NearestNeighbors(n_neighbors=2) # creating an object of the NearestNeighbors class
nbrs=neighb.fit(x) # fitting the data to the object
```

```
distances,indices=nbrs.kneighbors(x) # finding the nearest neighbours
```

Step-3: Sorting and plot the distances between the data points

```
distances = np.sort(distances, axis = 0) # sorting the distances  
distances = distances[:, 1] # taking the second column of the sorted distances  
plt.rcParams['figure.figsize'] = (5,3) # setting the figure size  
plt.plot(distances) # plotting the distances  
plt.show() # showing the plot
```



Step- 4: Implement DBSCAN Model

From the above plot, we note the maximum curvature of the curve is about eight, and thus we picked our **Eps as 8**.

We now have our two parameters as: **MinPoints = 4, Eps = 8**

Now that we have the parameters let us **implement the DBSCAN model**.

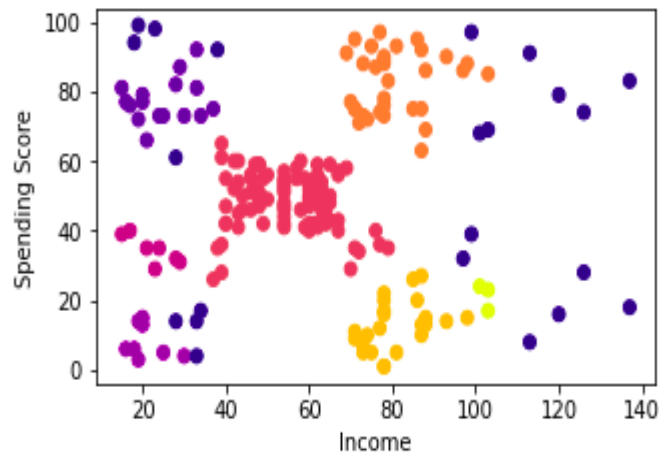
cluster the data into five clusters

```
dbscan = DBSCAN(eps = 8, min_samples = 4).fit(x) # fitting the model  
labels = dbscan.labels_ # getting the labels
```

Step-4: Visualizing the Clusters:

Plot the clusters

```
plt.scatter(x[:, 0], x[:,1], c = labels, cmap= "plasma") # plotting the clusters  
plt.xlabel("Income") # X-axis label  
plt.ylabel("Spending Score") # Y-axis label  
plt.show() # showing the plot
```

**PROGRAM:****# importing libraries**

```
import numpy as nm
```

```
import matplotlib.pyplot as plt
```

```
import pandas as pd
```

Loading the dataset

```
data = pd.read_csv("Mall_Customers_dataset.csv")
```

```
data # Print the Data
```

Checking Head of the Data

```
data.head()
```

#we check if the dataset has any missing values.

```
data.isnull().any().any()
```

#Extract the Annual Income and the Spending Score columns and apply our DBSCAN model to them.

```
x = data.loc[:, ['Annual Income (k$)','Spending Score (1-100)']].values
```

```
print(x.shape)
```

#Compute data proximity from each other using Nearest Neighbours

```
from sklearn.neighbors import NearestNeighbors # importing the library
```

```
neighb = NearestNeighbors(n_neighbors=2)
```

creating an object of the NearestNeighbors class

```
nbrs=neighb.fit(x) # fitting the data to the object
```

```
distances,indices=nbrs.kneighbors(x) # finding the nearest neighbours
```

Sort and plot the distances results

```
distances = np.sort(distances, axis = 0) # sorting the distances
```

```
distances = distances[:, 1] # taking the second column of the sorted distances
```

```
plt.rcParams['figure.figsize'] = (5,3) # setting the figure size
```

```
plt.plot(distances) # plotting the distances
```

```
plt.show() # showing the plot
```

#Implementing the DBSCAN model

```
from sklearn.cluster import DBSCAN
```

cluster the data into five clusters

```
dbscan = DBSCAN(eps = 8, min_samples = 4).fit(x) # fitting the model
```

```
labels = dbscan.labels_ # getting the labels
```

```
labels
```

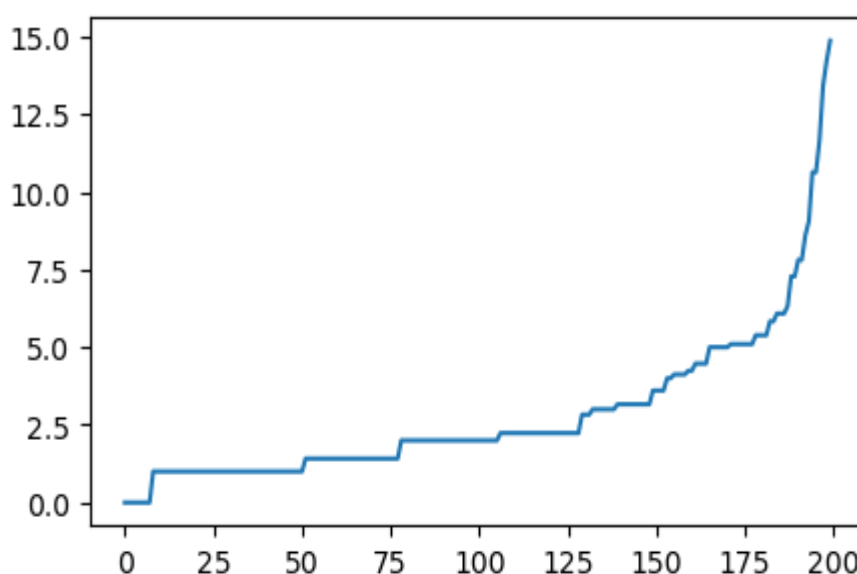
Plot the clusters

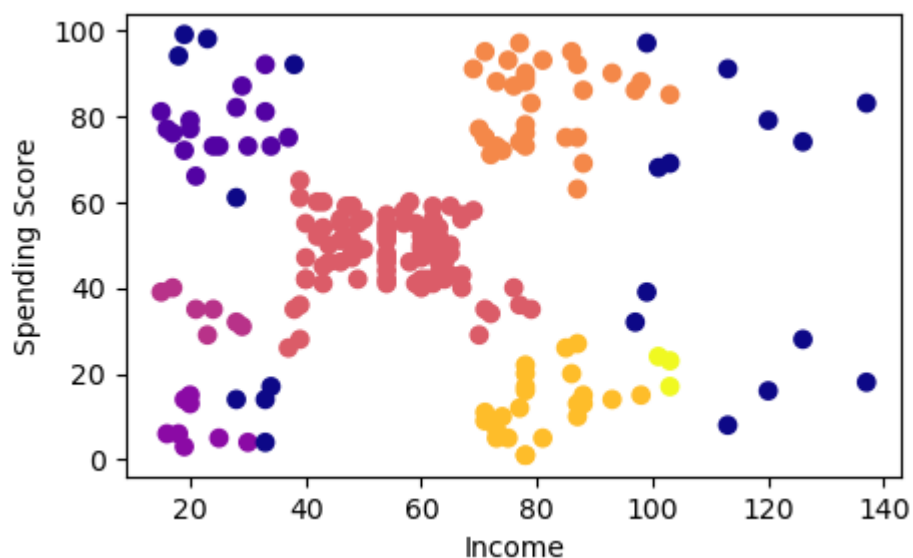
```
plt.scatter(x[:, 0], x[:,1], c = labels, cmap= "plasma") # plotting the clusters
```

```
plt.xlabel("Income") # X-axis label
```

```
plt.ylabel("Spending Score") # Y-axis label
```

```
plt.show() # showing the plot
```

INPUT/OUTPUT:**# Plot the clusters****# Optimal No. of Clusters**



CONCLUSION: Program is executed successfully without any error.

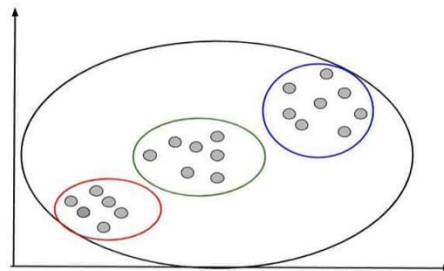
EXERCISE-10

AIM: Implementation of Algorithm

DESCRIPTION:

Hierarchal Clustering: Hierarchical Clustering groups similar objects into one cluster. The final cluster in the Hierarchical cluster combines all clusters into one cluster.

An example of Hierarchical clustering is **Dendrogram**. Hierarchical clustering cluster the **data points based on its similarity**. Hierarchical clustering continues clustering until one single cluster left. you can see in this image. Hierarchical clustering combines all three smaller clusters into one final cluster.



Type of Hierarchical Clustering

Hierarchical Clustering is of 2 types-

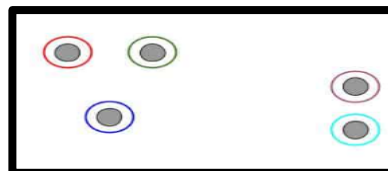
1. **Agglomerative Hierarchical Clustering.**
2. **Divisive Hierarchical Clustering.**

1. Agglomerative Hierarchical Clustering.

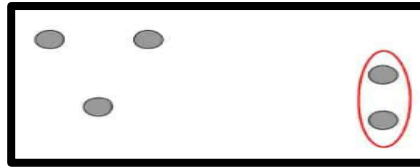
Agglomerative Hierarchical Clustering uses a **bottom-up approach** to form clusters. That means it starts from single data points. Then it clusters the closer data points into one cluster. The same process repeats until it gets one single cluster.

Steps to Perform Hierarchical Clustering:

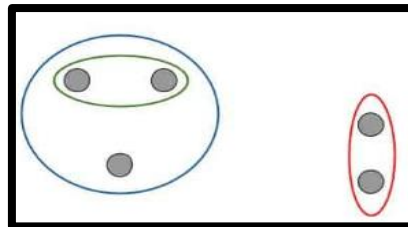
Step 1- Make each data point a single cluster. Suppose that forms n clusters.



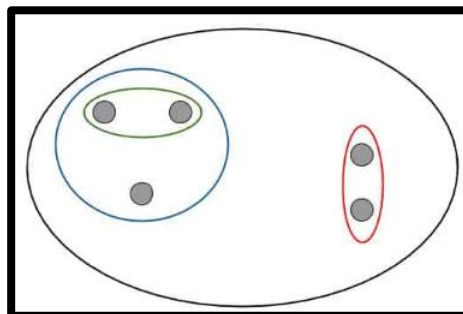
Step 2- Take the 2 closet data points and make them one cluster. Now the total clusters become $n-1$.



Step 3- Take the 2 closet clusters and make them one cluster. Now the total clusters become $n-2$.



Step 4- Repeat Step 3 until only one cluster is left.



Python Implementation of Agglomerative Hierarchical Clustering Algorithm:

We have a dataset of **Mall_Customers_dataset.csv**, 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**, we need to **calculate some patterns**, as it is an **unsupervised method**, so we don't know what to calculate exactly.

The **steps to be followed for the implementation** are given below:

- Data Pre-processing
- Finding the optimal number of clusters using the elbow method
- Training the K-means algorithm on the training dataset
- Visualizing the clusters

1. Data Pre-processing:

(a) Importing Libraries: firstly, we will import the libraries for our model, which is part of data pre-processing. The code is given below:

```
# importing libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

(b) Importing the Dataset: Next, we will import the dataset that we need to use. So here, we are using the **Mall_Customer_data.csv** dataset. It can be imported using the below code:

```
# Importing the dataset

dataset = pd.read_csv('Mall_Customers_dataset.csv')
```

(c) Extracting Independent Variables: Here we don't need any dependent variable for data pre-processing step as it is a clustering problem, and we have no idea about what to determine. So we will just add a line of code for the matrix of features.

```
x = dataset.iloc[:, [3, 4]].values
```

Step-2: We have loaded dataset. Now it's time to find the optimal number of clusters. And for that we need to create a Dendrogram

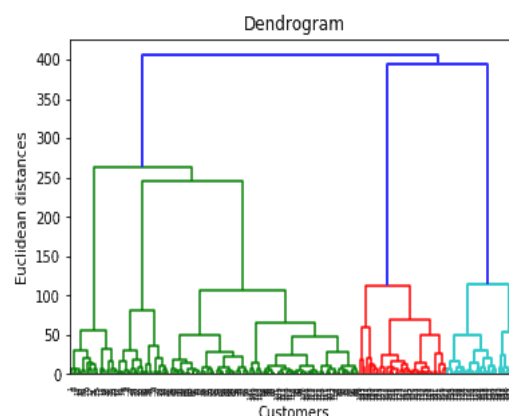
Create Dendrogram to find the Optimal Number of Clusters

```
scipy.cluster.hierarchy as sch dendro = sch.dendrogram(sch.linkage(X, method = 'ward'))
```

Here in the code **“sch”** is the short code for **scipy.cluster.hierarchy.**

“dendro” is the variable name. It may be anything. And **“Dendrogram”** is the function name.

So, after implementing this code, we will get our Dendrogram.



As I discussed that cut the **horizontal line with longest line** that **traverses maximum distance** up and down **without intersecting the merging points**. In that dendrogram, the optimal number of clusters are 5.

Step- 3: Training the Agglomerative Hierarchical Clustering algorithm on the training dataset:

```
from sklearn.cluster import AgglomerativeClustering  
hc = AgglomerativeClustering(n_clusters = 5, affinity = 'euclidean', linkage = 'ward')  
y_hc = hc.fit_predict(X)
```

Step-4: Visualizing the Clusters:

The last step is to visualize the clusters. As we have 5 clusters for our model, so we will visualize each cluster one by one. To visualize the clusters will use scatter plot using **plt.scatter()** function of matplotlib.

#visualizing the clusters

```
plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s = 100, c = 'red', label = 'Cluster 1') plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s = 100, c = 'blue', label = 'Cluster 2') plt.scatter(X[y_hc == 2, 0], X[y_hc == 2, 1], s = 100, c = 'green', label = 'Cluster 3') plt.scatter(X[y_hc == 3, 0], X[y_hc == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4') plt.scatter(X[y_hc == 4, 0], X[y_hc == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5') plt.title('Clusters of customers') plt.xlabel('Annual Income (k$)')  
plt.ylabel('Spending Score (1-100)')  
plt.legend()  
plt.show()
```

In above **lines of code**, we have written code for each clusters, ranging from 1 to 5. The first coordinate of the **plt.scatter**, i.e., **x[y_hc == 0, 0]** containing the x value for the showing the matrix of features values, and the y_predict is ranging from 0 to 1.



The output image is clearly showing the five different clusters with different colors. The clusters are formed between two parameters of the dataset; Annual income of customer and Spending. We can change the colors and labels as per the requirement or choice. We can also observe some points from the above patterns, which are given below:

- Cluster1 shows the customers with average salary and average spending so we can categorize these customers as
- Cluster2 shows the customer has a high income but low spending, so we can categorize them as careful.
- Cluster3 shows the low income and also low spending so they can be categorized as sensible.
- Cluster4 shows the customers with low income with very high spending so they can be categorized as careless.
- Cluster5 shows the customers with high income and high spending so they can be categorized as target, and these customers can be the most profitable customers for the mall owner.

PROGRAM:**# importing libraries**

```
import numpy as nm
```

```
import matplotlib.pyplot as plt
```

```
import pandas as pd
```

Loading the dataset

```
dataset = pd.read_csv('Mall_Customers_dataset.csv')
```

```
print(dataset)
```

```
x = dataset.iloc[:, [3, 4]].values
```

```
print(x)
```

#Create Dendrogram to find the Optimal Number of Clusters

```
import scipy.cluster.hierarchy as sch
```

```
dendro = sch.dendrogram(sch.linkage(X, method = 'ward'))
```

```
plt.title('Dendrogram')
```

```
plt.xlabel('Customers')
```

```
plt.ylabel('Euclidean distances')
```

```
plt.show()
```

#Fitting Agglomerative Hierarchical Clustering to the dataset

```
from sklearn.cluster import AgglomerativeClustering
```

```
hc = AgglomerativeClustering(n_clusters = 5, affinity = 'euclidean', linkage = 'ward')
```

```
y_hc = hc.fit_predict(X)
```

```
y_hc
```

#Visualise the clusters

```
plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
```

```
plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
```

```
plt.scatter(X[y_hc == 2, 0], X[y_hc == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
```

```
plt.scatter(X[y_hc == 3, 0], X[y_hc == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
```

```
plt.scatter(X[y_hc == 4, 0], X[y_hc == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
```

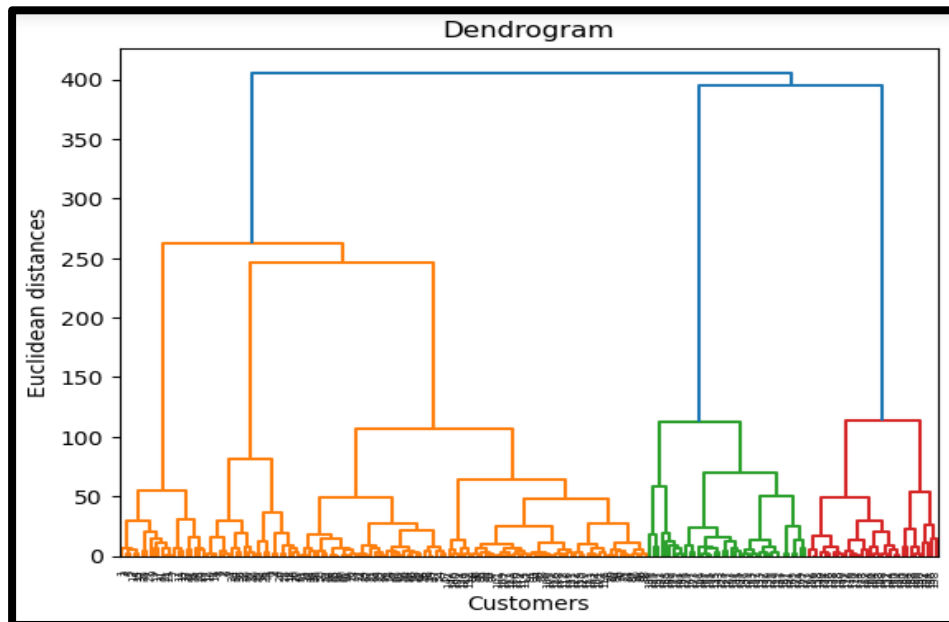
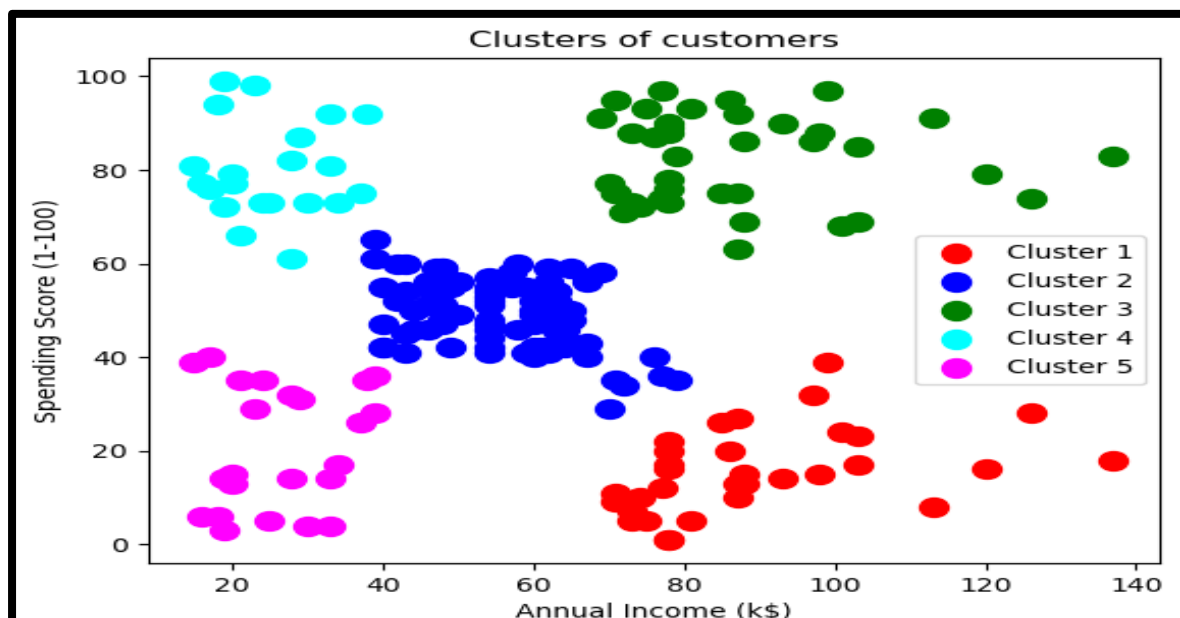
```
plt.title('Clusters of customers')
```

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

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

```
plt.legend()
```

```
plt.show()
```

INPUT/OUTPUT:**# Creating Dendrogram****# Optimal No. of Clusters****CONCLUSION:** Program is executed successfully without any error.

EXERCISE-11

AIM: IMPLEMENT BOOSTING ENSEMBLE METHOD ON A GIVEN DATASET FOR A SAMPLE TRAINING DATA SET STORED AS A '.CSV' FILE. COMPUTE THE ACCURACY OF THE CLASSIFIER.

DESCRIPTION:

Definition: The term '**Boosting**' refers to a **family of algorithms** which converts **weak learner** to **strong learners**.

To convert **weak learner to strong learner**, we'll combine the prediction of **each weak learner** using methods like:

- Using **average/ weighted average**
- considering **prediction has higher vote**

How Boosting Algorithm Works? To find weak rule, we apply base learning (ML) algorithms with a **different distribution**. Each time base learning algorithm is applied, it **generates a new weak prediction rule**. This is an **iterative process**. After much iteration, the boosting algorithm **combines these weak rules into a single strong prediction rule**.

For choosing the right distribution, here are the following steps:

Step 1: The base learner takes all the distributions and **assigns equal weight or attention** to each observation.

Step 2: If there is **any prediction error caused by first base learning algorithm**, then we pay higher attention to observations having prediction error. Then, **we apply the next base learning algorithm**.

Step 3: Iterate Step 2 till the limit of base learning algorithm is reached or **higher accuracy is achieved**.

Finally, it combines the **outputs from weak learner and creates a strong learner** which eventually **improves the prediction power of the model**. Boosting pays higher focus on examples which are **mis-classified or have higher errors by preceding weak rules**.

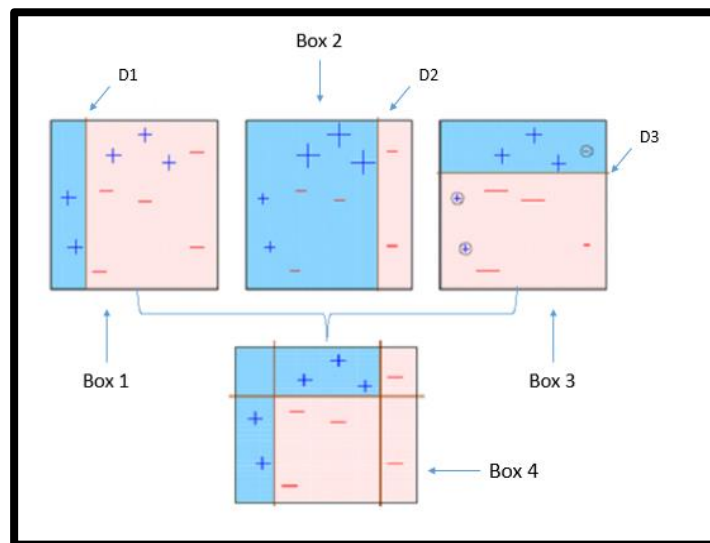
Types of Boosting Algorithms:

Underlying engine used for boosting algorithms can be anything. It can be decision stump, **margin-maximizing classification algorithm etc**. There are many boosting algorithms which use other types of engine such as:

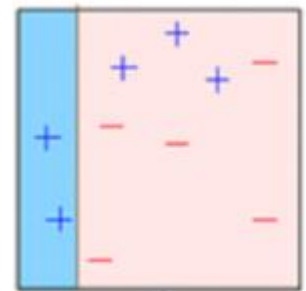
1. AdaBoost (Adaptive Boosting)

2. Gradient Tree Boosting

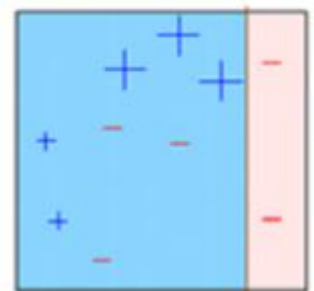
3. XGBoost

1. **AdaBoost Algorithm:**

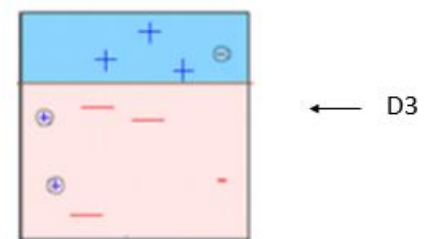
Box 1: You can see that we have assigned equal weights to each data point and applied a decision stump to classify them as + (plus) or - (minus). The decision stump (D1) has generated a vertical line at the left side to classify the data points. We see that, this vertical line has incorrectly predicted three + (plus) as - (minus). In such case, we'll assign higher weights to these three + (plus) and apply another decision stump.



Box 2: Here, you can see that the size of three incorrectly predicted + (plus) is bigger as compared to the rest of the data points. In this case, the second decision stump (D2) will try to predict them correctly. Now, a vertical line (D2) at the right side of this box has classified three mis-classified + (plus) correctly. But again, it has caused mis-classification errors. This time with three - (minus). Again, we will assign higher weight to three - (minus) and apply another decision stump.



Box 3: Here, three - (minus) are given higher weights. A decision stump (D3) is applied to predict these mis-classified observations correctly. This time a horizontal line is generated to classify + (plus) and - (minus) based on the higher weight of mis-classified observations.



Box 4: Here, we have combined D1, D2 and D3 to form a strong prediction having complex rule as compared to individual weak learner. You can see that this algorithm has classified these observation quite well as compared to any of individual weak learner.



AdaBoost (Adaptive Boosting) : It works on similar method as discussed above.

It fits a sequence of **weak learners on different weighted training data**. It starts by predicting original data set and gives equal weight to each observation. If prediction is incorrect using the first learner, then it gives higher weight to observation which have been predicted incorrectly. Being an iterative process, it continues to add learner(s) until a limit is reached in the number of models or accuracy.

Mostly, we use **decision stamps with AdaBoost**. But, we can use any machine learning algorithms as base learner if it accepts weight on training data set. We can use **AdaBoost algorithms for both classification and regression problem**.

Implementation of the Boosting involves below steps:

6. Install the Packages:

(c) **Numpy:** Numpy Python library is used for including any type of **mathematical operation in the code**. It is the fundamental package for scientific calculation in Python. It also supports to **add large, multidimensional arrays and matrices**. So, in Python, we can import it as:

import numpy as np

Pandas: The last library is the Pandas library, which is one of the most famous Python libraries and used for **importing and managing the datasets**. It is an **open-source data manipulation and analysis library**. It will be imported as below:

import pandas as pd

Here, we have used **pd** as a short name for this library.

7. Importing the Dataset:

read_csv() function: Now to import the dataset, we will use **read_csv() function** of **pandas library**, which is used to read a csv file and performs various operations on it. Using this function, we can read a csv file locally as well as through an URL. **We can use read_csv function as below:**

For Eg: `dataset = pd.read_csv('iris.csv')`

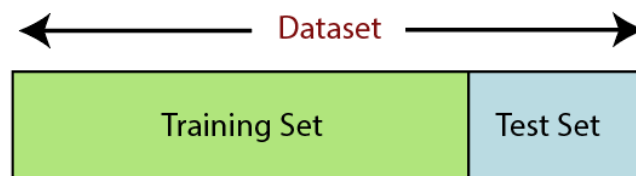
8. Separating Independent and Dependent Variables:

In machine learning, it is important to distinguish the matrix of features (independent variables) and dependent variables from dataset.

Separating Independent and Dependent Variable

```
X = data.iloc[:, :-1]
y = data.iloc[:, -1]
print("Shape of X is %s and shape of y is %s" % (X.shape, y.shape))
# Number of unique classes in the data set
total_classes = y.nunique()
print("Number of unique species in dataset are: ", total_classes)
distribution = y.value_counts()
print(distribution)
```

4. Splitting dataset into training and test set: we divide our dataset into a training set and test set. This is one of the crucial steps of data pre-processing as by doing this, we can enhance the performance of our machine learning model.



Training Set: A subset of dataset to train the machine learning model, and we already know the output.

Test set: A subset of dataset to test the machine learning model, and by using the test set, model predicts the output.

For splitting the dataset, we will use the below lines of code:

```
# training and testing data
from sklearn.model_selection import train_test_split
# assign test data size 25%
X_train, X_val, Y_train, Y_val = train_test_split(X, y, test_size=0.25, random_state=28)
```

Explanation: We set test_size=0.25, which means **25%** of the whole data set will be assigned to the testing part, and the remaining **75%** will be used for the model's **training**.

5. Creating AdaBoost Classifier : Now, let's train our model using the Linear SVM Classifier

```
from sklearn.ensemble import AdaBoostClassifier
adb = AdaBoostClassifier()
adb_model = adb.fit(X_train, Y_train)
```

8. Find the Accuracy of the Model: Accuracy score in machine learning is an **evaluation metric** that **measures the number of correct predictions made by a model** in relation to the **total**

number of predictions made. We calculate it by dividing the number of correct predictions by the total number of predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

```
print("The accuracy of the model on validation set is", adb_model.score(X_val,Y_val))
```

PROGRAM:

importing the libraries

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
import warnings
warnings.filterwarnings("ignore")
```

importing the dataset

Reading the dataset from the csv file

separator is a vertical line, as seen in the dataset

```
data = pd.read_csv("Iris.csv")
```

Printing the shape of the dataset

```
print(data.shape)
```

Separating Independent and Dependent Variable

```
#data = data.drop('Id',axis=1)
```

```
X = data.iloc[:, :-1]
```

```
y = data.iloc[:, -1]
```

```
print("Shape of X is %s and shape of y is %s"%(X.shape,y.shape))
```

```
total_classes = y.nunique()
```

```
print("Number of unique species in dataset are: ",total_classes)
```

```
distribution = y.value_counts()
```

```
print(distribution)
```

Splitting the Dataset into training and testing

```
from sklearn.model_selection import train_test_split
```

assign test data size 25%

```
X_train, X_val, Y_train, Y_val = train_test_split( X, y, test_size=0.25,  
random_state=28)
```

Creating Adaboost Classifier

```
adb = AdaBoostClassifier()  
adb_model = adb.fit(X_train,Y_train)
```

Print the Accuracy

```
print("The accuracy of the model on validation set is", adb_model.score(X_val,Y_val))
```

INPUT/OUTPUT:

```
The accuracy of the model on validation set is 0.9210526315789473
```

CONCLUSION: Program is executed successfully without any error.