EX.NO.:1
DATE:

IMPLEMENT BREADTH FIRST SEARCH

AIM:

To implement the Breadth First Search.

ALGORITHM:

1. Start the program.

2.Create a queue data structure and a visited set or array.
Enqueue the starting node into the queue and mark it as visited.

3. While the queue is not empty:
   a. Dequeue a node from the queue.
   b. Process the dequeued node (e.g., print it or perform some operation).
   c. Enqueue all adjacent nodes of the dequeued node that have not been visited and mark them as visited.

4. Repeat step 3 until the queue is empty.

5.Stop the program.

PROGRAM:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = [] # List for visited nodes.
queue = []     #Initialize a queue
def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)
  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

OUTPUT:

```
IDLE Shell 3.12.1
File  Edit  Shell  Debug  Options  Window  Help
    Python 3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more information.
>>>
    = RESTART: E:\python\1.py
    Following is the Breadth-First Search
    5 3 7 2 4 8
>>>
```

EX.NO.:2

DATE:

IMPLEMENT DEPTH FIRST SEARCH

AIM:

To implement the Depth first search.

ALGORITHM :

1. Start the program.
2. Create a set or array to keep track of visited nodes.
3. Choose a starting node and mark it as visited.
4. Recursively explore each unvisited neighbor of the current node.
5. Repeat step 3 until all reachable nodes are visited.
6. Stop the program.

PROGRAM:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set() # Set to keep track of visited nodes of graph.
def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

OUTPUT:

IDLE Shell 3.12.1

File Edit Shell Debug Options Window Help

```
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:/python/2.py
Following is the Depth-First Search
5
3
2
4
8
7
>>>
```

EX.NO.:3 :

DATE:

ANALYSIS OF BREADTH FIRST AND DEPTH FIRST SEARCH IN TERMS OF TIME AND SPACE

AIM: To implement the analysis of breadth-first and depth-first search in terms of time and space.

ALGORITHM:

Breadth First Search — Time and Space Complexity

1. Start BFS, the time complexity is also determined by the **number of vertices (nodes) and edges** in the graph.

2. BFS **visits all the vertices at each level of the graph** before moving to the next level.

3. In the **worst case** (as we always talk about the **upper bound** in Big O notation), BFS may **visit all vertices and edges** in the graph.

4. Therefore, the time complexity of BFS is **O(V + E)**, where V represents the number of vertices and E represents the number of edges in the graph.

5. The space complexity of BFS depends on the **maximum number of vertices in the queue at any given time**.

6. In the worst case, if the graph is complete, all vertices at each level will be stored in the queue.

7. Therefore, the space complexity of BFS is **O(V)**, where V represents the number of vertices in the graph.

Depth First Search — Time and Space Complexity

1. In DFS, the time complexity is determined by the number of vertices (nodes) and edges in the graph. For each vertex, DFS visits **all its adjacent vertices** recursively.

2. In the **worst case**, DFS may visit **all vertices and edges** in the graph.

3. Therefore, the time complexity of DFS is **O(V + E)**, where V represents the number of vertices and E represents the number of edges in the graph.

4. The space complexity of DFS depends on the **maximum depth of recursion**, if the graph is a straight line or a long path, the **DFS recursion can go as deep as the number of vertices**.

5. Therefore, the space complexity of DFS is **O(V)**, where V represents the number of vertices in the graph.

6. Stop the program

PROGRAM:

```
from collections import deque
import time
class Graph:
    def __init__(self):
        self.graph = {}
    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)
    def bfs(self, start):
        visited = set()
        queue = deque([start])
        visited.add(start)
        while queue:
            node = queue.popleft()
            for neighbor in self.graph.get(node, []):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
    def dfs_util(self, node, visited):
        visited.add(node)
```

```
        for neighbor in self.graph.get(node, []):
            if neighbor not in visited:
                self.dfs_util(neighbor, visited)

    def dfs(self, start):
        visited = set()
        self.dfs_util(start, visited)

def analyze_bfs_dfs(graph, start_node):
    # Breadth-First Search
    bfs_start_time = time.time()
    graph.bfs(start_node)
    bfs_end_time = time.time()
    bfs_time = bfs_end_time - bfs_start_time

    # Depth-First Search
    dfs_start_time = time.time()
    graph.dfs(start_node)
    dfs_end_time = time.time()
    dfs_time = dfs_end_time - dfs_start_time

    # Analyzing space complexity
    bfs_space = len(graph.graph)
    dfs_space = len(graph.graph)
```

```
    return bfs_time, dfs_time, bfs_space, dfs_space

if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.add_edge(3, 3)
    start_node = 0
    bfs_time, dfs_time, bfs_space, dfs_space = analyze_bfs_dfs(g, start_node)
    print("BFS Time:", bfs_time)
    print("DFS Time:", dfs_time)
    print("BFS Space:", bfs_space)
    print("DFS Space:", dfs_space)
```

OUTPUT:

Output   Clear

```
BFS Time: 1.5974044799804688e-05
DFS Time: 3.5762786865234375e-06
BFS Space: 4
DFS Space: 4

=== Code Execution Successful ===
```

EX.NO:4
DATE:

IMPLEMENT AND COMPARE GREEDY AND A * ALGORITHM

AIM: To implement and compare Greedy and A * Algorithm

ALGORITHM:

Greedy Algorithm:

1. Start the initialization.
2. Iteration : At each step, select the best immediate option without considering the future consequences.
3. Evaluate the heuristic function to determine the “best” option.
4. Update the current state to the selected option.
5. Termination :Stop when reaching the goal state or no feasible options are available.

A* Algorithm:

1. Start the initialization.
2. Iteration :Maintain a priority queue (open list) of states to explore, sorted by their estimated total cost.
3. Calculate the cost function f(n)=g(n)+h(n), where g(n) is the cost from the initial state to node n, and h(n) is the heuristic estimates from node n to the goal.
4. Expand the node with the lowest f-value.
5. Update the cost and parent pointers for each successor node.
6. When reaching the goal state or when the open list is empty.
7. Stop the program.

PROGRAM:

```
import heapq
class Graph:
    def __init__(self):
        self.graph = {}
    def add_edge(self, u, v, w):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append((v, w))
    def greedy_search(self, start, goal):
        visited = set()
        queue = [(0, start)]
        while queue:
            cost, node = heapq.heappop(queue)
            if node == goal:
                return cost
            if node not in visited:
                visited.add(node)
                for neighbor, weight in self.graph.get(node, []):
                    heapq.heappush(queue, (weight, neighbor))
        return float('inf')
```

```
    def astar_search(self, start, goal, heuristic):
        visited = set()
        queue = [(0 + heuristic[start], start)]
        while queue:
            cost, node = heapq.heappop(queue)
            if node == goal:
                return cost
            if node not in visited:
                visited.add(node)
                for neighbor, weight in self.graph.get(node, []):
                    heapq.heappush(queue, (cost + weight + heuristic[neighbor],
neighbor))
        return float('inf')
def main():
    g = Graph()
    g.add_edge('S', 'A', 1)
    g.add_edge('S', 'B', 5)
    g.add_edge('A', 'C', 3)
    g.add_edge('B', 'C', 2)
    g.add_edge('B', 'D', 4)
    g.add_edge('C', 'D', 1)
```

```
    g.add_edge('C', 'G', 5)


    g.add_edge('D', 'G', 2)
    start_node = 'S'
    goal_node = 'G'
    heuristic = {'S': 7, 'A': 6, 'B': 2, 'C': 4, 'D': 2, 'G': 0}
    # Greedy Search
    greedy_cost = g.greedy_search(start_node, goal_node)
    print("Greedy Search Cost:", greedy_cost)
    # A* Search
    astar_cost = g.astar_search(start_node, goal_node, heuristic)
    print("A* Search Cost:", astar_cost)


    # Output comparison
    if greedy_cost == astar_cost:
        print("Both algorithms found the same cost.")
    elif greedy_cost < astar_cost:
        print("Greedy Search found a lower cost.")
    else:
        print("A* Search found a lower cost.")
if __name__ == "__main__":
    main()
```

OUTPUT:

IDLE Shell 3.12.1

File Edit Shell Debug Options Window Help

```
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\python\4.py
Greedy Search Cost: 2
A* Search Cost: 22
Greedy Search found a lower cost.
>>>
```

SUPERVISED LEARNING

EX.NO:5

DATE:

IMPLEMENT THE NON PARAMETRIC LOCALLY WEIGHTED REGRESSION ALGORITHM IN ORDER TO FIT DATA POINTS SELECT APPROPRIATE DATA SET FOR YOUR EXPERIMENT AND DRAW GRAPHS

AIM: To implement the Non parametric locally weighted regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

ALGORITHM:

1. Read the Given data Sample to X and the curve (linear or non linear) to Y.
2. Set the value for Smoothening parameter or Free parameter say τ.
3. Set the bias /Point of interest set x0 which is a subset of X
4. Determine the weight matrix using :

$$w(x, x_o) = e^{-\frac{(x-x_o)^2}{2\tau^2}}$$

5. Determine the value of model term parameter β using:

$$\hat{\beta}(x_o) = (X^T W X)^{-1} X^T W y$$

6.Prediction = x0*β

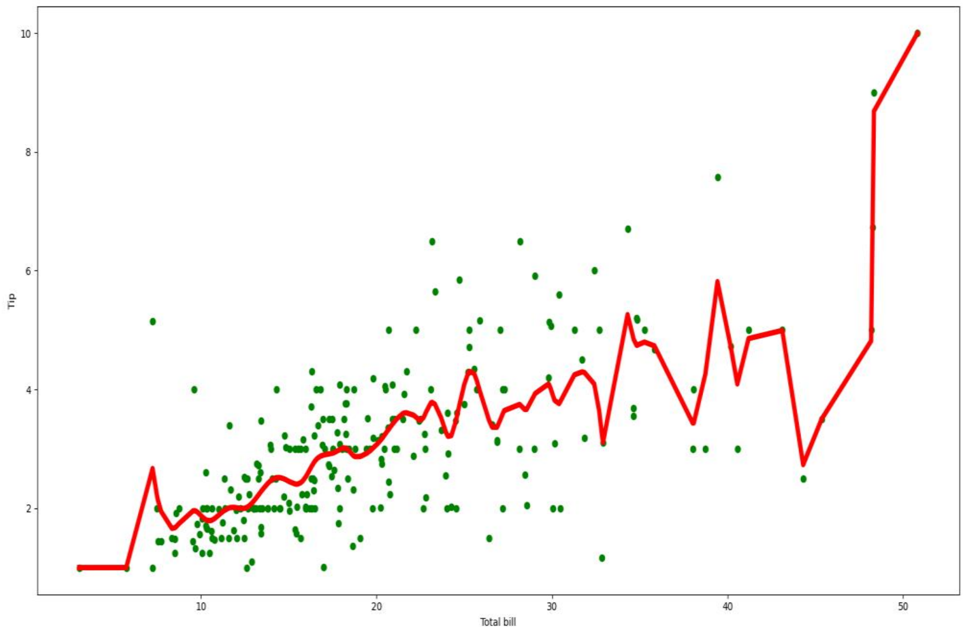
7.Stop the program.

PROGRAM

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
def kernel(point, xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights
def localWeight(point, xmat, ymat, k):
    wei = kernel(point,xmat,k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W
def localWeightRegression(xmat, ymat, k):
    m,n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
```

```
    return ypred
# load data points
data = pd.read_csv('10-dataset.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)
#preparing and add 1 in bill
mbill = np.mat(bill)
mtip = np.mat(tip)
m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T))
#set k here
ypred = localWeightRegression(X,mtip,0.5)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0]
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(bill,tip, color='green')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show();
```

OUTPUT



EX.No:6
DATE :

WRITE A PROGRAM TO DEMONSTRATE THE WORKING OF THE DECISION TREE BASED ALGORITHM

AIM: To implement demonstrate the working of the decision tree based algorithm.

ALGORITHM:

1. Start the program
2. Begin the tree with the root node, says S, which contains the complete dataset.
3. Find the best attribute in the dataset using **Attribute Selection Measure (ASM).**
4. Divide the S into subsets that contains possible values for the best attributes.
5. Generate the decision tree node, which contains the best attribute.
6. Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.
7. Stop the program.

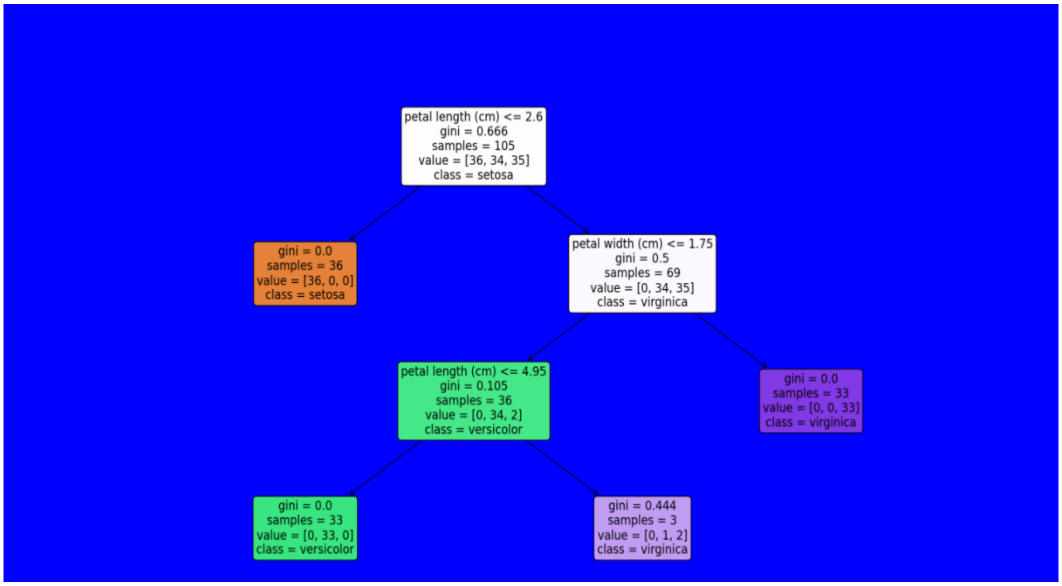
PROGRAM

```
# Importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn import tree
# Loading the dataset
iris = load_iris()
#converting the data to a pandas dataframe
data = pd.DataFrame(data = iris.data, columns = iris.feature_names)
#creating a separate column for the target variable of iris dataset
data['Species'] = iris.target
#replacing the categories of target variable with the actual names of the species
target = np.unique(iris.target)
target_n = np.unique(iris.target_names)
target_dict = dict(zip(target, target_n))
data['Species'] = data['Species'].replace(target_dict)
```

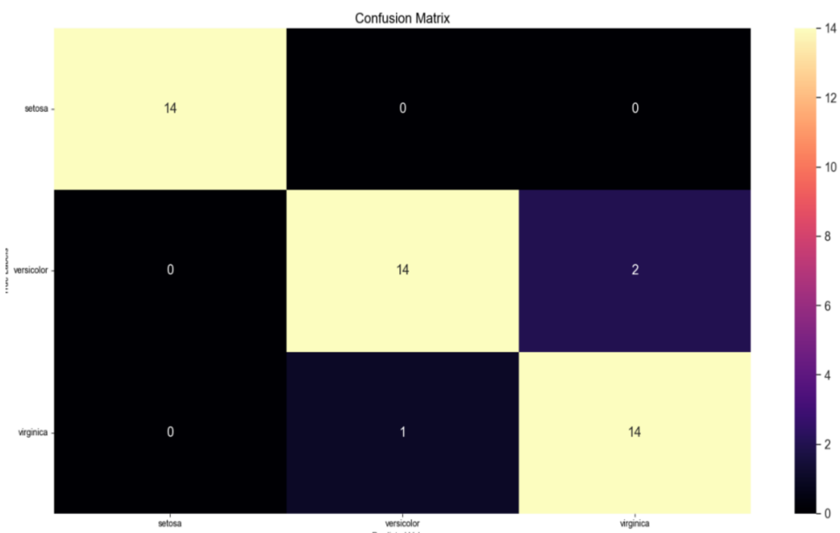
```
# Separating the independent dependent variables of the dataset

x = data.drop(columns = "Species")

y = data["Species"]

names_features = x.columns

target_labels = y.unique()

# Splitting the dataset into training and testing datasets

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3,
random_state = 93)

# Importing the Decision Tree classifier class from sklearn

from sklearn.tree import DecisionTreeClassifier

# Creating an instance of the classifier class

dtc = DecisionTreeClassifier(max_depth = 3, random_state = 93)

# Fitting the training dataset to the model

dtc.fit(x_train, y_train)

# Plotting the Decision Tree

plt.figure(figsize = (30, 10), facecolor = 'b')

Tree = tree.plot_tree(dtc, feature_names = names_features, class_names =
target_labels, rounded = True, filled = True, fontsize = 14)

plt.show()

y_pred = dtc.predict(x_test)

# Finding the confusion matrix

confusion_matrix = metrics.confusion_matrix(y_test, y_pred)
```

```
matrix = pd.DataFrame(confusion_matrix)

axis = plt.axes()

sns.set(font_scale = 1.3)

plt.figure(figsize = (10,7))

# Plotting heatmap

sns.heatmap(matrix, annot = True, fmt = "g", ax = axis, cmap = "magma")

axis.set_title('Confusion Matrix')

axis.set_xlabel("Predicted Values", fontsize = 10)

axis.set_xticklabels([''] + target_labels)

axis.set_ylabel( "True Labels", fontsize = 10)

axis.set_yticklabels(list(target_labels), rotation = 0)

plt.show()
```

OUTPUT



petal length (cm) <= 2.6
gini = 0.666
samples = 105
value = [36, 34, 35]
class = setosa
gini = 0.0
samples = 36
value = [36, 0, 0]
class = setosa
petal width (cm) <= 1.75
gini = 0.5
samples = 69
value = [0, 34, 35]
class = virginica
petal length (cm) <= 4.95
gini = 0.105
samples = 36
value = [0, 34, 2]
class = versicolor
gini = 0.0
samples = 33
value = [0, 0, 33]
class = virginica
gini = 0.0
samples = 33
value = [0, 33, 0]
class = versicolor
gini = 0.444
samples = 3
value = [0, 1, 2]
class = virginica



Confusion Matrix
setosa
versicolor
virginica
14
0
0
0
14
2
0
1
14
setosa
versicolor
virginica
Predicted Values
14
12
10
8
6
4
2
0

EX.No:7

DATE :

BUILD AN ARTIFICIAL NEURAL NETWORK BY IMPLEMENTING THE BACK PROPAGATION ALGORITHM AND TEST THE SAME USING APPROPRIATE DATA SETS.

AIM:

To implement the Build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data sets.

ALGORITHM:

1. Start the program.
2. **Initialization:**

Define the neural network architecture including the number of layers, number of neurons in each layer, activation functions, and learning rate.

Initialize weights and biases randomly or using some initialization technique.

3. **Forward Propagation:**

For each data point in the training set:

Input the data into the input layer of the neural network.

Compute the weighted sum of inputs and biases for each neuron in the hidden layers.

Apply the activation function to the computed sums to get the output of each neuron in the hidden layers.

Repeat the process for subsequent layers until the output layer is reached.

Compute the output of the neural network.

4. **Calculate Error:**

Compute the error between the predicted output and the actual output using an appropriate loss function.

5. **Backpropagation:**

Compute the gradient of the loss function with respect to the weights and biases of the network.

Update the weights and biases using the gradients and the learning rate to minimize the error.

Repeat this process for all data points in the training set.

6. **Training:**

Repeat steps 2-4 for a fixed number of iterations (epochs) or until convergence.

Monitor the loss function to ensure it is decreasing over epochs.

7. **Testing:**

Once training is complete, use the trained neural network to make predictions on a separate test dataset.

8. Evaluate the performance of the neural network using appropriate metrics such as accuracy, precision, recall, etc.

Iterate and Tune:

Based on the performance on the test dataset, adjust hyperparameters such as the learning rate, number of hidden layers, number of neurons in each layer, etc.

Re-train the neural network using the updated hyper parameters.

9. **Deployment**:

Once satisfied with the performance, deploy the trained neural network for making predictions on new, unseen data.

10. Stop the program.

PROGRAM

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) #maximum of X array longitudinally
y = y/100
#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))
#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)
#Variable initialization
epoch=5 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
```

```
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))

bout=np.random.uniform(size=(1,output_neurons))

#draws a random range of numbers uniformly of dim x*y

for i in range(epoch):

    #Forward Propogation

    hinp1=np.dot(X,wh)

    hinp=hinp1 + bh

    hlayer_act = sigmoid(hinp)

    outinp1=np.dot(hlayer_act,wout)

    outinp= outinp1+bout

    output = sigmoid(outinp)

    #Backpropagation

    EO = y-output

    outgrad = derivatives_sigmoid(output)

    d_output = EO * outgrad

    EH = d_output.dot(wout.T)

    hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer
wts contributed to error

    d_hiddenlayer = EH * hiddengrad

    wout += hlayer_act.T.dot(d_output) *lr   # dotproduct of nextlayererror
and currentlayerop

    wh += X.T.dot(d_hiddenlayer) *lr
```

```
 print ("-----------Epoch-", i+1, "Starts----------")


   print("Input: \n" + str(X))
    print("Actual Output: \n" + str(y))
    print("Predicted Output: \n" ,output)
    print ("-----------Epoch-", i+1, "Ends----------\n")
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

OUTPUT

```
IDLE Shell 3.12.1
File  Edit  Shell  Debug  Options  Window  Help

========================== RESTART: E:\python\6.py ==========================

========================== RESTART: E:\python\7.py ==========================
-----------Epoch- 1 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.80770709]
 [0.79315112]
 [0.80820483]]
-----------Epoch- 1 Ends----------

-----------Epoch- 2 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.8087915 ]
 [0.79421001]
 [0.80929214]]
-----------Epoch- 2 Ends----------

-----------Epoch- 3 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.80985378]
 [0.79524776]
 [0.81035723]]
-----------Epoch- 3 Ends----------
```

:

```
IDLE Shell 3.12.1
File  Edit  Shell  Debug  Options  Window  Help
 [0.79524776]
 [0.81035723]]
-----------Epoch- 3 Ends----------

-----------Epoch- 4 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.81089459]
 [0.79626498]
 [0.81140075]]
-----------Epoch- 4 Ends----------

-----------Epoch- 5 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.81191457]
 [0.79726227]
 [0.81242335]]
-----------Epoch- 5 Ends----------

Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.81191457]
 [0.79726227]
 [0.81242335]]
>>>
```

EX.No:8

DATE:

WRITE A PROGRAM TO IMPLEMENT THE NAÏVE BAYESIAN CLASSIFIER

AIM:

To write a program to implement the naïve Bayesian classifier.

ALGORITHM:

1. Start the program.
2. We start by importing dataset and necessary dependencies
3. Calculate Prior Probability of Classes P(y)
4. Calculate the Likelihood Table for all features
5. Now, Calculate Posterior Probability for each class using the Naive Bayesian equation. The Class with maximum probability is the outcome of the prediction.
6. Stop the program.

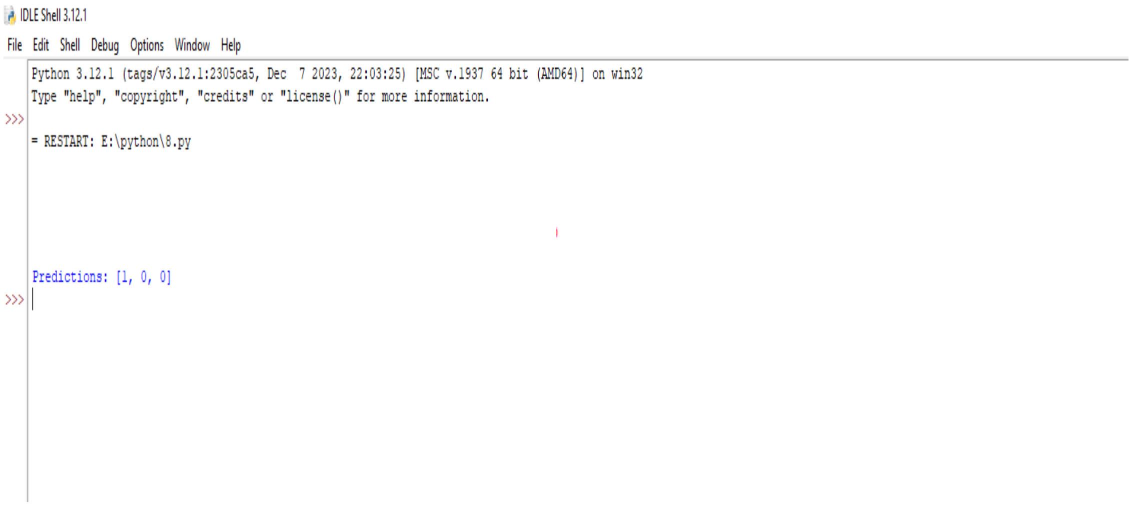
PROGRAM

```
import numpy as np
class NaiveBayesClassifier:
    def __init__(self):
        self.class_probabilities = {}
        self.feature_probabilities = {}
    def fit(self, X, y):
        # Calculate class probabilities
        classes, counts = np.unique(y, return_counts=True)
        total_samples = len(y)
        for c, count in zip(classes, counts):
            self.class_probabilities[c] = count / total_samples
        # Calculate feature probabilities
        self.feature_probabilities = {}
        for feature_index in range(X.shape[1]):
            self.feature_probabilities[feature_index] = {}
            unique_values = np.unique(X[:, feature_index])
            for value in unique_values:
                self.feature_probabilities[feature_index][value] = {}
                for c in classes:
                    samples_in_class = X[y == c]
```

```
                count_with_value = np.sum(samples_in_class[:, feature_index]
== value)

                self.feature_probabilities[feature_index][value][c] =
count_with_value / counts[c]

    def predict(self, X):

        predictions = []

        for sample in X:

            probabilities = {c: np.log(self.class_probabilities[c]) for c in
self.class_probabilities}

            for feature_index, value in enumerate(sample):

                for c in probabilities:

                    if value in self.feature_probabilities[feature_index]:

                        probabilities[c] +=
np.log(self.feature_probabilities[feature_index][value][c])

            predicted_class = max(probabilities, key=probabilities.get)

            predictions.append(predicted_class)

        return predictions

# Example usage:

if __name__ == "__main__":

    # Sample dataset

    X_train = np.array([[1, 'S'], [1, 'M'], [1, 'M'], [1, 'S'], [1, 'S'],

                        [2, 'S'], [2, 'M'], [2, 'M'], [2, 'L'], [2, 'L'],

                        [3, 'L'], [3, 'M'], [3, 'M'], [3, 'L'], [3, 'L']])

    y_train = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0])
```

```
X_test = np.array([[2, 'S'], [3, 'M'], [3, 'S']])

# Initialize and train the Naive Bayes classifier

nb_classifier = NaiveBayesClassifier()

nb_classifier.fit(X_train, y_train)

# Make predictions

predictions = nb_classifier.predict(X_test)

print("Predictions:", predictions)
```

OUTPUT:


IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\python\8.py
Predictions: [1, 0, 0]
>>>

UNSUPERVISED LEARNING

EX.No :9

DATE:

IMPLEMENTING NEURAL NETWORK USING SELF-ORGANIZING MAPS

AIM:

To implementing the neural network using Self organizing maps.

ALGORITHM

1. Start the program
2. Initialize the weights w_{ij} random value may be assumed. Initialize the learning rate α.
3. Calculate squared Euclidean distance.

   D(j) = Σ (wij – xi)^2 where i=1 to n and j=1 to m
4. Find index J, when D(j) is minimum that will be considered as winning index.
5. For each j within a specific neighborhood of j and for all i, calculate the new weight.

   wij(new)=wij(old) + α[xi – wij(old)]
6. Update the learning rule by using :

   α(t+1) = 0.5 * t
7. Test the Stopping Condition.
8. Stop the program.

PROGRAM

```
import math
class SOM:
    # Function here computes the winning vector
    # by Euclidean distance
    def winner(self, weights, sample):
        D0 = 0
        D1 = 0
        for i in range(len(sample)):
            D0 = D0 + math.pow((sample[i] - weights[0][i]), 2)
            D1 = D1 + math.pow((sample[i] - weights[1][i]), 2)
        # Selecting the cluster with smallest distance as winning cluster
        if D0 < D1:
            return 0
        else:
            return 1
    # Function here updates the winning vector
    def update(self, weights, sample, J, alpha):
        # Here iterating over the weights of winning cluster and modifying
them
        for i in range(len(weights[0])):
            weights[J][i] = weights[J][i] + alpha * (sample[i] - weights[J][i])
```

```python
    return weights
# Driver code
def main():
    # Training Examples ( m, n )
    T = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]
    m, n = len(T), len(T[0])
    # weight initialization ( n, C )
    weights = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
    # training
    ob = SOM()
    epochs = 3
    alpha = 0.5
    for i in range(epochs):
        for j in range(m):
            # training sample
            sample = T[j]
            # Compute winner vector
            J = ob.winner(weights, sample)
            # Update winning vector
            weights = ob.update(weights, sample, J, alpha)
    # classify test sample
```

```
s = [0, 0, 0, 1]
 J = ob.winner(weights, s)
    print("Test Sample s belongs to Cluster : ", J)
    print("Trained weights : ", weights)
if __name__ == "__main__":
    main()
```

OUTPUT

IDLE Shell 3.12.1

File Edit Shell Debug Options Window Help

```
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\python\9.py
Test Sample s belongs to Cluster :  0
Trained weights :  [[0.003125, 0.009375, 0.6640625, 0.9984375], [0.996875, 0.334375, 0.0109375, 0.0046875]]
>>>
```

EX.NO:10
DATE :

IMPLEMENTING K-MEANS ALGORITHM TO CLUSTER A SET OF DATA

AIM *:*

To implement the k-Means algorithm to cluster a set of data.

ALGORITHM

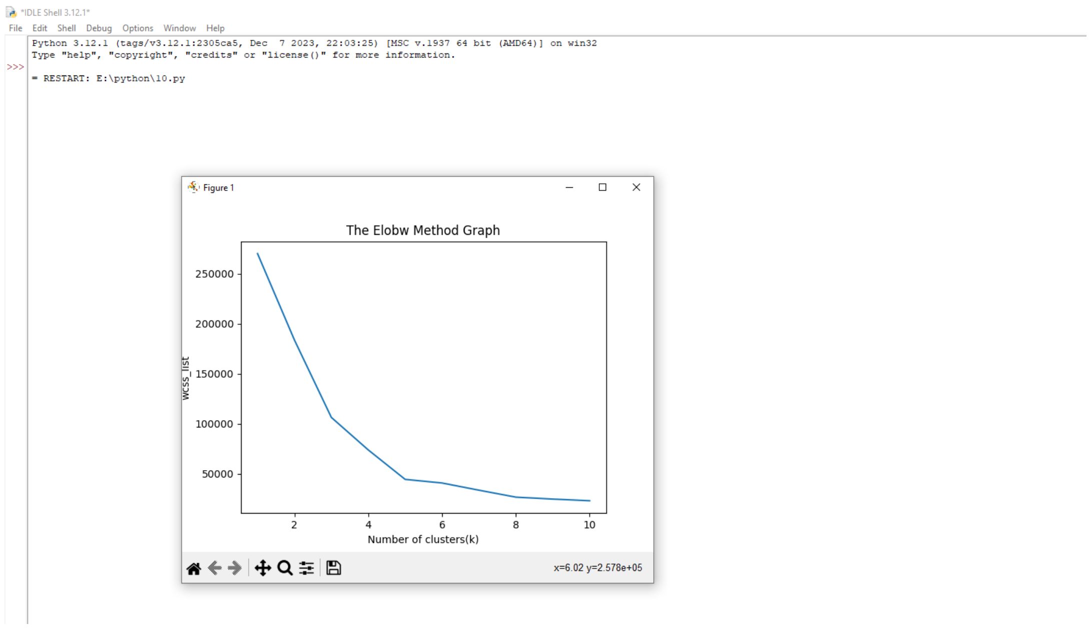
1. Start the program
2. Select the value of K to decide the number of clusters (n_clusters) to be formed.
3. Select random K points that will act as cluster centroids (cluster_centers).
4. Assign each data point, based on their distance from the randomly selected points (Centroid), to the nearest/closest centroid, which will form the predefined clusters.
5. Place a new centroid of each cluster.
6. Repeat step no.3, which reassigns each datapoint to the new closest centroid of each cluster.
7. If any reassignment occurs, then go to step 4; else, go to step 7.
8. Stop the program

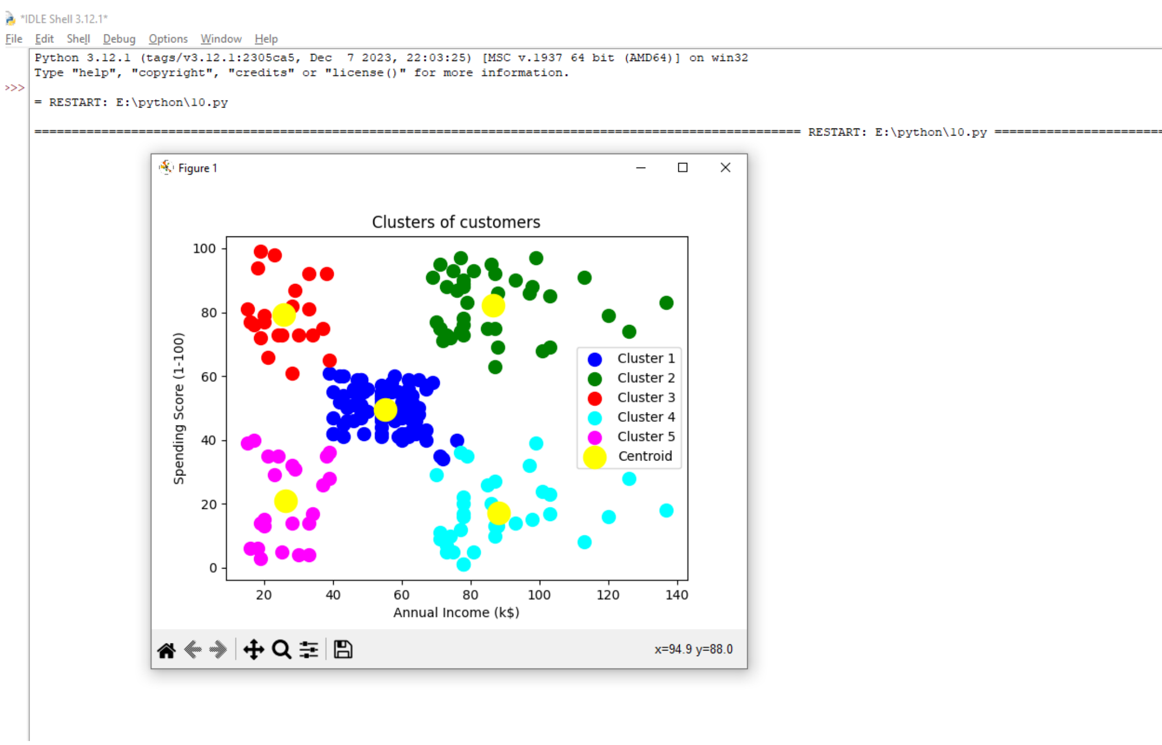
PROGRAM

```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
# Importing the dataset
dataset = pd.read_csv('Mall_Customers.csv')
x = dataset.iloc[:, [3, 4]].values
#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 Elobw Method Graph')
mtp.xlabel('Number of clusters(k)')
mtp.ylabel('wcss_list')
```

```
mtp.show()

#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)

#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()
```

OUTPUT


IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\python\10.py
Figure 1
The Elobw Method Graph
250000
200000
150000
100000
50000
wcss_list
2
4
6
8
10
Number of clusters(k)
x=6.02 y=2.578e+05


IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\python\10.py
RESTART: E:\python\10.py
Figure 1
Clusters of customers
100
80
60
40
20
0
Spending Score (1-100)
20
40
60
80
100
120
140
Annual Income (k$)
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Centroid
x=94.9 y=88.0

EX.No:11
DATE:

IMPLEMENTING HIERACHICAL CLUSTERING ALGORITHM

AIM:

To implement the hierarchical clustering algorithm.

ALGORITHM

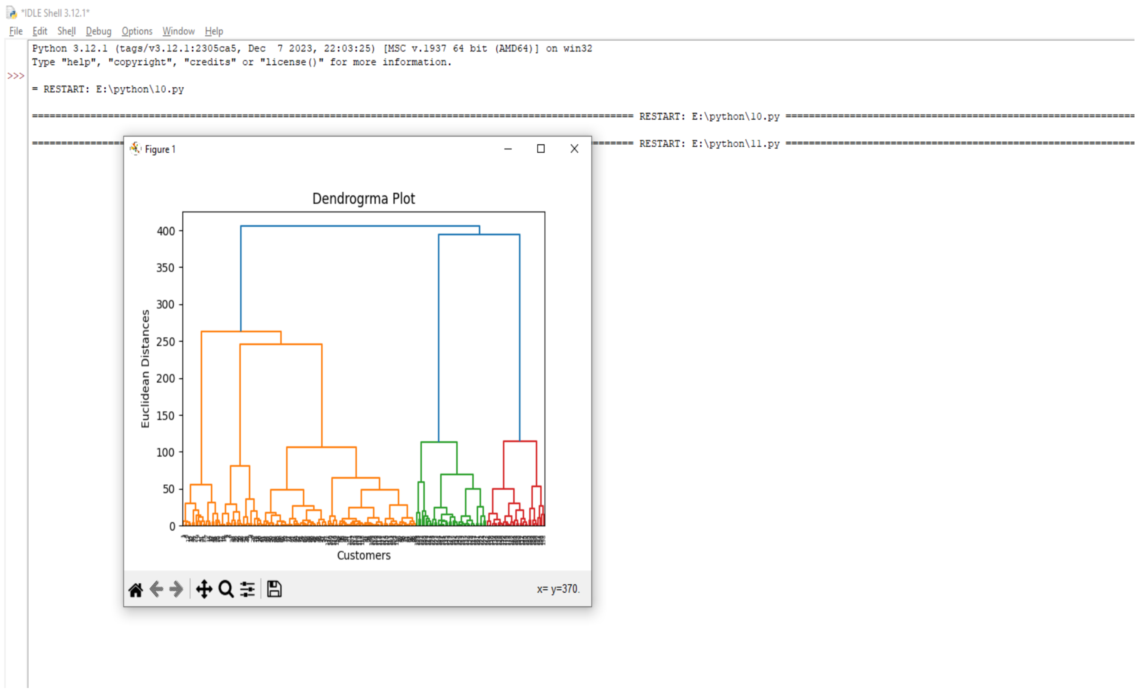
1. Start the program
2. we will import the libraries and datasets for our model.
3. Finding the optimal number of clusters using the Dendrogram
4. Training the hierarchical clustering model
5. Visualizing the clusters
6. Stop the program.

PROGRAM

```
# Importing the libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
# Importing the dataset
dataset = pd.read_csv('Mall_Customers.csv')
x = dataset.iloc[:, [3, 4]].values
#Finding the optimal number of clusters using the dendrogram
import scipy.cluster.hierarchy as shc
dendro = shc.dendrogram(shc.linkage(x, method="ward"))
mtp.title("Dendrogrma Plot")
mtp.ylabel("Euclidean Distances")
mtp.xlabel("Customers")
mtp.show()
#training the hierarchical model on dataset
from sklearn.cluster import AgglomerativeClustering
hc= AgglomerativeClustering(n_clusters=5, affinity='euclidean',
linkage='ward')
y_pred= hc.fit_predict(x)
#visulaizing the clusters
```

```
mtp.scatter(x[y_pred == 0, 0], x[y_pred == 0, 1], s = 100, c = 'blue', label = 'Cluster 1')

mtp.scatter(x[y_pred == 1, 0], x[y_pred == 1, 1], s = 100, c = 'green', label = 'Cluster 2')

mtp.scatter(x[y_pred== 2, 0], x[y_pred == 2, 1], s = 100, c = 'red', label = 'Cluster 3')

mtp.scatter(x[y_pred == 3, 0], x[y_pred == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')

mtp.scatter(x[y_pred == 4, 0], x[y_pred == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')

mtp.title('Clusters of customers')

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

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

mtp.legend()

mtp.show()
```

OUTPUT


IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: E:\python\10.py
RESTART: E:\python\10.py
RESTART: E:\python\11.py
Figure 1
Dendrogrma Plot
400
350
300
250
200
150
100
50
0
Euclidean Distances
Customers
x= y=370.