EXPT:1 IMPLEMENT BREADTH FIRST SEARCH

Aim: To implement Breadth first search algorithm in python.

Algorithm:

1. Start: Insert the starting node into a queue. Mark the starting node as visited.

2. Explore: While the queue is not empty:

    - o Remove a node from the queue and print its value
    - o Insert all unvisited adjacent nodes of the removed node into the queue. Mark each adjacent node as visited to avoid revisiting.

3. End: Repeat the process until the queue is empty.

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:

Following is the Breadth-First Search

5 3 7 2 4 8

Result:

Thus the program to implement BFS algorithm is successfully executed and verified.

EXPT: 2. IMPLEMENT DEPTH FIRST SEARCH

Aim: To implement Depth First Search algorithm in python.

Algorithm:

1. Start by putting any one of the graph's vertices on top of a stack.

2. Take the top item of the stack and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4. Keep repeating steps 2 and 3 until the stack is empty.

Program:

```
graph = {
   'A' : ['B','C'],
   'B' : ['D', 'E'],
   'C' : ['F'],
   'D' : [],
   'E' : ['F'],
   'F' : []
}
visited = set() # Set to keep track of visited nodes.
def dfs(visited, graph, node):
   if node not in visited:
      print (node)
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)
# Driver Code
dfs(visited, graph, 'A')
```

OUTPUT:

A B D E F C

Result: Thus the program to implement DFS algorithm is successfully executed and verified.

EXPT: 3 ANALYSIS OF BREADTH FIRST AND DEPTH FIRST SEARCH IN TERMS OF TIME AND SPACE

AIM: To analysis BFS and DFS in terms of time and space

Algorithm:

Step 1: Implement BFS and DFS algorithm

Step 2: Initiate the timer at the beginning of the code execution point

Step 3: Stop the timer and end of program execution

Step 4: Record the execution time of both BFS and DFS and compare.

Program:

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}
visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue
import time
# Save timestamp
start = time.time()
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:
    s = queue.pop(0)
    print (s, end = " ")
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
# Driver Code
bfs(visited, graph, 'A')
end = time.time()
print(end - start)


import time
# Save timestamp
start = time.time()
graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
```

```
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
# Driver Code
dfs(visited, graph, 'A')
end = time.time()

print(end - start)
```

Output:

BFS EXECUTION TIME: 0.00099945068359375

DFS EXECUTION TIME: 0.00099834057248261

Result: Thus the program to analyse BFS and DFS in terms of time and space is successfully executed and verified.

Implement and compare Greedy and A* algorithms

Aim: To implement and compare Greedy and A* algorithms

Step 1: Add the beginning node to the open list

Step 2: Repeat the following step

- In the open list, find the square with the lowest F cost, which denotes the current square. Now we move to the closed square.

- Consider 8 squares adjacent to the current square and ignore it if it is on the closed list or if it is not workable. Do the following if it is workable.

- Check if it is on the open list; if not, add it. You need to make the current square as this square's a parent. You will now record the different costs of the square, like the F, G, and H costs.

  – If it is on the open list, use G cost to measure the better path. The lower the G cost, the better the path. If this path is better, make the current square as the parent square. Now you need to recalculate the other scores – the G and F scores of this square. You'll stop:

- If you find the path, you need to check the closed list and add the target square to it.

- There is no path if the open list is empty and you cannot find the target square.

Step 3. Now you can save the path and work backward, starting from the target square, going to the parent square from each square you go, till it takes you to the starting square. You've found your path now.

Program (A* Algorithm):

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)
    closed_set = set()
    g = {} #store distance from starting node
    parents = {}# parents contains an adjacency map of all nodes

    #ditance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node


    while len(open_set) > 0:
       n = None

       #node with lowest f() is found
       for v in open_set:
          if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
             n = v


       if n == stop_node or Graph_nodes[n] == None:
          pass
       else:
```

```
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight


                #for each node m,compare its distance from start i.e g(m) to the
                #from start through n node
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n

                        #if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)

        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []

            while parents[n] != n:
                path.append(n)
                n = parents[n]

            path.append(start_node)

            path.reverse()

            print('Path found: {}'.format(path))
            return path


        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)

    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
```

```
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```

Program:

```
# Selection sort in Python (Greedy Algorithm)
def selectionSort(array, size):
        for step in range(size):
         min_idx = step
         for i in range(step + 1, size):
              # to sort in descending order, change > to < in this line
             # select the minimum element in each loop
             if array[i] < array[min_idx]:
                 min_idx = i
          # put min at the correct position
        (array[step], array[min_idx]) = (array[min_idx], array[step])
data = [-2, 45, 0, 11, -9]
size = len(data)
selectionSort(data, size)
print('Sorted Array in Ascending Order:')
print(data)
```

Output:
A* : Path found: ['A', 'E', 'D', 'G']
Greedy Algorithm: Sorted Array in Ascending Order:
[-9, -2, 0, 11, 45]

Time Complexity A* worse case time complexity is O(E), where E is the number of edges in the graph
Time Complexity Greedy Worst :$O(n^2)$ where n is number of elements.

Result: Thus the program to implement and compare Greedy and A* algorithms is successfully executed and verified.

Expt_5 IMPLEMENT THE NON-PARAMETRIC LOCALLY WEIGHTED REGRESSION ALGORITHM IN ORDER TO FIT DATA POINTS.

Aim:
To implement the non-parametric locally weighted regression algorithm in order to fit data points.

Algorithm
1. Import the necessary library files

2. Load the dataset

3. Extract total bill and tips paid from the data set to ColA and ColB variable

4. Creating a matrix (one) that contains ones corresponding to the size of rows in the ColB variable

5. Perform the horizontal Stacking of the data which contains transpose data of one and ColA matrix

6. Define the localWeightRegression function and pass the above data ,colb matrix and kernel value

7. Call the localWeightRegression function

8. Store the stacked matrix data into two different variables separately (m and n)

9. Create a zero vector that corresponds to the range of m

10. For each position of the vector calculate prediction from the passed value localWeight function

11. Define the local weight function

12. Calculate the local weights for the data points from the data received from the kernel function and return the value

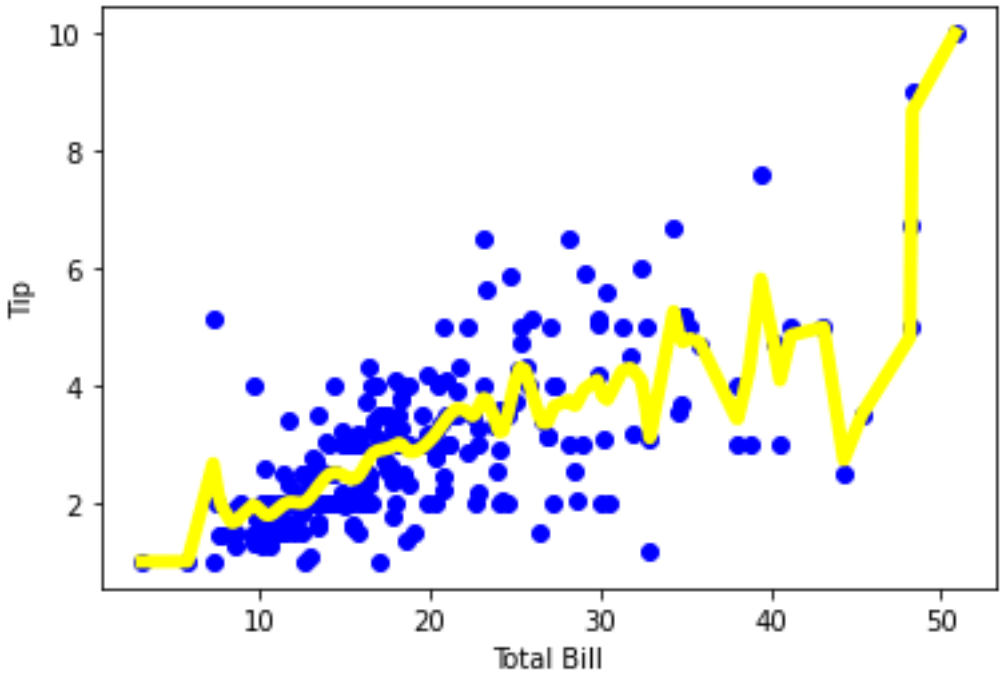
13. Define the kernel function to perform regression

Program:

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

# Gaussian kernel
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):
    wt = kernel(point, xmat, k)
    W = (X.T * (wt*X)).I * (X.T * wt * 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
data = pd.read_csv('C:/Users/user/Desktop/NMP/AIML_LAB_CSE/10-dataset.csv')
colA = np.array(data.total_bill)
colB = np.array(data.tip)
mcolA = np.mat(colA)
mcolB = np.mat(colB)
m = np.shape(mcolB)[1]
one = np.ones((1, m), dtype = int)
# horizontal stacking
X = np.hstack((one.T, mcolA.T))
print(X.shape)
ypred = localWeightRegression(X, mcolB, 0.5)
xsort = X.copy()
xsort.sort(axis=0)
plt.scatter(colA, colB, color='blue')
plt.plot(xsort[:, 1], ypred[X[:, 1].argsort(0)], color='yellow', linewidth=5)
plt.xlabel('Total Bill')
plt.ylabel('Tip')
plt.show()
```

Output:



Result:

Thus the program for Locally Weighted Regression algorithm is successfully implemented and executed.

Expt :6 DEMONSTRATE THE WORKING OF THE DECISION TREE BASED ALGORITHM.

Aim: To demonstrate the working of the decision tree based algorithm in scikit-learn and report the outcomes for iris dataset.

Algorithm:
1. Import necessary Library files
2. Load the iris data set
3. The data is split into features (X) and target (y).X contains feature information of iris data set (sepal length, sepal width, petal length, petal width) and y contains iris class (Iris setosa, Iris virginica and Iris versicolor) labels
4. The data is split into training and test sets using the train_test_split function from the sklearn.model_selection module.
5. Create an instance of Decision Tree Classifier
6. Fit the classifier to the training data X_train and y_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

```
#Decision TREE
import pandas as pd
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import confusion_matrix
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random_state=4)
# Create Decision Tree classifer object
clf = DecisionTreeClassifier()
# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)
# performing predictions on the test dataset
y_pred = clf.predict(X_test)
# metrics are used to find accuracy or error
print("ACCURACY OF THE MODEL:", metrics.accuracy_score(y_test, y_pred))
accuracy = metrics.accuracy_score(y_test, y_pred)
precision = metrics.precision_score(y_test, y_pred,average='micro')
recall = metrics.recall_score(y_test, y_pred,average='micro')
```

Output:

```
ACCURACY OF THE MODEL: 0.9777777777777777
PRECISION OF THE MODEL: 0.9777777777777777
RECALL OF THE MODEL: 0.9777777777777777
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        21
           1       1.00      0.90      0.95        10
           2       0.93      1.00      0.97        14

    accuracy                           0.98        45
   macro avg       0.98      0.97      0.97        45
weighted avg       0.98      0.98      0.98        45
```

Result: Thus the program to demonstrate the working of the decision tree based algorithm in scikit-learn and report the outcomes for iris dataset is successfully implemented and executed.

EXPT 7 ARTIFICIAL NEURAL NETWORK USING BACK PROPAGATION ALGORITHM

Aim: To build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data

Algorithm:

1. Import necessary Library files
2. Load the breast cancer data set
3. The data is split into features (X) and target (y).
4. The data is split into training and test sets using the train_test_split function from the sklearn.model_selection module.
5. Perform scaling operation of data.
6. Create an instance of Multi layer Multi-layer Perceptron classifier
7. Fit the classifier to the training data X_train and y_train
8. Perform the prediction
9. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

Program:

```
# Import necessary libraries
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the Breast Cancer dataset
cancer_data = load_breast_cancer()
X, y = cancer_data.data, cancer_data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create an MLPClassifier model
mlp = MLPClassifier(hidden_layer_sizes=(64, 32),max_iter=1000, random_state=42)

# Train the model on the training data
mlp.fit(X_train, y_train)

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

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
#Accuracy: 0.97

# Generate a classification report
class_report = classification_report(y_test, y_pred)
```

```
print("Classification Report:\n", class_report)

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.95      0.96        43
           1       0.97      0.99      0.98        71

    accuracy                           0.97       114
   macro avg       0.97      0.97      0.97       114
weighted avg       0.97      0.97      0.97       114
```

Result: Thus the program to build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data is successfully implemented and executed.

Expt:8. Program to implement the naïve Bayesian classifier.

Aim:

To implement naïve Bayes classifier in scikit-learn and report the outcomes for iris dataset.

Algorithm

1. Import necessary Library files
2. Load the iris data set
3. The data is split into features (X) and target (y).X contains feature information of iris data set (sepal length, sepal width, petal length, petal width) and y contains iris class (Iris setosa, Iris virginica and Iris versicolor) labels
4. The data is split into training and test sets using the train_test_split function from the sklearn.model_selection module.
5. Create an instance of Gaussian Naïve Bayes classfier
6. Fit the classifier to the training data X_train and y_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

Program:

```
import pandas as pd
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import confusion_matrix
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random_state=4)
NBclassifier = GaussianNB()
NBclassifier.fit(X_train, y_train)
# performing predictions on the test dataset
y_pred = NBclassifier.predict(X_test)
# metrics are used to find accuracy or error
print("ACCURACY OF THE MODEL:", metrics.accuracy_score(y_test, y_pred))
accuracy = metrics.accuracy_score(y_test, y_pred)
precision = metrics.precision_score(y_test, y_pred,average='micro')
recall = metrics.recall_score(y_test, y_pred,average='micro')
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

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

Output:

ACCURACY OF THE MODEL: 0.9777777777777777

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	21
1	0.91	1.00	0.95	10
2	1.00	0.93	0.96	14
accuracy			0.98	45
macro avg	0.97	0.98	0.97	45
weighted avg	0.98	0.98	0.98	45

Result: Thus the program to implement naïve Bayes classifier in scikit-learn and report the outcomes for iris dataset are successfully implemented and executed.

Expt 9: IMPLEMENT NEURAL NETWORK USING SELF-ORGANIZING MAPS

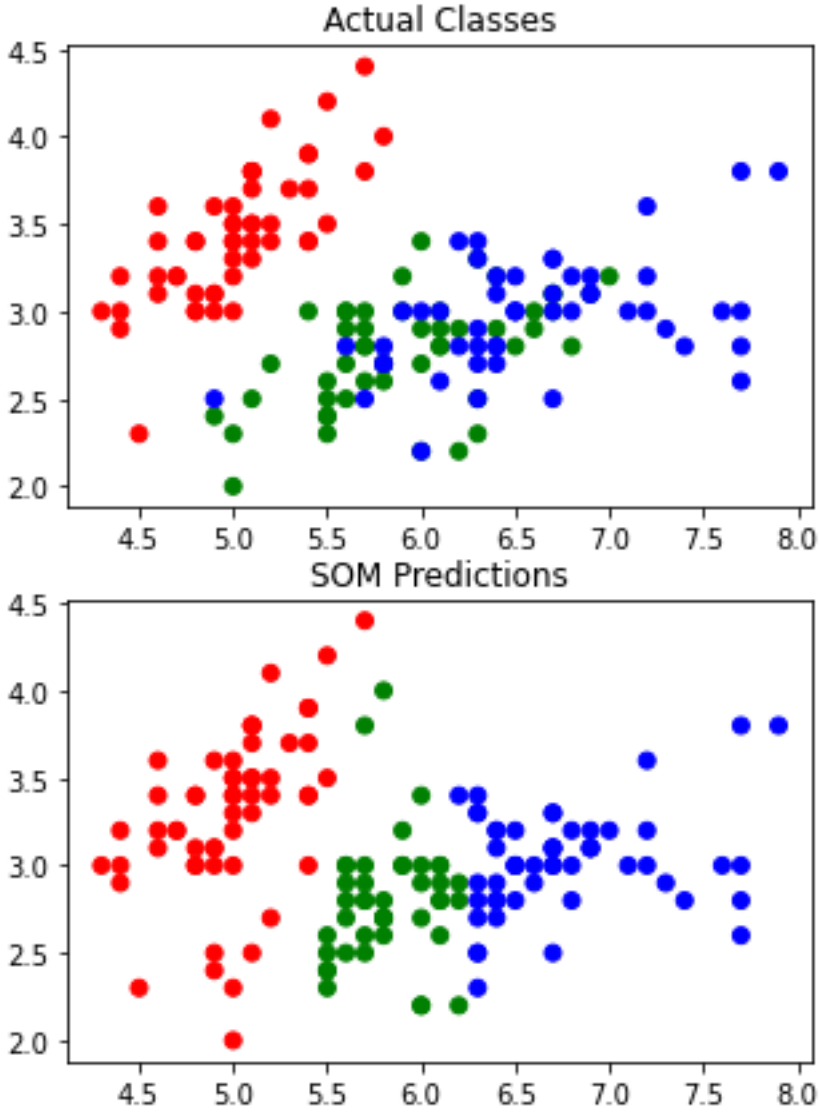
Aim:

To implement neural network using self-organizing maps.

Algorithm:

1. Import necessary Library files
2. Load the iris data set
3. The data is split into features (iris_data) and target (iris_label). Iris_data contains feature information of iris data set (sepal length, sepal width, petal length, petal width) and iris_label contains iris class (Iris setosa, Iris virginica and Iris versicolor) labels
4. Define the SOM classifier
5. Fit the classifier to the training data
6. Perform the prediction
7. Plot the clusters.

```
# pip install sklearn-som
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn_som.som import SOM
# Load iris data
iris = datasets.load_iris()
iris_data = iris.data
iris_label = iris.target
# Extract just two features (just for ease of visualization)
iris_data = iris_data[:, :2]
# Build a 3x1 SOM (3 clusters)
som = SOM(m=3, n=1, dim=2, random_state=1234)
# Fit it to the data
som.fit(iris_data)
# Assign each datapoint to its predicted cluster
predictions = som.predict(iris_data)
# Plot the results
fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(5,7))
x = iris_data[:,0]
y = iris_data[:,1]
colors = ['red', 'green', 'blue']
ax[0].scatter(x, y, c=iris_label, cmap=ListedColormap(colors))
ax[0].title.set_text('Actual Classes')
ax[1].scatter(x, y, c=predictions, cmap=ListedColormap(colors))
ax[1].title.set_text('SOM Predictions')
plt.savefig('iris_example.png')
```



Result:

Thus the program to implement neural network using self-organizing maps is successfully implemented and executed.

Expt 10 : k-Means algorithm

Aim:

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

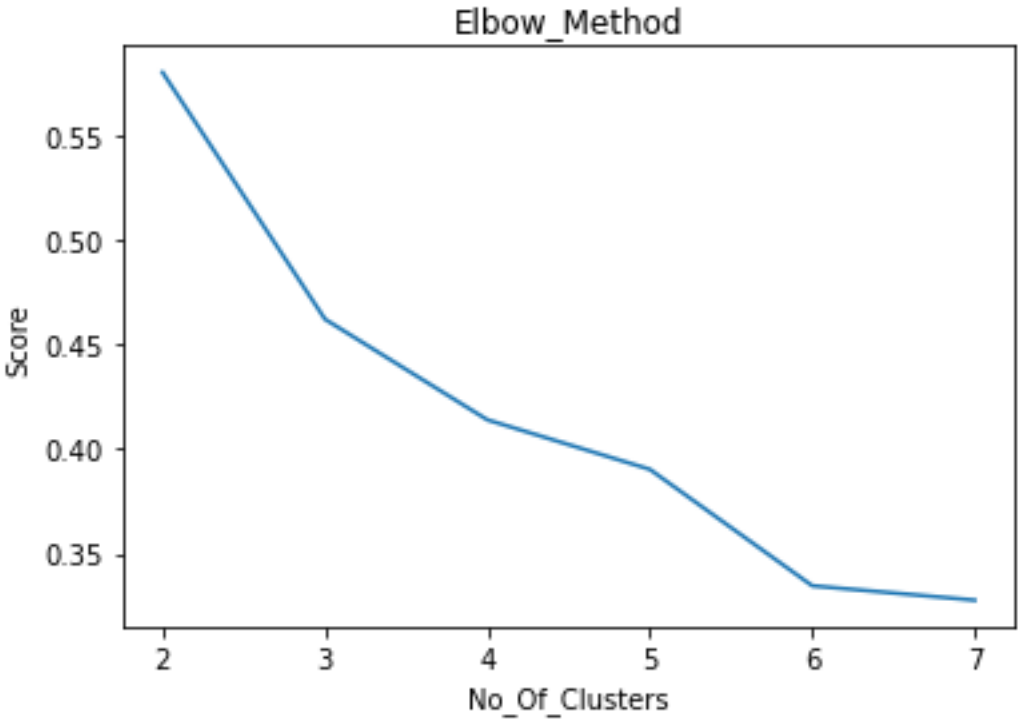
Algorithm:

1. Import necessary Library files
2. Load the iris data set
3. Drop Id and Species column in the data set.
4. Perform scaling operation on the data
5. Create an instance of Kmeans classfier
6. Fit the data
7. Plot the silhouette_score for kmeans clustering

Program:

```
import pandas as pd
df=pd.read_csv('C:/Users/user/Desktop/NMP/AIML_LAB_IT/DATASET/Iris.csv')
df = df.drop(["Id", "Species"], axis="columns")
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df = scaler.fit_transform(df)
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters = 4, random_state = 0, n_init='auto')
kmeans.fit(df)
from sklearn.metrics import silhouette_score
K = range(2, 8)
fits = []
score = []
import seaborn as sns
for k in K:
    # train the model for current value of k on training data
    model = KMeans(n_clusters = k, random_state = 0, n_init='auto').fit(df)
    # append the model to fits
    fits.append(model)
    # Append the silhouette score to scores
    score.append(silhouette_score(df, model.labels_, metric='euclidean'))

sns.lineplot(x=K, y= score).set(title='Elbow_Method', xlabel='No_Of_Clusters', ylabel='Score')
```

Output:



Result:

Thus the program to implement k-means clustering is successfully implemented and executed.

Expt 11: Hierarchical clustering algorithm

Aim:

To implement hierarchical clustering algorithm ('Agglomerative Clustering)

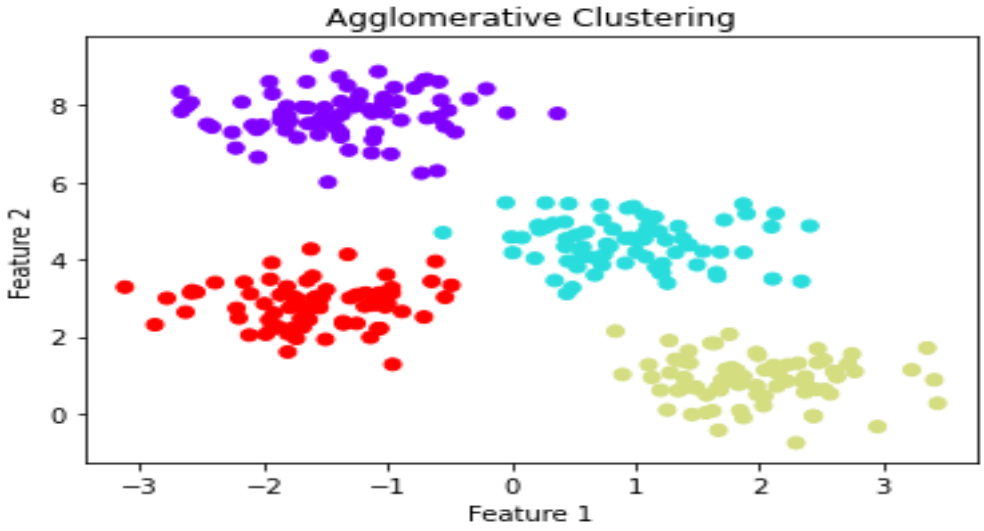
Algorithm:

1. Import necessary library files
2. Artificially generate data points using make_blobs method for 300 samples
3. Create an instance of Agglomerative Clustering
4. Predict the labels for the features in the generated data
5. Plot the graph.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
# Generate sample data
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
# Perform agglomerative clustering
agg_clustering = AgglomerativeClustering(n_clusters=4)
y_pred = agg_clustering.fit_predict(X)
# Plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='rainbow')
plt.title('Agglomerative Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

Output:



Result:

Thus the program to implement hierarchical clustering algorithm is successfully implemented and executed.