PROGRAM: 1

IMPLEMENT A* ALGORITHM

CODE:

```
def aStarAlgo(start_node, stop_node):
        open_set = set(start_node)
        closed_set = set()
        g = {}
        parents = {}
        g[start_node] = 0

        parents[start_node] = start_node

        while len(open_set) > 0:
            n = None

            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):

                    if m not in open_set and m not in closed_set:
                        open_set.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight
```

```
            else:
                if g[m] > g[n] + weight
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)
    if n == None:
        print('Path does not exist!')
        return None

    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

    open_set.remove(n)
    closed_set.add(n)

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

```
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
        H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
        }
        return H_dist[n]

Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

OUTPUT:

```
Path found: ['A', 'E', 'D', 'G']
['A', 'E', 'D', 'G']
```

PROGRAM: 2

IMPLEMENT AO* ALGORITHM

CODE:

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self):
        self.aoStar(self.start, False)

    def getNeighbors(self, v):
        return self.graph.get(v,'')

    def getStatus(self,v):
        return self.status.get(v,0)

    def setStatus(self,v, val):
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0)

    def setHeuristicNodeValue(self, n, value):
```

```
        self.H[n]=value


    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")
    def computeMinimumCostChildNodes(self, v):
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v):
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)
                if flag==True:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList
                    flag=False
                else:
                    if minimumCost>cost:
                        minimumCost=cost
                        costToChildNodeListDict[minimumCost]=nodeList
        return minimumCost, costToChildNodeListDict[minimumCost]
```

```
    def aoStar(self, v, backTracking):
        print("HEURISTIC VALUES :", self.H)
        print("SOLUTION GRAPH :", self.solutionGraph)
        print("PROCESSING NODE :", v)
        print("-----------------------------------------------------------------------------------------")
        if self.getStatus(v) >= 0:
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v,len(childNodeList))
            solved=True
        for childNode in childNodeList:
            self.parent[childNode]=v
            if self.getStatus(childNode)!=-1:
                solved=solved & False
        if solved==True:
            self.setStatus(v,-1)
            self.solutionGraph[v]=childNodeList
        if v!=self.start:
            self.aoStar(self.parent[v], True)
        if backTracking==False:
            for childNode in childNodeList:
                self.setStatus(childNode,0)
                self.aoStar(childNode, False)
h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
```

```
    'G': [[('I', 1)]]
    }
G1= Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
graph2 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'D': [[('E', 1), ('F', 1)]]
    }
G2 = Graph(graph2, h2, 'A')
G2.applyAOStar()
G2.printSolution()
```

OUTPUT:

```
HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : B
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
```

SOLUTION GRAPH : {}

PROCESSING NODE : A

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : G

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : B

HEURISTIC VALUES : {'A': 7, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : A

HEURISTIC VALUES : {'A': 9, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

SOLUTION GRAPH : {}

PROCESSING NODE : I

HEURISTIC VALUES : {'A': 9, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': []}

PROCESSING NODE : G

HEURISTIC VALUES : {'A': 9, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I']}

PROCESSING NODE : B

HEURISTIC VALUES : {'A': 9, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : A

HEURISTIC VALUES : {'A': 3, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : C

HEURISTIC VALUES : {'A': 3, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : A

HEURISTIC VALUES : {'A': 3, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : J

HEURISTIC VALUES : {'A': 3, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': []}

PROCESSING NODE : C

HEURISTIC VALUES : {'A': 3, 'B': 2, 'C': 1, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T': 3}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J']}

PROCESSING NODE : A

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}

HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : B

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : G

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 0, 'H': 7}

SOLUTION GRAPH : {'G': []}

PROCESSING NODE : B

HEURISTIC VALUES: {'A': 7, 'B': 1, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 0, 'H': 7}

SOLUTION GRAPH: {'G': [], 'B': ['G']}

PROCESSING NODE: A

HEURISTIC VALUES: {'A': 2, 'B': 1, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 0, 'H': 7}

SOLUTION GRAPH: {'G': [], 'B': ['G']}

PROCESSING NODE: C

HEURISTIC VALUES: {'A': 2, 'B': 1, 'C': 0, 'D': 10, 'E': 4, 'F': 4, 'G': 0, 'H': 7}

SOLUTION GRAPH: {'G': [], 'B': ['G'], 'C': []}

PROCESSING NODE: A

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

{'G': [], 'B': ['G'], 'C': [], 'A': ['B', 'C']}

PROGRAM: 5

BUILD AN ANN USING BACK PROPOGATION ALGORITHM

CODE:

```
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)
y = y/100

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

def derivatives_sigmoid(x):
    return x * (1 - x)

epoch=7000
lr=0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1

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))

for i in range(epoch):
```

```
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)

    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)

    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr

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

print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

OUTPUT:

Input:

```
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
```

Actual Output:

```
[[0.92]
 [0.86]
 [0.89]]
```

Predicted Output:

```
 [[0.89309893]
 [0.87794188]
 [0.89899306]]
```

PROGRAM: 6

NAÏVE BAYESIAN CLASSIFIER FOR A GIVEN .CSV FILE

CODE:

```
import csv
import random
import math
def loadCsv(filename):
    lines = csv.reader(open(filename, "r"));
    dataset = list(lines)
    for i in range(len(dataset)):

        dataset[i] = [float(x) for x in dataset[i]]
    return dataset
def splitDataset(dataset, splitRatio):

    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    copy = list(dataset)
    while len(trainSet) < trainSize:

        index = random.randrange(len(copy))
        trainSet.append(copy.pop(index))
    return [trainSet, copy]
def separateByClass(dataset):
    separated = {}

    for i in range(len(dataset)):
        vector = dataset[i]
```

```
        if (vector[-1] not in separated):
            separated[vector[-1]] = []
        separated[vector[-1]].append(vector)
    return separated
def mean(numbers):
    return sum(numbers)/float(len(numbers))
def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)
def summarize(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
    del summaries[-1]
    return summaries
def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    summaries = {}
    for classValue, instances in separated.items():

        summaries[classValue] = summarize(instances)
    return summaries
def calculateProbability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))
    return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent
def calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.items():

        probabilities[classValue] = 1
```

```
    for i in range(len(classSummaries)):
        mean, stdev = classSummaries[i]
        x = inputVector[i]
        probabilities[classValue] *= calculateProbability(x, mean, stdev)
        return probabilities
def predict(summaries, inputVector):
    probabilities = calculateClassProbabilities(summaries, inputVector)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.items():

        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel
def getPredictions(summaries, testSet):
    predictions = []
    for i in range(len(testSet)):
        result = predict(summaries, testSet[i])
        predictions.append(result)
    return predictions
def getAccuracy(testSet, predictions):
    correct = 0
    for i in range(len(testSet)):
        if testSet[i][-1] == predictions[i]:
            correct += 1
    return (correct/float(len(testSet))) * 100.0
def main():
    filename = 'C:\\Users\\DELL\\.jupyter\\pima-indians-diabetes.csv'
    splitRatio = 0.67
```

```
    dataset = loadCsv(filename)
    trainingSet, testSet = splitDataset(dataset, splitRatio)
    print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),
len(trainingSet),
    len(testSet)))

    summaries = summarizeByClass(trainingSet);

    predictions = getPredictions(summaries, testSet)
    accuracy = getAccuracy(testSet, predictions)
    print('Accuracy of the classifier is : {0}%'.format(accuracy))
main()
```

OUTPUT:

Split 768 rows into train=514 and test=254 rows

Accuracy of the classifier is : 35.43307086614173%

PROGRAM: 7

APPLY EM ALGORITHM TO CLUSTER A SET OF DATA STORE IN A .CSV FILE

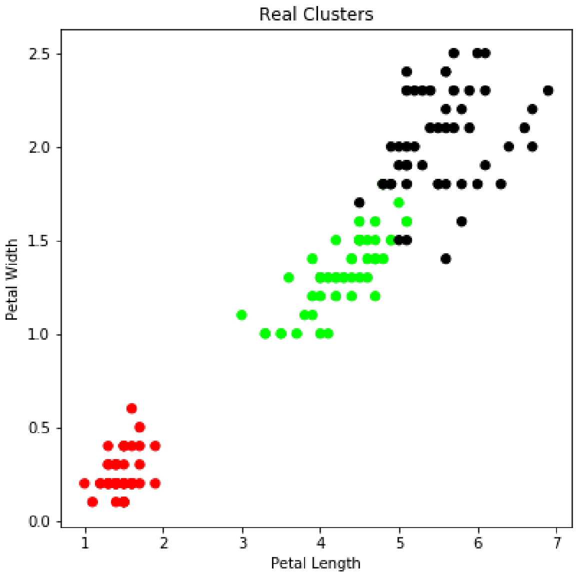
CODE:

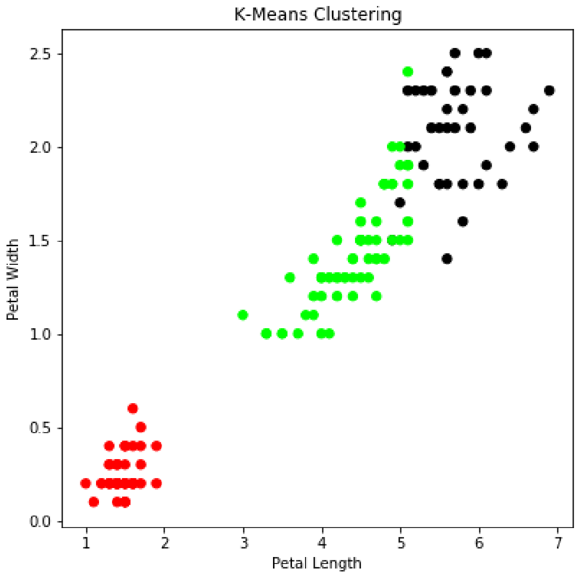
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import pandas as pd
import numpy as np

iris = datasets.load_iris()
X = pd.DataFrame(iris.data)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

model =KMeans(n_clusters=3)
model.fit(X)

plt.figure(figsize=(14,14))
colormap = np.array(['red', 'lime', 'black'])

plt.subplot(2, 2, 1)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.Targets], s=40)
plt.title('Real Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
```

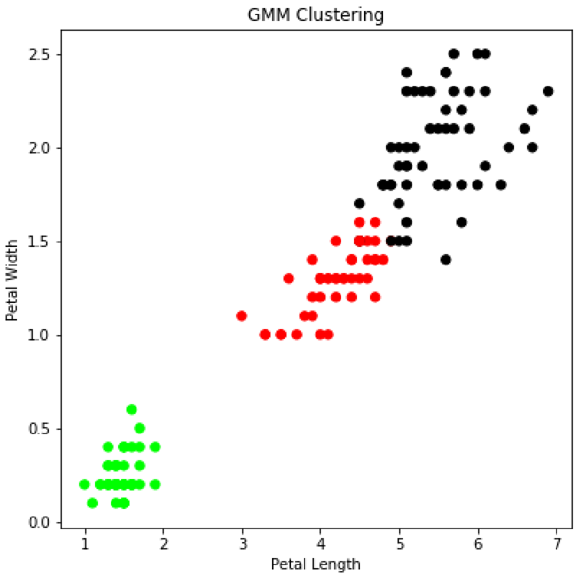
```
plt.subplot(2, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

from sklearn import preprocessing

scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y = gmm.predict(xs)
plt.subplot(2, 2, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[gmm_y], s=40)
plt.title('GMM Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
print('Observation: The GMM using EM algorithm based clustering matched the true labels more closely than the Kmeans.')
```

OUTPUT:

Observation: The GMM using EM algorithm based clustering matched the true labels more closely than the Kmeans.







PROGRAM: 8

IMPLEMENT K-NEAREST NEIGHBOUR ALGORITHM TO CLASSIFY IRIS DATASET

CODE:

```
from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn import datasets


iris=datasets.load_iris()

print("Iris Data set loaded...")


x_train, x_test, y_train, y_test = train_test_split(iris.data,iris.target,test_size=0.1)

print("Dataset is split into training and testing...")

print("Size of trainng data and its label",x_train.shape,y_train.shape)

print("Size of trainng data and its label",x_test.shape, y_test.shape)


for i in range(len(iris.target_names)):

    print("Label", i , "-",str(iris.target_names[i]))


classifier = KNeighborsClassifier(n_neighbors=1)


classifier.fit(x_train, y_train)


y_pred=classifier.predict(x_test)


print("Results of Classification using K-nn with K=1 ")

for r in range(0,len(x_test)):
```

```
    print(" Sample:", str(x_test[r]), " Actual-label:", str(y_test[r]), " Predicted-label:",str(y_pred[r]))
print("Classification Accuracy :" , classifier.score(x_test,y_test));
from sklearn.metrics import classification_report, confusion_matrix
print('Confusion Matrix')
print(confusion_matrix(y_test,y_pred))
print('Accuracy Metrics')
print(classification_report(y_test,y_pred))
```

OUTPUT:

```
Iris Data set loaded...
Dataset is split into training and testing...
Size of trainng data and its label (135, 4) (135,)
Size of trainng data and its label (15, 4) (15,)
Label 0 - setosa
Label 1 - versicolor
Label 2 - virginica
Results of Classification using K-nn with K=1
 Sample: [5.  3.5 1.6 0.6]  Actual-label: 0  Predicted-label: 0
 Sample: [6.6 3.  4.4 1.4]  Actual-label: 1  Predicted-label: 1
 Sample: [6.7 3.  5.2 2.3]  Actual-label: 2  Predicted-label: 2
 Sample: [4.8 3.  1.4 0.3]  Actual-label: 0  Predicted-label: 0
 Sample: [6.3 3.4 5.6 2.4]  Actual-label: 2  Predicted-label: 2
 Sample: [5.2 4.1 1.5 0.1]  Actual-label: 0  Predicted-label: 0
 Sample: [5.6 3.  4.5 1.5]  Actual-label: 1  Predicted-label: 1
 Sample: [6.5 3.  5.5 1.8]  Actual-label: 2  Predicted-label: 2
 Sample: [6.8 3.  5.5 2.1]  Actual-label: 2  Predicted-label: 2
```

Sample: [5.9 3. 4.2 1.5] Actual-label: 1 Predicted-label: 1

Sample: [7.4 2.8 6.1 1.9] Actual-label: 2 Predicted-label: 2

Sample: [5.6 2.9 3.6 1.3] Actual-label: 1 Predicted-label: 1

Sample: [5.8 2.7 4.1 1.] Actual-label: 1 Predicted-label: 1

Sample: [4.9 3.1 1.5 0.1] Actual-label: 0 Predicted-label: 0

Sample: [7.2 3.6 6.1 2.5] Actual-label: 2 Predicted-label: 2

Classification Accuracy : 1.0

Confusion Matrix

```
[[4 0 0]
 [0 5 0]
 [0 0 6]]
```

Accuracy Metrics

	precision	recall	f1-score	support
0	1.00	1.00	1.00	4
1	1.00	1.00	1.00	5
2	1.00	1.00	1.00	6
avg / total	1.00	1.00	1.00	15

PROGRAM: 9

IMPLEMENT NON-PARAMETRIC LOCALLY WEIGHTED REGRESSION

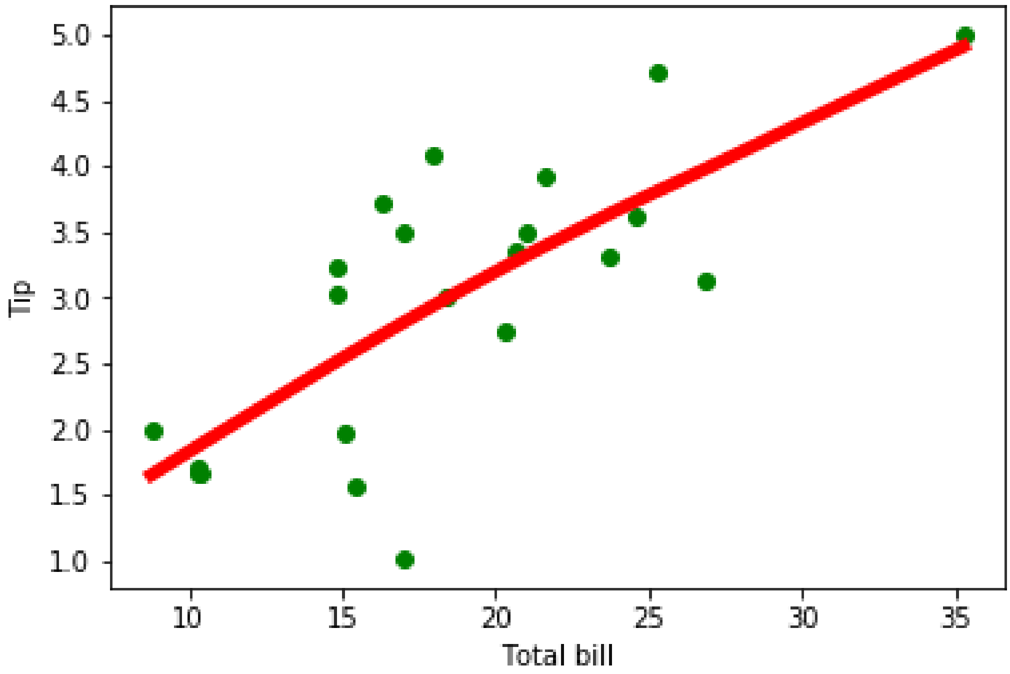
CODE:

```
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

def graphPlot(X,ypred):

    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();

data = pd.read_csv('C:\\Users\\DELL\\.jupyter\\10data_tips.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)
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))
ypred = localWeightRegression(X,mtip,8)
graphPlot(X,ypred)
```

OUTPUT:

The following is the output for dataset with 10 examples (10 rows and 2 columns)



The following is the output for dataset with 10 examples (10 rows and 2 columns)