**PROGRAM 1**

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

#distance 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 reconstructing 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

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,

'D': [('G', 1)],

'E': [('D', 6)],

}

aStarAlgo('A', 'G')

**PROGRAM 2**

class Graph:

def \_\_init\_\_(self, graph, heuristicNodeList, startNode): #instantiate graph object with graph topology, heuristic values, start node

self.graph = graph

self.H=heuristicNodeList

self.start=startNode

self.parent={}

self.status={}

self.solutionGraph={}

def applyAOStar(self): # starts a recursive AO\* algorithm

self.aoStar(self.start, False)

def getNeighbors(self, v): # gets the Neighbors of a given node

return self.graph.get(v,'')

def getStatus(self,v): # return the status of a given node

return self.status.get(v,0)

def setStatus(self,v, val): # set the status of a given node

self.status[v]=val

def getHeuristicNodeValue(self, n):

return self.H.get(n,0) # always return the heuristic value of a given node

def setHeuristicNodeValue(self, n, value):

self.H[n]=value # set the revised heuristic value of a given node

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): # Computes the Minimum Cost of child nodes of a given node v

minimumCost=0

costToChildNodeListDict={}

costToChildNodeListDict[minimumCost]=[]

flag=True

for nodeInfoTupleList in self.getNeighbors(v): # iterate over all the set of child node/s

cost=0

nodeList=[]

for c, weight in nodeInfoTupleList:

cost=cost+self.getHeuristicNodeValue(c)+weight

nodeList.append(c)

if flag==True: # initialize Minimum Cost with the cost of first set of child node/s

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s

flag=False

else: # checking the Minimum Cost nodes with the current Minimum Cost

if minimumCost>cost:

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s

return minimumCost, costToChildNodeListDict[minimumCost] # return Minimum Cost and Minimum Cost child node/s

def aoStar(self, v, backTracking): # AO\* algorithm for a start node and backTracking status flag

print("HEURISTIC VALUES :", self.H)

print("SOLUTION GRAPH :", self.solutionGraph)

print("PROCESSING NODE :", v)

print("-----------------------------------------------------------------------------------------")

if self.getStatus(v) >= 0: # if status node v >= 0, compute Minimum Cost nodes of v

minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)

print(minimumCost, childNodeList)

self.setHeuristicNodeValue(v, minimumCost)

self.setStatus(v,len(childNodeList))

solved=True # check the Minimum Cost nodes of v are solved

for childNode in childNodeList:

self.parent[childNode]=v

if self.getStatus(childNode)!=-1:

solved=solved & False

if solved==True: # if the Minimum Cost nodes of v are solved, set the current node status as solved(-1)

self.setStatus(v,-1)

self.solutionGraph[v]=childNodeList # update the solution graph with the solved nodes which may be a part of solution

if v!=self.start: # check the current node is the start node for backtracking the current node value

self.aoStar(self.parent[v], True) # backtracking the current node value with backtracking status set to true

if backTracking==False: # check the current call is not for backtracking

for childNode in childNodeList: # for each Minimum Cost child node

self.setStatus(childNode,0) # set the status of child node to 0(needs exploration)

self.aoStar(childNode, False) # Minimum Cost child node is further explored with backtracking status as false

#for simplicity we ll consider heuristic distances given

print ("Graph - 1")

h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}

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} # Heuristic values of Nodes

graph2 = { # Graph of Nodes and Edges

'A': [[('B', 1), ('C', 1)], [('D', 1)]], # Neighbors of Node 'A', B, C & D with repective weights

'B': [[('G', 1)], [('H', 1)]], # Neighbors are included in a list of lists

'D': [[('E', 1), ('F', 1)]] # Each sublist indicate a "OR" node or "AND" nodes

}

G2 = Graph(graph2, h2, 'A') # Instantiate Graph object with graph, heuristic values and start Node

G2.applyAOStar() # Run the AO\* algorithm

G2.printSolution() # Print the solution graph as output of the AO\* algorithm search

**PROGRAM 3**

import numpy as np

import pandas as pd

data = pd.read\_csv('finds.csv')

concepts = np.array(data.iloc[:,0:-1])

target = np.array(data.iloc[:,-1])

def learn(concepts, target):

specific\_h = concepts[0].copy()

general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]

for i, h in enumerate(concepts):

if target[i] == "yes":

for x in range(len(specific\_h)):

if h[x]!= specific\_h[x]:

specific\_h[x] ='?'

general\_h[x][x] ='?'

if target[i] == "no":

for x in range(len(specific\_h)):

if h[x]!= specific\_h[x]:

general\_h[x][x] = specific\_h[x]

else:

general\_h[x][x] = '?'

indices = [i for i, val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]

for i in indices:

general\_h.remove(['?', '?', '?', '?', '?', '?'])

return specific\_h, general\_h

s\_final, g\_final = learn(concepts, target)

print("Final S: ", s\_final, sep="\n")

print("Final G: ", g\_final, sep="\n")

**PROGRAM 4**

import numpy as np

import pandas as pd

from pprint import pprint

data = pd.read\_csv("playtennis.csv")

data\_size= len(data)

treenodes = []

tree = ("ROOT": data)

def total\_entropy (data, col):

mvdict = {}

for elem in data[col]:

if elem in mydict.keys():

mydict[elem] += 1

else:

mydict[elem] = 1

total = sum(mydict.values ())

E = 0

for key in mydict.keys ():

E += entropy(mydict[key], total)

return E

def entropy (num, denom):

return -(num/denom)\* np. log2(num/denom)

def get\_sorted\_data(data, column):

sort = {}

for column name in get\_attributes(data, column):

sort [column\_name] = data.loc[data[column]==column\_name]

return sort

def get\_attributes (data, column):

return data [column]. unique() .tolist()

def InfoGain(total\_entropy, sorted data, entropy\_by\_attribute):

length = data\_size

total = 0

for col, df in sorted\_data. items():

total += (len(df) / length) \* entropy\_by\_attribute[col]

return total\_entropy – total

def get\_entropy \_by\_attribute(sorted\_data):

entropies = {}

for key, df in sorted\_data.items():

entropies[key] = total\_entropy(df, 'PlayTennis')

return entropies

def drop\_node(data, column):

return data.drop(column, axis=1)

def id3(tree):

for branch, data in tree.items ():

# Make sure it's a DataFrame

if not isinstance(data, pd.DataFrame):

continue

#Fetch column names so you can use them to iterate later

columns = data.columns

# Calculate the Entropy for the entire dataset

total\_entropy\_for\_data = total\_entropy(data.values, -1)

# If only one column is left, it means we're done.

if len(columns) == 1

break

info\_gain\_list = []

for i in range(0, len(data.columns)-1):

# Sort the rows w.r.t o/p

sorted\_rows = get\_sorted\_data(data, columns[i])

#calculate the entropy w.r.t to each attribute based on sorted columns

entropy-by-attribute = get\_entropy\_by, attribute (sorted rows)

# get the info gain

info\_gain = InfoGain(total\_entropy\_for\_data, sorted\_rows, entropy\_by\_attribute)

# save it

info\_gain\_list.append(info\_gain)

# Find index of max info gain

node = info\_gain\_list.index(max(info gain list))

# sort the data into branches based on the new node

branches = get\_sorted\_data(data, columns [node])

# If we've reached the end of iterations, just assign the value, else drop the sorted column

for attr, df in branches.items () :

if (total\_entropy(df, columns [-1]) == 0):

branches [attr] = df.iloc[0,-1]

else:

branches[attr] = df.drop(columns[node], axis=1)

# Keep track of nodes already done

treenodes.append (columns [node])

# add the new branches to the tree

child = {columns [node]: {}}

tree [branch] = child

tree [branch][columns [node]] = branches

# ID3

id3(tree [branch][columns [node]])

x = id3(tree)

pprint (tree, depth=5)

**PROGRAM 5**

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=7000#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

#bout += np.sum(d\_output, axis=0, keepdims=True)\*lr

wh += X.T.dot(d\_hiddenlayer) \*lr

#bh += np.sum(d\_hiddenlayer, axis=0, keepdims=True)\*lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)

**PROGRAM 6**

import csv

import random

import math

def loadcsv(filename):

lines = csv.reader(open(filename, "r"));

dataset = list(lines)

for i in range(len(dataset)):

#converting strings into numbers for processing

dataset[i] = [float(x) for x in dataset[i]]

return dataset

def splitdataset(dataset, splitratio):

#67% training size

trainsize = int(len(dataset) \* splitratio);

trainset = []

copy = list(dataset);

while len(trainset) < trainsize:

#generate indices for the dataset list randomly to pick ele for training data

index = random.randrange(len(copy));

trainset.append(copy.pop(index))

return [trainset, copy]

def separatebyclass(dataset):

separated = {} #dictionary of classes 1 and 0

#creates a dictionary of classes 1 and 0 where the values are

#the instances belonging to each class

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): #creates a dictionary of classes

summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)];

del summaries[-1] #excluding labels +ve or -ve

return summaries

def summarizebyclass(dataset):

separated = separatebyclass(dataset);

#print(separated)

summaries = {}

for classvalue, instances in separated.items():

#for key,value in dic.items()

#summaries is a dic of tuples(mean,std) for each class value

summaries[classvalue] = summarize(instances) #summarize is used to cal to mean and std

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 = {} # probabilities contains the all prob of all class of test data

for classvalue, classsummaries in summaries.items():#class and attribute information as mean and sd

probabilities[classvalue] = 1

for i in range(len(classsummaries)):

mean, stdev = classsummaries[i] #take mean and sd of every attribute for class 0 and 1 seperaely

x = inputvector[i] #testvector's first attribute

probabilities[classvalue] \*= calculateprobability(x, mean, stdev)

return probabilities

def predict(summaries, inputvector): #training and test data is passed

probabilities = calculateclassprobabilities(summaries, inputvector)

bestLabel, bestProb = None, -1

for classvalue, probability in probabilities.items():#assigns that class which has the highest probability

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 = 'naivedata.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)))

# prepare model

summaries = summarizebyclass(trainingset);

#print(summaries)

# test model

predictions = getpredictions(summaries, testset) #find the predictions of test data with the training data

accuracy = getaccuracy(testset, predictions)

print('Accuracy of the classifier is : {0}%'.format(accuracy))

main()

**PROGRAM 7**

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn import datasets

import pandas as pd

import numpy as np

dataset=load\_iris()

# print(dataset)

X=pd.DataFrame(dataset.data)

X.columns=['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width']

y=pd.DataFrame(dataset.target)

y.columns=['Targets']

# print(X)

model=KMeans(n\_clusters=3)

model.fit(X)

plt.figure(figsize=(14,14))

colormap=np.array(['red','lime','black'])

# REAL PLOT

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')

# K-PLOT

plt.subplot(2,2,2)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[model.labels\_],s=40)

plt.title('KMeans Clustering')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

# GMM PLOT

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 Classification')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

print(‘Observation: The GMM using EM algo based clustering matched true labels more closely than the Kmeans’)

**PROGRAM 8**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn. datasets import load \_iris

data = load\_iris()

df = pd.DataFrame (data. data, columns=data.feature\_names)

df['Class'] = data.target\_names [data.target]

df.head()

x= df.iloc[ : , : -1]. values

y = df.Class. values

print(x[:5])

print(y[:5])

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split (x, y, test\_size = 0.2)

from sklearn.neighbors import KNeighborsClassifier

knn\_classifier = KNeighborsClassifier(n\_neighbors=5)

knn\_classifier.fit(x\_train, y\_train)

predictions = knn\_classifier.predict(x\_test)

print(predictions)

from sklearn.metrics import accuracy\_score, confusion\_matrix

print("Training accuracy Score is : ", accuracy\_score(y\_train, knn\_classifier.predict(x\_train)))

print ("Testing accuracy Score is :", accuracy\_score(y\_test, knn\_classifier.predict(x\_test)))

print ("Training Confusion Matrix is : \n", confusion\_matrix(y\_train, knn\_classifier.predict(x\_train)))

print("Testing Confusion Matrix is : \n", confusion\_matrix(y\_test, knn\_classifier.predict(x\_test)))

**PROGRAM 9**

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(np1.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();

# 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(np1.ones(m))

X = np.hstack((one.T,mbill.T))

#set k here

ypred = localWeightRegression(X,mtip,8)

graphPlot(X,ypred)