Program 1

```
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')
```

Program 3

```
import numpy as np
import pandas as pd
data = pd.DataFrame(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)
print("Final G: ",g_final)
```

Program 2

```
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()
```

Program 4

```
import numpy as np
import pandas as pd
from pprint import pprint
data=pd.read_csv("/content/playtennis.csv"
)
data_size=len(data)
treenodes=[]
tree={"ROOT":data}
def total_entropy(data,col):
mydict={}
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,entrop
y_by_attribute):
length=data_size
total=0
for col,df in sorted_data.items():
total+=(len(df)/length)*entropy_by_attribu
te[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,'PlayTenni
s')
return entropies
def drop_node(data,column):
return data.drop(column,axis=1)
def id3(tree):
for branch,data in tree.items():
if not isinstance(data,pd.DataFrame):
continue
columns=data.columns
total_entropy_for_data=total_entropy(data.
values,-1)
if len(columns)==1:
break
info_gain_list=[]
for i in range(0,len(data.columns)-1):
sorted_rows=get_sorted_data(data,columns
[i])
entropy_by_attribute=get_entropy_by_attri
bute(sorted_rows)
info_gain=InfoGain(total_entropy_for_dat
a,sorted_rows,entropy_by_attribute)
info_gain_list.append(info_gain)
node=info_gain_list.index(max(info_gain_
list))
branches=get_sorted_data(data,columns[no
de])
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)
treenodes.append(columns[node])
child={columns[node]:{}}
tree[branch]=child
tree[branch][columns[node]]=branches
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)
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_neuro
ns,hiddenlayer_neurons))
bh =
np.random.uniform(size=(hiddenlayer_neu
rons))
wout =
np.random.uniform(size=(hiddenlayer_neu
rons,output_neurons))
bout =
np.random.uniform(size=(1,output_neuron
s))
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)+"\n")
print("Actual Output: \n" + str(y)+"\n")
print("Predicted Output: \n",output)
```

Program 9:

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import datasets
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],xma
t,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='r
ed',linewidth=5)
plt.xlabel('Total bill')
plt.ylabel('tip')
plt.show()
data=pd.read_csv('data10_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)
```

Program 6:

```
import csv import random
import pandas as pd 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)
varience=sum([pow(x-avg,2) for x in
numbers])/float(len(numbers)-1)
return math.sqrt(varience)
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(instan
ces) 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 ='DBetes.csv'
splitRatio=0.70
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()
```

Program 7:

```
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')
```

Program 8:

```
import pandas as pd
import numpy as np
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 is :
',confusion_matrix(y_train,knn_classifier.predict(x_train)))
print('Testing Confusion is :
',confusion_matrix(y_test,knn_classifier.predict(x_test)))
```