Labprogram1

```
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': 10,
            'B': 8,
            'C': 5,
            'D': 7,
```

```
            'E': 3,
            'F': 6,
            'G': 5,
            'H': 3,
            'I': 1,
            'J': 0
        }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 1),('H', 7)] ,
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)],

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

LabProgram2

```
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 AQ* 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 given node
    self.status[v]=val

def getHeuristicNodeValue(self,n):
    return self.H.get(n,0) #always rtum the heuristic valve of 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):
    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: # 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 fora start node and backTracking status flag
    print("HEURISTIC VALUES:", self.H)
    print("SOLUTION GRAPH :", self.solutionGraph)
    print("PROCESSING NODE:", v)
    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(graph1,h1,'A')
G2.applyAOStar()
G2.printSolution()
```

Lab 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)
```

LabProgram 4

```
import pandas as pd
import numpy as np

dataset= pd.read_csv('playtennis.csv')
dataset
```

```
def entropy(target_col):
    elements,counts = np.unique(target_col,return_counts = True)
    entropy = np.sum([(-counts[i]/np.sum(counts))*np.log2(counts[i]/np.sum(counts)) for i in range(len(elements))])
    return entropy


def InfoGain(data,split_attribute_name,target_name="PlayTennis"):
    total_entropy = entropy(data[target_name])
    vals,counts= np.unique(data[split_attribute_name],return_counts=True)
    Weighted_Entropy =
np.sum([(counts[i]/np.sum(counts))*entropy(data.where(data[split_attribute_name]==vals[i]).dropna()[target_name]) for i in range(len(vals))])
    InfoGain = total_entropy - Weighted_Entropy
    return InfoGain

def ID3(data,originaldata,features,target_attribute_name="PlayTennis",parent_node_class = None):
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]
    elif len(data)==0:
        return
np.unique(originaldata[target_attribute_name])[np.argmax(np.unique(originaldata[target_attribute_name],return_counts=True)[1])]
    elif len(features) ==0:
        return parent_node_class
    else:
        parent_node_class =
np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attribute_name],return_counts=True)[1])]
        item_values = [InfoGain(data,feature,target_attribute_name) for feature in features] #Return the information gain values for the features in the dataset
        best_feature_index = np.argmax(item_values)
        best_feature = features[best_feature_index]
        tree = {best_feature:{}}
        features = [i for i in features if i != best_feature]
        for value in np.unique(data[best_feature]):
            value = value
            sub_data = data.where(data[best_feature] == value).dropna()
            subtree = ID3(sub_data,dataset,features,target_attribute_name,parent_node_class)
            tree[best_feature][value] = subtree
        return(tree)

tree = ID3(dataset,dataset,dataset.columns[:-1])
print(dataset.head())
print(' \nDisplay Tree\n',tree)
```

LabProgram 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
```

```
#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))


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

LabProgram 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)):
        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,stddev):
    exponent=math.exp(-(math.pow(x-mean,2)/(2*math.pow(stddev,2))))
    return(1/(math.sqrt(2*math.pi)*stddev))*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='5_pima-indians-diabetes.data.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()
```

LabProgram 7

. Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using the k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
```

```
import pandas as pd
import numpy as np
# import some data to play with
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']

# Build the K Means Model
model = KMeans(n_clusters=3)
model.fit(X) # model.labels


plt.figure(figsize=(14,14))
colormap = np.array(['red', 'lime', 'black'])
# Plot the Original Classifications using Petal features
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')
# Plot the Models Classifications
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')

# General EM for GMM
from sklearn import preprocessing
# transform your data such that its distribution will have a
# mean value 0 and standard deviation of 1.
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.')
```

LabProgram 8

```
#Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print
#both correct and wrong predictions. Java/Python ML library classes can be used for this problem.


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)
print(df)

df['Class'] = data.target_names[data.target]
x = df.iloc[:, :-1].values
y = df.Class.values

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=7)
knn_classifier.fit(x_train, y_train)
predictions = knn_classifier.predict(x_test)
print(y_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)))
```

LabProgram 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(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('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)
```