1 Implement A* Search 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')
```

2. Implement AO* Search algorithm.

```
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  consider heuristic distances given
print ("Graph")
h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
graph = {
   '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(graph, h1, 'A')
G1.applyAOStar()
G1.printSolution()
```

3. For a given set of training data examples stored in a .CSV file, implement and demonstrate the

Candidate-Elimination algorithmto output a description of the set of all hypotheses consistent with the training examples.

```
'''For a given set of training data examples stored in a .CSV file, implement
and demonstrate the Candidate-Elimination algorithm to output a description of
the set of all hypotheses consistent with the training examples.'''

import csv
file=open('lab3ds.csv')
data=list(csv.reader(file))[1:]
concepts=[]
target=[]
for i in data:
    concepts.append(i[:-1])
    target.append(i[-1])
specific_h=['0']*len(concepts[0])
general_h= [['?' for i in range(len(specific_h))] for i in range(len(specific_h))]
for i,instance in enumerate(concepts):
    if target[i]=="Yes":
        for x in range(len(specific_h)):
            if specific_h[x]== '0':
                specific_h[x]=instance[x]
            elif instance[x]!=specific_h[x]:
                specific_h[x]='?'
                general_h[x][x] = '?'
    if target[i]=="No":
        for x in range(len(specific_h)):
            if instance[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(['?','?','?','?','?','?'])
print("Final Specific:",specific_h,sep="\n")
print("Final General:",general_h,sep="\n")
```

4. Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge toclassify a new sample.

```
def find_entropy(df):
    Class = df.keys()[-1]   #To make the code generic, changing target variable class name
    entropy = 0
    values = df[Class].unique()
    for value in values:
        fraction = df[Class].value_counts()[value]/len(df[Class])
        entropy += -fraction*np.log2(fraction)
    return entropy
def find_entropy_attribute(df,attribute):
    Class = df.keys()[-1]   #To make the code generic, changing target variable class name
    target_variables = df[Class].unique()  #This gives all 'Yes' and 'No'
```

```
    variables = df[attribute].unique()    #This gives different features in that attribute (like 'Hot','Cold' in
Temperature)
    entropy2 = 0
    for variable in variables:
        entropy = 0
        for target_variable in target_variables:
            num = len(df[attribute][df[attribute]==variable][df[Class] ==target_variable])
            den = len(df[attribute][df[attribute]==variable])
            fraction = num/(den+eps)
            entropy += -fraction*log(fraction+eps)
        fraction2 = den/len(df)
        entropy2 += -fraction2*entropy
    return abs(entropy2)
def find_winner(df):

    IG = []
    for key in df.keys()[:-1]:#         Entropy_att.append(find_entropy_attribute(df,key))
        IG.append(find_entropy(df)-find_entropy_attribute(df,key))
    return df.keys()[:-1][np.argmax(IG)]
def get_subtable(df, node,value):
    return df[df[node] == value].reset_index(drop=True)
def buildTree(df,tree=None):
   #To make the code generic, changing target variable class name  #Here we build our decision tree  #Get
attribute with maximum information gain
    node = find_winner(df)#Get distinct value of that attribute e.g Salary is node and Low,Med and High are
values
    attValue = np.unique(

        df[node])#Create an empty dictionary to create tree
    if tree is None:
        tree={}
        tree[node] = {}#We make loop to construct a tree by calling this function recursively. #In this we check
if the subset is pure and stops if it is pure.
    for value in attValue:
        subtable = get_subtable(df,node,value)
        clValue,counts = np.unique(subtable['play'],return_counts=True)
        if len(counts)==1:#Checking purity of subset
            tree[node][value] = clValue[0]
        else:
            tree[node][value] = buildTree(subtable) #Calling the function recursively
    return tree
import pandas as pd
import numpy as np
eps = np.finfo(float).eps
from numpy import log2 as log
df = pd.read_csv('tennis.csv')
print("\n Given Play Tennis Data Set:\n\n",df)
tree= buildTree(df)
import pprint
pprint.pprint(tree)

test={'Outlook':'Sunny','Temperature':'Hot','Humidity':'High','Wind':'Weak'}
def func(test, tree, default=None):
    attribute = next(iter(tree))
    print(attribute)
```

```
    if test[attribute] in tree[attribute].keys():
        print(tree[attribute].keys())
        print(test[attribute])
        result = tree[attribute][test[attribute]]
        if isinstance(result, dict):
            return func(test, result)
        else:
            return result
    else:
        return default
ans = func(test, tree)
print(ans)
```

5. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
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=10 #Setting training iterations
lr=0.01 #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
    wh += X.T.dot(d_hiddenlayer) *lr

    print ("-----------Epoch-", i+1, "Starts----------")
    print("Input: \n" + str(X))
    print("Actual Output: \n" + str(y))
    print("Predicted Output: \n" ,output)
    print ("-----------Epoch-", i+1, "Ends----------\n")

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

6 Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# Load Data from CSV
data = pd.read_csv('tennis.csv')
print("The first 5 Values of data is :\n", data.head())

# obtain training attributes
X = data.iloc[:, :-1]
print("\nThe First 5 values of the train attributes is\n", X.head())

# obtain training labels or target values
Y = data.iloc[:, -1]
print("\nThe First 5 values of target values is\n", Y.head())

# convert categorical values into numbers
obj1= LabelEncoder()
X.Outlook = obj1.fit_transform(X.Outlook)
print("\n The Encoded and Transformed Data in Outlook \n",X.Outlook)

obj2 = LabelEncoder()
X.Temperature = obj2.fit_transform(X.Temperature)

obj3 = LabelEncoder()
X.Humidity = obj3.fit_transform(X.Humidity)

obj4 = LabelEncoder()
X.Wind = obj4.fit_transform(X.Wind)
```

```
print("\n The Encoded and Transformed Training Examples \n", X.head())

obj5 = LabelEncoder()
Y = obj5.fit_transform(Y)
print("The class Label encoded in numerical form is",Y)

# Create the training and test data from the original data set.

X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.20)

#Training the classification Model using Gaussian Naive Bayes
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, Y_train)

from sklearn.metrics import accuracy_score
print("Accuracy is:", accuracy_score(classifier.predict(X_test), Y_test))
```

7. Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using 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.

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

dataset=load_iris()
print("\n IRIS Dataset:\n", dataset.data)
print("\n IRIS Features:\n", dataset.feature_names)
print("\n IRIS Target:\n", dataset.target)
print("\n IRIS Target:\n", dataset.target_names)

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

plt.figure(figsize=(8,5))
colormap=np.array(['red','lime','blue'])

# Plotting without clustering
plt.subplot(1,3,1)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.Targets],s=20)
plt.title('Before Clustering')

# Plotting with K-Means Clustering
plt.subplot(1,3,2)
```

```
model=KMeans(n_clusters=3)
model.fit(X)
predY=np.choose(model.labels_,[0,1,2]).astype(np.int64)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[predY],s=20)
plt.title('KMeans Clustering')

# Plotting with GMM using EM Clustering
scaler=preprocessing.StandardScaler()
scaler.fit(X)
xsa=scaler.transform(X)
xs=pd.DataFrame(xsa,columns=X.columns)
gmm=GaussianMixture(n_components=3)
gmm.fit(xs)

y_cluster_gmm=gmm.predict(xs)
plt.subplot(1,3,3)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y_cluster_gmm],s=20)
plt.title('GMM with EM Clustering')
```

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
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
import matplotlib.pyplot as plt

assigned_names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

# Read dataset to pandas dataframe
dataset = pd.read_csv("iris2.csv", names=assigned_names)
X = dataset.iloc[:, :-1]
y = dataset.iloc[:, -1]
print(X.head())

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.10)

classifier = KNeighborsClassifier(n_neighbors=5).fit(Xtrain, ytrain)

ypred = classifier.predict(Xtest)

i = 0
print ("\n-------------------------------------------------------------------------")
print ('%-25s %-25s %-25s' % ('Original Label', 'Predicted Label', 'Correct/Wrong'))
print ("-------------------------------------------------------------------------")
for label in ytest:
    print ('%-25s %-25s' % (label, ypred[i]), end="")
    if (label == ypred[i]):
        print (' %-25s' % ('Correct'))
    else:
        print (' %-25s' % ('Wrong'))
    i = i + 1
```

```
print ("-----------------------------------------------------------------------")
print("\nConfusion Matrix:\n",metrics.confusion_matrix(ytest, ypred))
print ("-----------------------------------------------------------------------")
print("\nClassification Report:\n",metrics.classification_report(ytest, ypred))
print ("-----------------------------------------------------------------------")
print('Accuracy of the classifer is %0.2f' % metrics.accuracy_score(ytest,ypred))
print ("-----------------------------------------------------------------------")
plt.plot(Xtest,ytest,'ro')
plt.plot(Xtest,ytest,'b+')
```

9. Implement the non-parametric Locally Weighted Regressionalgorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs

```
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

# load data points
data = pd.read_csv('tips.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)

#preparing the data
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))

#set k here
ypred = localWeightRegression(X,mtip,0.5)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0]
```

```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(bill,tip, color='yellow')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'black', linewidth=2)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show();
```