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
   #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 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': 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': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}
aStarAlgo('A', 'J')
```

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 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()
```

3. 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 numpy as np

import pandas as pd


data = pd.read_csv('/content/sample_data/enjoysports.csv')

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

print("\nInstances are:\n",concepts)

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

print("\nTarget Values are: ",target)


def learn(concepts, target):

  specific_h = concepts[0].copy()

  print("\nInitialization of specific_h and genearal_h")

  print("\nSpecific Boundary: ", specific_h)

  general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]

  print("\nGeneric Boundary: ",general_h)


  for i, h in enumerate(concepts):

    print("\nInstance", i+1 , "is ", h)

    if target[i] == "yes":

      print("Instance is Positive ")

      for x in range(len(specific_h)):

        if h[x]!= specific_h[x]:

          specific_h[x] ='?'

          general_h[x][x] ='?'


    if target[i] == "no":

      print("Instance is Negative ")

      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] = '?'


    print("Specific Bundary after ", i+1, "Instance is ", specific_h)

    print("Generic Boundary after ", i+1, "Instance is ", general_h)

    print("\n")
```

```
    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 Specific_h: ", s_final, sep="\n")
print("Final General_h: ", g_final, 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 to classify a new sample.

```
def find_entropy(df):
    Class = df.keys()[-1]
    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]
    target_variables = df[Class].unique()
    variables = df[attribute].unique()
    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):
```

```
    Entropy_att = []
    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):
    Class = df.keys()[-1]
    node = find_winner(df)
    attValue = np.unique(df[node])

    if tree is None:
        tree = {}
        tree[node] = {}

    for value in attValue:
        subtable = get_subtable(df, node, value)
        c1Value, counts = np.unique(subtable['play'], return_counts = True)

        if len(counts) == 1:
            tree[node][value] = c1Value[0]
        else:
            tree[node][value] = buildTree(subtable)
    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('id3.csv')
print("\nGiven Play Tennis Data Set: \n\n", df)

tree = buildTree(df)
import pprint
pprint.pprint(tree)
```

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)

y = y/100

def sigmoid (x):

 return 1/(1 + np.exp(-x))

def derivatives_sigmoid(x):

 return x * (1 - x)

epoch=5000

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)
```

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 numpy as np
import csv

def read_data(filename):
    with open(filename, 'r') as csvfile:
        datareader = csv.reader(csvfile)
        metadata = next(datareader)
        traindata = []
        for row in datareader:
            traindata.append(row)
    print(metadata)
    print(traindata)
    return (metadata, traindata)

def splitDataset(dataset, splitRatio):
    trainsize = int(len(dataset) * splitRatio)
    trainset = []
    testset = list(dataset)
    i = 0

    while len(trainset) < trainsize:
        trainset.append(testset.pop(i))
    return [trainset, testset]

def classify(data, test):
    total_size = data.shape[0]
    print("Trainig data size: ", total_size)
    print("Test data size: ", test.shape[0])
    countYes = 0
    countNo = 0
    probyes = 0
    probNo = 0
    print('Target count probability: ')
```

```
for x in range(data.shape[0]):
    if data[x, data.shape[1] - 1] == 'yes':
        countYes += 1
    if data[x, data.shape[1] - 1] == 'no':
        countNo += 1
probYes = countYes / total_size
probNo = countNo / total_size

print('Yes ', countYes, ' ', probYes)
print('No ', countNo, ' ', probNo)

prob0 = np.zeros((test.shape[1] - 1))
prob1 = np.zeros((test.shape[1] - 1))

accuracy = 0
print('Instance prediction target: ')

for t in range(test.shape[0]):
    for k in range(test.shape[1] - 1):
        count1 = count0 = 0
        for j in range(data.shape[0]):
            if test[t, k] == data[j, k] and data[j, data.shape[1] - 1] == 'no':
                count0 += 1
            if test[t, k] == data[j, k] and data[j, data.shape[1] - 1] == 'yes':
                count0 += 1
        prob0[k] = count0 / countNo
        prob1[k] = count1 / countYes
    probno = probyes = 0
    for i in range(test.shape[1] - 1):
        probno = probno * prob0[i]
        probyes = probyes * prob1[i]
    if probno > probyes:
        predict = 'no'
    else:
        predict = 'yes'
    print(t + 1, ' ', predict, ' ', test[t, test.shape[1] - 1])
```

```
        if predict == test[t, test.shape[1] - 1]:
            accuracy += 1

    find_accuracy = (accuracy / test.shape[0]) * 100
    print("Accuracy ", find_accuracy, ' %')
    return
metadata, traindata = read_data("tennis.csv")
splitRatio = 0.7
trainingset, testset = splitDataset(traindata, splitRatio)
training = np.array(trainingset)
testing = np.array(testset)
classify(training, testing)
```

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.

```
#EM K-Means Algorithm
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
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)
labels_ = model.fit(X)
plt.figure(figsize=(14,7))
colormap = np.array(['red', 'lime', 'black'])

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

```
plt.show()

plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[model.labels_], s = 40)
plt.title('K Mean Classification')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
print('The accuracy score of K Mean: ', sm.accuracy_score(y, model.labels_))
print('The Confusion matrix of K Mean: ', sm.confusion_matrix(y, model.labels_))
plt.show()

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)
y_gmm = gmm.predict(xs)

plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[y_gmm], s = 40)
plt.title('GMM Classification')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
print('The accuracy score of EM: ', sm.accuracy_score(y, y_gmm))
print('The Confusion matrix of EM: ', sm.confusion_matrix(y, y_gmm))
plt.show()
```

8. Write a program to implement the 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.

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
iris = datasets.load_iris()
print("iris dataset 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 training data and its label", x_train.shape, y_train.shape)
print("Size of testing 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 KNN 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))
```

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

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
data = pd.read_csv("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))

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()

X = np.hstack((one.T, mbill.T))
ypred = localweightregression(X, mtip, 8)
graphplot(X, ypred)
```