```

# (1). Implement A* Search algorithm.

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
```

```python

def heuristic(n):
    H_dist = {
        'A': 2,
        'B': 6,
        'C': 2,
        'D': 3,
        'S': 4,
        'G': 0,
    }
    return H_dist[n]


Graph_nodes = {
    'A': [('B', 3), ('C', 1)],
    'B': [('D', 3)],
    'C': [('D', 1), ('G', 5)],
    'D': [('G', 3)],
    'S': [('A', 1)],
    'G': []
}
aStarAlgo('S', 'G')
aStarAlgo('A', 'B')
aStarAlgo('B', 'S')

'''
---Output---
Path found: ['S', 'A', 'C', 'D', 'G']
Path found: ['A', 'B']
Path does not exist!
'''
```

```python

# (2). Implement AO* Search algorithm.


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]
```

```python
    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': 38, 'B': 17, 'C': 9, 'D': 27, 'E': 5, 'F': 10, 'G': 3,
      'H': 4,'I': 15, 'J': 10}

graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('E', 1)], [('F', 1)]],
    'C': [[('G', 1)], [('H', 1)]],
    'D': [[('I', 1), ('J', 1)]]
}

G1 = Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()

print("HEURISTIC VALUES :", G1.H)
print("SOLUTION GRAPH :", G1.solutionGraph)

print('status:', G1.status)
print('parent:', G1.parent)
```

(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 allhypotheses consistent with the training examples.

```
import numpy as np
import pandas as pd

data = pd.read_csv('data1.csv')

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

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


def learn(concepts, target):

    specific_h = concepts[0].copy()
    print('initialization of specific_h and general_h')
    print(specific_h)

    general_h = [['?' for i in range(len(specific_h))] for i in
                                                            range(len(specific_h))]
    print(general_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] = '?'
                print(specific_h)
            print(specific_h)

        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] = '?'

        print('steps of candidate Elimation Algorithm ', i + 1)
        print(specific_h)
        print(general_h)

    indeces = [i for i, val in enumerate(general_h) if val ==
                                                    ['?', '?', '?', '?', '?', '?']]

    for i in indeces:
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    return specific_h, general_h


s_final, g_final = learn(concepts, target)
print('----------------final answer----------------\n')
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.

```
import math
import pandas as pd
from pprint import pprint
from collections import Counter

def entropy(probs):
    return sum([-prob * math.log(prob, 2) for prob in probs])


def entropy_list(a_list):

    cnt = Counter(x for x in a_list)
    num_instance = len(a_list) * 1.0
    probs = [x / num_instance for x in cnt.values()]
    return entropy(probs)


def info_gain(df, split, target, trace=0):

    df_split = df.groupby(split)
    nobs = len(df.index) * 1.0
    df_agg_ent = df_split.agg({target: [entropy_list, lambda x: len(x) / nobs]})
    df_agg_ent.columns = ["entropy", "propObserved"]

    new_entropy = sum(df_agg_ent["entropy"] * df_agg_ent["propObserved"])
    old_entropy = entropy_list(df[target])
    return old_entropy - new_entropy


def id3(df, target, attribute_name, default_class=None):

    cnt = Counter(x for x in df[target])
    if len(cnt) == 1:
        return next(iter(cnt))

    elif df.empty or (not attribute_name):
        return default_class

    else:
        default_class = max(cnt.keys())
        gains = [info_gain(df, attr, target) for attr in attribute_name]
        index_max = gains.index(max(gains))
        best_attr = attribute_name[index_max]
        tree = {best_attr: {}}
        remaining_attr = [x for x in attribute_name if x != best_attr]

        for attr_val, data_subset in df.groupby(best_attr):
            subtree = id3(data_subset, target, remaining_attr, default_class)
            tree[best_attr][attr_val] = subtree

        return tree


def classify(instance, tree, default=None):
    attribute = next(iter(tree))
    if instance[attribute] in tree[attribute].keys():
        result = tree[attribute][instance[attribute]]
        if isinstance(result, dict):
            return classify(instance, result)
        else:
            return result
    else:
        return default
```

```
df_tennis = pd.read_csv('id3.csv')
print(df_tennis)

attribute_names = list(df_tennis.columns)
attribute_names.remove('PlayTennis')

tree = id3(df_tennis, 'PlayTennis', attribute_names)

print('\n\n The resultant decision tree is: \n\n')
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

input_neurons = 2
hidden_layer_neurons = 2
output_neurons = 2

input_ = np.random.randint(1, 100, input_neurons)
output = np.array([1.0, 0.0])

hidden_biass = np.random.rand(1, hidden_layer_neurons)
output_biass = np.random.rand(1, output_neurons)
hidden_weight = np.random.rand(input_neurons, hidden_layer_neurons)
output_weight = np.random.rand(hidden_layer_neurons, output_neurons)


def sigmoid(layer):
    return 1 / (1 + np.exp(-layer))


def gradient(layer):
    return layer * (1 - layer)


for i in range(2000):

    hidden_layer = np.dot(input_, hidden_weight)
    hidden_layer = sigmoid(hidden_layer + hidden_biass)

    output_layer = np.dot(hidden_layer, output_weight)
    output_layer = sigmoid(output_layer + output_biass)

    error = (output - output_layer)
    gradient_outputLayer = gradient(output_layer)

    error_terms_output = gradient_outputLayer * error
    error_terms_hidden = gradient(hidden_layer) *
                                    np.dot(error_terms_output, output_weight.T)

    gradient_hidden_weights = np.dot(input_.reshape(input_neurons, 1),
                            error_terms_hidden.reshape(1, hidden_layer_neurons))
    gradient_output_weights = np.dot(hidden_layer.reshape(hidden_layer_neurons, 1),
                                  error_terms_output.reshape(1, output_neurons))

    hidden_weight = hidden_weight + 0.05 * gradient_hidden_weights
    output_weight = output_weight + 0.05 * gradient_output_weights

    print('**********************************')
    print('Iteration: ', i, ':::', error)
    print('####- output - #####', output_layer)
```

(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.

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics

df = pd.read_csv("pima_indian.csv")
feature_col_names = ['num_preg', 'glucose_conc', 'diastolic_bp', 'thickness',
                     'insulin', 'bmi', 'diab_pred', 'age']
predicted_class_names = ['diabetes']

X = df[feature_col_names].values
y = df[predicted_class_names].values

xtrain, xtest, ytrain, ytest = train_test_split(X, y, test_size=0.33)

print('\n the total number of Training Data :', ytrain.shape)
print('\n the total number of Test Data :', ytest.shape)

clf = GaussianNB().fit(xtrain, ytrain.ravel())
predicted = clf.predict(xtest)

predictTestData = clf.predict([[1, 189, 60, 23, 846, 30.1, 0.398, 59]])

print('\n Confusion matrix')
print(metrics.confusion_matrix(ytest, predicted))

print('Accuracy of the classifier is', metrics.accuracy_score(ytest, predicted))
print('The value of Precision', metrics.precision_score(ytest, predicted))
print('The value of Recall', metrics.recall_score(ytest, predicted))

print("Predicted Value for individual Test Data:", predictTestData)
```

(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.

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

names = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width', 'Class']

dataset = pd.read_csv("Kdataset.csv", names=names)

X = dataset.iloc[:, :-1]

label = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica': 2}

y = [label[c] for c in dataset.iloc[:, -1]]

plt.figure(figsize=(14, 7))
colormap = np.array(['red', 'lime', 'black'])

# REAL PLOT

plt.subplot(1, 3, 1)
plt.title('Real')
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y])

# K-PLOT

model = KMeans(n_clusters=3, random_state=0).fit(X)
plt.subplot(1, 3, 2)
plt.title('KMeans')
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_])

print('The accuracy score K-Mean: ', metrics.accuracy_score(y, model.labels_))
print('The Confusion matrix K-Mean:\n', metrics.confusion_matrix(y, model.labels_))

# GMM PLOT

gmm = GaussianMixture(n_components=3, random_state=0).fit(X)
y_cluster_gmm = gmm.predict(X)

plt.subplot(1, 3, 3)
plt.title('GMM Classification')
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm])

print('The accuracy score of EM: ', metrics.accuracy_score(y, y_cluster_gmm))
print('The Confusion matrix of EM:\n ', metrics.confusion_matrix(y, y_cluster_gmm))

plt.show()
```

```
(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 pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

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

dataset = pd.read_csv('Kdataset.csv')
X = dataset.iloc[:, :-1]
y = dataset.iloc[:, -1]

print('sepal-length', 'sepal-width', 'petal-length', 'petal-width')
print(X.head())
print('Target value')
print(y.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)

print("\n-----------------------------------------------------------------------")
print('%-25s %-25s %-25s' % ('Original Label', 'Predicted Label', 'Correct/Wrong'))
print("-----------------------------------------------------------------------")

i = 0
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("-----------------------------------------------------------------------")
```

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

```
import numpy as np
import numpy as np1
import pandas as pd
import matplotlib.pyplot as plt


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


# 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, 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='green')
ax.plot(xsort[:, 1], ypred[SortIndex], color='red', linewidth=5)

plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show()
```