A star search:

```
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 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 exit!')
            return None
        if n==stop_node:
            path=[]
            while parents[n]!=n:
                path.append(n)
                n=parents[n]
            path.append(start_node)
            path.reverse()
```

```python
            print('path found:{}'.format(path))
            return path
        open_set.remove(n)
        closed_set.add(n)
    print('path does not exit!')
    return None
def get_neighbors(v):
    if v in Graph_nodes:
        return  Graph_nodes[v]
    else:
        return None
def heuristic(n):
    H_dist={
        'A':5,
        'B':4,
        'C':4,
        'D':0,
    }
    return H_dist[n]
Graph_nodes={
    'A': [('B',2),('C',3)],
    'B': [('A', 2), ('D', 8)],
    'C': [('A', 3), ('D', 1)],
    'D': [('B', 8), ('C', 1)]
}
aStarAlgo('A','D')
```

CANDIDATE ELIMINATION ALGORITHM:

```python
import csv

with open("trainingexamples.csv") as f:
    csv_file = csv.reader(f)
    data = list(csv_file)

    specific = data[0][:-1]
    general = [['?' for i in range(len(specific))] for j in range(len(specific))]

    for i in data:
        if i[-1] == "Yes":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    specific[j] = "?"
                    general[j][j] = "?"

        elif i[-1] == "No":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    general[j][j] = specific[j]
                else:
                    general[j][j] = "?"

        print("\nStep " + str(data.index(i)+1) + " of Candidate Elimination Algorithm")
        print(specific)
        print(general)

    gh = [] # gh = general Hypothesis
    for i in general:
        for j in i:
            if j != '?':
                gh.append(i)
                break
    print("\nFinal Specific hypothesis:\n", specific)
    print("\nFinal General hypothesis:\n", gh)
```

ID3 ALGORITHM :

```
import pandas as pd
"""
Pandas is a Python library used for working with data
sets. It has functions for analyzing, cleaning,
exploring,
and manipulating data.
"""
from pprint import pprint
"""The pprint module provides a capability to “pretty-
print” arbitrary Python data
structures in a form which can be used as input to the
interpreter"""
from sklearn.feature_selection import
mutual_info_classif
from collections import Counter
"It is a subclass of dict that is designed to count the
occurrences of elements in a collection."

def id3(df, target_attribute, attribute_names,
default_class=None):
    cnt = Counter(x for x in df[target_attribute])
    if len(cnt) == 1:
        return next(iter(cnt))

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

    else:
        gainz =
mutual_info_classif(df[attribute_names],
df[target_attribute], discrete_features=True)
        index_of_max = gainz.tolist().index(max(gainz))
        best_attr = attribute_names[index_of_max]
        tree = {best_attr: {}}
        remaining_attribute_names = [i for i in
attribute_names if i != best_attr]
```

```
        for attr_val, data_subset in
df.groupby(best_attr):
            subtree = id3(data_subset,
target_attribute, remaining_attribute_names,
default_class)
            tree[best_attr][attr_val] = subtree

        return tree


df = pd.read_csv("p-tennis.csv")

attribute_names = df.columns.tolist()
print("List of attribute names")

attribute_names.remove("PlayTennis")

for colname in df.select_dtypes("object"):
    df[colname], _ = df[colname].factorize()
"""This method is useful for obtaining a numeric
representation of an array when all that matters
is identifying distinct values. """

print(df)

tree = id3(df, "PlayTennis", attribute_names)
print("The tree structure")
pprint(tree)
```

ANN ALGORITHM :

```
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 = 5000  # 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))  # 2,3
bh = np.random.uniform(size=(1, hiddenlayer_neurons))
# 1,3
wout = np.random.uniform(size=(hiddenlayer_neurons,
output_neurons))  # 3,1
bout = np.random.uniform(size=(1, output_neurons))  #
1,1
```

```
for i in range(epoch):
    # Forward Propogation
    hinp = np.dot(X, wh) + bh
    hlayer_act = sigmoid(hinp)  # HIDDEN LAYER
ACTIVATION FUNCTION
    outinp = np.dot(hlayer_act, wout) + bout
    output = sigmoid(outinp)

    outgrad = derivatives_sigmoid(output)
    hiddengrad = derivatives_sigmoid(hlayer_act)

    EO = y - output  # ERROR AT OUTPUT LAYER
    d_output = EO * outgrad

    EH = d_output.dot(wout.T)  # ERROR AT HIDDEN LAYER
(TRANSPOSE => COZ REVERSE(BACK))
    d_hiddenlayer = EH * hiddengrad

    wout += hlayer_act.T.dot(d_output) * lr  # REMEMBER
WOUT IS 3*1 MATRIX
    wh += X.T.dot(d_hiddenlayer) * lr

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

NAÏVE BAYES :

```
import csv
import random
import math

"Spilt the Dataset into Training and Testing Data"
"Train Data = 691, Test Data = 77 rows "
"Compare the Actual and prediction Values for the
algorithm, then Come up with an accuracy "

def loadcsv(filename):
    dataset = list(csv.reader(open(filename,"r")))
    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 = []
    trainSet,testSet =
dataset[:trainSize],dataset[trainSize:]
    return [trainSet, testSet]

def mean(numbers):
    return sum(numbers)/(len(numbers))

def stdev(numbers):
    avg = mean(numbers)
    v = 0
    for x in numbers:
        v += (x-avg)**2
    return math.sqrt(v/(len(numbers)-1))

"For Each of the Attribute in Train Set, it will
calculate Mean and SD that is the Summaries"
def summarizeByClass(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)
    summaries = {}
    for classValue, instances in separated.items():
        summaries[classValue] = [(mean(attribute),
stdev(attribute)) for attribute in zip(*instances)][:-
1]
    return summaries

def calculateProbability(x, mean, stdev):
    exponent = math.exp((-(x-mean)**2)/(2*(stdev**2)))
    return (1 / math.sqrt(2*math.pi*(stdev**2))) *
exponent

def predict(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in
summaries.items():
        probabilities[classValue] = 1
        "Assume Class Value to be 1 initially"
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *=
calculateProbability(x, mean, stdev)
            "Takes mean and SD of each attribute to
calculate the probability"
    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/(len(testSet))) * 100.0

filename = 'diabetes2.csv'
splitRatio = 0.9
dataset = loadcsv(filename)
actual = []
trainingSet, testSet = splitDataset(dataset,
splitRatio)
for i in range(len(testSet)):
    vector = testSet[i]
    actual.append(vector[-1])
print('Split {0} rows into train={1} and test={2}
rows'.format(len(dataset), len(trainingSet),
len(testSet)))
summaries = summarizeByClass(trainingSet) #will have
(mean,sd) for all attributes.(for class 1 & 0
separately)
predictions = getPredictions(summaries, testSet)
print('\nActual values:\n',actual)
print("\nPredictions:\n",predictions)
accuracy = getAccuracy(testSet, predictions)
print("Accuracy",accuracy)
```

K MEANS AND EM ALGORITHM :

```
import matplotlib.pyplot as plt
"For plotting Graphs"
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import sklearn.metrics as sm
"Metrics to measure accuracy"
import pandas as pd
"For Processing of Dataframes"
import numpy as np
"For Processing of Arrays"

iris = datasets.load_iris()

X = pd.DataFrame(iris.data)
"Converted into Pandas Data frame becoz pandas allows
us to name the columns and we can access using those
names only"
X.columns =
['Sepal_Length','Sepal_Width','Petal_Length','Petal_Wid
th']
Y = pd.DataFrame(iris.target)
Y.columns = ['Targets']

"Target would be 0,1 and 2 that is Setosa, Versicolor
and Verginica"

print(X)
print(Y)
colormap = np.array(['red', 'lime', 'black'])
"red for Setosa,lime for Versicolor and black for
Verginica"

plt.subplot(1,2,1)
plt.scatter(X.Petal_Length, X.Petal_Width,
c=colormap[Y.Targets], s=40)
"Here you are creating References for colormap using
Targets that is 0 for red,1 for lime and 2 for black"
```

```
plt.title('Real Clustering')

model1 = KMeans(n_clusters=3)
model1.fit(X)

plt.subplot(1,2,2)
plt.scatter(X.Petal_Length, X.Petal_Width,
c=colormap[model1.labels_], s=40)
plt.title('K Mean Clustering')
plt.show()

model2 = GaussianMixture(n_components=3)
model2.fit(X)

plt.subplot(1,2,1)
plt.scatter(X.Petal_Length, X.Petal_Width,
c=colormap[model2.predict(X)], s=40)
plt.title('EM Clustering')
plt.show()

print("Actual Target is:\n", iris.target)
print("K Means:\n",model1.labels_)
print("EM:\n",model2.predict(X))
print("Accuracy of KMeans is
",sm.accuracy_score(Y,model1.labels_))
print("Accuracy of EM is ",sm.accuracy_score(Y,
model2.predict(X)))
```

KNN ALGORITHM :

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.model_selection import train_test_split

iris_dataset=load_iris()

#display the iris dataset
print("\n IRIS FEATURES \ TARGET NAMES: \n ",
iris_dataset.target_names)
for i in range(len(iris_dataset.target_names)):

print("\n[{0}]:[{1}]".format(i,iris_dataset.target_name
s[i]))

print("\n IRIS DATA :\n",iris_dataset["data"])

#split the data into training and testing data
X_train, X_test, y_train, y_test =
train_test_split(iris_dataset["data"],
iris_dataset["target"], random_state=0)

print("\n Target :\n",iris_dataset["target"])
#print("\n")
#print(len(iris_dataset["target"]))
print("\n X TRAIN \n", X_train)
print("\n X TEST \n", X_test)
print("\n Y TRAIN \n", y_train)
print("\n Y TEST \n", y_test)

#train and fit the model
kn = KNeighborsClassifier(n_neighbors=5)
```

```
kn.fit(X_train, y_train)

#predicting from model
x_new = np.array([[5, 2.9, 1, 0.2]])
print("\n XNEW \n",x_new)
prediction = kn.predict(x_new)
print("\n Predicted target value:
{}\n".format(prediction))
print("\n Predicted feature name:
{}\n".format(iris_dataset["target_names"][prediction]))

i=1
x= X_test[i]
x_new = np.array([x])
print("\n XNEW \n",x_new)

for i in range(len(X_test)):
  x = X_test[i]
  x_new = np.array([x])
  prediction = kn.predict(x_new)      #predict method
returns label
  print("\n Actual : {0} {1}, Predicted
:{2}{3}".format(y_test[i],0,prediction,iris_dataset["ta
rget_names"][ prediction]))
print("\n TEST SCORE[ACCURACY]:
{:.2f}\n".format(kn.score(X_test, y_test)))
```

REGRESSION :

```
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 and add 1 in bill
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='green')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red',
linewidth=5)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show();
```

AO STAR SEARCH :

```
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)  # GET IS
INBUILT,RETURNS VALUE OF THE KEY. IF KEY NOT PRESENT
THEN RETURN "SECOND PARAMETER"

    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(FOR START NODE, STATUS
WILL BE RETURNED AS 0)
            minimumCost, childNodeList =
self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v, len(childNodeList))  #
THEN STATUS KEEPS UPDATING (HOW MANY TO VISIT(NO OF
CHILDREN))
```

```
            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)  # THIS IS WHAT
SETS THE TERMINATING CONDITION
                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
```

```
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}  # Heuristic values of Nodes
graph2 = {  # Graph of Nodes and Edges
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],  # Neighbors of Node 'A', B, C & D with repective weights
    'B': [[('G', 1)], [('H', 1)]],  # Neighbors are included in a list of lists
    'D': [[('E', 1), ('F', 1)]]  # Each sublist indicate a "OR" node or "AND" nodes
}

G2 = Graph(graph2, h2, 'A')  # Instantiate Graph object with graph, heuristic values and start Node
G2.applyAOStar()  # Run the AO* algorithm
G2.printSolution()  # Print the solution graph as output of the AO* algorithm search
```