1. A STAR

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)
    closed_set = set()
    g = {} #store distance from starting node
    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 == 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

def heuristic(n):
    H_dist = {
    'A': 11,
    'B': 6,
    'C': 99,
    'D': 1,
    'E': 7,
    'G': 0,
   }
    return H_dist[n]

Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
   }

aStarAlgo('A', 'G')
```

2. AO STAR SEARCH

```
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): # 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

    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=-1
        costToChildNodeList=[]
        for nodeInfoTupleList in self.getNeighbors(v):
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)
            if minimumCost==-1 or minimumCost>cost:
```

```
                minimumCost=cost
                costToChildNodeList=nodeList
        return minimumCost, costToChildNodeList

    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: # 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:
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList

            if v!=self.start:
                self.aoStar(self.parent[v], True)

            if backTracking==False: # check the current call is not for backtracking
                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}
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.CANDIDATE ELIMINATION : (NUMPY AND PANDAS)

```
import numpy as np
import pandas as pd


# Loading Data from a CSV File
data = pd.DataFrame(pd.read_csv('finds.csv'))



# Separating concept features from Target
concepts = np.array(data.iloc[:,0:-1])


# Isolating target into a separate DataFrame
target = np.array(data.iloc[:,-1])
def learn(concepts, target):
    specific_h=[0,0,0,0,0,0]
    print ('s0',specific_h)
    specific_h = concepts[0].copy()
    print('s1',specific_h)
    general_h = [["?" for i in range(len(concepts[0]))] for j in range(len(concepts[0]))]
    print('g0',general_h)
    for i, h in enumerate(concepts):
        if target[i] == "Yes":
            for x in range(len(h)): # Change values in S & G only if values change
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'

        if target[i] == "No":
            for x in range(len(h)):
```

```
            if h[x] != specific_h[x]:
                general_h[x][x] = specific_h[x]
            else:
                general_h[x][x] = '?'
        print(f"s{i}",specific_h)
        print(f"g{i}",general_h)
        print()

    # find indices where we have empty rows, meaning those that are unchanged
    indices = [i for i,val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
        # remove those rows from general_h
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    # Return final values
    return specific_h,general_h
s_final, g_final = learn(concepts, target)
print('Final specific Hypothesis : ',s_final)
print('Final general Hypothesis :',g_final)
```

4. ID3 (NUMPY,PANDAS,MATH)

```
import numpy as np
import pandas as pd
```

```
import math

class Node:
    def __init__(self,l):
        self.label=l
        self.branch={}

#Calculayes Entropy of a dataset
def entropy(data):
    total_ex=len(data)
    p_ex=len(data.loc[data['play']=='yes'])
    n_ex=len(data.loc[data['play']=='no'])
    en=0
    if(p_ex>0):
        en = -(p_ex/float(total_ex))*(math.log(p_ex,2)- math.log(total_ex,2))
    if(n_ex>0):
        en += -(n_ex/float(total_ex))*(math.log(n_ex,2)- math.log(total_ex,2))
    return en


#Calculates Gain of an attribute
def gain(data_s,attrib):
    values=set(data_s[attrib])
    gain=entropy(data_s)
    for val in values:
        gain= gain -
len(data_s.loc[data_s[attrib]==val])/float(len(data_s))*entropy(data_s[data_s[attrib]==val])
    return gain


#Gets attribute with highest gain
def get_attr(data):
    attribute=" "
    max_gain=0
    for attr in data.columns[:-1]:
        g=gain(data,attr)
        if g>max_gain:
            max_gain=g
            attribute =attr
    return attribute



#Constructs Decision Tree
```

```
def decision_tree(data):
    root=Node("NULL")

    #If Entropy is 0, All data is Yes/No.
    if(entropy(data)==0):
        if(len(data.loc[data['play']=='yes']) == len(data)):
            root.label='yes'
        else:
            root.label='no'
        return root
    #If only one attribute is left,Tree is complete
    if(len(data.columns)==1):
        return
    else:
        #Get the attribute with highest gain.
        attr=get_attr(data)
        root.label=attr
        values=set(data[attr])
        for value in values:
            root.branch[value]=decision_tree(data.loc[data[attr]==value].drop(attr,axis=1))
        return root


def test(tree,test_str):
    if not tree.branch:
        return tree.label
    return test(tree.branch[str(test_str[tree.label])],test_str)



data=pd.read_csv('playtennis.csv')
print("Number of Records : ",len(data))
tree=decision_tree(data)
test_str={
"outlook" : "Sunny",
"temperature" : "Hot",
"humidity" : "high",
"wind" : "Weak" ,
}
print(test(tree,test_str))
```

5. BACK PROPAGATION : (NUMPY,TIME,MATPLOTLIB.PYPLOT)

```
import numpy as np
import time
import matplotlib.pyplot as plt
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
lr=0.1
inputlayer_neurons=2
hiddenlayer_neurons=3
output_neurons=1


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

def derivatives_sigmoid(x):
   return x*(1-x)

A=[]
B=[]
for i in range(5):
   epoch=700*i
   A.append(epoch)
   hidden_weight=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
   hidden_bias=np.random.uniform(size=(1,hiddenlayer_neurons))
   output_weight=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
   output_bias=np.random.uniform(size=(1,output_neurons))

   start_time=time.time()

   for i in range(epoch):
      hidden_input=np.dot(X,hidden_weight)+hidden_bias
      hidden_output=sigmoid(hidden_input)

      output_input=np.dot(hidden_output,output_weight)+output_bias
      output_output=sigmoid(output_input)

      Output_Error=y-output_output
```

```
        Output_Difference=Output_Error*derivatives_sigmoid(output_output)

        Hidden_Error=Output_Difference.dot(output_weight.T)
        Hidden_Difference=Hidden_Error*derivatives_sigmoid(hidden_output)

        output_weight=output_weight+hidden_output.T.dot(Output_Difference)*lr
        output_bias=output_bias+np.sum(Output_Difference,axis=0,keepdims=True)*lr

        hidden_weight=hidden_weight+X.T.dot(Hidden_Difference)*lr
        hidden_bias=hidden_bias+np.sum(Hidden_Difference,axis=0,keepdims=True)*lr

    B.append((time.time() - start_time))

plt.plot(A, B)
plt.xlabel('x - axis')
plt.ylabel('y - axis')
plt.title('My first graph!')
plt.show()
print("input:\n" + str(X))
print("Actual output:\n" + str(y))
print("Predicted output \n"+str(output_output))
```

6. NAIVE BAYESIAN CLASSIFIER : (MATH,NUMPY,PANDAS)

NOTE : LOADING OF CSV IS DONE USING PANDAS. SO,1st ROW OF CSV IS IGNORED. TAKE CARE IN DATASET(JUST COPY AND PASTE THE 1st ROW in the LAST ROW/INSERT A NEW ROW AT BEGINNING)

```
import math
import numpy as np
```

```
import pandas as pd

#Creates a dictionary where the keys are the target values
#and classifies the records based on the target value.
#Appending the whole record except the last column(The target value)

def separateByClass(dataset):
    separate={}
    for i in range(len(dataset)):
        if(dataset[i][-1] not in separate):
            separate[dataset[i][-1]]=[]
        separate[dataset[i][-1]].append(dataset[i][0:-1])
    return separate


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

def stdev(numbers):
    avg=mean(numbers)
    varience=sum([math.pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(varience)




#Takes in a set of class records,Here,Its "1.0" & "0.0".
#Summarizes the mean and stddev of each attribute for those set of class records.

def getclassdetails(class_records):
    mean_std=[(mean(attribute),stdev(attribute)) for attribute in zip(*class_records)]
    return mean_std




#Here,We are creating another dictionary after segregating the records
#The values for each key is the set of mean and stdev instead of set of records.

def summarizeByClass(dataset):
    separated=separateByClass(dataset)
    summaries={}
    #items returns key,value pair in the dictionary.
    for classval, class_records in separated.items():
```

```
        summaries[classval]=getclassdetails(class_records)
    return summaries




#Calculates the probabilty given an attribute x

def cal_attr_probability(mean,stdev,x):
    expo=math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))
    result=(1/(math.sqrt(2*math.pi)*stdev))*expo
    return result


#Calculates the probabilty of all classes(Target value) for
#one record based on summaries

def calprobability(summaries,record):
    probability={}
    #items returns key,value pair in the dictionary
    for classval, class_det in summaries.items():
        probability[classval]=1
        for i in range(len(class_det)):
            mean,stdev=class_det[i]
            attr=record[i]
            probability[classval]*=cal_attr_probability(mean,stdev,attr)
    return probability



#Takes in one record and classifies it

def predict_this(summaries,record):
    probability=calprobability(summaries,record)
    bestlabel,bestprob=None, -1
    #Finding the classval with the most probabilty
    for classval, prob in probability.items():
        if bestlabel is None or prob>bestprob:
            bestprob=prob
            bestlabel=classval
    return bestlabel
```

```
#The real main function. Takes in testset,Returns Classfied class values array

def prediction(summaries,testset):
    predict=[]
    for i in range(len(testset)):
        result=predict_this(summaries,testset[i])
        predict.append(result)
    return predict



# Finds accuracy of prediction

def getaccuracy(testset,prediction):
    correct=0
    for i in range(len(testset)):
        if testset[i][-1]==prediction[i]:
            correct+=1
    return (float(correct)/float(len(testset)))*100.0



def main():
    trainingset=np.array(pd.DataFrame(pd.read_csv('NaiveBayesDiabetes.csv')),float)
    testset=np.array(pd.DataFrame(pd.read_csv('NaiveBayesDiabetes1.csv')),float)
    print("records in training dataset={0} and test dataset={1}
rows".format(len(trainingset),len(testset)))
    summaries=summarizeByClass(trainingset)
    predict=prediction(summaries,testset)
    print(predict)
    accuracy=getaccuracy(testset,predict)
    print(accuracy)

main()
```

7. EM AND KMEANS : (PANDAS,NUMPY,MATPLOTLIB.PYPLOT,4 SKLEARN)

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


# 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']
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')
plt.show()

# K MEANS MODEL
model = KMeans(n_clusters=3)
model.fit(X)

# 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')
plt.show()
```

```
# GMM MODEL
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)
gmm_y = gmm.predict(xs)

# Plot the Models Classifications
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')
plt.show()
print('Observation: The GMM using EM algorithm based clustering matched the true labels more closely than the Kmeans.')
```

8. K Nearest Neighbors : (4 SKLEARN)

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix


# Load dataset
iris=datasets.load_iris()
print("Iris Data set loaded...")


# Split the data into train and test samples
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)


# Prints Label no. and their names
for i in range(len(iris.target_names)):
    print("Label", i , "-",str(iris.target_names[i]))


# Create object of KNN classifier
model = KNeighborsClassifier(n_neighbors=1)

# Perform Training
model.fit(x_train, y_train)

# Perform testing
y_pred=model.predict(x_test)


# Display the results
print("Results of Classification using K-nn with K=1 ")
for i in range(len(x_test)):
    print(" Sample:", str(x_test[i]), " Actual-label:", str(y_test[i]), " Predicted-label:", str(y_pred[i]))
print("Classification Accuracy :" , model.score(x_test,y_test));
print('Confusion Matrix :\n',confusion_matrix(y_test,y_pred))
print('Accuracy Metrics : \n',classification_report(y_test,y_pred))
```

9 . Locally weighted Regression : (MATPLOTLIB.PYPLOT,PANDAS,NUMPY)

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

#Creates Weight Matrix i.e Set of Distances of the point from all points in the dataset
def kernel(point,xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye(m)) # eye - identity matrix
    for j in range(m):
        diff = point - xmat[j]
```

```
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights


#Gets Weight of a point from all points in the dataset and then Calculates the localweight
def localWeight(point,xmat,ymat,k):
    wei = kernel(point,xmat,k)
#.I means inverse of matrix,.T means Transpose of matrix
    W = (xmat.T*(wei*xmat)).I*(xmat.T*(wei*ymat.T))
    return W


#Runs localWeight for all points and creates a prediction matrix
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):
    #Argsort sorts the indices in ascending order of the values and returns the indices
    sortindex = X[:,1].argsort(0)
    xsort = X[sortindex][:,0]
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    #Plot clusters
    ax.scatter(bill,tip, color='green')
    #Plot the line
    ax.plot(xsort[:,1],ypred[sortindex], color = 'red', linewidth=5)
    plt.xlabel('Total bill')
    plt.ylabel('Tip')
    plt.show();



# load data points
data = pd.read_csv('program9_dataset.csv')

# We use only Bill amount and Tips data
bill = np.array(data.total_bill)
tip = np.array(data.tip)
```

```
# .mat will convert nd array to 2D array
mbill = np.mat(bill)
mtip = np.mat(tip)

#Getting number of records as m
m= len(bill)

#Creating a identity matrix of order m
one = np.mat(np.ones(m))

#Combining the Identity matrix with Bill matrix and craeating a matrix of mx2
X = np.hstack((one.T,mbill.T)) # 244 rows, 2 cols


# increase k to get smooth curves
ypred = localWeightRegression(X,mtip,3)
graphPlot(X,ypred)
```