#P1: 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!')
```

```python
    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': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J':0
    }
    return H_dist[n]

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')
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': [('A', 2), ('C', 1), ('G', 9)],
    'C': [('B', 1)],
    'D': [('E', 6), ('G', 1)],
    'E': [('A', 3), ('D', 6)],
    'G': [('B', 9), ('D', 1)]
}
aStarAlgo ('A', 'G')
```

# P2: AO* Star Program

```
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]
    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)
            print(minimumCost, childNodeList)
            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)
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()
print ("Graph - 2")
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5,
'H': 7}
graph2 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'D': [[('E', 1), ('F', 1)]]
}

G2 = Graph(graph2, h2, 'A')
G2.applyAOStar()
G2.printSolution()
```

# P3: Candidate-Elimination Algorithm

```
import numpy as np
import pandas as pd
data=pd.DataFrame(data=pd.read_csv('P3.csv'))
concepts=np.array(data.iloc[:,0:-1])
target=np.array(data.iloc[:,-1])
def learn(concepts,target):
    specific_h=concepts[0].copy()
    general_h=[["?" for i in range(len(specific_h))] for i in
range(len(specific_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]='?'
        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]='?'
    indices=[i for i,val in enumerate(general_h) if
val==['?','?','?','?','?','?']]
    print(indices)
    for i in indices:
        general_h.remove(['?','?','?','?','?','?'])
        return specific_h,general_h
s_final,g_final=learn(concepts,target)
print("final S:",s_final,sep="\n")
print("final G:",g_final,sep="\n")
data.head()
```

#P4: ID3 Algorithm

```
import pandas as pd
df_tennis=pd.DataFrame(data=pd.read_csv('P4.csv'))
print(df_tennis)
def entropy(probs):
    import math
    return sum([-prob*math.log(prob,2)for prob in probs])

def entropy_of_list(a_list):
    from collections import Counter
    cnt=Counter(x for x in a_list)
    print("No and Yes class:",a_list.name,cnt)
    num_instances=len(a_list)*1.0
    probs=[x/num_instances for x in cnt.values()]
    return entropy(probs)
print(df_tennis['playtennis'])
total_entropy=entropy_of_list(df_tennis['playtennis'])
print("entropy of given playtennis dataset:",total_entropy)

def
information_gain(df,split_attribute_name,target_attribute_name,trace
=0):
    print("info gain calculation of",split_attribute_name)
    df_split=df.groupby(split_attribute_name)
    for name,group in df_split:
        print(name)
        print(group)
    nobs=len(df.index)*1.0
    df_agg1=df_split.agg({target_attribute_name:lambda
x:entropy_of_list(x)})
    df_agg2=df_split.agg({target_attribute_name:lambda
x:len(x)/nobs})
    df_agg1.columns=['entropy']
    df_agg2.columns=['propotion']
    new_entropy=sum(df_agg1['entropy']*df_agg2['propotion'])
    old_entropy=entropy_of_list(df[target_attribute_name])
    return old_entropy-new_entropy
print("info gain for outlook
is:"+str(information_gain(df_tennis,'outlook','playtennis')),"\n")
print("info gain for humidity
is:"+str(information_gain(df_tennis,'humidity','playtennis')),"\n")
```

```
print("info gain for temperature
is:"+str(information_gain(df_tennis,'temperature','playtennis')),"\n
")

def
id3(df,target_attribute_name,attribute_names,default_class=None):
    from collections import Counter
    cnt=Counter(x for x in df[target_attribute_name])
    if len(cnt)==1:
        return next(iter(cnt))
    elif df.empty or (not attribute_names):
        return default_class
    else:
        default_class=max(cnt.keys())
        gainz=[information_gain(df,attr,target_attribute_name )for
attr in attribute_names]
        index_of_max=gainz.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_name,remaining_attribute_names,defa
ult_class)
        tree[best_attr][attr_val]=subtree
    return tree

attribute_names=list(df_tennis.columns)
print("list of attributes:",attribute_names)
attribute_names.remove('playtennis')
print("list of attributes:",attribute_names)
from pprint import pprint
tree=id3(df_tennis,"playtennis",attribute_names)
pprint("\n\nThe result decision tree is:\n")
pprint(tree)
```

#P5: Back Propogation 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)
y=y/100

class NeuralNetwork(object):
    def __init__(self):
        #parameter
        self.inputsize = 2
        self.outputsize = 1
        self.hiddensize = 3
        self.w1 = np.random.rand(self.inputsize ,self.hiddensize)
        self.w2 = np.random.rand(self.hiddensize ,self.outputsize)

    def feedforward (self , x):
        self.z = np.dot(x,self.w1)
        self.z2=self.sigmoid(self.z)
        self.z3 = np.dot(self.z2, self.w2)
        output = self.sigmoid(self.z3)
        return output

    def sigmoid (self, s, derive = False):
        if(derive == True):
            return s*(1-s)
        return 1/(1+np.exp(-s))

    def backward (self, x, y, output):
        self.output_error = y-output
        self.output_delta =self.output_error*self.sigmoid(output,
derive =True)
        self.z2_error = self.output_delta.dot(self.w2.T)
        self.z2_delta = self.z2_error * self.sigmoid(self.z2 ,derive
= True)
        self.w1+=x.T.dot(self.z2_delta)
        self.w2+= self.z2.T.dot(self.output_delta)
    def train(self, x, y):
        output =self.feedforward(x)
        self.backward(x,y,output)

NN = NeuralNetwork()

for i in range(50000):
    if(i%100 == 0):
        print("loss : " +str(np.mean(np.square(y - 
NN.feedforward(x)))))
    NN.train(x,y)

print("Input : " + str(x))
print("Actual output : " +str(y))
print("pridicted output : " +str(NN.feedforward(x)))
print("loss : " +str(np.mean(np.square(y - NN.feedforward(x)))))
```

#P6: Bayesian Nive Algorithm

```
import pandas as pd
import numpy as np
data = pd.DataFrame(data=pd.read_csv('Human.csv'))
person = pd.DataFrame(data=pd.read_csv('Person.csv'))

n_male = data['Gender'][data['Gender'] == 'male'].count()
n_female = data['Gender'][data['Gender'] == 'female'].count()
total_ppl = data['Gender'].count()

P_male = n_male/total_ppl
P_female = n_female/total_ppl

data_means = data.groupby('Gender').mean()
data_variance = data.groupby('Gender').var()

male_height_mean = data_means ['Height'][data_variance.index == 
'male'].values[0]
male_weight_mean = data_means ['Weight'] [data_variance.index == 
'male'].values[0]
male_footsize_mean = data_means [ 'Foot_Size'][data_variance.index 
== 'male'].values[0]

male_height_variance =data_variance['Height'] [data_variance.index 
== 'male'].values[0]
male_weight_variance =data_variance['Weight'] [data_variance.index 
== 'male'].values[0]
male_footsize_variance 
=data_variance['Foot_Size'][data_variance.index =='male'].values[0]

female_height_mean = data_means [ 'Height'] [data_variance.index == 
'female'].values[0]
female_weight_mean = data_means [ 'Weight'] [data_variance.index == 
'female'].values[0]
```

```
female_footsize_mean =data_means ['Foot_Size'] [data_variance.index
== 'female'].values[0]

female_height_variance = data_variance['Height']
[data_variance.index == 'female'].values[0]
female_weight_variance = data_variance[
'Weight'][data_variance.index == 'female'].values[0]
female_footsize_variance
=data_variance['Foot_Size'][data_variance.index ==
'female'].values[0]
def p_x_given_y(x, mean_y, variance_y):

    p = 1/(np.sqrt(2*np.pi*variance_y))*np.exp((-(x-
mean_y)**2)/(2*variance_y))
    return p
PMale = P_male *p_x_given_y (person['Height'][0], male_height_mean,
male_height_variance)
p_x_given_y (person['Weight'][0], male_weight_mean,
male_weight_variance) * p_x_given_y (person['Foot_Size'][0],
male_footsize_mean, male_footsize_variance)
PFemale = P_female * p_x_given_y (person['Height'][0],
female_height_mean, female_height_variance)
p_x_given_y(person['Weight'][0], female_weight_mean,
female_weight_variance) * p_x_given_y (person['Foot_Size'][0],
female_footsize_mean, female_footsize_variance)
if(PMale > PFemale):
    print("The given data belongs to Male with Probability of
",PMale)
else:
    print("The given data belongs to Female with Probability of
",PFemale)
```

#P7: EM Algorithm using K-Means Algorithm

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

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']
kmeans=KMeans (n_clusters = 3)
clusters=kmeans.fit_predict(X)
from scipy.stats import mode
labels= np.zeros_like(clusters)
for i in range(3):
   cat=(clusters == i)
   labels[cat] = mode(iris. target [cat])[0]
acc = accuracy_score (iris.target, labels)
print('Accuracy = ',acc)
plt.figure(figsize = (10,10))
colormap = np.array([ 'red', 'lime', 'blue'])
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.subplot(2,2,2)
plt.scatter (X.Petal_Length, X. Petal_Width, c = colormap[labels], s
= 40)
plt.title('KMeans Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal width')



from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
scaled_X = scaler.transform(X)
xs = pd.DataFrame(scaled_X, columns = X.columns)

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture (n_components = 3)
gmm_y=gmm.fit_predict(xs)
labels = np.zeros_like(clusters)
for i in range(3):
  cat = (gmm_y== i)
  labels [cat] = mode (iris. target [cat])[0]
acc=accuracy_score (iris.target, labels)
print("Accuracy using GMM = ",acc)
plt.subplot(2,2,3)
plt.scatter(X.Petal_Length, X. Petal_Width, c = colormap[gmm_y],s =
40)
plt.subplots_adjust (hspace= 0.4,wspace = 0.4)
plt.title('GMM Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal width')
plt.show()
```

# P8: K-Nearest Algorithm

```
import pandas as pd
import numpy as np
import sklearn as sl

from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier

iris=datasets.load_iris()
iris.data.shape, iris.target.shape
X_train,X_test,y_train, y_test=train_test_split(iris.data,
iris.target, test_size=0.2, random_state=0)
X_train.shape, y_train.shape
X_test.shape, y_train. shape
clf=KNeighborsClassifier()
clf.fit(X_train,y_train)
KNeighborsClassifier (algorithm= 'auto', leaf_size=30,
metric='minkow ski',metric_params=None, n_jobs=None, n_neighbors=5,
p=2,weights=' uniform')
clf.score (X_test,y_test)
accuracy=clf.score (X_test,y_test)
print (accuracy)
example_measures=np.array([[4.7,3.2,2,0.2], [5.1,2.4,4.3,1.3]])
example=example_measures.reshape(2,-1)
prediction=clf.predict(example)
print (prediction)
```

#P9: Regression Algorithm

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
def kernal (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):
    wt=kernal (point, xmat,k)
    w=(x.T*(wt*x)).I* (x.T*wt*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)
        print (ypred[i])
    return ypred
data=pd.read_csv('Tips.csv')
cola=np.array(data.total_bill)
colb=np.array(data.tip)
mcola= np.mat (cola)
mcolb=np.mat (colb)
m=np.shape (mcolb) [1]
one=np.ones((1,m), dtype=int)
x=np.hstack((one. T,mcola.T))
print(x.shape)
ypred=localweightregression (x,mcolb, 0.5)
xsort=x.copy()
xsort.sort(axis=0)
plt.scatter (cola, colb, color='blue')
plt.plot(xsort[:,1],ypred [x[:,1].argsort (0)],
color='yellow',linewidth=5)
plt.xlabel('Total Bill')
plt.ylabel('tip')
plt.show()
```