A-Star

```
def aStarAlgo(start_node, stop_node):



    open_set = set(start_node) # {A}, len{open_set}=1
    closed_set = set()
    g = {} # store the distance from starting node
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node # parents['S']='S"

    while len(open_set) > 0 :
        n = None
        for v in open_set: # v='A'/'D'
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v # n='S'

        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)      # m=A weight=3 {'A','D','S'}
len{open_set}=2
                parents[m] = n       # parents={'S':S,'A':S}
len{parent}=2
                g[m] = g[n] + weight # g={'S':0,'A':3, 'D':4} len{g}=2



        #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)# {'A','D'} len=2
        closed_set.add(n) #{S} len=1

    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 = {
        'S': 11.5,
        'A': 10.1,
        'B': 5.8,
        'C': 3.4,
        'D': 9.2,
        'E': 7.1,
        'F': 3.5,
        'G': 0
    }

    return H_dist[n]
```

```
#Describe your graph here
Graph_nodes = {

    'S': [('A', 3), ('D', 4)],
    'A': [('B', 4), ('D', 5)],
    'B': [('C', 4), ('E', 5)],
    'C': [],
    'D': [('A', 5), ('E', 2)],
    'E': [('B', 5), ('F', 4)],
    'F': [('G', 3.5)] ,
    'G': []

}

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

AO-Star

```
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)
            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': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
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()

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()
```

Candidate Elimination Algorithm for EnjoySport

```
import numpy as np
import pandas as pd

data = pd.read_csv('enjoysport.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')
```

Decision Tree Using ID3 Algorithm

```
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)
```

Backpropagation Algorithm

```
import numpy as np
inputNeurons=2
hiddenlayerNeurons=2
outputNeurons=2

input  = np.random.randint(1,100,inputNeurons)
output = np.array([5.0,10.0])
hidden_layer=np.random.rand(1,hiddenlayerNeurons)

hidden_biass=np.random.rand(1,hiddenlayerNeurons)
output_bias=np.random.rand(1,outputNeurons)
hidden_weights=np.random.rand(inputNeurons,hiddenlayerNeurons)
output_weights=np.random.rand(hiddenlayerNeurons,outputNeurons)

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

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

for i in range(50):

    hidden_layer=np.dot(input,hidden_weights)
    hidden_layer=sigmoid(hidden_layer+hidden_biass)

    output_layer=np.dot(hidden_layer,output_weights)
```

```
    output_layer=sigmoid(output_layer+output_bias)

    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_
weights.T)

    gradient_hidden_weights = 
np.dot(input.reshape(inputNeurons,1),error_terms_hidden.reshape(1,hiddenlaye
rNeurons))
    gradient_ouput_weights = 
np.dot(hidden_layer.reshape(hiddenlayerNeurons,1),error_terms_output.reshape
(1,outputNeurons))

    hidden_weights = hidden_weights + 0.05*gradient_hidden_weights
    output_weights = output_weights + 0.05*gradient_ouput_weights

    print("**********************")
    print("iteration:",i,"::::",error)
    print("#####output######",output_layer)
```

Naïve Bayes Classifier

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

# Load Data from CSV
data = pd.read_csv('id3.csv')
print("The first 5 Values of data is :\n", data.head())

# obtain train data and train output
X = data.iloc[:, :-1]
print("\nThe First 5 values of the train data is\n", X.head())

y = data.iloc[:, -1]
print("\nThe First 5 values of train output is\n", y.head())

# convert them in numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)
```

```
le_Wind = LabelEncoder()
X.Wind = le_Wind.fit_transform(X.Wind)

print("\nNow the Train output is\n", X.head())

le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n",y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)

classifier = GaussianNB()
classifier.fit(X_train, y_train)
predicted = classifier.predict(X_test)

predictTestData = classifier.predict([[1, 0, 1, 0]])

from sklearn.metrics import accuracy_score
print("Accuracy is:", accuracy_score(classifier.predict(X_test), y_test))
print("Predicted Value for individual Test Data:", predictTestData)
```

K-Means & EM Algorithm

```
from sklearn import datasets
from sklearn import metrics
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
print(iris)
X_train,X_test,y_train,y_test = train_test_split(iris.data,iris.target)
model =KMeans(n_clusters=3)
model.fit(X_train,y_train)
model.score
print('K-Mean: ',metrics.accuracy_score(y_test,model.predict(X_test)))

#-------Expectation and Maximization----------
from sklearn.mixture import GaussianMixture
model2 = GaussianMixture(n_components=3)
model2.fit(X_train,y_train)
model2.score
print('EM Algorithm:',metrics.accuracy_score(y_test,model2.predict(X_test)))
```

KNN Algorithm

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
iris=datasets.load_iris()
print("Iris Data set loaded...")
x_train, x_test, y_train, y_test =
train_test_split(iris.data,iris.target,test_size=0.1)
#random_state=0
for i in range(len(iris.target_names)):
    print("Label", i , "-",str(iris.target_names[i]))
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train, y_train)
y_pred=classifier.predict(x_test)
print("Results of Classification using K-nn with K=5 ")
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));
```

Locally Weighted Regression

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

x = np.linspace(-5, 5, 1000)
y = np.log(np.abs((x ** 2) - 1) + 0.5)
x = x + np.random.normal(scale=0.05, size=1000)
plt.scatter(x, y, alpha=0.3)
def local_regression(x0, x, y, tau):
    x0 = np.r_[1, x0]
    x = np.c_[np.ones(len(x)), x]
    xw =x.T * radial_kernel(x0, x, tau)
    beta = np.linalg.pinv(xw @ x) @ xw @ y
    return x0 @ beta



def radial_kernel(x0, x, tau):
    return np.exp(np.sum((x - x0) ** 2, axis=1) / (-2 * tau ** 2))



def plot_lr(tau):
    domain = np.linspace(-5, 5, num=500)
    pred = [local_regression(x0, x, y, tau) for x0 in domain]
    plt.scatter(x, y, alpha=0.3)
    plt.plot(domain, pred, color="red")
    return plt
```

plot_lr(1).show()

Output : check above console - click plots