A star algo

```
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)# {'F','B'} len=2
        closed_set.add(n) #{A} len=1
  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': 10,    'B': 8,  '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': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 1),('H', 7)] ,
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)],
}
aStarAlgo('A', 'J')
```

AO* algo

```
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 STARTNODE:",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]
Minimum Cost child node/s
    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)
            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

```
import csv
with open("trainingexamples.csv") as f:
    csv_file = csv.reader(f)
    data = list(csv_file)
    specific = data[1][:-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
import math
def base_entropy(dataset):
  p = 0
  n = 0
  target = dataset.iloc[:, -1]
  targets = list(set(target))
  for i in target:
    if i == targets[0]:
      p = p + 1
    else:         n = n + 1
  if p == 0 or n == 0:
    return 0
  elif p == n:
    return 1
  else:       entropy = 0 - (
      ((p / (p + n)) * (math.log2(p / (p + n))) + (n / (p + n)) * (math.log2(n/ (p + n)))))
    return entropy
def entropy(dataset, feature, attribute):
  p = 0       n = 0
  target = dataset.iloc[:, -1]
  targets = list(set(target))
  for i, j in zip(feature, target):
    if i == attribute and j == targets[0]:
      p = p + 1
    elif i == attribute and j == targets[1]:
      n = n + 1
    if p == 0 or n == 0:
      return 0
    elif p == n:
      return 1
    else:            entropy = 0 - (
        ((p / (p + n)) * (math.log2(p / (p + n))) + (n / (p + n)) * (math.log2(n/ (p + n)))))
      return entropy
def counter(target, attribute, i):
  p = 0         n = 0
  targets = list(set(target))
  for j, k in zip(target, attribute):
    if j == targets[0] and k == i:
      p = p + 1
    elif j == targets[1] and k == i:
      n = n + 1
  return p, n
def Information_Gain(dataset, feature):
  Distinct = list(set(feature))
Info_Gain = 0
  for i in Distinct:
    Info_Gain = Info_Gain + feature.count(i) / len(feature) * entropy(dataset,feature, i)
```

```
        Info_Gain = base_entropy(dataset) - Info_Gain
    return Info_Gain
def generate_childs(dataset, attribute_index):
    distinct = list(dataset.iloc[:, attribute_index])
    childs = dict()
    for i in distinct:
        childs[i] = counter(dataset.iloc[:, -1], dataset.iloc[:, attribute_index], i)
    return childs
def modify_data_set(dataset,index, feature, impurity):
    size = len(dataset)
    subdata = dataset[dataset[feature] == impurity]
    del (subdata[subdata.columns[index]])
    return subdata
def greatest_information_gain(dataset):
    max = -1
    attribute_index = 0
    size = len(dataset.columns) - 1
    for i in range(0, size):
        feature = list(dataset.iloc[:, i])
        i_g = Information_Gain(dataset, feature)
        if max < i_g:
            max = i_g
            attribute_index = i
    return attribute_index
def construct_tree(dataset, tree):
    target = dataset.iloc[:, -1]
    impure_childs = []
    attribute_index = greatest_information_gain(dataset)
    childs = generate_childs(dataset, attribute_index)
    tree[dataset.columns[attribute_index]] = childs
    targets = list(set(dataset.iloc[:, -1]))
    for k, v in childs.items():
        if v[0] == 0:
            tree[k] = targets[1]
        elif v[1] == 0:
            tree[k] = targets[0]
        elif v[0] != 0 or v[1] != 0:
            impure_childs.append(k)
    for i in impure_childs:
        sub = modify_data_set(dataset,attribute_index,
        dataset.columns[attribute_index], i)
        tree = construct_tree(sub, tree)
    return tree
def main():
    df = pd.read_csv("playtennis.csv")
    tree = dict()
    result = construct_tree(df, tree)
    for key, value in result.items():
        print(key, " => ", value)
if __name__ == "__main__":
    main()
import numpy as np
```

Regression algo

```
import matplotlib.pyplot as plt
def local_regression(x0, X, Y, tau):
    x0 = [1, x0]
    X = [[1, i] for i in X]
    X = np.asarray(X)
    xw = (X.T) * np.exp(np.sum((X - x0) ** 2, axis=1) / (-2 * tau))
    beta = np.linalg.pinv(xw @ X) @ xw @ Y @ x0
    return beta
def draw(tau):
    prediction = [local_regression(x0, X, Y, tau) for x0 in domain]
    plt.plot(X, Y, 'o', color='black')
    plt.plot(domain, prediction, color='red')
    plt.show()
X = np.linspace(-3, 3, num=1000)
domain = X
Y = np.log(np.abs(X ** 2 - 1) + .5)
draw(10)
draw(0.1)
draw(0.01)
draw(0.001)
```

NAÏVE-BAYES

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np
from sklearn import datasets
iris = datasets.load_iris()
data = iris.data
target = iris.target
x_train, x_test, y_train, y_test = train_test_split(data, target, test_size=0.30)

# Create a Naive Bayes classifier
clf = GaussianNB()
clf.fit(x_train, y_train)
y_pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

NEURAL NETWOK

```
import numpy

def sigmoid(sop):
   return 1.0/(1+numpy.exp(-1*sop))
def sigmoid_sop_deriv(sop):
   return sigmoid(sop)*(1.0-sigmoid(sop))
x1, x2 = 0.1, 0.3
target = 0.03
learning_rate = 0.1
w1 =numpy.random.rand()
w2 =numpy.random.rand()
for k in range(50000):
   y = w1*x1 + w2*x2
   predicted = sigmoid(y)
   g1 = 2*(predicted-target) # error_predicted_deriv
   g2 = sigmoid_sop_deriv(y)
   gradw1 = x1*g2*g1
   gradw2 = x2*g2*g1
   w1 = w1 - learning_rate*gradw1
   w2 = w2 - learning_rate*gradw2
print("Inputs : ", x1, x2)
print("Expected Target : ", target)
print("Predicted Target: ", predicted)
print("Accuracy : ", (1-numpy.abs(target-predicted))*100, "%")
```

CLUSTERING

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import sklearn.metrics as sm
import pandas as pd
import numpy as np
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']
colormap = np.array(['red', 'lime', 'black'])
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("Accuracy of KMeans is ",sm.accuracy_score(Y,model1.predict(X)))
print("Accuracy of EM is ",sm.accuracy_score(Y, model2.predict(X)))
```

KNN

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn import datasets
iris = datasets.load_iris()
x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.30)
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print('Correct predictions:', accuracy)
print('Wrong predictions:', 1 - accuracy)
```