```
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_neighbours(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 not found')
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 doesn't exist")
return None
def get_neighbours(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  }
Path found : ['A', 'E', 'D', 'G']
Out[4]:
['A', 'E', 'D', 'G']
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')
```

```
def recAOStar(n):
global finalPath
print('Expanding node:',n)
and_nodes = []
or_nodes = []
if n in allNodes:
if 'AND' in allNodes[n]:
and_nodes = allNodes[n]['AND']
if 'OR' in allNodes[n]:
or_nodes = allNodes[n]['OR']
if len(and_nodes)==0 and len(or_nodes)==0:
return
solvable = False
marked = {}
while not solvable:
if len(marked)==len(and_nodes)+len(or_nodes):
min_cost_least, min_cost_group_least =
least_cost_grop(and_nodes, or_nodes, {})
solvable = True
change_heuristic(n, min_cost_least)
optimal_child_group[n] = min_cost_group_least
continuemin_cost, min_cost_group =
least_cost_group(and_nodes, or_nodes,marked)
is_expanded = False
if len(min_cost_group)>1:
if min_cost_group[0] in allNodes:
is_expanded = True
recAOStar(min_cost_group[0])
if min_cost_group[1] in allNodes:
is_expanded = True
recAOStar(min_cost_group[1])
else:
if min_cost_group in allNodes:
is_expanded = True
recAOStar(min_cost_group)
if is_expanded:
min_cost_verify, min_cost_group_verify =
least_cost_group(and_nodes, or_nodes,
if min_cost_group==min_cost_group_verify:
solvable = True
change_heuristic(n, min_cost_verify)
optimal_child_group[n] = min_cost_group
else:
solvable = True
change_heuristic(n, min_cost)
optimal_child_group[n] = min_cost_group
marked[min_cost_group] = 1
return heuristic(n)
def least_cost_group(and_nodes, or_nodes,
marked):
node_wise_cost = {}
for node_pair in and_nodes:
```

```
if not node_pair[0]+node_pair[1] in marked:
cost = 0
cost = cost + heuristic(node_pair[0]) +
heuristic(node_pair[1]) + 2
node_wise_cost[node_pair[0]+node_pair[1]] =
cost
for node in or_nodes:
if not node in marked:
cost = 0
cost = cost + heuristic(node) + 1
node_wise_cost[node] = cost
min_cost = 999999
min_cost_group = None
for costKey in node_wise_cost:
if node_wise_cost[costKey]<min_cost:
min_cost = node_wise_cost[costKey]
min_cost_group = costKey
return [min_cost, min_cost_group]
def heuristic(n):
return H_dist[n]
def change_heuristic(n,cost):
H_dist[n] = cost
return
def print_path(node):
print(optimal_child_group[node], end="")
node = optimal_child_group[node]
if len(node)>1:
if node[0] in optimal_child_group:
print("->",end="")
print_path(node[0])
if node[1] in optimal_child_group:
print("->",end="")
print_path(node[1])
else:
if node in optimal_child_group:
print("->",end="")
print_path(node)
H_dist = {'A':-1, 'B':4, 'C':2, 'D':3, 'E':6,
'F':8, 'G':2, 'H':0, 'I':0, 'J':0}
allNodes = {'A' :{'AND':[('C','D')],
'OR':['B']} ,
'B' :{'OR':['E','F']} ,
'C' :{'OR':['G'], 'AND':[('H','I')]} ,
'D' :{'OR':['J']}
}
optimal_child_group = {}
optimal_cost = recAOStar('A')
print('Nodes which give optimal cost are:')
print_path('A')
print("\nOptimal Cost is : ",optimal_cost)
```

```
import numpy as np
import pandas as pd
data =
pd.DataFrame(data=pd.read_csv('Training.csv'))
print(data)
concepts = np.array(data.iloc[:,0:-1])
target = np.array(data.iloc[:,-1])
def learn(concepts, target):
specific_h = concepts[0].copy()
print("\nInitialization of specific_h and
general_h")
print("\n",specific_h)
general_h = [["?" for i in
range(len(specific_h))] for i in
range(len(specific_h))]
print("\n",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] = '?'
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(" \nsteps of Candidate Elimination
Algorithm",i+1)
print("\nSpecific_h ",i+1,"\n ")
print(specific_h)
print("\ngeneral_h ", i+1, "\n ")
print(general_h)
indices = [i for i, val in enumerate(general_h)
if val == ['?', '?', '?', '?', '?', '?'
for i in indices:
general_h.remove(['?','?','?','?','?','?'])
return specific_h, general_h
s_final, g_final = learn(concepts, target)
print("\nFinal Specific_h:", s_final,
sep="\n")
print("\nFinal General_h:", g_final,
sep="\n")
```

```
import math
def dataset_split(data, arc, val):
newData = []
for rec in data:
if rec[arc] == val:
reducedSet = list(rec[:arc])
reducedSet.extend(rec[arc+1:])
newData.append(reducedSet)
return newData
def calc_entropy(data):
entries = len(data)
labels = {}
for rec in data:
label = rec[-1]
if label not in labels.keys():
labels[label] = 0
labels[label] += 1
entropy = 0.0
for key in labels:
prob = float(labels[key])/entries
entropy -= prob * math.log(prob, 2)
return entropy
def attribute_selection(data):
features = len(data[0]) - 1
baseEntropy = calc_entropy(data)
max_InfoGain = 0.0
bestAttr = -1
for i in range(features):
AttrList = [rec[i] for rec in data]
uniqueVals = set(AttrList)
newEntropy = 0.0
attrEntropy = 0.0
for value in uniqueVals:
newData = dataset_split(data, i, value)
prob = len(newData)/float(len(data))
newEntropy = prob * calc_entropy(newData)
attrEntropy += newEntropy
```

```
infoGain = baseEntropy - attrEntropy
if infoGain > max_InfoGain:
max_InfoGain = infoGain
bestAttr = i
return bestAttr
def decision_tree(data, labels):
classList = [rec[-1] for rec in data]
if classList.count(classList[0]) ==
len(classList):
return classList[0]
maxGainNode = attribute_selection(data)
treeLabel = labels[maxGainNode]
theTree = {treeLabel: {}}
del(labels[maxGainNode])
nodeValues = [rec[maxGainNode] for rec in data]
uniqueVals = set(nodeValues)
for value in uniqueVals:
subLabels = labels[:]
theTree[treeLabel][value] =
decision_tree(dataset_split(data,
maxGainNode, value),
return theTree
def print_tree(tree, level):
if tree == 'yes' or tree == 'no':
print(' '*level, 'd=', tree)
return
for key,value in tree.items():
print(' ' *level, key)
print_tree(value, level*2)
with open('tennis.csv', 'r') as csvfile:
fdata = [line.strip() for line in csvfile]
metadata = fdata[0].split(',')
train_data = [x.split(',') for x in fdata[1:]]
tree = decision_tree(train_data, metadata)
print_tree(tree, 1)
print(tree)
```

```
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)
def sigmoid(x):
return 1 / (1 + np.exp(-x))
def der_sigmoid(x):
return x * (1 - x)
epoch = 5000
lr = 0.01
neurons_i = 2
neurons_h = 3
neurons_o = 1
weight_h = np.random.uniform(size=(neurons_i,
neurons_h))
bias_h = np.random.uniform(size=(1,
neurons_h))
```

```
weight_o = np.random.uniform(size=(neurons_h,
neurons_o))
bias_o = np.random.uniform(size=(1,
neurons_o))
for i in range(epoch):
inp_h = np.dot(X, weight_h) + bias_h
out_h = sigmoid(inp_h)
inp_o = np.dot(out_h, weight_o) + bias_o
out_o = sigmoid(inp_o)
err_o = y - out_o
grad_o = der_sigmoid(out_o)
delta_o = err_o * grad_o
err_h = delta_o.dot(weight_o.T)
grad_h = der_sigmoid(out_h)
delta_h = err_h * grad_h
weight_o += out_h.T.dot(delta_o) * lr
weight_h += X.T.dot(delta_h) * lr
print('Input: ', X)
print('Actual: ', y)
print('Predicted: ', out_o)
```

```python
import pandas as pd
import numpy as np
mush = pd.read_csv('mushrooms.csv')
mush = mush.replace('?',np.nan)
mush.dropna(axis=1,inplace=True)
target = 'class'
features = mush.columns[mush.columns!=target]
target_classes=mush[target].unique()
test = mush.sample(frac = .3)
mush = mush.drop(test.index)
cond_probs = {}
target_class_prob = {}
for t in target_classes:
mush_t = mush[mush[target]==t][features]
target_class_prob[t] =
float(len(mush_t)/len(mush))
class_prob = {}
for col in mush_t.columns:
col_prob = {}
for val,cnt in
mush_t[col].value_counts().iteritems():
pr = cnt/len(mush_t)
col_prob[val] = pr
class_prob[col] = col_prob
cond_probs[t] = class_prob
def calc_probs(x):
probs = {}
for t in target_classes:
```

```
p = target_class_prob[t]
for col,val in x.iteritems():
try:
p *= cond_probs[t][col][val]
except:
p = 0
probs[t] = p
return probs
def classify(x):
probs = calc_probs(x)
max = 0
max_class = ' '
for cl,pr in probs.items():
if pr>max:
max = pr
max_class = cl
return max_class
b = []
for i in mush.index:
b.append(classify(mush.loc[i,features]) ==
mush.loc[i,target])
print(sum(b)," correct of ",len(mush))
print('Accuracy : ',sum(b)/len(mush))
b = []
for i in test.index:
b.append(classify(test.loc[i,features]) ==
test.loc[i,target])
print(sum(b)," correct of ",len(test))
print('Accuracy : ',sum(b)/len(test))
```

```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.cluster import KMeans
data = pd.read_csv('ex.csv')
f1 = data['V1'].values
f2 = data['V2'].values
X = np.array(list(zip(f1,f2)))
print("x: ",X)
print("Graph for whole dataset")
plt.scatter(f1,f2,c='black')
plt.show()
KMeans =KMeans(2)
```

```
labels =KMeans.fit(X).predict(X)
print("labels for KMeans:",labels)
print('Graph using KMeans Algorithm')
plt.scatter(f1,f2,c = labels)
centroids =KMeans.cluster_centers_
print("centroids: ",centroids)
plt.scatter(centroids[:,0],centroids[:,1],mar
ker ='*',c='red')
plt.show()
gmm=GaussianMixture(2)
Labels=gmm.fit(X).predict(X)
print("Labels for GMM: ",labels)
print('Graph using EM Algorithm')
plt.scatter(f1,f2,c=labels)
plt.show()
```

```
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()
targets = iris_dataset.target_names
print('class : number')
for i in range(len(targets)):
print(targets[i]," : ",i)
```

```
X_train, X_test, Y_train, Y_test =
train_test_split(iris_dataset['data'],iris_da
taset['targ
kn = KNeighborsClassifier(1)
kn.fit(X_train,Y_train)
for i in range(len(X_test)):
x_new = np.array([X_test[i]])
prediction = kn.predict(x_new)
print("Actual:[{0}][{1}],Predicted:{2}
{3}".format(Y_test[i],targets[Y_test[i]],pred
ict
print("\nAccuracy:",kn.score(X_test,Y_test))
```

```
from math import ceil
import numpy as np
from scipy import linalg
def lowess(x, y, f=2./3., iter=3):
n = len(x)
r = int(ceil(f*n))
h = [np.sort(np.abs(x-x[i]))[r] for i in
range(n)]
w = np.clip(np.abs((x[:,None]-x[None,:])/h) ,
0.0 , 1.0)
w = (1- w**3) ** 3
yest = np.zeros(n)
delta = np.ones(n)
for iteration in range(iter):
for i in range(n):
weights = delta*w[:,i]
b = np.array([np.sum(weights*y) ,
np.sum(weights*y*x)])
A = np.array([[np.sum(weights) ,
np.sum(weights*x)],
```

```
 [np.sum(weights*x),np.sum(weights*x*x)]])
beta = linalg.solve(A,b)
yest[i] = beta[0] + beta[1]*x[i]
residuals = y - yest
s = np.median(np.abs(residuals))
delta = np.clip(residuals/(6.0*s),-1,1)
delta = (1 - delta**2) ** 2
return yest
if __name__=='__main__':
import math
n = 100
x = np.linspace(0 , 2*math.pi , n)
y = np.sin(x) + 0.3 * np.random.randn(n)
f = 0.25
yest = lowess(x,y,f,3)
import pylab as pl
pl.clf()
pl.plot(x,y,label='y noisy')
pl.plot(x,yest,label='y predicted')
pl.legend()
pl.show()
```