Program1

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {} #store distance from starting node
    parents = {}# parents contains an adjacency map of all nodes
    #ditance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        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):
                #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)
                    parents[m] = n
                    g[m] = g[n] + weight
                #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)
        closed_set.add(n)

    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 = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }

    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

output:

```
Path found: ['A', 'E', 'D', 'G']
 ['A', 'E', 'D', 'G']
```

PROGRAM 2

```
class Graph:
   def _init_(self, graph, heuristicNodeList, startNode):  #instantiate graph object
with graph topology, heuristic values, start node

      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()
```

OUTPUT:

PROGRAM 3

```
dataarr=[]
with open('desktop/prom3.csv') as f:
    for line in f:
        dataarr.append(line.strip().split(','))

rows = len(dataarr)
cols = len(dataarr[0])
shypo = ['0']*(cols-1)
ghypo = [['?']*(cols-1)]
print("Initial Specific Hypothesis is = ", shypo)
print("Initial General Hypothesis is = ", ghypo)


for x in range(1, rows):
    lst = dataarr[x]

    if lst[cols-1] == "1":
        for i in range(0, cols-1):
            if shypo[i] == lst[i]:
                continue
            shypo[i] = '?' if shypo[i] != '0' else lst[i]
            for g in ghypo:
                if g[i] != '?' and shypo[i] == '?':
                    ghypo.remove(g)

    elif lst[cols-1] == "0":
        ghypo.clear()
        for i in range(0, cols-1):
            if lst[i] != shypo[i] and shypo[i] != '?':
                temp_list = ['?']*i + [shypo[i]] + (['?']*(cols-2-i))
                if temp_list not in ghypo:
                    ghypo.append(temp_list)
```

```
    print("S Hypothesis after row ", x, " = ", shypo)
    print("G Hypothesis after row ", x, " = ", ghypo)
    print("Final SHypothesis ", shypo)
    print("Final GHypothesis ", ghypo)
```

DATA SET:

sunny	warm	normal	strong	warm	same	yes
sunny	warm	high	strong	warm	same	yes
rain	cold	high	strong	warm	change	no
sunny	warm	high	strong	cool	change	yes

OUTPUT:

```
Initial Specific Hypothesis is =  ['0', '0', '0', '0', '0', '0']
Initial General Hypothesis is =  [['?', '?', '?', '?', '?', '?']
S Hypothesis after row  1  =  ['0', '0', '0', '0', '0', '0']
G Hypothesis after row  1  =  [['?', '?', '?', '?', '?', '?']]
Final SHypothesis  ['0', '0', '0', '0', '0', '0']
Final GHypothesis  [['?', '?', '?', '?', '?', '?']]
S Hypothesis after row  2  =  ['0', '0', '0', '0', '0', '0']
G Hypothesis after row  2  =  [['?', '?', '?', '?', '?', '?']]
Final SHypothesis  ['0', '0', '0', '0', '0', '0']
Final GHypothesis  [['?', '?', '?', '?', '?', '?']]
S Hypothesis after row  3  =  ['0', '0', '0', '0', '0', '0']
G Hypothesis after row  3  =  [['?', '?', '?', '?', '?', '?']]
Final SHypothesis  ['0', '0', '0', '0', '0', '0']
Final GHypothesis  [['?', '?', '?', '?', '?', '?']]
```

\

PROGRAM 4:

```
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 formula calculation
        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), subLabels)
    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('desktop/prog4.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)
```

DATASET:

outlook	temperature	humidity	wind	playtennis
sunny	hot	strong	weak	no
sunny	hot	high	strong	no
overcast	hot	high	weak	yes
rain	mild	weak	weak	yes
rain	cool	normal	weak	yes
rain	cool	normal	strong	no
overcast	cool	normal	strong	yes
sunny	mild	high	weak	no
sunny	cool	normal	weak	yes
rain	mild	normal	weak	yes
sunny	mild	normal	strong	yes
overcast	mild	high	strong	yes
overcast	hot	normal	weak	yes
rain	mild	high	strong	no

OUTPUT:

```
humidity
   weak
     d = yes
   normal
     outlook
         sunny
                 d = yes
         overcast
                 d = yes
         rain
                 wind
                                 weak
                                                                 d = yes
                                 strong
                                                                 d = no
   strong
     d = no
   high
     outlook
         sunny
                 d = no
         overcast
                 d = yes
         rain
                 d = no
{'humidity': {'weak': 'yes', 'normal': {'outlook': {'sunny': 'yes', 'overc
ast': 'yes', 'rain': {'wind': {'weak': 'yes', 'strong': 'no'}}}}, 'strong'
: 'no', 'high': {'outlook': {'sunny': 'no', 'overcast': 'yes', 'rain': 'no
'}}}}
```

PROGRAM 5:

```
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)
```

OUTPUT:

```
Input:  [[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual:  [[0.92]
 [0.86]
 [0.89]]
Predicted:  [[0.94125622]
 [0.93158805]
 [0.94317674]]
```

PROGRAM 6:

```
import pandas as pd
import numpy as np
mush = pd.read_csv("desktop/pgrm6.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))
    # Test data
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))
```

DATASET:

class	capshape	capsurface	capcolor	bruises
1	1	1	1	5
1	1	1	2	5
2	1	1	2	10
3	2	1	1	10
3	3	2	2	5
2	2	2	2	10
1	2	1	1	5
1	3	2	1	10
3	2	2	2	10
1	2	2	2	10
2	2	1	2	10
2	1	2	1	10
3	2	1	2	5
1	2	1	2	10
1	2	1	2	5

OUTPUT:

```
1 correct of 11
Accuracy: 0.09090909090909091
2 correct of 11
Accuracy: 0.18181818181818182
2 correct of 11
Accuracy: 0.18181818181818182
3 correct of 11
Accuracy: 0.2727272727272727
4 correct of 11
Accuracy: 0.36363636363636365
5 correct of 11
Accuracy: 0.45454545454545453
5 correct of 11
Accuracy: 0.45454545454545453
6 correct of 11
Accuracy: 0.5454545454545454
7 correct of 11
Accuracy: 0.6363636363636364
7 correct of 11
Accuracy: 0.6363636363636364
7 correct of 11
Accuracy: 0.6363636363636364
1 correct of 4
Accuracy: 0.25
```

PROGRAM 7:

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

names = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width', 'Class']

dataset = pd.read_csv("desktop/prog7.csv", names=names)

X = dataset.iloc[:, :-1]

label = {'Iris-setosa': 0,'Iris-versicolor': 1, 'Iris-virginica': 2}

y = [label[c] for c in dataset.iloc[:, -1]]

plt.figure(figsize=(14,7))
colormap=np.array(['red','lime','black'])

# REAL PLOT
plt.subplot(1,3,1)
plt.title('Real')
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y])

# K-PLOT
model=KMeans(n_clusters=3, random_state=0).fit(X)
plt.subplot(1,3,2)
plt.title('KMeans')
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[model.labels_])

print('The accuracy score of K-Mean: ',metrics.accuracy_score(y, model.labels_))
print('The Confusion matrixof K-Mean:\n',metrics.confusion_matrix(y,
model.labels_))

# GMM PLOT
gmm=GaussianMixture(n_components=3, random_state=0).fit(X)
```

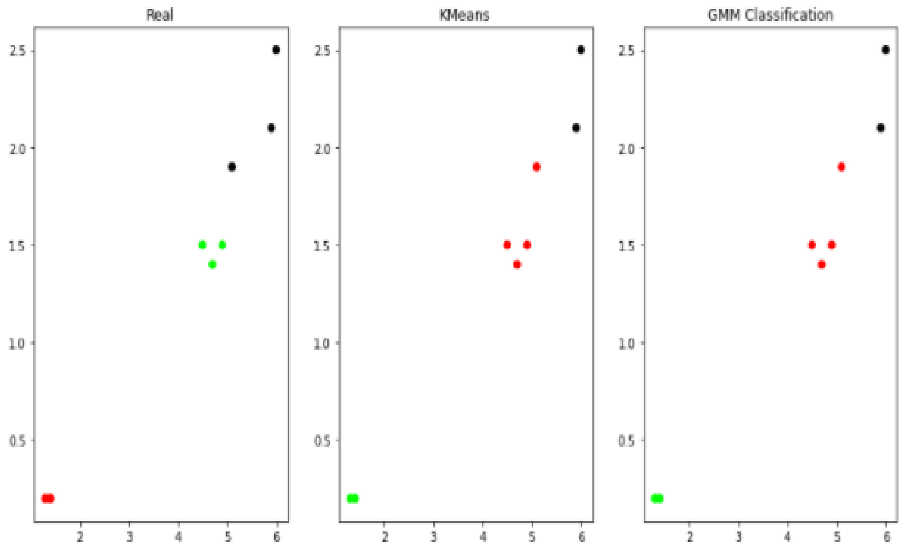
```
y_cluster_gmm=gmm.predict(X)
plt.subplot(1,3,3)
plt.title('GMM Classification')
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y_cluster_gmm])

print('The accuracy score of EM: ',metrics.accuracy_score(y, y_cluster_gmm))
print('The Confusion matrix of EM:\n ',metrics.confusion_matrix(y,
y_cluster_gmm))
```

DATASET:

5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
7	3.2	4.7	1.4	Iris-versicolor
6.4	3.2	4.5	1.5	Iris-versicolor
6.9	3.1	4.9	1.5	Iris-versicolor
6.5	3.3	6	2.5	Iris-virginica
5.6	2.7	5.1	1.9	Iris-virginica
7.1	3	5.9	2.1	Iris-virginica

OUTPUT:

```
The accuracy score of K-Mean:  0.2222222222222222
The Confusion matrixof K-Mean:
 [[0 3 0]
 [3 0 0]
 [1 0 2]]
The accuracy score of EM:  0.2222222222222222
The Confusion matrix of EM:
  [[0 3 0]
 [3 0 0]
 [1 0 2]]
```



PROGRAM 8:

```
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_dataset["target"])
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]],
prediction,
    targets[prediction]))

    print("\nAccuracy: ", kn.score(X_test, y_test))
```

OUTPUT:

```
Accuracy:  0.9473684210526315
Actual:[2] [virginica],Predicted:[2] ['virginica']

Accuracy:  0.9473684210526315
Actual:[1] [versicolor],Predicted:[1] ['versicolor']

Accuracy:  0.9473684210526315
Actual:[0] [setosa],Predicted:[0] ['setosa']

Accuracy:  0.9473684210526315
Actual:[2] [virginica],Predicted:[2] ['virginica']

Accuracy:  0.9473684210526315
Actual:[0] [setosa],Predicted:[0] ['setosa']

Accuracy:  0.9473684210526315
Actual:[2] [virginica],Predicted:[2] ['virginica']

Accuracy:  0.9473684210526315
```

PROGRAM 9 :

```
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)

    # Straight Line Fitting
    # x=np.linspace(0,2.5,n) # For Linear
```

```
# y= 1 + 0.25*np.random.randn(n) # For Linear

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()
```

OUTPUT: