In [4]:
```
class Graph:
    def __init__(self, adjac_list):
        self.adjac_list = adjac_list

    def get_neighbours(self, v):
        return self.adjac_list[v]

    def h(self, n):
        H = {
            'S': 14,
            'A': 12,
            'B': 11,
            'C': 6,
            'D': 4,
            'E': 11,
            'G': 0
        }
        return H[n]

    def a_star_algorithm(self, start, stop):
        open_lst = set([start])
        closed_lst = set([])

        g = {}
        g[start] = 0

        parent = {}
        parent[start] = start

        while len(open_lst)>0:
            n = None
            for v in open_lst:
                if n==None or g[v]+self.h(v)<g[n]+self.h(n):
                    n = v
            if n==None:
                print('Path does not exist!')
                return None
            if n==stop:
                path = []
                while parent[n] != n:
                    path.append(n)
                    n=parent[n]
                path.append(start)
                path.reverse()
                print('Path found: {}'.format(path))
                return path
            for (m, weight) in self.get_neighbours(n):
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    parent[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m]>g[n] + weight:
                        g[m] = g[n] + weight
                        parent[m] = n
                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)

            open_lst.remove(n)
            closed_lst.add(n)
```

```python
        print('Path does not exist!')
        return None

adjac_lis = {
    'S': [('A', 4), ('B', 3)],
    'B': [('C', 7), ('D', 10)],
    'A': [('E', 5), ('D', 12)],
    'C': [('D', 2)],
    'D': [('G', 5)],
    'E': [('G', 16)],
    'G': None
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('S', 'G')
```

```
Path found: ['S', 'B', 'C', 'D', 'G']
```

Out[4]:
```
['S', 'B', 'C', 'D', 'G']
```

In [ ]:

In [1]:
```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode): #instantiate graph object with graph topology, heuristic

        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self): # starts a recursive AO* algorithm
        self.aoStar(self.start, False)

    def getNeighbors(self, v): # gets the Neighbors of a given node
        return self.graph.get(v,'')

    def getStatus(self,v): # return the status of a given node
        return self.status.get(v,0)

    def setStatus(self,v, val): # set the status of a given node
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0) # always return the heuristic value of a given node

    def setHeuristicNodeValue(self, n, value):
        self.H[n]=value # set the revised heuristic value of a given node


    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE STARTNODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v): # Computes the Minimum Cost of child nodes of a given node v
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v): # iterate over all the set of child node/s
            cost=0
```

```
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)

            if flag==True: # initialize Minimum Cost with the cost of first set of child node/s
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s
                flag=False
            else: # checking the Minimum Cost nodes with the current Minimum Cost
                if minimumCost>cost:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s


        return minimumCost, costToChildNodeListDict[minimumCost] # return Minimum Cost and Minimum Cost child node/s


    def aoStar(self, v, backTracking): # AO* algorithm for a start node and backTracking status flag

        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 # check the Minimum Cost nodes of v are solved

            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode)!=-1:
                    solved=solved & False

            if solved==True: # if the Minimum Cost nodes of v are solved, set the current node status as solved(-1)
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList # update the solution graph with the solved nodes which may be a


            if v!=self.start: # check the current node is the start node for backtracking the current node value
```

```
                self.aoStar(self.parent[v], True) # backtracking the current node value with backtracking status set

            if backTracking==False: # check the current call is not for backtracking
                for childNode in childNodeList: # for each Minimum Cost child node
                    self.setStatus(childNode,0) # set the status of child node to 0(needs exploration)
                    self.aoStar(childNode, False) # Minimum Cost child node is further explored with backtracking sta



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} # Heuristic values of Nodes
graph2 = { # Graph of Nodes and Edges
    'A': [[('B', 1), ('C', 1)], [('D', 1)]], # Neighbors of Node 'A', B, C & D with repective weights
    'B': [[('G', 1)], [('H', 1)]], # Neighbors are included in a list of lists
    'D': [[('E', 1), ('F', 1)]] # Each sublist indicate a "OR" node or "AND" nodes
}

G2 = Graph(graph2, h2, 'A') # Instantiate Graph object with graph, heuristic values and start Node
G2.applyAOStar() # Run the AO* algorithm
G2.printSolution() # print the solution graph as AO* Algorithm search
```

```
HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : B
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : G
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : B
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 10, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
SOLUTION GRAPH : {}
PROCESSING NODE : I
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 0, 'J': 1, 'T': 3}
SOLUTION GRAPH : {'I': []}
PROCESSING NODE : G
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I']}
PROCESSING NODE : B
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 12, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE : C
-----------------------------------------------------------------------------------------
```

```
HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE : J
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': []}
PROCESSING NODE : C
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 1, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J']}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE STARTNODE: A
------------------------------------------------------------
{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}
------------------------------------------------------------
HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : E
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
```

```
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : F
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 0, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': [], 'F': []}
PROCESSING NODE : D
-----------------------------------------------------------------------------------------
HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 2, 'E': 0, 'F': 0, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': [], 'F': [], 'D': ['E', 'F']}
PROCESSING NODE : A
-----------------------------------------------------------------------------------------
FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE STARTNODE: A
------------------------------------------------------------
{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}
------------------------------------------------------------
```

In [ ]:

In [7]:

```
import pandas as pd
import numpy as np
data=pd.DataFrame(data=pd.read_csv('/home/student/Desktop/dataset/finds.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==['?','?','?','?','?','?']]
        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()
```

```
final s:
['sunny' 'warm' '?' 'strong' '?' '?']
final g:
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]
```

Out[7]:

	sky	temp	humidity	wind	water	forecast	enjoy
0	sunny	warm	normal	strong	warm	same	yes
1	sunny	warm	high	strong	warm	same	yes
2	rainy	cold	high	strong	warm	change	no
3	sunny	warm	high	strong	cold	change	yes

In [3]:
```
import math
import csv
def load_csv(filename):
    lines=csv.reader(open(filename,"r"))
    dataset=list(lines)
    headers=dataset.pop(0)
    return dataset,headers

class Node:
    def __init__(self,attribute):
        self.attribute=attribute
        self.children=[]
        self.answer=""

def subtables(data,col,delete):
    dic={}
    coldata=[row[col] for row in data]
    attr=list(set(coldata))
    for k in attr:
        dic[k]=[]
    for y in range(len(data)):
        key=data[y][col]
        if delete:
            del data[y][col]
        dic[key].append(data[y])
    return attr,dic

def entropy(s):
    attr=list(set(s))
    if len(attr)==1:
        return 0
    counts=[0,0]
    for i in range(2):
        counts[i]=sum([1 for x in s if attr[i]==x])/(len(s)*1.0)
    sums=0
    for cnt in counts:
        sums+=-1*cnt*math.log(cnt,2)
    return sums

def compute_gain(data,col):
    attvalues,dic=subtables(data,col,delete=False)
    total_entropy=entropy([row[-1] for row in data])
```

```
    for x in range(len(attvalues)):
        ratio=len(dic[attvalues[x]])/(len(data)*1.0)
        entro=entropy([row[-1] for row in dic[attvalues[x]]])
        total_entropy-=ratio*entro
    return total_entropy

def build_tree(data,features):
    lastcol=[row[-1] for row in data]
    if(len(set(lastcol)))==1:
        node=Node(" ")
        node.answer=lastcol[0]
        return node
    n=len(data[0])-1
    gains=[compute_gain(data,col)for col in range(n)]
    split=gains.index(max(gains))
    node = Node(features[split])
    fea=features[:split]+features[split+1:]
    attr,dic=subtables(data,split,delete=True)
    for x in range(len(attr)):
        child=build_tree(dic[attr[x]],fea)
        node.children.append((attr[x],child))
    return node

def print_tree(node,level):
    if node.answer!=" ":
        print(" "*level,node.answer)
        return
    print(" "*level,node.attribute)
    for value,n in node.children:
        print(""*(level+1),value)
        print_tree(n,level+2)

def classify(node,x_test,features):
    if node.answer!="":
        return
    pos=features.index(node.attribute)
    for values,n in node.children:
        if x_test[pos]==values:
            classify(n,x_test,features)

#main program
dataset,features=load_csv("/home/student/Desktop/dataset/playtennis.csv")
node=build_tree(dataset,features)
print("The decision tree for the dataset using id3 algorithm is")
```

```
print_tree(node,0)
testdata,features=load_csv("/home/student/Desktop/dataset/test_tennis.csv")
for xtest in testdata:
    print("the test instance:",xtest)
    print("the predicted lable:",end="")
    classify(node,xtest,features)
```

```
The decision tree for the dataset using id3 algorithm is

the test instance: ['rain', 'cool', 'normal', 'strong']
the predicted lable:the test instance: ['sunny', 'mild', 'normal', 'strong']
the predicted lable:
```

In [2]:

```
import pandas as pd
training_data=pd.read_csv("/home/student/Desktop/dataset/playtennis.csv")
training_data
```

Out[2]:

	Outlook	Temperature	Humidity	Wind	Answer
0	sunny	hot	high	weak	no
1	sunny	hot	high	strong	no
2	overcast	hot	high	weak	yes
3	rain	mild	high	weak	yes
4	rain	cool	normal	weak	yes
5	rain	cool	normal	strong	no
6	overcast	cool	normal	strong	yes
7	sunny	mild	high	weak	no
8	sunny	cool	normal	weak	yes
9	rain	mild	normal	weak	yes
10	sunny	mild	normal	strong	yes
11	overcast	mild	high	strong	yes
12	overcast	hot	normal	weak	yes
13	rain	mild	high	strong	no

In [3]:
```
 import pandas as pd
training_data=pd.read_csv("/home/student/Desktop/dataset/test_tennis.csv")
training_data
```

Out[3]:

	Outlook	Temperature	Humidity	Wind
0	rain	cool	normal	strong
1	sunny	mild	normal	strong

In [ ]:

In [5]:
```
import random
from math import exp
from random import seed
def initialize_network(n_inputs, n_hidden, n_outputs):
    network =list()
    hidden_layer = [{'weights':[random.uniform(-0.5,0.5) for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer =[{'weights' :[random.uniform(-0.5,0.5) for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
def transfer(activation):
    return 1.0/ (1.0 + exp(-activation))
def forward_propagate(network ,row):
    inputs = row
    for layer in network:
        new_inputs =[]
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output']= transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs
def transfer_derivative(output):
    return output * (1.0 - output)
def backward_propagate_error(network ,expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1 :
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
```

```
                errors.append(expected[j] - neuron['output'])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[: -1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']
def train_network(network, train, l_rate, n_epoch, n_outputs):
        for epoch in range(n_epoch):
            sum_error = 0
            for row in train:
                outputs = forward_propagate(network, row)
                expected = [0 for i in range(n_outputs)]
                expected[row[-1]] = 1
                sum_error += sum([(expected[i] -outputs[i])**2 for i in range(len(expected))])
                backward_propagate_error(network, expected)
                update_weights(network, row, l_rate)
            print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
seed(1)
dataset = [[2.7810836,2.550537003,0],
           [1.465489372,2.362125076,0],
           [3.396561688,4.400293529,0],
           [1.38807019,1.850220317,0],
           [3.06407232,3.0050220317,0],
           [7.627531214,2.759262235,1],
           [5.332441248,2.088626775,1],
           [6.922596716,1.77106367,1],
           [8.675418651,-0.242068655,1],
           [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0])-1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs , 2, n_outputs)
print(network)
train_network(network, dataset, 0.5,10, n_outputs)
for layer in network:
    print(layer)
```

```
[[{'weights': [-0.3656357558875988, 0.3474337369372327, 0.26377461897661403]}, {'weights': [-0.2449309742605783, -0.0
04564912908059049, -0.050508935211261874]}], [{'weights': [0.15159297272276295, 0.2887233511355132, -0.40614041322576
51]}, {'weights': [-0.4716525234779937, 0.3357651039198697, -0.06723293209494663]}]]
>epoch=0, lrate=0.500, error=4.763
>epoch=1, lrate=0.500, error=4.558
>epoch=2, lrate=0.500, error=4.316
>epoch=3, lrate=0.500, error=4.035
>epoch=4, lrate=0.500, error=3.733
>epoch=5, lrate=0.500, error=3.428
>epoch=6, lrate=0.500, error=3.132
>epoch=7, lrate=0.500, error=2.850
>epoch=8, lrate=0.500, error=2.588
>epoch=9, lrate=0.500, error=2.348
[{'weights': [-1.1463897474725036, 1.3042284004924503, 0.5852017931585984], 'output': 0.03442281577237726, 'delta': -
0.008364387542565752}, {'weights': [-0.5385173279741822, 0.35104917838159383, 0.05286718071658475], 'output': 0.06401
680323288057, 'delta': -0.004583419945797485}]
[{'weights': [1.463301526815239, 0.836981834952207, -0.7888850651698373], 'output': 0.34390741525894336, 'delta': -0.
0775975857483978}, {'weights': [-1.6666896342474495, -0.1345911518872607, 0.6857645557122467], 'output': 0.6406516742
64048, 'delta': 0.0827281317861522}]
```

```
In [ ]:
```

In [1]:
```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics

iris=datasets.load_iris()
print("Iris dataset loaded")

X,y=datasets.load_iris(return_X_y=True)
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.05,random_state=0)

print("Dataset is aplit into trainingand testing...")
print("Size of training data and its lable",X_train.shape,y_train.shape)
print("Size of training data and its lable",X_test.shape,y_test.shape)

for i in range(len(iris.target_names)):
    print("Lable",i,"-",str(iris.target_names[i]))

gnb=GaussianNB()
y_pred=gnb.fit(X_train,y_train).predict(X_test)
print("confusion matrix:\n",metrics.confusion_matrix(y_test,y_pred))
print("Results of classification using Navis Bayes")
for r in range(0,len(X_test)):
    print("Sample:",str(X_test[r]),"Actual_Lable:",str(y_test[r]),"predicted_lable:",str(y_pred[r]))
print("Classification accuracy:",gnb.score(X_test,y_test))
print("Other reports:\n",metrics.classification_report(y_test,y_pred))
```

```
Iris dataset loaded
Dataset is aplit into trainingand testing...
Size of training data and its lable (142, 4) (142,)
Size of training data and its lable (8, 4) (8,)
Lable 0 - setosa
Lable 1 - versicolor
Lable 2 - virginica
confusion matrix:
 [[3 0 0]
 [0 2 0]
 [0 0 3]]
Results of classification using Navis Bayes
Sample: [5.8 2.8 5.1 2.4] Actual_Lable: 2 predicted_lable: 2
Sample: [6.  2.2 4.  1. ] Actual_Lable: 1 predicted_lable: 1
Sample: [5.5 4.2 1.4 0.2] Actual_Lable: 0 predicted_lable: 0
Sample: [7.3 2.9 6.3 1.8] Actual_Lable: 2 predicted_lable: 2
Sample: [5.  3.4 1.5 0.2] Actual_Lable: 0 predicted_lable: 0
Sample: [6.3 3.3 6.  2.5] Actual_Lable: 2 predicted_lable: 2
Sample: [5.  3.5 1.3 0.3] Actual_Lable: 0 predicted_lable: 0
Sample: [6.7 3.1 4.7 1.5] Actual_Lable: 1 predicted_lable: 1
Classification accuracy: 1.0
Other reports:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         3
           1       1.00      1.00      1.00         2
           2       1.00      1.00      1.00         3

    accuracy                           1.00         8
   macro avg       1.00      1.00      1.00         8
weighted avg       1.00      1.00      1.00         8
```

In [ ]:

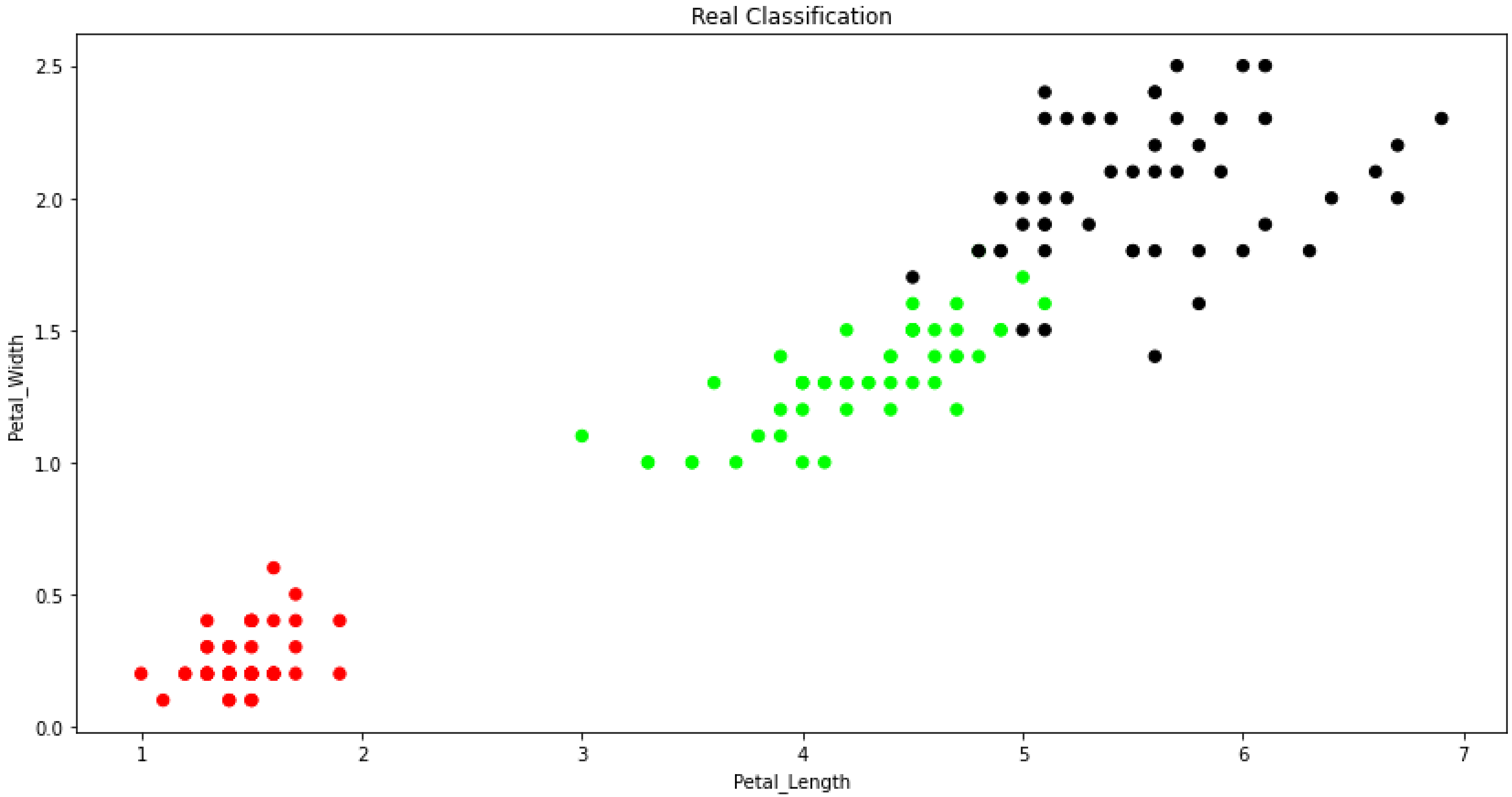
Program 7 - EMA: Expectation Maximization Algorithm

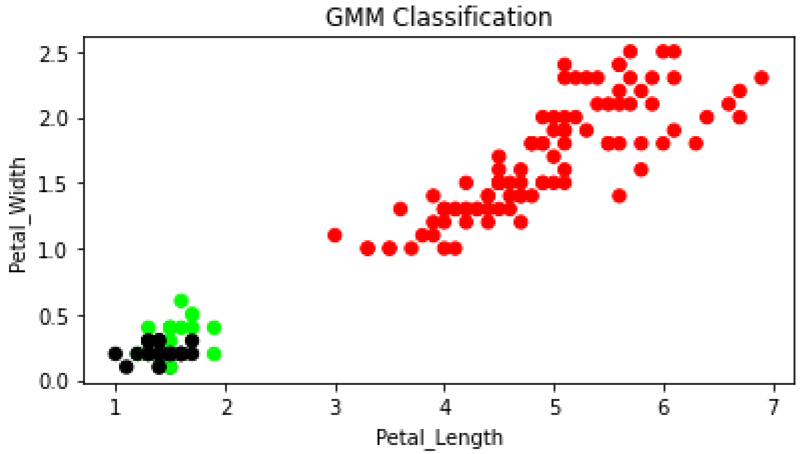
In [18]:

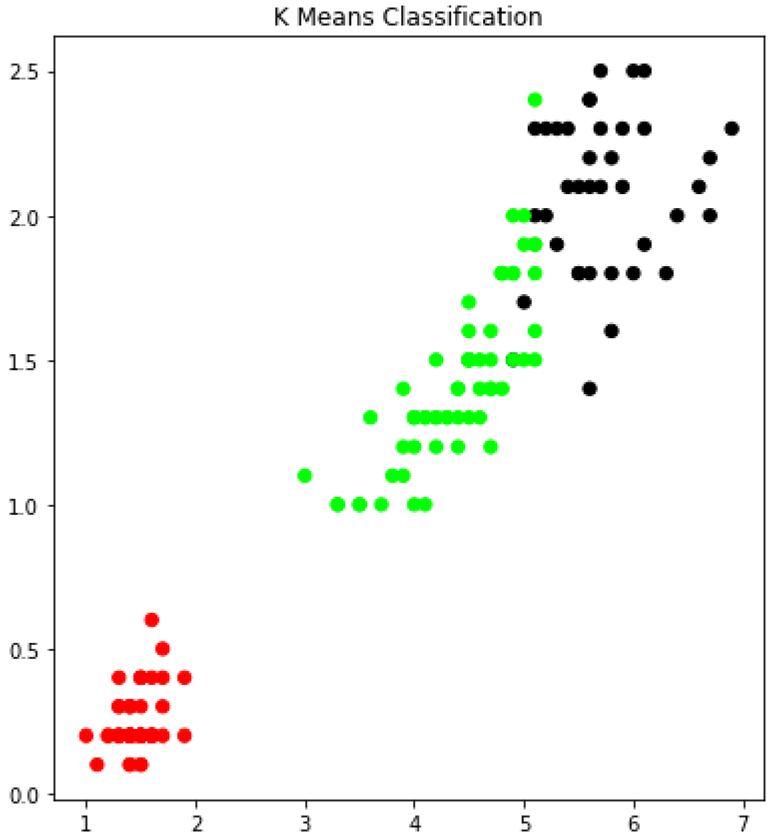
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.mixture import GaussianMixture

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"]

model = KMeans(n_clusters = 3)
model.fit(X)
model.labels_

plt.figure(figsize = (14, 7))
colormap = np.array(["red", "lime", "black"])
plt.subplot(1, 1, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[y.Targets], s = 40)
plt.title("Real Classification")
plt.xlabel("Petal_Length")
plt.ylabel("Petal_Width")
plt.figure(figsize = (14, 7))
predY = np.choose(model.labels_, [0, 1, 2]).astype(np.int64)
plt.subplot(1, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[predY], s = 40)
plt.title("K Means Classification")

scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
```

```
gmm = GaussianMixture(n_components = 3)
gmm.fit(xs)
y_cluster_gmm = gmm.predict(xs)
plt.subplot(2, 2, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c = colormap[y_cluster_gmm], s = 40)
plt.title("GMM Classification")
plt.xlabel("Petal_Length")
plt.ylabel("Petal_Width")

print("Observation: The GMM using EM algorithm based clustering matched the true label more closely than the K-MEANS"
```

```
Observation: The GMM using EM algorithm based clustering matched the true label more closely than the K-MEANS
```




GMM Classification
Petal_Width
Petal_Length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


K Means Classification
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7

```
In [ ]:
```

Program 8 - KNN: K-Nearest Neighbor

In [10]:
```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets

iris = datasets.load_iris()
print("Iris dataset loaded")

x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size = 0.1)

print("Dataset is split into training and testing: ")
print("Size of training data and its label: " , x_train.shape, y_train.shape)
print("Size of testining data and its label: ", x_test.shape, y_test.shape)

for i in range(len(iris.target_names)):
    print("Label: ", i, "-", str(iris.target_names[i]))

classifier = KNeighborsClassifier(n_neighbors = 2)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)

print("Results of classification using KNN with K = 2: ")
for r in range(0, len(x_test)):
    print("Sample: ", str(x_test[r]), "\t Actual_Label: ", str(y_test[r]), "\t Predicted_Label: ", str(y_pred[r]))
print("Classification accuracy: ", classifier.score(x_test, y_test))
```

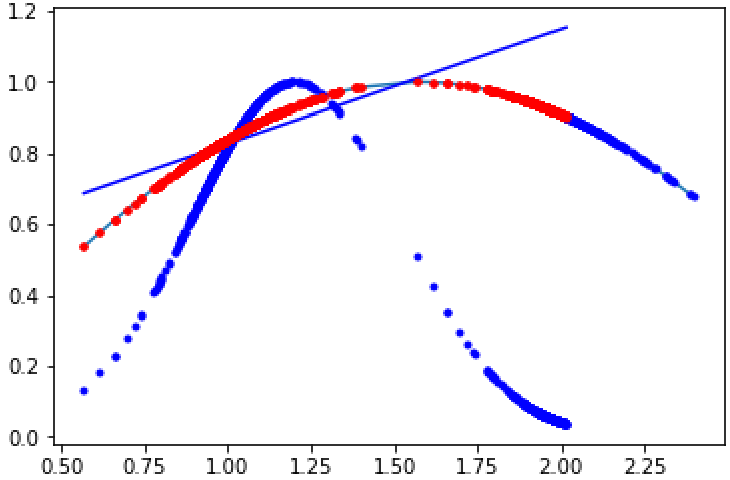
```
Iris dataset loaded
Dataset is split into training and testing:
Size of training data and its label:  (135, 4) (135,)
Size of testining data and its label:  (15, 4) (15,)
Label:  0 - setosa
Label:  1 - versicolor
Label:  2 - virginica
Results of classification using KNN with K = 2:
Sample:  [6.7 3.1 5.6 2.4]       Actual_Label:  2        Predicted_Label:  2
Sample:  [6.6 2.9 4.6 1.3]       Actual_Label:  1        Predicted_Label:  1
Sample:  [6.3 2.5 4.9 1.5]       Actual_Label:  1        Predicted_Label:  2
Sample:  [6.5 3.  5.8 2.2]       Actual_Label:  2        Predicted_Label:  2
Sample:  [6.6 3.  4.4 1.4]       Actual_Label:  1        Predicted_Label:  1
Sample:  [4.9 3.1 1.5 0.1]       Actual_Label:  0        Predicted_Label:  0
Sample:  [5.6 2.9 3.6 1.3]       Actual_Label:  1        Predicted_Label:  1
Sample:  [6.9 3.2 5.7 2.3]       Actual_Label:  2        Predicted_Label:  2
Sample:  [6.3 3.4 5.6 2.4]       Actual_Label:  2        Predicted_Label:  2
Sample:  [4.6 3.1 1.5 0.2]       Actual_Label:  0        Predicted_Label:  0
Sample:  [5.8 2.7 3.9 1.2]       Actual_Label:  1        Predicted_Label:  1
Sample:  [5.  3.  1.6 0.2]       Actual_Label:  0        Predicted_Label:  0
Sample:  [7.7 2.8 6.7 2. ]       Actual_Label:  2        Predicted_Label:  2
Sample:  [4.9 2.4 3.3 1. ]       Actual_Label:  1        Predicted_Label:  1
Sample:  [5.8 2.7 5.1 1.9]       Actual_Label:  2        Predicted_Label:  2
Classification accuracy:  0.9333333333333333
```

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import load_boston
import matplotlib.pyplot as plt
%matplotlib inline
import math
import warnings
warnings.filterwarnings('ignore')
boston =load_boston()
features=pd.DataFrame(boston.data,columns=boston.feature_names)
target=pd.DataFrame(boston.target,columns=['target'])
data=pd.concat([features,target],axis=1)
x=data['RM']
X1=sorted(np.array(x/x.mean()))
X=X1+[i+1 for i in X1]
Y=np.sin(X)
plt.plot(X,Y)

n=int(0.8*len(X))
x_train=X[:n]
y_train=Y[:n]
x_test=X[n:]
y_test=Y[n:]
w=np.exp([-(1.2-i)**2/(2*0.1)for i in x_train])
plt.plot(x_train,y_train,'r.')
plt.plot(x_train,w,'b.')

def h(x,a,b):
    return a*x + b

def error(a,x,b,y,w):
    e=0
    m=len(x)
    for i in range(m):
        e+=np.power(h(x[i],a,b,)-y[i],2)*w[i]
    return(1/(2*m))*e


def step_gradient(a,x,b,y,learning_rate,w):
    grad_a=0
    grad_b=0
    m=len(x)
```

```
    for i in range(m):
        grad_a += (2/m)*((h(x[i],a,b)-y[i])*x[i])*w[i]
        grad_b += (2/m)*(h(x[i],a,b)-y[i])*w[i]
    a=a-(grad_a * learning_rate)
    b=b-(grad_b * learning_rate)
    return a,b

def descend(initial_a,initial_b,x,y,learning_rate,iterations,w):
    a=initial_a
    b=initial_b
    for i in range(iterations):
        e=error(a,x,b,y,w)
        if i%1000==0:
            print(f"Error:{e}---- a:{a}, b:{b}")
        a,b=step_gradient(a,x,b,y, learning_rate,w)
    return a,b

a=1.8600662368042573
b=-0.7962243178421666
learning_rate=0.01
iterations=10000
final_a, final_b = descend(a,b,x_train,y_train, learning_rate,iterations,w)
H=[i*final_a+final_b for i in x_train]
plt.plot(x_train,y_train,'r.',x_train,H,'b')
print(error(a,x_test,b,y_test,w))
print(error(final_a,x_test,final_b,y_test,w))
plt.plot(x_test,y_test,'b.',x_train,y_train,'r.')
```

```
Error:0.06614137226206705---- a:1.8600662368042573, b:-0.7962243178421666
Error:0.01831248988715221---- a:1.3533605603913972, b:-0.6206735673234249
Error:0.011422762970211432---- a:1.1032234861838637, b:-0.347590814908577
Error:0.007176247674245229---- a:0.9068452261129998, b:-0.13319830250762849
Error:0.0045588881799908---- a:0.7526720746347257, b:0.0351175247039557
Error:0.0029456664570710403---- a:0.6316334187867452, b:0.16725934893398114
Error:0.0019513497294632626---- a:0.536608078323685, b:0.2710015934995427
Error:0.001338497980224941---- a:0.46200533867114346, b:0.3524478227325071
Error:0.0009607639482851428---- a:0.4034360271954487, b:0.41638983867834906
Error:0.0007279458172072266---- a:0.35745428091221954, b:0.4665896016596849
1.6930984012182055
0.037219754002487955
```

Out[1]:
```
[<matplotlib.lines.Line2D at 0x7f6a3a560c70>,
 <matplotlib.lines.Line2D at 0x7f6a3a560d30>]
```



In [ ]:

In [ ]: