**BANGALORE INSTITUTE OF TECHNOLOGY**
**K.R.ROAD, V.V.PURAM, BANGALORE-560 004**

**Department of Computer Science & Engineering**

# ARTIFICIAL INTELLIGENCE & MACHINE LEARNING LABORATORY

**VII-Sem CS&E**
**18CSL76**

**Prepared By:**
**Dr. M.S. Bhargavi**
**Prof. Madhuri J.**

**DEPT OF CS&E**

| ARTIFICIAL INTELLIGENCE & MACHINE LEARNING LABORATORY<br>[As per Choice Based Credit System (CBCS) scheme]<br>(Effective from the academic year 2018- 2019)<br>SEMESTER – VII | | | |
|---|---|---|---|
| Subject Code | 18CSL76 | IA Marks | 40 |
| Number of Lecture Hours/Week | 01I + 02P | Exam Marks | 60 |
| Total Number of Lecture Hours | 40 | Exam Hours | 03 |
| **CREDITS – 02** | | | |

**Description (If any):**

1. The programs can be implemented in either JAVA or Python.
2. For Problems 1 to 6 and 10, programs are to be developed without using the built-in classes or APIs of Java/Python.
3. Data sets can be taken from standard repositories (https://archive.ics.uci.edu/ml/datasets.html) or constructed by the students.

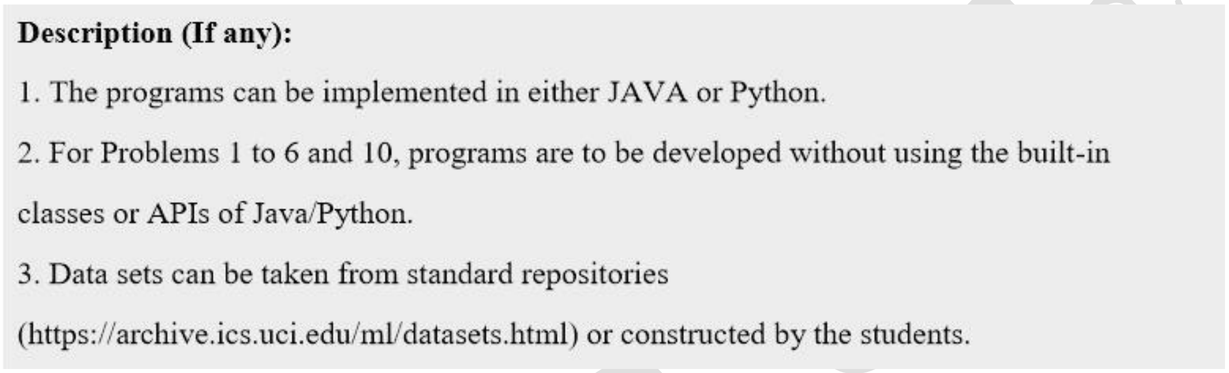
**Lab Experiments:**

1. Implement A* Search algorithm.
2. Implement AO* Search algorithm.
3. For a given set of training data examples stored in a .CSV file, implement and demonstrate the **Candidate-Elimination algorithm** to output a description of the set of all hypotheses consistent with the training examples.
4. Write a program to demonstrate the working of the decision tree based **ID3 algorithm**. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.
5. Build an Artificial Neural Network by implementing the **Backpropagation algorithm** and test the same using appropriate data sets.
6. Write a program to implement the **naïve Bayesian classifier** for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.
7. Apply **EM algorithm** to cluster a set of data stored in a .CSV file. Use the same data set for clustering using ***k*-Means algorithm**. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.
8. Write a program to implement ***k*-Nearest Neighbour algorithm** to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.
9. Implement the non-parametric **Locally Weighted Regression algorithm** in order to fit data points. Select appropriate data set for your experiment and draw graphs.

# ARTIFICIAL INTELLIGENCE & MACHINE LEARNING LABORATORY

[As per Choice Based Credit System (CBCS) scheme]

**SEMESTER – VII** **Subject Code: 18CSL76**



**Description (If any):**

1. The programs can be implemented in either JAVA or Python.

2. For Problems 1 to 6 and 10, programs are to be developed without using the built-in classes or APIs of Java/Python.

3. Data sets can be taken from standard repositories (https://archive.ics.uci.edu/ml/datasets.html) or constructed by the students.

## Program 1

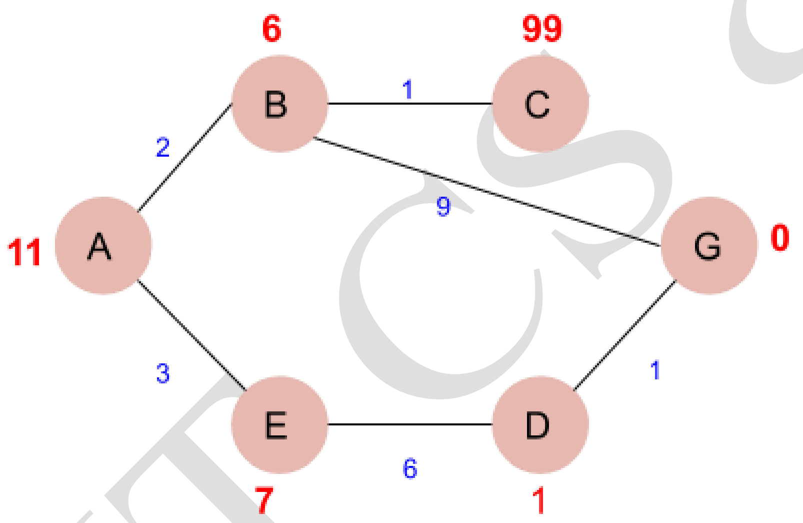
**Implement A* Search Algorithm**

**A * algorithm** is a searching algorithm that searches for the shortest path between the *initial and the final state*. It is used in various applications, such as *maps*. The A* algorithm uses a modified evaluation function and a Best-First search. A* minimizes the total path cost. Under the right conditions A* provides the cheapest cost solution in the optimal time.

**A* algorithm:**

**f(n) = g(n) + h(n)**

- **h(n) = cost of the cheapest path from node 'n' to a goal state.**
  **g(n) = cost of the cheapest path from the initial state to node 'n'.**



- g(x) + h(x) = f(x)
- 0+ 11 =11
- Thus for A, we can write
- A=11

Now from A, we can go to point B or point E, so we compute f(x) for each of them
- A → B = 2 + 6 = 8
- A → E = 3 + 7 = 10

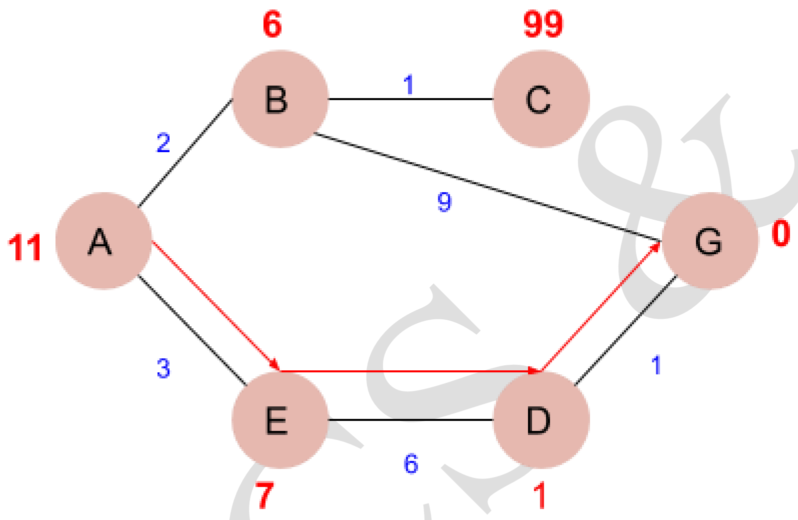
Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B

Since there is no path between C and G, the heuristic cost is set infinity or a very high value
- A → B → C = (2 + 1) + 99= 102
- A → B → G = (2 + 9 ) + 0 = 11

Here the path A → B → G has the least cost but it is still more than the cost of A → E, thus we explore this path further A → E → D = (3 + 6) + 1 = 10
Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path. And compute the f(x) for the children of D

A → E → D → G = (3 + 6 + 1) +0 =**10**

Now comparing all the paths that lead us to the goal, we conclude that A → E → D → G is the most cost-effective path to get from A to G.



**SOURCE CODE IN PYTHON**

```
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

#Distance of starting node from itself is zero
    g[start_node]=0
#start_node is root node i.e it has no parent
#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):
#node ‘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:
                        g[m]=g[n]+weight
                        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 reconstructing 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
#becuse 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 neighbour 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 fuction 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]
Graph_nodes={
    'A':[('B',2),('E',3)],
    'B':[('C',1),('G',9)],

    'D':[('G',1)],
    'E':[('D',5)],

    }
aStarAlgo('A','G')
```

**OUTPUT:**

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

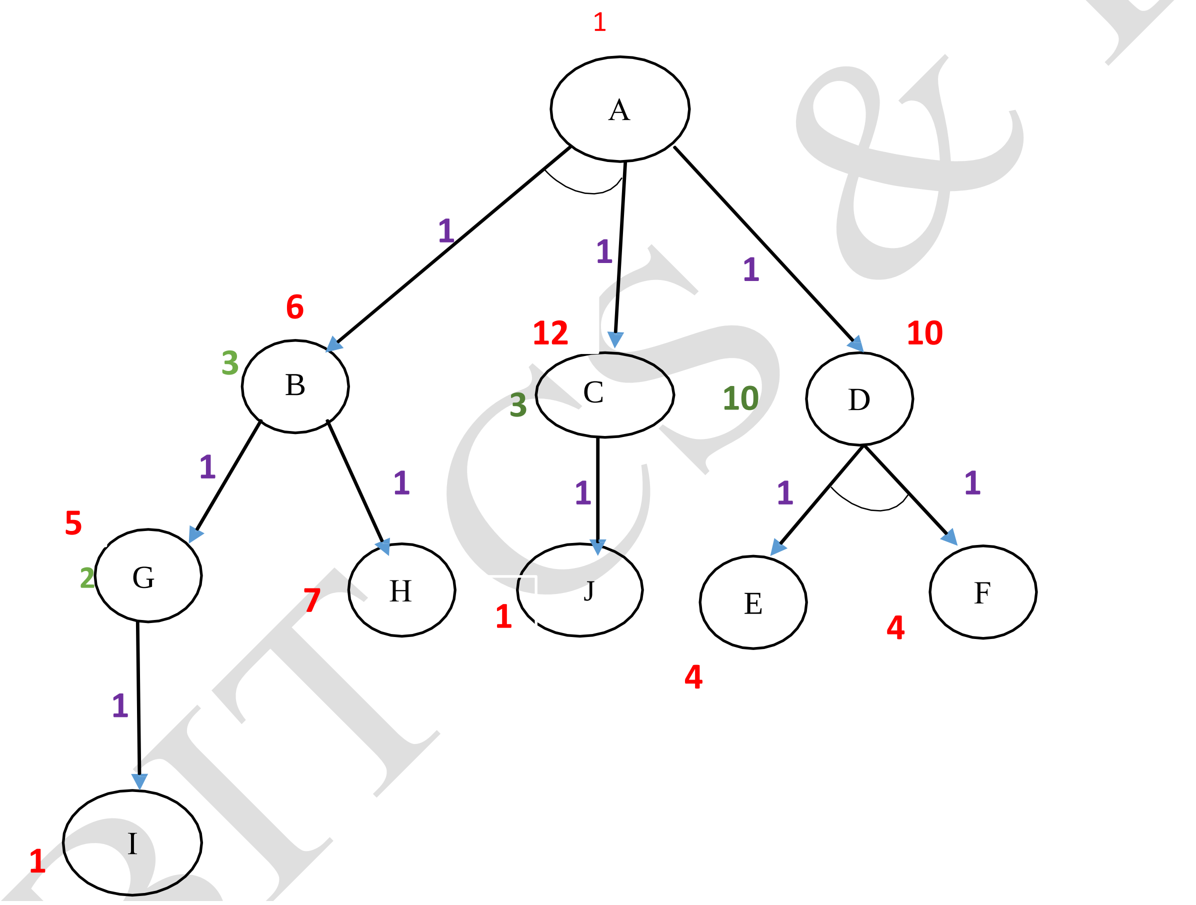
## Program 2

**Implement AO* Search Algorithm**

AO* Algorithm basically based on problem decomposition (Breakdown problem into small pieces) When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, **AND-OR graphs** or **AND - OR trees** are used for representing the solution.

The decomposition of the problem or problem reduction generates AND arcs.

### How AO* works



Basically, We will calculate the **cost function** here **(F(n)= G (n) + H (n))**

**H: heuristic/ estimated** value of the nodes. and **G:** actual cost or edge value (here unit value).

Here we have taken the **edges value 1** , meaning we have to focus solely on the **heuristic value.**

1. The **Purple color** values are **edge values (here all are same that is one).**

2. The **Red color** values are **Heuristic values for nodes.**

3. **The Green color** values are **New Heuristic values for nodes.**

**Procedure:**

1. In the above diagram we have two ways from **A to D** or **A to B-C** (because of and condition). calculate cost to select a path

   **F(A-D)= 1+10 = 11** and **F(A-BC) = 1 + 1 + 6 +12 = 20**
2. As we can see **F(A-D)** is less than **F(A-BC)** then the algorithm choose the path **F(A-D).**
3. Form D we have one choice that is **F-E.**

   **F(A-D-FE) = 1+1+ 4 +4 =10**
4. Basically **10** is the cost of reaching **FE from D.** And **Heuristic value of node D** also denote the cost of reaching **FE from D**. So, the new Heuristic value of D is 10.
5. And the Cost from A-D remain same that is **11**.
6. Suppose we have searched this path and we have got the **Goal State**, then we will never explore the other path. (this is what AO* says but here we are going to explore other path as well to see what happen)

**SOURCE CODE IN PYTHON**

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

   def getNeighbors(self,v):                       #gets the neighbours 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 start value:",self.start)
      print("---------------------------------------")
      print(self.solutionGraph)
      print("---------------------------------------")

   def computeMinimumCostChildNodes(self,v):
#coputes 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 nodes
         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
                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 & backtracking status flog
        print("Heuristic values:",self.H)
        print("Solution Graph:",self.solutionGraph)
        print("Processing Graph:",v)
        print("-----------------------------------")
        if self.getStatus(v)>=0:       #if status node v>=0, copute minimum cost nodes of v
            minimumCost,childNodeList=self.computeMinimumCostChildNodes(v)
            print(minimumCost,childNodeList)
            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:
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList
            if v!=self.start:
                self.aoStar(self.parent[v],True)
            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 status as false
h1={'A':1,'B':6,'C':2,'D':12,'E':2,'F':1,'G':5,'H':7,'I':7,'J':1}
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)]],#neighbours of node A,B,C&D with respective weights
      'B':[[('G',1)],[('H',1)]],        #neighbours are included in a list of lists
      'D':[[('E',1),('F',1)]]  #Each sub list indicate a "OR" node or "AND" nodes
      }
G2=Graph(graph2,h2,'A')  #instantiate graph object with graph,heuristic values & start node
G2.applyAOStar()         #Run the AO* algorithm
G2.printSolution()       #print the solution graph as output of the AO* algorithm search
```

## OUTPUT 1:

```
Heuristic values: {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
Solution Graph: {}
Processing Graph: A
------------------------------------
10 ['B', 'C']
Heuristic values: {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
Solution Graph: {}
Processing Graph: B
------------------------------------
6 ['G']
Heuristic values: {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
Solution Graph: {}
Processing Graph: A
------------------------------------
10 ['B', 'C']
Heuristic values: {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
Solution Graph: {}
Processing Graph: G
------------------------------------
8 ['I']
Heuristic values: {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1}
Solution Graph: {}
Processing Graph: B
------------------------------------
8 ['H']
Heuristic values: {'A': 10, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1}
Solution Graph: {}
Processing Graph: A
------------------------------------
12 ['B', 'C']
Heuristic values: {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1}
```

```
Solution Graph: {}
Processing Graph: I
-----------------------------------
0 []
Heuristic values: {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 0, 'J': 1}
Solution Graph: {'I': []}
Processing Graph: G
-----------------------------------
1 ['I']
Heuristic values: {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}
Solution Graph: {'I': [], 'G': ['I']}
Processing Graph: B
-----------------------------------
2 ['G']
Heuristic values: {'A': 12, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}
Solution Graph: {'I': [], 'G': ['I'], 'B': ['G']}
Processing Graph: A
-----------------------------------
6 ['B', 'C']
Heuristic values: {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}
Solution Graph: {'I': [], 'G': ['I'], 'B': ['G']}
Processing Graph: C
-----------------------------------
2 ['J']
Heuristic values: {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}
Solution Graph: {'I': [], 'G': ['I'], 'B': ['G']}
Processing Graph: A
-----------------------------------
6 ['B', 'C']
Heuristic values: {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}
Solution Graph: {'I': [], 'G': ['I'], 'B': ['G']}
Processing Graph: J
-----------------------------------
0 []
Heuristic values: {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0}
Solution Graph: {'I': [], 'G': ['I'], 'B': ['G'], 'J': []}
Processing Graph: C
-----------------------------------
1 ['J']
Heuristic values: {'A': 6, 'B': 2, 'C': 1, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0}
Solution Graph: {'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J']}
Processing Graph: A
-----------------------------------
5 ['B', 'C']
For graph solution, traverse the graph from start value: 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 Graph: A
------------------------------------
11 ['D']
Heuristic values: {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
Solution Graph: {}
Processing Graph: D
------------------------------------
10 ['E', 'F']
Heuristic values: {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
Solution Graph: {}
Processing Graph: A
------------------------------------
11 ['D']
Heuristic values: {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
Solution Graph: {}
Processing Graph: E
------------------------------------
0 []
Heuristic values: {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
Solution Graph: {'E': []}
Processing Graph: D
------------------------------------
6 ['E', 'F']
Heuristic values: {'A': 11, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
Solution Graph: {'E': []}
Processing Graph: A
------------------------------------
7 ['D']
Heuristic values: {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
Solution Graph: {'E': []}
Processing Graph: F
------------------------------------
0 []
Heuristic values: {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 0, 'G': 5, 'H': 7}
Solution Graph: {'E': [], 'F': []}
Processing Graph: D
------------------------------------
2 ['E', 'F']
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 Graph: A
------------------------------------
```

```
3 ['D']
For graph solution, traverse the graph from start value: A
---------------------------------------
{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}
```

## Program 3

**For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.**

### Candidate-Elimination algorithm

***G*  maximally general hypotheses in H**
**S  maximally specific hypotheses in H**
**For each training example = <x, c(x)>**
**Case1: If d is a positive example**

Remove from G any hypothesis that is inconsistent with d
For each hypothesis s in S that is not consistent with d

- Remove s from S.
- Add to S all minimal generalizations h of s such that
  - h consistent with d
  - Some member of G is more general than h
- Remove from S any hypothesis is that is more general than another hypothesis is in S

**Case 2: If d is a negative example**

Remove from S any hypothesis that is inconsistent with d for each hypothesis g in G that is not consistent with d

- Remove g from G.
- Add to G all minimal specializations h of g such that
  - h consistent with d
  - Some member of S is more specific than h
- Remove from G any hypothesis that is less general than

**Dataset considered:**

| Example | Sky | Air Temp | Humidity | Wind | Water | Forecast | Enjoy Sport |
|---|---|---|---|---|---|---|---|
| D1 | Sunny | Warm | Normal | Strong | Warm | Same | Yes |
| D2 | Sunny | Warm | High | Strong | Warm | Same | Yes |
| D3 | Rainy | Cold | High | Strong | Warm | Change | No |
| D1 | Sunny | Warm | High | Strong | Cool | Change | Yes |

### The steps of Candidate Elimination algorithm are as follows:

**Initially,**

**$S_0$ = {**<'ϕ', 'ϕ', 'ϕ', 'ϕ', 'ϕ', 'ϕ'>}

**$G_0$ = {<?,?,?,?,?,?>}**

**Step 1:**

**D1=**{Sunny, Warm, Normal, Strong, Water, Same} = Positive training example Therefore update specific hypothesis S.

**S1**= {Sunny, Warm, Normal, Strong, Water, Same}

**$G_1$**= {<? , ? , ? , ? , ? , ?>}

**Step 2:**

**D2** = {Sunny,Warm,High,Strong, Warm, Same} = Positive Training example, Therefore specific hypothesis S is updated.

**S2**= {Sunny, Warm, ?, Strong, Water, Same}

**G1, G2**= {<? , ? , ? , ? , ? , ?>}

**Step 3:**

Training examples 1 and 2 forces the specific hypothesis to be general

**D3 = {**Rainy, Cold, High, Strong, Warm, Change, No} = Negative example. It forces the general boundary G to become specific.

$G_2$ becomes more specific in $G_3$.

**S2, S3**= {Sunny, Warm, ?, Strong, Water, Same}

**G3**= {<Sunny, ?, ?, ?, ?, ?><?, Warm, ?, ?, ?, ?, ?><?, ?, ?, ?, ?, Same> }

**D4** = {Sunny, Warm, High, Strong, Cool, Change} = Positive Training example, therefore specific hypothesis S is updated from S3to $S_4$.

**S4**= {Sunny, Warm,?, Strong, ?, ?}

**G3, G4**= {<Sunny, ?, ?, ?, ?, ?><?, Warm, ?, ?, ?, ?, ?><?, ?, ?, ?, ?, Same> }

**SOURCE CODE IN PYTHON**

```
import numpy as np
import pandas as pd
data=pd.read_csv('data.csv')
print('The dataset is:')
print(data)
concepts=np.array(data.iloc[:,0:-1])
print('\nThe concepts are:\n',concepts)
target=np.array(data.iloc[:,-1])
print('\nThe target is:\n',target)
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("\nFinal S:",s_final)
print("\nFinal G:",g_final)
```

## OUTPUT:

**The dataset is:**

| | Example | Sky | Air Temp | Humidity | Water | Forecast | Enjoy Sport |
|---|---|---|---|---|---|---|---|
| 0 | D1 | Sunny | Warm | Normal | Warm | Same | Yes |
| 1 | D2 | Sunny | Warm | High | Warm | Same | Yes |
| 2 | D3 | Rainy | Cold | High | Warm | Change | No |
| 3 | D1 | Sunny | Warm | High | Cool | Change | Yes |

**The concepts are:**

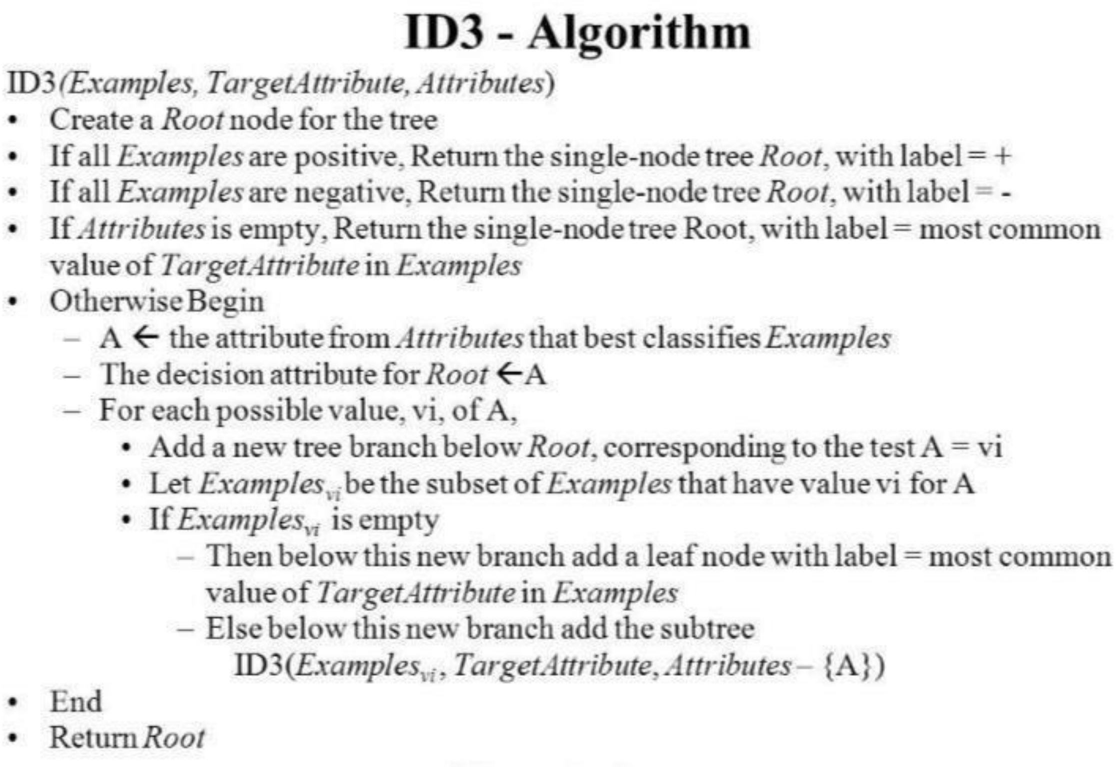
| | | | | | | |
|---|---|---|---|---|---|---|
| D1 | Sunny | Warm | Normal | Strong | Warm | Same |
| D2 | Sunny | Warm | High | Strong | Warm | Same |
| D3 | Rainy | Cold | High | Strong | Warm | Change |
| D1 | Sunny | Warm | High | Strong | Cool | Change |

**The target is:**

```
['Yes' 'Yes' 'No' 'Yes']

Final S: ['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final G: [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]
```

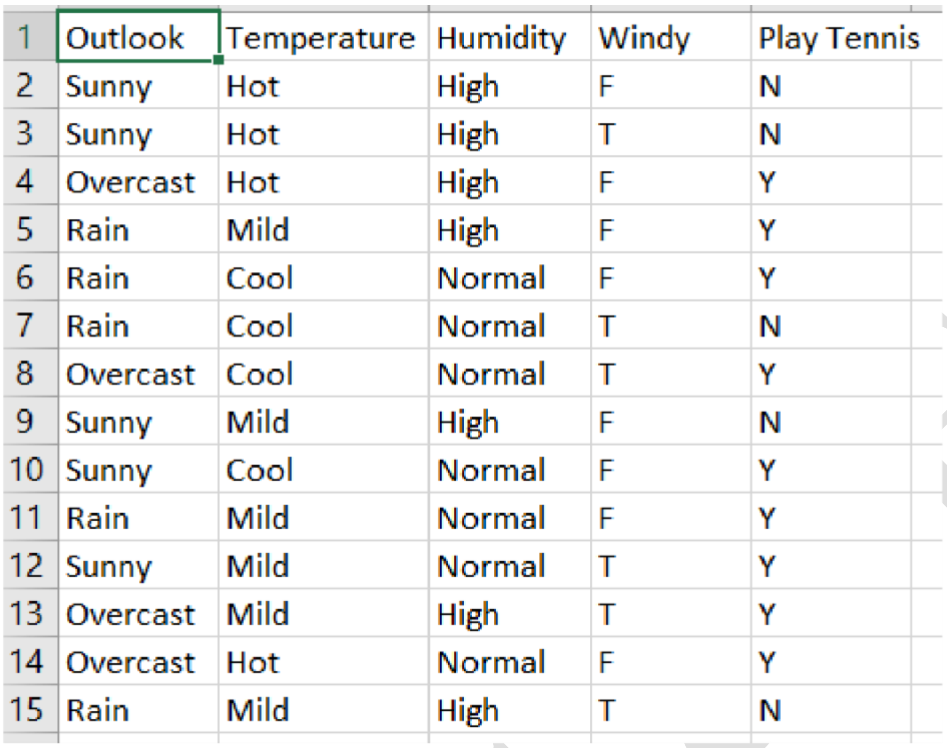
**Program 4**

**Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.**



## ID3 - Algorithm

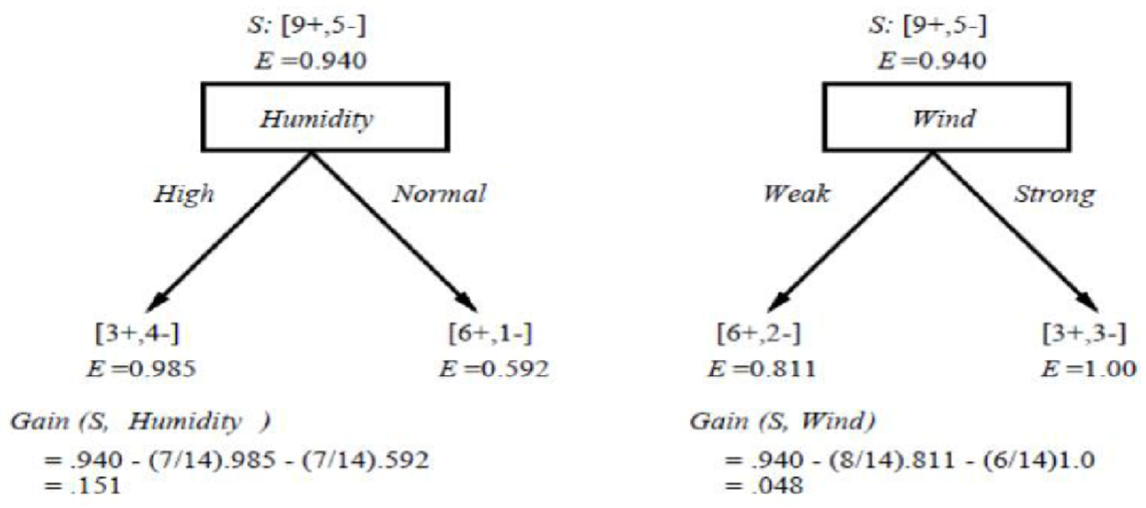
```
ID3(Examples, TargetAttribute, Attributes)
• Create a Root node for the tree
• If all Examples are positive, Return the single-node tree Root, with label = +
• If all Examples are negative, Return the single-node tree Root, with label = -
• If Attributes is empty, Return the single-node tree Root, with label = most common
  value of TargetAttribute in Examples
• Otherwise Begin
    – A ← the attribute from Attributes that best classifies Examples
    – The decision attribute for Root ← A
    – For each possible value, vi, of A,
        • Add a new tree branch below Root, corresponding to the test A = vi
        • Let $Examples_{vi}$ be the subset of Examples that have value vi for A
        • If $Examples_{vi}$ is empty
            – Then below this new branch add a leaf node with label = most common
              value of TargetAttribute in Examples
            – Else below this new branch add the subtree
                ID3($Examples_{vi}$, TargetAttribute, Attributes – {A})
• End
• Return Root
```



| 1 | Outlook | Temperature | Humidity | Windy | Play Tennis |
|---|---|---|---|---|---|
| 2 | Sunny | Hot | High | F | N |
| 3 | Sunny | Hot | High | T | N |
| 4 | Overcast | Hot | High | F | Y |
| 5 | Rain | Mild | High | F | Y |
| 6 | Rain | Cool | Normal | F | Y |
| 7 | Rain | Cool | Normal | T | N |
| 8 | Overcast | Cool | Normal | T | Y |
| 9 | Sunny | Mild | High | F | N |
| 10 | Sunny | Cool | Normal | F | Y |
| 11 | Rain | Mild | Normal | F | Y |
| 12 | Sunny | Mild | Normal | T | Y |
| 13 | Overcast | Mild | High | T | Y |
| 14 | Overcast | Hot | Normal | F | Y |
| 15 | Rain | Mild | High | T | N |

**Compute the Gain and identify which attribute is the best as illustrated below**

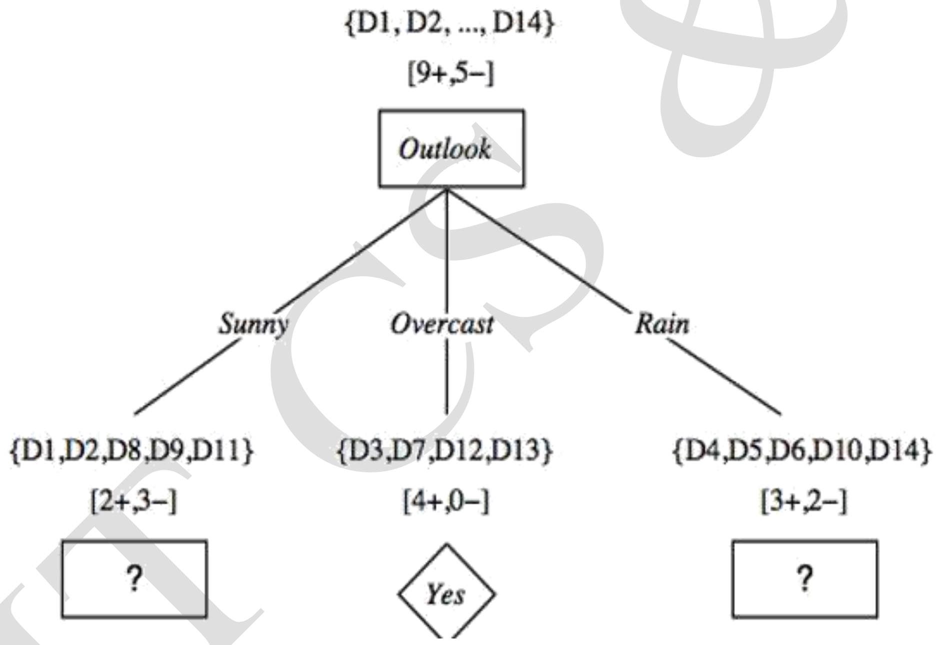
**Which attribute is the best classifier?**



**Which attribute to test at the root?**

- Which attribute should be tested at the root?
    - *Gain*(*S*, *Outlook*) = 0.246
    - *Gain*(*S*, *Humidity*) = 0.151
    - *Gain*(*S*, *Wind*) = 0.048
    - *Gain*(*S*, *Temperature*) = 0.029
- *Outlook* provides the best prediction for the target
- Lets grow the tree:
    - add to the tree a successor for each possible value of *Outlook*
    - partition the training samples according to the value of *Outlook*

**After first step**



**SOURCE CODE IN PYTHON**

```
import pandas as pd
import math
class Node:
    def __init__(self, l):
        self.label = l
        self.branch = {}
def entropy(data):
    total_ex = len(data)
    p_ex = len(data.loc[data['PlayTennis']=='Yes'])
    n_ex = len(data.loc[data['PlayTennis']=='No'])
    en = 0
    if(p_ex>0):
        en = -(p_ex/float(total_ex)) * (math.log(p_ex,2)-math.log(total_ex,2))
    if(n_ex>0):
        en += -(n_ex/float(total_ex)) * (math.log(n_ex,2)-math.log(total_ex,2))
    return en
def gain(en_s,data_s,attrib):
    values = set(data_s[attrib])
    gain = en_s
    for value in values:
        gain -= len(data_s.loc[data_s[attrib]==value])/float(len(data_s)) *
entropy(data_s.loc[data_s[attrib]==value])
    return gain
def get_attr(data):
    en_s = entropy(data)
    attribute = ""
    max_gain = 0
    for attr in data.columns[:len(data.columns)-1]:
        g = gain(en_s, data, attr)
        if g > max_gain:
            max_gain = g
            attribute = attr
    return attribute
def decision_tree(data):
    root = Node("NULL")
    if(entropy(data)==0):
        if(len(data.loc[data[data.columns[-1]]=="Yes"]) == len(data)):
            root.label = "Yes"
        else:
            root.label = "No"
        return root

    if(len(data.columns)==1):
        return
    else:
        attr = get_attr(data)
        root.label = attr
        values = set(data[attr])
        for value in values:
            root.branch[value] = decision_tree(data.loc[data[attr]==value].drop(attr,axis=1))
        return root
def get_rules(root, rule, rules):
    if not root.branch:
        rules.append(rule[:-1]+"=>"+root.label)
        return rules
    for val in root.branch:
        get_rules(root.branch[val], rule+root.label+"="+str(val)+"^", rules)
    return rules
def test(tree, test_str):
    if not tree.branch:
        return tree.label
    if str(test_str[tree.label])=="True" or str(test_str[tree.label])=="False":
         return tree.branch(str(test_str[tree.label]))
    return test(tree.branch[str(test_str[tree.label])], test_str)
data = pd.read_csv("tennis.csv")
tree = decision_tree(data)
rules = get_rules(tree,"",[])
for rule in rules:
    print(rule)
test_str = {}
print("Enter the test case input: ")
for attr in data.columns[:-1]:
    test_str[attr] = input(attr+": ")
print(test_str)
print(test(tree, test_str))
```

**OUTPUT 1:**

```
{'Overcast', 'Rain', 'Sunny'}
{'Hot', 'Mild', 'Cool'}
{'Normal', 'High'}
{'T', 'F'}
{'Mild', 'Cool'}
{'Normal', 'High'}
{'T', 'F'}
{'Hot', 'Mild', 'Cool'}
{'Normal', 'High'}
{'T', 'F'}
['Outlook=Overcast  => Y', 'Outlook=Rain ^ Windy=T  => N', 'Outlook=Rain ^ Windy=F  =>
Y',
'Outlook=Sunny ^ Humidity=Normal  => Y', 'Outlook=Sunny ^ Humidity=High  => N']
Enter test case input
Outlook: Rain
Temperature: Mild
Humidity: High
Windy: T
{'Outlook': 'Rain', 'Temperature': 'Mild', 'Humidity': 'High', 'Windy': 'T'}
N
```

**OUTPUT 2:**

```
{'Overcast', 'Rain', 'Sunny'}
{'Hot', 'Mild', 'Cool'}
{'Normal', 'High'}
{'T', 'F'}
{'Mild', 'Cool'}
{'Normal', 'High'}
{'T', 'F'}
{'Hot', 'Mild', 'Cool'}
{'Normal', 'High'}
{'T', 'F'}
['Outlook=Overcast => Y', 'Outlook=Rain ^ Windy=T  => N', 'Outlook=Rain ^ Windy=F  =>
Y',
'Outlook=Sunny ^ Humidity=Normal  => Y', 'Outlook=Sunny ^ Humidity=High  => N']
```

**Enter test case input**

**Outlook:** Rain

**Temperature:** Mild

**Humidity:** Normal

**Windy:** F

```
{'Outlook': 'Rain', 'Temperature': 'Mild', 'Humidity': 'Normal', 'Windy': 'F'}
```

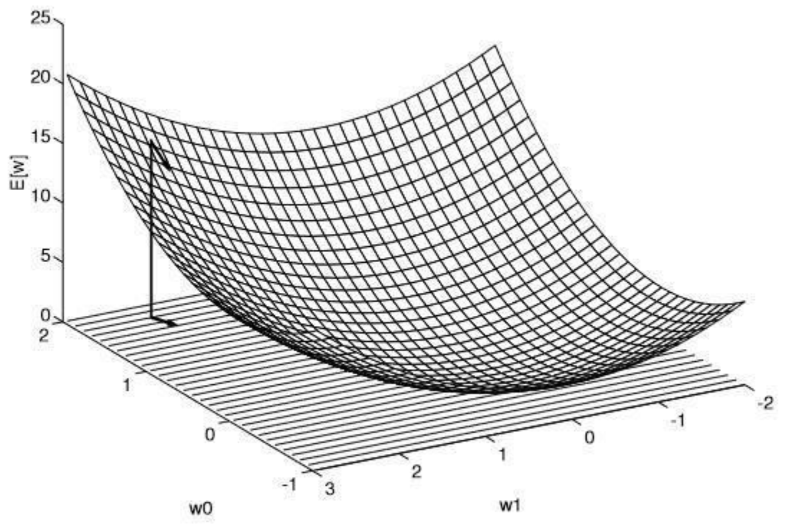
**Y**

## Program 5

**Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.**

**Back propagation Algorithm**

### Gradient Descent

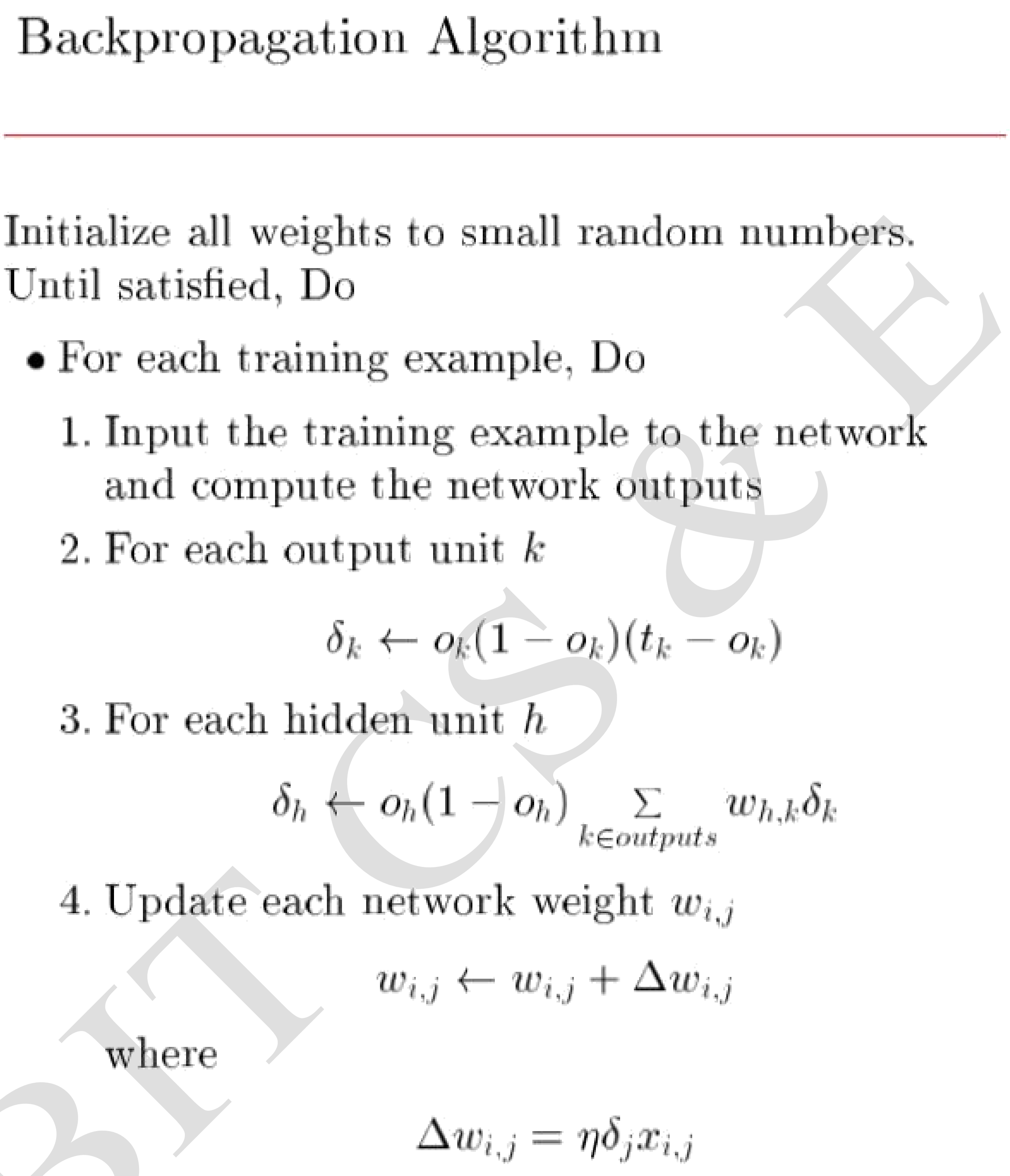


Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training rule:

$$\Delta\vec{w} = -\eta\nabla E[\vec{w}]$$



## Backpropagation Algorithm

```
Initialize all weights to small random numbers.
Until satisfied, Do
  • For each training example, Do
    1. Input the training example to the network
       and compute the network outputs
    2. For each output unit $k$
           $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
    3. For each hidden unit $h$
           $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$
    4. Update each network weight $w_{i,j}$
           $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$
       where
           $\Delta w_{i,j} = \eta \delta_j x_{i,j}$
```

The Back propagation algorithm is a supervised learning method for multilayer feed-forward networks from the field of Artificial Neural Networks, the back propagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. Back propagation can be used for both classification and regression problems.

Algorithm works in four steps

1. Initialize Network.
2. Forward Propagate.
3. Back Propagate Error.
4. Train the network

**1. Initialize Network**

Initialize the network weights to small random numbers in the range of 0 to 1. The function named **initialize_network()** that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs. **2. Forward Propagate**

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. We call this forward-propagation.

We can break forward propagation down into three parts:

- Neuron Activation.
- Neuron Transfer.
- Forward Propagation.

**2.1 Neuron Activation:**. Neuron activation is calculated as the weighted sum of the inputs. **activation = sum(weight_i * input_i) + bias**

Where weight is a network weight, input is an input, i is the index of a weight or an input and bias is a special weight that has no input to multiply with (or you can think of the input as always being 1.0). The implementation of this is in a function named **activate ().**

**2.2Neuron Transfer**

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used. It is traditional to use the sigmoid activation function.We can transfer an activation function using the sigmoid function as follows: **output = 1 / (1 + e^(-activation))**

The function named **transfer()** that implements the sigmoid equation

**2. Forward Propagation**

Forward propagating an input is straightforward. All of the outputs from one layer become inputs to the neurons on the next layer. The function named **forward_propagate ()** that implements the forward propagation for a row of data from our dataset with our neural network.

## 3. Error Backpropagation

**3.1** The first step is to calculate the error for each output neuron, this will give us our error signal (input) to propagate backwards through the network.  e error for a given neuron can be calculated as follows:   **error = (expected - output) * transfer_derivative(output)**
Where expected is the expected output value for the neuron, output is the output value for the neuron and **transfer_derivative ()** calculates the slope of the neuron's output value

**3.2** The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:

**error = (weight_k * error_j) * transfer_derivative(output)**

Where error_j is the error signal from the jth neuron in the output layer, weight_k is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron
The function named **backward_propagate_error ()** that implements this procedure.

## 4. Train Network

The network is trained using stochastic gradient descent. This involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, back propagating the error and updating the network weights. This part is broken down into two sections:

- Update Weights.
- Train Network.

### 4.1 Update Weights

Once errors are calculated for each neuron in the network via the back propagation method above, they can be used to update weights. Network weights are updated as follows:

**weight = weight + learning_rate * error * input**

The function named **update_weights ()** that updates the weights for a network given an input row of data, a learning rate and assume that a forward and backward propagation have already been performed.

### 4.2 Train Network

As mentioned, the network is updated using stochastic gradient descent. This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset. The function **train_network**() implements the training of an already initialized neural network with a given training dataset, learning rate, fixed number of epochs and an expected number of output values. The sum squared error between the expected output and the network output is accumulated each epoch and printed.

**SOURCE CODE IN PYTHON**

```
from math import exp
from random import seed
from random import random
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in
range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
range(n_outputs)]
    network.append(output_layer)
    return network
# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))
# Forward propagate input to a network output
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
```

```
 # Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)
 # Backpropagate error and store in neurons
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'])
 # Update network weights with error
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']
 # Train a network for a fixed number of epochs
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
           # print(expected)
            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))
 # Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [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)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
 print(layer)
```

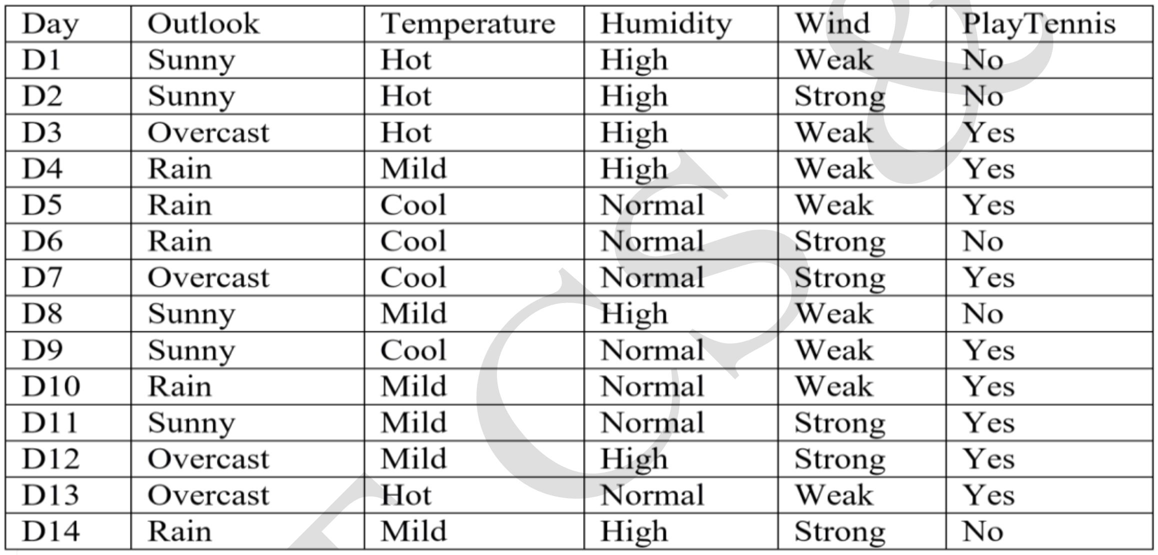
## OUTPUT:

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output':
0.029980305604426185, 'delta': -0.0059546604162323625}, {'weights':
[0.37711098142462157, 0.06259098945529 89, 0.2765123702642716],
 'output': 0.9456229000211323, 'delta': 0.0026279652850863837}]
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output':
0.23648794202357587, 'delta': -0.04270059278364587}, {'weights': [-2.5584149848484263,
1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta':
0.03803132596437354}]
```

## Program 6

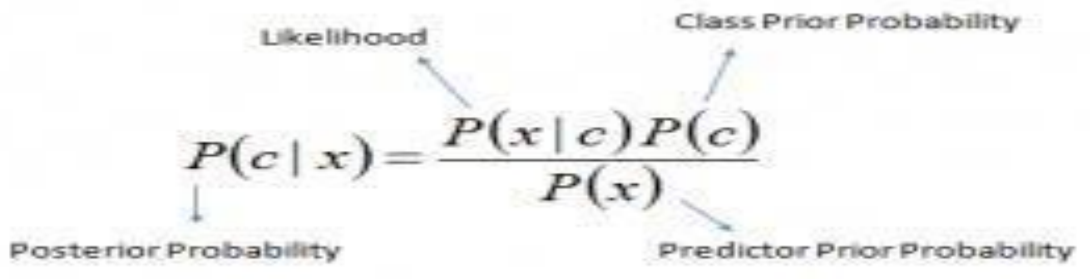
**Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets. naïve Bayesian classifier**

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'. Data Set :PlayTennis example



| Day | Outlook | Temperature | Humidity | Wind | PlayTennis |
|---|---|---|---|---|---|
| D1 | Sunny | Hot | High | Weak | No |
| D2 | Sunny | Hot | High | Strong | No |
| D3 | Overcast | Hot | High | Weak | Yes |
| D4 | Rain | Mild | High | Weak | Yes |
| D5 | Rain | Cool | Normal | Weak | Yes |
| D6 | Rain | Cool | Normal | Strong | No |
| D7 | Overcast | Cool | Normal | Strong | Yes |
| D8 | Sunny | Mild | High | Weak | No |
| D9 | Sunny | Cool | Normal | Weak | Yes |
| D10 | Rain | Mild | Normal | Weak | Yes |
| D11 | Sunny | Mild | Normal | Strong | Yes |
| D12 | Overcast | Mild | High | Strong | Yes |
| D13 | Overcast | Hot | Normal | Weak | Yes |
| D14 | Rain | Mild | High | Strong | No |

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c).



$$P(c \mid x) = \frac{P(x \mid c)P(c)}{P(x)}$$

$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

Above,

- P(c|x) is the posterior probability of class (c, target) given predictor (x, attributes).
- P(c) is the prior probability of class.
- P(x|c) is the likelihood which is the probability of predictor given class.
- P(x) is the prior probability of predictor.

So here we have 4 attributes. What we need to do is to create "look-up tables" for each of these attributes, and write in the probability that a game of tennis will be played based on this attribute. In these tables we have to note that there are 5 cases of not being able to play a game, and 9 cases of being able to play a game.

We also must note the following probabilities for P(C):
P(Play=Yes) = 9/14
P(Play=No) = 5/14

| OUTLOOK | Play = Yes | Play = No | Total |
|---|---|---|---|
| Sunny | 2/9 | 3/5 | 5/14 |
| Overcast | 4/9 | 0/5 | 4/14 |
| Rain | 3/9 | 2/5 | 5/14 |

| TEMPERATURE | Play = Yes | Play = No | Total |
|---|---|---|---|
| Hot | 2/9 | 2/5 | 4/14 |
| Mild | 4/9 | 2/5 | 6/14 |
| Cool | 3/9 | 1/5 | 4/14 |

| HUMIDITY | Play = Yes | Play = No | Total |
|---|---|---|---|
| High | 3/9 | 4/5 | 7/14 |
| Normal | 6/9 | 1/5 | 7/14 |

| **WIND** | **Play = Yes** | **Play = No** | **Total** |
|---|---|---|---|
| **Strong** | **3/9** | **3/5** | **6/14** |
| **Weak** | **6/9** | **2/5** | **8/14** |

**Testing**

For this, say we were given a new instance, and we want to know if we can play a game or not, then we need to look up the results from the tables above. So, this new instance is:
X = (Outlook=Sunny, Temperature=Cool, Humidity=High, Wind=Strong)

Firstly we look at the probability that we can play the game, so we use the lookup tables to get:

P(Outlook=Sunny | Play=Yes) = 2/9
P(Temperature=Cool | Play=Yes) = 3/9
P(Humidity=High | Play=Yes) = 3/9
P(Wind=Strong | Play=Yes) = 3/9
P(Play=Yes) = 9/14

Next we consider the fact that we cannot play a game:

P(Outlook=Sunny | Play=No) = 3/5
P(Temperature=Cool | Play=No) = 1/5
P(Humidity=High | Play=No) = 4/5
P(Wind=Strong | Play=No) = 3/5
P(Play=No) = 5/14

Then, using those results, you have to multiple the whole lot together. So you multiple all the probabilities for Play=Yes such as:

P(X|Play=Yes)P(Play=Yes) = (2/9) * (3/9) * (3/9) * (3/9) * (9/14) = 0.0053

And this gives us a value that represents 'P(X|C)P(C)', or in this case 'P(X|Play=Yes)P(Play=Yes)'.

We also have to do the same thing for Play=No:

P(X|Play=No)P(Play=No) = (3/5) * (1/5) * (4/5) * (3/5) * (5/14) = 0.0206

Finally, we have to divide both results by the evidence, or 'P(X)'. The evidence for both equations is the same, and we can find the values we need within the 'Total' columns of the look-up tables. Therefore:
P(X) = P(Outlook=Sunny) * P(Temperature=Cool) * P(Humidity=High) * P(Wind=Strong) P(X)

= (5/14) * (4/14) * (7/14) * (6/14)
P(X) = 0.02186
Then, dividing the results by this value:

P(Play=Yes | X) = 0.0053/0.02186 = 0.2424
P(Play=No | X) = 0.0206/0.02186 = 0.9421

So, given the probabilities, can we play a game or not? To do this, we look at both probabilities and see which once has the highest value, and that is our answer. Therefore:

P(Play=Yes | X) = 0.2424
P(Play=No | X) = 0.9421

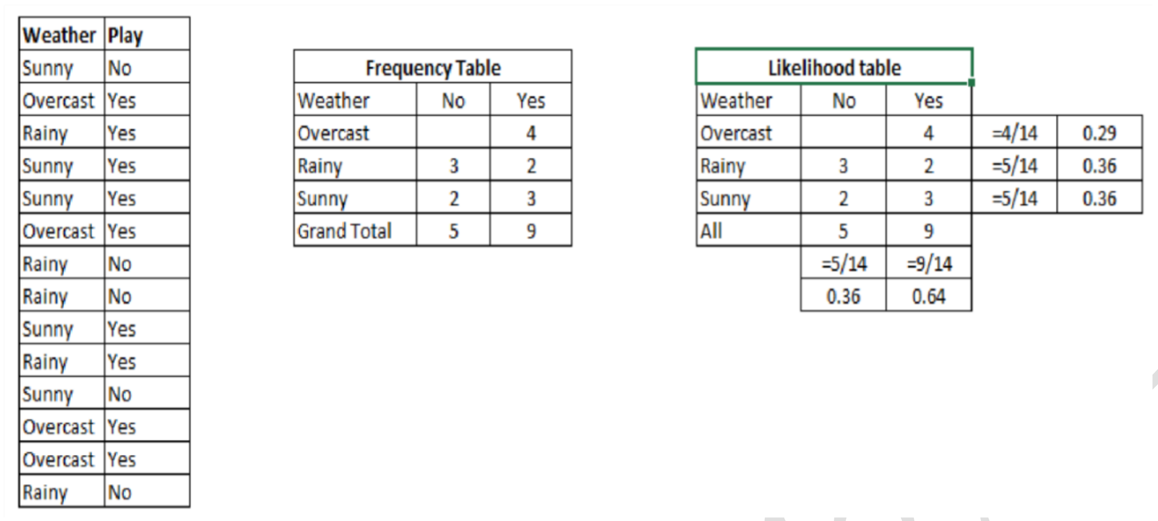
Since 0.9421 is greater than 0.2424 then the answer is 'no', we cannot play a game of tennis today.

Let's understand it using an example. Below I have a training data set of weather and corresponding target variable 'Play' (suggesting possibilities of playing). Now, we need to classify whether players will play or not based on weather condition. Let's follow the below steps to perform it.

**Step 1:** Convert the data set into a frequency table

**Step 2:** Create Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing is 0.64.

**Step 3:** Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.



| Weather | Play |
|---|---|
| Sunny | No |
| Overcast | Yes |
| Rainy | Yes |
| Sunny | Yes |
| Sunny | Yes |
| Overcast | Yes |
| Rainy | No |
| Rainy | No |
| Sunny | Yes |
| Rainy | Yes |
| Sunny | No |
| Overcast | Yes |
| Overcast | Yes |
| Rainy | No |

Frequency Table

| Weather | No | Yes |
|---|---|---|
| Overcast | | 4 |
| Rainy | 3 | 2 |
| Sunny | 2 | 3 |
| Grand Total | 5 | 9 |

Likelihood table

| Weather | No | Yes | | |
|---|---|---|---|---|
| Overcast | | 4 | =4/14 | 0.29 |
| Rainy | 3 | 2 | =5/14 | 0.36 |
| Sunny | 2 | 3 | =5/14 | 0.36 |
| All | 5 | 9 | | |
| | =5/14 | =9/14 | | |
| | 0.36 | 0.64 | | |

**SOURCE CODE IN PYTHON**

```
 import pandas as pd
#reading the dataset
data=pd.read_csv('data.csv')
#calculating the total no.,no. of positive and no. of negative instances
te=len(data)
np=len(data.loc[data[data.columns[-1]]=='Yes'])
nn=te-np
#dividing the dataset into training and test
training=data.sample(frac=0.75,replace=False)
#test=pd.concat([data, training, training]).drop_duplicates(keep=False)
test=pd.concat([data, training]).drop_duplicates(keep=False)
print('Training Set : \n',training)
print('\nTest Data Set : \n',test)
#For every value of each attribute calculate the negative and positive probability
prob={}
for col in training.columns[:-1]:
   prob[col]={}
   vals=set(data[col])
   for val in vals:
      temp=training.loc[training[col]==val]
      pe=len(temp.loc[temp[temp.columns[-1]]=='Yes'])
      ne=len(temp)-pe
      prob[col][val]=[pe/np,ne/nn]
#Using Bayes Theorem to Predict the output
prediction=[]
right_prediction=0
for i in range(len(test)):
   row=test.iloc[i,:]
```

```
    fpp=np/te
    fpn=nn/te
    for col in test.columns[:-1]:
        fpp*=prob[col][row[col]][0]
        fpn*=prob[col][row[col]][1]
    if fpp>fpn:
        prediction.append('Yes')
    else:
        prediction.append('No')
    if prediction[-1]==row[-1]:
        right_prediction+=1
#output
print('\nActual Values : ',list(test[test.columns[-1]]))
print('Predicted : ',prediction)
print('Accuracy : ',right_prediction/len(test))
```

## OUTPUT 1:

**Training Set :**

| | Outlook | Temperature | Humidity | Windy | PlayTennis |
|---|---|---|---|---|---|
| 5 | Rainy | Coo l | Normal | True | No |
| 0 | Sunny | Hot | High | False | No |
| 3 | Rainy | Mild | High | False | Yes |
| 9 | Rainy | Mild | Normal | False | Yes |
| 6 | Overcast | Cool | Normal | True | Yes |
| 4 | Rainy | Cool | Normal | False | Yes |
| 8 | Sunny | Cool | Normal | False | Yes |
| 1 | Sunny | Hot | High | True | No |
| 7 | Sunny | Mild | High | False | No |
| 10 | Sunny | Mild | Normal | True | Yes |

**Test Data Set :**

| | Outlook | Temperature | Humidity | Windy | PlayTennis |
|---|---|---|---|---|---|
| **2** | Overcast | Hot | High | False | Yes |
| **11** | Overcast | Mild | | | |
| **12** | Overcast | Hot | Normal | False | Yes |
| **13** | Rainy | Mild | High | True | No |

```
Actual Values :  ['Yes', 'Yes', 'Yes', 'No']
Predicted :  ['Yes', 'No', 'No', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No', 'Yes', 'Yes', 'No',
'No'] Accuracy :  0.5
```

## <u>OUTPUT 2:</u>

**Training Set :**

| | Outlook | Temperature | Humidity | Windy | PlayTennis |
|---|---|---|---|---|---|
| 13 | Rainy | Mild | High | True | No |
| 4 | Rainy | Cool | Normal | False | Yes |
| 8 | Sunny | Cool | Normal | False | Yes |
| 10 | Sunny | Mild | Normal | True | Yes |
| 5 | Rainy | Cool | Normal | True | No |
| 11 | Overcast | Mild | High | True | Yes |
| 7 | Sunny | Mild | High | False | No |
| 1 | Sunny | Hot | High | True | No |
| 6 | Overcast | Cool | Normal | True | Yes |
| 2 | Overcast | Hot | High | False | Yes |

**Test Data Set :**

|  | Outlook | Temperature | Humidity | Windy | PlayTennis |
|---|---|---|---|---|---|
| 0 | Sunny | Hot | High | False | No |
| 3 | Rainy | Mild | High | False | Yes |
| 9 | Rainy | Mild | Normal | False | Yes |
| 12 | Overcast | Hot | Normal | False | Yes |

```
Actual Values :  ['No', 'Yes', 'Yes', 'Yes']
Predicted :  ['No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes', 'Yes', 'Yes', 'Yes',
'Yes']
Accuracy :  0.75
```

## Program 7

**Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.**

### Task

This program has to cluster the given IRIS dataset using K-means and EM algorithm. The accuracy of the clustering results is compared.

**Expectation-Maximization (EM) algorithm**

**Step 1:** An initial guess is made for the model's parameters and a probability distribution is created. This is sometimes called the "E-Step" for the "Expected" distribution.

**Step 2:** Newly observed data is fed into the model.

**Step 3:** The probability distribution from the E-step is drawn to include the new data. This is sometimes called the "M-step."

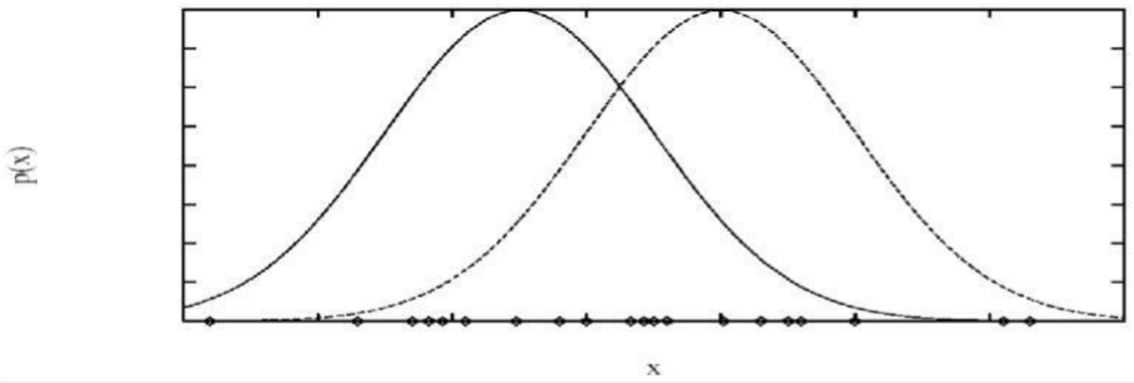
**Step 4:** Steps 2 through 4 are repeated until stability.

**Algorithm :**

## Expectation Maximization (EM) Algorithm

- When to use:
    - Data is only partially observable
    - Unsupervised clustering (target value unobservable)
    - Supervised learning (some instance attributes unobservable)
- Some uses:
    - Train Bayesian Belief Networks
    - Unsupervised clustering (AUTOCLASS)
    - Learning Hidden Markov Models

## Generating Data from Mixture of *k* Gaussians



- **Each instance x generated by**

1. Choosing one of the *k* Gaussians with uniform probability
2. Generating an instance at random according to that Gaussian

## EM for Estimating *k* Means

- Given:
  - Instances from *X* generated by mixture of *k* Gaussian distributions
  - Unknown means $<\mu_1,\ldots,\mu_k>$ of the *k* Gaussians
  - Don't know which instance $x_i$ was generated by which Gaussian
- Determine:
  - Maximum likelihood estimates of $<\mu_1,\ldots,\mu_k>$
- Think of full description of each instance as

  $y_i = < x_i, z_{i1}, z_{i2}>$ where
  - $z_{ij}$ is 1 if $x_i$ generated by *j*th Gaussian
  - $x_i$ observable
  - $z_{ij}$ unobservable

## • EM Algorithm: Pick random initial $h = <\mu_1, \mu_2>$ then iterate

**E step:** Calculate the expected value $E[z_{ij}]$ of each hidden variable $z_{ij}$, assuming the current hypothesis $h = <\mu_1, \mu_2>$ holds.

$$E[z_{ij}] = \frac{p(x = x_i|\mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i|\mu = \mu_n)} = \frac{e^{-\frac{1}{2\sigma^2}(x_i-\mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i-\mu_n)^2}}$$

**M step:** Calculate a new maximum likelihood hypothesis $h' = <\mu'_1, \mu'_2>$, assuming the value taken on by each hidden variable $z_{ij}$ is its expected value $E[z_{ij}]$ calculated above. Replace $h = <\mu_1, \mu_2>$ by $h' = <\mu'_1, \mu'_2>$.

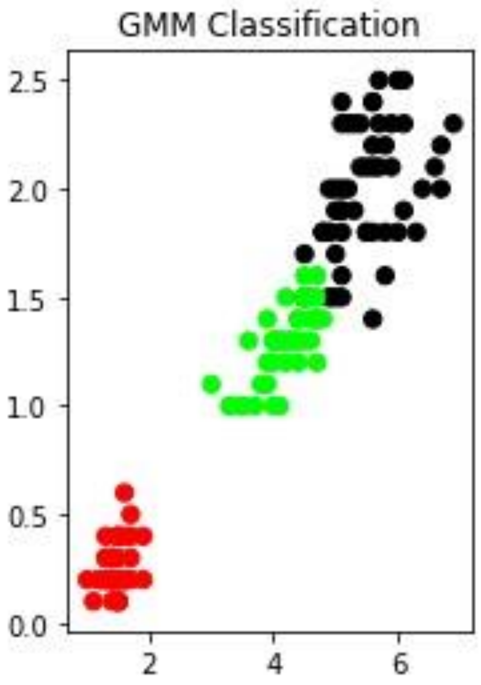
$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}]\ x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

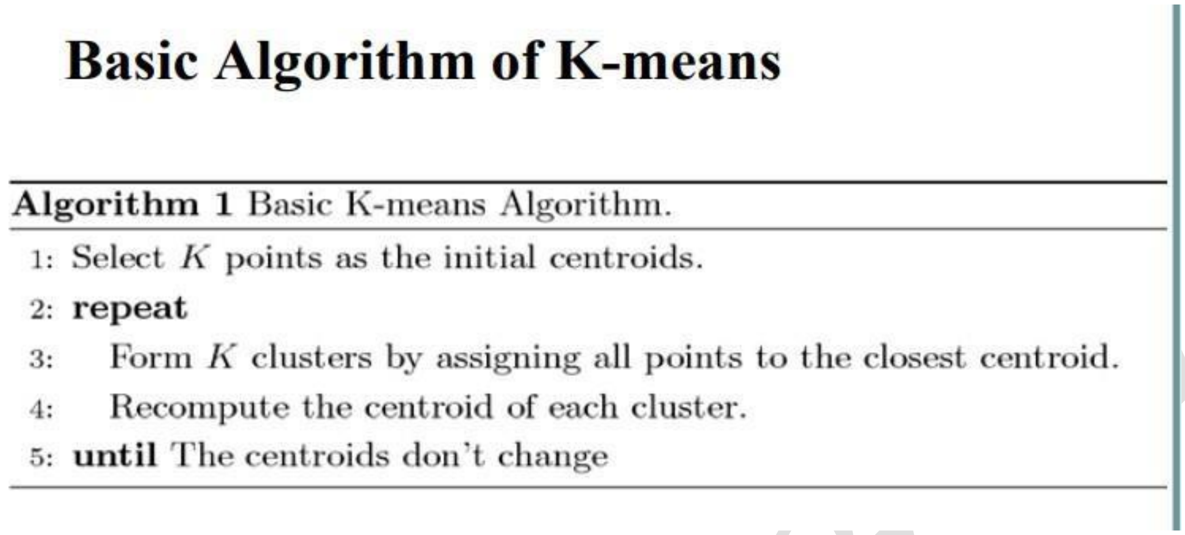
**SOURCE CODE IN PYTHON: EM ALGORITHM**

```
from sklearn import datasets
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
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']
plt.figure(figsize=(14,7))
colormap = np.array(['red','lime','black'])
plt.subplot(1,2,1)
plt.scatter(x.Sepal_Length, x.Sepal_Width, c = colormap[y.Targets], s=40)
plt.title('Sepal')
plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = colormap[y.Targets], s=40)
plt.title('Petal')
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_gmm = gmm.predict(xs)
plt.figure(figsize=(14,7))
colormap = np.array(['red','lime','black'])
plt.subplot(1,2,1)
plt.scatter(x.Petal_Length, x.Petal_Width, c = colormap[y.Targets], s=40)
plt.title('Real')
plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = colormap[y_gmm], s=40)
plt.title('EM')
print("accuracy_score", accuracy_score(y.Targets, y_gmm))
print("confusion_matrix\n", confusion_matrix(y.Targets, y_gmm))
```

## OUTPUT :

**Accuracy Score = 0.9666666666666667**





## Basic Algorithm of K-means

**Algorithm 1** Basic K-means Algorithm.

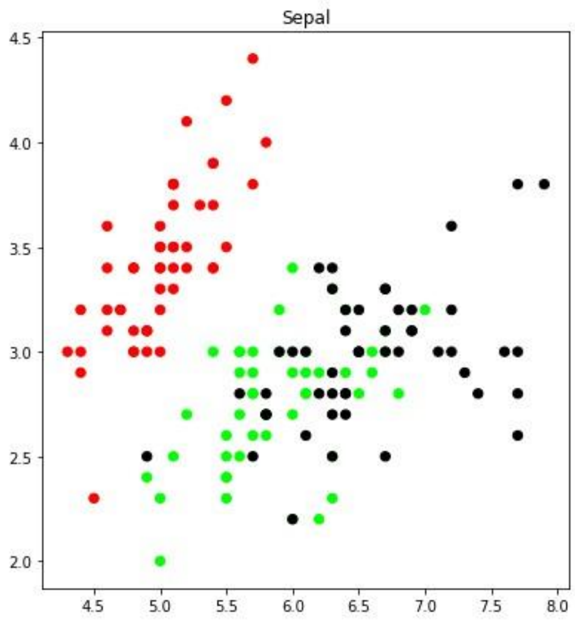
```
1: Select $K$ points as the initial centroids.
2: repeat
3:     Form $K$ clusters by assigning all points to the closest centroid.
4:     Recompute the centroid of each cluster.
5: until The centroids don't change
```

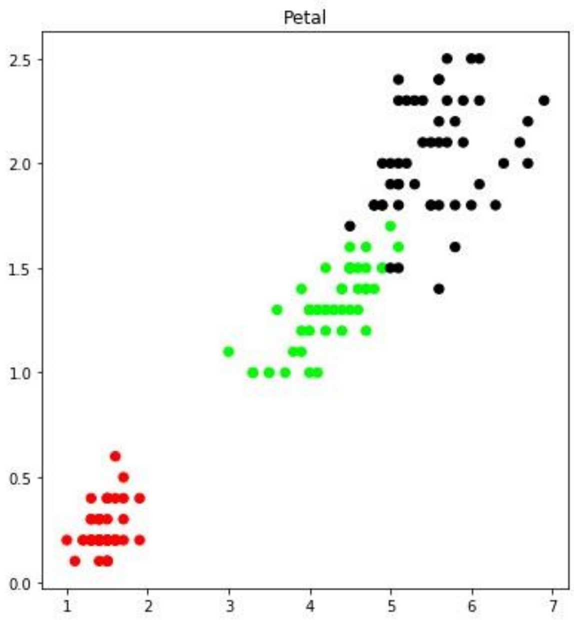
**SOURCE CODE IN PYTHON: K-means ALGORITHM**

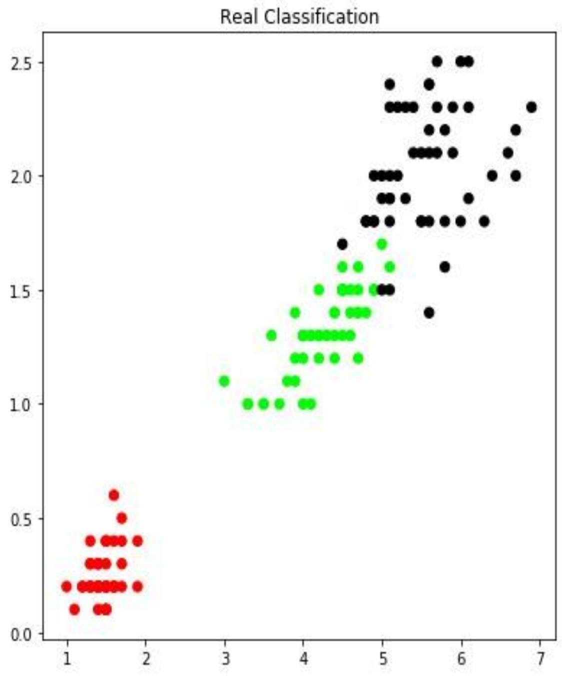
```
 from sklearn import datasets
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from sklearn.cluster import KMeans

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']

plt.figure(figsize=(14,7))
colormap = np.array(['red','lime','black'])
plt.subplot(1,2,1)
plt.scatter(x.Sepal_Length, x.Sepal_Width, c = colormap[y.Targets], s=40)
plt.title('Sepal')
plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = colormap[y.Targets], s=40)
plt.title('Petal')

model = KMeans(n_clusters = 3)
model.fit(x)

plt.figure(figsize=(14,7))
colormap = np.array(['red','lime','black'])
plt.subplot(1,2,1)
plt.scatter(x.Petal_Length, x.Petal_Width, c = colormap[y.Targets], s=40)
plt.title('Real')
plt.subplot(1,2,2)
plt.scatter(x.Petal_Length, x.Petal_Width, c = colormap[model.labels_], s=40)
plt.title('kmeans')

print("accuracy_score", accuracy_score(y.Targets, model.labels_))
print("confusion_matrix\n", confusion_matrix(y.Targets,model.labels_))
```

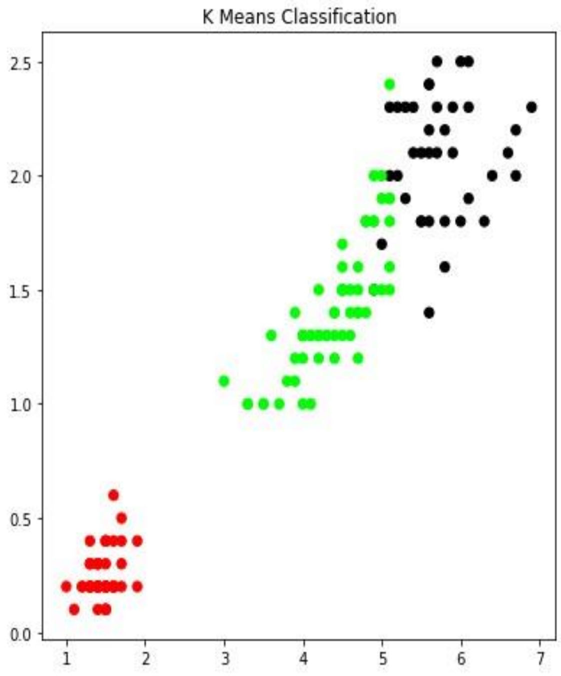
## OUTPUT :

```
Accuracy Score = 0.8933333333333333
```









## Program 8

**Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.**

KNN falls in the supervised learning family of algorithms. Informally, this means that we are given alabeled dataset consisting of training observations (x, y) and would like to capture the relationshipbetween x and y.
More formally, our goal is to learn a function h: X→Y so that given an unseenobservation x, h(x) can confidently predict the corresponding output y.
The KNN classifier is also a non-parametric and instance-based learning algorithm.

a) Non-parametric means it makes no explicit assumptions about the functional form of h, avoiding the dangers of m is modeling the underlying distribution of the data. For example, suppose our data is highly non-Gaussian but the learning model we choose assumes a Gaussian form. In that case, our algorithm would make extremely poor predictions.
b) Instance-based learning means that our algorithm doesn't explicitly learn a model. Instead, itchooses to memorize the training instances which are subsequently used as "knowledge" forthe prediction phase. Concretely, this means that only when a query to our database is made (i.e. when we ask it to predict a label given an input), will the algorithm use the traininginstances to spit out an answer.

In the classification setting, the K-nearest neighbor algorithm essentially boils down to forming amajority vote between the K most similar instances to a given "unseen" observation. Similarity isdefined according to a distance metric between two data points.This program classifies the IRIS data set examples by using the k-Nearest Neighbour algorithm. The new instance has to be classified based on its k nearest neighbors.

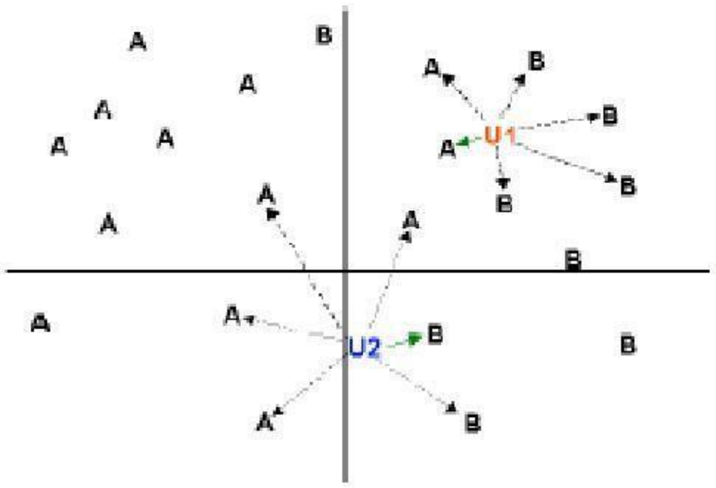
### k Nearest Neighbour ALGORITHM:

Let m be the number of training data samples. Let p be an unknown point.

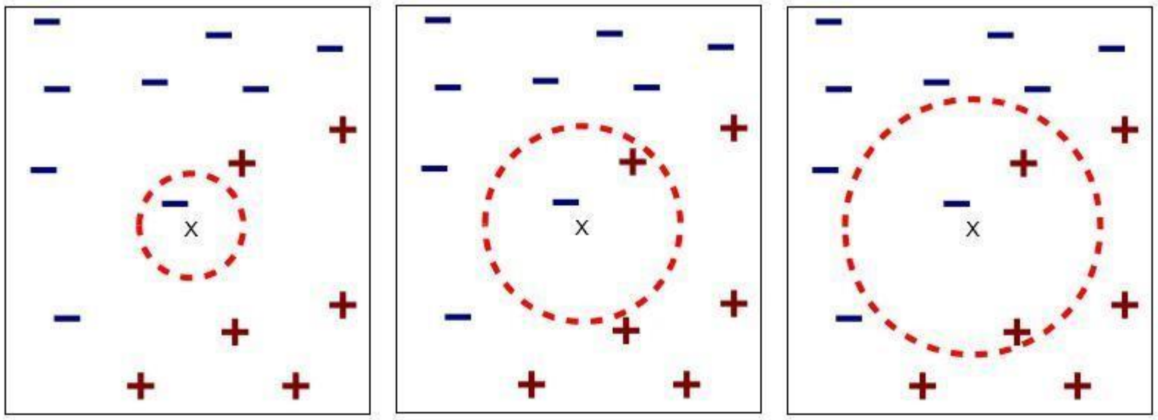
1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. for i=0 to m:
   Calculate Euclidean distance d (arr[i], p).
3. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
4. Return the majority label among S.

## K-Nearest-Neighbor Algorithm

- **Principle: points (documents) that are close in the space belong to the same class**



## Definition of Nearest Neighbor



(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

**SOURCE CODE IN PYTHON**

```
import csv
import random
import math
import operator
def loadDataset(filename, split, trainingSet=[], testSet=[]):
with open(filename) as csvfile:
    lines = csv.reader(csvfile)
    dataset = list(lines)
    for x in range(len(dataset)-1):
        for y in range(4):
            dataset[x][y] = float(dataset[x][y])
        if random.random() < split:
            trainingSet.append(dataset[x])
        else:
            testSet.append(dataset[x])
def euclideanDistance(instance1, instance2, length):
distance = 0
for x in range(length):
    distance += pow((instance1[x] - instance2[x]), 2)
return math.sqrt(distance)

def getNeighbors(trainingSet, testInstance, k):
distances = []
length = len(testInstance)-1
for x in range(len(trainingSet)):
    dist = euclideanDistance(testInstance, trainingSet[x], length)
    distances.append((trainingSet[x], dist))
distances.sort(key=operator.itemgetter(1))
neighbors = []
for x in range(k):
    neighbors.append(distances[x][0])
return neighbors

def getResponse(neighbors):
classVotes = {}
for x in range(len(neighbors)):
    response = neighbors[x][-1]
    if response in classVotes:
            classVotes[response] += 1
    else:
            classVotes[response] = 1
```

```
sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reverse=True)
return sortedVotes[0][0]

def getAccuracy(testSet, predictions):
correct = 0
for x in range(len(testSet)):
    if testSet[x][-1] == predictions[x]:
            correct += 1
return (correct/float(len(testSet))) * 100.0

def main():
# prepare data
trainingSet=[]
testSet=[]
split = 0.67
loadDataset('KNN-input.csv', split, trainingSet, testSet)
print ('\n Number of Training data: ' + (repr(len(trainingSet))))
print (' Number of Test Data: ' + (repr(len(testSet))))
# generate predictions
predictions=[]
k = 3
print('\n The predictions are: ')
for x in range(len(testSet)):
    neighbors = getNeighbors(trainingSet, testSet[x], k)
    result = getResponse(neighbors)
    predictions.append(result)
    print(' predicted=' + repr(result) + ', actual=' + repr(testSet[x][-1]))
accuracy = getAccuracy(testSet, predictions)
print('\n The Accuracy is: ' + repr(accuracy) + '%')

main()
```

**OUTPUT:**

```
 Number of Training data: 99
 Number of Test Data: 50

 The predictions are:   predicted='Iris-setosa',
actual='Iris-setosa'  predicted='Iris-setosa', actual='Iris-
setosa'  predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
```

```
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-setosa', actual='Iris-setosa'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-virginica', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-virginica', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-versicolor', actual='Iris-versicolor'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
```

```
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
predicted='Iris-virginica', actual='Iris-virginica'
```

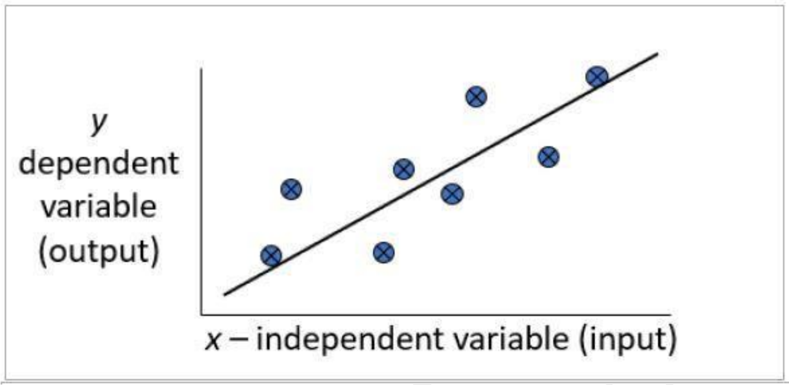
**The Accuracy is: 96.0%**

## Program 9

**Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.**

**Regression** is a technique from statistics that is used to predict values of a desired target quantity when the target quantity is continuous.

- In regression, we seek to identify (or estimate) a continuous variable y associated with a given input vector x.
- y is called the dependent variable.
- x is called the independent variable.



Loess/Lowess Regression: Loess regression is a nonparametric technique that uses *local weighted* regression to fit a smooth curve through points in a scatter plot.
Lowess Algorithm: Locally weighted regression is a very powerful non-parametric model used in statistical learning .Given a *dataset* X, y, we attempt to find a *model* parameter β(x) that minimizes *residual sum of weighted squared errors*. The weights are given by a *kernel function(k or w)* which can be chosen arbitrarily .

**Locally Weighted Regression Algorithm:**

1. Read the Given data Sample to X and the curve (linear or non linear) to Y
2. Set the value for Smoothening parameter or free parameter say τ 3. Set the bias /Point of interest set X0 which is a subset of X 4. Determine the weight matrix using:

$$w(x, x_0) = e^{-\frac{(x-x_0)^2}{2\tau^2}}$$

5. Determine the value of model term parameter β using :

$$\hat{\beta}(x_o) = (X^T W X)^{-1} X^T W y$$

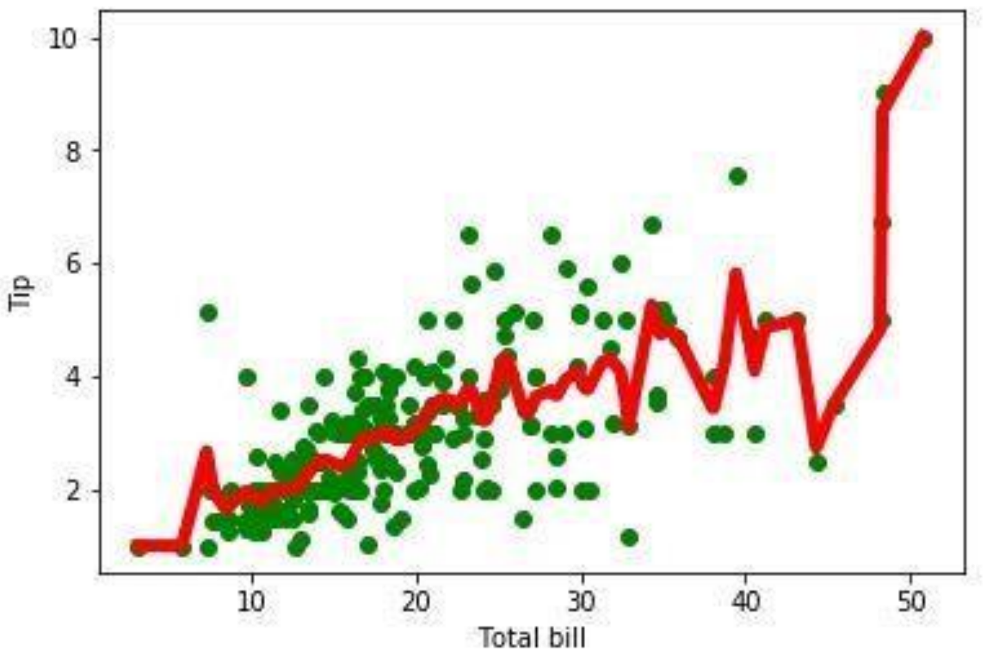
Prediction = x0*β

## SOURCE CODE IN PYTHON

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

def kernel(point,xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye((m))) # eye - identity matrix
    for j in range(m):
        diff = point - X[j]
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights

def localWeight(point,xmat,ymat,k):
    wei = kernel(point,xmat,k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W

def localWeightRegression(xmat,ymat,k):
    m,n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
    return ypred

def graphPlot(X,ypred):
    sortindex = X[:,1].argsort(0) #argsort - index of the smallest
    xsort = X[sortindex][:,0]
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.scatter(bill,tip, color='green')
    ax.plot(xsort[:,1],ypred[sortindex], color = 'red', linewidth=5)
    plt.xlabel('Total bill')
    plt.ylabel('Tip')
    plt.show();
# load data points
data = pd.read_csv('tips.csv')
bill = np.array(data.total_bill) # We use only Bill amount and Tips data tip =
np.array(data.tip)
tip = np.array(data.tip)

mbill = np.mat(bill) # .mat will convert nd array is converted in 2D array mtip =
np.mat(tip)
```

```
mtip = np.mat(tip)
m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
#print(X)
X = np.hstack((one.T,mbill.T)) # 244 rows, 2 cols
ypred = localWeightRegression(X,mtip,0.5) # increase k to get smooth curves
graphPlot(X,ypred)
```

**OUTPUT:**


Tip
Total bill