# CITY ENGINEERING COLLEGE

(Doddakallsandra, off. Kanakapura Road, Bangalore – 560061)

**NAAC ACCREDITED**

Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE, New Delhi

# LAB MANUAL

## ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

## (Subject Code: 18CSL76)

**For**

## VII SEMESTER

NAME: ______________________________

USN: ______________________________

SEMESTER & SECTION: ____________________

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**ACADEMIC YEAR: 2021-2022**

# CITY ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

## VISION

To contribute to Global Development by producing Knowledgeable and Quality professionals who are Innovative and Successful in advanced field of Computer Science & Engineering to adapt the changing Employment demands and social needs.

## MISSION

**M1**: To provide Quality Education for students, to build Confidence by developing their Technical Skills to make them Competitive Computer Science Engineers.
**M2**: To facilitate Innovation & Research for students and faculty and to provide Internship opportunities
**M3**: To Collaborate with educational institutions and industries for Excellence in Teaching and Research.

**ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY**
(Effective from the academic year 2018 -2019)
SEMESTER – VII

| | | | |
|---|---|---|---|
| Subject Code | 18CSL76 | IA Marks | 40 |
| Number of Lecture Hours/Week | 01I + 02P | Exam Marks | 60 |
| Total Number of Lecture Hours | 36 | Exam Hours | 03 |

**CREDITS – 02**

Course Learning Objectives: This course (18CSL76) will enable students to:

Implement and evaluate AI and ML algorithms in and Python programming language.

Descriptions (if any):

Installation procedure of the required software must be demonstrated, carried out in groups and documented in the journal.

**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 **Back propagation 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 dataset. 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.

| |
|---|
| **Laboratory outcomes:** The students should be able to: |
| • Implement and demonstrate AI and ML algorithms.<br>• Evaluate different algorithms. |
| **Conduction of Practical Examination:** |
| • All laboratory experiments are to be included for practical examination.<br>• Students are allowed to pick one experiment from the lot.<br>• Strictly follow the instructions as printed on the cover page of answer script<br>• Marks distribution: Procedure + Conduction + Viva:**15 + 70 +15 (100)**<br>**Change of experiment is allowed only once and marks allotted to the procedure part to be made zero.** |

1. **LAB PROGRAM:** 1

2. **TITLE: A* SEARCH ALGORITHM**
3. **AIM:**
   Implement A* Search algorithm

4. A* Search algorithm

   1. **Initialize**: set OPEN=[s], CLOSED=[], g(s)=0, f(s)=h(s)
   2. **Fail:** If OPEN=[], then terminate and fail
   3. **Select:** Select a state with minimum cost ,n, from OPEN and save in CLOSED
   4. **Terminate**: If n∈G then terminate with success and return f(s)
   5. **Expand:** For each successors , m of n
      For each successor, m, insert m in OPEN only if
      if m∉ [OPEN∪CLOSED]
      set g(m)= g[n]+C[n,m]
      Set f(m)=g(m)+h(n)
      if m∈[OPEN∪CLOSED]
      set g(m)= min{ g[m], g(n)+C[n,m]}
      Set f(m)=g(m)+h(m)
      If f[m] has decreased and m ∈ CLOSED move m to OPEN
   6. **Loop**: Goto step 2

## 5. Implementation/ Program 1:

```
def aStarAlgo(start_node, stop_node):

        open_set = set(start_node)
        closed_set = set()
        g = {}                          #store distance from starting node
        parents = {}                    #  parents contains an adjacency map of all nodes

                                        #ditance of starting node from itself is zero
        g[start_node] = 0
                                         #start_node is root node i.e it has no parent nodes
                                        #so start_node is set to its own parent node
        parents[start_node] = start_node

        while len(open_set) > 0:
            n = None
                                                #node with lowest f() is found
            for v in open_set:
                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                    n = v

            if n == stop_node or Graph_nodes[n] == None:
                pass
            else:
```

```
        for (m, weight) in get_neighbors(n):
                                                                    #nodes 'm' not in first and last set are added to first
                                                                    #n is set its parent
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight


                                                            #for each node m,compare its distance from start i.e g(m) to the
                                                            #from start through n node
            else:
                if g[m] > g[n] + weight:
                                                            #update g(m)
                    g[m] = g[n] + weight
                                                            #change parent of m to n
                    parents[m] = n

                                                            #if m in closed set,remove and add to open
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

    if n == None:
        print('Path does not exist!')
        return None

                                                                    # if the current node is the stop_node
                                                            # then we begin reconstructin the path from it to the start_node
    if n == stop_node:
        path = []
         while parents[n] != n:
            path.append(n)
            n = parents[n]
         path.append(start_node)
         path.reverse()

        print('Path found: {}'.format(path))
        return path

                                                            # remove n from the open_list, and add it to closed_list
                                                            # because all of his neighbors were inspected
    open_set.remove(n)
    closed_set.add(n)

print('Path does not exist!')
return None
                                                                    #define fuction to return neighbor and its distance
                                                                    #from the passed node
```

```
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
                                        #for simplicity we ll consider heuristic distances given
                                        #and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 10,
            'B': 8,
            'C': 5,
            'D': 7,
            'E': 3,
            'F': 6,
            'G': 5,
            'H': 3,
            'I': 1,
            'J': 0
        }

        return H_dist[n]

                                        #Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 1),('H', 7)] ,
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)],

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

**6. Result/Output:**

```
Path found: ['A', 'F', 'G', 'I', 'J']

 ['A', 'F', 'G', 'I', 'J']
```

1. **LAB PROGRAM:** 2

2. **TITLE: AO* SEARCH ALGORITHM**
3. **AIM:**
   Implement AO* Search algorithm

4. AO* Search algorithm

1. Let *GRAPH* consist only of the node representing the initial state. (Call this node *INIT.)* Compute *h'(INIT)*
2. Until *INIT* is labeled *SOLVED* or until *INIT's h'* value becomes greater than *FUTILITY,* repeat the following procedure:
   (a) Trace the labeled arcs from *INIT* and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node *NODE.*
   (b) Generate the successors of *NODE.* If there are none, then assign *FUTILITY* as the *h'* value of *NODE.* This is equivalent to saying that *NODE* is not solvable. If there are successors, then for each one (called *SUCCESSOR)* that is not also an ancestor of *NODE* do the following:
      (i) Add *SUCCESSOR* to *GRAPH.*
      (ii) If *SUCCESSOR* is a terminal node, label it *SOLVED* and assign it an *h'* value of 0.
      (iii) If *SUCCESSOR* is not a terminal node, compute its *h'* value.
   (c) Propagate the newly discovered information up the graph by doing the following: Let *S* be a set of nodes that have been labeled *SOLVED* or whose *h'* values have been changed and so need to have values propagated back to their parents. Initialize 5 to *NODE.* Until *S* is empty, repeat the, following procedure:
      (i) If possible, select from *S* a node none of whose descendants in *GRAPH* occurs in *S.* If there is no such node, select any node from *S.* Call this node *CURRENT,* and remove it from *S.*
      (ii) Compute the cost of each of the arcs emerging from *CURRENT.* The cost of each arc is equal to the sum of the *h'* values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as *CURRENT'S* new *h'* value the minimum of the costs just computed for the arcs emerging from it.
      (iii) Mark the best path out of *CURRENT* by marking the arc that had the minimum cost as computed in the previous step.
      (iv) Mark *CURRENT SOLVED* if all of the nodes connected to it through the new labeled arc have been labeled *SOLVED.*
      (v) If *CURRENT* has been labeled *SOLVED* or if the cost of *CURRENT* was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of *CURRENT* to *S.*

## 5. Implementation/ Program 2:

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):   #instantiate graph object with
                                                                 graph topology, heuristic values, start node
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}
```

```
def applyAOStar(self):                                    #  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 START
      NODE:",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
```

```python
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 part of
                                                  solution

        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 to 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, '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 output of the AO* algorithm search
```

## 6. Result/Output:

```
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 START NODE: 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 START NODE: A
------------------------------------------------------------
{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}
------------------------------------------------------------
```

1. **LAB PROGRAM:** 3

2. **TITLE:** Candidate-Elimination algorithm
3. **AIM:**
   - 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. **Candidate-Elimination algorithm:**

Initialize G to the set of maximally general hypotheses in H
1. Initialize S to the set of maximally specific hypotheses in H
2. For each training example d, do
    2.1. If d is a positive example
        Remove from G any hypothesis 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 is consistent with d,
              and some member of G is more general than h
            Remove from S, hypothesis that is more general than another hypothesis in S
    2.2. If d is a negative example
        Remove from S any hypothesis 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 is consistent with d, and

        some member of S is more specific than h
        Remove from G any hypothesis that is less general than another hypothesis in G

## 5. Implementation/Program3:

```
import csv
a=[]
with open("enjoysport.csv","r") as csvfile:
    fdata=csv.reader(csvfile)
    for row in fdata:
        a.append(row)
        print(row)
num_att=len(a[0])-1
S=['0']*num_att
G=['?']*num_att
print(S)
print(G)
temp=[]
for i in range(0,num_att):
    S[i]=a[0][i]
print("...........................................")
for i in range(0,len(a)):
```

```
    if a[i][num_att]=="Yes":
        for j in range(0,num_att):
            if S[j]!=a[i][j]:
                S[j]='?'
        for j in range(0,num_att):
            for k in range(0,len(temp)):
                if temp[k][j]!=S[j] and temp[k][j]!='?':
                    del temp[k]
    if a[i][num_att]=='No':
        for j in range(0,num_att):
            if a[i][j]!=S[j] and S[j]!='?':
                G[j]=S[j]
                temp.append(G)
                G=['?']*num_att
    print(S)
    if len(temp)==0:
        print(G)
    else:
        print(temp)
    print("...............................................................")
```

## 6. Result/Output:

```
['sunny', 'warm', 'normal', 'strong', 'warm', 'same', 'Yes']
['sunny', 'warm', 'high', 'strong', 'warm', 'same', 'Yes']
['rainy', 'cold', 'high', 'strong', 'warm', 'change', 'No']
['sunny', 'warm', 'high', 'strong', 'cool', 'change', 'Yes']
['0', '0', '0', '0', '0', '0']
['?', '?', '?', '?', '?', '?']
...............................................
['sunny', 'warm', 'normal', 'strong', 'warm', 'same']
['?', '?', '?', '?', '?', '?']
.....................................................................
['sunny', 'warm', '?', 'strong', 'warm', 'same']
['?', '?', '?', '?', '?', '?']
.....................................................................
['sunny', 'warm', '?', 'strong', 'warm', 'same']
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]
.....................................................................
['sunny', 'warm', '?', 'strong', '?', '?']
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]
.....................................................................
```

## Training Data Set : enjoysport.csv

| | | | | | | |
|---|---|---|---|---|---|---|
| sunny | warm | normal | strong | warm | same | Yes |
| sunny | warm | high | strong | warm | same | Yes |
| rainy | cold | high | strong | warm | change | No |
| sunny | warm | high | strong | cool | change | Yes |

# 1. Lab Program : 4

2. **TITLE: ID3 ALGORITHM**

3. **AIM:**

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

## 4. ID3 algorithm:

**Algorithm:** ID3(Examples, TargetAttribute, Attributes) Input:

Examples are the training examples.

Targetattribute is the attribute whose value is to be predicted by the tree.

Attributes is a list of other attributes that may be tested by the learned decision tree.

Output: Returns a decision tree that correctly classiJies the given Examples Method:

1. Create a Root node for the tree
2. If all Examples are positive, Return the single-node tree Root, with label = +
3. If all Examples are negative, Return the single-node tree Root, with label = -
4. If Attributes is empty,

Return the single-node tree Root, with label = most common value of TargetAttribute in Examples

Else

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

## 5. Implementation/Program:

```
import pandas as pd
import math

df = pd.read_csv('/Users/Documents/Python Scripts/PlayTennis.csv')
print("\n Input Data Set is:\n", df)

t = df.keys()[-1]
print('Target Attribute is: ', t)
# Get the attribute names from input dataset
attribute_names = list(df.keys())
#Remove the target attribute from the attribute names list
attribute_names.remove(t)
print('Predicting Attributes: ', attribute_names)
#Function to calculate the entropy of collection S
def entropy(probs):
```

```
    return sum( [-prob*math.log(prob, 2) for prob in probs])
#Function to calulate the entropy of the given Data Sets/List with
#respect to target attributes
def entropy_of_list(ls,value):
    from collections import Counter
    cnt = Counter(x for x in ls)# Counter calculates the propotion of class
    print('Target attribute class count(Yes/No)=',dict(cnt))
    total_instances = len(ls)
    print("Total no of instances/records associated with {0} is: {1}".format(value,total_instances ))
    probs = [x / total_instances for x in cnt.values()]  # x means no of YES/NO
    print("Probability of Class {0} is: {1:.4f}".format(min(cnt),min(probs)))
    print("Probability of Class {0} is: {1:.4f}".format(max(cnt),max(probs)))
    return entropy(probs) # Call Entropy

def information_gain(df, split_attribute, target_attribute,battr):
    print("\n\n-----Information Gain Calculation of ",split_attribute, " --------")
    df_split = df.groupby(split_attribute) # group the data based on attribute values
    glist=[]
    for gname,group in df_split:
        print('Grouped Attribute Values \n',group)
        glist.append(gname)

    glist.reverse()
    nobs = len(df.index) * 1.0
    df_agg1=df_split.agg({target_attribute:lambda x:entropy_of_list(x, glist.pop())})
    df_agg2=df_split.agg({target_attribute :lambda x:len(x)/nobs})

    df_agg1.columns=['Entropy']
    df_agg2.columns=['Proportion']

    # Calculate Information Gain:
    new_entropy = sum( df_agg1['Entropy'] * df_agg2['Proportion'])
    if battr !='S':
        old_entropy = entropy_of_list(df[target_attribute],'S-'+df.iloc[0][df.columns.get_loc(battr)])
    else:
        old_entropy = entropy_of_list(df[target_attribute],battr)
    return old_entropy - new_entropy



def id3(df, target_attribute, attribute_names, default_class=None,default_attr='S'):

    from collections import Counter
    cnt = Counter(x for x in df[target_attribute])# class of YES /NO

    ## First check: Is this split of the dataset homogeneous?
    if len(cnt) == 1:
        return next(iter(cnt))  # next input data set, or raises StopIteration when EOF is hit.

    ## Second check: Is this split of the dataset empty? if yes, return a default value
    elif df.empty or (not attribute_names):
        return default_class  # Return None for Empty Data Set

    ## Otherwise: This dataset is ready to be devied up!
```

```
    else:
        # Get Default Value for next recursive call of this function:
        default_class = max(cnt.keys()) #No of YES and NO Class
        # Compute the Information Gain of the attributes:
        gainz=[]
        for attr in attribute_names:
            ig= information_gain(df, attr, target_attribute,default_attr)
            gainz.append(ig)
            print('Information gain of ',attr,' is : ',ig)

        index_of_max = gainz.index(max(gainz))
        best_attr = attribute_names[index_of_max
        print("\nAttribute with the maximum gain is: ", best_attr)
        # Create an empty tree, to be populated in a moment
        tree = {best_attr:{}} # Initiate the tree with best attribute as a node
        remaining_attribute_names =[i for i in attribute_names if i != best_attr]

        # Split dataset-On each split, recursively call this algorithm.Populate the empty tree with
subtrees, which
        # are the result of the recursive call
        for attr_val, data_subset in df.groupby(best_attr):
            subtree = id3(data_subset,target_attribute,
remaining_attribute_names,default_class,best_attr)
            tree[best_attr][attr_val] = subtree
        return tree

    from pprint import pprint
tree = id3(df,t,attribute_names)
print("\nThe Resultant Decision Tree is:")
print(tree)

def classify(instance, tree,default=None): # Instance of Play Tennis with Predicted
    attribute = next(iter(tree)) # Outlook/Humidity/Wind
    if instance[attribute] in tree[attribute].keys(): # Value of the attributs in  set of Tree keys
        result = tree[attribute][instance[attribute]]
        if isinstance(result, dict): # this is a tree, delve deeper
            return classify(instance, result)
        else:
            return result # this is a label
    else:
        return default

df_new=pd.read_csv('/Users/Documents/Python Scripts/PlayTennisTest.csv')
df_new['predicted'] = df_new.apply(classify, axis=1, args=(tree,'?'))
print(df_new)
```

## 6. Result/Output:

```
 Input Data Set is:
      Outlook Temperature Humidity    Wind PlayTennis
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
Target Attribute is:  PlayTennis
Predicting Attributes:  ['Outlook', 'Temperature', 'Humidity', 'Wind']

The Resultant Decision Tree is:
{'Outlook': {'Overcast': 'Yes', 'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}}, 'Sunny': {'Humidity': {'High': 'No', 'Norma
l': 'Yes'}}}}

 Testing Samples are :
  Outlook Temperature Humidity  Wind PlayTennis predicted
0   Sunny         Hot     High  Weak          ?        No
1    Rain        Mild     High  Weak          ?       Yes
```

## Training Data Set : PlayTennis.csv

| Outlook | Temperatu | Humidity | Wind | PlayTennis |
|---|---|---|---|---|
| Sunny | Hot | High | Weak | No |
| Sunny | Hot | High | Strong | No |
| Overcast | Hot | High | Weak | Yes |
| Rain | Mild | High | Weak | Yes |
| Rain | Cool | Normal | Weak | Yes |
| Rain | Cool | Normal | Strong | No |
| Overcast | Cool | Normal | Strong | Yes |
| Sunny | Mild | High | Weak | No |
| Sunny | Cool | Normal | Weak | Yes |
| Rain | Mild | Normal | Weak | Yes |
| Sunny | Mild | Normal | Strong | Yes |
| Overcast | Mild | High | Strong | Yes |
| Overcast | Hot | Normal | Weak | Yes |
| Rain | Mild | High | Strong | No |

## Testing Data Set : PlayTennisTest.csv

| Outlook | Temperatu | Humidity | Wind | PlayTennis |
|---|---|---|---|---|
| Sunny | Hot | High | Weak | ? |
| Rain | Mild | High | Weak | ? |

1. **LAB PROGRAM: 5**

2. **TITLE: BACKPROPAGATION ALGORITHM**
3. **AIM:**
Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.

4. **Backpropagation Algorithm:**

**Algorithm:**
BACKPROPOGATION (training_examples,η,$n_{in}$,$n_{out}$,$n_{hidden}$)
Each training example is a pair of the form (x,t ), where x is the vector of network input values, and t is the vector of target network output values.
η is the learning rate (e.g., .O5). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of units in the hidden layer, and $n_{out}$ the number of output units.
The input from unit i into unit j is denoted xji, and the weight from unit i to unit j is denoted wji.

1. Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
2. Initialize all network weights to small random numbers
3. Until the termination condition is met, Do
    - For each ***(x,t*** ) in *trainingaxamples,* **Do**
        - *Propagate the input forward through the network:*
            1. Input the instance x to the network and compute the output *O,* of every unit ***u*** in the network.

            *Propagate the errors backward through the network:*
            2. For each network output unit k, calculate its error term *δk*

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

            3. For each hidden unit *h,* calculate its error term δ*h*

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

            4. Update each network weight ***wji***

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

## 5. Implementation/ Program:

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]))
y = np.array(([92], [86], [89]))
y = y/100

def sigmoid(x):
    return 1/(1 + np.exp(-x))

def derivatives_sigmoid(x):
    return x * (1 - x)

epoch=10000
```

```
lr=0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
output_neurons = 1

wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bias_hidden=np.random.uniform(size=(1,hiddenlayer_neurons))
weight_hidden=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bias_output=np.random.uniform(size=(1,output_neurons))

for i in range(epoch):
    hinp1=np.dot(X,wh)
    hinp= hinp1 + bias_hidden
    hlayer_activation = sigmoid(hinp)

    outinp1=np.dot(hlayer_activation,weight_hidden)
    outinp= outinp1+ bias_output
    output = sigmoid(outinp)

    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO * outgrad
    EH = d_output.dot(weight_hidden.T)
    hiddengrad = derivatives_sigmoid(hlayer_activation)
    d_hiddenlayer = EH * hiddengrad

    weight_hidden += hlayer_activation.T.dot(d_output) *lr
    bias_hidden += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr

    wh += X.T.dot(d_hiddenlayer) *lr
    bias_output += np.sum(d_output, axis=0,keepdims=True) *lr

print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

## 6. **Result/Output:**

```
Input:
[[2 9]
 [1 5]
 [3 6]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.89312029]
 [0.87792011]
 [0.89768518]]
```

1. **LAB PROGRAM: 6**

2. TITLE: **NAÏVE BAYESIAN CLASSIFIER**
3. AIM:
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.

4. **Algorithm:**

**Algorithm:**
**NaiveBaiseClassifier(training_examples, New_Instance)**

Each instance ***x*** is described by a conjunction of attribute values($a_i$) and the target V can take j finite set of values.

a. For each value j in target estimate the P(Vj)
b. For each attribute in the training example estimate Estimate the P(ai|Vj)
c. Classify each instance as per the rule in equation

$$v_{NB} = \operatorname*{argmax}_{v_j \in V} P(v_j) \prod_i P(a_i|v_j)$$

Where $V_{NB}$ denotes the target value output by the naive Bayes classifier

d. Output $V_{NB}$

## 5. Implementation/Program :

```
import numpy as np
import math
import csv
import pdb
def read_data(filename):

    with open(filename,'r') as csvfile:
        datareader = csv.reader(csvfile)
        metadata = next(datareader)
        traindata=[]
        for row in datareader:
            traindata.append(row)

    return (metadata, traindata)

def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    testset = list(dataset)
    i=0
    while len(trainSet) < trainSize:
        trainSet.append(testset.pop(i))
    return [trainSet, testset]
```

```
def classify(data,test):

    total_size = data.shape[0]
    print("\n")
    print("training data size=",total_size)
    print("test data size=",test.shape[0])

    countYes = 0
    countNo = 0
    probYes = 0
    probNo = 0
    print("\n")
    print("target    count    probability")

    for x in range(data.shape[0]):
        if data[x,data.shape[1]-1] == 'yes':
            countYes +=1
        if data[x,data.shape[1]-1] == 'no':
            countNo +=1

    probYes=countYes/total_size
    probNo= countNo / total_size

    print('Yes',"\t",countYes,"\t",probYes)
    print('No',"\t",countNo,"\t",probNo)


    prob0 =np.zeros((test.shape[1]-1))
    prob1 =np.zeros((test.shape[1]-1))
    accuracy=0
    print("\n")
    print("instance prediction  target")

    for t in range(test.shape[0]):
        for k in range (test.shape[1]-1):
            count1=count0=0
            for j in range (data.shape[0]):
                #how many times appeared with no
                if test[t,k] == data[j,k] and data[j,data.shape[1]-1]=='no':
                    count0+=1
                #how many times appeared with yes
                if test[t,k]==data[j,k] and data[j,data.shape[1]-1]=='yes':
                    count1+=1
            prob0[k]=count0/countNo
            prob1[k]=count1/countYes

        probno=probNo
        probyes=probYes
        for i in range(test.shape[1]-1):
            probno=probno*prob0[i]
```

```
            probyes=probyes*prob1[i]
        if probno>probyes:
            predict='no'
        else:
            predict='yes'

        print(t+1,"\t",predict,"\t    ",test[t,test.shape[1]-1])
        if predict == test[t,test.shape[1]-1]:
            accuracy+=1
    final_accuracy=(accuracy/test.shape[0])*100
    print("accuracy",final_accuracy,"%")
    return

metadata,traindata= read_data("/Users/Chachu/Documents/Python Scripts/tennis.csv")
splitRatio=0.6
trainingset, testset=splitDataset(traindata, splitRatio)
training=np.array(trainingset)
print("\n The Training data set are:")
for x in trainingset:
    print(x)

testing=np.array(testset)
print("\n The Test data set are:")
for x in testing:
    print(x)
classify(training,testing)
```

## 6. Result /Output:

```
 The Training data set are:
['sunny', 'hot', 'high', 'Weak', 'no']
['sunny', 'hot', 'high', 'Strong', 'no']
['overcast', 'hot', 'high', 'Weak', 'yes']
['rainy', 'mild', 'high', 'Weak', 'yes']
['rainy', 'cool', 'normal', 'Weak', 'yes']
['rainy', 'cool', 'normal', 'Strong', 'no']
['overcast', 'cool', 'normal', 'Strong', 'yes']
['sunny', 'mild', 'high', 'Weak', 'no']

 The Test data set are:
['sunny' 'cool' 'normal' 'Weak' 'yes']
['rainy' 'mild' 'normal' 'Weak' 'yes']
['sunny' 'mild' 'normal' 'Strong' 'yes']
['overcast' 'mild' 'high' 'Strong' 'yes']
['overcast' 'hot' 'normal' 'Weak' 'yes']
['rainy' 'mild' 'high' 'Strong' 'no']
```

```
training data size= 8
test data size= 6


target    count    probability
Yes       4       0.5
No        4       0.5


instance prediction  target
1         no          yes
2         yes         yes
3         no          yes
4         yes         yes
5         yes         yes
6         no          no
accuracy 66.66666666666666 %
```

## Training Data Set : tennis.csv

| outlook | temp | humidity | windy | answer |
|---|---|---|---|---|
| sunny | hot | high | Weak | no |
| sunny | hot | high | Strong | no |
| overcast | hot | high | Weak | yes |
| rainy | mild | high | Weak | yes |
| rainy | cool | normal | Weak | yes |
| rainy | cool | normal | Strong | no |
| overcast | cool | normal | Strong | yes |
| sunny | mild | high | Weak | no |
| sunny | cool | normal | Weak | yes |
| rainy | mild | normal | Weak | yes |
| sunny | mild | normal | Strong | yes |
| overcast | mild | high | Strong | yes |
| overcast | hot | normal | Weak | yes |
| rainy | mild | high | Strong | no |

1. **LAB PROGRAM: 7**

2. TITLE: **CLUSTERING BASED ON EM ALGORITHM AND K-MEANS**

3. AIM:

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.

4. THEORY:

**Expectation Maximization algorithm**

- The basic approach and logic of this clustering method is as follows.
- Suppose we measure a single continuous variable in a large sample of observations. Further, suppose that the sample consists of two clusters of observations with different means (and perhaps different standard deviations); within each sample, the distribution of values for the continuous variable follows the normal distribution.
- The goal of EM clustering is to estimate the means and standard deviations for each cluster so as to maximize the likelihood of the observed data (distribution).
- Put another way, the EM algorithm attempts to approximate the observed distributions of values based on mixtures of different distributions in different clusters. The results of EM clustering are different from those computed by k-means clustering.
- The latter will assign observations to clusters to maximize the distances between clusters. The EM algorithm does not compute actual assignments of observations to clusters, but classification probabilities.
- In other words, each observation belongs to each cluster with a certain probability. Of course, as a final result we can usually review an actual assignment of observations to clusters, based on the (largest) classification probability.

**K means Clustering**

- The algorithm will categorize the items into k groups of similarity. To calculate that similarity, we will use the euclidean distance as measurement.
- The algorithm works as follows:
    1. First we initialize k points, called means, randomly.
    2. We categorize each item to its closest mean and we update the mean's coordinates, which are the averages of the items categorized in that mean so far.
    3. We repeat the process for a given number of iterations and at the end, we have our clusters.
- The "points" mentioned above are called means, because they hold the mean values of the items categorized in it. To initialize these means, we have a lot of options. An intuitive method is to initialize the means at random items in the data set. Another method is to initialize the means at random values between the boundaries of the data set (if for a feature x the items have values in [0,3], we will initialize the means with values for x at [0,3]).
- **Pseudocode:**
    1. Initialize k means with random values
    2. For a given number of iterations: Iterate

```
        through items:
            Find the mean closest to the item Assign
            item to mean
              Update mean
```

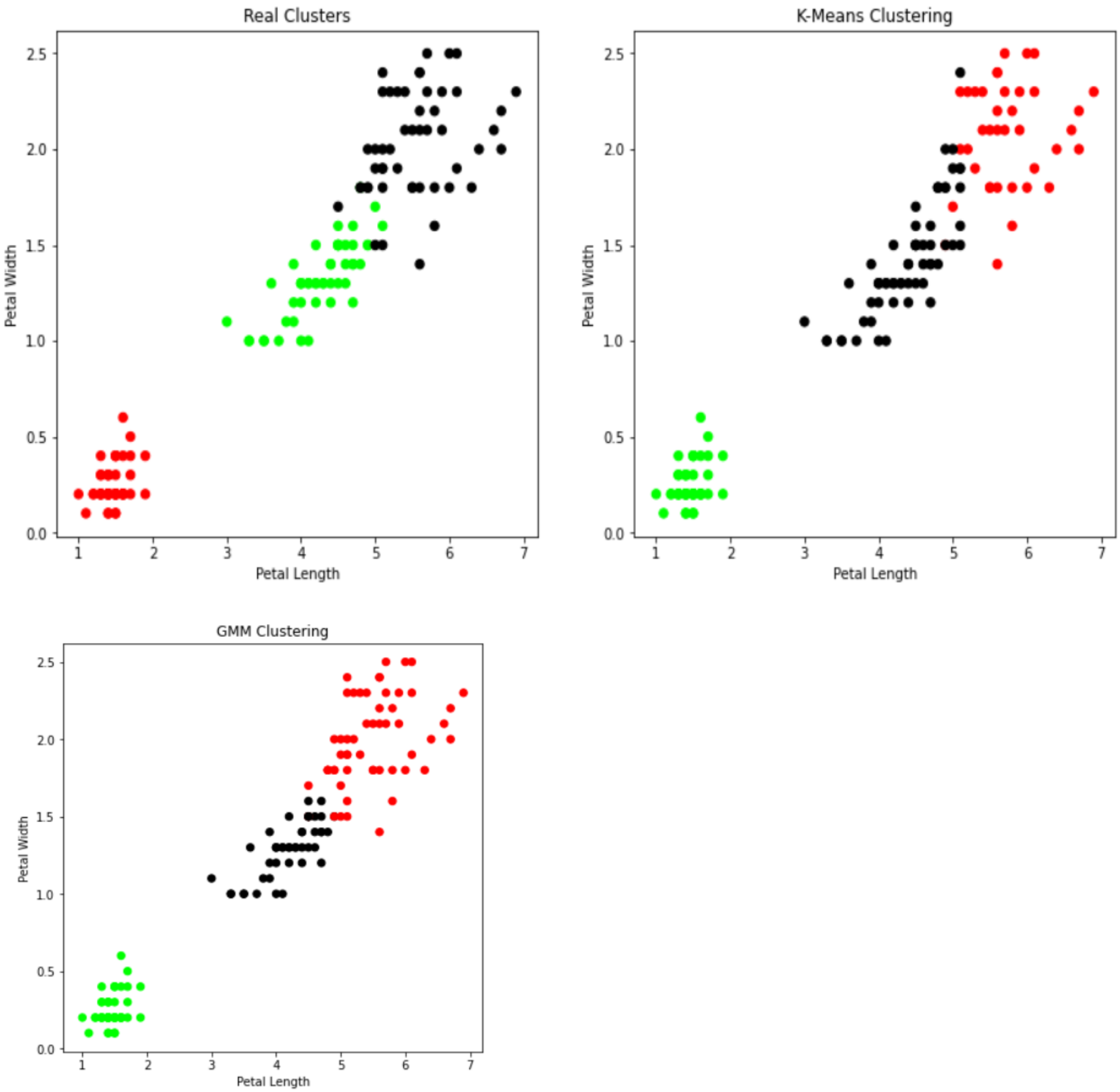
## 5. Implementation/Program :

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import pandas as pd
import numpy as np
                              # import some data to play with
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']

# Build the K Means Model
model = KMeans(n_clusters=3)
model.fit(X)      # model.labels_ : Gives cluster no for which samples belongs to

# # Visualise the clustering results
plt.figure(figsize=(14,14))
colormap = np.array(['red', 'lime', 'black'])
# Plot the Original Classifications using Petal features
plt.subplot(2, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
# Plot the Models Classifications
plt.subplot(2, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

# General EM for GMM
from sklearn import preprocessing
# transform your data such that its distribution will have a # mean value 0 and standard
deviation of 1.
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y = gmm.predict(xs)
plt.subplot(2, 2, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[gmm_y], s=40)
```

```
plt.title('GMM Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
print('Observation: The GMM using EM algorithm based clustering matched the true labels more closely than the Kmeans.')
```

## 6. Result/Output:

```
Observation: The GMM using EM algorithm based clustering matched the true labels more closely than the Kmeans.
```



1. **LAB PROGRAM: 8**

2. TITLE: **K-NEAREST NEIGHBOUR**

3. AIM:

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.

**4.** THEORY:

- K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.
- It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data.

- **Algorithm**

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

  Method:

  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.

     Return the majority label among S.

## 5. Implementation/ Program:

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets

# Load dataset
iris=datasets.load_iris()
print("Iris Data set loaded...")

# Split the data into train and test samples
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 trainng data and its label",x_train.shape,y_train.shape)
print("Size of trainng data and its label",x_test.shape, y_test.shape)

# Prints Label no. and their names
for i in range(len(iris.target_names)):
    print("Label", i , "-",str(iris.target_names[i]))
    # Create object of KNN classifier
classifier = KNeighborsClassifier(n_neighbors=1)
```

```
# Perform Training
classifier.fit(x_train, y_train) # Perform testing
y_pred=classifier.predict(x_test)

# Display the results
print("Results of Classification using K-nn with K=1 ")
for r in range(0,len(x_test)):
    print(" Sample:", str(x_test[r]), " Actual-label:", str(y_test[r]), " Predicted-label:", str(y_pred[r]))
print("Classification Accuracy :" , classifier.score(x_test,y_test));

from sklearn.metrics import classification_report, confusion_matrix
print('Confusion Matrix')
print(confusion_matrix(y_test,y_pred))
print('Accuracy Metrics')
print(classification_report(y_test,y_pred))
```

## 6. Result/ Output:

```
Iris Data set loaded...
Dataset is split into training and testing...
Size of trainng data and its label (135, 4) (135,)
Size of trainng data and its label (15, 4) (15,)
Label 0 - setosa
Label 1 - versicolor
Label 2 - virginica
Results of Classification using K-nn with K=1
 Sample: [5.6 2.8 4.9 2. ]  Actual-label: 2  Predicted-label: 2
 Sample: [5.6 3.  4.5 1.5]  Actual-label: 1  Predicted-label: 1
 Sample: [6.  2.7 5.1 1.6]  Actual-label: 1  Predicted-label: 2
 Sample: [6.5 3.2 5.1 2. ]  Actual-label: 2  Predicted-label: 2
 Sample: [5.2 3.4 1.4 0.2]  Actual-label: 0  Predicted-label: 0
 Sample: [5.  3.5 1.6 0.6]  Actual-label: 0  Predicted-label: 0
 Sample: [5.2 3.5 1.5 0.2]  Actual-label: 0  Predicted-label: 0
 Sample: [5.7 2.8 4.5 1.3]  Actual-label: 1  Predicted-label: 1
 Sample: [5.8 4.  1.2 0.2]  Actual-label: 0  Predicted-label: 0
 Sample: [6.4 2.8 5.6 2.2]  Actual-label: 2  Predicted-label: 2
 Sample: [6.4 2.9 4.3 1.3]  Actual-label: 1  Predicted-label: 1
 Sample: [6.2 2.2 4.5 1.5]  Actual-label: 1  Predicted-label: 1
 Sample: [5.5 2.4 3.8 1.1]  Actual-label: 1  Predicted-label: 1
 Sample: [4.8 3.4 1.6 0.2]  Actual-label: 0  Predicted-label: 0
 Sample: [6.3 2.8 5.1 1.5]  Actual-label: 2  Predicted-label: 1
Classification Accuracy : 0.8666666666666667

Confusion Matrix
[[5 0 0]
 [0 5 1]
 [0 1 3]]
Accuracy Metrics
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       0.83      0.83      0.83         6
           2       0.75      0.75      0.75         4

    accuracy                           0.87        15
   macro avg       0.86      0.86      0.86        15
weighted avg       0.87      0.87      0.87        15
```

1. **LAB PROGRAM: 9**

2. TITLE: **LOCALLY WEIGHTED REGRESSION ALGORITHM**

3. AIM:

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

**4.** THEORY:

- Given a dataset X, y, we attempt to find a linear model h(x) that minimizes residual sum of squared errors. The solution is given by Normal equations.
- Linear model can only fit a straight line, however, it can be empowered by polynomial features to get more powerful models. Still, we have to decide and fix the number and types of features ahead.
- Alternate approach is given by locally weighted regression.
- Given a dataset X, y, we attempt to find a model h(x) that minimizes residual sum of weighted squared errors.
- The weights are given by a kernel function which can be chosen arbitrarily and in my case I chose a Gaussian kernel.
- The solution is very similar to Normal equations, we only need to insert diagonal weight matrix W.

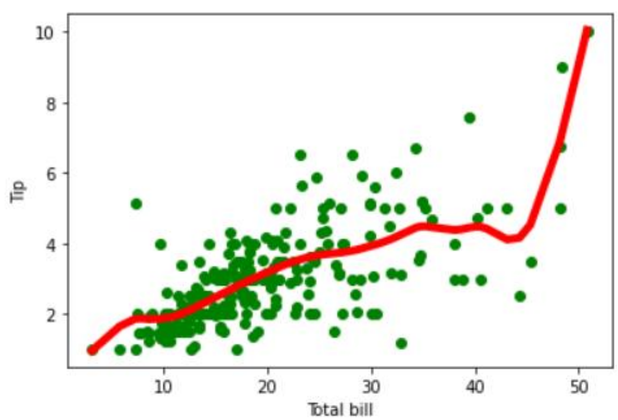
## 5. Implementation/ Program:

```
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]
```

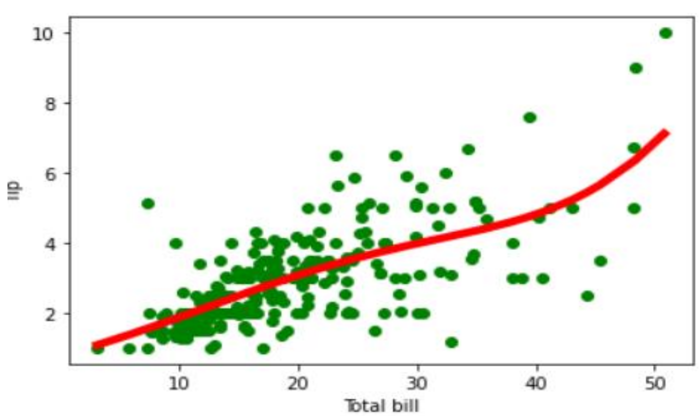
```
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.scatter(bill,tip, color='green')
    ax.plot(xsort[:,1],ypred[sortindex], color = 'red', linewidth=5)
    plt.xlabel('Total bill')
    plt.ylabel('Tip')
    plt.show();

# load data points
data = pd.read_csv('/Users/Chachu/Documents/Python Scripts/data10_tips.csv')
bill = np.array(data.total_bill) # We use only Bill amount and Tips data
tip = np.array(data.tip)

mbill = np.mat(bill) # .mat will convert nd array is converted in 2D array
mtip = np.mat(tip)
m= np.shape(mbill)[1]
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T)) # 244 rows, 2 cols

ypred = localWeightRegression(X,mtip,2) # increase k to get smooth curves
graphPlot(X,ypred)
```

## 6. Result/ Output:

Regression with parameter k = 2 ( ypred = localWeightRegression(X,mtip,**2**))




Regression with parameter k = 8 ( ypred = localWeightRegression(X,mtip,**8**))



# Training Data Set : data10_tips.csv (Sample)

| total_bill | tip | sex | smoker | day | time | size |
|---|---|---|---|---|---|---|
| 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| 21.01 | 3.5 | Male | No | Sun | Dinner | 3 |
| 23.68 | 3.31 | Male | No | Sun | Dinner | 2 |
| 24.59 | 3.61 | Female | No | Sun | Dinner | 4 |
| 25.29 | 4.71 | Male | No | Sun | Dinner | 4 |
| 8.77 | 2 | Male | No | Sun | Dinner | 2 |
| 26.88 | 3.12 | Male | No | Sun | Dinner | 4 |
| 15.04 | 1.96 | Male | No | Sun | Dinner | 2 |
| 14.78 | 3.23 | Male | No | Sun | Dinner | 2 |
| 10.27 | 1.71 | Male | No | Sun | Dinner | 2 |
| 35.26 | 5 | Female | No | Sun | Dinner | 4 |
| 15.42 | 1.57 | Male | No | Sun | Dinner | 2 |
| 18.43 | 3 | Male | No | Sun | Dinner | 4 |
| 14.83 | 3.02 | Female | No | Sun | Dinner | 2 |
| 21.58 | 3.92 | Male | No | Sun | Dinner | 2 |
| 10.33 | 1.67 | Female | No | Sun | Dinner | 3 |
| 16.29 | 3.71 | Male | No | Sun | Dinner | 3 |
| 16.97 | 3.5 | Female | No | Sun | Dinner | 3 |
| 20.65 | 3.35 | Male | No | Sat | Dinner | 3 |
| 17.92 | 4.08 | Male | No | Sat | Dinner | 2 |
| 20.29 | 2.75 | Female | No | Sat | Dinner | 2 |

************************************************************************************************