**ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY**

**(Effective from the academic year 2018 -2019)**

**SEMESTER – VII Course Code 18CSL76**

**CIE Marks:** 40 **SEE Marks:** 60

**Number of Contact Hours/Week** 0:0:2 **Total Number of Lab Contact Hours** 36

**Exam Hours** 03 **Credits – 2**

**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.**

**Programs List:**

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 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

**Laboratory Course Outcomes**: The student should be able to:

* Implement and demonstrate AI and ML algorithms.
* Evaluate different algorithms.

**Conduct of Practical Examination:**

* Experiment distribution for laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity.
* For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity.
* Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only.
* Marks Distribution *(Coursed to change in accordance with university regulations)*
* For laboratories having only one part – Procedure + Execution + Viva-Voce: 15+70+15 = 100 Marks
* For laboratories having PART A and PART B

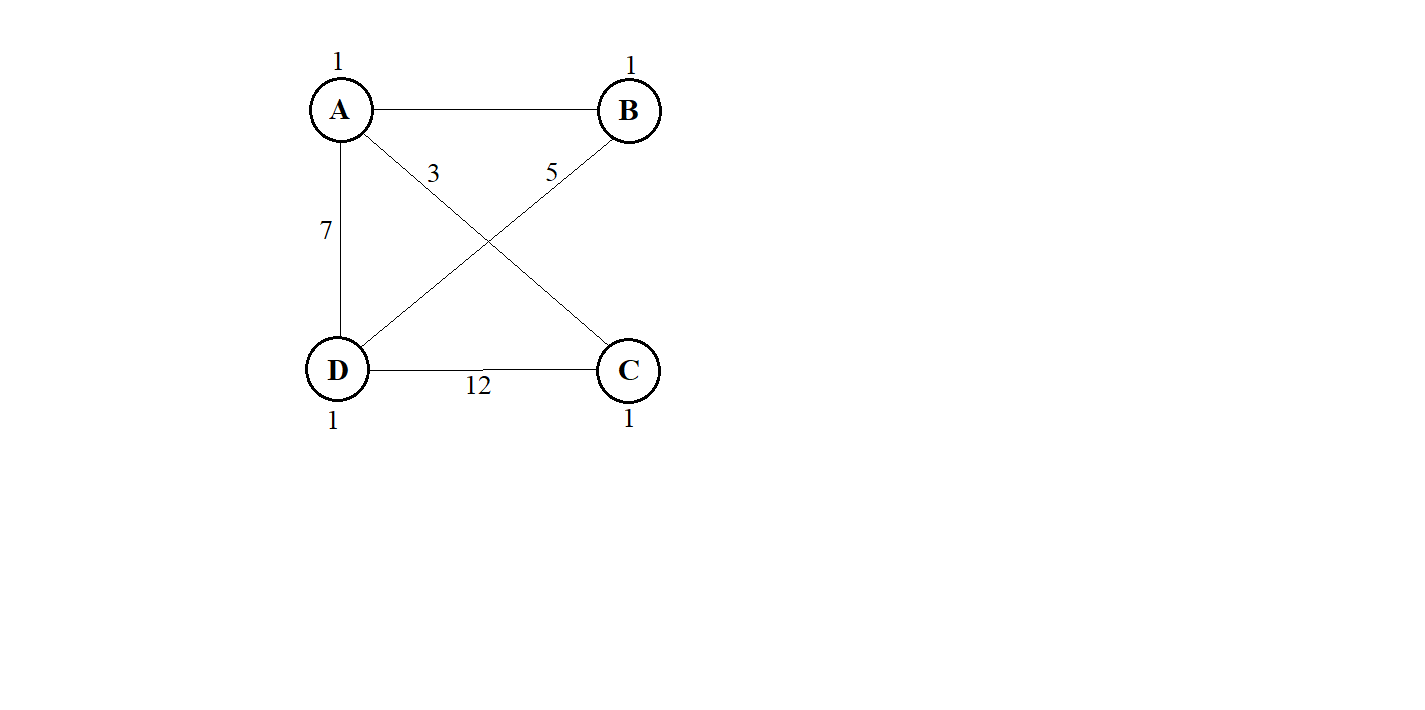
i. Part A – Procedure + Execution + Viva = 6 + 28 + 6 = 40 Marks

ii. Part B – Procedure + Execution + Viva = 9 + 42 + 9 = 60 Marks

PROGRAM.NO.1

***Implement A\* Search algorithm.***

from collections import deque  
class Graph:  
 def \_\_init\_\_(self, adjac\_lis):  
 self.adjac\_lis = adjac\_lis  
  
 def get\_neighbors(self, v):  
 return self.adjac\_lis[v]  
  
 def h(self, n):  
 H = {  
 'A': 1,  
 'B': 1,  
 'C': 1,  
 'D': 1  
 }  
 return H[n]  
  
 def a\_star\_algorithm(self, start, stop):  
 open\_lst = set([start])  
 closed\_lst = set([])  
 poo = {}  
 poo[start] = 0  
  
 par = {}  
 par[start] = start  
  
 while len(open\_lst) > 0:  
 n = None  
 for v in open\_lst:  
 if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):  
 n = v  
  
 if n == None:  
 print('Path does not exist')  
 return None  
  
 if n == stop:  
 reconst\_path = []  
 while par[n] != n:  
 reconst\_path.append(n)  
 n = par[n]  
  
 reconst\_path.append(start)  
 reconst\_path.reverse()  
 print('Path found: {}'.format(reconst\_path))  
 return reconst\_path  
  
 for (m, weight) in self.get\_neighbors(n):  
 if m not in open\_lst and m not in closed\_lst:  
 open\_lst.add(m)  
 par[m] = n  
 poo[m] = poo[n] + weight  
 else:  
 if poo[m] > poo[n] + weight:  
 poo[m] = poo[n] + weight  
 par[m] = n  
 if m in closed\_lst:  
 closed\_lst.remove(n)  
 open\_lst.add(m)  
  
 open\_lst.remove(n)  
 closed\_lst.add(n)  
  
 print('Path does not exist')  
 return None  
  
  
adjac\_lis = {  
 'A': [('B', 1), ('C', 3), ('D', 7)],  
 'B': [('D', 5)],  
 'C': [('D', 12)]  
}  
  
graph1 = Graph(adjac\_lis)  
graph1.a\_star\_algorithm('A', 'D')

***OUTPUT:-***

Path found: ['A', 'B', 'D']

PROGRAM.No.2(AO\* Algorithm)

***Implement AO\* Search algorithm.***

#### AO\* 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.

**AND-OR Graph**

**The figure shows an AND-OR graph**

1. To pass any exam, we have two options, either cheating or hard work.
2. In this graph we are given two choices, first do cheating **or (The red line)** work hard and **(The arc)** pass.
3. When we have more than one choice and we have to pick one, we apply **OR condition** to choose one.(That's what we did here).
   * Basically the **ARC** here denote **AND condition**.
   * Here we have replicated the arc between the work hard and the pass because by doing the hard work possibility of passing an exam is more than cheating.

#### A\* Vs AO\*

1. Both are part of informed search technique and use heuristic values to solve the problem.
2. The solution is guaranteed in both algorithm.
3. A\* **always** gives an **optimal solution** (shortest path with low cost) But It is not guaranteed to that **AO\***  always provide **an optimal solutions**.
4. **Reason:** Because AO\* does not explore all the solution path once it got solution.

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* 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)  
 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: *# 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*print("Graph - 1")  
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()

**Output:**

Graph - 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : A

-----------------------------------------------------------------------------------------

5 ['B', 'C']

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

------------------------------------------------------------

{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}

------------------------------------------------------------

Program.No.3(Candidate Elimination Algorithm)

***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.***

import numpy as np  
import pandas as pd  
  
*# Loading Data from a CSV File*data = pd.DataFrame(data=pd.read\_csv('finds.csv'))  
*# Separating concept features from Target*concepts = np.array(data.iloc[:, 0:-1])  
*# Isolating target into a separate DataFrame*target = np.array(data.iloc[:, -1])  
  
  
def learn(concepts, target):  
 specific\_h = concepts[0].copy()  
  
general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]  
  
 *# The learning iterations* for i, h in enumerate(concepts):  
 *# Checking if the hypothesis has a positive target* if target[i] == "Yes":  
 for x in range(len(specific\_h)):  
 *# Change values in S & G only if values change* if h[x] != specific\_h[x]:  
 specific\_h[x] = '?'  
 general\_h[x][x] = '?'  
  
  
 *# Checking if the hypothesis has a negative target* if target[i] == "No":  
 for x in range(len(specific\_h)):  
 *# For negative hypothesis change values only in G* if h[x] != specific\_h[x]:  
 general\_h[x][x] = specific\_h[x]  
 else:  
 general\_h[x][x] = '?'  
  
 *# find indices where we have empty rows, meaning those that are unchanged* indices = [i for i, val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]  
 for i in indices:  
 *# remove those rows from general\_h* general\_h.remove(['?', '?', '?', '?', '?', '?'])  
 *# Return final values* return specific\_h, general\_h  
  
  
s\_final, g\_final = learn(concepts, target)  
print("Final S:", s\_final, sep="\n")  
print("Final G:", g\_final, sep="\n")  
data.head()

##### OUTPUT:- (Note:-Use Enjoysport.csv file)

Final S:

['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final G:

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

PROGRAM. NO.4(Decision trees)

***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.***

import math  
import csv  
  
  
def load\_csv(filename):  
 lines = csv.reader(open(filename, "r"))  
 dataset = list(lines)  
 headers = dataset.pop(0)  
 return dataset, headers  
  
  
class Node:  
  
 def \_\_init\_\_(self, attribute):  
 self.attribute = attribute  
 self.children = []  
 self.answer = ""  
  
  
def subtables(data, col, delete):  
 dic = {}  
 coldata = [row[col] for row in data]  
 attr = list(set(coldata))  
 for k in attr:  
 dic[k] = []  
 for y in range(len(data)):  
 key = data[y][col]  
 if delete:  
 del data[y][col]  
 dic[key].append(data[y])  
  
 return attr, dic  
  
  
def entropy(S):  
 attr = list(set(S))  
 if len(attr) == 1: *# if all are +ve/-ve then entropy = 0* return 0  
 counts = [0, 0] *# Only two values possible 'yes' or 'no'* for i in range(2):  
 counts[i] = sum([1 for x in S if attr[i] == x]) / (len(S) \* 1.0)  
 sums = 0  
 for cnt in counts:  
 sums += -1 \* cnt \* math.log(cnt, 2)  
 return sums  
  
  
def compute\_gain(data, col):  
 attValues, dic = subtables(data, col, delete=False)  
  
 total\_entropy = entropy([row[-1] for row in data])  
 for x in range(len(attValues)):  
 ratio = len(dic[attValues[x]]) / (len(data) \* 1.0)  
 entro = entropy([row[-1] for row in dic[attValues[x]]])  
 total\_entropy -= ratio \* entro  
 return total\_entropy  
  
  
def build\_tree(data, features):  
 lastcol = [row[-1] for row in data]  
 if (len(set(lastcol))) == 1: *# If all samples have same labels return that label* node = Node("")  
 node.answer = lastcol[0]  
 return node  
 n = len(data[0]) - 1  
 gains = [compute\_gain(data, col) for col in range(n)]  
 split = gains.index(max(gains)) *# Find max gains and returns index* node = Node(features[split]) *# 'node' stores attribute selected* fea = features[:split] + features[split + 1:]  
 attr, dic = subtables(data, split, delete=True)  
 for x in range(len(attr)):  
 child = build\_tree(dic[attr[x]], fea)  
 node.children.append((attr[x], child))  
 return node  
  
  
def print\_tree(node, level):  
 if node.answer != "":  
 print(" " \* level, node.answer) *# Displays leaf node yes/no* return  
 print(" " \* level, node.attribute) *# Displays attribute Name* for value, n in node.children:  
 print(" " \* (level + 1), value)  
 print\_tree(n, level + 2)  
  
  
def classify(node, x\_test, features):  
 if node.answer != "":  
 print(node.answer)  
 return  
 pos = features.index(node.attribute)  
 for value, n in node.children:  
 if x\_test[pos] == value:  
 classify(n, x\_test, features)  
  
  
dataset, features = load\_csv("data3.csv") *# Read Tennis data*node = build\_tree(dataset, features) *# Build decision tree*print("The decision tree for the dataset using ID3 algorithm is ")  
print\_tree(node, 0)  
  
testdata, features = load\_csv("data3.csv")  
for xtest in testdata:  
 print("The test instance : ", xtest)  
 print("The predicted label : ")  
 classify(node, xtest, features)

**Output:**

**NOTE: Use playtennis.csv/data3.csv**

The decision tree for the dataset using ID3 algorithm is

Outlook

rain

Wind

weak

yes

strong

no

overcast

yes

sunny

Humidity

normal

yes

high

no

The test instance : ['sunny', 'hot', 'high', 'weak', 'no']

The predicted label :

no

The test instance : ['sunny', 'hot', 'high', 'strong', 'no']

The predicted label :

no

The test instance : ['overcast', 'hot', 'high', 'weak', 'yes']

The predicted label :

yes

The test instance : ['rain', 'mild', 'high', 'weak', 'yes']

The predicted label :

yes

The test instance : ['rain', 'cool', 'normal', 'weak', 'yes']

The predicted label :

yes

The test instance : ['rain', 'cool', 'normal', 'strong', 'no']

The predicted label :

no

The test instance : ['overcast', 'cool', 'normal', 'strong', 'yes']

The predicted label :

yes

The test instance : ['sunny', 'mild', 'high', 'weak', 'no']

The predicted label :

no

The test instance : ['sunny', 'cool', 'normal', 'weak', 'yes']

The predicted label :

yes

The test instance : ['rain', 'mild', 'normal', 'weak', 'yes']

The predicted label :

yes

The test instance : ['sunny', 'mild', 'normal', 'strong', 'yes']

The predicted label :

yes

The test instance : ['overcast', 'mild', 'high', 'strong', 'yes']

The predicted label :

yes

The test instance : ['overcast', 'hot', 'normal', 'weak', 'yes']

The predicted label :

yes

The test instance : ['rain', 'mild', 'high', 'strong', 'no']

The predicted label :

no

***PROGRAM.No.5 (Back propagation Algorithm)***

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

import numpy as np  
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)  
y = np.array(([92], [86], [89]), dtype=float)  
X = X/np.amax(X, axis=0)  
y = y/100  
  
  
def sigmoid(x):  
 return 1/(1+np.exp(-x))  
  
  
def derivatives\_sigmoid(x):  
 return x\*(1-x)  
  
  
epoch = 7000  
lr = 0.1  
inputLayer\_neurons = 2  
hiddenLayer\_neurons = 3  
output\_neurons = 1  
wh = np.random.uniform(size=(inputLayer\_neurons, hiddenLayer\_neurons))  
bh = np.random.uniform(size=(1, hiddenLayer\_neurons))  
wout = np.random.uniform(size=(hiddenLayer\_neurons, output\_neurons))  
bout = np.random.uniform(size=(1, output\_neurons))  
  
for i in range(epoch):  
 hinp1 = np.dot(X, wh)  
  
 hinp = hinp1 + bh  
 hlayer\_act = sigmoid(hinp)  
 outinp1 = np.dot(hlayer\_act, wout)  
 outinp = outinp1 + bout  
 output = sigmoid(outinp)  
 EO = y - output  
  
 outgrad = derivatives\_sigmoid(output)  
 d\_output = EO \* outgrad  
 EH = d\_output.dot(wout.T)  
 hiddengrad = derivatives\_sigmoid(hlayer\_act)  
 d\_hiddenlayer = EH \* hiddengrad  
 wout += hlayer\_act.T.dot(d\_output)\*lr  
 wh += X.T.dot(d\_hiddenlayer)\*lr  
  
print('Input: \n' + str(X))  
print('Actual Output: \n' + str(y))  
print('Predicted output: \n', output)

***OUTPUT:-***

Input:

[[0.66666667 1. ]

[0.33333333 0.55555556]

[1. 0.66666667]]

Actual Output:

[[0.92]

[0.86]

[0.89]]

Predicted output:

[[0.89871247]

[0.871736 ]

[0.89883643]]

***PROGRAM.NO.6 (Naive bayes Classifier)***

***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.***

import pandas as pd  
msg=pd.read\_csv('naivetext1.csv', names=['message', 'label'])  
print('The dimensions of the dataset', msg.shape)  
msg['labelnum']=msg.label.map({'pos': 1, 'neg': 0})  
X=msg.message  
y=msg.labelnum  
*#print(X)  
#print(y)  
#splitting the dataset into train and test data*from sklearn.model\_selection import train\_test\_split  
xtrain,xtest,ytrain,ytest=train\_test\_split(X,y)  
print('dimensions of train and test sets')  
print(xtrain.shape)  
print(xtest.shape)  
print(ytrain.shape)  
print(ytest.shape)  
  
*#output of count vectoriser is a sparse matrix*from sklearn.feature\_extraction.text import CountVectorizer  
count\_vect = CountVectorizer()  
xtrain\_dtm = count\_vect.fit\_transform(xtrain)  
*#print(count\_vect.vocabulary\_)*xtest\_dtm=count\_vect.transform(xtest)  
  
*# Training Naive Bayes (NB) classifier on training data.*from sklearn.naive\_bayes import MultinomialNB  
clf = MultinomialNB().fit(xtrain\_dtm,ytrain)  
*#print(clf)*predicted = clf.predict(xtest\_dtm)  
*#printing accuracy metrics*from sklearn import metrics  
print('Accuracy metrics')  
print('Accuracy of the classifer is',metrics.accuracy\_score(ytest,predicted))  
print('Confusion matrix')  
print(metrics.confusion\_matrix(ytest,predicted))  
print('Recall and Precison ')  
print(metrics.recall\_score(ytest,predicted))  
print(metrics.precision\_score(ytest,predicted))

###### OUTPUT :- (NOTE:-Use naivetext1.csv as dataset)

The dimensions of the dataset (18, 2)

dimensions of train and test sets

(13,)

(5,)

(13,)

(5,)

Accuracy metrics

Accuracy of the classifer is 0.6

Confusion matrix

[[1 1]

[1 2]]

Recall and Precison

0.6666666666666666

0.6666666666666666

###### PROGRAM .No.7(K-Means and EM algorithm)

***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***.

import numpy as np import pandas as pd

from matplotlib import pyplot as plt from sklearn.mixture import GaussianMixture from sklearn.cluster import KMeans

data = pd.read\_csv('data/ex.csv')

f1 = data['V1'].values f2 = data['V2'].values X = np.array(list(zip(f1, f2))) print("x: ", X)

print('Graph for whole dataset') plt.scatter(f1, f2, c='black') # size can be set by adding s=size as param plt.show()

kmeans = KMeans(2) labels = kmeans.fit(X).predict(X)

print("labels for kmeans:", labels)

print('Graph using Kmeans Algorithm')

plt.scatter(f1, f2, c=labels)

centroids = kmeans.cluster\_centers\_ print("centroids:", centroids) plt.scatter(centroids[:, 0], centroids[:, 1], marker='\*', c='red')

plt.show()

gmm = GaussianMixture(2) labels = gmm.fit(X).predict(X)

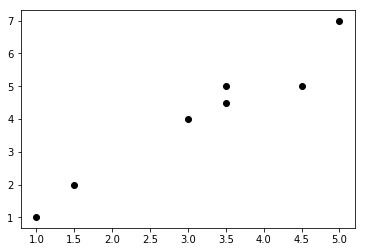
print("Labels for GMM: ", labels)

print('Graph using EM Algorithm') plt.scatter(f1, f2, c=labels) plt.show()

OUTPUT:- (NOTE:-USE ex.csv )

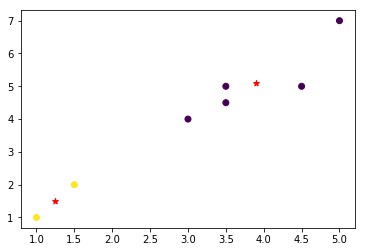
x: [[ 1. 1. ] [ 1.5 2. ] [ 3. 4. ] [ 5. 7. ] [ 3.5 5. ] [ 4.5 5. ] [ 3.5 4.5]]

Graph for whole dataset



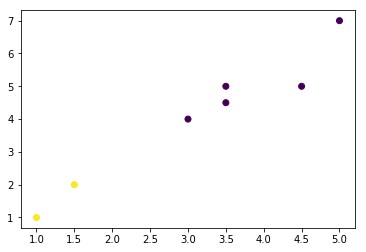
labels for kmeans: [1 1 0 0 0 0 0] Graph using Kmeans Algorithm

centroids: [[ 3.9 5.1 ] [ 1.25 1.5 ]]



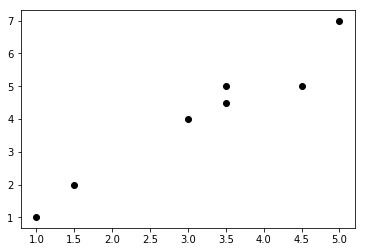
Labels for GMM: [1 1 0 0 0 0 0]

Graph using EM Algorithm



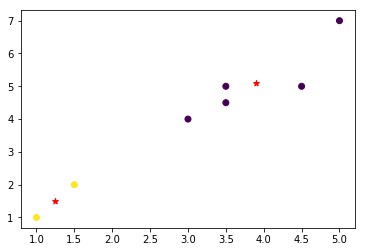
x: [[ 1. 1. ] [ 1.5 2. ] [ 3. 4. ] [ 5. 7. ] [ 3.5 5. ] [ 4.5 5. ] [ 3.5 4.5]]

Graph for whole dataset



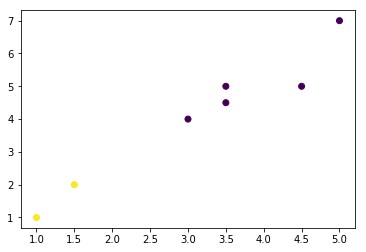
labels for kmeans: [1 1 0 0 0 0 0] Graph using Kmeans Algorithm

centroids: [[ 3.9 5.1 ] [ 1.25 1.5 ]]



Labels for GMM: [1 1 0 0 0 0 0]

Graph using EM Algorithm



###### PROGRAM .NO.8(KNN)

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

from sklearn.model\_selection import train\_test\_split  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.metrics import classification\_report,confusion\_matrix  
from sklearn import datasets  
iris = datasets.load\_iris()  
iris\_data = iris.data  
iris\_labels = iris.target  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(iris\_data, iris\_labels, test\_size=0.20)  
classifier = KNeighborsClassifier(n\_neighbors=5)  
classifier.fit(x\_train, y\_train)  
y\_pred = classifier.predict(x\_test)  
print('Confusion matrix is as follows')  
print(confusion\_matrix(y\_test, y\_pred))  
print('Accuracy Metrics')  
print(classification\_report(y\_test, y\_pred))

OUTPUT:-

Confusion matrix is as follows

[[ 9 0 0]

[ 0 8 0]

[ 0 1 12]]

Accuracy Metrics

precision recall f1-score support

0 1.00 1.00 1.00 9

1 0.89 1.00 0.94 8

2 1.00 0.92 0.96 13

accuracy 0.97 30

macro avg 0.96 0.97 0.97 30

weighted avg 0.97 0.97 0.97 30

###### PROGRAM .NO .9(Locally Weighted Regression Algorithm)

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

import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
  
  
def kernel(point, xmat, k):  
 m, n = np.shape(xmat)  
 weights = np.mat(np.eye((m)))  
 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  
  
  
*# load data points*data = pd.read\_csv('tips.csv')  
bill = np.array(data.total\_bill)  
tip = np.array(data.tip)  
  
*# preparing and add 1 in bill*mbill = np.mat(data.total\_bill)  
mtip = np.mat(data.tip)  
m = np.shape(mbill)[1]  
one = np.mat(np.ones(m))  
X = np.hstack((one.T, mbill.T))  
  
*# set k here*ypred = localWeightRegression(X, mtip, 0.5)  
SortIndex = X[:, 1].argsort(0)  
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()

***OUTPUT:-***

