VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Belgaum-590014, Karnataka.

Artificial Intelligence and Machine Learning

[18CSL76]

FOR
7th semester CSE

BACHELOR OF ENGINEERING
In
COMPUTER SCIENCE & ENGINEERING

Lab in charge
Dr. Aruna M.G
Associate Professor, Dept. of CSE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
M S ENGINEERING COLLEGE
Navarathna Agrahara, Sadahalli P.O. Bangalore-562110

CONTENTS

Experiment - 01

Aim : Implement A* Search algorithm.

PSEUDO CODE:

Step 1 : let openList equal empty list of nodes.
Step 2 : let closedList equal empty list of nodes.
Step 3 : put startNode on the openList (leave it's f at zero)
while openList is not empty
Step 4 : let currentNode equal the node with the least f value
remove currentNode from the openList
add currentNode to the closedList
Step 5 : if currentNode is the goal
You've found the exit!
Step 6 : let children of the currentNode equal the adjacent nodes
Step 7 : for each child in the children
if child is in the closedList
continue to beginning of for loop
child.g = currentNode.g + distance b/w child and current
child.h = distance from child to end
child.f = child.g + child.h
if child.position is in the openList's nodes positions
if child.g is higher than the openList node's g
continue to beginning of for loop
add the child to the openList

PROGRAM:

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        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):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
```

```
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None
        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
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

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': [('A', 2), ('C', 1), ('G', 9)],
    'C': [('B', 1)],
    'D': [('E', 6), ('G', 1)],
    'E': [('A', 3), ('D', 6)],
    'G': [('B', 9), ('D', 1)]
}
```

aStarAlgo('A', 'G')

OUTPUT:

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

Experiment - 02

Aim : Implement AO* Search algorithm.

PSEUDO CODE:

1. Initialise the graph to start node
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
3. Pick any of these nodes and expand it and if it has no successors call this value *FUTILITY* otherwise calculate only *f'* for each of the successors.
4. If *f'* is 0 then mark the node as *SOLVED*
5. Change the value of *f'* for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as *SOLVED* then mark the parent node as *SOLVED*.
7. If starting node is *SOLVED* or value greater than *FUTILITY*, stop, else repeat from 2

PROGRAM:

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self):
        self.aoStar(self.start, False)

    def getNeighbors(self, v):
        return self.graph.get(v,'')

    def getStatus(self,v):
        return self.status.get(v,0)

    def setStatus(self,v, val):
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0)

    def setHeuristicNodeValue(self, n, value):
```

```
        self.H[n]=value
    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START
NODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v):
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v):
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)
            if flag==True:
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList
                flag=False
            else:
                if minimumCost>cost:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList
        return minimumCost, costToChildNodeListDict[minimumCost]
    def aoStar(self, v, backTracking):
        print("HEURISTIC VALUES :", self.H)
        print("SOLUTION GRAPH :", self.solutionGraph)
        print("PROCESSING NODE :", v)
        print("-------------------------------------------------------------------------------------------")
        if self.getStatus(v) >= 0:
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
            print(minimumCost, childNodeList)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v,len(childNodeList))
            solved=True
            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode)!=-1:
                    solved=solved & False
            if solved==True:
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList
            if v!=self.start:
                self.aoStar(self.parent[v], True)
```

```
            if backTracking==False:
                for childNode in childNodeList:
                    self.setStatus(childNode,0)
                    self.aoStar(childNode, False)


print ("Graph - 2")
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
graph2 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'D': [[('E', 1), ('F', 1)]]
}
G2 = Graph(graph2, h2, 'A')
G2.applyAOStar()
G2.printSolution()
```

OUTPUT:

```
Graph - 2
HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-------------------------------------------------------------------------------------
11 ['D']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : 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 NODE : A
-------------------------------------------------------------------------------------
11 ['D']
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {}
PROCESSING NODE : E
-------------------------------------------------------------------------------------
0 []
HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4, 'G': 5, 'H': 7}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : 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 NODE : 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 NODE : 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 NODE : 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 NODE : A
--------------------------------------------------------------------------------
3 ['D']
FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A
------------------------------------------------------------
{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}
```

Experiment - 03

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.

PSEUDO CODE:

1. Load data set
2. G <-maximally general hypotheses in H
3. S <- maximally specific hypotheses in H
4. For each training example d=<x,c(x)>

Case 1 : 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 that is more general than another hypothesis 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

o h consistent with d
o Some member of S is more specific than h
• Remove from G any hypothesis that is less general than another hypothesis in G

PROGRAM:

```
import numpy as np
import pandas as pd

data = pd.DataFrame(data=pd.read_csv('lab2.csv'))
concepts = np.array(data.iloc[:,0:-1])

target = np.array(data.iloc[:,-1])
def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("initialization of specific_h and general_h")
```

```
    print(specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print(general_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] = '?'
                    print(" steps of Candidate Elimination Algorithm",i+1)
                    print("Specific_h ",i+1,"\n ")
                    print(specific_h)
                    print("general_h ", i+1, "\n ")
                    print(general_h)

    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
            general_h.remove(['?', '?', '?', '?', '?', '?'])

    return specific_h, general_h

s_final, g_final = learn(concepts, target)
print("Final Specific_h:", s_final, sep="\n")
print("Final General_h:", g_final, sep="\n")
```

DATASET:

sunny	wam	normal	strong	warm	same	yes
sunny	warm	high	strong	warm	same	yes
rainy	cold	high	strong	warm	change	no
sunny	warm	high	strong	cool	chane	yes

OUTPUT:

```
initialization of specific_h and general_h
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
```

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
steps of Candidate Elimination Algorithm 3
Specific_h 3

['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
general_h 3

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
steps of Candidate Elimination Algorithm 3
Specific_h 3

['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
general_h 3

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
steps of Candidate Elimination Algorithm 8
Specific_h 8

['?' '?' '?' 'Strong' '?' '?']
general_h 8

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
Final Specific_h:
['?' '?' '?' 'Strong' '?' '?']
Final General_h:
[['?', '?', '?', 'Strong', '?', '?']]

Experiment - 04

Aim: 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.

PSEUDO CODE:

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*

PROGRAM:

```
import numpy as np
import math
from data_loader import read_data
class Node:
    def __init__(self, attribute):
        self.attribute = attribute
        self.children = []
        self.answer = ""

    def __str__(self):
        return self.attribute

def subtables(data, col, delete):
    dict = {}
    items = np.unique(data[:, col])
    count = np.zeros((items.shape[0], 1), dtype=np.int32)
```

```
    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                count[x] += 1

    for x in range(items.shape[0]):
        dict[items[x]] = np.empty((int(count[x]), data.shape[1]), dtype="|S32")
        pos = 0
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                dict[items[x]][pos] = data[y]
                pos += 1
        if delete:
            dict[items[x]] = np.delete(dict[items[x]], col, 1)

    return items, dict

def entropy(S):
    items = np.unique(S)

    if items.size == 1:
        return 0

    counts = np.zeros((items.shape[0], 1))
    sums = 0

    for x in range(items.shape[0]):
        counts[x] = sum(S == items[x]) / (S.size * 1.0)

    for count in counts:
        sums += -1 * count * math.log(count, 2)
    return sums

def gain_ratio(data, col):
    items, dict = subtables(data, col, delete=False)

    total_size = data.shape[0]
    entropies = np.zeros((items.shape[0], 1))
    intrinsic = np.zeros((items.shape[0], 1))

    for x in range(items.shape[0]):
        ratio = dict[items[x]].shape[0]/(total_size * 1.0)
        entropies[x] = ratio * entropy(dict[items[x]][:, -1])
        intrinsic[x] = ratio * math.log(ratio, 2)

    total_entropy = entropy(data[:, -1])
    iv = -1 * sum(intrinsic)
```

```
    for x in range(entropies.shape[0]):
        total_entropy -= entropies[x]

    return total_entropy / iv

def create_node(data, metadata):

    if (np.unique(data[:, -1])).shape[0] == 1:
        node = Node("")
        node.answer = np.unique(data[:, -1])[0]
        return node

    gains = np.zeros((data.shape[1] - 1, 1))

    for col in range(data.shape[1] - 1):
        gains[col] = gain_ratio(data, col)

    split = np.argmax(gains)

    node = Node(metadata[split])
    metadata = np.delete(metadata, split, 0)

    items, dict = subtables(data, split, delete=True)

    for x in range(items.shape[0]):
        child = create_node(dict[items[x]], metadata)
        node.children.append((items[x], child))

    return node

def empty(size):
    s = ""
    for x in range(size):
        s += "   "
    return s

def print_tree(node, level):
    if node.answer != "":
        print(empty(level), node.answer)
        return

    print(empty(level), node.attribute)

    for value, n in node.children:
        print(empty(level + 1), value)
        print_tree(n, level + 2)
```

```
metadata, traindata = read_data("tennis.data")

data = np.array(traindata)

node = create_node(data, metadata)

print_tree(node, 0)
```

DATASET:

outlook	temp	humidity	wind	answer
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

OUTPUT:

```
outlook
   overcast
      b'yes'
   rain
      wind
         b'strong'
            b'no'
         b'weak'
            b'yes'
   sunny
      humidity
         b'high'
            b'no'
         b'normal'
            b'yes'
```

Experiment -05

Aim: Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate datasets.

PSEUDO CODE:

function BackProp (D, η, n_{in}, n_{hidden}, n_{out})

- D is the training set consists of m pairs: $\{(x_i, y_i)^m\}$
- η is the learning rate as an example (0.1)
- n_{in}, n_{hidden} e n_{out} are the numbero of imput hidden and output unit of neural network

Make a feed-forward network with n_{in}, n_{hidden} e n_{out} units
Initialize all the weight to short randomly number (es. [-0.05 0.05])
Repeat until termination condition are verifyed:
For any sample in D:
Forward propagate the network computing the output o_u of every unit u of the network
Back propagate the errors onto the network:

- For every output unit k, compute the error δ_k: $\delta_k = o_k(1-o_k)(t_k - o_k)$
- For every hidden unit h compute the error δ_h: $\delta_h = o_h(1-o_h)\sum_{k \in outputs} w_{kh}\delta_k$
- Update the network weight w_{ji}: $w_{ji} = w_{ji} + \Delta w_{ji}$, where $\Delta w_{ji} = \eta \delta_j x_{ji}$

(x_{ji} is the input of unit j from coming from unit i)

PROGRAM:

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0)
def sigmoid (x):
   return (1/(1 + np.exp(-x)))
def 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
    bout += np.sum(d_output, axis=0,keepdims=True) *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:
[[92.]
 [86.]
 [89.]]
Predicted Output:
 [[0.99999913]
 [0.99999746]
 [0.99999922]]
```

Experiment-06

Aim: Write a program to implement the naive Bayesian classifier for a sample training data set stored as .csv file. Compute the accuracy of the classifier, considering few data sets.

PSEUDO CODE:

Input:

Training dataset T,

F= (f_1, f_2, f_3,.., f_n) // value of the predictor variable in testing dataset.

Output:

A class of testing dataset.

Step:

1. Read the training dataset T;
2. Calculate the mean and standard deviation of the predictor variables in each class;
3. Repeat

   Calculate the probability of f_i using the gauss density equation in each class;

   Until the probability of all predictor variables (f_1, f_2, f_3,.., f_n) has been calculated.
4. Calculate the likelihood for each class;
5. Get the greatest likelihood;

PROGRAM:

```
import csv
import random
import math

def loadCsv(filename):
        lines = csv.reader(open(filename, "r"))
        dataset = list(lines)
        for i in range(len(dataset)):
                dataset[i] = [float(x) for x in dataset[i]]
        return dataset

def splitDataset(dataset, splitRatio):
        trainSize = int(len(dataset) * splitRatio)
        trainSet = []
        copy = list(dataset)
        while len(trainSet) < trainSize:
```

```
                index = random.randrange(len(copy))
                trainSet.append(copy.pop(index))
        return [trainSet, copy]

def separateByClass(dataset):
        separated = {}
        for i in range(len(dataset)):
                vector = dataset[i]
                if (vector[-1] not in separated):
                        separated[vector[-1]] = []
                separated[vector[-1]].append(vector)
        return separated

def mean(numbers):
        return sum(numbers)/float(len(numbers))

def stdev(numbers):
        avg = mean(numbers)
        variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
        return math.sqrt(variance)

def summarize(dataset):
        summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
        del summaries[-1]
        return summaries

def summarizeByClass(dataset):
        separated = separateByClass(dataset)
        summaries = {}
        for classValue, instances in separated.items():
                summaries[classValue] = summarize(instances)
        return summaries

def calculateProbability(x, mean, stdev):
        exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))
        return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent

def calculateClassProbabilities(summaries, inputVector):
        probabilities = {}
        for classValue, classSummaries in summaries.items():
                probabilities[classValue] = 1
                for i in range(len(classSummaries)):
                        mean, stdev = classSummaries[i]
                        x = inputVector[i]
                        probabilities[classValue] *= calculateProbability(x, mean, stdev)
        return probabilities

def predict(summaries, inputVector):
```

```
        probabilities = calculateClassProbabilities(summaries, inputVector)
        bestLabel, bestProb = None, -1
        for classValue, probability in probabilities.items():
                if bestLabel is None or probability > bestProb:
                        bestProb = probability
                        bestLabel = classValue
        return bestLabel

def getPredictions(summaries, testSet):
        predictions = []
        for i in range(len(testSet)):
                result = predict(summaries, testSet[i])
                predictions.append(result)
        return predictions

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

def main():
        filename = 'naivedata.csv'
        splitRatio = 0.67
        dataset = loadCsv(filename)
        trainingSet, testSet = splitDataset(dataset, splitRatio)
        print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),
len(trainingSet), len(testSet)))
        # prepare model
        summaries = summarizeByClass(trainingSet)
        # test model
        predictions = getPredictions(summaries, testSet)
        accuracy = getAccuracy(testSet, predictions)
        print('Accuracy of the classifier is : {0}%'.format(accuracy))

main()
```

DATASET:

6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1
5	116	74	0	0	25.6	0.201	30	0
3	78	50	32	88	31	0.248	26	1
10	115	0	0	0	35.3	0.134	29	0
2	197	70	45	543	30.5	0.158	53	1
8	125	96	0	0	0	0.232	54	1
4	110	92	0	0	37.6	0.191	30	0
10	168	74	0	0	38	0.537	34	1
10	139	80	0	0	27.1	1.441	57	0

OUTPUT:

```
Split 768 rows into train=514 and test=254 rows
Accuracy of the classifier is : 72.44094488188976%
```

Experiment - 07

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.

PSEUDO CODE:

1. Load data set
2. Clusters the data into k groups where k is predefined.
3. Select k points at random as cluster centers.
4. Assign objects to their closest cluster center according to the Euclidean distance function.
5. Calculate the centroid or mean of all objects in each cluster.
6. Repeat steps 3, 4 and 5 until the same points are assigned to each cluster in consecutive rounds.

PROGRAM:

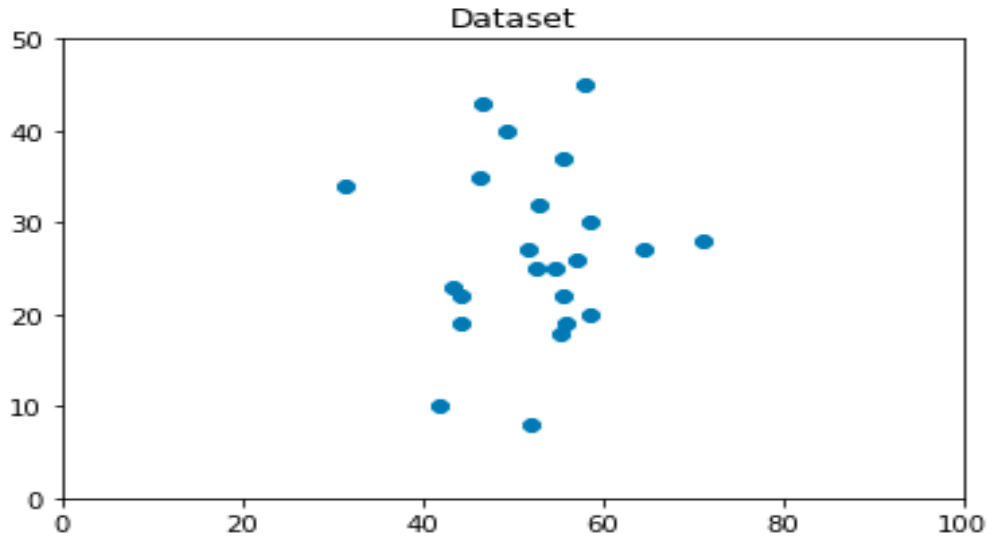
```
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
import pandas as pd
X=pd.read_csv("kmeansdata.csv")
x1 = X['Distance_Feature'].values
x2 = X['Speeding_Feature'].values
X = np.array(list(zip(x1, x2))).reshape(len(x1), 2)
plt.plot()
plt.xlim([0, 100])
plt.ylim([0, 50])
plt.title('Dataset')
plt.scatter(x1, x2)
plt.show()
gmm = GaussianMixture(n_components=3)
gmm.fit(X)
em_predictions = gmm.predict(X)
print("\nEM predictions")
print(em_predictions)
print("mean:\n",gmm.means_)
print('\n')
print("Covariances\n",gmm.covariances_)
print(X)
plt.title('Exceptation Maximum')
plt.scatter(X[:,0], X[:,1],c=em_predictions,s=50)
plt.show()
```

```
import matplotlib.pyplot as plt1
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
print(kmeans.cluster_centers_)
print(kmeans.labels_)
plt.title('KMEANS')
plt1.scatter(X[:,0], X[:,1], c=kmeans.labels_, cmap='rainbow')
plt1.scatter(kmeans.cluster_centers_[:,0] ,kmeans.cluster_centers_[:,1], color='black')
```

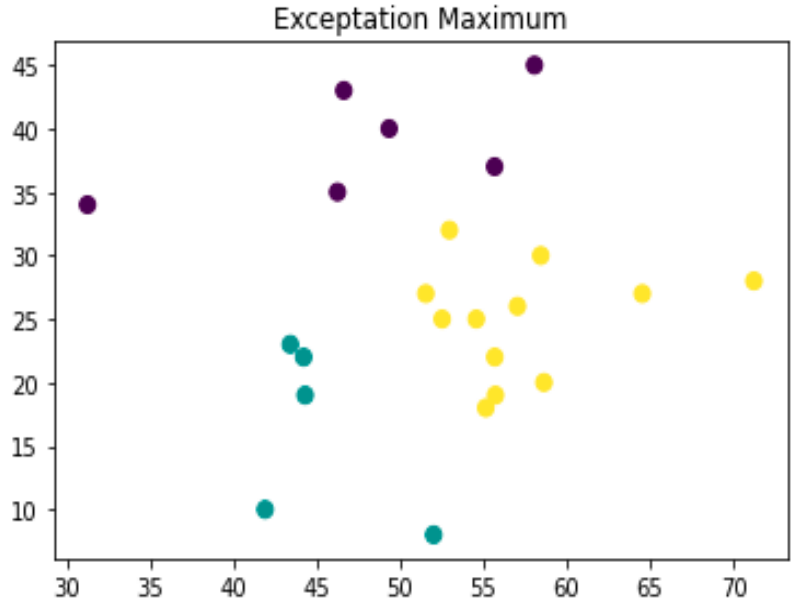
DATASET:

Driver_ID	Distance_	Speeding_Feature
3.42E+09	71.24	28
3.42E+09	52.53	25
3.42E+09	64.54	27
3.42E+09	55.69	22
3.42E+09	54.58	25
3.42E+09	41.91	10
3.42E+09	58.64	20
3.42E+09	52.02	8
3.42E+09	31.25	34

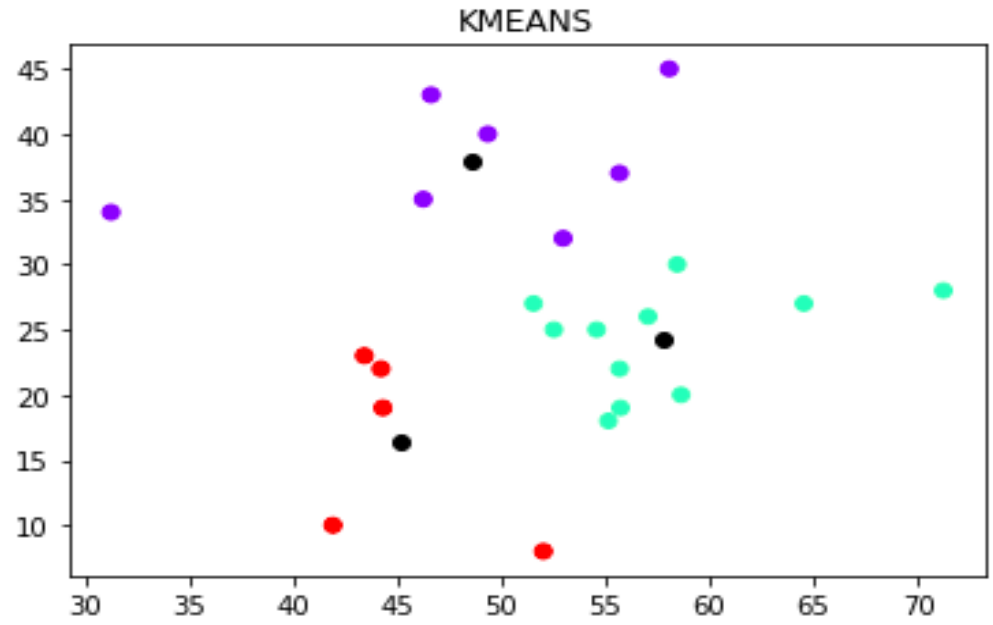
OUTPUT:



```
EM predictions
[2 2 2 2 2 1 2 1 0 1 0 0 1 2 0 2 0 2 2 2 1 0 2]
mean:
 [[47.81649609 38.96676066]
 [45.52364659 15.12663491]
 [56.2895734  24.72458957]]


Covariances
 [[[ 74.43313242  21.91196052]
  [ 21.91196052  16.92381491]]

 [[ 15.50444953 -11.70433701]
  [-11.70433701  39.34887084]]

 [[ 39.25509622   6.40503301]
  [  6.40503301  18.32286182]]]
[[71.24 28.  ]
 [52.53 25.  ]
 [64.54 27.  ]
 [55.69 22.  ]
 [54.58 25.  ]
 [41.91 10.  ]
 [58.64 20.  ]
 [52.02  8.  ]
 [31.25 34.  ]
 [44.31 19.  ]
 [49.35 40.  ]
 [58.07 45.  ]
 [44.22 22.  ]
 [55.73 19.  ]
 [46.63 43.  ]
 [52.97 32.  ]
 [46.25 35.  ]
 [51.55 27.  ]
 [57.05 26.  ]
 [58.45 30.  ]
 [43.42 23.  ]
 [55.68 37.  ]
 [55.15 18.  ]]
```




```
[[48.6        38.        ]
 [57.74090909 24.27272727]
 [45.176      16.4       ]]
[1 1 1 1 1 2 1 2 0 2 0 0 2 1 0 0 0 1 1 1 2 0 1]
```



Experiment - 08

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.

PSEUDO CODE:

. Load the data
2. Initialize the value of k
3. For getting the predicted class, iterate from 1 to total number of training data points
i. Calculate the distance between test data and each row of training data. Here we will use Euclidean distance as our distance metric since it's the most popular method. The other metrics that can be used are Chebyshev, cosine, etc.
ii. Sort the calculated distances in ascending order based on distance values .
iii. Get top k rows from the sorted array
4. Get the most frequent class of these rows i.e Get the labels of the selected K entries
5. Return the predicted class

- If regression, return the mean of the K labels
- If classification, return the mode of the K labels

PROGRAM:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
import pandas as pd
dataset = pd.read_csv('iris.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 5].values
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0, test_size = 0.25)
classifier = KNeighborsClassifier(n_neighbors = 5)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print('Confusion matrix is as follows\n',cm)
print('Accuracy Metrics')
print(classification_report(y_test, y_pred))
print("correct prediction",accuracy_score(y_test, y_pred))
print("wrong prediction",(1-accuracy_score(y_test, y_pred)))
```

DATASET:

Id	SepalLeng	SepalWid	PetalLeng	PetalWidt	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5	3.6	1.4	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	Iris-setosa
7	4.6	3.4	1.4	0.3	Iris-setosa
8	5	3.4	1.5	0.2	Iris-setosa
9	4.4	2.9	1.4	0.2	Iris-setosa

OUTPUT:

```
Confusion matrix is as follows
 [[13  0  0]
 [ 0 16  0]
 [ 0  0  9]]
Accuracy Metrics
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        13
Iris-versicolor       1.00      1.00      1.00        16
 Iris-virginica       1.00      1.00      1.00         9

       accuracy                           1.00        38
      macro avg       1.00      1.00      1.00        38
   weighted avg       1.00      1.00      1.00        38

correct prediction 1.0
wrong prediction 0.0
```

Experiment - 09

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.

PSEUDO CODE:

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:

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

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

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

6. Prediction = x0*β

PROGRAM:

```
from numpy import *
import operator
from os import listdir
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import numpy.linalg
from scipy.stats.stats import pearsonr
def kernel(point,xmat, k):
    m,n = shape(xmat)
    weights = mat(eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j,j] = 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 = shape(xmat)
    ypred = zeros(m)
    for i in range(m):
        ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
    return ypred
data = pd.read_csv('tips.csv')
```

```
bill = array(data.total_bill)
tip = array(data.tip)
mbill = mat(bill)
mtip = mat(tip)
m= shape(mbill)[1]
one = mat(ones(m))
X= hstack((one.T,mbill.T))
ypred = localWeightRegression(X,mtip,0.2)
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();
```

DATASET:

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

OUTPUT: