PROGRAM 1:

```
#!/usr/bin/env python
# coding: utf-8

# In[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')
```

PROGRAM 2:

```
# Recursive implementation of AO* aglorithm by Dr. K PARAMESHA, Professor, VVCE,
Mysuru, INDIA

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)
            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
```

PROGRAM 3:

```
import random
import csv
def g_0(n):
    return ("?",)*n

def s_0(n):
    return ('Ф',)*n
def more_general(h1, h2):
    more_general_parts = []
    for x, y in zip(h1, h2):
        mg = x == "?" or (x != "Ф" and (x == y or y == "Ф"))
        more_general_parts.append(mg)
    return all(more_general_parts)
def fulfills(example, hypothesis):
    ### the implementation is the same as for hypotheses:
    return more_general(hypothesis, example)

def min_generalizations(h, x):
    h_new = list(h)
    for i in range(len(h)):
        if not fulfills(x[i:i+1], h[i:i+1]):
            h_new[i] = '?' if h[i] != 'Ф' else x[i]
    return [tuple(h_new)]
def min_specializations(h, domains, x):
    results = []
    for i in range(len(h)):
        if h[i] == "?":
            for val in domains[i]:
                if x[i] != val:
                    h_new = h[:i] + (val,) + h[i+1:]
                    results.append(h_new)
        elif h[i] != "Ф":
            h_new = h[:i] + ('Ф',) + h[i+1:]
            results.append(h_new)
    return results
with open('trainingexamples.csv')  as csvFile:
        examples = [tuple(line) for line in csv.reader(csvFile)]
def get_domains(examples):
    d = [set() for i in examples[0]]
    for x in examples:
        for i, xi in enumerate(x):
            d[i].add(xi)
    return [list(sorted(x)) for x in d]
get_domains(examples)


def candidate_elimination(examples):
    domains = get_domains(examples)[:-1]

    G = set([g_0(len(domains))])
    S = set([s_0(len(domains))])
```

```
    i = 0
    print("\n G[{0}]:".format(i), G)
    print("\n S[{0}]:".format(i), S)
    for xcx in examples:
        i = i + 1
        x, cx = xcx[:-1], xcx[-1]  # Splitting data into attributes and decisions
        if cx == 'Y':  # x is positive example
            G = {g for g in G if fulfills(x, g)}
            S = generalize_S(x, G, S)
        else:  # x is negative example
            S = {s for s in S if not fulfills(x, s)}
            G = specialize_G(x, domains, G, S)
        print("\n G[{0}]:".format(i), G)
        print("\n S[{0}]:".format(i), S)
    return
def generalize_S(x, G, S):
    S_prev = list(S)
    for s in S_prev:
        if s not in S:
            continue
        if not fulfills(x, s):
            S.remove(s)
            Splus = min_generalizations(s, x)
            ## keep only generalizations that have a counterpart in G
            S.update([h for h in Splus if any([more_general(g,h)
                                               for g in G])])
            ## remove hypotheses less specific than any other in S
            S.difference_update([h for h in S if
                                 any([more_general(h, h1)
                                      for h1 in S if h != h1])])
    return S
def specialize_G(x, domains, G, S):
    G_prev = list(G)
    for g in G_prev:
        if g not in G:
            continue
        if fulfills(x, g):
            G.remove(g)
            Gminus = min_specializations(g, domains, x)
            ## keep only specializations that have a conuterpart in S
            G.update([h for h in Gminus if any([more_general(h, s)
                                                for s in S])])
            ## remove hypotheses less general than any other in G
            G.difference_update([h for h in G if
                                 any([more_general(g1, h)
                                      for g1 in G if h != g1])])
    return G
candidate_elimination(examples)
```

PROGRAM 4:

```
# -*- coding: utf-8 -*-
"""ID3_2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1amD6hMNBxoXXebhuF70NJfNnQu18n9QZ
"""

import pandas as pd
from pandas import DataFrame
df_tennis = pd.read_csv('tennis.csv')
print("\n Given Play Tennis Data Set:\n\n",df_tennis)
#df_tennis.columns[0]
```

```
df_tennis.keys()[0]

#Function to calculate the entropy of probaility of observations
# -p*log2*p

def entropy(probs):
    import math
    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(a_list):
    from collections import Counter
    cnt = Counter(x for x in a_list)   # Counter calculates the propotion of class
    num_instances = len(a_list)*1.0   # = 14
    print("\n Number of Instances of the Current Sub Class is
{0}:".format(num_instances ))
    probs = [x / num_instances for x in cnt.values()]  # x means no of YES/NO
    print("\n Classes:",min(cnt),max(cnt))
    print(" \n Probabilities of Class {0} is {1}:".format(min(cnt),min(probs)))
    print(" \n Probabilities of Class {0} is {1}:".format(max(cnt),max(probs)))
    return entropy(probs) # Call Entropy :

    # The initial entropy of the YES/NO attribute for our dataset.
print("\n  INPUT DATA SET FOR ENTROPY CALCULATION:\n", df_tennis['PlayTennis'])

total_entropy = entropy_of_list(df_tennis['PlayTennis'])
print("\n Total Entropy of PlayTennis Data Set:",total_entropy)

def information_gain(df, split_attribute_name, target_attribute_name, trace=0):
    print("Information Gain Calculation of ",split_attribute_name)
    '''
    Takes a DataFrame of attributes, and quantifies the entropy of a target
    attribute after performing a split along the values of another attribute.
    '''
    # Split Data by Possible Vals of Attribute:
    df_split = df.groupby(split_attribute_name)
    for name,group in df_split:
            print("Name:\n",name)
            print("Group:\n",group)
    # Calculate Entropy for Target Attribute, as well as
    # Proportion of Obs in Each Data-Split
    nobs = len(df.index) * 1.0
    print("NOBS",nobs)
    df_agg_ent = df_split.agg({target_attribute_name : [entropy_of_list, lambda x:
len(x)/nobs] })[target_attribute_name]
    print([target_attribute_name])
    print(" Entropy List ",entropy_of_list)
    print("DFAGGENT",df_agg_ent)
    df_agg_ent.columns = ['Entropy', 'PropObservations']
    if trace: # helps understand what fxn is doing:
        print(df_agg_ent)

    # Calculate Information Gain:
    new_entropy = sum( df_agg_ent['Entropy'] * df_agg_ent['PropObservations'] )
    old_entropy = entropy_of_list(df[target_attribute_name])
    return old_entropy - new_entropy


print('Info-gain for Outlook is :'+str( information_gain(df_tennis, 'Outlook',
'PlayTennis')),"\n")
print('\n Info-gain for Humidity is: ' + str( information_gain(df_tennis,
'Humidity', 'PlayTennis')),"\n")
print('\n Info-gain for Wind is:' + str( information_gain(df_tennis, 'Wind',
'PlayTennis')),"\n")
print('\n Info-gain for Temperature is:' + str( information_gain(df_tennis,
'Temperature','PlayTennis')),"\n")
```

```
def id3(df, target_attribute_name, attribute_names, default_class=None):

    ## Tally target attribute:
    from collections import Counter
    cnt = Counter(x for x in df[target_attribute_name])# 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 = [information_gain(df, attr, target_attribute_name) for attr in
attribute_names] #
        index_of_max = gainz.index(max(gainz)) # Index of Best Attribute
        # Choose Best Attribute to split on:
        best_attr = attribute_names[index_of_max]

        # Create an empty tree, to be populated in a moment
        tree = {best_attr:{}} # Iniiate 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_name,
                        remaining_attribute_names,
                        default_class)
            tree[best_attr][attr_val] = subtree
        return tree
# Get Predictor Names (all but 'class')
attribute_names = list(df_tennis.columns)
print("List of Attributes:", attribute_names)
attribute_names.remove('PlayTennis') #Remove the class attribute
print("Predicting Attributes:", attribute_names)
# Run Algorithm:
from pprint import pprint
tree = id3(df_tennis,'PlayTennis',attribute_names)
print("\n\nThe Resultant Decision Tree is :\n")
#print(tree)
pprint(tree)
attribute = next(iter(tree))
print("Best Attribute :\n",attribute)
print("Tree Keys:\n",tree[attribute].keys())

def classify(instance, tree, default=None): # Instance of Play Tennis with
Predicted

    print("Instance:",instance)
    attribute = next(iter(tree)) # Outlook/Humidity/Wind
    print("Attribute:",attribute) # [Key /Attribute Both are same ]

    # print("Insance of Attribute :",instance[attribute],attribute)
```

```
    if instance[attribute] in tree[attribute].keys(): # Value of the attributs in
set of Tree keys
        result = tree[attribute][instance[attribute]]
        print("Instance Attribute:",instance[attribute],"TreeKeys
:",tree[attribute].keys())
        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_tennis['predicted'] = df_tennis.apply(classify, axis=1, args=(tree,'No') )
    # classify func allows for a default arg: when tree doesn't have answer for a
particular
    # combitation of attribute-values, we can use 'no' as the default guess

df_tennis[['PlayTennis', 'predicted']]
training_data = df_tennis.iloc[1:-4] # all but last four instances
test_data  = df_tennis.iloc[-4:] # just the last four
train_tree = id3(training_data, 'PlayTennis', attribute_names)

test_data['predicted2'] = test_data.apply(                                  # <----
test_data source
                                          classify,
                                          axis=1,
                                          args=(train_tree,'Yes') ) # <----
train_data tree


print ('\n\n Accuracy is : ' + str(
sum(test_data['PlayTennis']==test_data['predicted2'] ) / (1.0*len(test_data.index))
))
```

PROGRAM 5:

```
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) # maximum of X array longitudinally
y = y/100

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

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)

#Variable initialization
epoch=5000     #Setting training iterations
lr=0.1         #Setting learning rate
inputlayer_neurons = 2              #number of features in data set
hiddenlayer_neurons = 3      #number of hidden layers neurons
output_neurons = 1           #number of neurons at output layer

#weight and bias initialization
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))


#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
```

```
#Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)

#Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)

#how much hidden layer wts contributed to error
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad

# dotproduct of nextlayererror and currentlayerop
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)
```

PROGRAM 6:

```
#Write a program to implement the naïve Bayesian classifier for a sample training
data set
#stored as a .CSV file. Compute the accuracy of the classifier, considering few
test data sets.
print("\nNaive Bayes Classifier for concept learning problem")
import csv
import random
import math
import operator

def safe_div(x,y):
 if y == 0:
  return 0
 return x/y

# 1.Data Handling
# 1.1 Loading the Data from csv file of ConceptLearning dataset.
def loadCsv(filename):
  lines = csv.reader(open(filename))
  dataset = list(lines)
  for i in range(len(dataset)):
    dataset[i] = [float(x) for x in dataset[i]]
  return dataset


#1.2 Splitting the Data set into Training Set
def splitDataset(dataset, splitRatio):
  trainSize = int(len(dataset) * splitRatio)
  trainSet = []
  copy = list(dataset)
  i=0
  while len(trainSet) < trainSize:
  #index = random.randrange(len(copy))
    trainSet.append(copy.pop(i))
  return [trainSet, copy]
```

```
#2.Summarize Data
#The naive bayes model is comprised of a
#summary of the data in the training dataset.
#This summary is then used when making predictions.
#involves the mean and the standard deviation for each attribute, by class value

#2.1: Separate Data By Class
#Function to categorize the dataset in terms of classes
#The function assumes that the last attribute (-1) is the class value.
#The function returns a map of class values to lists of data instances.

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

#The mean is the central middle or central tendency of the data,
# and we will use it as the middle of our gaussian distribution
# when calculating probabilities

#2.2 : Calculate Mean
def mean(numbers):
  return safe_div(sum(numbers),float(len(numbers)))

#The standard deviation describes the variation of spread of the data,
#and we will use it to characterize the expected spread of each attribute
#in our Gaussian distribution when calculating probabilities.

#2.3 : Calculate Standard Deviation
def stdev(numbers):
  avg = mean(numbers)
  variance = safe_div(sum([pow(x-avg,2) for x in numbers]),float(len(numbers)-1))
  return math.sqrt(variance)

#2.4 : Summarize Dataset
#Summarize Data Set for a list of instances (for a class value)
#The zip function groups the values for each attribute across our data instances
#into their own lists so that we can compute the mean and standard deviation values
#for the attribute.

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

#2.5 : Summarize Attributes By Class
#We can pull it all together by first separating our training dataset into
#instances grouped by class.Then calculate the summaries for each attribute.

def summarizeByClass(dataset):
  separated = separateByClass(dataset)
  summaries = {}
  for classValue, instances in separated.items():
    summaries[classValue] = summarize(instances)
  print("Summarize Attributes By Class")
  print(summaries)
  print(" ")
  return summaries

#3.Make Prediction
#3.1 Calculate Probaility Density Function
def calculateProbability(x, mean, stdev):
```

```
  exponent = math.exp(-safe_div(math.pow(x-mean,2),(2*math.pow(stdev,2))))
  final = safe_div(1 , (math.sqrt(2*math.pi) * stdev)) * exponent
  return final

#3.2 Calculate Class Probabilities
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

#3.3 Prediction : look for the largest probability and return the associated class
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

#4.Make Predictions
# Function which return predictions for list of predictions
# For each instance
def getPredictions(summaries, testSet):
  predictions = []
  for i in range(len(testSet)):
    result = predict(summaries, testSet[i])
    predictions.append(result)
  return predictions

#5. Computing Accuracy
def getAccuracy(testSet, predictions):
  correct = 0
  for i in range(len(testSet)):
    if testSet[i][-1] == predictions[i]:
      correct += 1
  accuracy = safe_div(correct,float(len(testSet))) * 100.0
  return accuracy

def main():
  filename = 'diabetes2.csv'
  splitRatio = 0.9
  dataset = loadCsv(filename)
  trainingSet, testSet = splitDataset(dataset, splitRatio)
  print('Split {0} rows into'.format(len(dataset)))
  print('Number of Training data: ' + (repr(len(trainingSet))))
  print('Number of Test Data: ' + (repr(len(testSet))))
  print("\nThe values assumed for the concept learning attributes are\n")
  print("OUTLOOK=> Sunny=1 Overcast=2 Rain=3\nTEMPERATURE=> Hot=1 Mild=2
Cool=3\nHUMIDITY=> High=1 Normal=2\nWIND=> Weak=1 Strong=2")
  print("TARGET CONCEPT:PLAY TENNIS=> Yes=10 No=5")
  print("\nThe Training set are:")
  for x in trainingSet:
    print(x)
    print("\nThe Test data set are:")
  for x in testSet:
    print(x)
  print("\n")

# prepare model
  summaries = summarizeByClass(trainingSet)
```

```
# test model
  predictions = getPredictions(summaries, testSet)
  actual = []
  for i in range(len(testSet)):
   vector = testSet[i]
  actual.append(vector[-1])

# Since there are five attribute values, each attribute constitutes to 20%
accuracy. So if all attributes
#match with predictions then 100% accuracy
  print('Actual values: {0}%'.format(actual))
  print('Predictions: {0}%'.format(predictions))
  accuracy = getAccuracy(testSet, predictions)
  print('Accuracy: {0}%'.format(accuracy))

main()
```

PROGRAM 7:

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

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


l1 = [0,1,2]
def rename(s):
       l2 = []
       for i in s:
              if i not in l2:
                     l2.append(i)

       for i in range(len(s)):
              pos = l2.index(s[i])
              s[i] = l1[pos]

       return s

# import some data to play with
iris = datasets.load_iris()

print("\n IRIS DATA :",iris.data);
print("\n IRIS FEATURES :\n",iris.feature_names)
print("\n IRIS TARGET  :\n",iris.target)
print("\n IRIS TARGET NAMES:\n",iris.target_names)


# Store the inputs as a Pandas Dataframe and set the column names
X = pd.DataFrame(iris.data)

#print(X)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

#print(X.columns) #print("X:",x)
#print("Y:",y)
```

```
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

# Set the size of the plot
plt.figure(figsize=(14,7))

# Create a colormap
colormap = np.array(['red', 'lime', 'black'])

# Plot Sepal
plt.subplot(1,2,1)
plt.scatter(X.Sepal_Length,X.Sepal_Width, c=colormap[y.Targets], s=40)
plt.title('Sepal')

plt.subplot(1,2,2)
plt.scatter(X.Petal_Length,X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Petal')
plt.show()

print("Actual Target is:\n", iris.target)

# K Means Cluster
model = KMeans(n_clusters=3)
model.fit(X)

# Set the size of the plot
plt.figure(figsize=(14,7))

# Create a colormap
colormap = np.array(['red', 'lime', 'black'])

# Plot the Original Classifications
plt.subplot(1,2,1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Classification')

# Plot the Models Classifications
plt.subplot(1,2,2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K Mean Classification')
plt.show()

km = rename(model.labels_)
print("\nWhat KMeans thought: \n", km)
print("Accuracy of KMeans is ",sm.accuracy_score(y, km))
print("Confusion Matrix for KMeans is \n",sm.confusion_matrix(y, km))

#The GaussianMixture scikit-learn class can be used to model this problem
#and estimate the parameters of the distributions using the expectation-
maximization algorithm.

from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
print("\n",xs.sample(5))

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)

y_cluster_gmm = gmm.predict(xs)

plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')
```

```
plt.show()

em = rename(y_cluster_gmm)
print("\nWhat EM thought: \n", em)
print("Accuracy of EM is ",sm.accuracy_score(y, em))
print("Confusion Matrix for EM is \n", sm.confusion_matrix(y, em))
```

PROGRAM 8:

```
# Python program to demonstrate # KNN classification algorithm # on IRISdataset
#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.

#import the dataset and library files
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.model_selection import train_test_split

iris_dataset=load_iris()

#display the iris dataset
print("\n IRIS FEATURES \ TARGET NAMES: \n ", iris_dataset.target_names)
for i in range(len(iris_dataset.target_names)):
    print("\n[{0}]:[{1}]".format(i,iris_dataset.target_names[i]))

print("\n IRIS DATA :\n",iris_dataset["data"])

#split the data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(iris_dataset["data"],
iris_dataset["target"], random_state=0)

print("\n Target :\n",iris_dataset["target"])
print("\n X TRAIN \n", X_train)
print("\n X TEST \n", X_test)
print("\n Y TRAIN \n", y_train)
print("\n Y TEST \n", y_test)

#train and fit the model
kn = KNeighborsClassifier(n_neighbors=5)
kn.fit(X_train, y_train)

#predicting from model
x_new = np.array([[5, 2.9, 1, 0.2]])
print("\n XNEW \n",x_new)
prediction = kn.predict(x_new)
print("\n Predicted target value: {}\n".format(prediction))
print("\n Predicted feature name:
{}\n".format(iris_dataset["target_names"][prediction]))

i=1
x= X_test[i]
x_new = np.array([x])
print("\n XNEW \n",x_new)

for i in range(len(X_test)):
  x = X_test[i]
  x_new = np.array([x])
  prediction = kn.predict(x_new)
  print("\n Actual : {0} {1}, Predicted
:{2}{3}".format(y_test[i],iris_dataset["target_names"][y_test[i]],prediction,iris_d
ataset["target_names"][ prediction]))
```

```
print("\n TEST SCORE[ACCURACY]: {:.2f}\n".format(kn.score(X_test, y_test)))
```

PROGRAM 9:

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

def local_regression(x0, X, Y, tau):
    x0 = [1, x0]
    X = [[1, i] for i in X]
    X = np.asarray(X)
    xw = (X.T) * np.exp(np.sum((X - x0) ** 2, axis=1) / (-2 * tau))
    beta = np.linalg.pinv(xw @ X) @ xw @ Y @ x0
    return beta

def draw(tau):
    prediction = [local_regression(x0, X, Y, tau) for x0 in domain]
    plt.plot(X, Y, 'o', color='black')
    plt.plot(domain, prediction, color='red')
    plt.show()

X = np.linspace(-3, 3, num=1000)
domain = X
Y = np.log(np.abs(X ** 2 - 1) + .5)

draw(10)
draw(0.1)
draw(0.01)
draw(0.001)
```