PRACTICAL NO 1

AIM:- Breadth First Search & Iterative Depth First Search

 Implement the Breadth First Search algorithm to solve a given problem.

INTRODUCTION:-

Breadth first search algorithm is an algorithm used to find the shortest path from one node to another in a graph. It begins by searching through the nodes closest to the starting node, and then expands outward to the more distant nodes.
Breadth-First Search (BFS) is a fundamental algorithm in artificial intelligence and computer science that helps solve problems such as finding the shortest path, checking graph connectivity, and traversing all the vertices of a graph.
The main advantage of the breadth first search algorithm is that it is simple to implement and can be easily understood. Moreover, this algorithm can be used to find the shortest path between two nodes in a graph.

Here's how BFS works:

- Select a starting node: Choose a random node to start from, also known as the root or source node.

- Traverse the graph layer-wise: Visit all the nodes and their children nodes in the order they appear in the queue. The queue uses a First In First Out (FIFO) principle.

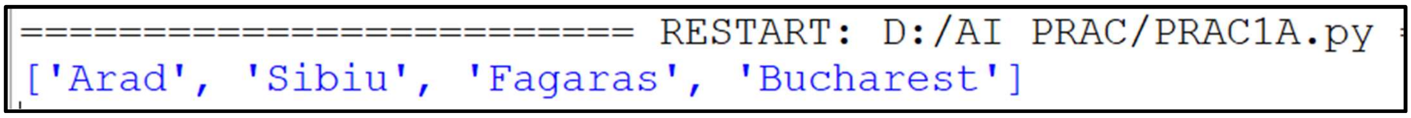
- Expand outward: Start with the nodes closest to the starting node and then move on to more distant nodes.

BFS is often used in tree traversal and network routing algorithms.

CODE:-

```
class first:
    def __init__(self,graph,start,goal):
        self.start = start
        self.goal = goal
        self.graph=graph
    def bfs_shortest_path(self):
        explored = []
        queue = [[self.start]]
        if self.start == self.goal:
            return "That was easy! Start = goal"
        while queue:
            path = queue.pop(0)
            node = path[-1]
            neighbours = self.graph[node]
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                if neighbour == self.goal:
                    return new_path
            explored.append(node)
        return "path doesn't exist"
graph = {'Arad': ['Zerind', 'Sibiu', 'Timisoara'],
         'Bucharest': ['Urziceni','Pitesti', 'Giurgiu','Fagaras'],
         'Craiova': ['Dobreta', 'Rimnicu Vilcea', 'Pitesti'],
         'Dobreta': ['Mehadia'],
         'Eforie': ['Hirsova'],
         'Iasai': ['Vaslui','Neamt'],
         'Lugoj': ['Timisoara','Mehadia'],
         'Oradea': ['Zerind','Sibiu'],
         'Pitesti': ['Rimnicu Vilcea'],
         'Urziceni': ['Vaslui'],
         'Zerind': ['Oradea','Arad'],
         'Sibiu': ['Oradea','Arad','Rimnicu Vilcea','Fagaras'],
         'Timisoara': ['Arad','Lugoj'],
         'Mehadia': ['Lugoj','Dobreta'],
         'Rimnicu Vilcea': ['Sibiu','Pitesti','Craiova'],
         'Fagaras': ['Sibiu','Bucharest'],
         'Giurgiu': ['Bucharest'],
         'Vaslui': ['Urziceni','Iasai'],
         'Neamt': ['Iasai']
         }
f=first(graph,'Arad','Bucharest')
print(f.bfs_shortest_path())
```

OUTPUT:-



```
========================= RESTART: D:/AI PRAC/PRAC1A.py
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
```

 Implement the Iterative Depth First Search algorithm to solve the same problem.

INTRODUCTION:-

Depth-first search is a traversing algorithm used in tree and graph-like data structures. It generally starts by exploring the deepest node in the frontier. Starting at the root node, the algorithm proceeds to search to the deepest level of the search tree until nodes with no successors are reached. Suppose the node with unexpanded successors is encountered then the search backtracks to the next deepest node to explore alternative paths. Depth-first search (DFS) explores a graph by selecting a path and traversing it as deeply as possible before backtracking.

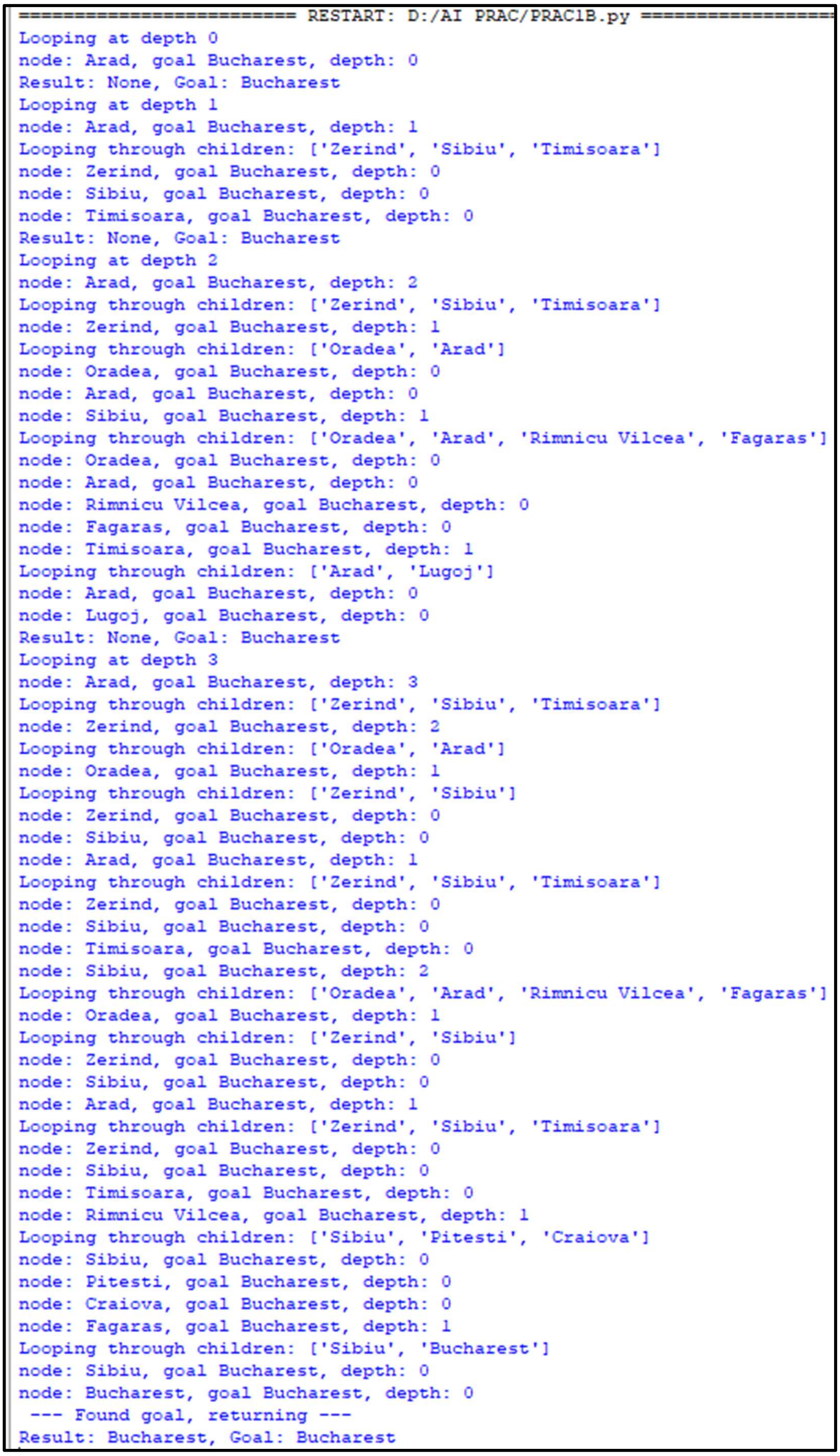
- Originally it starts at the root node, then it expands all of its one branch until it reaches a dead end, then backtracks to the most recent unexplored node, repeating until all nodes are visited or a specific condition is met. (As shown in the above image, starting from node A, DFS explores its successor B, then proceeds to its descendants until reaching a dead end at node D. It then backtracks to node B and explores its remaining successors).

- This systematic exploration continues until all nodes are visited or the search terminates. (In our case after exploring all the nodes of B. DFS explores the right side node i.e C then F and and then G. After exploring the node G. All the nodes are visited. It will terminate.

CODE:-

```
graph = {'Arad': ['Zerind', 'Sibiu', 'Timisoara'],
        'Bucharest': ['Urziceni','Pitesti', 'Giurgiu','Fagaras'],
        'Craiova': ['Dobreta', 'Rimnicu Vilcea', 'Pitesti'],
        'Dobreta': ['Mehadia'],
        'Eforie': ['Hirsova'],
        'Iasai': ['Vaslui','Neamt'],
        'Lugoj': ['Timisoara','Mehadia'],
        'Oradea': ['Zerind','Sibiu'],
        'Pitesti': ['Rimnicu Vilcea'],
        'Urziceni': ['Vaslui'],
        'Zerind': ['Oradea','Arad'],
        'Sibiu': ['Oradea','Arad','Rimnicu Vilcea','Fagaras'],
        'Timisoara': ['Arad','Lugoj'],
        'Mehadia': ['Lugoj','Dobreta'],
        'Rimnicu Vilcea': ['Sibiu','Pitesti','Craiova'],
        'Fagaras': ['Sibiu','Bucharest'],
        'Giurgiu': ['Bucharest'],
        'Vaslui': ['Urziceni','Iasai'],
        'Neamt': ['Iasai']
        }

def IDDFS(root, goal):
   depth = 0
   while True:
      print ("Looping at depth %i " % (depth))
      result = DLS(root, goal, depth)
      print ("Result: %s, Goal: %s" % (result, goal))
      if result == goal:
         return result
      depth = depth +1

def DLS(node, goal, depth):
   print ("node: %s, goal %s, depth: %i" % (node, goal, depth))
   if depth == 0 and node == goal:
      print (" --- Found goal, returning --- ")
      return node
   elif depth > 0:
      print ("Looping through children: %s" % (graph.get(node, [])))
      for child in graph.get(node, []):
         if goal == DLS(child, goal, depth-1):
            return goal


IDDFS('Arad', 'Bucharest')
```

OUTPUT:-



```
========================= RESTART: D:/AI PRAC/PRAC1B.py ==================
Looping at depth 0
node: Arad, goal Bucharest, depth: 0
Result: None, Goal: Bucharest
Looping at depth 1
node: Arad, goal Bucharest, depth: 1
Looping through children: ['Zerind', 'Sibiu', 'Timisoara']
node: Zerind, goal Bucharest, depth: 0
node: Sibiu, goal Bucharest, depth: 0
node: Timisoara, goal Bucharest, depth: 0
Result: None, Goal: Bucharest
Looping at depth 2
node: Arad, goal Bucharest, depth: 2
Looping through children: ['Zerind', 'Sibiu', 'Timisoara']
node: Zerind, goal Bucharest, depth: 1
Looping through children: ['Oradea', 'Arad']
node: Oradea, goal Bucharest, depth: 0
node: Arad, goal Bucharest, depth: 0
node: Sibiu, goal Bucharest, depth: 1
Looping through children: ['Oradea', 'Arad', 'Rimnicu Vilcea', 'Fagaras']
node: Oradea, goal Bucharest, depth: 0
node: Arad, goal Bucharest, depth: 0
node: Rimnicu Vilcea, goal Bucharest, depth: 0
node: Fagaras, goal Bucharest, depth: 0
node: Timisoara, goal Bucharest, depth: 1
Looping through children: ['Arad', 'Lugoj']
node: Arad, goal Bucharest, depth: 0
node: Lugoj, goal Bucharest, depth: 0
Result: None, Goal: Bucharest
Looping at depth 3
node: Arad, goal Bucharest, depth: 3
Looping through children: ['Zerind', 'Sibiu', 'Timisoara']
node: Zerind, goal Bucharest, depth: 2
Looping through children: ['Oradea', 'Arad']
node: Oradea, goal Bucharest, depth: 1
Looping through children: ['Zerind', 'Sibiu']
node: Zerind, goal Bucharest, depth: 0
node: Sibiu, goal Bucharest, depth: 0
node: Arad, goal Bucharest, depth: 1
Looping through children: ['Zerind', 'Sibiu', 'Timisoara']
node: Zerind, goal Bucharest, depth: 0
node: Sibiu, goal Bucharest, depth: 0
node: Timisoara, goal Bucharest, depth: 0
node: Sibiu, goal Bucharest, depth: 2
Looping through children: ['Oradea', 'Arad', 'Rimnicu Vilcea', 'Fagaras']
node: Oradea, goal Bucharest, depth: 1
Looping through children: ['Zerind', 'Sibiu']
node: Zerind, goal Bucharest, depth: 0
node: Sibiu, goal Bucharest, depth: 0
node: Arad, goal Bucharest, depth: 1
Looping through children: ['Zerind', 'Sibiu', 'Timisoara']
node: Zerind, goal Bucharest, depth: 0
node: Sibiu, goal Bucharest, depth: 0
node: Timisoara, goal Bucharest, depth: 0
node: Rimnicu Vilcea, goal Bucharest, depth: 1
Looping through children: ['Sibiu', 'Pitesti', 'Craiova']
node: Sibiu, goal Bucharest, depth: 0
node: Pitesti, goal Bucharest, depth: 0
node: Craiova, goal Bucharest, depth: 0
node: Fagaras, goal Bucharest, depth: 1
Looping through children: ['Sibiu', 'Bucharest']
node: Sibiu, goal Bucharest, depth: 0
node: Bucharest, goal Bucharest, depth: 0
 --- Found goal, returning ---
Result: Bucharest, Goal: Bucharest
```

PRACTICAL NO 2

AIM:- Implement the A* Search algorithm for solving a pathfinding problem.

INTRODUCTION:-

It is a search algorithm used to find the shortest path between an initial and a final point. It is often used for map traversal to find the shortest path. A* was initially designed as a graph traversal problem to help build a robot that can find its own course. It remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

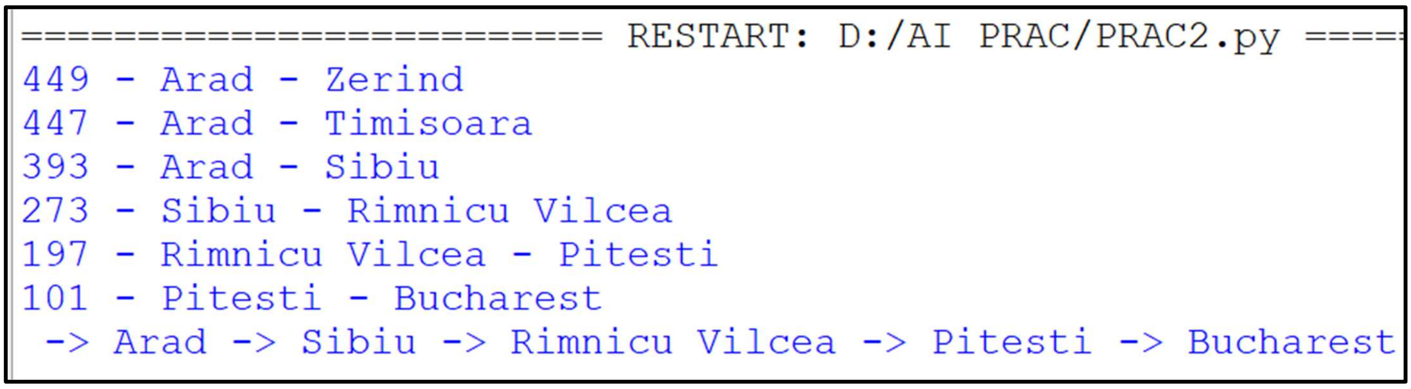
Another aspect that makes A* so powerful is its implementation of weighted graphs. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

A major drawback of the algorithm is its space and time complexity. It takes a large amount of space to store all possible paths and a lot of time to find them.

CODE:-

```
graph = {'Arad': ['Zerind', 'Timisoara', 'Sibiu'],
        'Bucharest': ['Urziceni','Pitesti', 'Giurgiu','Fagaras'],
        'Craiova': ['Dobreta', 'Rimnicu Vilcea', 'Pitesti'],
        'Dobreta': ['Mehadia'],
        'Eforie': ['Hirsova'],
        'Iasai': ['Vaslui','Neamt'],
        'Lugoj': ['Timisoara','Mehadia'],
        'Oradea': ['Zerind','Sibiu'],
        'Pitesti': ['Rimnicu Vilcea','Bucharest','Craiova'],
        'Urziceni': ['Vaslui'],
        'Zerind': ['Oradea','Arad'],
        'Sibiu': ['Oradea','Arad','Rimnicu Vilcea','Fagaras'],
        'Timisoara': ['Arad','Lugoj'],
        'Mehadia': ['Lugoj','Dobreta'],
        'Rimnicu Vilcea': ['Sibiu','Pitesti','Craiova'],
        'Fagaras': ['Sibiu','Bucharest'],
        'Giurgiu': ['Bucharest'],
        'Vaslui': ['Urziceni','Iasai'],
        'Neamt': ['Iasai']}
pc = {('Arad','Zerind'):75,
     ('Arad','Timisoara'):118,
     ('Arad','Sibiu'):140,
     ('Zerind','Oradea'):71,
     ('Zerind','Arad'):75,
     ('Timisoara','Arad'):118,
     ('Timisoara','Lugoj'):111,
     ('Sibiu','Arad'):140,
     ('Sibiu','Rimnicu Vilcea'):80,
     ('Sibiu','Fagaras'):99,
     ('Sibiu','Oradea'):151,
     ('Oradea','Zerind'):71,
     ('Oradea','Sibiu'):151,
     ('Lugoj','Timisoara'):111,
     ('Lugoj','Mehadia'):70,
     ('Rimnicu Vilcea','Sibiu'):80,
     ('Rimnicu Vilcea','Pitesti'):97,
     ('Rimnicu Vilcea','Craiova'):146,
     ('Fagaras','Sibiu'):99,
     ('Fagaras','Bucharest'):211,
     ('Mehadia','Lugoj'):70,
     ('Mehadia','Dobreta'):75,
     ('Pitesti','Rimnicu Vilcea'):97,
     ('Pitesti','Bucharest'):101,
     ('Pitesti','Craiova'):138,
     ('Craiova','Rimnicu Vilcea'):146,
     ('Craiova','Dobreta'):120,
     ('Craiova','Pitesti'):138,
     ('Bucharest','Fagaras'):211,
     ('Bucharest','Bucharest'):0,
     ('Bucharest','Pitesti'):101,
     ('Bucharest','Giurgiu'):90,
     ('Bucharest','Urziceni'):85,
     }
locs={'Arad': 366,
     'Bucharest': 0,
     'Craiova': 160,
     'Dobreta': 242,
     'Eforie': 161,
     'Iasai': 226,
     'Lugoj': 244,
     'Oradea': 380,
     'Pitesti': 100,
     'Urziceni': 80,
     'Zerind': 374,
     'Sibiu': 253,
     'Timisoara': 329,
     'Mehadia': 241,
     'Rimnicu Vilcea': 193,
     'Fagaras':176,
     'Giurgiu': 77,
     'Vaslui': 199,
     'Neamt': 234
     }
def DFS(g, v, goal, explored, path_so_far,m):
     explored.add(v)
     node=[]
     if v == goal:
         return path_so_far +" -> "+ v
     for w in g[v]:
         if w not in explored:
             f=locs.get(w)+pc.get((v,w))
             if m>f:
                 m=f
                 print("%i - %s - %s " %(m,v,w))
                 node=w

     p = DFS(g, node, goal, explored, path_so_far +" -> "+ v,m)
     if p:
         return p
     return ""

print(DFS(graph, 'Arad', 'Bucharest', set(), "",1000))
```

OUTPUT:-



```
========================= RESTART: D:/AI PRAC/PRAC2.py ====
449 - Arad - Zerind
447 - Arad - Timisoara
393 - Arad - Sibiu
273 - Sibiu - Rimnicu Vilcea
197 - Rimnicu Vilcea - Pitesti
101 - Pitesti - Bucharest
 -> Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
```

PRACTICAL NO 3

AIM:- Decision Tree Learning

 Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.

 Evaluate the accuracy and effectiveness of the decision tree on test data.

 Visualize and interpret the generated decision tree.

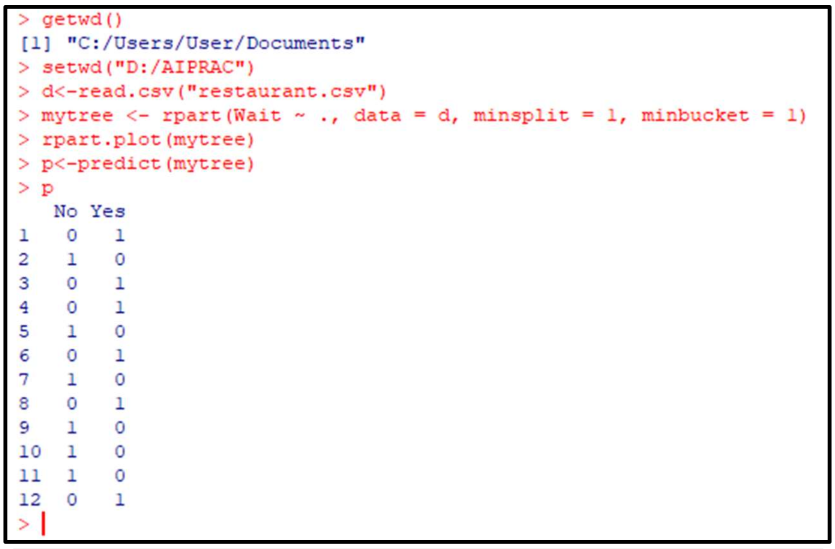
INTRODUCTION:-

- Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome.
- In a Decision tree, there are two nodes, which are the Decision Node and Leaf Node. Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
- The decisions or the test are performed on the basis of features of the given dataset.
- *It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.*
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the CART algorithm, which stands for Classification and Regression Tree algorithm.
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.
- Below diagram explains the general structure of a decision tree:

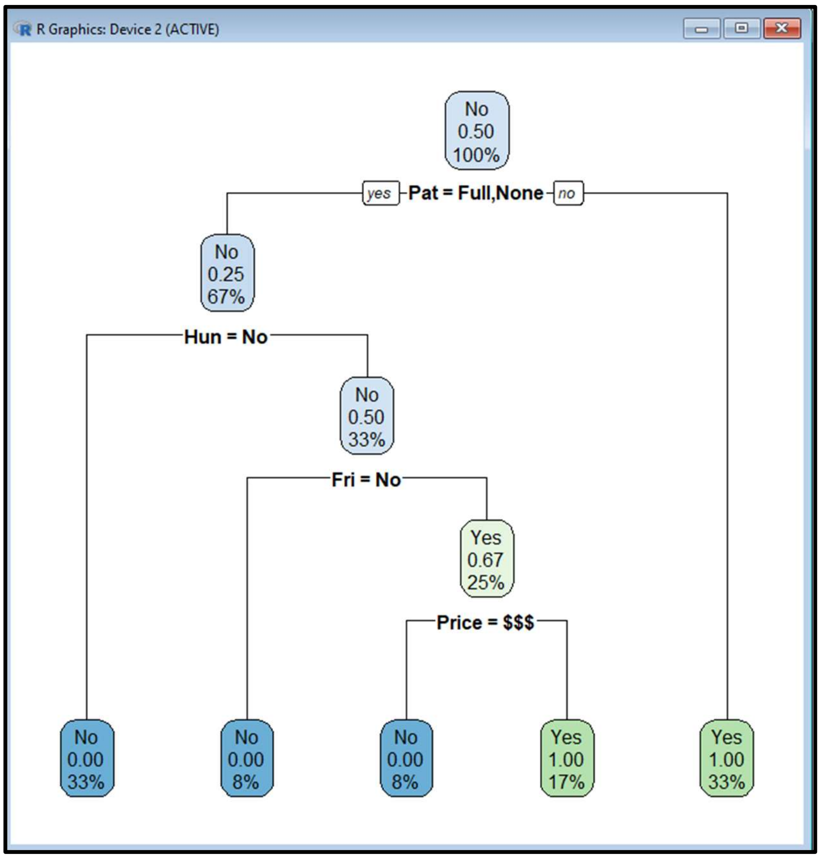
CODE:-

```
install.packages("rpart")
library(rpart)
install.packages("rpart.plot")
library(rpart.plot)
getwd()
setwd("D:/AIPRAC")
d<-read.csv("restaurant.csv")
mytree <- rpart(Wait ~ ., data = d, minsplit = 1, minbucket = 1)
rpart.plot(mytree)
p<-predict(mytree)
p
```

OUTPUT:-



```
> getwd()
[1] "C:/Users/User/Documents"
> setwd("D:/AIPRAC")
> d<-read.csv("restaurant.csv")
> mytree <- rpart(Wait ~ ., data = d, minsplit = 1, minbucket = 1)
> rpart.plot(mytree)
> p<-predict(mytree)
> p
   No Yes
1   0   1
2   1   0
3   0   1
4   0   1
5   1   0
6   0   1
7   1   0
8   0   1
9   1   0
10  1   0
11  1   0
12  0   1
>
```



PRACTICAL NO 4

AIM:- Feed Forward Backpropagation Neural Network
 Implement the Feed Forward Backpropagation algorithm to train a neural network.
 Use a given dataset to train the neural network for a specific task.
 Evaluate the performance of the trained network on test data

INTRODUCTION:-

A Feedforward Neural Network (FNN) is a type of artificial neural network where connections between the nodes do not form cycles. This characteristic differentiates it from recurrent neural networks (RNNs). The network consists of an input layer, one or more hidden layers, and an output layer. Information flows in one direction—from input to output—hence the name "feedforward."

Structure of a Feedforward Neural Network

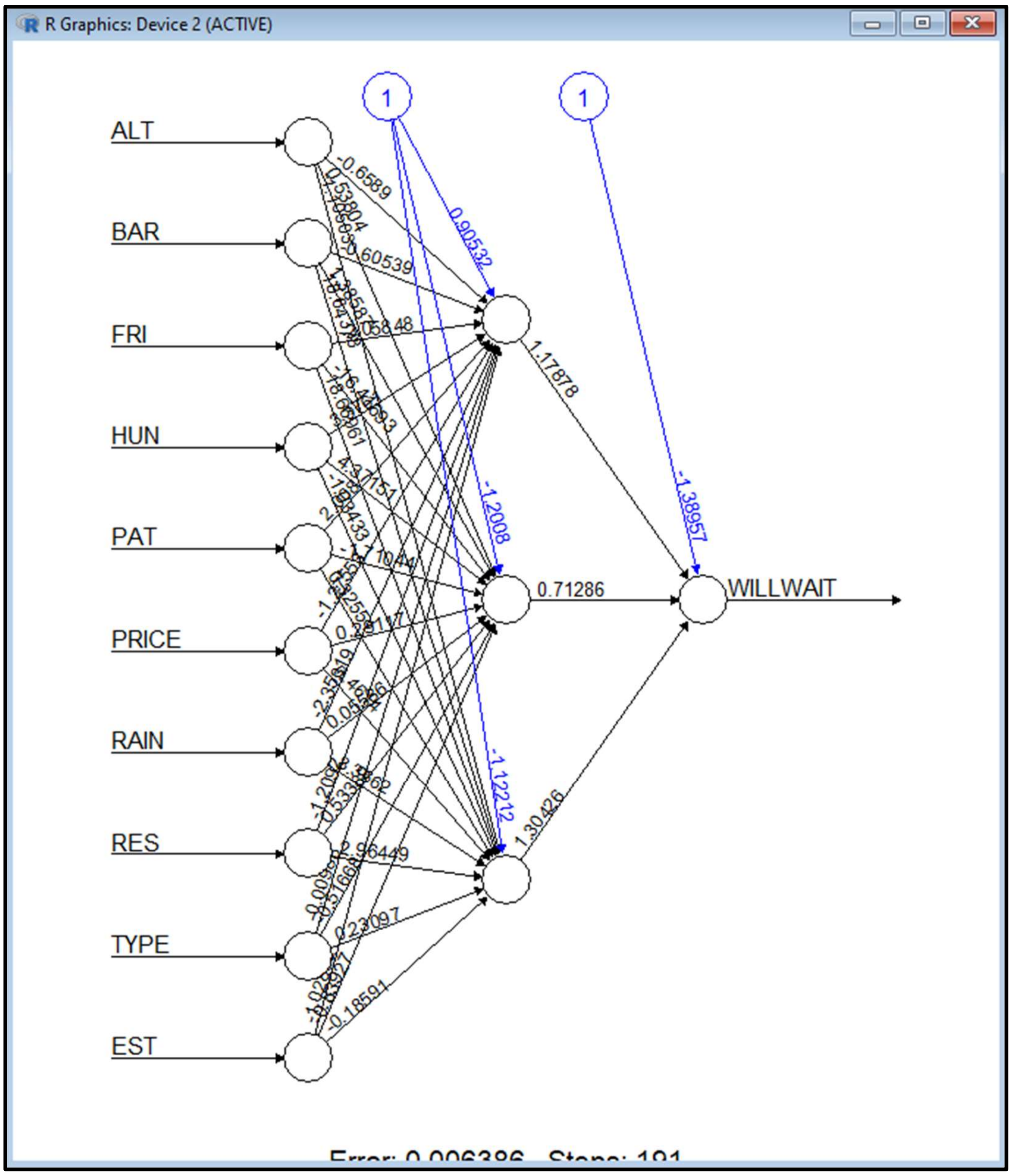
1. Input Layer: The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.
2. Hidden Layers: One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function.
3. Output Layer: The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.

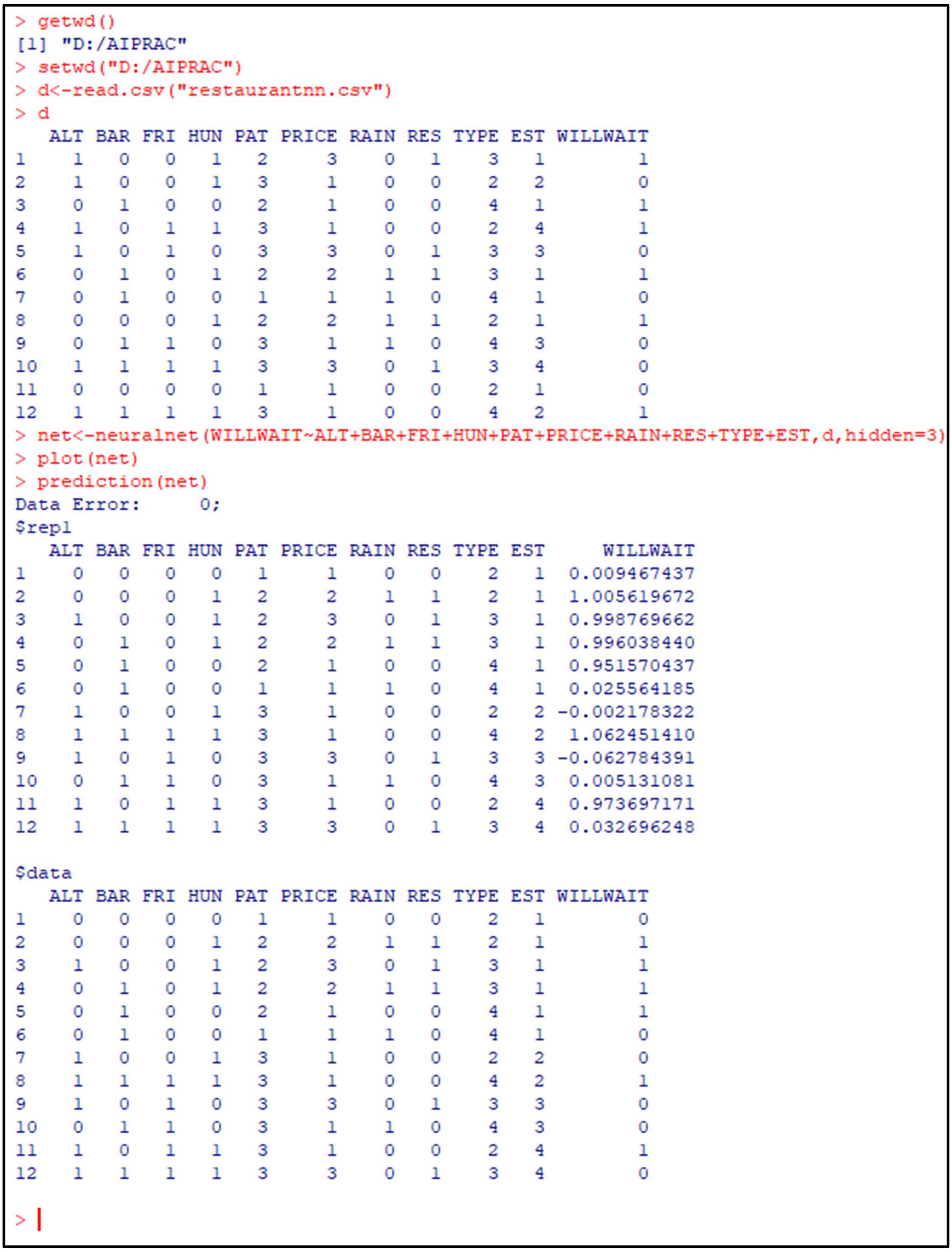
Each connection between neurons in these layers has an associated weight that is adjusted during the training process to minimize the error in predictions.

CODE:-

```
install.packages("neuralnet")
library(neuralnet)
getwd()
setwd("D:/AIPRAC")
d<-read.csv("restaurantnn.csv")
d
net<-
neuralnet(WILLWAIT~ALT+BAR+FRI+HUN+PAT+PRICE+RAIN+RES+TYPE+EST,d,hidden=3
)
plot(net)
prediction(net)
```

OUTPUT:-





```
> getwd()
[1] "D:/AIPRAC"
> setwd("D:/AIPRAC")
> d<-read.csv("restaurantnn.csv")
> d
   ALT BAR FRI HUN PAT PRICE RAIN RES TYPE EST WILLWAIT
1    1   0   0   1   2     3    0   1    3   1        1
2    1   0   0   1   3     1    0   0    2   2        0
3    0   1   0   0   2     1    0   0    4   1        1
4    1   0   1   1   3     1    0   0    2   4        1
5    1   0   1   0   3     3    0   1    3   3        0
6    0   1   0   1   2     2    1   1    3   1        1
7    0   1   0   0   1     1    1   0    4   1        0
8    0   0   0   1   2     2    1   1    2   1        1
9    0   1   1   0   3     1    1   0    4   3        0
10   1   1   1   1   3     3    0   1    3   4        0
11   0   0   0   0   1     1    0   0    2   1        0
12   1   1   1   1   3     1    0   0    4   2        1
> net<-neuralnet(WILLWAIT~ALT+BAR+FRI+HUN+PAT+PRICE+RAIN+RES+TYPE+EST,d,hidden=3)
> plot(net)
> prediction(net)
Data Error:     0;
$rep1
   ALT BAR FRI HUN PAT PRICE RAIN RES TYPE EST     WILLWAIT
1    0   0   0   0   1     1    0   0    2   1  0.009467437
2    0   0   0   1   2     2    1   1    2   1  1.005619672
3    1   0   0   1   2     3    0   1    3   1  0.998769662
4    0   1   0   1   2     2    1   1    3   1  0.996038440
5    0   1   0   0   2     1    0   0    4   1  0.951570437
6    0   1   0   0   1     1    1   0    4   1  0.025564185
7    1   0   0   1   3     1    0   0    2   2 -0.002178322
8    1   1   1   1   3     1    0   0    4   2  1.062451410
9    1   0   1   0   3     3    0   1    3   3 -0.062784391
10   0   1   1   0   3     1    1   0    4   3  0.005131081
11   1   0   1   1   3     1    0   0    2   4  0.973697171
12   1   1   1   1   3     3    0   1    3   4  0.032696248

$data
   ALT BAR FRI HUN PAT PRICE RAIN RES TYPE EST WILLWAIT
1    0   0   0   0   1     1    0   0    2   1        0
2    0   0   0   1   2     2    1   1    2   1        1
3    1   0   0   1   2     3    0   1    3   1        1
4    0   1   0   1   2     2    1   1    3   1        1
5    0   1   0   0   2     1    0   0    4   1        1
6    0   1   0   0   1     1    1   0    4   1        0
7    1   0   0   1   3     1    0   0    2   2        0
8    1   1   1   1   3     1    0   0    4   2        1
9    1   0   1   0   3     3    0   1    3   3        0
10   0   1   1   0   3     1    1   0    4   3        0
11   1   0   1   1   3     1    0   0    2   4        1
12   1   1   1   1   3     3    0   1    3   4        0

> |
```

PRACTICAL NO 5

AIM:- Adaboost Ensemble Learning

 Implement the Adaboost algorithm to create an ensemble of weak classifiers.

 Train the ensemble model on a given dataset and evaluate its performance.

 Compare the results with individual weak classifiers.

INTRODUCTION:-

AdaBoost short for Adaptive Boosting is an ensemble learning used in machine learning for classification and regression problems. The main idea behind AdaBoost is to iteratively train the weak classifier on the training dataset with each successive classifier giving more weightage to the data points that are misclassified. The final AdaBoost model is decided by combining all the weak classifier that has been used for training with the weightage given to the models according to their accuracies. The weak model which has the highest accuracy is given the highest weightage while the model which has the lowest accuracy is given a lower weightage. To gain deeper information about AdaBoost, it's critical to be acquainted with some key principles associated with the algorithm:

1. Weak Learners

Weak novices are the individual fashions that make up the ensemble. These are generally fashions with accuracy barely higher than random hazards. In the context of AdaBoost, weak beginners are trained sequentially, with each new model focusing on the instances that preceding models determined difficult to classify.

2. Strong Classifier

The strong classifier, additionally known as the ensemble, is the final version created by combining the predictions of all weak first-year students. It has the collective know-how of all of the fashions and is capable of making correct predictions.

3. Weighted Voting

In AdaBoost, every susceptible learner contributes to the very last prediction with a weight-based totally on its Performance. This weighted vote-casting machine ensures that the greater correct fashions have a greater say in the final choice.

4. Error Rate

The error rate is the degree of ways a vulnerable learner plays on the schooling statistics. It is used to calculate the load assigned to each vulnerable learner. Models with lower error fees are given higher weights.

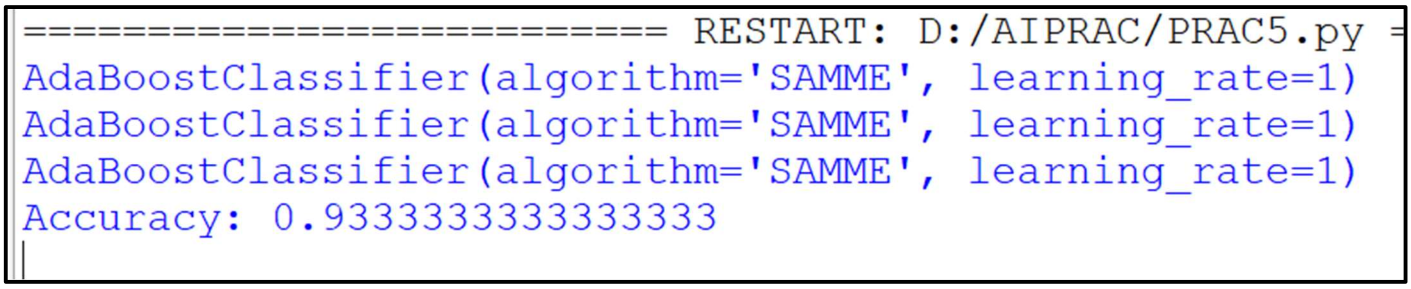
5. Iterations

The range of iterations or rounds in AdaBoost is a hyperparameter that determines what number of susceptible newbies are educated. Increasing the range of iterations may additionally result in a more complex ensemble; however, it can also increase the risk of overfitting.

CODE:-

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import metrics

iris = datasets.load_iris()

X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

abc = AdaBoostClassifier(algorithm='SAMME',n_estimators=50,learning_rate=1)
print(abc)

model = abc.fit(X_train, y_train)
print(model)

y_pred = model.predict(X_test)
print(model)

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

OUTPUT:-



```
========================== RESTART: D:/AIPRAC/PRAC5.py =
AdaBoostClassifier(algorithm='SAMME', learning_rate=1)
AdaBoostClassifier(algorithm='SAMME', learning_rate=1)
AdaBoostClassifier(algorithm='SAMME', learning_rate=1)
Accuracy: 0.9333333333333333
```

PRACTICAL NO 6

AIM:- Naive Bayes' Classifier

 Implement the Naive Bayes' algorithm for classification.
 Train a Naive Bayes' model using a given dataset and calculate class probabilities.
 Evaluate the accuracy of the model on test data and analyze the results.

INTRODUCTION:-

- Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
- It is mainly used in *text classification* that includes a high-dimensional training dataset.
- Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
- Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

Why is it called Naïve Bayes?

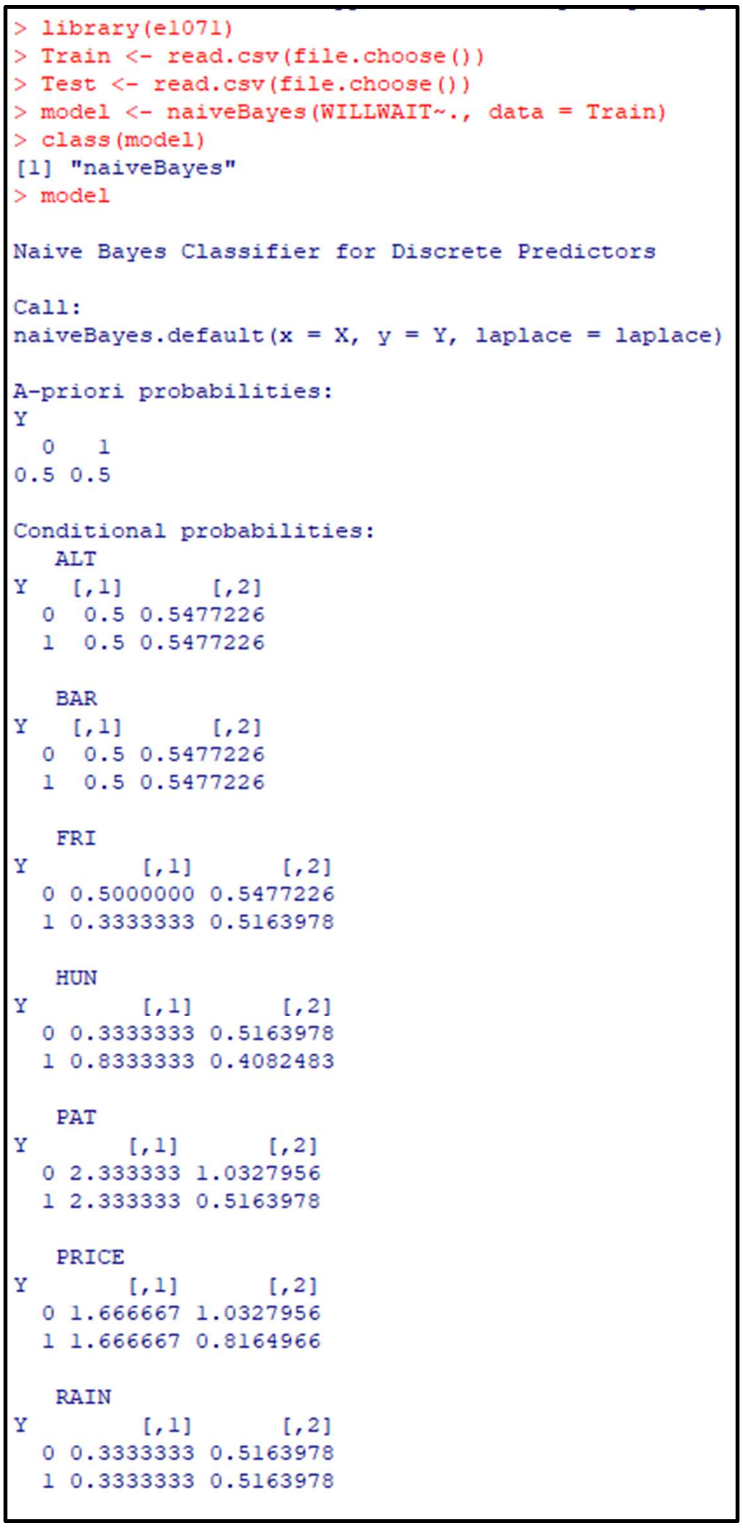
The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

- Naïve: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
- Bayes: It is called Bayes because it depends on the principle of Bayes' Theorem.

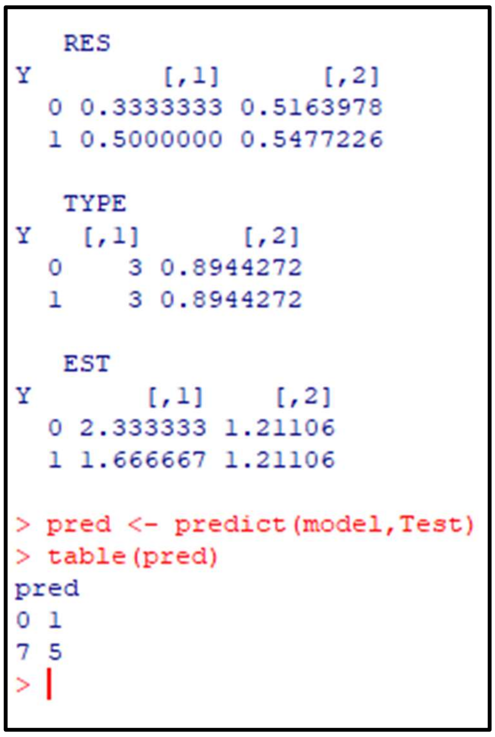
CODE:-

```
install.packages("e1071")
library(e1071)
Train <- read.csv(file.choose())
Test <- read.csv(file.choose())
model <- naiveBayes(WILLWAIT~., data = Train)
class(model)
model
pred <- predict(model,Test)
table(pred)
```

OUTPUT:-



```
> library(e1071)
> Train <- read.csv(file.choose())
> Test <- read.csv(file.choose())
> model <- naiveBayes(WILLWAIT~., data = Train)
> class(model)
[1] "naiveBayes"
> model

Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = X, y = Y, laplace = laplace)

A-priori probabilities:
Y
  0   1 
0.5 0.5 

Conditional probabilities:
   ALT
Y   [,1]      [,2]
  0  0.5 0.5477226
  1  0.5 0.5477226

   BAR
Y   [,1]      [,2]
  0  0.5 0.5477226
  1  0.5 0.5477226

   FRI
Y        [,1]      [,2]
  0 0.5000000 0.5477226
  1 0.3333333 0.5163978

   HUN
Y        [,1]      [,2]
  0 0.3333333 0.5163978
  1 0.8333333 0.4082483

   PAT
Y       [,1]      [,2]
  0 2.333333 1.0327956
  1 2.333333 0.5163978

   PRICE
Y       [,1]      [,2]
  0 1.666667 1.0327956
  1 1.666667 0.8164966

   RAIN
Y        [,1]      [,2]
  0 0.3333333 0.5163978
  1 0.3333333 0.5163978
```



```
   RES
Y          [,1]      [,2]
  0 0.3333333 0.5163978
  1 0.5000000 0.5477226

   TYPE
Y   [,1]      [,2]
  0    3 0.8944272
  1    3 0.8944272

   EST
Y         [,1]    [,2]
  0 2.333333 1.21106
  1 1.666667 1.21106

> pred <- predict(model,Test)
> table(pred)
pred
0 1
7 5
> |
```

PRACTICAL NO 7

AIM:- K-Nearest Neighbors (K-NN)

 Implement the K-NN algorithm for classification or regression.
 Apply the K-NN algorithm to a given dataset and predict the class or value for test data.
 Evaluate the accuracy or error of the predictions and analyze the results.

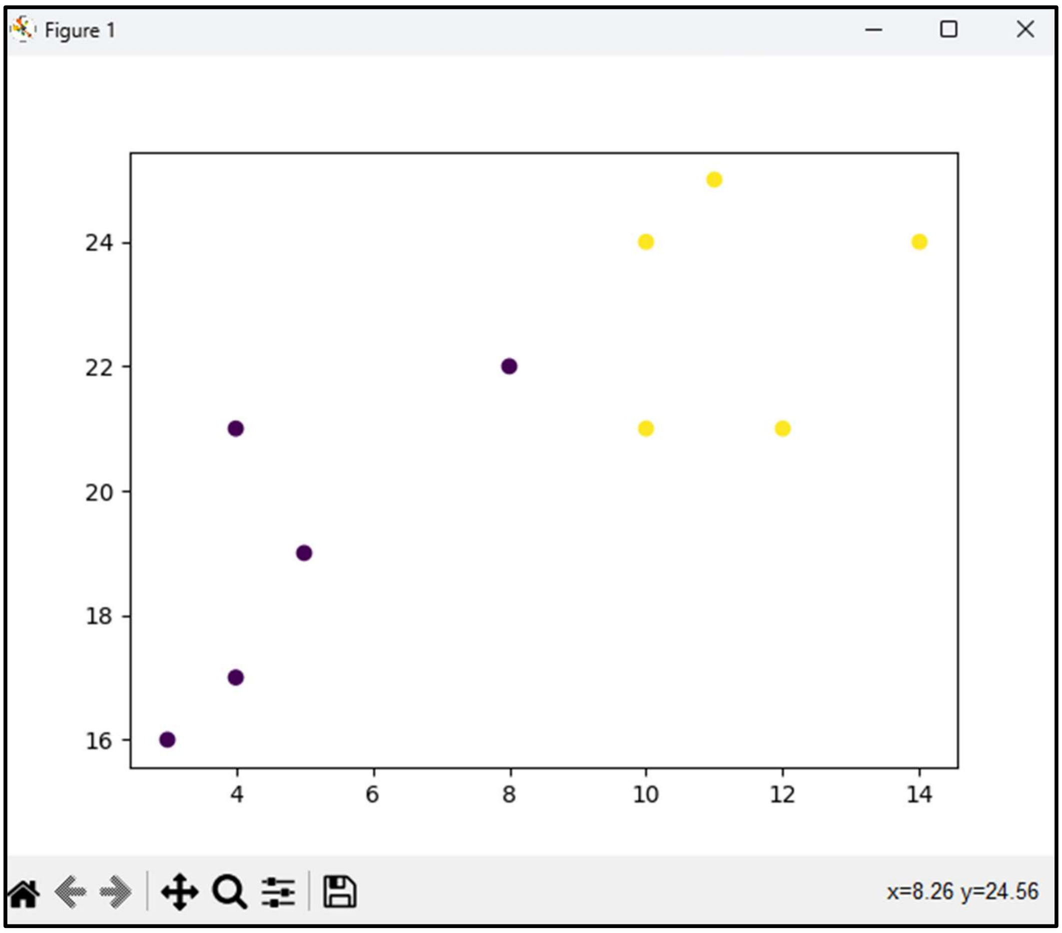
INTRODUCTION:-

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a non-parametric algorithm, which means it does not make any assumption on underlying data.
- It is also called a lazy learner algorithm because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.
- Example: Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So for this identification, we can use the KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs images and based on the most similar features it will put it in either cat or dog category.

CODE:-

```
import matplotlib.pyplot as plt

x = [4, 5, 10, 4, 3, 11, 14 , 8, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
classes = [0, 0, 1, 0, 0, 1, 1, 0, 1, 1]

plt.scatter(x, y, c=classes)
plt.show()

from sklearn.neighbors import KNeighborsClassifier

data = list(zip(x, y))
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(data, classes)

new_x = 8
new_y = 21
new_point = [(new_x, new_y)]
prediction = knn.predict(new_point)

plt.scatter(x + [new_x], y + [new_y], c=classes + [prediction[0]])
plt.text(x=new_x-1.7, y=new_y-0.7, s=f"new point, class: {prediction[0]}")
plt.show()
```

OUTPUT:-



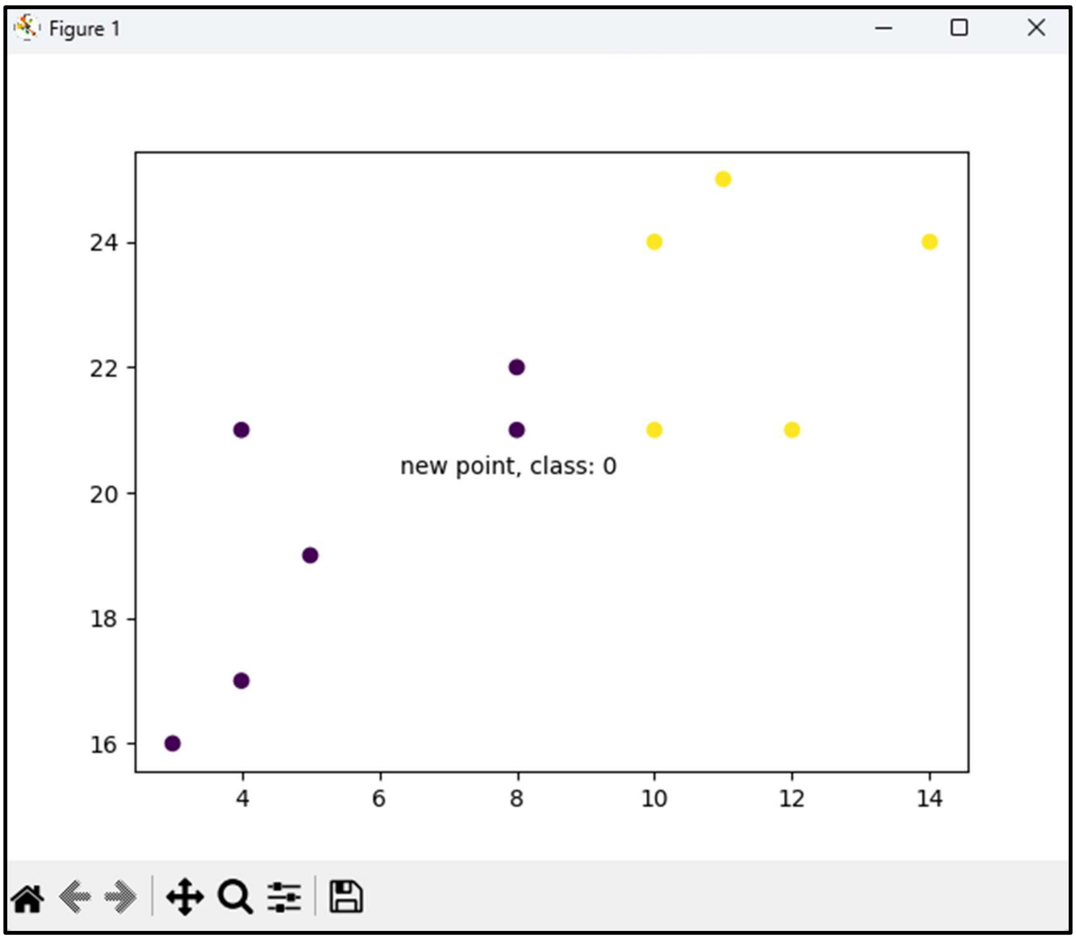


Figure 1
new point, class: 0
24
22
20
18
16
4
6
8
10
12
14

PRACTICAL NO 8

AIM:- Association Rule Mining

 Implement the Association Rule Mining algorithm (e.g., Apriori) to find frequent itemsets.
 Generate association rules from the frequent itemsets and calculate their support and confidence.
 Interpret and analyze the discovered association rules.

INTRODUCTION:-

Association rule mining finds interesting associations and relationships among large sets of data items. This rule shows how frequently a itemset occurs in a transaction. A typical example is a Market Based Analysis. Market Based Analysis is one of the key techniques used by large relations to show associations between items.It allows retailers to identify relationships between the items that people buy together frequently.

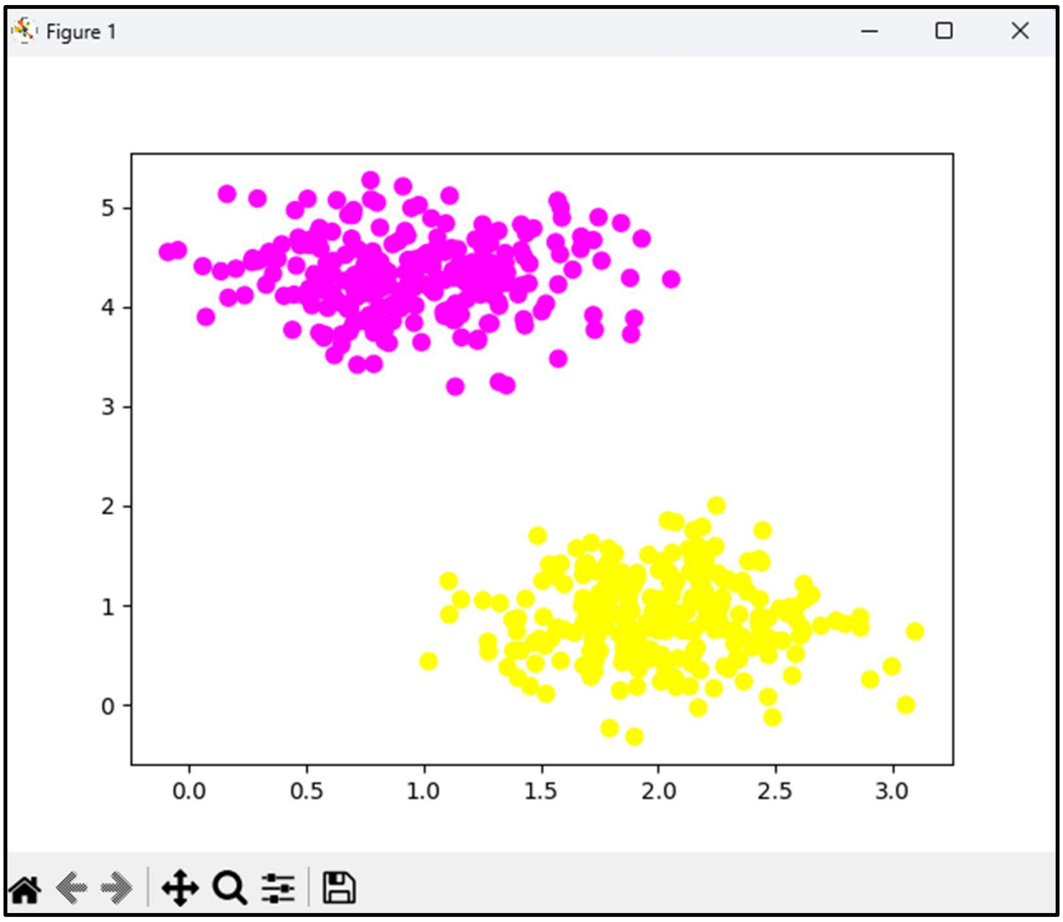
Given a set of transactions, we can find rules that will predict the occurrence of an item based on the occurrences of other items in the transaction. Association rule learning is a type of unsupervised learning technique that checks for the dependency of one data item on another data item and maps accordingly so that it can be more profitable. It tries to find some interesting relations or associations among the variables of dataset. It is based on different rules to discover the interesting relations between variables in the database.

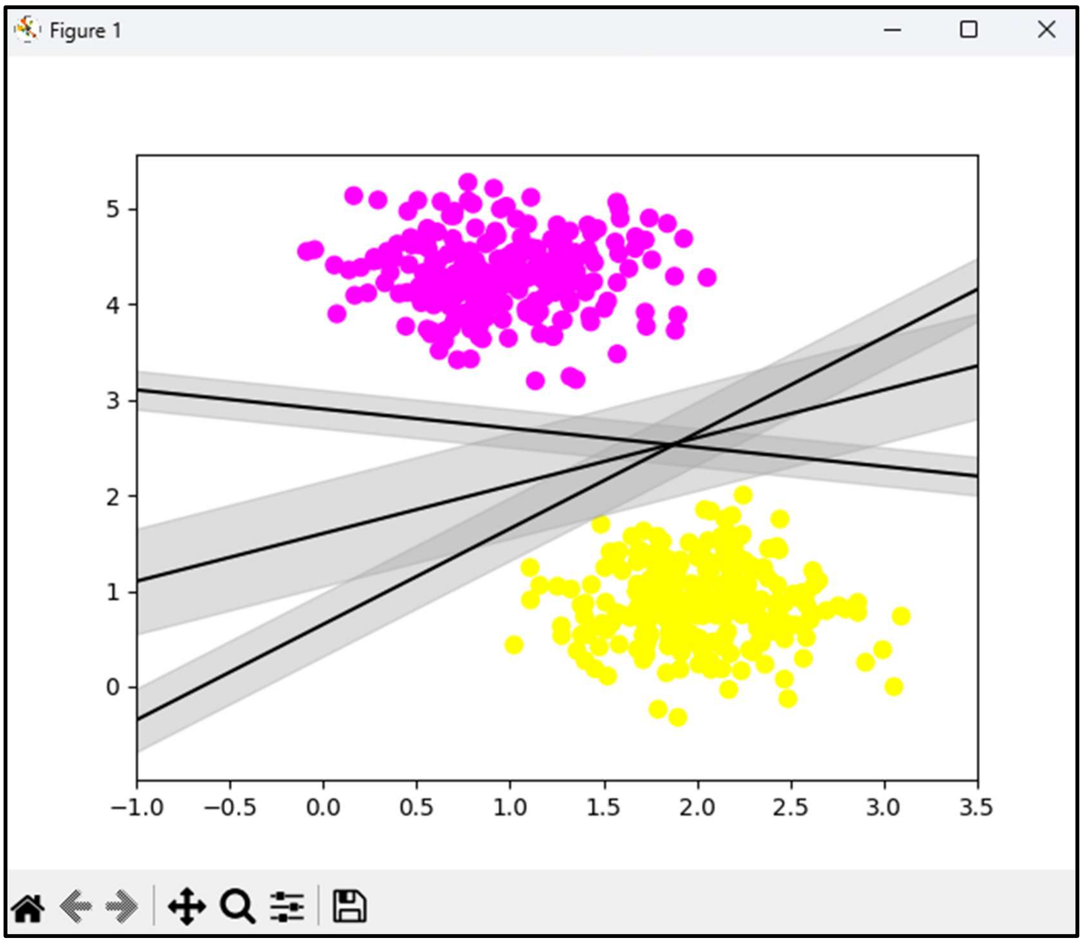
The association rule learning is one of the very important concepts of machine learning, and it is employed in Market Basket analysis, Web usage mining, continuous production, etc. Here market basket analysis is a technique used by the various big retailer to discover the associations between items. We can understand it by taking an example of a supermarket, as in a supermarket, all products that are purchased together are put together.
For example, if a customer buys bread, he most likely can also buy butter, eggs, or milk, so these products are stored within a shelf or mostly nearby.

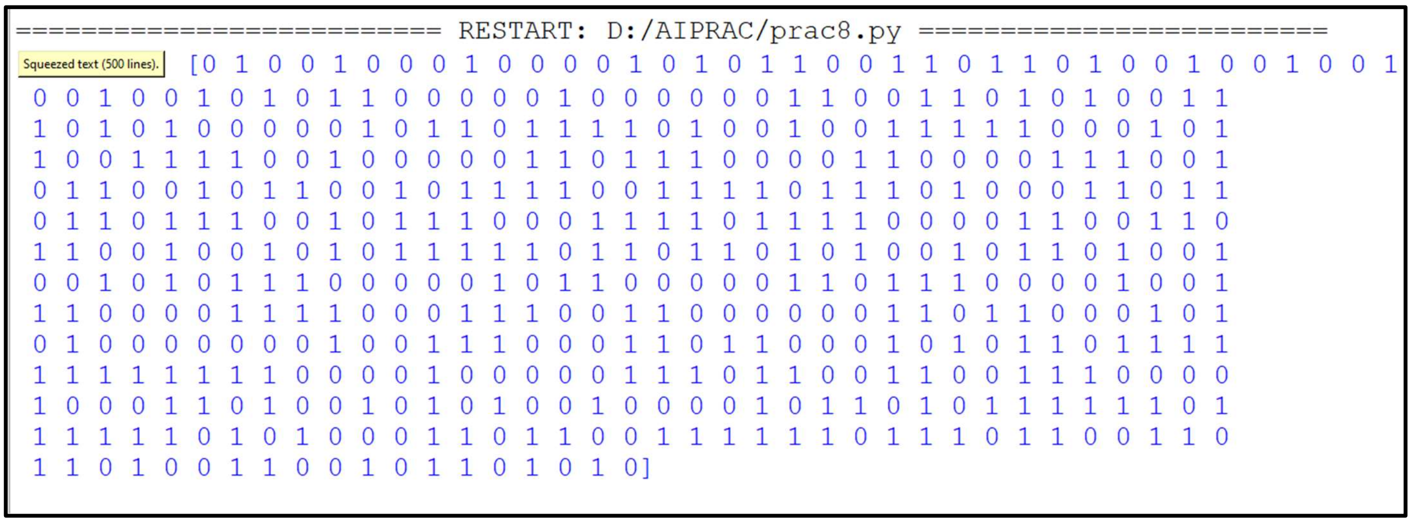
CODE:-

```
from sklearn.datasets import make_blobs
X, Y = make_blobs(n_samples=500, centers=2,random_state=0, cluster_std=0.40)

import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap='spring');
plt.show()

import numpy as np
xfit = np.linspace(-1, 3.5)

plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap='spring')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
                     color='#AAAAAA', alpha=0.4)

plt.xlim(-1, 3.5);
plt.show()
print(X,Y)
```

OUTPUT:-







```
========================== RESTART: D:/AIPRAC/prac8.py ==========================
Squeezed text (500 lines).  [0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 1 1 0 1 1 0 1 0 0 1 0 0 1 0 0 1
 0 0 1 0 0 1 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 0 1 1
 1 0 1 0 1 0 0 0 0 0 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 1 1 1 0 0 0 1 0 1
 1 0 0 1 1 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 1 0 0 0 0 1 1 0 0 0 0 1 1 1 0 0 1
 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 1 1 0 0 1 1 1 1 0 1 1 1 0 1 0 0 0 1 1 0 1 1
 0 1 1 0 1 1 1 0 0 1 0 1 1 1 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 1 1 0 0 1 1 0
 1 1 0 0 1 0 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 1 0 1 0 1 0 0 1 0 1 1 0 1 0 0 1
 0 0 1 0 1 0 1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 1 0 1 1 1 0 0 0 0 1 0 0 1
 1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 1 1 0 1 1 0 0 0 1 0 1
 0 1 0 0 0 0 0 0 0 1 0 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 1 0 1 1 0 1 1 1 1
 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0 1 1 1 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0
 1 0 0 0 1 1 0 1 0 0 1 0 1 0 1 0 0 1 0 0 0 0 1 0 1 1 0 1 0 1 1 1 1 1 1 0 1
 1 1 1 1 1 0 1 0 1 0 0 0 1 1 0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 1 0
 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 0 1 0]
```