Introduction to Artificial Intelligence

&

Machine Learning

(20AM51)

List of Experiments (Artificial Intelligence)

INDEX

List of Experiments (Machine Learning)

S.No	Experiment Name	Page No
1	Implement and demonstrate FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .csv file.	
2	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.	
3	Write a program to demonstrate the working of the decision tree classifier. Use appropriate dataset for building the decision tree and apply this knowledge to classify a new sample.	
4	Write a program to demonstrate the working of Decision tree Regressor Use appropriate dataset for decision tree Regressor.	
5	Write a program to demonstrate the working of Random Forest classifier. Use appropriate dataset for Random Forest Classifier.	
6	Write a program to demonstrate the working of Logistic Regression classifier. Use appropriate dataset for Logistic Regression.	

Aim:

1.Implementation of DFS for water jug problem using LISP/PROLOG.

Problem Statement:

There are two jugs of volume A litre and B litre. Neither has any measuring mark on it. There is a pump that can be used to fill the jugs with water. How can you get exactly x litre of water into the A litre jug. Assuming that we have unlimited supply of water.

Solution:

The state space for this problem can be described as the set of ordered **pairs of integers (x,y)**

Where,

x represents the quantity of water in the 4-gallon jug x= 0,1,2,3,4

y represents the quantity of water in 3-gallon jug y=0,1,2,3

Start State: (0,0)

Goal State: (2,0)

Generate production rules for the water jug problem

We basically perform three operations to achieve the goal.

1. Fill water jug.
2. Empty water jug
3. and Transfer water jug

Rule	State	Process

1	(X,Y | X<4)	(4,Y) {Fill 4-gallon jug}
2	(X,Y |Y<3)	(X,3) {Fill 3-gallon jug}
3	(X,Y |X>0)	(0,Y) {Empty 4-gallon jug}
4	(X,Y | Y>0)	(X,0) {Empty 3-gallon jug}
5	(X,Y | X+Y>=4 ^ Y>0)	(4,Y-(4-X)) {Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}
6	(X,Y | X+Y>=3 ^X>0)	(X-(3-Y),3) {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}
7	(X,Y | X+Y<=4 ^Y>0)	(X+Y,0) {Pour all water from 3-gallon jug into 4-gallon jug}
8	(X,Y | X+Y <=3^ X>0)	(0,X+Y) {Pour all water from 4-gallon jug into 3-gallon jug}
9	(0,2)	(2,0) {Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

Initialization:

Start State: (0,0)

Apply Rule 2:

Fill 3-gallon jug

Now the state is (x,3)

Iteration 1:

Current State: (x,3)

Apply Rule 7:

Pour all water from 3-gallon jug into 4-gallon jug

Now the state is (3,0)

Iteration 2:

Current State : (3,0)

Apply Rule 2:

Fill 3-gallon jug

Now the state is (3,3)

Iteration 3:

Current State:(3,3)

Apply Rule 5:

Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full

Now the state is (4,2)

Iteration 4:

Current State : (4,2)

Apply Rule 3:

Empty 4-gallon jug

Now state is (0,2)

Iteration 5:

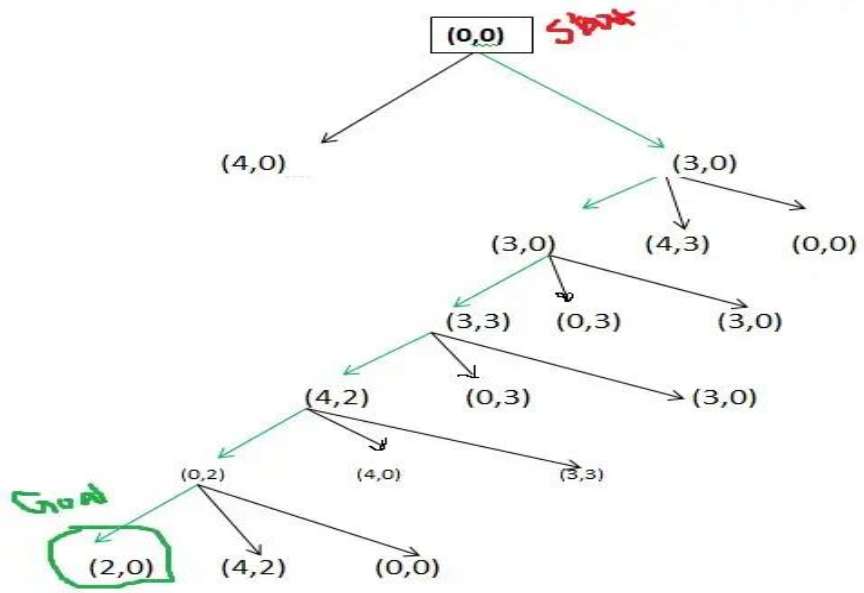
Current State : (0,2)

Apply Rule 9:

Pour 2 gallon water from 3 gallon jug into 4 gallon jug

Now the state is (2,0)-- **Goal Achieved.**

Water Jug Solution using DFS (Depth First Search)



Program:

```
print("Rule 1:Fill x\n Rule 2:Fill y\n Rule 3:Empty x\n Rule 4:Empty y\n Rule 5:From y to x\n Rule 6:From x to y\n Rule 7:From y to x complete\n Rule 8:From x to y complete\n")

cap_x = int(input("Enter the jug 1 capacity: "))

cap_y = int(input("Enter the jug 2 capacity: "))

req_lis = list(map(str,input("Enter the required amount of water and in the jug you needed with space seperate: ").split()))

req_amount = int(req_lis[0])

req_jug = req_lis[1]

x=y=0

while(True):
```

```
rule = int(input("Enter the rule: "))
if rule==1:
    if x<cap_x:
        x = cap_x
if rule==2:
    if y<cap_y:
        y = cap_y
if rule==3:
    if x>0:
        x = 0
if rule==4:
    if y>0:
        y = 0
if rule==5:
    if 0<x+y>=cap_x and y>0:
        x,y = cap_x,y-(cap_x-x)
if rule==6:
    if 0<x+y>=cap_y and x>0:
        x,y = x-(cap_y-y),cap_y
if rule==7:
    if 0<x+y<=cap_x and y>=0:
        x = x+y
```

```
        y = 0
if rule==8:
    if 0<x+y<=cap_y and x>=0:
        y = x+y
        x = 0
print("x :",x)
print("y :",y)
if req_jug=='x':
    if req_amount==x:
        print("Goal reached")
        break
elif req_jug=='y':
    if req_amount==y:
        print("Goal reached")
         break
```

OutPut:

```
Rule 1:Fill x
 Rule 2:Fill y
 Rule 3:Empty x
 Rule 4:Empty y
 Rule 5:From y to x
 Rule 6:From x to y
 Rule 7:From y to x complete
 Rule 8:From x to y complete

Enter the jug 1 capacity: 4
Enter the jug 2 capacity: 3
Enter the required amount of water and in the jug you needed with space seperate: 2 x
Enter the rule: 1
x : 4
y : 0
Enter the rule: 6
x : 1
y : 3
Enter the rule: 4
x : 1
y : 0
Enter the rule: 8
x : 0
y : 1
Enter the rule: 1

x : 4
y : 1
Enter the rule: 6
x : 2
y : 3
Goal reached
```

Aim:

2. Implementation of BFS for tic-tac-toe problem using LISP/PROLOG/Java.

Problem Statement:

- The game is to be played between two people (in this program between HUMAN and COMPUTER).
- One of the player chooses 'O' and the other 'X' to mark their respective cells.
- The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X').
- If no one wins, then the game is said to be draw.

O	X	O
O	X	X
X	O	X

- **Implementation** In our program the moves taken by the computer and the human are chosen randomly. We use rand() function for this. **What more can be done in the program?** The program is in not played optimally by both sides because the moves are chosen randomly. The program can be easily modified so that both players play optimally (which will fall under the category of Artificial Intelligence). Also the program can be modified such that the user himself gives the input (using scanf() or cin). The above changes are left as an exercise to the readers. **Winning Strategy – An**

Interesting Fact If both the players play optimally then it is destined that you will never lose (“although the match can still be drawn”). It doesn’t matter whether you play first or second.In another ways – “ Two expert players will always draw ”. Isn’t this interesting ?

Program:

```
# Set up the game board as a list

board = ["-", "-", "-",

"-", "-", "-",

"-", "-", "-"]


# Define a function to print the game board

def print_board():

print(board[0] + " | " + board[1] + " | " + board[2])

print(board[3] + " | " + board[4] + " | " + board[5])

print(board[6] + " | " + board[7] + " | " + board[8])


# Define a function to handle a player's turn

def take_turn(player):

print(player + "'s turn.")

position = input("Choose a position from 1-9: ")

while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:

position = input("Invalid input. Choose a position from 1-9: ")
```

```
position = int(position) - 1
while board[position] != "-":
position = int(input("Position already taken. Choose a different position: ")) - 1
board[position] = player
print_board()
 # Define a function to check if the game is over
def check_game_over():
if (board[0] == board[1] == board[2] != "-") or \
(board[3] == board[4] == board[5] != "-") or \
(board[6] == board[7] == board[8] != "-") or \
(board[0] == board[3] == board[6] != "-") or \
(board[1] == board[4] == board[7] != "-") or \
(board[2] == board[5] == board[8] != "-") or \
(board[0] == board[4] == board[8] != "-") or \
(board[2] == board[4] == board[6] != "-"):
return "win"
# Check for a tie
elif "-" not in board:
return "tie"
# Game is not over
else:
return "play"
```

```
# Define the main game loop
def play_game():
print_board()
current_player = "X"
game_over = False
while not game_over:
take_turn(current_player)
game_result = check_game_over()
if game_result == "win":
print(current_player + " wins!")
game_over = True
elif game_result == "tie":
print("It's a tie!")
game_over = True
else:
# Switch to the other player
current_player = "O" if current_player == "X" else "X"
 # Start the game
play_game()
```

Output:

```
 - | - | -
 - | - | -
 - | - | -
 X's turn.
 Choose a position from 1-9: 5
 - | - | -
 - | X | -
 - | - | -
 O's turn.
 Choose a position from 1-9: 1
 O | - | -
 - | X | -
 - | - | -
 X's turn.
 Choose a position from 1-9: 9
 O | - | -
 - | X | -
 - | - | X
 O's turn.
 Choose a position from 1-9: 2

O | O | -
- | X | -
- | - | X
X's turn.
Choose a position from 1-9: 4
O | O | -
X | X | -
- | - | X
O's turn.
Choose a position from 1-9: 6
O | O | -
X | X | O
- | - | X
X's turn.
Choose a position from 1-9: 4
Position already taken. Choose a different position: 3
O | O | X
X | X | O
- | - | X
O's turn.
Choose a position from 1-9: 7
O | O | X
X | X | O
O | - | X
X's turn.
```

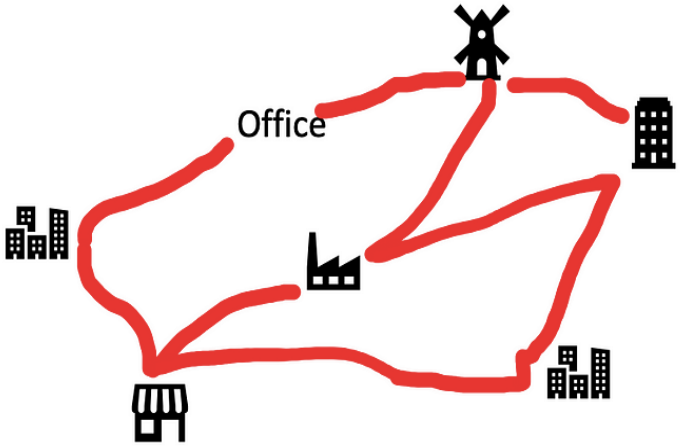
```
Choose a position from 1-9: 8
O | O | X
X | X | O
O | X | X
It's a tie!
```

Aim:

3.Implementation of TSP using heuristic approach using JAVA/LISP/PROLOG.

Problem statement

A list of n cities are Given with the distance between any two cities. Now, you have to start with your office and to visit all the cities only once each and return to your office. What is the shortest path can you take? This problem is called the Traveling Salesman Problem (TSP).



Program:

```
import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;
```

```
public class TSPNearestNeighbor {

    public static int tsp(int[][] graph, int start) {
        int numNodes = graph.length;
        boolean[] visited = new boolean[numNodes];
        visited[start] = true;
        int current = start;
        int cost = 0;
        List<Integer> path = new ArrayList<>();
        path.add(start);

        for (int i = 0; i < numNodes - 1; i++) {
            int next = -1;
            int minDist = Integer.MAX_VALUE;

            for (int j = 0; j < numNodes; j++) {
                if (!visited[j] && graph[current][j] < minDist) {
                    next = j;
                    minDist = graph[current][j];
                }
            }
```

```
            visited[next] = true;
            path.add(next);
            cost += minDist;
            current = next;
        }
        cost += graph[current][start];
        path.add(start);

        System.out.println("Path: " + path);
        return cost;
    }
    public static void main(String[] args) {
        int[][] graph = {
            {0, 2, 9, 10},
            {1, 0, 6, 4},
            {15, 7, 0, 8},
            {6, 3, 12, 0}
        };

        int startNode = 0;
        int minCost = tsp(graph, startNode);
```

```
        System.out.println("Minimum cost: " + minCost);
    }
}
```

OutPut:

Path: [0, 1, 3, 2, 0]

Minimum cost: 33

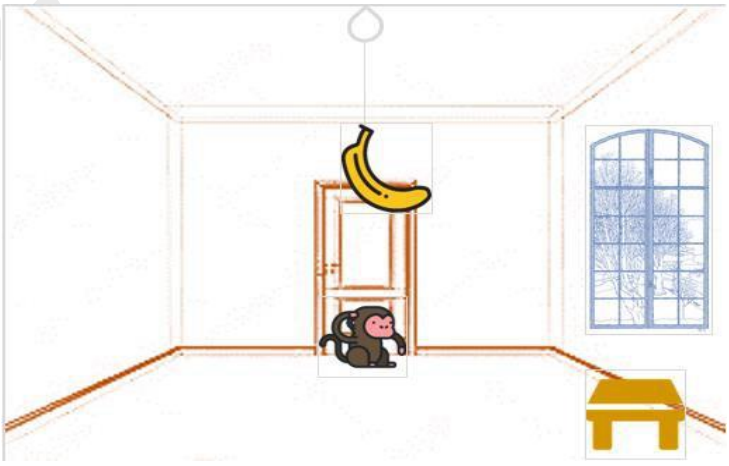
Aim:

4. Implementation of Monkey Banana Problem using LISP/PROLOG

Problem Statement

Suppose the problem is as given below −

- A hungry monkey is in a room, and he is near the door.
- The monkey is on the floor.
- Bananas have been hung from the center of the ceiling of the room.
- There is a block (or chair) present in the room near the window.
- The monkey wants the banana, but cannot reach it.



So how can the monkey get the banana?

So if the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case –

Assumptions:

- Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.
- If the block position is not at the center, then monkey can drag it to the center.
- If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
- When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

We have some predicates that will move from one state to another state, by performing action.

- When the block is at the middle, and monkey is on top of the block, and monkey does not have the banana (i.e. ***has not*** state), then using the ***grasp*** action, it will change from ***has not*** state to ***have*** state.
- From the floor, it can move to the top of the block (i.e. ***on top*** state), by performing the action ***climb***.
- The **push** or **drag** operation moves the block from one place to another.
- Monkey can move from one place to another using **walk** or **move** clauses.
- Another predicate will be canget(). Here we pass a state, so this will perform move predicate from one state to another using different actions,

then perform canget() on state 2. When we have reached to the state '**has**>', this indicates '**has banana**'. We will stop the execution.

Program:

```
on(floor,monkey).

on(floor,box).

in(room,monkey).

in(room,box).

at(ceiling,banana).

strong(monkey).

Gras0p(monkey).

climb(monkey,box).

push(monkey,box):-

strong(monkey).

under(banana,box):-

push(monkey,box).

canreach(banana,monkey):-

at(floor,banana);

    at(ceiling,banana),

    under(banana,box),

    climb(monkey,box).

canget(banana,monkey):-

canreach(banana,monkey),grasp(monkey).
```

Output:

```
?- ["D:/monkey.pl"].
true.

?- push(monkey,box).
true.

?- under(banana,box).
true.

?- canreach(banana,monkey).
true.

?- canget(banana,monkey)
|    .
true.

?- trace.
true.
```

```
[trace]  ?- canreach(banana,monkey).
   Call: (10) canreach(banana, monkey) ? creep
   Call: (11) at(floor, banana) ? creep
   Fail: (11) at(floor, banana) ? creep
   Redo: (10) canreach(banana, monkey) ? creep
   Call: (11) at(ceiling, banana) ? creep
   Exit: (11) at(ceiling, banana) ? creep
   Call: (11) under(banana, box) ? creep
   Call: (12) push(monkey, box) ? creep
   Call: (13) strong(monkey) ? creep
   Exit: (13) strong(monkey) ? creep
   Exit: (12) push(monkey, box) ? creep
   Exit: (11) under(banana, box) ? creep
   Call: (11) climb(monkey, box) ? creep
   Exit: (11) climb(monkey, box) ? creep
   Exit: (10) canreach(banana, monkey) ? creep
true.
```

```
[trace]  ?- canget(banana,monkey).
   Call: (10) canget(banana, monkey) ? creep
   Call: (11) canreach(banana, monkey) ? creep
   Call: (12) at(floor, banana) ? creep
   Fail: (12) at(floor, banana) ? creep
   Redo: (11) canreach(banana, monkey) ? creep
   Call: (12) at(ceiling, banana) ? creep
   Exit: (12) at(ceiling, banana) ? creep
   Call: (12) under(banana, box) ? creep
   Call: (13) push(monkey, box) ? creep
   Call: (14) strong(monkey) ? creep
   Exit: (14) strong(monkey) ? creep
   Exit: (13) push(monkey, box) ? creep
   Exit: (12) under(banana, box) ? creep
   Call: (12) climb(monkey, box) ? creep
   Exit: (12) climb(monkey, box) ? creep
   Exit: (11) canreach(banana, monkey) ? creep
   Call: (11) grasp(monkey) ? creep
   Exit: (11) grasp(monkey) ? creep
   Exit: (10) canget(banana, monkey) ? creep
true.
```

AIM:

5) Implementation of Hill-climbing to solve 8- Puzzle Problem.

Problem Statement :

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- 0Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state

Features of Hill Climbing:

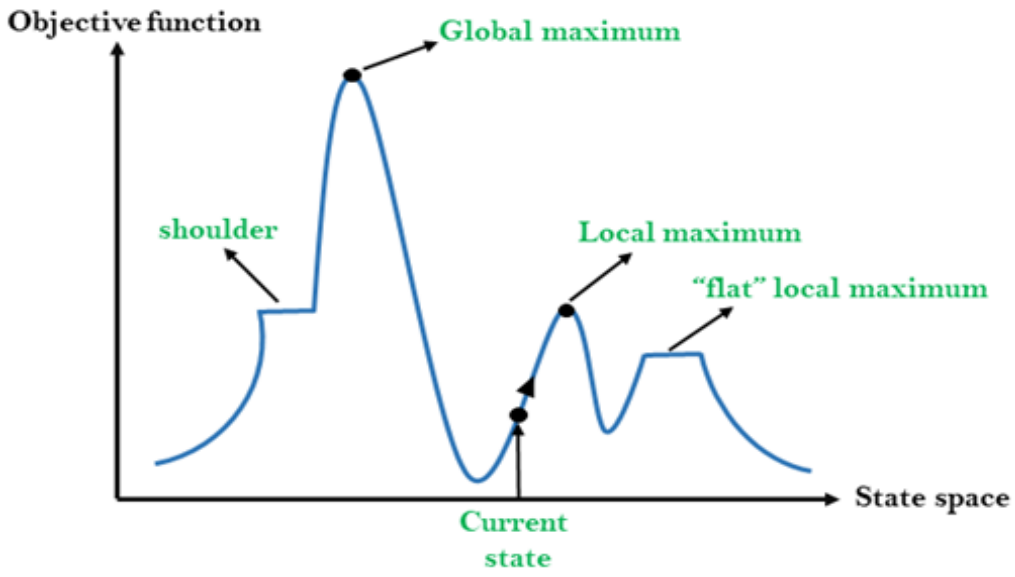
Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:\

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state**. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
    1. If it is goal state, then return success and quit.
    2. Else if it is better than the current state then assign new state as a current state.
    3. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors.

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
    1. Let SUCC be a state such that any successor of the current state will be better than it.
    2. For each operator that applies to the current state:
        I. Apply the new operator and generate a new state.

II. Evaluate the new state.

III. If it is goal state, then return it and quit, else compare it to the SUCC.

IV. If it is better than SUCC, then set new state as SUCC.

V. If the SUCC is better than the current state, then set current state to SUCC.

- **Step 5:** Exit.

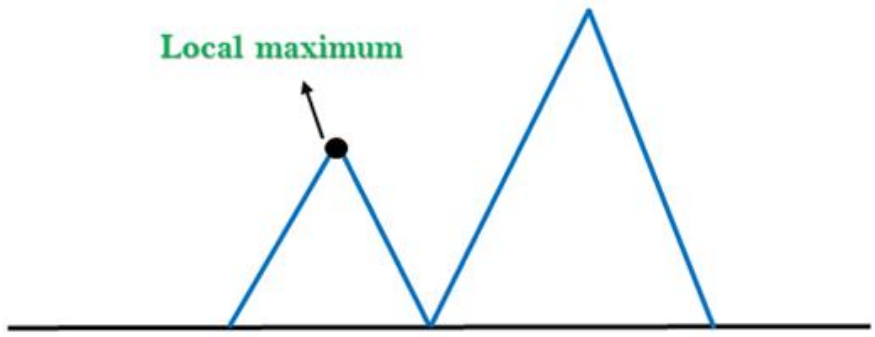
3. Stochastic hill climbing:

Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

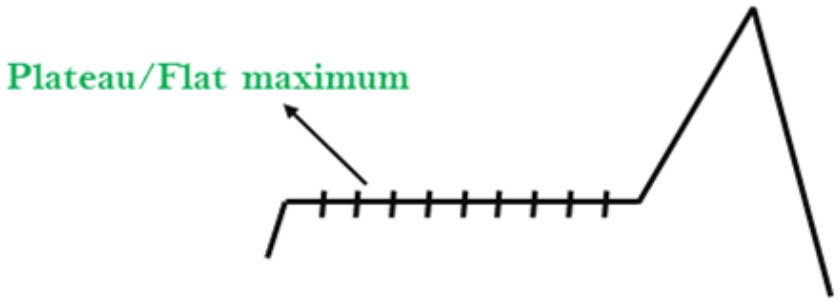
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.

Program:

```
import random
import numpy as np
import networkx as nx
#coordinate of the points/cities
coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6],
[6,7], [15,20], [10,9], [12,12]])
#adjacency matrix for a weighted graph based on the given coordinates
def generate_matrix(coordinate):
    matrix = []
    for i in range(len(coordinate)):
        for j in range(len(coordinate)) :
            p = np.linalg.norm(coordinate[i] - coordinate[j])
            matrix.append(p)
    matrix = np.reshape(matrix, (len(coordinate),len(coordinate)))
    #print(matrix)
    return matrix
#finds a random solution
def solution(matrix):
    points = list(range(0, len(matrix)))
    solution = []
    for i in range(0, len(matrix)):
        random_point = points[random.randint(0, len(points) - 1)]
        solution.append(random_point)
        points.remove(random_point)
    return solution
```

```
#calculate the path based on the random solution
def path_length(matrix, solution):
    cycle_length = 0
    for i in range(0, len(solution)):
        cycle_length += matrix[solution[i]][solution[i - 1]]
    return cycle_length

#generate neighbors of the random solution by swapping cities and returns the
best neighbor
def neighbors(matrix, solution):
    neighbors = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbor = solution.copy()
            neighbor[i] = solution[j]
            neighbor[j] = solution[i]
            neighbors.append(neighbor)

    #assume that the first neighbor in the list is the best neighbor
    best_neighbor = neighbors[0]
    best_path = path_length(matrix, best_neighbor)

    #check if there is a better neighbor
    for neighbor in neighbors:
        current_path = path_length(matrix, neighbor)
        if current_path < best_path:
            best_path = current_path
            best_neighbor = neighbor
    return best_neighbor, best_path
```

```
def hill_climbing(coordinate):
    matrix = generate_matrix(coordinate)
    current_solution = solution(matrix)
    current_path = path_length(matrix, current_solution)
    neighbor = neighbors(matrix,current_solution)[0]
    best_neighbor, best_neighbor_path = neighbors(matrix, neighbor)
    while best_neighbor_path < current_path:
        current_solution = best_neighbor
        current_path = best_neighbor_path
        neighbor = neighbors(matrix, current_solution)[0]
        best_neighbor, best_neighbor_path = neighbors(matrix, neighbor)
   return current_path, current_solution
final_solution = hill_climbing(coordinate)
print("The solution is \n", final_solution[1])
```

OUTPUT :

The solution is

[1, 2, 4, 8, 3, 10, 9, 7, 5, 0, 6]

Aim:

6.Implementation of Simulated Annealing Algorithm using LISP/PROLOG.

Problem Statement:

A cost function *f: R^n –> R*, find an *n*-tuple that minimizes the value of *f*. minimizing the value of a function is algorithmically equivalent to maximization (since we can redefine the cost function as 1-f).

Many of you with a background in calculus/analysis are likely familiar with simple optimization for single variable functions. For instance, the function *f(x) = x^2 +*

2x can be optimized setting the first derivative equal to zero, obtaining the solution *x = -1* yielding the minimum value *f(-1) = -1*. This technique suffices for simple functions with few variables. However, it is often the case that researchers are interested in optimizing functions of several variables, in which case the solution can only be obtained computationally.

- Move all points 0 or 1 units in a random direction
- Shift input elements randomly
- Swap random elements in input sequence
- Permute input sequence
- Partition input sequence into a random number of segments and permute segments

Program:

```
import random

import math

class Solution:

    def __init__(self, CVRMSE, configuration):

        self.CVRMSE = CVRMSE

        self.config = configuration

T = 1

Tmin = 0.0001
```

```
alpha = 0.9

numIterations = 100

def genRandSol():

    # Instantiating for the sake of compilation

    a = [1, 2, 3, 4, 5]

    return Solution(-1.0, a)

def neighbor(currentSol):

    return currentSol

def cost(inputConfiguration):

    return -1.0

# Mapping from [0, M*N] --> [0,M]x[0,N]

def indexToPoints(index):

    points = [index % M, index//M]

    return points

M = 5
```

```
N = 5

sourceArray = [['X' for i in range(N)] for j in range(M)]

min = Solution(float('inf'), None)

currentSol = genRandSol()



while T > Tmin:

    for i in range(numIterations):

        # Reassigns global minimum accordingly

        if currentSol.CVRMSE < min.CVRMSE:

            min = currentSol

        newSol = neighbor(currentSol)

        ap = math.exp((currentSol.CVRMSE - newSol.CVRMSE)/T)

        if ap > random.uniform(0, 1):

            currentSol = newSol

    T *= alpha  # Decreases T, cooling phase
```

```
# Returns minimum value based on optimization

print(min.CVRMSE, "\n\n")

for i in range(M):

    for j in range(N):

        sourceArray[i][j] = "X"

# Displays

for obj in min.config:

    coord = indexToPoints(obj)

    sourceArray[coord[0]][coord[1]] = "-"

# Displays optimal location

for i in range(M):

    row = ""

    for j in range(N):

        row += sourceArray[i][j] + " "

    print(row)
```

Output

-1.0

[X, -, X, X, X]

[-, X, X, X, X]

[-, X, X, X, X]

[-, X, X, X, X]

[-, X, X, X, X]

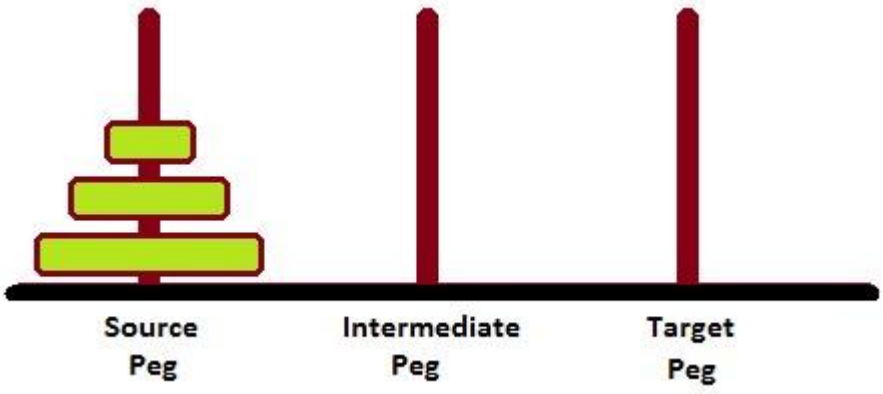
Aim:

7. Implementation of Towers Of Hanoi Problem using LISP/PROLOG

Problem Statement It is a famous puzzle to move N disks from the source peg/tower to the target peg/tower using the intermediate peg as an auxiliary holding peg. There are two conditions that are to be followed while solving this problem −

- A larger disk cannot be placed on a smaller disk.
- Only one disk can be moved at a time.

The following diagram depicts the starting setup for N=3 disks.



To solve this, write one procedure move(N, Source, Target, auxiliary). Here N number of disks will have to be shifted from Source peg to Target peg keeping Auxiliary peg as intermediate.

For example –

- Move(3, source, target, auxiliary).

- Move top disk from source to target
- Move top disk from source to auxiliary
- Move top disk from target to auxiliary
- Move top disk from source to target
- Move top disk from auxiliary to source
- Move top disk from auxiliary to target
- Move top disk from source to target

Program:

```
move(1,X,Y,_) :-
  write('Move top disk from '), write(X), write(' to '), write(Y), nl.
move(N,X,Y,Z) :-
  N>1,
  M is N-1,
  move(M,X,Z,Y),
  move(1,X,Y,_),
  move(M,Z,Y,X).
```

Output:

```
| ?- [towersofhanoi].
yes
| ?- move(4,source,target,auxiliary).
Move top disk from source to auxiliary
Move top disk from source to target
Move top disk from auxiliary to target
Move top disk from source to auxiliary
Move top disk from target to source
Move top disk from target to auxiliary
Move top disk from source to auxiliary
```

Move top disk from source to target
Move top disk from auxiliary to target
Move top disk from auxiliary to source
Move top disk from target to source
Move top disk from auxiliary to target
Move top disk from source to auxiliary
Move top disk from source to target
Move top disk from auxiliary to target

true ?

(31 ms) yes

MACHINE LEARNING PROGRAMS

AIM:

1.Implement and demonstrate FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .csv file.

The find-S algorithm is a basic concept learning algorithm in machine learning. The find-S algorithm finds the most specific hypothesis that fits all the positive examples. We have to note here that the algorithm considers only those positive training example. The find-S algorithm starts with the most specific hypothesis and generalizes this hypothesis each time it fails to classify an observed positive training data. Hence, the Find-S algorithm moves from the most specific hypothesis to the most general hypothesis.

Important Representation:

1. **?** indicates that any value is acceptable for the attribute.
2. specify a single required value (e.g., Cold) for the attribute.
3. **ϕ** indicates that no value is acceptable.
4. The most **general hypothesis** is represented by: **{?, ?, ?, ?, ?, ?}**
5. The most **specific hypothesis** is represented by: **{ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}**

Steps Involved In Find-S :

Start with the most specific hypothesis. **h = {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}**

Take the next example and if it is negative, then no changes occur to the hypothesis.

If the example is positive and we find that our initial hypothesis is too specific then we update our current hypothesis to a general condition.

Keep repeating the above steps till all the training examples are complete.

After we have completed all the training examples we will have the final hypothesis when can use to classify the new examples.

FIND-S Algorithm

1. Initialize h to the most specific hypothesis in H
2. For each positive training instance x

    For each attribute constraint a_i in h

        If the constraint a_i is satisfied by x

        Then do nothing

        Else replace a_i in h by the next more general constraint that is satisfied by x
3. Output hypothesis h

Data set:

Outlook	Temperature	Humidity	Wind	Play Tennis
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Weak	Yes

Program:

```
import csv

num_attributes = 4
```

```
a = []
print("\n The Given Training Data Set \n")
with open('finds.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            a.append (row)
            print(row)
print("\n The initial value of hypothesis: ")
hypothesis = ['0'] * num_attributes
print(hypothesis)
for j in range(0,num_attributes):
     hypothesis[j] = a[0][j];
print("\n Find S: Finding a Maximally Specific Hypothesis\n")
for i in range(0,len(a)):
   if a[i][num_attributes]=='Yes':
          for j in range(0,num_attributes):
              if a[i][j]!=hypothesis[j]:
                      hypothesis[j]='?'
              else :
                      hypothesis[j]= a[i][j]
   print(" For Training instance No:{0} the hypothesis is".format(i),hypothesis)
```

```
print("\n The Maximally Specific Hypothesis for a given TrainingExamples
:\n")
print(hypothesis)
```

Output:

```
The Given Training Data Set

['Overcast', 'Hot', 'High', 'Weak', 'Yes']
['Rain', 'Mild', 'High', 'Weak', 'Yes']
['Rain', 'Cool', 'Normal', 'Strong', 'No']
['Overcast', 'Cool', 'Normal', 'Weak', 'Yes']
['Overcast', 'Hot', 'High', 'Weak', 'Yes']

 The initial value of hypothesis:
['0', '0', '0', '0']

 Find S: Finding a Maximally Specific Hypothesis

 For Training instance No:0 the hypothesis is

['Overcast', 'Hot', 'High', 'Weak']

 For Training instance No:1 the hypothesis is

['?', '?', 'High', 'Weak']

 For Training instance No:2 the hypothesis is

 ['?', '?', 'High', 'Weak']
```

For Training instance No:3 the hypothesis is

['?', '?', '?', 'Weak']

For Training instance No:4 the hypothesis is

['?', '?', '?', 'Weak']

The Maximally Specific Hypothesis for a given TrainingExamples :

['?', '?', '?', 'Weak']

AIM:

2.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.

Problem Statement

The candidate elimination algorithm incrementally builds the version space given a hypothesis space H and a set E of examples. The examples are added one by one; each example possibly shrinks the version space by removing the hypotheses that are inconsistent with the example. The candidate elimination algorithm does this by updating the general and specific boundary for each new example.

- This is an extended form of the Find-S algorithm.
- Consider both positive and negative examples.
- Actually, positive examples are used here as the Find-S algorithm (Basically they are generalizing from the specification).
- While the negative example is specified in the generalizing form.

Terms Used:

- **Concept learning:** Concept learning is basically the learning task of the machine (Learn by Train data)
- **General Hypothesis:** Not Specifying features to learn the machine.
- **G = {‘?’, ‘?’,’?’,’?’…}:** Number of attributes
- **Specific Hypothesis:** Specifying features to learn machine (Specific feature)
- **S= {‘pi’,’pi’,’pi’…}: The number** of pi depends on a number of attributes.
- **Version Space:** It is an intermediate of general hypothesis and Specific hypothesis. It not only just writes one hypothesis but a set of all possible hypotheses based on training data-set.

Algorithm:

Step1: Load Data set

Step2: Initialize General Hypothesis and Specific Hypothesis.

Step3: For each training example

Step4: If example is positive example

```
if attribute_value == hypothesis_value:
   Do nothing
else:
   replace attribute value with '?' (Basically generalizing it)
```

Step5: If example is Negative example

Make generalize hypothesis more specific.

DataSet:

Sky	Temperature	Humid	Wind	Water	Forest	Output
sunny	warm	normal	strong	warm	same	yes
sunny	warm	high	strong	warm	same	yes
rainy	cold	high	strong	warm	change	no
sunny	warm	high	strong	cool	change	yes

Program:

In [203]:
```
import pandas as pd
data=pd.read_excel(r"C:\Users\B V N DURGA VINAY\Downloads\candidate.csv")
data
```

Out[203]:

	sky	temperature	humid	wind	water	forest	output
0	sunny	warm	normal	strong	warm	same	yes
1	sunny	warm	high	strong	warm	same	yes
2	rainy	cold	high	strong	warm	change	no
3	sunny	warm	high	strong	cool	change	yes

In [204]:
```
data.shape
```

Out[204]: (4, 7)

In [205]:
```
S=[]
for i in range(0,data.shape[1]-1):
    S.append('0')
print(S)
```

```
['0', '0', '0', '0', '0', '0']
```

In [206]:
```
G = [['?' for i in range(data.shape[1]-1)] for i in range(data.shape[1]-1)]
print(G)
```

```
G = [['?' for i in range(data.shape[1]-1)] for i in range(data.shape[1]-1)]
print(G)
```

```
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
```

```
x = data.iloc[:,:-1]
y = data.iloc[:,-1]
a = X.values.tolist()
b = Y.values.tolist()
print(a)
print(b)
```

```
[['sunny', 'warm', 'normal', 'strong', 'warm', 'same'], ['sunny', 'warm', 'high', 'strong', 'warm', 'same'], ['rainy', 'col
d', 'high', 'strong', 'warm', 'change'], ['sunny', 'warm', 'high', 'strong', 'cool', 'change']]
['yes', 'yes', 'no', 'yes']
```

```
l=a[0]
m=0
for i in range(0,data.shape[0]):
        n=b[i]
        k=list(a[i])
        print(n)
        print(k)
        if(n=='yes'):
            for j in range(0,len(l)):
                    if k[j]!=S[j] and S[j]!='0':
                        S[j]='?'
                    elif k[j]!=S[j] and S[j]=='0':
```

```
                elif k[j]!=S[j] and S[j]=='0':
                    S=k
        else:
            for j in range(0,len(l)):
                if k[j]!=S[j]:
                    G[m][j]=S[j]
                    m=m+1
                else:
                    G[m][j]='?'
                    m=m+1
print("G:",G)
```

```
yes
['sunny', 'warm', 'normal', 'strong', 'warm', 'same']
yes
['sunny', 'warm', 'high', 'strong', 'warm', 'same']
no
['rainy', 'cold', 'high', 'strong', 'warm', 'change']
yes
['sunny', 'warm', 'high', 'strong', 'cool', 'change']
G: [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]
```

```
G1 = list(G)
r = [G[j].count('?') for j in range(len(G))]
for i in r:
    if i>data.shape[1]-2:
        G.pop(r.index(i))
```

```
    if i>data.shape[1]-2:
        G.pop(r.index(i))
p = []
for i in range(len(a)):
    if b[i]=='yes':
        p.append(a[i])
c = 0
for i in range(len(G1[0])):
    if p[c][i]!=G1[c][i]:
        G1.pop(c)
    else:
        c=c+1
print("G is",G1)
print("S is",S)
```

OutPut:

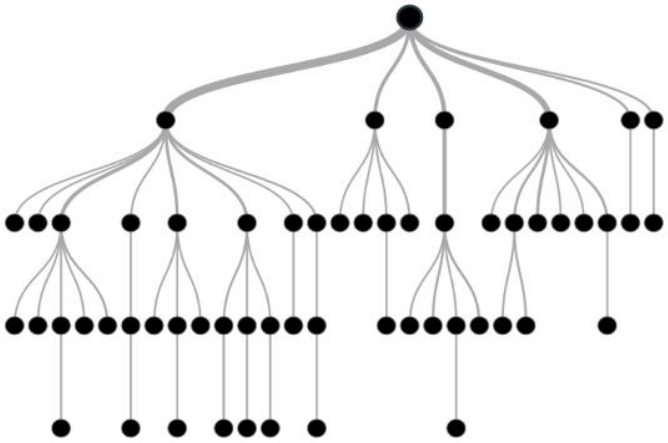
```
G is [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]
S is ['sunny', 'warm', '?', 'strong', '?', '?']
```

AIM:

3 .Write a program to demonstrate the working of the decision tree classifier. Use appropriate dataset for building the decision tree and apply this knowledge to classify a new sample.

Problem Statement:

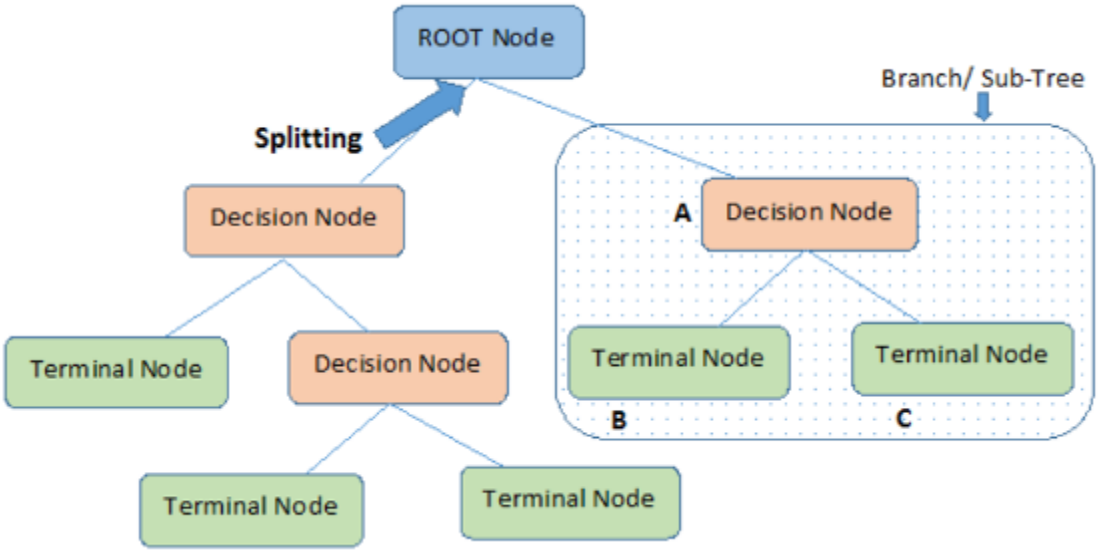
It is a tool that has applications spanning several different areas. Decision trees can be used for classification as well as regression problems. The name itself suggests that it uses a flowchart like a tree structure to show the predictions that result from a series of feature-based splits. It starts with a root node and ends with a decision made by leaves.



Before learning more about decision trees let's get familiar with some of the terminologies:

- ***Root Nodes*** – It is the node present at the beginning of a decision tree from this node the population starts dividing according to various features.
- ***Decision Nodes*** – the nodes we get after splitting the root nodes are called Decision Node
- ***Leaf Nodes*** – the nodes where further splitting is not possible are called leaf nodes or terminal nodes

- ***Sub-tree*** – just like a small portion of a graph is called sub-graph similarly a sub-section of this decision tree is called sub-tree.
- ***Pruning*** – is nothing but cutting down some nodes to stop overfitting.



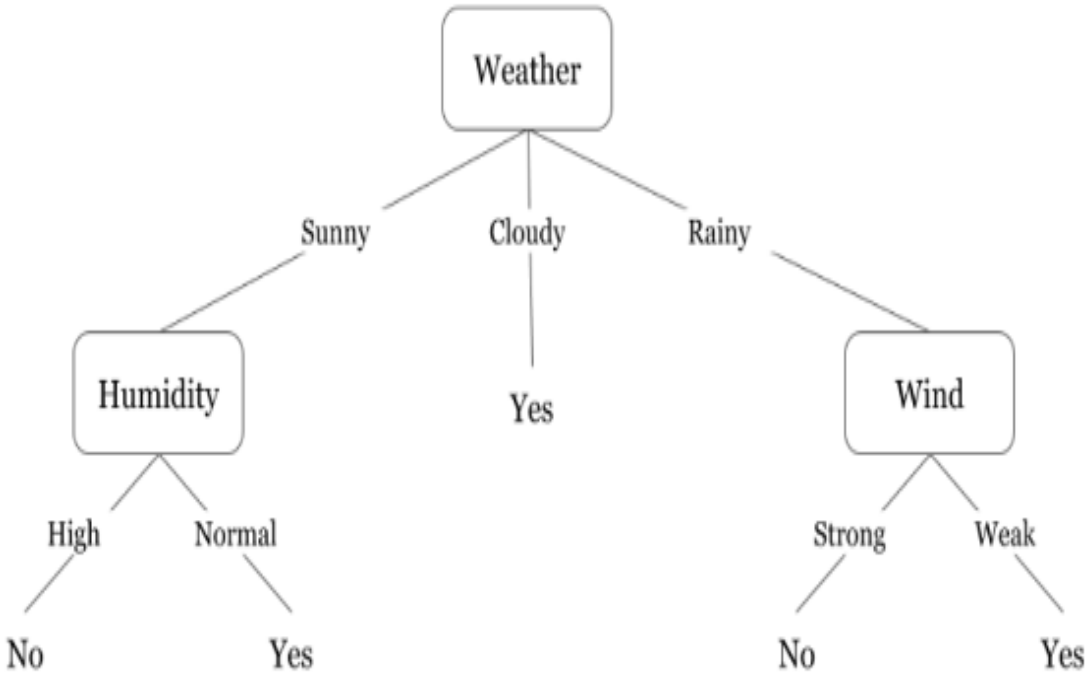
Example of Decision Tree

Data Set:

Day	Weather	Temperature	Humidity	Wind	Play?
1	Sunny	Hot	High	Weak	No
2	Cloudy	Hot	High	Weak	Yes
3	Sunny	Mild	Normal	Strong	Yes
4	Cloudy	Mild	High	Strong	Yes
5	Rainy	Mild	High	Strong	No
6	Rainy	Cool	Normal	Strong	No
7	Rainy	Mild	High	Weak	Yes
8	Sunny	Hot	High	Strong	No
9	Cloudy	Hot	Normal	Weak	Yes
10	Rainy	Mild	High	Strong	No

Decision trees are upside down which means the root is at the top and then this root is split into various several nodes. Decision trees are nothing but a bunch of if-else statements in layman terms. It checks if the condition is true and if it is then it goes to the next node attached to that decision.

In the below diagram the tree will first ask what is the weather? Is it sunny, cloudy, or rainy? If yes, then it will go to the next feature which is humidity and wind. It will again check if there is a strong wind or weak, if it's a weak wind and it's rainy then the person may go and play.



Program:

In [24]:
```
import pandas as pd
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
data=pd.read_csv("diabetes.csv")
data
```

Out[24]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
...	...	...	...	...	...	...	...	...	...
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

768 rows × 9 columns

In [25]:
```
x=data.drop(["Outcome"],axis=1)
y=data["Outcome"]
x.shape
```

Out[25]: (768, 8)

In [26]:
```
y.shape
```

Out[26]: (768,)

In [27]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2)
```

In [30]:
```
from sklearn import metrics
clf = DecisionTreeClassifier()
clf = clf.fit(x_train,y_train)
y_pred = clf.predict(x_test)
```

In [31]:
```
print(y_pred)
```

```
[1 1 0 0 1 1 1 1 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0 1 1 0 0 1 0 1
 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1
 0 1 1 1 0 0 0 1 1 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 1 0 0 1 1 0 0
 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0
 1 0 1 0 1 0]
```

In [29]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.7077922077922078
```

AIM:

4.Write a program to demonstrate the working of Decision tree regressor. Use appropriate dataset for decision tree regressor.

Problem Statement:

Decision tree regression observes features of an object and trains a model in the structure of a tree to predict data in the future to produce meaningful continuous output. Continuous output means that the

output/result is not discrete, i.e., it is not represented just by a discrete, known set of numbers or values.

Discrete output example: A weather prediction model that predicts whether or not there'll be rain on a particular day.

Continuous output example: A profit prediction model that states the probable profit that can be generated from the sale of a product.

Here, continuous values are predicted with the help of a decision tree regression model.

Step 1: Import the required libraries.

Step 2: Initialize and print the Dataset.

Step 3: Select all the rows and column 1 from the dataset to "X".

Step 4: Select all of the rows and column 2 from the dataset to "y".

Step 5: Fit decision tree regressor to the dataset.

Step 6: Predicting a new value.

Step 7: Visualising the result.

Program:

```
import pandas as pd

data=pd.read_csv('C:/Users/ML Lab/Desktop/attendance.csv')

print(data)
```

output:

```
   Attendence  Marks
0         100     99
1          95     95
2          90     89
3          85     84
4          80     79
5          75     73
6          70     68
```

```
X=data[['Attendance']]  #input

print(X.shape)
```

output:

```
(7, 1)
```

```
#Store Marks from data(dataframe) in to y in 1D for LinearRegression

y=data['Marks'] #output

print(y.shape)
```

output:

```
(7,)
```

```
#Split the training set and test data set from the original X and y using
train_test_split()
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test= train_test_split(X,y,test_size=0.3)
 #Bulid the Linear Regression Model
# training the dataset
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor(random_state = 0)
model.fit(X_train,y_train)
#Model Prediction on X_test data
y_pred=model.predict(X_test)
#Test on New Instance
print(model.predict([[68]]))
```

output:

```
[73.]
```

```
from sklearn.metrics import r2_score,mean_squared_error
print("R2 score fit",r2_score(y_test,y_pred))
print("Mean squared error: ", mean_squared_error(y_test, y_pred))
df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print(df)
```

output:

```
R2 score fit 0.835820895522388
Mean squared error:  22.0
   Actual  Predicted
1      95       99.0
6      68       73.0
2      89       84.0
```

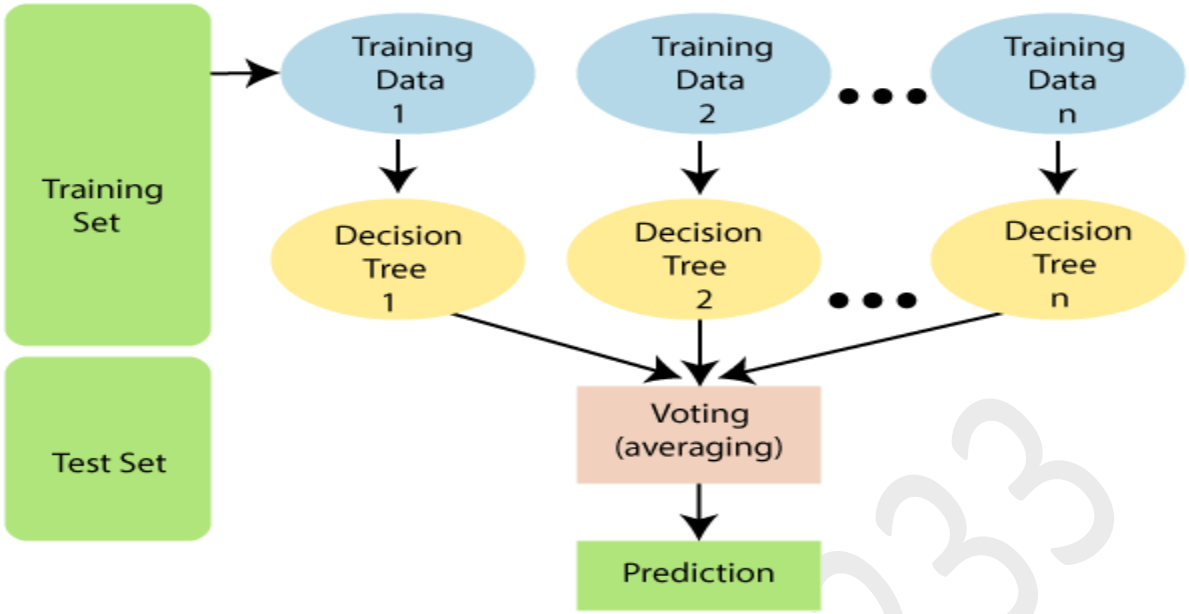
AIM:

5.Write a program to demonstrate the working of Random Forest classifier. Use appropriate dataset for Random Forest Classifier+

Problem Statement:

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of **ensemble learning,** which is a process of *combining multiple classifiers to solve a complex problem and to improve the performance of the model.*

As the name suggests, ***"Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset."*** Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.



Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.

The Working process can be explained in the below steps and diagram:

Algorithm:

Step-1: Select random K data points from the training set.

Step-2: Build the decision trees associated with the selected data points (Subsets).

Step-3: Choose the number N for decision trees that you want to build.

Step-4: Repeat Step 1 & 2.

Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

Program:

In [83]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as mlp
ds=pd.read_csv("User_Data.csv")
ds
```

Out[83]:

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0
...	...	...	...	...	...
395	15691863	Female	46	41000	1
396	15706071	Male	51	23000	1
397	15654296	Female	50	20000	1
398	15755018	Male	36	33000	0
399	15594041	Female	49	36000	1

400 rows × 5 columns

In [84]:
```
x=ds.iloc[:,[2,3]].values
y=ds.iloc[:,4].values
```

In [85]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2)
from sklearn.preprocessing import StandardScaler
st=StandardScaler()
x_train=st.fit_transform(x_train)
x_test=st.transform(x_test)
```

In [86]:
```
from sklearn.ensemble import RandomForestClassifier
classifier= RandomForestClassifier(n_estimators= 10, criterion="entropy")
classifier.fit(x_train, y_train)
y_pred=classifier.predict(x_test)
```

In [87]:
```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
cm
```

Out[87]:
```
array([[53,  5],
       [ 5, 17]], dtype=int64)
```

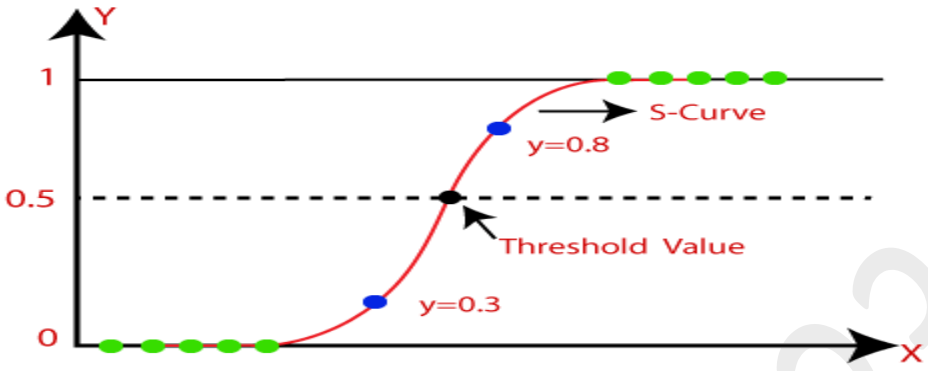
AIM:

6.Write a program to demonstrate the working of Logistic Regression classifier. Use appropriate dataset for Logistic Regression.

Problem Statement:

- Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.
- Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, **it gives the probabilistic values which lie between 0 and 1**.
- Logistic Regression is much similar to the Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas **Logistic regression is used for solving the classification problems**.
- In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
- The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.
- Logistic Regression is a significant machine learning algorithm because it has the ability to provide probabilities and classify new data using continuous and discrete datasets.

- Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification. The below image is showing the logistic function:



Logistic Function (Sigmoid Function):

- The sigmoid function is a mathematical function used to map the predicted values to probabilities.
- It maps any real value into another value within a range of 0 and 1.
- The value of the logistic regression must be between 0 and 1, which cannot go beyond this limit, so it forms a curve like the "S" form. The S-form curve is called the Sigmoid function or the logistic function.
- In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.

Assumptions for Logistic Regression:

- The dependent variable must be categorical in nature.

- The independent variable should not have multi-collinearity.

Logistic Regression Equation:

The Logistic regression equation can be obtained from the Linear Regression equation. The mathematical steps to get Logistic Regression equations are given below:

- The equation of the straight line can be written as:

$$y= b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n$$

- In Logistic Regression y can be between 0 and 1 only, so for this let's divide the above equation by (1-y):

$$\frac{y}{1-y}\text{; 0 for y= 0, and infinity for y=1}$$

- But the range should be between -[infinity] to +[infinity], then take logarithm of the equation it will become:

$$log\left[\frac{y}{1-y}\right] = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n$$

Program:

```
#Data Pre-procesing Step
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
#importing datasets
data_set= pd.read_csv('User_Data.csv')
data_set
```

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0
...	...	...	...	...	...
395	15691863	Female	46	41000	1
396	15706071	Male	51	23000	1
397	15654296	Female	50	20000	1
398	15755018	Male	36	33000	0
399	15594041	Female	49	36000	1

400 rows × 5 columns

In [2]:
```
#Extracting Independent and dependent Variable
x= data_set.iloc[:, [2,3]].values
y= data_set.iloc[:, 4].values
```

In [3]:
```
# Splitting the dataset into training and test set.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)
```

In [4]:
```
#feature Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
x_train= st_x.fit_transform(x_train)
x_test= st_x.transform(x_test)
```

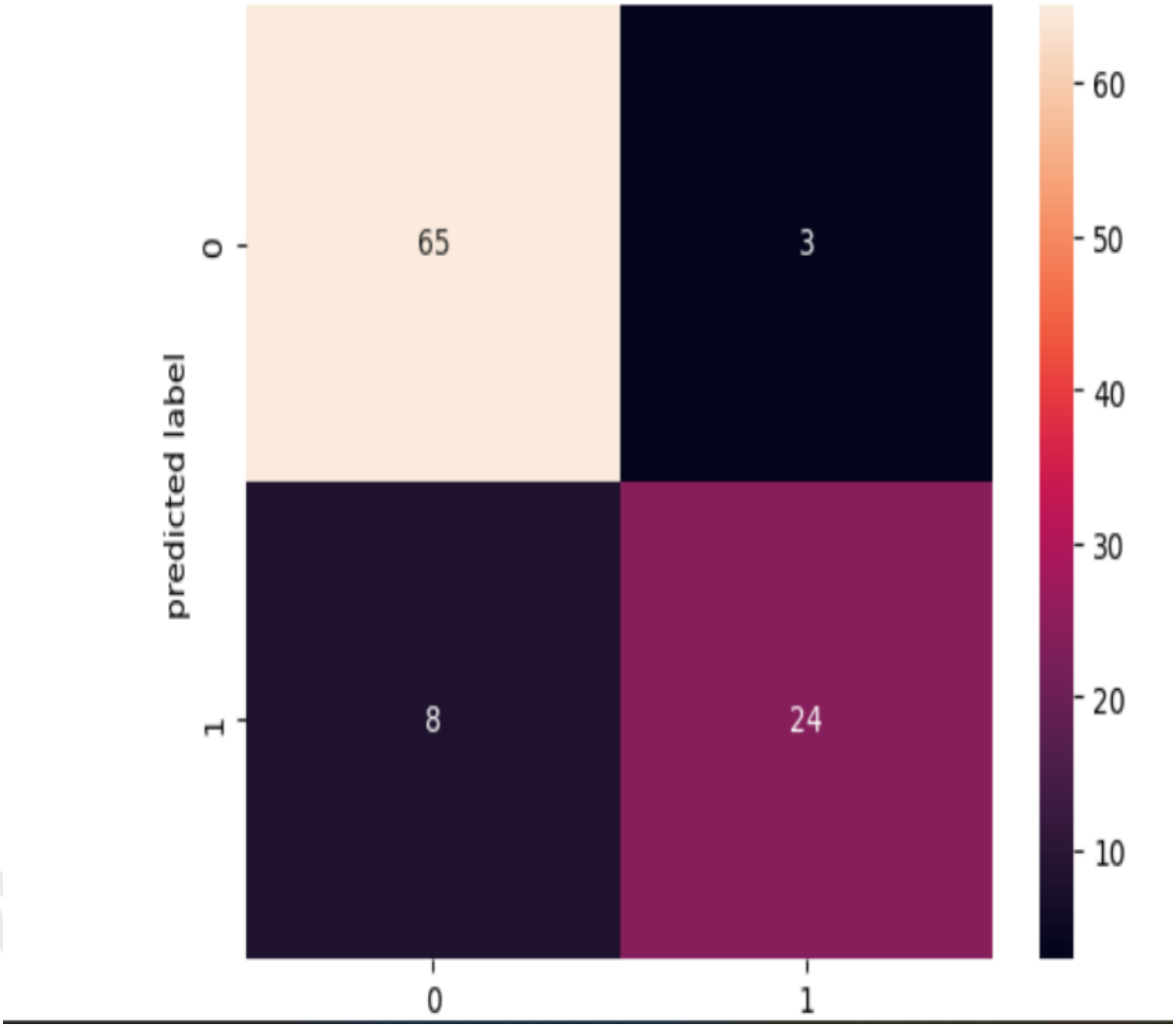
In [5]:
```
#Fitting Logistic Regression to the training set
from sklearn.linear_model import LogisticRegression
classifier= LogisticRegression(random_state=0)
classifier.fit(x_train, y_train)
```

Out[5]:
```
LogisticRegression(random_state=0)
```

In [6]:
```
#Predicting the test set result
y_pred= classifier.predict(x_test)
```

In [28]:
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.heatmap(cm,annot=True)
plt.xlabel("Actual Label")
plt.ylabel("predicted label")
```

Out[28]: Text(50.722222222222214, 0.5, 'predicted label')



```
In [23]: from sklearn.metrics import classification_report
target_names=['with','without']
print(classification_report(y_test,y_pred,target_names=target_names))
```

	precision	recall	f1-score	support
with	0.89	0.96	0.92	68
without	0.89	0.75	0.81	32
accuracy			0.89	100
macro avg	0.89	0.85	0.87	100
weighted avg	0.89	0.89	0.89	100

GIT HUB ID:

github.com/sravanthikotapati