



SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)
DELHI-NCR CAMPUS, GHAZIABAD (U.P.)

ARTIFICIAL INTELLIGENCE LAB

(Subject Code: (18CSC305J))
B.TECH IIL Year / VI Semester

NAME-

REG. No.-



DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
FACULTY OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY,
delhi ncr CAMPUS, MODINAGAR
SIKRI KALAN, DELHI MEERUT ROAD, DIST. – GHAZIABAD - 201204
www.srmup.in

Even Semester (JAN-2021)

BONAFIDE CERTIFICATE

Registration no.

*Certified to be the bonafide record of work done by _
Of 6th semester 3rd year B.TECH degree course in SRM INSTITUTE OF SCIENCE
& TECHNOLOGY, DELHI-NCR Campus for the Department of **Computer
Science & Engineering**, in Artificial Intelligence Laboratory during the
academic year **2020-21**.*

Lab In charge

Head of the department

*Submitted for end semester examination held on __/____/____ at SRM INSTITUTE
OF SCIENCE & TECHNOLOGY, DELHI-NCR Campus.*

Internal Examiner-I

Internal Examiner-II

INDEX

Exp No.	Title of Experiment	Page No.	Date of Experiment	Date of Completion of Experiment	Teacher's Signature
1	Implementation of Toy problem Example-Implement water jug problem				
2	Developing Agent Program for Real World Problem.				
3	Implementation of Constraint satisfaction problem, Example: Implement N- queen Problem				
4	To Implementation and Analysis of BFS and DFS for Application.				
5	To implement Best first search and A* algorithm.				
6	To implement Minimax Algorithm.				
7	Implementation of unification and resolution for real world problems.				
8	Implementation of knowledge representation schemes – use cases.				
9	Implementation of uncertain methods for an application.				
10	Implementation of block world problem.				
11	Implementation of Learning Algo				
12	Development of ensemble model				
13					

Experiment 1

- **Aim** – Implementation of Toy problem
Example-Implement water jug problem.

- **Algorithm** –

Rule	State	Process
1	$(X, Y \mid X < 4)$	$(4, Y)$ {Fill 4-gallon jug}
2	$(X, Y \mid Y < 3)$	$(X, 3)$ {Fill 3-gallon jug}
3	$(X, Y \mid X > 0)$	$(0, Y)$ {Empty 4-gallon jug}
4	$(X, Y \mid Y > 0)$	$(X, 0)$ {Empty 3-gallon jug}
5	$(X, Y \mid X + Y \geq 4 \wedge Y > 0)$	$(4, Y - (4 - X))$ {Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}
6	$(X, Y \mid X + Y \geq 3 \wedge X > 0)$	$(X - (3 - Y), 3)$ {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}
7	$(X, Y \mid X + Y \leq 4 \wedge Y > 0)$	$(X + Y, 0)$ {Pour all water from 3-gallon jug into 4-gallon jug}
8	$(X, Y \mid X + Y \leq 3 \wedge X > 0)$	$(0, X + Y)$ {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}

- **Code** –

```

print("Water jug problem")
x=int(input("Enter X : "))
y=int(input("Enter Y : "))
while True:
    rn=int(input("Enter the rule no. : "))
    if rn==2:
        if y<3:
            x=0
            y=3
    if rn==3:
        if x>0:
            x=0
            y=3
    if rn==5:
```

```

    if x+y>4:
        x=4
        y=y-(4-x)
if m==7:
    if x+y<4:
        x=x+y
        y=0
if m==9:
    x=2
    y=0
print("X=",x)
print("Y=",y)
if x==2:
    print("The result is a goal state")
    break

```

- **Result –**

```

PS C:\Users\Anupriya Johri> & python "c:/Users/Anupriya Johri/Desktop/Anu college/AI exp/waterjug.py"
Water jug problem
Enter X : 0
Enter Y : 0
Enter the rule no. : 2
X= 0
Y= 3
Enter the rule no. : 3
X= 0
Y= 3
Enter the rule no. : 5
X= 0
Y= 3
Enter the rule no. : 7
X= 3
Y= 0
Enter the rule no. : 9
X= 2
Y= 0
The result is a goal state

```

Experiment 2

- Aim – Developing Agent Program for Real World Problem.

- Algorithm –

- Code –

```
import random
class Environment(object):
    def __init__(self):
        self.locationCondition = {'A': '0', 'B': '0'}
        self.locationCondition['A'] = random.randint(0, 1)
        self.locationCondition['B'] = random.randint(0, 1)
class SimpleReflexVacuumAgent(Environment):
    def __init__(self, Environment):
        print (Environment.locationCondition)
        Score = 0
        vacuumLocation = random.randint(0, 1)
        if vacuumLocation == 0:
            print ("Vacuum is randomly placed at Location A")
            if Environment.locationCondition['A'] == 1:
                print ("Location A is Dirty. ")
                Environment.locationCondition['A'] = 0;
                Score += 1
                print ("Location A has been Cleaned. :D")
                if Environment.locationCondition['B'] == 1:
                    print ("Location B is Dirty.")
                    print ("Moving to Location B...")
                    Score -= 1
                    Environment.locationCondition['B'] = 0;
                    Score += 1
                    print ("Location B has been Cleaned :D.")
            else:
                if Environment.locationCondition['B'] == 1:
                    print ("Location B is Dirty.")
                    Score -= 1
                    print ("Moving to Location B...")
                    Environment.locationCondition['B'] = 0;
                    Score += 1
                    print ("Location B has been Cleaned. :D")
        elif vacuumLocation == 1:
            print ("Vacuum is randomly placed at Location B. ")
            if Environment.locationCondition['B'] == 1:
                print ("Location B is Dirty")
                Environment.locationCondition['B'] = 0;
                Score += 1
```

```

        print ("Location B has been Cleaned")
    if Environment.locationCondition['A'] == 1:
        print ("Location A is Dirty")
        Score -= 1
        print ("Moving to Location A")
        Environment.locationCondition['A'] = 0;
        Score += 1
        print ("Location A has been Cleaned")
    else:
        if Environment.locationCondition['A'] == 1:
            print ("Location A is Dirty")
            print ("Moving to Location A")
            Score -= 1
            Environment.locationCondition['A'] = 0;
            Score += 1
            print ("Location A has been Cleaned")
    print (Environment.locationCondition)
    print ("Performance Measurement: " + str(Score))
theEnvironment = Environment()
theVacuum = SimpleReflexVacuumAgent(theEnvironment)

```

- **Result –**

```

PS C:\Users\Anupriya Johri> & python "c:/Users/Anupriya Johri/Desktop/Anu college/AI exp/realagent.py"
{'A': 1, 'B': 1}
Vacuum is randomly placed at Location A
Location A is Dirty.
Location A has been Cleaned. :D
Location B is Dirty.
Moving to Location B...
Location B has been Cleaned :D.
{'A': 0, 'B': 0}
Performance Measurement: 1
PS C:\Users\Anupriya Johri> 

```

Experiment 3

- **Aim** – Implementation of Constraint satisfaction problem
Example: Implement N- queen Problem

- **Algorithm** –
while there are untried configurations
{
 generate the next configuration
 if queens don't attack in this configuration then
 {
 print this configuration;
 }
}

- **Code** –
global N
N=int(input("enter no of queens : "))
def printSolution(board):
 for i in range(N):
 for j in range(N):
 print(board[i][j],end=" ")
 print(" ")
def isSafe(board,row,col):
 for i in range(col):
 if board[row][i]=='Q':
 return False

 for i,j in zip(range(row,-1,-1), range(col,-1,-1)):
 if board[i][j]=='Q':
 return False

 for i,j in zip(range(row,N,1), range(col,-1,-1)):
 if board[i][j]=='Q':
 return False

 return True

def solveNQUtil(board,col):
 if col>=N:
 return True

 for i in range(N):
 if isSafe(board,i,col):
 board[i][col]='Q'
 if solveNQUtil(board,col+1) == True:
 return True

 board[i][col]=0


```

        return False

def solveNQ():
    board = [[0 for i in range(N)] for j in range(N)]

    if solveNQUtil(board,0)==False:
        print("Solution does not exist")
        return False
    printSolution(board)
    return True

solveNQ()

```

- **Result –**

```

PS C:\Users\Anupriya Johri> & python "c:/Users/Anupriya Johri/Desktop/Anu college/AI exp/nqueen.py"
enter no of queens : 8
Q 0 0 0 0 0 0 0 0
0 0 0 0 0 0 Q 0
0 0 0 0 Q 0 0 0
0 0 0 0 0 0 0 Q
0 Q 0 0 0 0 0 0
0 0 0 Q 0 0 0 0
0 0 0 0 0 Q 0 0
0 0 Q 0 0 0 0 0

```

Experiment 4

- **Aim** – To Implementation and Analysis of BFS and DFS for Application.
- **Algorithm** –
 1. Create a node list (Queue) that initially contains the first node N and mark it as visited.
 2. Visit the adjacent unvisited vertex of N and insert it in a queue.
 3. If there are no remaining adjacent vertices left, remove the first vertex from the queue mark it as visited, display it.
 4. Repeat step 1 and step 2 until the queue is empty or the desired node is found.
- **Code** –

```
graph = {
    'S': ['A', 'B'],
    'A': ['C', 'D'],
    'B': ['G', 'H'],
    'C': ['E', 'F'],
    'D': [],
    'G': ['I'],
    'H': [],
    'E': ['K'],
    'F': [],
    'I': [],
    'K': []
}
visited=[]
queue=[]
def bfs(visited,graph,node):
    visited.append(node)
    queue.append(node)

    while queue:
        P=queue.pop(0)
        print(P,end=" ")

        for neighbour in graph[P]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

avisit=set()
def dfs(avisit,graph,node):
    if node not in avisit:
        print(node,end=" ")
        avisit.add(node)
        for neighbour in graph[node]:
            dfs(avisit,graph,neighbour)
```

```
print("Breadth first search")
bfs(visited,graph,'S')
print("\nDepth first search")
dfs(avisit,graph,'S')
```

- **Result –**

```
PS C:\Users\Anupriya Johri> & python "c:/Users/Anupriya Johri/Desktop/Anu college/AI exp/dfs bfs.py"
Breadth first search
S A B C D G H E F I K
Depth first search
S A C E K F D B G I H
```

Experiment 5

- **Aim-**To implement Best First Search and A* algorithm.

- **Algorithm-**

1. Best First Search-

Step 1: Place the starting node into the OPEN list.

Step 2: If the OPEN list is empty, Stop and return failure.

Step 3: Remove the node n , from the OPEN list which has the lowest value of $h(n)$, and places it in the CLOSED list.

 If node n is goal then return
 else

 Step 4: Expand the node n , and generate and check the successors of node n . and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 5.

 Step 5: For each successor node, algorithm checks for evaluation function $f(n)$, and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

 Step 6: Return to Step 2.

2. A*-

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function ($g+h$), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n' , check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest $g(n')$ value.

Step 6: Return to Step 2.

- **Code-**

1. Best First Search-

This class represent a graph

class Graph:

 # Initialize the class

 def __init__(self, graph_dict=None, directed=True):

 self.graph_dict = graph_dict or {}

 self.directed = directed

 if not directed:

 self.make_undirected()

 # Create an undirected graph by adding symmetric edges

 def make_undirected(self):

 for a in list(self.graph_dict.keys()):

 for (b, dist) in self.graph_dict[a].items():

```

        self.graph_dict.setdefault(b, {})[a] = dist
    # Add a link from A and B of given distance, and also add the inverse link if the
graph is undirected
    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance
    # Get neighbors or a neighbor
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)
    # Return a list of nodes in the graph
    def nodes(self):
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)
    # This class represent a node
class Node:
    # Initialize the class
    def __init__(self, name:str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0 # Distance to start node
        self.h = 0 # Distance to goal node
        self.f = 0 # Total cost
    # Compare nodes
    def __eq__(self, other):
        return self.name == other.name
    # Sort nodes
    def __lt__(self, other):
        return self.f < other.f
    # Print node
    def __repr__(self):
        return '({0},{1})'.format(self.position, self.f)
    # Best-first search
    def best_first_search(graph, heuristics, start, end):

        # Create lists for open nodes and closed nodes
        open = []
        closed = []

```

```

# Create a start node and an goal node
start_node = Node(start, None)
goal_node = Node(end, None)
# Add the start node
open.append(start_node)

# Loop until the open list is empty
while len(open) > 0:
    # Sort the open list to get the node with the lowest cost first
    open.sort()
    # Get the node with the lowest cost
    current_node = open.pop(0)
    # Add the current node to the closed list
    closed.append(current_node)

    # Check if we have reached the goal, return the path
    if current_node == goal_node:
        path = []
        while current_node != start_node:
            path.append(current_node.name + ': ' + str(current_node.g))
            current_node = current_node.parent
        path.append(start_node.name + ': ' + str(start_node.g))
        # Return reversed path
        return path[::-1]
    # Get neighbours
    neighbors = graph.get(current_node.name)
    # Loop neighbors
    for key, value in neighbors.items():
        # Create a neighbor node
        neighbor = Node(key, current_node)
        # Check if the neighbor is in the closed list
        if(neighbor in closed):
            continue
        # Calculate cost to goal
        neighbor.g = current_node.g + graph.get(current_node.name, neighbor.name)
        neighbor.h = heuristics.get(neighbor.name)
        neighbor.f = neighbor.h
        # Check if neighbor is in open list and if it has a lower f value
        if(add_to_open(open, neighbor) == True):
            # Everything is green, add neighbor to open list
            open.append(neighbor)
    # Return None, no path is found
    return None
# Check if a neighbor should be added to open list

```

```

def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f >= node.f):
            return False
    return True
# The main entry point for this module
def main():
    # Create a graph
    graph = Graph()
    # Create graph connections (Actual distance)
    graph.connect('Jaipur', 'Gurugram', 111)
    graph.connect('Jaipur', 'Mumbai', 85)
    graph.connect('Gurugram', 'Noida', 104)
    graph.connect('Gurugram', 'Sitapur', 140)
    graph.connect('Gurugram', 'Delhi', 183)
    graph.connect('Mumbai', 'Noida', 230)
    graph.connect('Mumbai', 'Kolkata', 67)
    graph.connect('Kolkata', 'Bilaspur', 191)
    graph.connect('Kolkata', 'Sitapur', 64)
    graph.connect('Noida', 'Delhi', 171)
    graph.connect('Noida', 'Madurai', 170)
    graph.connect('Noida', 'Pondicherry', 220)
    graph.connect('Sitapur', 'Delhi', 107)
    graph.connect('Bilaspur', 'Bern', 91)
    graph.connect('Bilaspur', 'Zurich', 85)
    graph.connect('Bern', 'Zurich', 120)
    graph.connect('Zurich', 'Memmingen', 184)
    graph.connect('Memmingen', 'Delhi', 55)
    graph.connect('Memmingen', 'Madurai', 115)
    graph.connect('Madurai', 'Delhi', 123)
    graph.connect('Madurai', 'Pondicherry', 189)
    graph.connect('Madurai', 'Raipur', 59)
    graph.connect('Raipur', 'Shimla', 81)
    graph.connect('Pondicherry', 'Lucknow', 102)
    graph.connect('Shimla', 'Lucknow', 126)
    # Make graph undirected, create symmetric connections
    graph.make_undirected()
    # Create heuristics (straight-line distance, air-travel distance)
    heuristics = {}
    heuristics['Bilaspur'] = 204
    heuristics['Bern'] = 247
    heuristics['Jaipur'] = 215
    heuristics['Kolkata'] = 137
    heuristics['Lucknow'] = 318

```

```

heuristics['Mumbai'] = 164
heuristics['Madurai'] = 120
heuristics['Memmingen'] = 47
heuristics['Noida'] = 132
heuristics['Pondicherry'] = 257
heuristics['Raipur'] = 168
heuristics['Sitapur'] = 75
heuristics['Shimla'] = 236
heuristics['Gurugram'] = 153
heuristics['Zurich'] = 157
heuristics['Delhi'] = 0
# Run search algorithm
path = best_first_search(graph, heuristics, 'Jaipur', 'Delhi')
print(path)
print()
# Tell python to run main method
if __name__ == "__main__": main()

```

2. A*-

```

from queue import PriorityQueue

#Creating Base Class
class State(object):
    def __init__(self, value, parent, start = 0, goal = 0):
        self.children = []
        self.parent = parent
        self.value = value
        self.dist = 0
        if parent:
            self.start = parent.start
            self.goal = parent.goal
            self.path = parent.path[:]
            self.path.append(value)
        else:
            self.path = [value]
            self.start = start
            self.goal = goal

    def GetDistance(self):
        pass

    def CreateChildren(self):
        pass

```



```

# Creating subclass
class State_String(State):
    def __init__(self, value, parent, start = 0, goal = 0 ):
        super(State_String, self).__init__(value, parent, start, goal)
        self.dist = self.GetDistance()

    def GetDistance(self):
        if self.value == self.goal:
            return 0
        dist = 0
        for i in range(len(self.goal)):
            letter = self.goal[i]
            dist += abs(i - self.value.index(letter))
        return dist

    def CreateChildren(self):
        if not self.children:
            for i in range(len(self.goal)-1):
                val = self.value
                val = val[:i] + val[i+1] + val[i] + val[i+2:]
                child = State_String(val, self)
                self.children.append(child)

# Creating a class that hold the final magic
class A_Star_Solver:
    def __init__(self, start, goal):
        self.path = []
        self.vistedQueue = []
        self.priorityQueue = PriorityQueue()
        self.start = start
        self.goal = goal

    def Solve(self):
        startState = State_String(self.start,0,self.start,self.goal)

        count = 0
        self.priorityQueue.put((0,count, startState))
        while(not self.path and self.priorityQueue.qsize()):
            closesetChild = self.priorityQueue.get()[2]
            closesetChild.CreateChildren()
            self.vistedQueue.append(closesetChild.value)
            for child in closesetChild.children:
                if child.value not in self.vistedQueue:
                    count += 1

```

```

        if not child.dist:
            self.path = child.path
            break
        self.priorityQueue.put((child.dist,count,child))
    if not self.path:
        print("Goal Of is not possible !" + self.goal )
    return self.path

# Calling all the existing stuffs
if __name__ == "__main__":
    start1 = "anupriya"
    goal1 = "ayirpuna"
    print("Starting....")
    a = A_Star_Solver(start1,goal1)
    a.Solve()
    for i in range(len(a.path)):
        print("{0}){1}".format(i,a.path[i]))

```

- **Result-**

1. **Best First Search-**



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: Python
PS C:\Users\Anupriya Johri> & python "c:/Users/Anupriya Johri/Desktop/Anu college/AI exp/bestandastar.py"
['Jaipur: 0', 'Gurugram: 111', 'Delhi: 294']
PS C:\Users\Anupriya Johri> 

```

2. **A*-**

```
PS C:\Users\Anupriya Johri> & python "c:/Users/Anupriya Johri/Desktop/Anu college/AI exp/astar.py"
Starting...
0)anupriya
1)anurpiya
2)anrupiya
3)anrpuiya
4)anrpiuya
5)anripuya
6)anirpuya
7)ainrpuya
8)airnpuya
9)airpnuya
10)airpunya
11)airpuyna
12)airpyuna
13)airypuna
14)aiyrpuna
15)ayirpuna
PS C:\Users\Anupriya Johri> |
```

Experiment 6

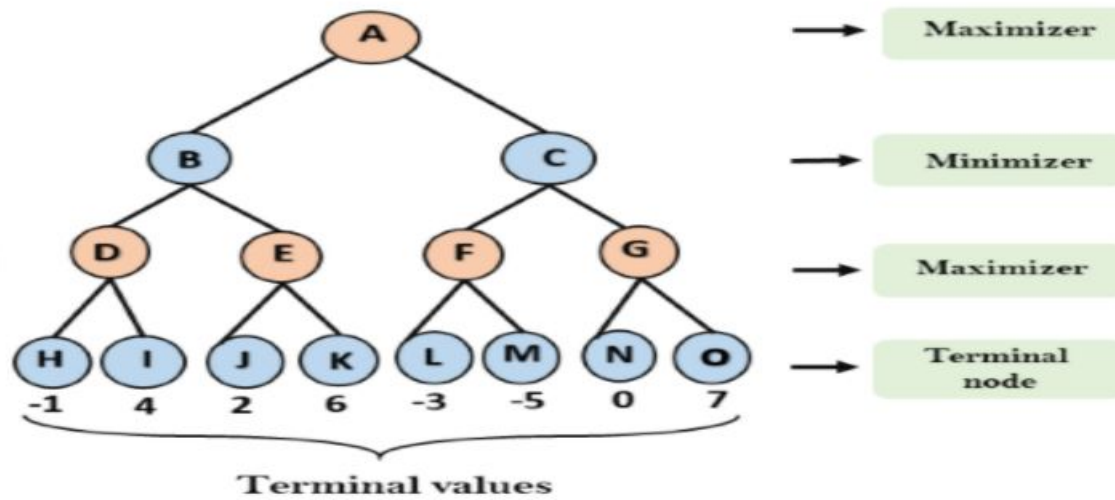
- Aim – To implement Minimax Algorithm.

- Algorithm –

```
function minimax(node, depth, Player)
1.if depth ==0 or node is a terminal node then
return value(node)
2.If Player = 'Max'                                     // for Maximizer Player
    set  $\alpha = -\infty$                                      // worst case value for MAX
    for each child of node do
    value= minimax(child, depth-1, 'MIN')
     $\alpha = \max(\alpha, \text{Value})$                              //gives Maximum of the values
return ( $\alpha$ )
    else                                                 // for Minimizer player
    set  $\alpha = +\infty$                                      // worst case value for MIN
    for each child of node do
    value= minimax(child, depth-1, 'MAX')
     $\alpha = \min(\alpha, \text{Value})$                              //gives minimum of the values
return ( $\alpha$ )
```

- Code –

```
import math
def minimax (curDepth, nodeIndex, maxTurn, scores,targetDepth) :
    if(curDepth==targetDepth):
        return scores[nodeIndex]
    if(maxTurn):
        return max(minimax(curDepth+1,
nodeIndex*2,False,scores,targetDepth),minimax(curDepth+1,
nodeIndex*2+1,False,scores,targetDepth))
    else:
        return min(minimax(curDepth+1,
nodeIndex*2,True,scores,targetDepth),minimax(curDepth+1,
nodeIndex*2+1,True,scores,targetDepth))
scores=[-1,4,2,6,-3,-5,0,7]
treeDepth=math.log(len(scores),2)
print("Optimal value is : ",end=" ")
print(minimax(0,0,True,scores,treeDepth))
```



- Result –

```
PS C:\Users\Anupriya Johri> & python "c:/Users/Anupriya Johri/Desktop/Anu college/AI exp/minmax.py"
Optimal value is : 4
PS C:\Users\Anupriya Johri> 
```

Experiment 7

- Aim – Implementation of unification and resolution for real world problems.
- Algorithm–

Prolog unification

When programming in Prolog, we spend a lot of time thinking about how variables and rules "match" or "are assigned." There are actually two aspects to this. The first, "unification," regards how terms are matched and variables assigned to make terms match. The second, "resolution," is described in separate notes. Resolution is only used if rules are involved. You may notice in these notes that no rules are involved since we are only talking about unification.

Terms

Prolog has three kinds of **terms**:

1. Constants like 42 (numbers) and franklin (atoms, i.e., lower-case words).
2. Variables like X and Person (words that start with upper-case).
3. Complex terms like parent(franklin, bo) and baz(X, quux(Y))

Two terms **unify** if they can be matched. Two terms can be matched if:

- they are the same term (obviously), or
- they contain variables that can be unified so that the two terms without variables are the same.

For example, suppose our knowledge base is:

```
woman(mia).  
loves(vincent, angela).  
loves(franklin, mia).
```

- mia and mia unify because they are the same.
- mia and X unify because X can be given the value mia so that the two terms (without variables) are the same.
- woman(mia) and woman(X) unify because X can be set to mia which results in identical terms.
- loves(X, mia) and loves(vincent, X) **cannot** unify because there is no assignment for X (given our knowledge base) that makes the two terms identical.
- loves(X, mia) and loves(franklin, X) also cannot unify (can you see why?).

We saw in the Prolog notes that we can "query" the knowledge base and get, say, all the people who love mia. When we query with loves(X, mia). we are asking Prolog to give us all the values for X that unify. These values are, essentially, the people who love mia.

Rule :

term1 and term2 unify whenever:

1. If term1 and term2 are **constants**, then term1 and term2 unify if and only if they are the same atom, or the same number.
2. If term1 is a **variable** and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2. (And vice versa.) (If they are both variables, they're both instantiated to each other, and we say that they share values.)
3. If term1 and term2 are **complex terms**, they unify if and only if:

a. They have the same **functor** and **arity**. The functor is the "function" name (this functor is foo: foo(X, bar)). The arity is the number of arguments for the functor (the arity for foo(X, bar) is 2).

b. All of their corresponding arguments unify. **Recursion!**

c. The variable instantiations are compatible (i.e., the same variable is not given two different unifications/values).

1. Two terms unify if and only if they unify for one of the above three reasons (there are no reasons left unstated).

Example

We'll use the = predicate to test if two terms unify. Prolog will answer "Yes" if they do, as well as any sufficient variable assignments to make the unification work.

Do these two terms unify?

1.

?- mia = mia.

o/p Ans:- Yes from Rule 1

2.

?- mia = X.

o/p Ans:- Yes, from rule 2.

3.

?- X = Y.

o/p Yes, from rule 2.

4.

$$?- k(s(g), Y) = k(s(g, X), Y).$$

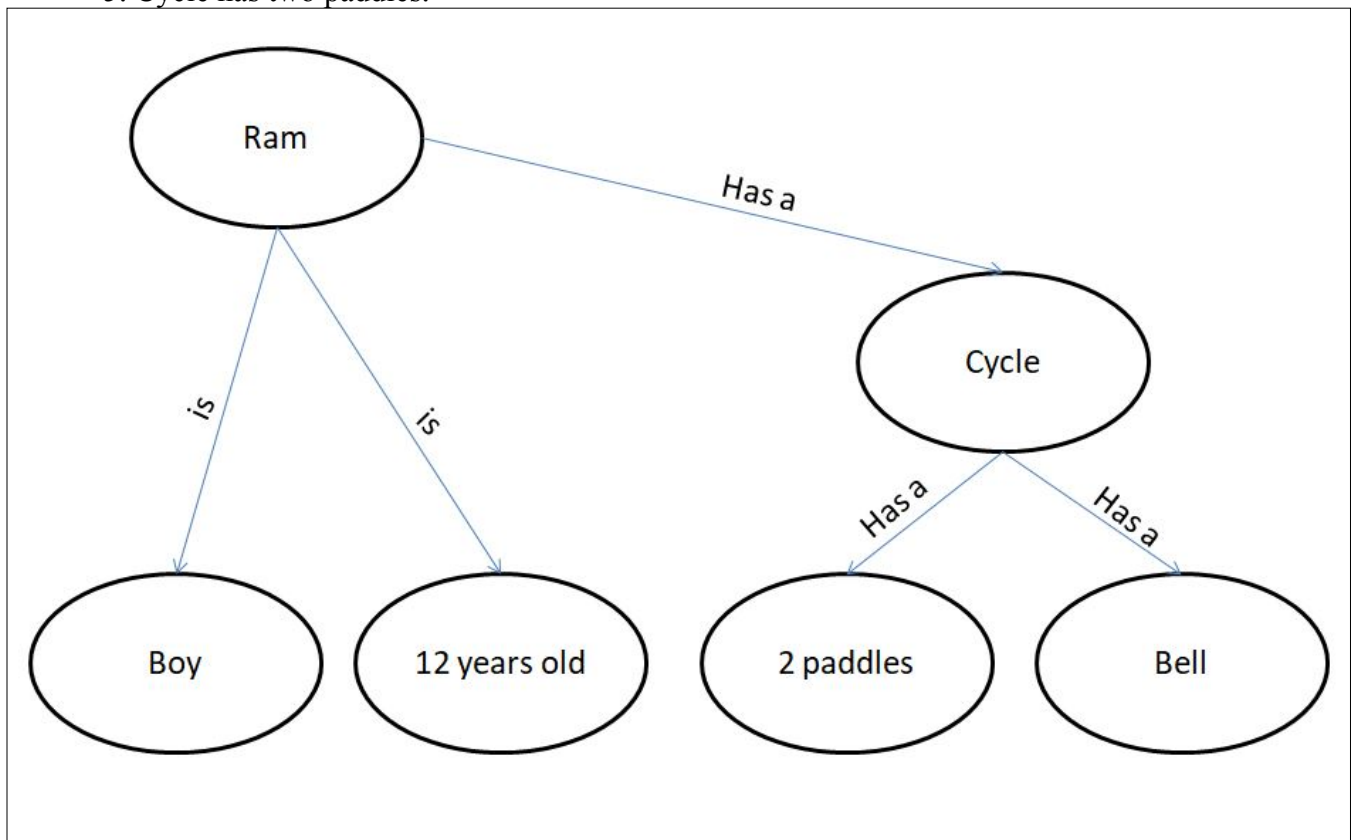
o/p No, these two terms do not unify because arity of $s(g)$ do not match with the arity of $s(g, X)$ due to which rule 3 fails in recursion.

Experiment 8

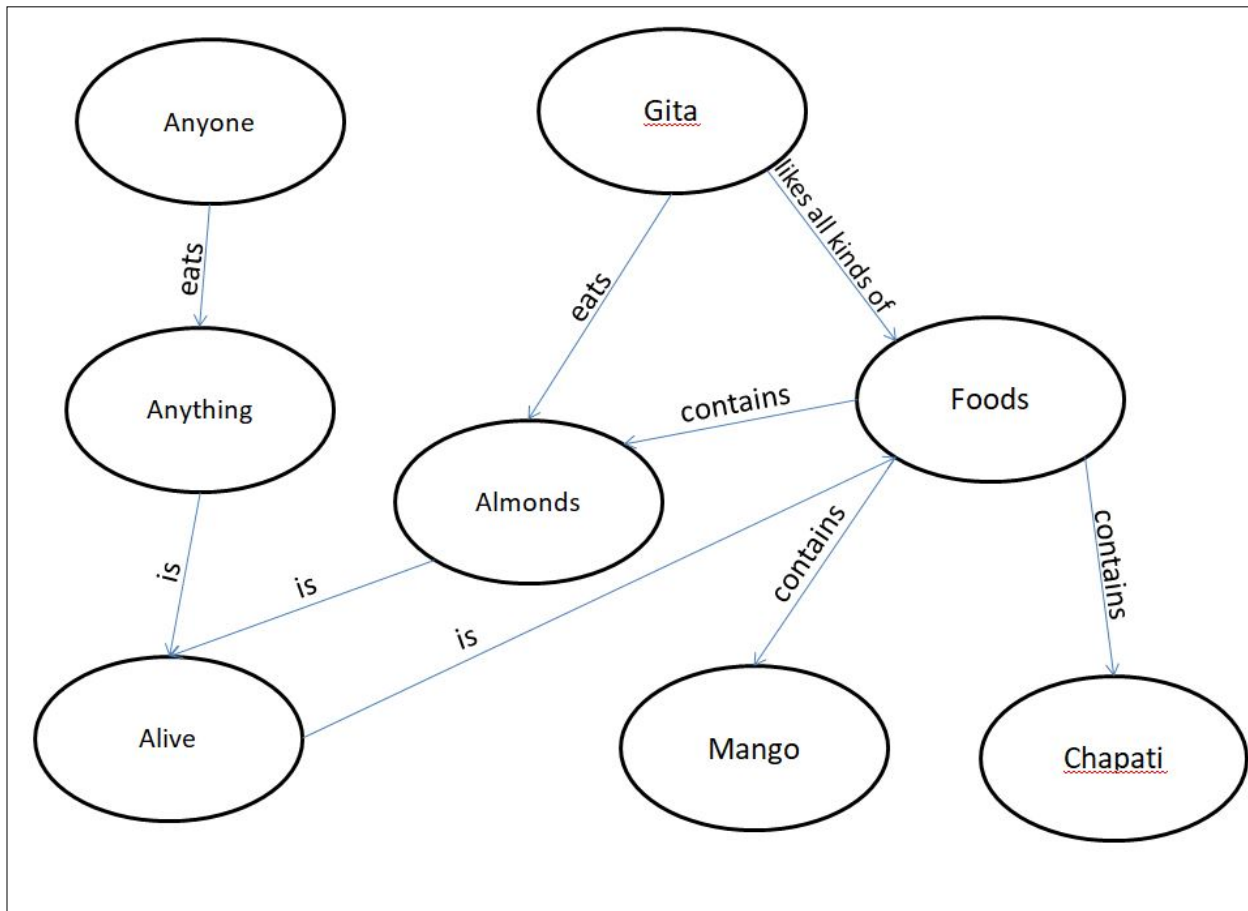
- Aim – Implementation of knowledge representation schemes – use cases.

- Semantic relations –

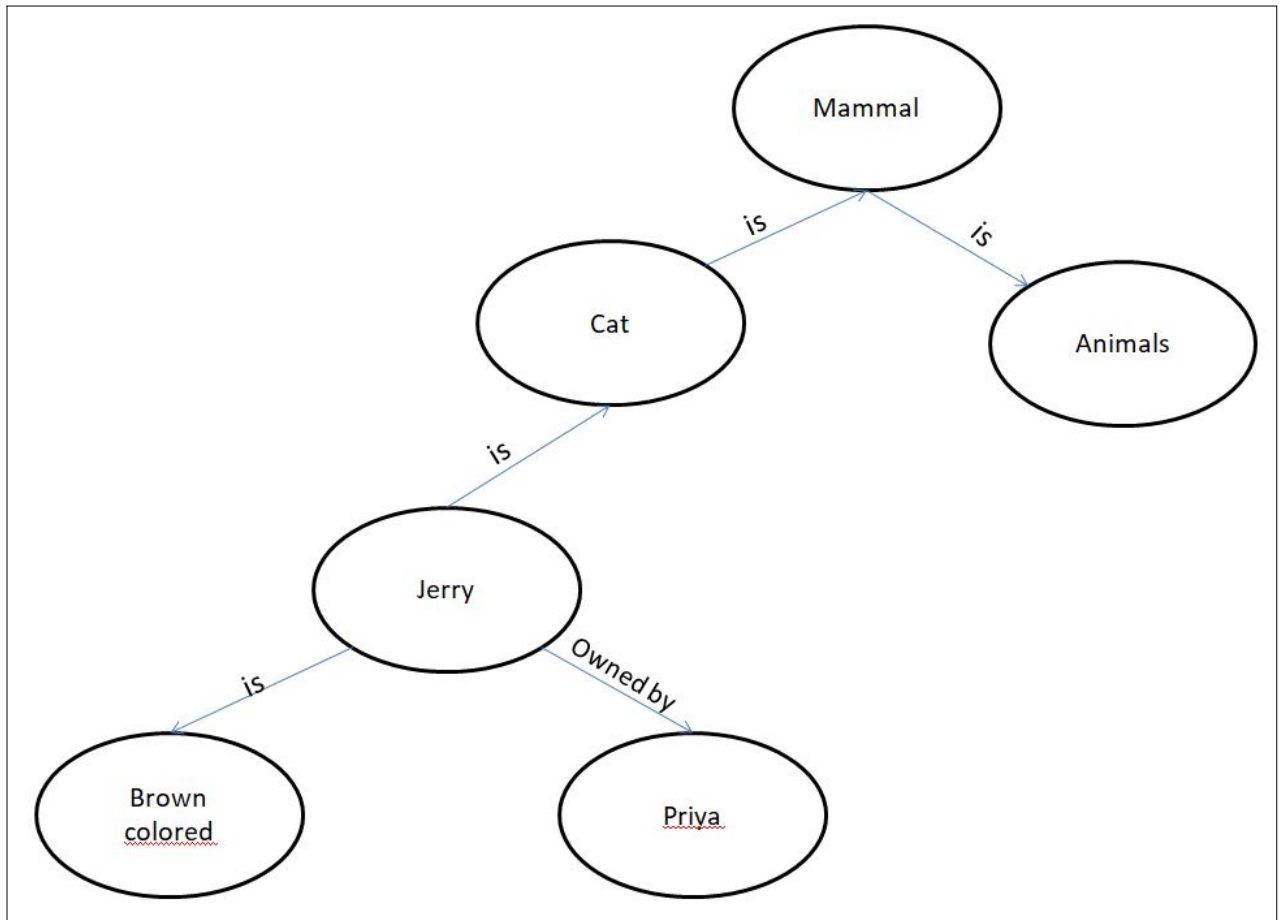
1. Ram has a cycle.
2. Ram is a boy.
3. Cycle has a bell.
4. Ram is 12 years old.
5. Cycle has two paddles.



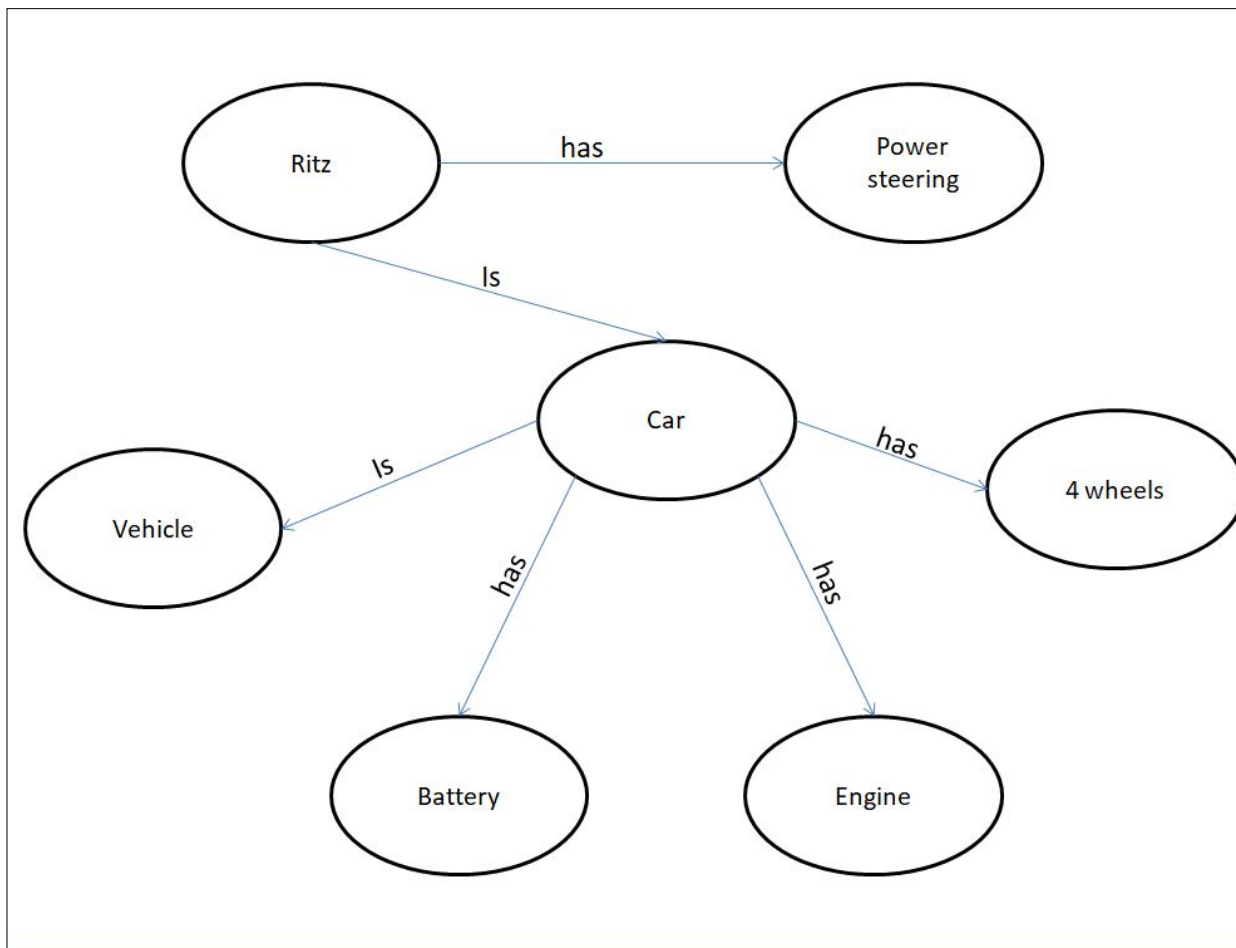
- b. 1. Gita likes all kinds of food.
2. Mango and chapati are food.
3. Gita eats almond and is still alive.
4. Anything eaten by anyone and is still alive is food.



- c. 1. Jerry is a cat.
2. Jerry is a mammal
3. Jerry is owned by Priya.
4. Jerry is brown colored.
5. All Mammals are animal.



- d.
1. Ritz is a car.
 2. Car has 4 wheels.
 3. Car is a vehicle.
 4. Car has engine.
 5. Car has battery.
 6. Ritz has power steering.



Experiment 9

- Aim – Implementation of uncertain methods for an application.

Code – Implementation of Uncertain method for an Application

$$\text{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

we can find the probability of an uncertain event by using the below formula.

Problem1:- Calculate the Probability of finding how many students got the 60 marks for given data set .

```
import numpy as np
```

```
import collections
```

```
npArray= np.array([60, 70, 70, 70, 80,90,60])
```

```
c=collections.Counter(npArray) # Generate a dictionary {"value":"nbOfOccurrences"}
```

```
arraySize=npArray.size
```

```
nbOfOccurrences=c[60] #assuming you want the proba to get 10
```

```
proba=(nbOfOccurrences/arraySize)*100
```

```
print(proba) #print 60.0
```

Problem2:- If In class 80 students and 60 students got 60 % marks then Calculate the Probability of finding how many students got the 60 marks for given data set .

```
#!/usr/bin/env python3

"""reducer.py"""

import sys

# Create a dictionary to map marks

Marksprob = {}

# Get input from stdin

for line in sys.stdin:

    #Remove spaces from beginning and end of the line

    line = line.strip()

    # parse the input from mapper.py

    ClassA, Marks = line.split('\t', 1)

# Create function that returns probability percent rounded to one decimal place

def event_probability(event_outcomes, sample_space):

    probability = (event_outcomes / sample_space) * 100

    return round(probability, 1)
```

```
# Sample Space
```

```
ClassA = 30
```

```
# Determine the probability of drawing a heart
```

```
Marks = 15
```

```
grade_probability = event_probability(Marks, ClassA)
```

```
# Print each probability
```

```
print(str(grade_probability) + '%')
```

Output:- 28.57

- **Result – The program has been executed successfully.**

Experiment 10

- **Aim** – Implementation of block world problem.

- **Algorithm** –

1. MOVE(B,A)- To lift block from B to A.
2. ON(B,A)- To place block B on A.
3. CLEAR(B)- To lift block B from the table.
4. PLACE(B)- To put the block B on table.

- **Code** –

```
class Strips(object):
    def __init__(self, name, preconds, effects, cost=1):
        self.name = name
        self.preconds = preconds
        self.effects = effects
        self.cost = cost
    def __repr__(self):
        return self.name
class STRIPS_domain(object):
    def __init__(self, feats_vals, actions):
        self.feats_vals = feats_vals
        self.actions = actions
class Planning_problem(object):
    def __init__(self, prob_domain, initial_state, goal):
        self.prob_domain = prob_domain
        self.initial_state = initial_state
        self.goal = goal
boolean = {True, False}
### blocks world
def move(x,y,z):
    """string for the 'move' action"""
    return 'move_'+x+'_from_'+y+'_to_'+z
def on(x):
    """string for the 'on' feature"""
    return x+'_is_on'
def clear(x):
    """string for the 'clear' feature"""
    return 'clear_'+x
def create_blocks_world(blocks = {'a','b','c','d'}):
    blocks_and_table = blocks | {'table'}
    stmap = {Strips(move(x,y,z),{on(x):y, clear(x):True, clear(z):True},
    {on(x):z, clear(y):True, clear(z):False})}
    for x in blocks:
        for y in blocks_and_table:
            for z in blocks:
                if x!=y and y!=z and z!=x:
                    stmap.update({Strips(move(x,y,'table'), {on(x):y, clear(x):True},
```



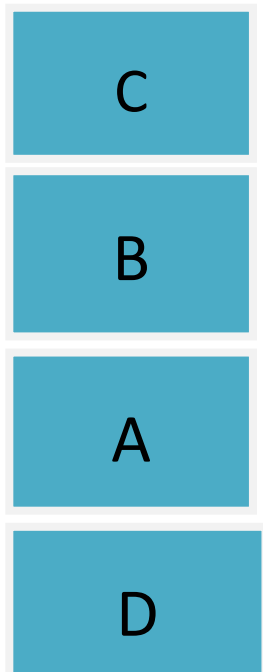
```

        {on(x):'table', clear(y):True}))}
for x in blocks:
    for y in blocks:
        for z in blocks:
            if x!=y:
                feats_vals = {on(x):blocks_and_table-{x} for x in blocks}
                feats_vals.update({clear(x):boolean for x in blocks_and_table})

return STRIPS_domain(feats_vals, stmap)
blocks1dom = create_blocks_world({'a','b','c'})
blocks1 = Planning_problem(blocks1dom,
{on('a'):'table', clear('a'):True,
on('b'):'c', clear('b'):True,
on('c'):'table', clear('c'):False}, # initial state
{on('a'):'b', on('c'):'a'}) #goal
blocks2dom = create_blocks_world({'a','b','c','d'})
tower4 = {clear('a'):True, on('a'):'b',
clear('b'):False, on('b'):'c',
clear('c'):False, on('c'):'d',
clear('d'):False, on('d'):'table'}
blocks2 = Planning_problem(blocks2dom,
tower4, # initial state
{on('d'):'c',on('c'):'b',on('b'):'a'}) #goal
blocks3 = Planning_problem(blocks2dom,
tower4, # initial state
{on('d'):'a', on('a'):'b', on('b'):'c'}) #goal

```

- **Result** – Goal achieved.
- **Output** –



EXP-11

AIM-(Implementation of Learning Algo)

```
# Machine Learning

# Perceptron Algorithm Python function


import numpy as np


def perceptron_single_step_update(

    feature_vector,

    label,

    current_theta,

    current_theta_0):

    theta = current_theta

    theta_0 = current_theta_0

    if label*(np.matmul(current_theta, feature_vector) +
current_theta_0) <= 0:

        theta = theta + label*feature_vector

        theta_0 = theta_0 + label

    return (theta, theta_0)


def perceptron(feature_matrix, labels, T):

    [m,n] = np.shape(feature_matrix)

    tt = np.zeros(n)

    tt_0 = 0
```

```
    for t in range(T):  
        for i in range(m):  
            vec = feature_matrix[i]  
            (tt, tt_0) = perceptron_single_step_update(vec,  
labels[i], tt, tt_0)  
        return (tt, tt_0)
```

AIM-Development of ensemble model

```
# importing utility modules

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error


# importing machine learning models for prediction

from sklearn.ensemble import RandomForestRegressor

import xgboost as xgb

from sklearn.linear_model import LinearRegression


# loading train data set in dataframe from train_data.csv file

df = pd.read_csv("train_data.csv")


# getting target data from the dataframe

target = df["target"]


# getting train data from the dataframe

train = df.drop("target")


# Splitting between train data into training and validation dataset

X_train, X_test, y_train, y_test = train_test_split(
    train, target, test_size=0.20)


# initializing all the model objects with default parameters

model_1 = LinearRegression()
```

```
model_2 = xgb.XGBRegressor()

model_3 = RandomForestRegressor()


# training all the model on the training dataset

model_1.fit(X_train, y_target)
model_2.fit(X_train, y_target)
model_3.fit(X_train, y_target)


# predicting the output on the validation dataset

pred_1 = model_1.predict(X_test)
pred_2 = model_2.predict(X_test)
pred_3 = model_3.predict(X_test)


# final prediction after averaging on the prediction of all 3 models

pred_final = (pred_1+pred_2+pred_3)/3.0


# printing the root mean squared error between real value and predicted
value

print(mean_squared_error(y_test, pred_final))
```