### DEPARTMENT OF NETWORKING AND COMMUNICATION LAB MANUAL

**ACADEMIC YEAR: 2021-22 EVEN SEMESTER**

#### **Programme (UG/PG) : UG**

**Semester : VI**

#### **Course Code : 18CSC305J**

**Course Title : Artificial Intelligence Lab**

#### **Section : H2**

**Year : Third**

***Submitted by***

RAVI MYTRESH(RA1911003011019)

***In partial fulfillment of the requirements for the degree of***

**BACHELOR OF TECHNOLOGY**



### FACULTY OF ENGINEERING AND TECHNOLOGY SRM UNIVERSITY

(Under section 3 of UGC Act, 1956) SRM Nagar, Kattankulathur- 603203 Kancheepuram District

**VISION OF THE INSTITUTE**

To emerge as a World - Class University in creating and disseminating knowledge, and providing students a unique learning experience in Science, Technology, Medicine, Management, and other areas of scholarship that will best serve the world and betterment of mankind

**MISSION OF THE INSTITUTE**

* MOVE UP through international alliances and collaborative initiatives to achieve global excellence.
* ACCOMPLISH A PROCESS to advance knowledge in a rigorous academic and research environment.
* ATTRACT AND BUILD PEOPLE in a rewarding and inspiring environment by fostering freedom, empowerment, creativity and innovation.

**VISION OF THE DEPARTMENT**

To Nurture as a globally recognizable department in imparting students high-quality education and providing high confidence, unique knowledge, and research experience in the field of networking, cyber security, forensics, information technology, cognitive computing, and the internet of things.

**MISSION OF THE DEPARTMENT**

* To provide world-class IT professionals with appropriate industry and research-based curriculum
* To train the students in such a way that leads to entrepreneurship and develops societal need-based industries
* To nourish the students as socially responsible professionals by providing them training in personality development, ethics and leadership program.

**Registration Number**

**LABORATORY RECORD**

###### Course Code:

**Name of the Course:**

###### Programme:

It is certified that this is a Bonafede record of the work carried out by

of class during the year 2021 -2022

Faculty In-Charge HoD

Internal Examiner

Date of the Examination

**LIST OF EXPERIMENTS & SCHEDULE**

###### COURSE CODE:

**COURSE TITLE:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Exp**  **No.** | **Name of the Experiment** | **Page**  **No** | **Date** | **Signature** |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| 6 |  |  |  |  |
| 7 |  |  |  |  |
| 8 |  |  |  |  |
| 9 |  |  |  |  |
| 10 |  |  |  |  |
| 11 |  |  |  |  |

**Faculty In charge HOD**

#### **HARDWARE AND SOFTWARE REQUIREMENTS**

**HARDWARE REQUIREMENTS**

* Thin Client
* Broadband – Internet Connection

**SOFTWARE REQUIREMENT**

* AWS Login ( Cloud9 Service )
* Google Colab

#### **Prepared By**

**Dr. S. Prabakeran**

#### **(Assistant Professor, Department of Networking and Communications)**

**MARK SPLIT UP**

|  |  |  |
| --- | --- | --- |
| 1 | Lab 1: Implementation of toy problems | **CLAP1:5 marks**  3 Exp= 2.5 marks  Viva= 2.5 marks |
| 2 | Lab 2: Developing agent programs for real world problems |
| 3 | Lab 3: Implementation of constraint satisfaction problems |
| 4 | Lab4: Implementation and Analysis of DFS and BFS for same application | **CLAP2:7.5 marks**  4 Exp= 4 marks  Hackerrank= 3.5 marks  (3 medium- 3 marks + 1 difficult – 0.5 mark) |
| 5 | Lab 5: Developing Best first search and A\* Algorithm for real world problems |
| 6 | Lab 6: Implementation of uncertain methods for an application (Fuzzy logic/ Dempster Shafer Theory) |
| 7 | Lab 7: Implementation of unification and resolution for real world problems. |
| 8 | Lab 8: Implementation of learning algorithms |  |
|  | for an application | **CLAP3:7.5 marks** |
| 9 | Lab 9:Implementation of NLP programs | 3 Exp= 5 |
|  |  | marks |
|  |  | Hackerrank= 2.5 marks  (2 difficult + 1 Advanced level) |
| 1  0 | Lab 10: Applying deep learning methods to  solve an application |
|  | **Course Project:**   * Problem statement/objective with technical depth : 2 marks * Execution and Github upload : 2 marks * Purpose of the problem statement (societal benefit) : 1 mark * Team Members : upto 4 max (expected 20-25 projects per faculty member in a section) | **CLAP4: 5 marks** |

**Experiment No:-1**

**Date:-04-01-2022**

**Date : 4-01-2022**

# TOY PROBLEM

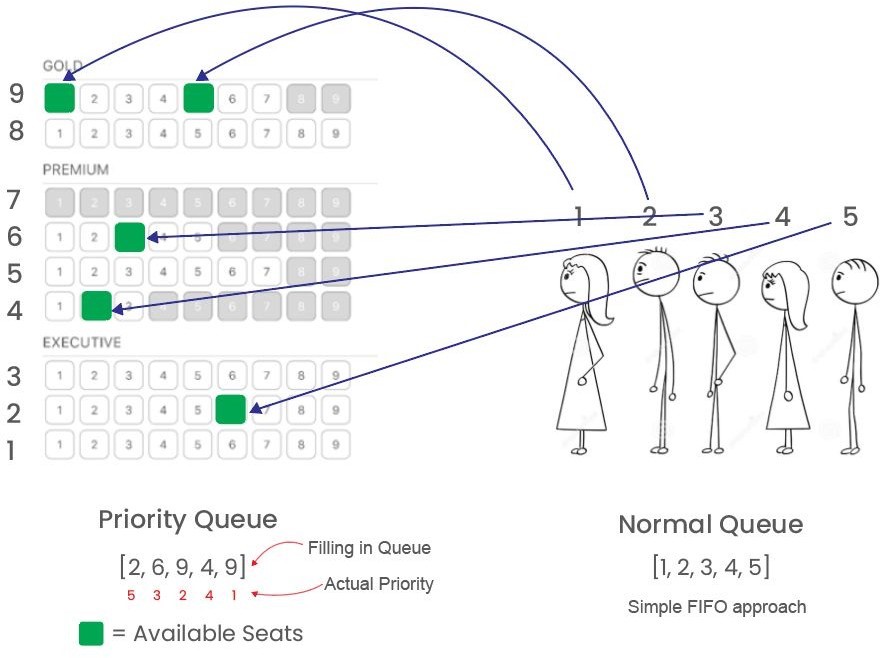
**Problem Statement :** Given an integer N and an array of seats[] where N is the number of people standing in a line to buy a movie ticket and seat[i] is the number of empty seats in the ith row of the movie theater. The task is to find the maximum amount a theater owner can make by selling movie tickets to N people. Price of a ticket is equal to the maximum number of empty seats among all the rows.

**Algorithm :**

1. Initialize queue q insert all seats array elements to the queue.
2. Tickets sold and the amount generated to be set to 0.
3. If tickets sold < N (People in the queue) and q top > 0
4. Then remove top element from queue and update total amount
5. Repeat step 3 and 4 until tickets sold = number of people in the queue.

**Optimization technique :** This problem can be solved by using a priority queue that will store the count of empty seats for every row and the maximum among them will be available at the top.

1. Create an empty priority\_queue q and traverse the seats[] array and insert all elements into the priority\_queue.
2. Initialize two integer variable ticketSold = 0 and ans = 0 that will store the number of tickets sold and the total collection of the amount so far.
3. Now check while ticketSold < N and q.top() > 0 then remove the top element from the priority\_queue and update ans by adding top element of the priority queue. Also store this top value in a variable temp and insert temp – 1 back to the priority\_queue.
4. Repeat these steps until all the people have been sold the tickets and print the final result.



**Tool : jupyter notebook**

**Programming code :**

def maxAmount(M, N, seats):

q = [] for i in range(M):

q.append(seats[i]) ticketSold = 0 ans = 0

q.sort(reverse = True) while (ticketSold < N and q[0] > 0):

ans = ans + q[0] temp = q[0] q = q[1:]

q.append(temp - 1)

q.sort(reverse = True) ticketSold += 1 return ans

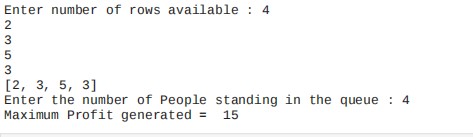
if \_\_name\_\_ == '\_\_main\_\_':

seats = [] rows = int(input("Enter number of rows available : ")) for i in range(0, rows): empty = int(input()) seats.append(empty)

print(seats) M = len(seats)

N = int(input("Enter the number of People standing in the queue : ")) print("Maximum Profit generated = ", maxAmount(N, M, seats))

**Output screen shots :**



**Result :** Successfully found out the maximum amount the theater owner can make by selling movie tickets to N people for a movie.

**Experiment No:-1B**

**Date:- -01-2022**

**TIC TAC TOE PROBLEM**

Problem Statement :

Two players, named ‘player1’ and ‘player2’, play a tic-tac-toe game on a grid of size ‘3 x 3’. Given an array ‘moves’ of size ‘n’, where each element of the array is a tuple of the form (row, column) representing a position on the grid. Players place their characters alternatively in the sequence of positions given in ‘moves’. Consider that ‘player1’ makes the first move. Your task is to return the winner of the game, i.e., the winning player’s name. If there is no winner and some positions remain unmarked, return ‘uncertain’. Otherwise, the game ends in a draw, i.e., when all positions are marked without any winner, return ‘draw’.

## Algorithm :

 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.

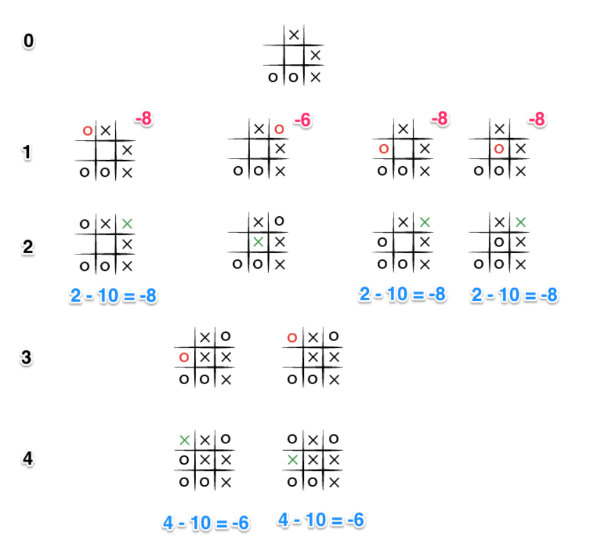
 The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).

 If no one wins, then the game is said to be draw.

**Optimization technique :** The key is to use Minimax algorithm .A back and forth between the two players, where the player whose "turn it is" desires to pick the move with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the scores for the opposing players moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state.

A description for the algorithm, assuming X is the "turn taking player,"

* If the game is over, return the score from X's perspective.
* Otherwise get a list of new game states for every possible move
* Create a scores list
* For each of these states add the minimax result of that state to the scores list
* If it's X's turn, return the maximum score from the scores list
* If it's O's turn, return the minimum score from the scores list



**Tool :** VS Code and Python 3.9.0

## Programming code :

## import random

## import copy as cp

## # NOTE use cp.deepcopy() so the temp variable isn't linked with the other

## class Cell:

## def \_\_init\_\_(self, position, location, max\_val, min\_val):

## self.position = position

## self.location = location # NOTE this is a list, [0] is row info and [1] is col info

## self.min\_val = min\_val

## self.max\_val = max\_val

## def generate\_cells(board):

## uboard = cp.deepcopy(board)

## for i in range(len(uboard)):

## for j in range(len(uboard[i])):

## if uboard[i][j] != 'X' and uboard[i][j] != 'O':

## uboard[i][j] = Cell(uboard[i][j], [i,j], 0, 0)

## uboard[i][j].max\_val = max\_val(board, [i, j])

## uboard[i][j].min\_val = min\_val(board, [i, j])

## # NOTE convert uboard[i][j] into list of maxval and minval from objects

## uboard[i][j] = [uboard[i][j].position, uboard[i][j].max\_val, uboard[i][j].min\_val]

## return uboard

## def max\_val(board, location): # NOTE only generates one max\_val by a given location, not entire board

## maxval = 0

## if board[location[0]][location[1]] != 'O' and board[location[0]][location[1]] != 'X':

## maxval += check\_horizontal(board, location[0], 'max') # need row

## maxval += check\_vertical(board, location[1], 'max') # need column

## # NOTE diagonal check is splitted into left and right diagonal for convienence

## maxval += left\_diagonal(board, location[0], location[1], 'max')

## maxval += right\_diagonal(board, location[0], location[1], 'max')

## return maxval

## def min\_val(board, location):

## minval = 0

## if board[location[0]][location[1]] != 'O' and board[location[0]][location[1]] != 'X':

## minval -= check\_horizontal(board, location[0], 'min')

## minval -= check\_vertical(board, location[1], 'min')

## minval -= left\_diagonal(board, location[0], location[1], 'min')

## minval -= right\_diagonal(board, location[0], location[1], 'min')

## return minval

## def check\_horizontal(board, row, u\_type):

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## for i in range(3): # 3 == len(board)'s row

## if board[row][i] != opposed:

## unfilled += 1

## if board[row][i] == sign:

## v += 1

## if unfilled == 3:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def check\_vertical(board, col, u\_type):

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## for i in range(3): # 3 == len(board)'s column

## if board[i][col] != opposed:

## unfilled += 1

## if board[i][col] == sign:

## v += 1

## if unfilled == 3:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def left\_diagonal(board, row, col, u\_type): # NOTE top\_left to bottom\_right diagonal check

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## if row == col:

## for i in range(3):

## if board[i][i] != opposed:

## unfilled += 1

## if board[i][i] == sign:

## v += 1

## if unfilled == 3:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def right\_diagonal(board, row, col, u\_type):

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## state = False

## for i in range(len(board)):

## if board[i][abs(i-2)] == board[row][col]:

## state = True

## if board[i][abs(i-2)] != opposed:

## unfilled += 1

## if board[i][abs(i-2)] == sign:

## v +=1

## if unfilled == 3 and state == True:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def dispUboard(uboard):

## print('\n')

## count = 0

## print("Utility Board:\n")

## for i in range(len(uboard)):

## for j in range(len(uboard[i])):

## count += 1

## if uboard[i][j] == 'O' or uboard[i][j] == 'X':

## print(' ',uboard[i][j],end=' ')

## else:

## print(uboard[i][j],end=' ')

## if count%3 == 0:

## print('\n')

## def checkWin(board, sign):

## if checkHorizontal(board, sign) == True:

## return True

## if checkVertical(board, sign) == True:

## return True

## if checkDiagonal(board, sign) == True:

## return True

## return False

## def checkTie(board):

## filled = 0

## for i in range(len(board)):

## for j in range(len(board[i])):

## if board[i][j] == 'O' or board[i][j] == 'X':

## filled += 1

## if filled == 9:

## return True

## return False

## def checkDiagonal(board, sign):

## for i in range(len(board)):

## filled = 0

## if board[0][0] == sign:

## for j in range(len(board[i])):

## if board[j][j] == sign:

## filled += 1

## elif board[0][2] == sign:

## for j in range(len(board[i])):

## if board[0+j][2-j] == sign:

## filled += 1

## if filled == 3:

## return True

## return False

## def checkHorizontal(board, sign): # NOTE BUGGY so fix it

## for i in range(len(board)):

## if board[i][0] == sign:

## filled = 0

## for j in range(len(board[i])):

## if board[i][j] == sign:

## filled += 1

## if filled == 3:

## return True

## return False

## def checkVertical(board, sign):

## for i in range(len(board)):

## if board[0][i] == sign:

## filled = 0

## for j in range(len(board[i])):

## if board[j][i] == sign:

## filled += 1

## if filled == 3:

## return True

## return False

## def dispboard(board):

## print('\n')

## count = 0

## print('Tictactoe Board:\n')

## for i in range(len(board)):

## for j in range(len(board[i])):

## count += 1

## print(board[i][j],end=' ')

## if count%3 == 0:

## print('\n')

## def checkCompatible(board, move, sign):

## i = 2

## if move <= 2:

## i = 0

## elif move >= 3 and move <= 5:

## i = 1

## loc = [i,(move-(i\*3))]

## if board[loc[0]][loc[1]] == move:

## board[loc[0]][loc[1]] = sign

## return True

## else:

## print("Please select an empty spot and try again.")

## return False

## def computerDecision(board):

## while (checkTie(board) == False) and (checkWin(board, 'X') == False):

## uboard = generate\_cells(board)

## dispUboard(uboard)

## dispboard(board)

## # TODO run minimax algorithm here

## computer\_decision = minimax\_algorithm(uboard)

## computer\_decision = int(computer\_decision)

## if checkCompatible(board, computer\_decision, 'O') == True:

## if checkTie(board) == True:

## dispboard(board)

## play\_again = input("\nThis is a tie game, to play again enter any key, otherwise enter 'q' to quit.\nYour decision: ")

## if play\_again == 'q':

## return

## else:

## board = [[0, 1, 2],[3, 4, 5],[6, 7, 8]]

## GameInitializer(board)

## elif checkWin(board, 'O') == True:

## dispboard(board)

## print("The computer won!")

## return

## else:

## playerDecision(board)

## else:

## computerDecision(board)

## def playerDecision(board):

## while (checkTie(board) == False) and (checkWin(board, 'O') == False):

## dispboard(board)

## player\_decision = input("\n(The player's turn) Enter the empty position you want to place your 'X': ")

## player\_decision = int(player\_decision)

## if checkCompatible(board, player\_decision, 'X') == True:

## if checkTie(board) == True:

## dispboard(board)

## play\_again = input("\nThis is a tie game, if you want to play again enter 'p', to quit enter any key.\nYour decision: ")

## if play\_again == 'q':

## return

## else:

## board = [[0, 1, 2],[3, 4, 5],[6, 7, 8]]

## GameInitializer(board)

## elif checkWin(board, 'X') == True:

## dispboard(board)

## print("The player won!")

## return

## else:

## computerDecision(board)

## else:

## playerDecision(board)

## def GameInitializer(board):

## choice = input("\nDo you want to go first or the computer goes first?\nEnter 'c' for computer first, or 'p' if you would like to go first\nYour Choice: ")

## if choice == 'c':

## computerDecision(board)

## elif choice == 'p':

## playerDecision(board)

## else:

## print("\nPlease enter 'c' or 'p' and try again.")

## GameInitializer(board)

## def minimax\_algorithm(ub): # should return a pos, such as 4, not index[1,1]

## optimal = 0

## options = []

## redundant\_optimal = [] # This adds the random feature for the computer decision.

## for i in range(len(ub)):

## for j in range(len(ub[i])):

## if ub[i][j] != 'X' and ub[i][j] != 'O':

## # NOTE uboard[i][j's 0 is position, 1 is maxval, 2 is minval

## if ub[i][j][1] >= 10:

## return ub[i][j][0]

## elif ub[i][j][2] <= -10:

## return ub[i][j][0]

## else:

## if abs(ub[i][j][1]) == abs(ub[i][j][2]):

## # NOTE if abs of max = abs of min, add 1 to their sum. Why? because we want to win more more than limiting the enemy

## options.append([abs(ub[i][j][1]) + abs(ub[i][j][2])+1, ub[i][j][0]])

## else: # NOTE, [0] is the total val of abs(max + min). [1] is the index

## options.append([abs(ub[i][j][1]) + abs(ub[i][j][2]), ub[i][j][0]])

## optimal = max(options) # NOTE for redundant\_optimal, [0] is index, [1] is val

## for i in range(len(options)):

## if options[i][0] == optimal[0]:

## redundant\_optimal.append(options[i][1])

## redundant\_optimal.append(optimal[1])

## randnum = random.randint(0,len(redundant\_optimal)-1)

## return redundant\_optimal[randnum]

## # NOTE play game here

## init\_board = [[0, 1, 2],

## [3, 4, 5],

## [6, 7, 8]]

## GameInitializer(init\_board)

## Output:-

## 1.png2.png3.png4.png5.png6.png7.png8.png9.png

**Result :**  The Tic Tac Toe problem was implemented successfully using minmax algorithm to evaluate the best moves with the highest score.

**Experiment No:-2**

**Date:-24-01-2022**

**GRAPH COLORING PROBLEM**

**Problem Statement: graph coloring** is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problem is ***Edge Coloring*** (No vertex is incident to two edges of same color).

## Algorithm :

1.Color first vertex with first color.

2. Do following for remaining V-1 vertices.

a) Consider the currently picked vertex and color it with the   
 lowest numbered color that has not been used on any previously colored vertices adjacent to it.

b) If all previously used colors appear on vertices adjacent to v, assign a new color to it.

**Optimization technique :** The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.

**Algorithm:**

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and return true.
6. If no recursive function returns true then return false.

**Tool :** Cloud9 ide and Python 3.9.0

## Programming code :

class Graph:

def \_init\_ (self, edges, N):

self.adj = [[] for \_ in range(N)]

for (src, dest) in edges:

self.adj[src].append(dest)

self.adj[dest].append(src)

def colorGraph(graph):

result = {}

for u in range(N):

assigned = set([result.get(i) for i in graph.adj[u] if i in result])

color = 1

for c in assigned:

if color != c:

break

color = color + 1

result[u] = color

for v in range(N):

print("color assigned to vertex", v, "is",colors[result[v]])

print("\n")

for v in range(N):

print("color assigned to Edge", v, "is",colors[result[v]+3])

if \_name\_ == '\_main\_':

colors= ["", "YELLOW", "RED", "BLUE", "ORANGE", "GREEN", "PINK", "BLACK", "BROWN", "WHITE", "PURPLE", "VIOLET"]

edges=[(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,0)]

N = 7

graph = Graph(edges, N)

colorGraph(graph)

## Output screen shots:

## 

## 

**Result :** A unique color was successfully assigned to each vertex and edge of the graph.

**Experiment No:-3**

**Date:-01-02-2022**

## Implementation of constraint satisfaction problem (Cryptarithmetic Problem )

## Algorithm :

 First, create a list of all the characters that need assigning to pass to Solve

 If all characters are assigned, return true if puzzle is solved, false otherwise

 Otherwise, consider the first unassigned character

 for (every possible choice among the digits not in use)

make that choice and then recursively try to assign the rest of the characters  
if recursion successful, return true  
if !successful, unmake assignment and try another digit

**Optimization technique :** The algorithm above actually has a lot in common with the permutations algorithm, it pretty much just creates all arrangements of the mapping from characters to digits and tries each until one works or all have been successfully tried. For a large puzzle, this could take a while.  
A smarter algorithm could take into account the structure of the puzzle and avoid going down dead-end paths. For example, if we assign the characters starting from the one’s place and moving to the left, at each stage, we can verify the correctness of what we have so far before we continue onwards. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

* Start by examining the rightmost digit of the topmost row, with a carry of 0
* If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
* If we are currently trying to assign a char in one of the addends  
  If char already assigned, just recur on the row beneath this one, adding value into the sum  
  If not assigned, then
  + for (every possible choice among the digits not in use)  
    make that choice and then on row beneath this one, if successful, return true  
    if !successful, unmake assignment and try another digit
  + return false if no assignment worked to trigger backtracking
* Else if trying to assign a char in the sum
* If char assigned & matches correct,  
  recur on next column to the left with carry, if success return true,
* If char assigned & doesn’t match, return false
* If char unassigned & correct digit already used, return false
* If char unassigned & correct digit unused,  
  assign it and recur on next column to left with carry, if success return true
* return false to trigger backtracking.

**Tool :** aws cloud9 and Python 3.9.0

## Programming code :

import itertools

import pdb

def get\_val(word, substitution):

s = 0

factor = 1

for let in reversed(word):

s += factor \* substitution[let]

factor \*= 10

return s

def solve(equation):

l, r = equation.lower().replace(' ', '').split('=')

print(l,r)

l = l.split('+')

print(l)

lets = set(r)

print(lets)

for word in l:

for let in word:

lets.add(let)

lets = list(lets)

print(lets)

digits = range(20)

for perm in itertools.permutations(digits, len(lets)):

sol = dict(zip(lets, perm))

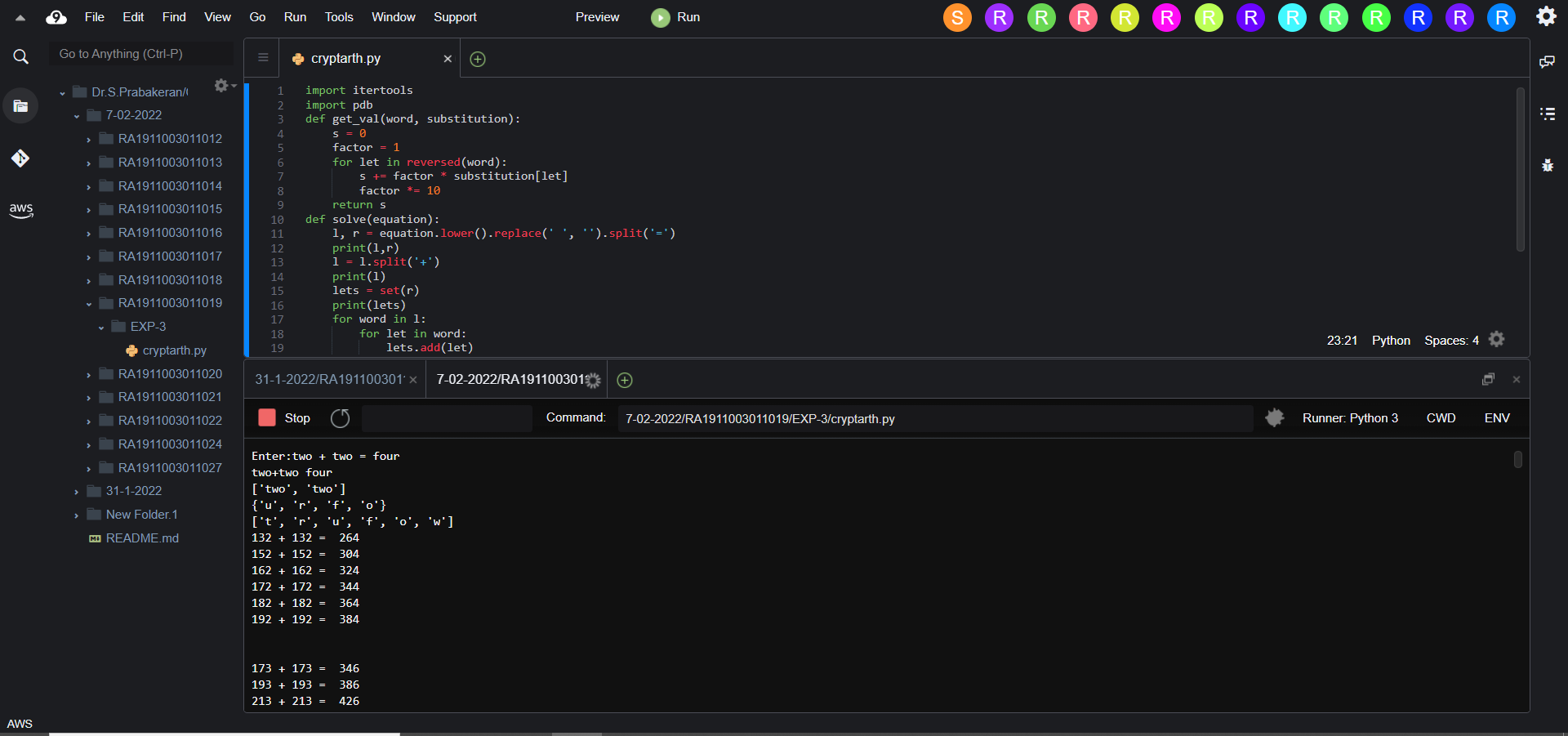
if sum(get\_val(word, sol) for word in l) == get\_val(r, sol):

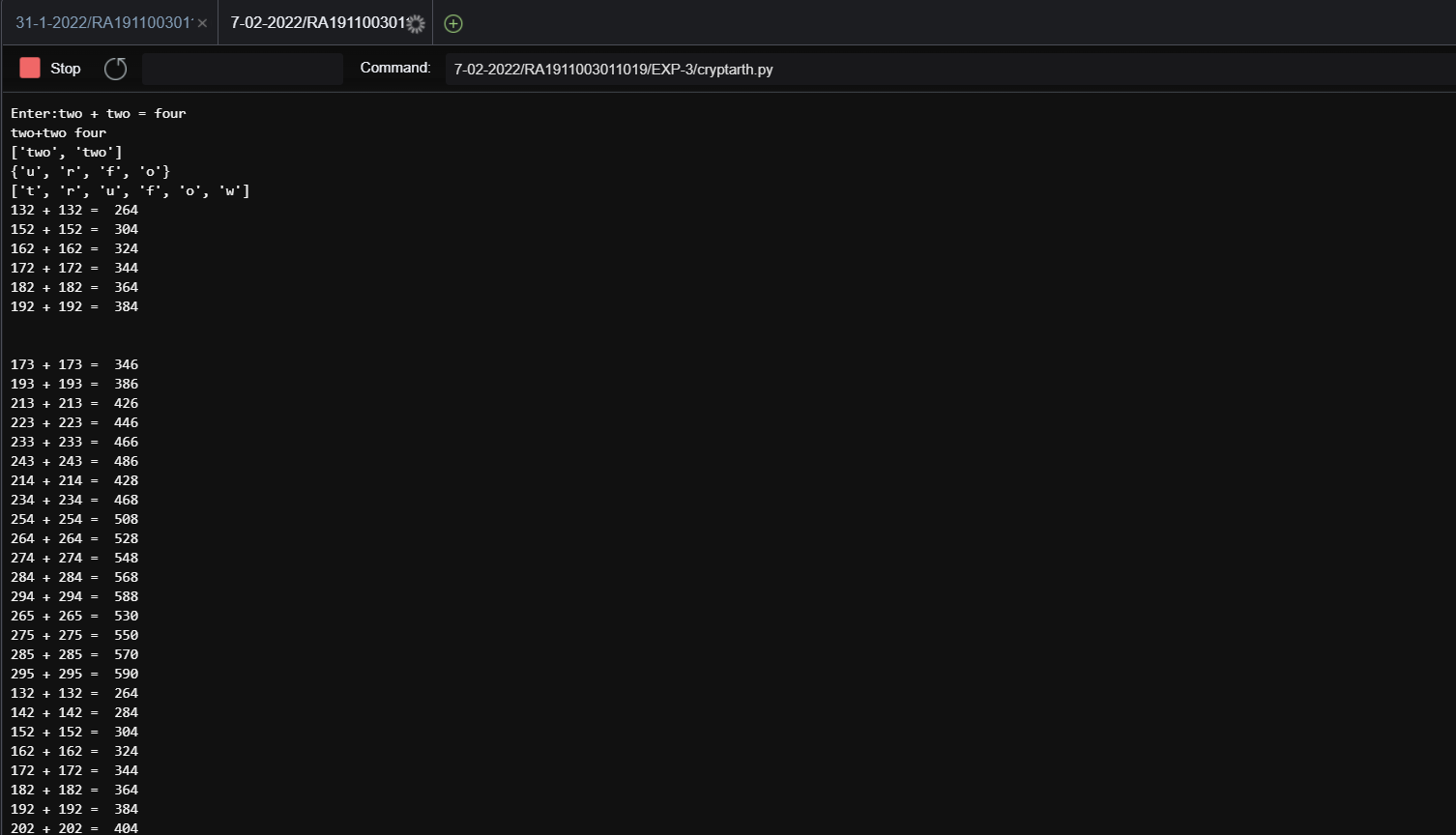
print(' + '.join(str(get\_val(word, sol)) for word in l) + " = ",get\_val(r, sol))

equation = input("Enter:")

solve(equation)

## Output screen shots :

****

****

**Result :** Successfully solved the given constraint satisfaction problem.

**Experiment No:-4**

**Date:-08-02-2022**

**Implementation And Analysis Of DFS And BFS**

**Breadth-First Search :** Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

## Algorithm :

1. Start by putting any one of the graph’s vertices at the back of the queue.
2. Now take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
4. Keep continuing steps two and three till the queue is empty.

**The Depth-First Search : The Depth-First Search is a recursive algorithm that uses the concept of backtracking.** It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

**Algorithm :**

1. We will start by putting any one of the graph's vertex on top of the stack.
2. After that take the top item of the stack and add it to the visited list of the vertex.
3. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
4. Lastly, keep repeating steps 2 and 3 until the stack is empty.

**Tool :** Aws Cloud9 and Python 3.9.0

## Programming code :

## BFS:

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] *# List for visited nodes.*

queue = [] *#Initialize a queue*

**def**bfs(visited, graph, node): *#function for BFS*

visited.append(node)

queue.append(node)

**while** queue: *# Creating loop to visit each node*

m = queue.pop(0)

**print** (m, end =" ")

**for** neighbour **in** graph[m]:

**if** neighbour **notin** visited:

visited.append(neighbour)

queue.append(neighbour)

*# Driver Code*

**print**("Following is the Breadth-First Search")

bfs(visited, graph, '5') *# function calling*

**DFS:**

# Using a Python dictionary to act as an adjacency list

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited =set() # Set to keep track of visited nodes of graph.

**defdfs**(visited, graph, node): #function for dfs

**if** node **notin** visited:

**print** (node)

visited.add(node)

**for** neighbour **in** graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

**print**("Following is the Depth-First Search")

dfs(visited, graph, '5')

## Output screen shots :

## 

## 

**Result :** Successfully Implemented BFS and DFS.

**Experiment No:-5**

**Date:-15-02-2022**

# Best First Search (Informed Search)

# 

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both

BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to Priority Queue.

**Algorithm :**

1) Create an empty PriorityQueue

PriorityQueue **pq**; 2) Insert "start" in pq.

pq.insert(start)

3) Until PriorityQueue is empty u = PriorityQueue.DeleteMin

If u is the goal

Exit

Else

Foreach neighbor v of u

If v "Unvisited"

Mark v "Visited" pq.insert(v)

Mark u "Examined"

End procedure

**A\* Algorithm**

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost

**Algorithm**

* Make an open list containing starting node o If it reaches the destination node : o Make a closed empty list
  + If it does not reach the destination node, then consider a node with the lowest f-score in the open list

We are finished

* Else :

Put the current node in the list and check its neighbors

* For each neighbor of the current node :
  + If the neighbor has a lower g value than the current node and is in the closed list:

Replace neighbor with this new node as the neighbor’s parent

* Else If (current g is lower and neighbor is in the open list):

Replace neighbor with the lower g value and change the neighbor’s parent to the current node.

* Else If the neighbor is not in both lists:

Add it to the open list and set its g

**Tool :** VS Code and Python 3.9.0

**Programming code :**

# A-star

# graph class class Graph:

# init 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 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 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 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)

# node class class Node:

# init 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.name, self.f))

# A\* search def astar\_search(graph, heuristics, start, end):

# lists for open nodes and closed nodes

open = []

closed = []

# a start node and an goal node start\_node = Node(start, None) goal\_node = Node(end, None)

# add start node

open.append(start\_node)

# loop until the open list is empty while len(open) > 0:

open.sort() # sort open list to get the node with the lowest cost first

current\_node = open.pop(0) # get node with the lowest cost closed.append(current\_node) # add current node to the closed list

# 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 path[::-1]

neighbors = graph.get(current\_node.name) # get neighbours

# loop neighbors for key, value in neighbors.items():

neighbor = Node(key, current\_node) # create neighbor node if(neighbor in closed): # check if the neighbor is in the closed list continue

# calculate full path cost

neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name) neighbor.h = heuristics.get(neighbor.name) neighbor.f = neighbor.g + 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

# create a graph graph = Graph() # create graph connections (Actual distance)

graph.connect('S', 'A', 2) graph.connect('S', 'G', 20) graph.connect('A', 'C', 7) graph.connect('C', 'G', 8) graph.connect('C', 'D', 9) graph.connect('D', 'G', 10)

# make graph undirected, create symmetric connections graph.make\_undirected()

# create heuristics (straight-line distance, air-travel distance) heuristics = {} heuristics['A'] = 5 heuristics['C'] = 8 heuristics['G'] = 7 heuristics['D'] = 6 heuristics['S'] = 9 # run the search algorithm path = astar\_search(graph, heuristics, 'S', 'G') print("Path:", path)

# Best First Search

from queue import PriorityQueue v = 5 graph = [[] for i in range(v)] def best\_first\_search(source, target, n):

visited = [0] \* n visited[0] = True pq = PriorityQueue() pq.put((0, source)) while pq.empty() == False:

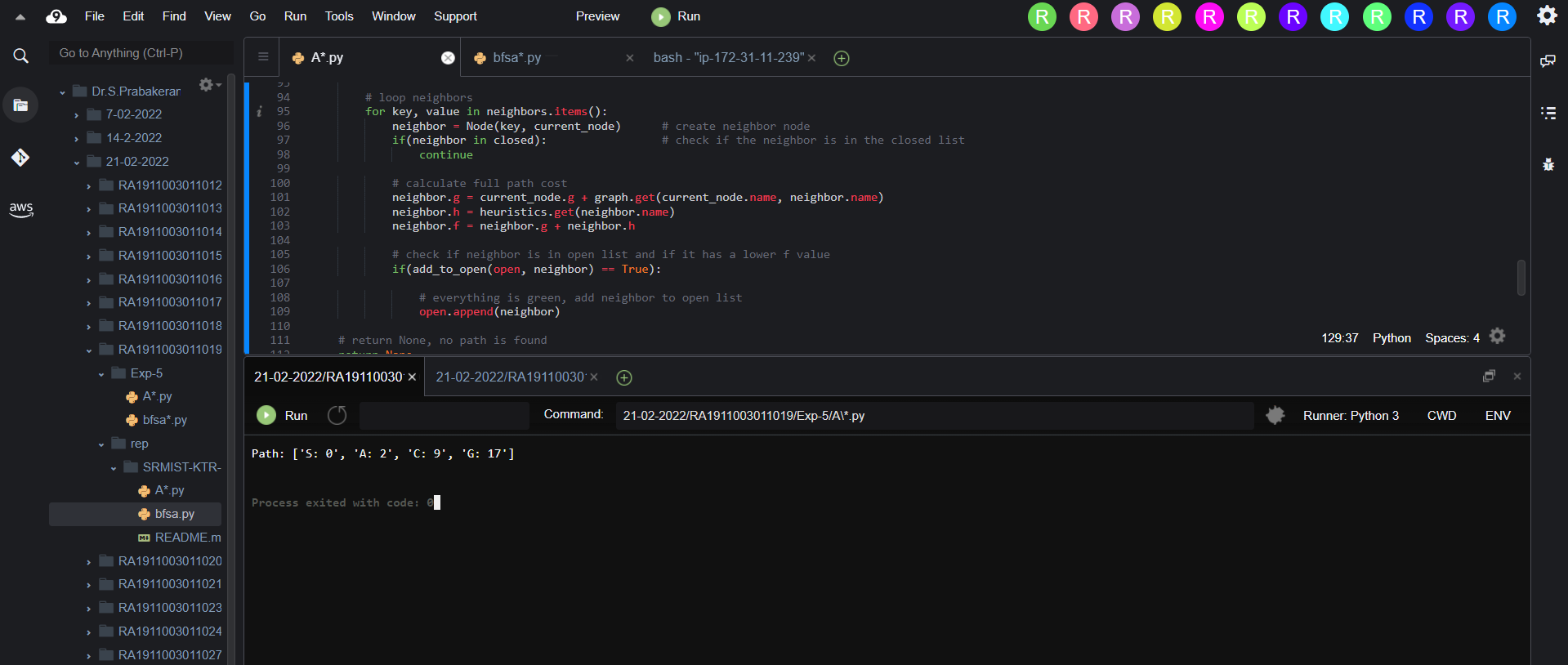
u = pq.get()[1] print(u, end=" ") if u == target:

break for v, c in graph[u]: if visited[v] == False: visited[v] = True pq.put((c, v)) print() def addedge(x, y, cost): graph[x].append((y, cost)) graph[y].append((x, cost)) addedge(0, 1, 5) addedge(0, 2, 1) addedge(2, 3, 2) addedge(1, 4, 1) addedge(3, 4, 2) source = 0 target = 4

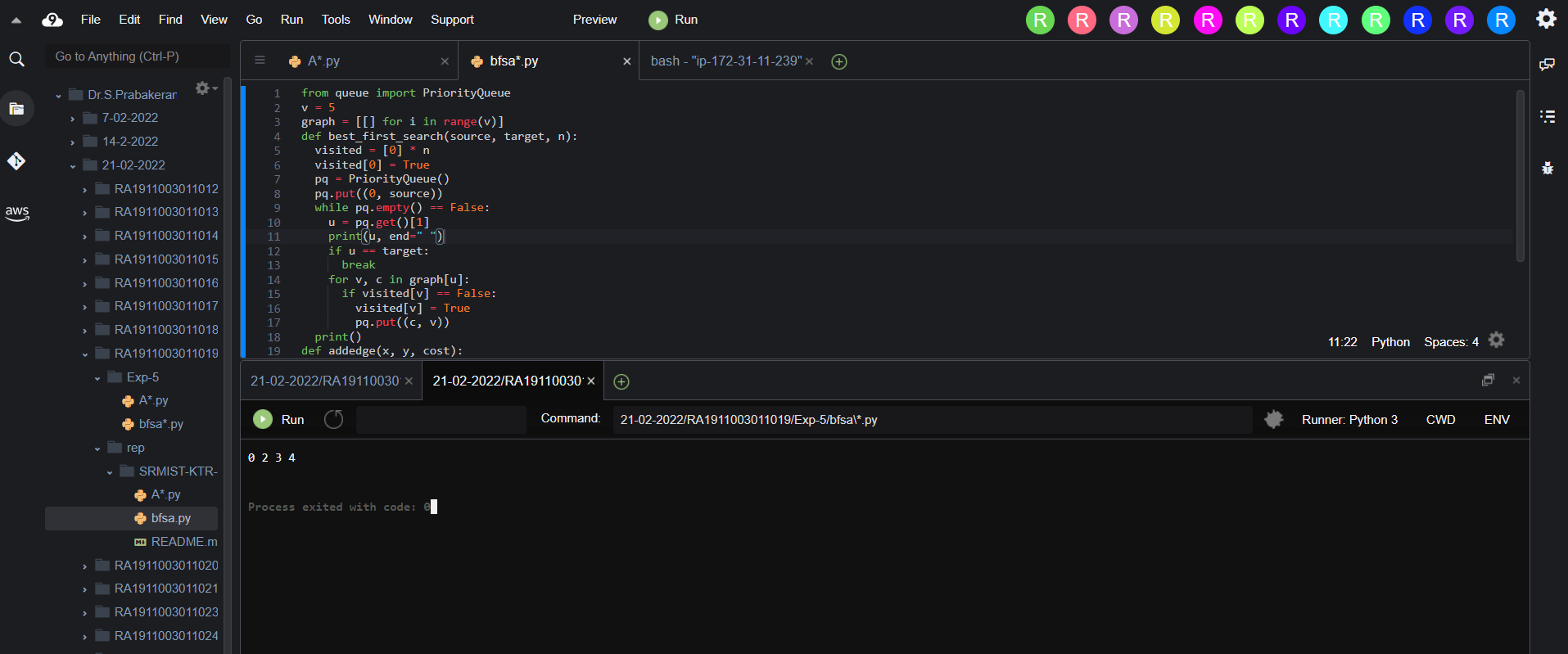
best\_first\_search(source, target, v)

**Output screen shots :**

**A\***



**BFSA :**



**Result :** A\* and Best first search algorithms were implemented successfully.

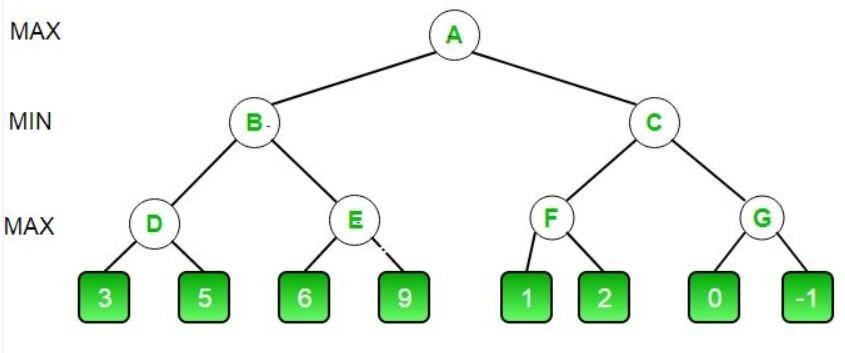
**Experiment No: 6**

**Date : 22-02-2022**

**MINIMAX ALGORITHM**

**AIM :** Developing a mini max algorithm for real world problems.

**PROBLEM :** Find the optimal value in the given tree of integer values in the most optimal way possible under the time complexity O(B^D).



**ALGORITHM MINIMAX APPROACH :**

1. Start traversing the given tree in top to bottom manner.

1. If node is a leaf node then return the value of the node.

1. If isMaximizingPlayer exist then bestVal = -INFINITY

1. For each child node, value = minimax(node, depth+1, false, alpha, beta)

1. bestVal = max( bestVal, value) and alpha = max( alpha, bestVal)

1. If beta <= alpha then stop traversing and return bestVal

1. Else, bestVal = +INFINITY

1. For each child node, value = minimax(node, depth+1, true, alpha, beta)
2. bestVal = min( bestVal, value) and beta = min( beta, bestVal)

1. if beta <= alpha the stop traversing and return bestVal

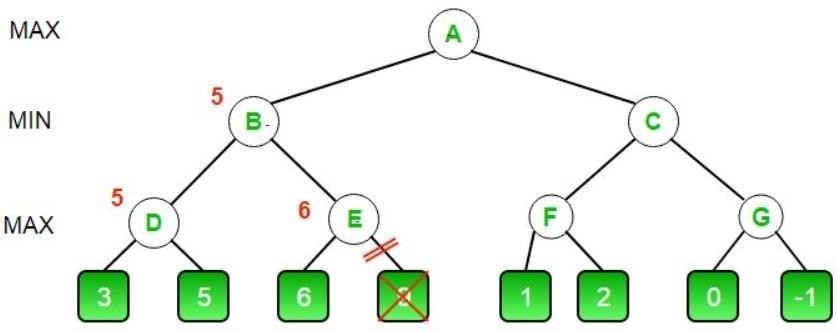
**OPTIMIZATION TECHNIQUE :**

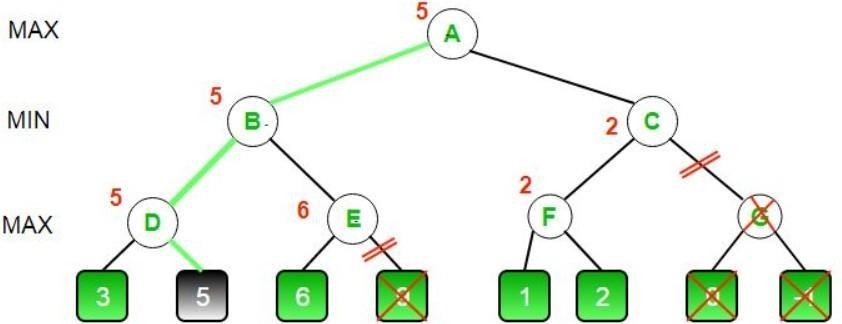
Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithms. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let’s define the parameters alpha and beta.

**Alpha** is the best value that the **maximizer** currently can guarantee at that level or above.

**Beta** is the best value that the **minimizer** currently can guarantee at that level or above.





**CODE (MINIMAX ALGORITHM) :**

MAX, MIN = 1000, -1000 def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

False, values, alpha, beta) best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = MAX for i in range(0, 2): val = minimax(depth + 1, nodeIndex \* 2 + i,

True, values, alpha, beta) best = min(best, val) beta = min(beta, best) if beta <= alpha: break

return best

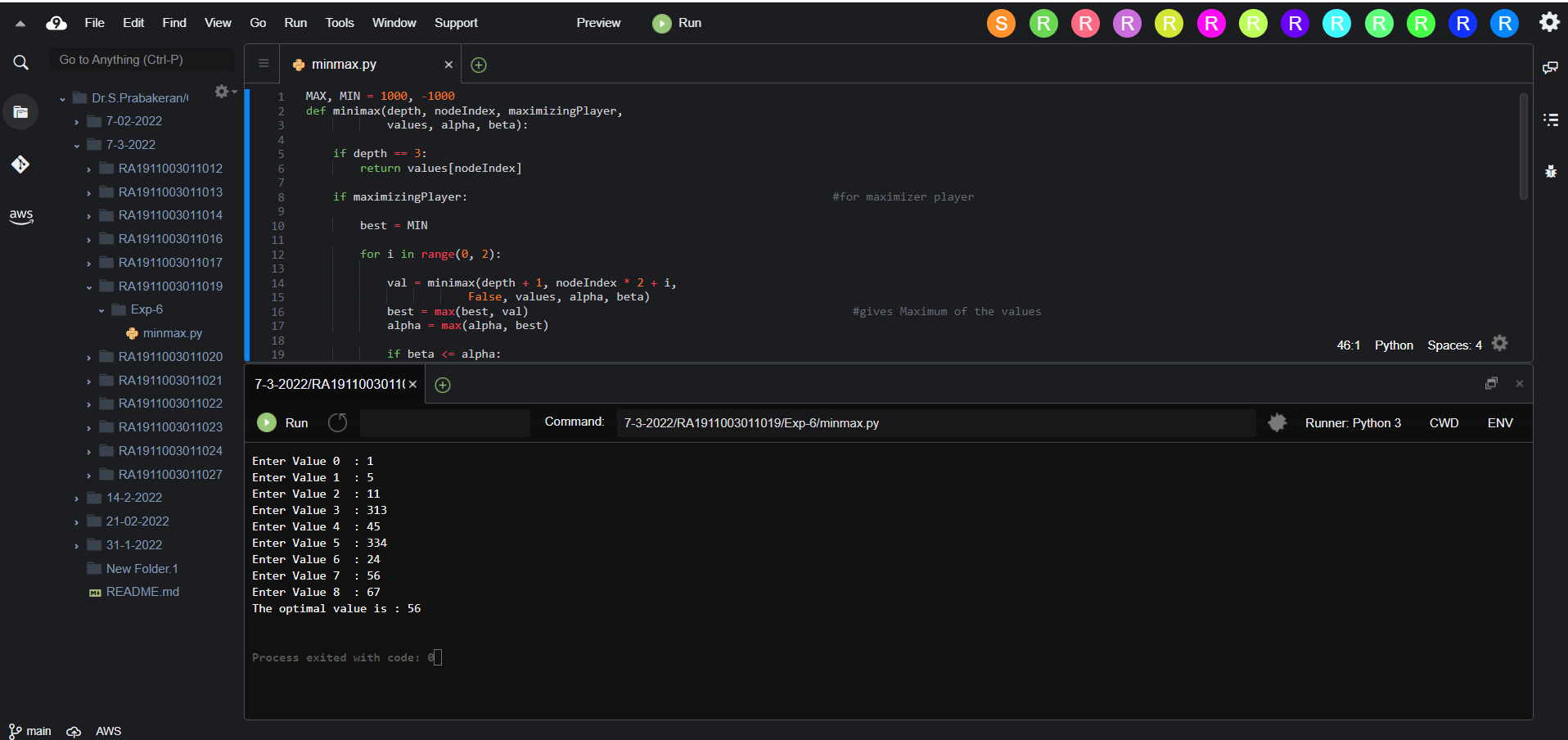
if \_\_name\_\_ == "\_\_main\_\_":

values = [] for i in range(0, 8):

x = int(input(f"Enter Value {i} : ")) values.append(x)

print ("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

**OUTPUT :**



**RESULT :** The Optimal value of the given tree successfully found using Minimax Algorithm with Alpha Beta Pruning in time complexity O(B^D).

**Experiment No : 7**

**Date:- 14-03-2022**

**IMPLEMENTATION OF UNCERTAIN METHODS OF AN APPLICATION**

**Problem Statement:**

To implement Fuzzy logic using matplotlib in python and find the graph of temperature, humidity and speed in different conditions.

**Algorithm:**

1. Locate the input, output, and state variables of the plane under consideration.
2. Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
3. Obtain the membership function for each fuzzy subset.
4. Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and the output of fuzzy subsets on the other side, thereby forming the rule base.
5. Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0, 1] and [-1, I] interval.
6. Carry out the fuzzification process.
7. Identify the output contributed from each rule using fuzzy approximate reasoning.
8. Combine the fuzzy outputs obtained from each rule.
9. Finally, apply defuzzification to form a crisp output.

**Optimization Technique:**

1. Decomposing the large-scale system into a collection of various subsystems.
2. Varying the plant dynamics slowly and linearizing the nonlinear plane dynamics about a set of operating points.
3. Organizing a set of state variables, control variables, or output features for the system under consideration.
4. Designing simple P, PD, PID controllers for the subsystems. Optimal controllers can also be designed.

**Uncertainty In this problem :** Fuzzy Logic - Temperature, Humidity.

**CODE :** M = 9 def puzzle(a): for i in range(M): for j in range(M): print(a[i][j],end = " ")

print() def solve(grid, row, col, num):

for x in range(9): if grid[row][x] == num:

return False

for x in range(9): if grid[x][col] == num:

return False

startRow = row - row % 3 startCol = col - col % 3 for i in range(3): for j in range(3): if grid[i + startRow][j + startCol] == num:

return False return True

def Suduko(grid, row, col):

if (row == M - 1 and col == M):

return True if col == M: row += 1 col = 0 if grid[row][col] > 0: return Suduko(grid, row, col + 1) for num in range(1, M + 1, 1):

if solve(grid, row, col, num):

grid[row][col] = num if Suduko(grid, row, col + 1):

return True grid[row][col] = 0 return False

'''0 means the cells where no value is assigned'''

grid = [[2, 5, 0, 0, 3, 0, 9, 0, 1],

[0, 1, 0, 0, 0, 4, 0, 0, 0],

[4, 0, 7, 0, 0, 0, 2, 0, 8],

[0, 0, 5, 2, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 9, 8, 1, 0, 0],

[0, 4, 0, 0, 0, 3, 0, 0, 0],

[0, 0, 0, 3, 6, 0, 0, 7, 2],

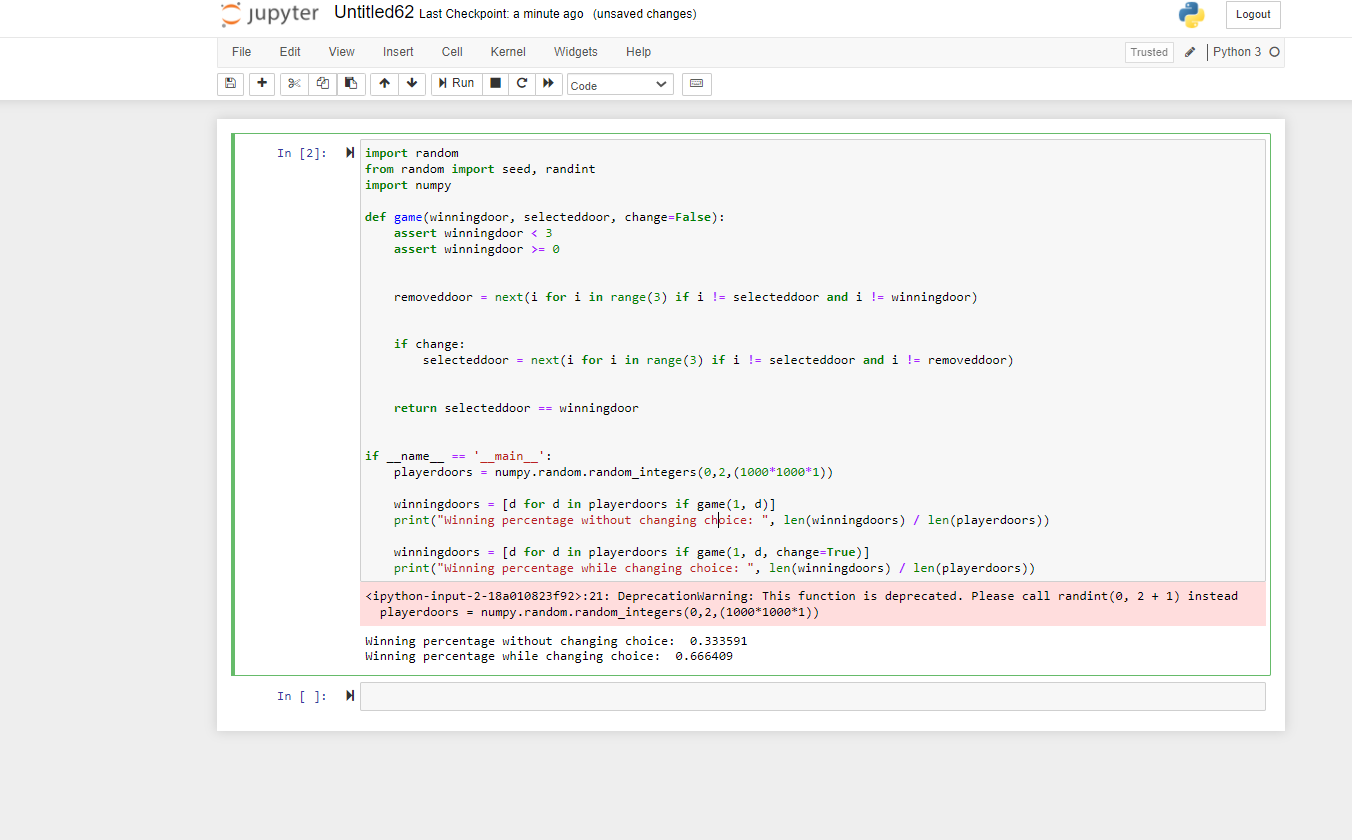
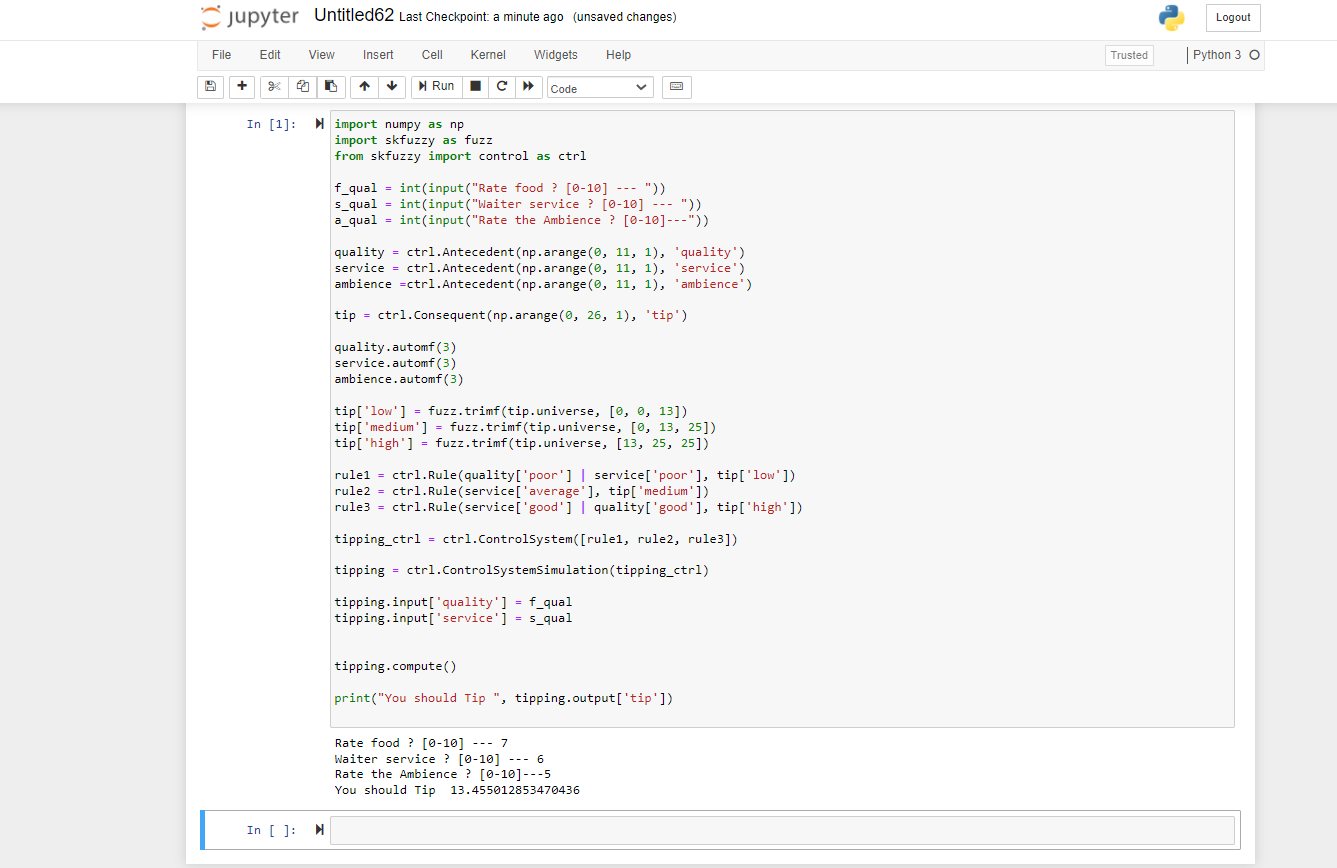
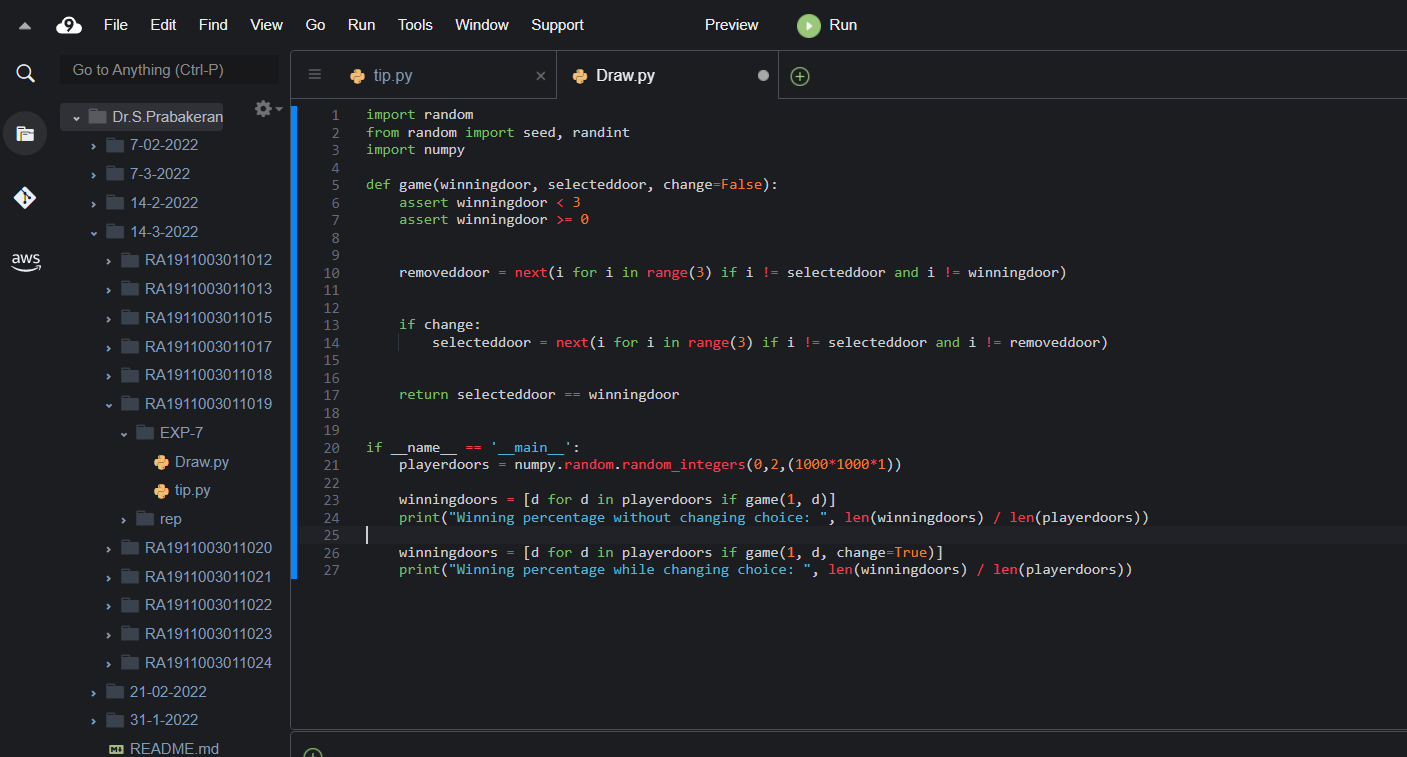
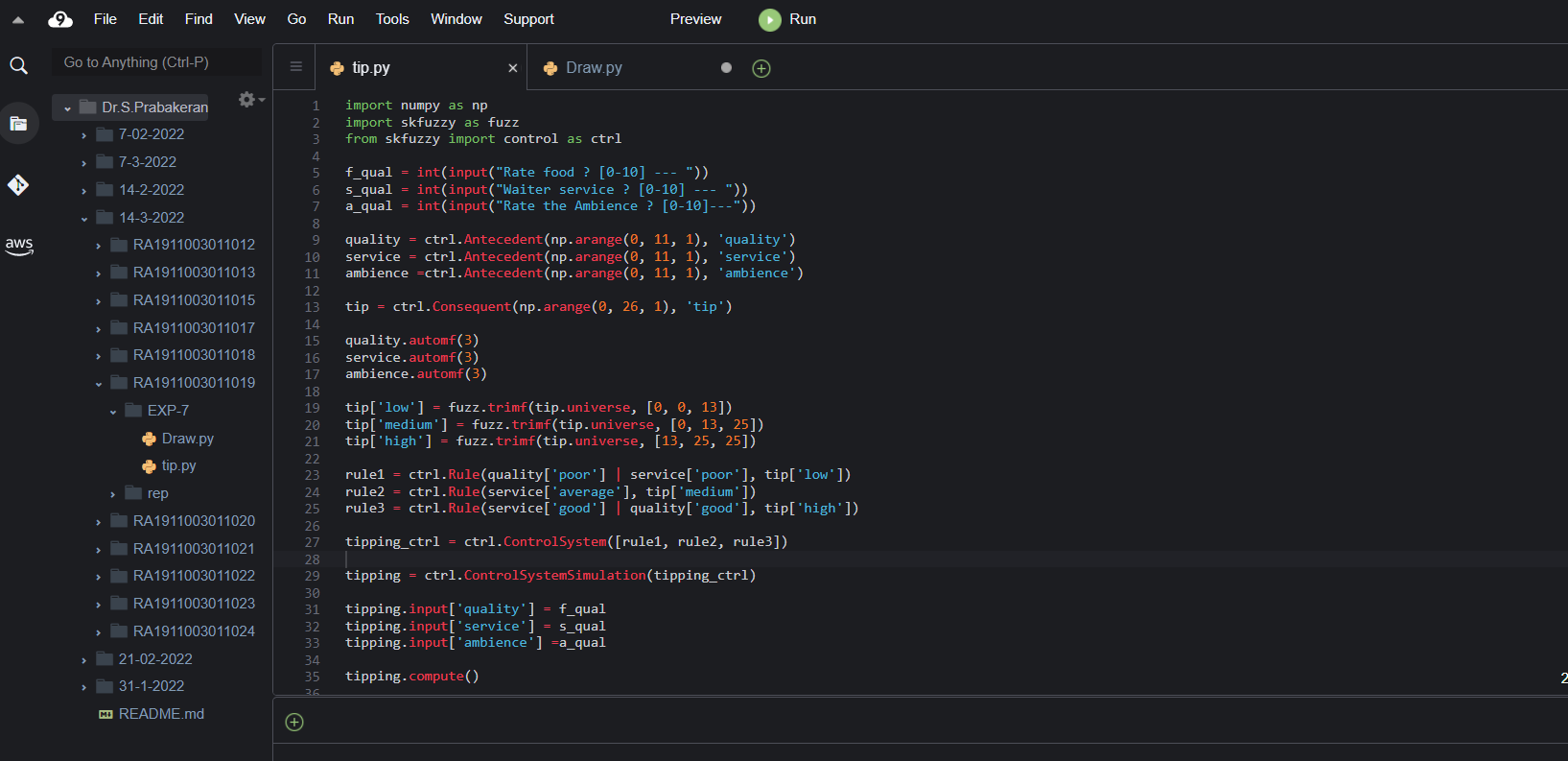
[0, 7, 0, 0, 0, 0, 0, 0, 3],

[9, 0, 3, 0, 0, 0, 6, 0, 4]]

if (Suduko(grid, 0, 0)): puzzle(grid) else:

print("Solution does not exist:(")

**OUTPUT :**



**Result:** We have successfully implemented fuzzy uncertainty problem using matplotlib and output is received

**Experiment No:-8**

**Date:-08-03-2022**

**UNIFICATION AND RESOLUTION**

**AIM:** Developing an algorithm for implementation of **UNIFICATION AND RESOLUTION**

**PROBLEM STATEMENT:** Developing an optimized technique using an appropriate artificial intelligence algorithm to solve the Unification and Resolution.

**ALGORITHM :**

1. function PL-RESOLUTION (KB, Q) returns true or false inputs: KB,

2. the knowledge base, group of sentences/facts in propositional logic

3. Q, the query, a sentence in propositional logic

4. clauses → the set of clauses in the CNF representation of KB ^ Q new → { }

5. loop do for each Ci, Cj in clauses do

6. resolvents → PL-RESOLVE (Ci, Cj)

7. if resolvents contains the empty clause the return true

8. new → new union resolvents

9. if new is a subset of clauses then return false

10. clauses → clauses union true

**OPTIMIZATION TECHNIQUE:**

Resolution basically works by using the principle of proof by contradiction. To find the conclusion we should negate the conclusion. Then the resolution rule is applied to the resulting clauses. Each clause that contains complementary literals is resolved to produce a2. new clause, which can be added to the set of facts (if it is not already present). This process continues until one of the two things happen:•There are no new clauses that can be added. An application of the resolution rule derives the empty clauseAn empty clause shows that the negation of the conclusion is a complete contradiction,hence the negation of the conclusion is invalid or false or the assertion is completely valid or true.

1. Convert the given statements in Predicate/Propositional Logic

2. Convert these statements into Conjunctive Normal Form

3. Negate the Conclusion (Proof by Contradiction)

4. Resolve using a Resolution Tree (Unification)

**UNIFICATION CODE:**

def get\_index\_comma(string):

index\_list = list()

par\_count = 0

for i in range(len(string)):

if string[i] == ',' and par\_count == 0:

index\_list.append(i)

elif string[i] == '(':

par\_count += 1

elif string[i] == ')':

par\_count -= 1

return index\_list

def is\_variable(expr):

for i in expr:

if i == '(' or i == ')':

return False

return True

def process\_expression(expr):

expr = expr.replace(' ', '')

index = None

for i in range(len(expr)):

if expr[i] == '(':

index = i

break

predicate\_symbol = expr[:index]

expr = expr.replace(predicate\_symbol, '')

expr = expr[1:len(expr) - 1]

arg\_list = list()

indices = get\_index\_comma(expr)

if len(indices) == 0:

arg\_list.append(expr)

else:

arg\_list.append(expr[:indices[0]])

for i, j in zip(indices, indices[1:]):

arg\_list.append(expr[i + 1:j])

arg\_list.append(expr[indices[len(indices) - 1] + 1:])

return predicate\_symbol, arg\_list

def get\_arg\_list(expr):

\_, arg\_list = process\_expression(expr)

flag = True

while flag:

flag = False

for i in arg\_list:

if not is\_variable(i):

flag = True

\_, tmp = process\_expression(i)

for j in tmp:

if j not in arg\_list:

arg\_list.append(j)

arg\_list.remove(i)

return arg\_list

def check\_occurs(var, expr):

arg\_list = get\_arg\_list(expr)

if var in arg\_list:

return True

return False

def unify(expr1, expr2):

if is\_variable(expr1) and is\_variable(expr2):

if expr1 == expr2:

return 'Null'

else:

return False

elif is\_variable(expr1) and not is\_variable(expr2):

if check\_occurs(expr1, expr2):

return False

else:

tmp = str(expr2) + '/' + str(expr1)

return tmp

elif not is\_variable(expr1) and is\_variable(expr2):

if check\_occurs(expr2, expr1):

return False

else:

tmp = str(expr1) + '/' + str(expr2)

return tmp

else:

predicate\_symbol\_1, arg\_list\_1 = process\_expression(expr1)

predicate\_symbol\_2, arg\_list\_2 = process\_expression(expr2)

# Step 2

if predicate\_symbol\_1 != predicate\_symbol\_2:

return False

# Step 3

elif len(arg\_list\_1) != len(arg\_list\_2):

return False

else:

# Step 4: Create substitution list

sub\_list = list()

# Step 5:

for i in range(len(arg\_list\_1)):

tmp = unify(arg\_list\_1[i], arg\_list\_2[i])

if not tmp:

return False

elif tmp == 'Null':

pass

else:

if type(tmp) == list:

for j in tmp:

sub\_list.append(j)

else:

sub\_list.append(tmp)

# Step 6

return sub\_list

if \_\_name\_\_ == '\_\_main\_\_':

f1 = 'Q(a, g(x, a), f(y))'

f2 = 'Q(a, g(f(b), a), x)'

# f1 = input('f1 : ')

# f2 = input('f2 : ')

result = unify(f1, f2)

if not result:

print('The process of Unification failed!')

else:

print('The process of Unification successful!')

print(result)

**RESOLUTION CODE:**

import copy

import time

class Parameter:

variable\_count = 1

def \_\_init\_\_(self, name=None):

if name:

self.type = "Constant"

self.name = name

else:

self.type = "Variable"

self.name = "v" + str(Parameter.variable\_count)

Parameter.variable\_count += 1

def isConstant(self):

return self.type == "Constant"

def unify(self, type\_, name):

self.type = type\_

self.name = name

def \_\_eq\_\_(self, other):

return self.name == other.name

def \_\_str\_\_(self):

return self.name

class Predicate:

def \_\_init\_\_(self, name, params):

self.name = name

self.params = params

def \_\_eq\_\_(self, other):

return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def \_\_str\_\_(self):

return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

def getNegatedPredicate(self):

return Predicate(negatePredicate(self.name), self.params)

class Sentence:

sentence\_count = 0

def \_\_init\_\_(self, string):

self.sentence\_index = Sentence.sentence\_count

Sentence.sentence\_count += 1

self.predicates = []

self.variable\_map = {}

local = {}

for predicate in string.split("|"):

name = predicate[:predicate.find("(")]

params = []

for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):

if param[0].islower():

if param not in local: # Variable

local[param] = Parameter()

self.variable\_map[local[param].name] = local[param]

new\_param = local[param]

else:

new\_param = Parameter(param)

self.variable\_map[param] = new\_param

params.append(new\_param)

self.predicates.append(Predicate(name, params))

def getPredicates(self):

return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):

return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate):

self.predicates.remove(predicate)

for key, val in self.variable\_map.items():

if not val:

self.variable\_map.pop(key)

def containsVariable(self):

return any(not param.isConstant() for param in self.variable\_map.values())

def \_\_eq\_\_(self, other):

if len(self.predicates) == 1 and self.predicates[0] == other:

return True

return False

def \_\_str\_\_(self):

return "".join([str(predicate) for predicate in self.predicates])

class KB:

def \_\_init\_\_(self, inputSentences):

self.inputSentences = [x.replace(" ", "") for x in inputSentences]

self.sentences = []

self.sentence\_map = {}

def prepareKB(self):

self.convertSentencesToCNF()

for sentence\_string in self.inputSentences:

sentence = Sentence(sentence\_string)

for predicate in sentence.getPredicates():

self.sentence\_map[predicate] = self.sentence\_map.get(

predicate, []) + [sentence]

def convertSentencesToCNF(self):

for sentenceIdx in range(len(self.inputSentences)):

# Do negation of the Premise and add them as literal

if "=>" in self.inputSentences[sentenceIdx]:

self.inputSentences[sentenceIdx] = negateAntecedent(

self.inputSentences[sentenceIdx])

def askQueries(self, queryList):

results = []

for query in queryList:

negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))

negatedPredicate = negatedQuery.predicates[0]

prev\_sentence\_map = copy.deepcopy(self.sentence\_map)

self.sentence\_map[negatedPredicate.name] = self.sentence\_map.get(

negatedPredicate.name, []) + [negatedQuery]

self.timeLimit = time.time() + 40

try:

result = self.resolve([negatedPredicate], [

False]\*(len(self.inputSentences) + 1))

except:

result = False

self.sentence\_map = prev\_sentence\_map

if result:

results.append("TRUE")

else:

results.append("FALSE")

return results

def resolve(self, queryStack, visited, depth=0):

if time.time() > self.timeLimit:

raise Exception

if queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate()

queryPredicateName = negatedQuery.name

if queryPredicateName not in self.sentence\_map:

return False

else:

queryPredicate = negatedQuery

for kb\_sentence in self.sentence\_map[queryPredicateName]:

if not visited[kb\_sentence.sentence\_index]:

for kbPredicate in kb\_sentence.findPredicates(queryPredicateName):

canUnify, substitution = performUnification(

copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

if canUnify:

newSentence = copy.deepcopy(kb\_sentence)

newSentence.removePredicate(kbPredicate)

newQueryStack = copy.deepcopy(queryStack)

if substitution:

for old, new in substitution.items():

if old in newSentence.variable\_map:

parameter = newSentence.variable\_map[old]

newSentence.variable\_map.pop(old)

parameter.unify(

"Variable" if new[0].islower() else "Constant", new)

newSentence.variable\_map[new] = parameter

for predicate in newQueryStack:

for index, param in enumerate(predicate.params):

if param.name in substitution:

new = substitution[param.name]

predicate.params[index].unify(

"Variable" if new[0].islower() else "Constant", new)

for predicate in newSentence.predicates:

newQueryStack.append(predicate)

new\_visited = copy.deepcopy(visited)

if kb\_sentence.containsVariable() and len(kb\_sentence.predicates) > 1:

new\_visited[kb\_sentence.sentence\_index] = True

if self.resolve(newQueryStack, new\_visited, depth + 1):

return True

return False

return True

def performUnification(queryPredicate, kbPredicate):

substitution = {}

if queryPredicate == kbPredicate:

return True, {}

else:

for query, kb in zip(queryPredicate.params, kbPredicate.params):

if query == kb:

continue

if kb.isConstant():

if not query.isConstant():

if query.name not in substitution:

substitution[query.name] = kb.name

elif substitution[query.name] != kb.name:

return False, {}

query.unify("Constant", kb.name)

else:

return False, {}

else:

if not query.isConstant():

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

kb.unify("Variable", query.name)

else:

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

return True, substitution

def negatePredicate(predicate):

return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):

antecedent = sentence[:sentence.find("=>")]

premise = []

for predicate in antecedent.split("&"):

premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find("=>") + 2:])

return "|".join(premise)

def getInput(filename):

with open(filename, "r") as file:

noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() for \_ in range(noOfQueries)]

noOfSentences = int(file.readline().strip())

inputSentences = [file.readline().strip()

for \_ in range(noOfSentences)]

return inputQueries, inputSentences

def printOutput(filename, results):

print(results)

with open(filename, "w") as file:

for line in results:

file.write(line)

file.write("\n")

file.close()

if \_\_name\_\_ == '\_\_main\_\_':

inputQueries\_, inputSentences\_ = getInput('input.txt')

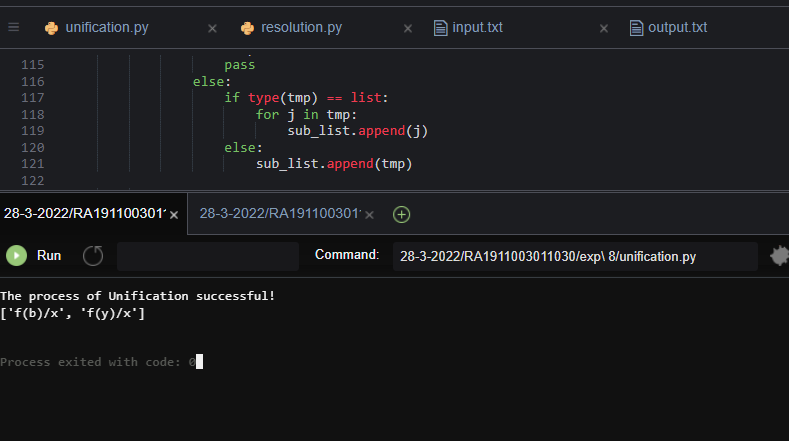
knowledgeBase = KB(inputSentences\_)

knowledgeBase.prepareKB()

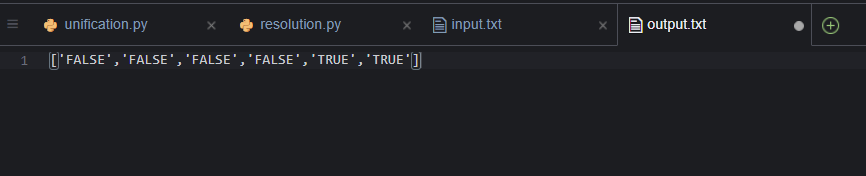
results\_ = knowledgeBase.askQueries(inputQueries\_)

printOutput("output.txt", results\_)

**UNIFICATION OUTPUT:**



**RESOLUTION OUTPUT:**



**RESULT:**

Developed Unification and Resolution Algorithm in Python for solving logical problems.

**Experiment No:-9**

**Date:-21-03-2022**

**Implementation of learning algorithms for an application**

**Aim:**

a) Implementation of Linear Regression algorithm to predict students score using the given dataset.

b) Implementation of Support Vector Classification algorithm to classify the cases of breast cancer

using the given dataset.

c) Implementation of K-means clustering algorithm to group the customers based on their

demographic detail using the given dataset.

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn import metrics

%matplotlib inline

Import required modules and packages

dataset = pd.read\_csv('….\student\_scores.csv')

dataset.head()

Import data set

Choose the right path for the dataset

dataset.describe() Descriptive statistics of the attributes

available in the dataset

dataset.plot(x='Hours', y='Scores', style='o')

plt.title('Hours vs Percentage')

plt.xlabel('Hours Studied')

plt.ylabel('Percentage Score')

plt.show()

Visualize the data.

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, 1].values

Identify the independent (X) and

dependent variables (y) in the data set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,

test\_size=0.2, random\_state=0)

print('X train shape: ', X\_train.shape)

print('Y train shape: ', Y\_train.shape)

print('X test shape: ', X\_test.shape)

Splitting the given data in to training set

(80%) and testing set (20%)

Beginners Level

Lab 11 - Implementation of Learning Algorithms for an Application

18CSC305J - ARTIFICIAL INTELLIGENCE Page 4

print('Y test shape: ', Y\_test.shape)

regressor = LinearRegression()

Model instantiation

regressor.fit(X\_train, y\_train) Model Training

print(regressor.intercept\_)

print(regressor.coef\_)

Finding out the coefficient (a) and

intercept (b) value of linear model

(y=aX+b)

y\_pred = regressor.predict(X\_test)

df = pd.DataFrame({'Actual': y\_test, 'Predicted': y\_pred})

print(df)

Testing the model

print('Mean Absolute Error:',

metrics.mean\_absolute\_error (y\_test, y\_pred))

print('Mean Squared Error:',

metrics.mean\_squared\_error (y\_test, y\_pred))

print('Root Mean Squared Error:',

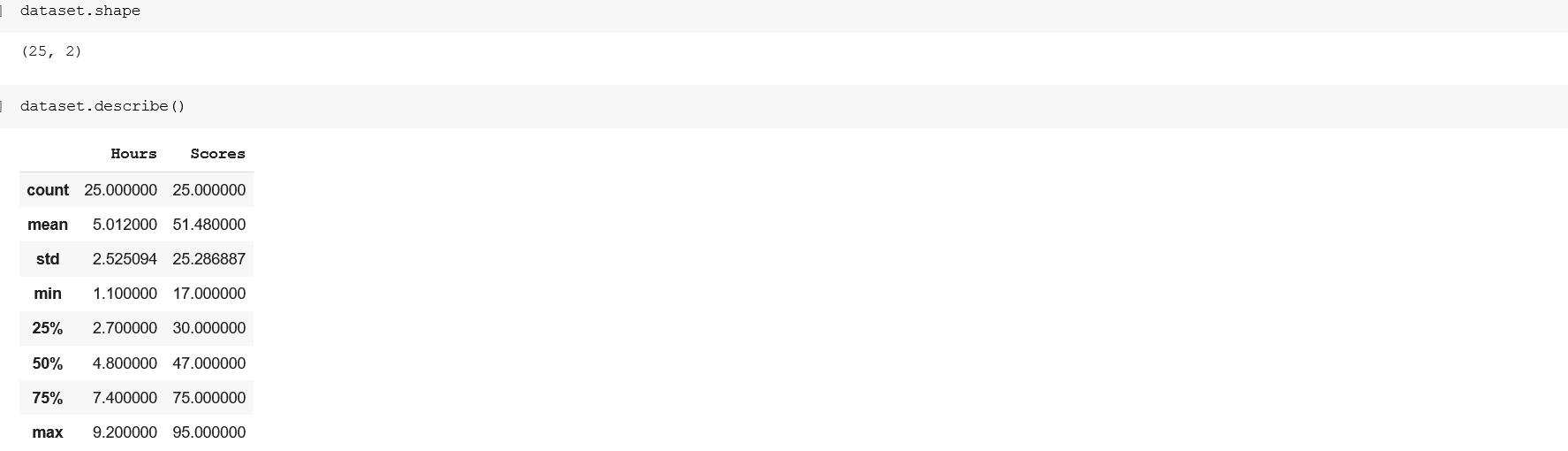
np.sqrt(metrics.mean\_squared\_error (y\_test, y\_pred)))

MAE, MSE, RMSE – Evaluation metrics

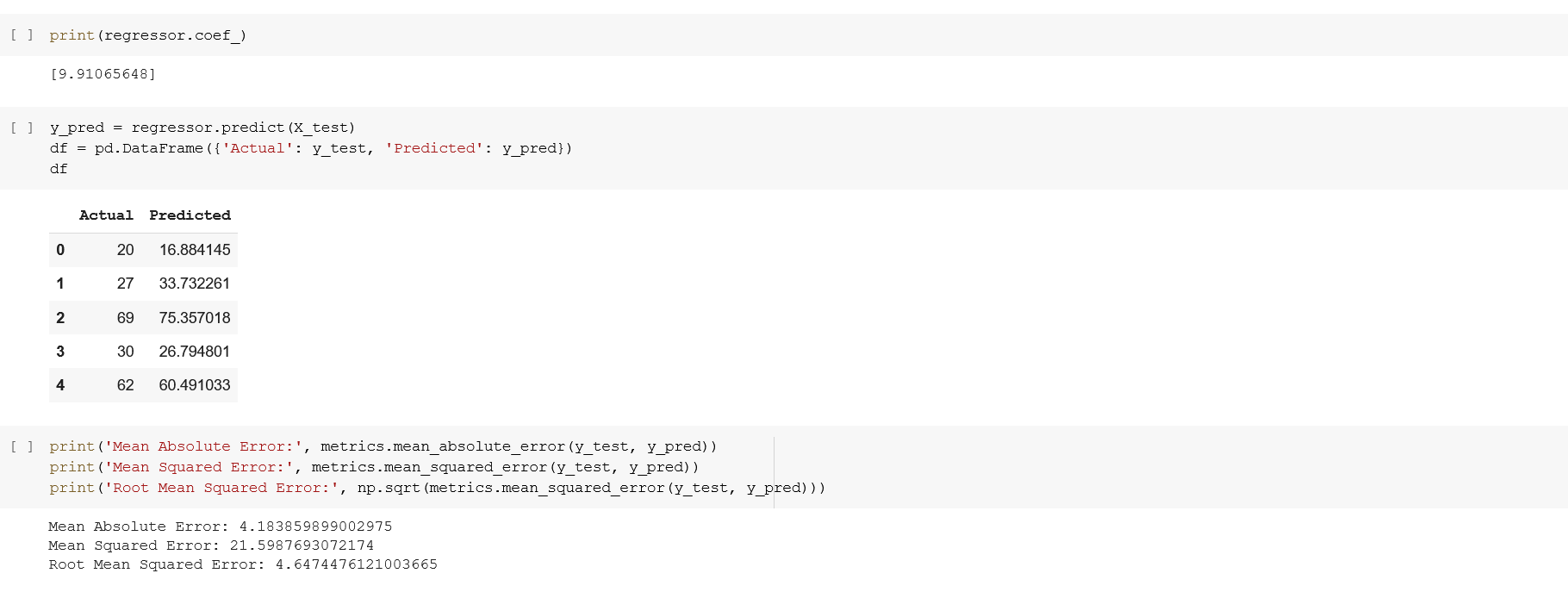
of Model

Discussion:









B)

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import confusion\_matrix,

classification\_report

Import required modules and packages

dataset = pd.read\_csv('….\diabetes data.csv')

print(dataset.head())

Import data set

Choose the right path for the dataset

def diagnosis(x):

if x=='M' :

return 1

if x=='B' :

return 0

dataset['diagnosis'] = dataset['diagnosis'].apply(diagnosis)

print(dataset)

Data cleaning process. Converting

categorical value in to numerical value.

M = malignant, B = benign

print("Any missing sample in data set:",

dataset.isnull().values.any(), "\n")

Check for any missing values in the data

set

dataset = dataset.replace([np.inf, -np.inf], np.nan)

dataset= dataset.fillna(dataset.mean())

dataset

Replace the missing value with its mean

value of the respective attribute

dataset= dataset.drop(columns=["Unnamed: 32"]) drop this column because it's not

necessary (null)

Y = dataset['diagnosis']

X = dataset.drop(columns=['diagnosis'])

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y,

test\_size=0.2, random\_state=9)

print('X train shape: ', X\_train.shape)

print('Y train shape: ', Y\_train.shape)

print('X test shape: ', X\_test.shape)

print('Y test shape: ', Y\_test.shape)

Splitting the given data in to training set

(80%) and testing set (20%)

svc\_classifier= SVC(kernel='poly') Model instantiation. Apply SVM with

different kernels 'linear', 'poly', 'rbf',

'sigmoid' and verify the accuracy of the

model

svc\_classifier.fit(X\_train,Y\_train) Model Training

y\_pred=svc\_classifier.predict(X\_test) Testing the model

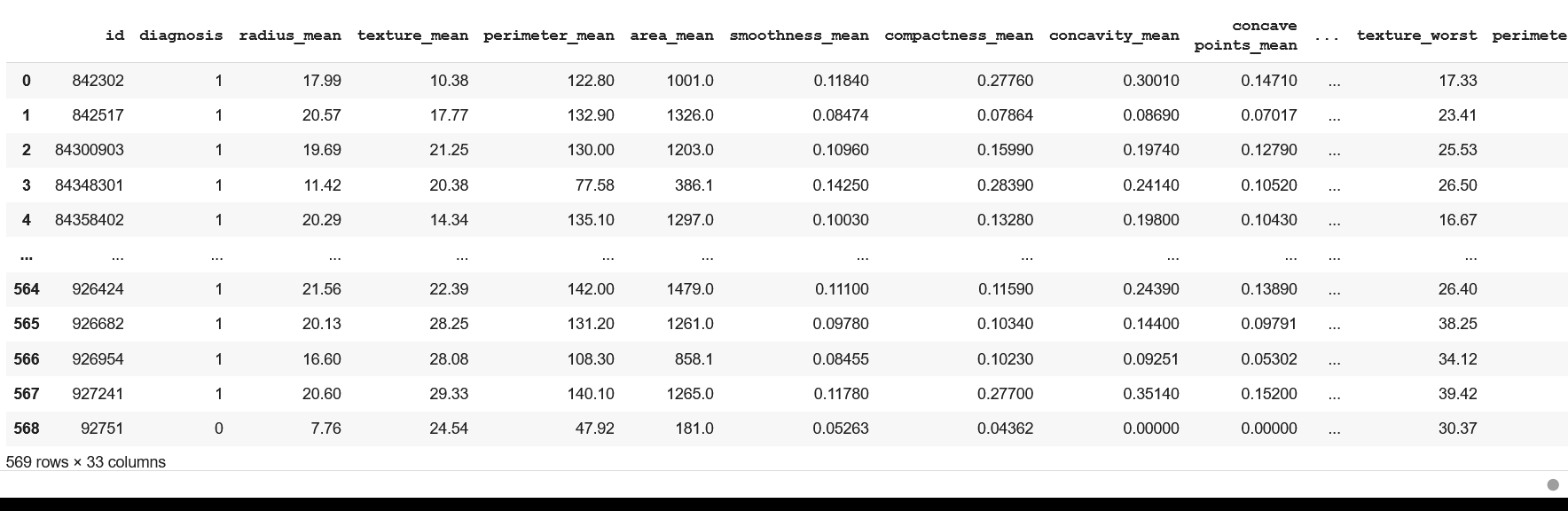
print(confusion\_matrix(Y\_test,y\_pred))

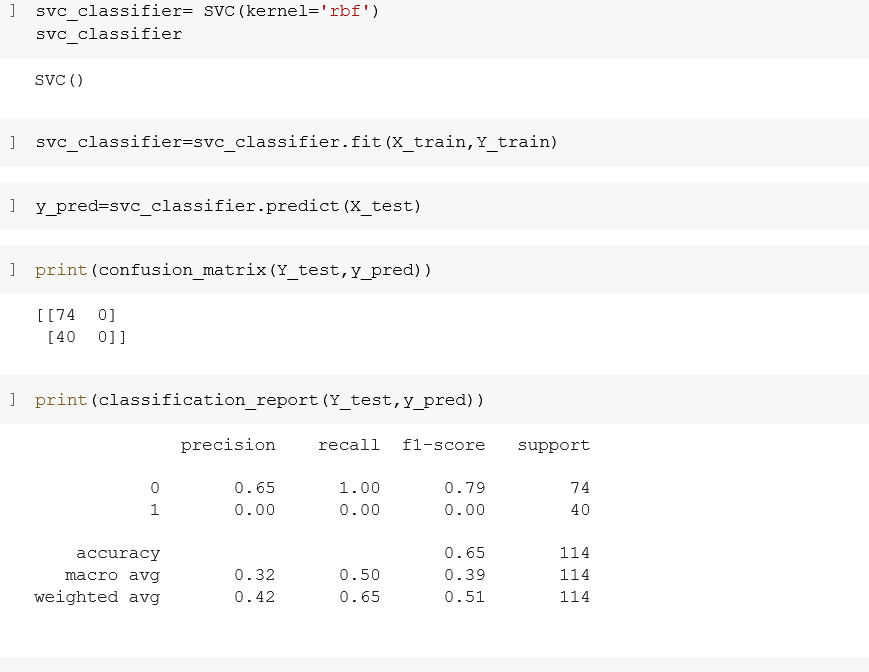
print(classification\_report(Y\_test,y\_pred))

Evaluation metrics to measure the

performance of the model







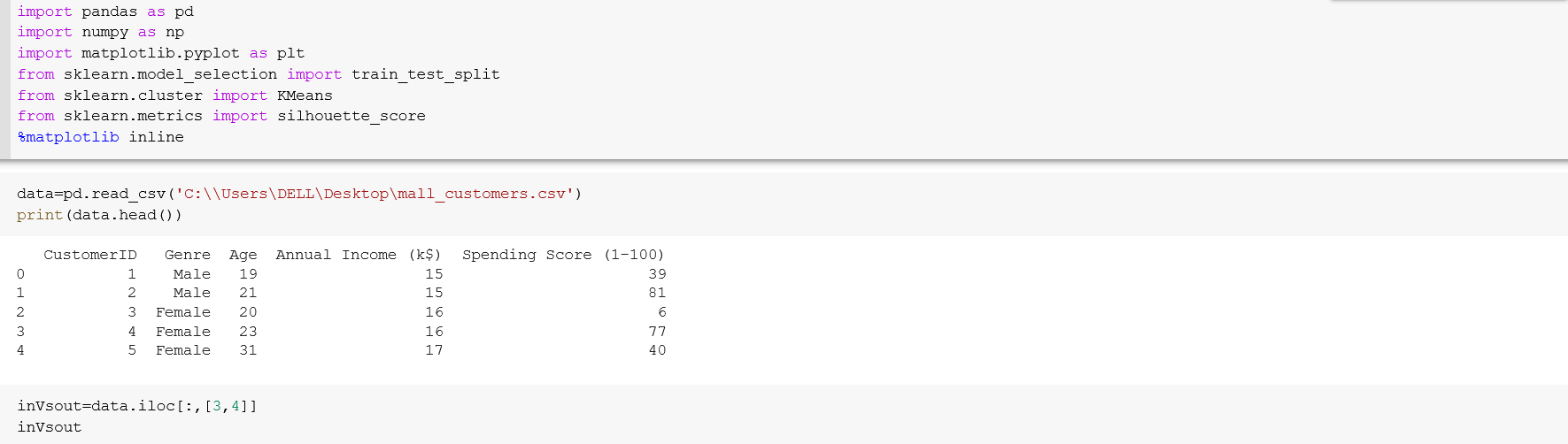
(c) Implementation of K-means clustering algorithm to group the customers based on

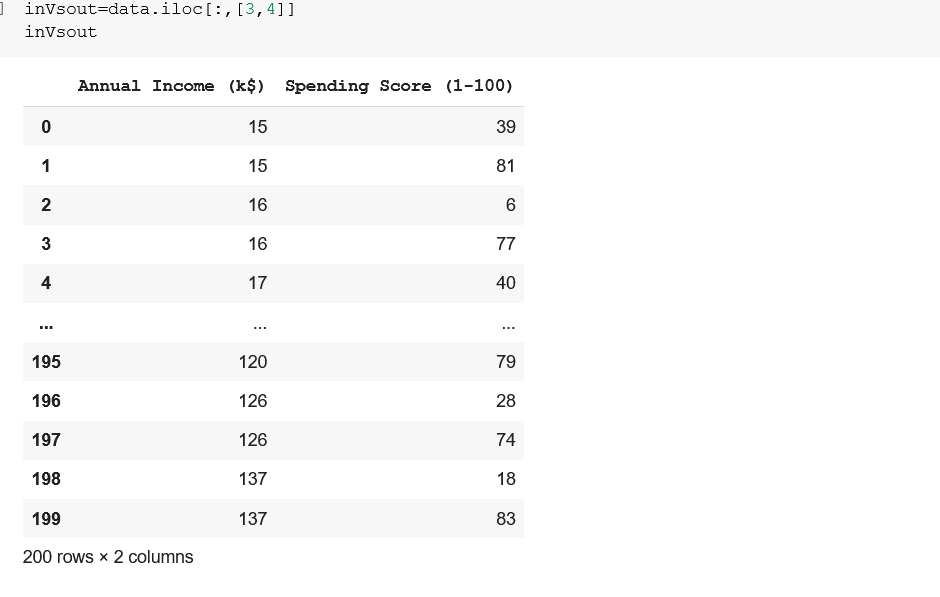
their demographic detail using the given dataset.

Problem: Client is owing a supermarket mall and through membership cards, client have some basic data

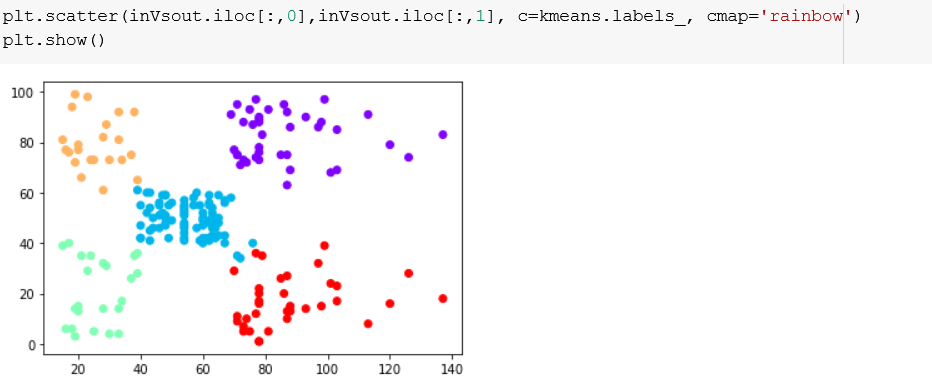
about your customers like Customer ID, age, gender, annual income and spending score. Help the client

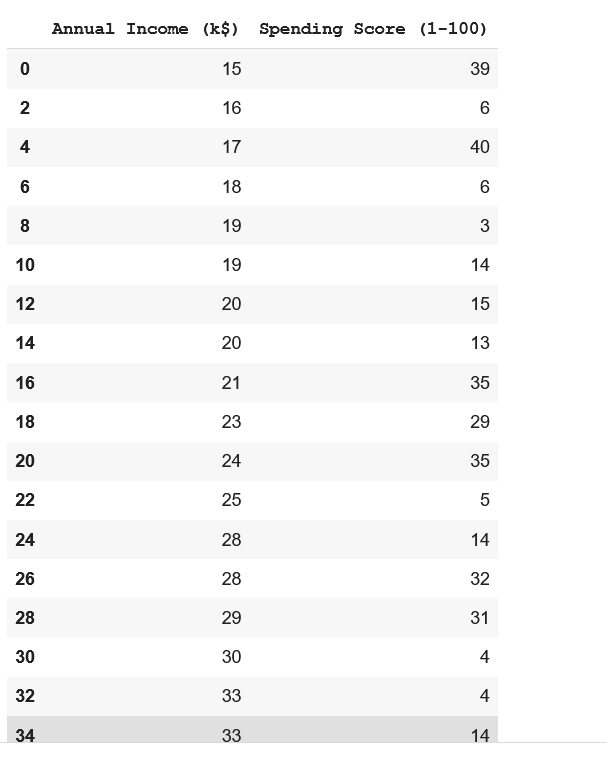
to understand the customers like who are the target customers so that the sense can be given to marketing

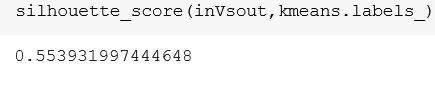












**Experiment No:-10**

**Date:-10-04-2022**

**To Implement NLP programs**

**Aim:-**To Implement NLP programs

NLP stands for Natural Language Processing, which is a part of Computer Science, Human language, and Artificial Intelligence. It is the technology that is used by machines to understand, analyse, manipulate, and interpret human's languages. It helps developers to organize knowledge for performing tasks such as translation, automatic summarization, Named Entity Recognition (NER), speech recognition, relationship extraction, and topic segmentation.

**code:-**

!pip install -q wordcloud

import wordcloud

import nltk

nltk.download('stopwords')

nltk.download('wordnet')

nltk.download('punkt')

nltk.download('averaged\_perceptron\_tagger')

import pandas as pd

import matplotlib.pyplot as plt

import io

import unicodedata

import numpy as np

import re

import string

# Constants

# POS (Parts Of Speech) for: nouns, adjectives, verbs and adverbs

DI\_POS\_TYPES = {'NN':'n', 'JJ':'a', 'VB':'v', 'RB':'r'}

POS\_TYPES = list(DI\_POS\_TYPES.keys())

# Constraints on tokens

MIN\_STR\_LEN = 3

RE\_VALID = '[a-zA-Z]'

# Upload from google drive

from google.colab import files

uploaded = files.upload()

print("len(uploaded.keys():", len(uploaded.keys()))

for fn in uploaded.keys():

print('User uploaded file "{name}" with length {length} bytes'.format(name=fn, length=len(uploaded[fn])))

# Get list of quotes

df\_quotes = pd.read\_csv(io.StringIO(uploaded['quotes.txt'].decode('utf-8')), sep='\t')

# Display

print("df\_quotes:")

print(df\_quotes.head().to\_string())

print(df\_quotes.describe())

# Convert quotes to list

li\_quotes = df\_quotes['Quote'].tolist()

print()

print("len(li\_quotes):", len(li\_quotes)

# Get stopwords, stemmer and lemmatizer

stopwords = nltk.corpus.stopwords.words('english')

stemmer = nltk.stem.PorterStemmer()

lemmatizer = nltk.stem.WordNetLemmatizer()

# Remove accents function

def remove\_accents(data):

return ''.join(x for x in unicodedata.normalize('NFKD', data) if x in string.ascii\_letters or x == " ")

# Process all quotes

li\_tokens = []

li\_token\_lists = []

li\_lem\_strings = []

for i,text in enumerate(li\_quotes):

# Tokenize by sentence, then by lowercase word

tokens = [word.lower() for sent in nltk.sent\_tokenize(text) for word in nltk.word\_tokenize(sent)]

# Process all tokens per quote

li\_tokens\_quote = []

li\_tokens\_quote\_lem = []

for token in tokens:

# Remove accents

t = remove\_accents(token)

# Remove punctuation

t = str(t).translate(string.punctuation)

li\_tokens\_quote.append(t)

# Add token that represents "no lemmatization match"

li\_tokens\_quote\_lem.append("-") # this token will be removed if a lemmatization match is found below

# Process each token

if t not in stopwords:

if re.search(RE\_VALID, t):

if len(t) >= MIN\_STR\_LEN:

# Note that the POS (Part Of Speech) is necessary as input to the lemmatizer

# (otherwise it assumes the word is a noun)

pos = nltk.pos\_tag([t])[0][1][:2]

pos2 = 'n' # set default to noun

if pos in DI\_POS\_TYPES:

pos2 = DI\_POS\_TYPES[pos]

stem = stemmer.stem(t)

lem = lemmatizer.lemmatize(t, pos=pos2) # lemmatize with the correct POS

if pos in POS\_TYPES:

li\_tokens.append((t, stem, lem, pos))

# Remove the "-" token and append the lemmatization match

li\_tokens\_quote\_lem = li\_tokens\_quote\_lem[:-1]

li\_tokens\_quote\_lem.append(lem)

# Build list of token lists from lemmatized tokens

li\_token\_lists.append(li\_tokens\_quote)

# Build list of strings from lemmatized tokens

str\_li\_tokens\_quote\_lem = ' '.join(li\_tokens\_quote\_lem)

li\_lem\_strings.append(str\_li\_tokens\_quote\_lem)

# Build resulting dataframes from lists

df\_token\_lists = pd.DataFrame(li\_token\_lists)

print("df\_token\_lists.head(5):")

print(df\_token\_lists.head(5).to\_string())

# Replace None with empty string

for c in df\_token\_lists:

if str(df\_token\_lists[c].dtype) in ('object', 'string\_', 'unicode\_'):

df\_token\_lists[c].fillna(value='', inplace=True)

df\_lem\_strings = pd.DataFrame(li\_lem\_strings, columns=['lem quote'])

print()

print("")

print("df\_lem\_strings.head():")

print(df\_lem\_strings.head().to\_string())

# Add counts

print("Group by lemmatized words, add count and sort:")

df\_all\_words = pd.DataFrame(li\_tokens, columns=['token', 'stem', 'lem', 'pos'])

df\_all\_words['counts'] = df\_all\_words.groupby(['lem'])['lem'].transform('count')

df\_all\_words = df\_all\_words.sort\_values(by=['counts', 'lem'], ascending=[False, True]).reset\_index()

print("Get just the first row in each lemmatized group")

df\_words = df\_all\_words.groupby('lem').first().sort\_values(by='counts', ascending=False).reset\_index()

print("df\_words.head(10):")

print(df\_words.head(10))

df\_words = df\_words[['lem', 'pos', 'counts']].head(200)

for v in POS\_TYPES:

df\_pos = df\_words[df\_words['pos'] == v]

print()

print("POS\_TYPE:", v)

print(df\_pos.head(10).to\_string())

li\_token\_lists\_flat = [y for x in li\_token\_lists for y in x] # flatten the list of token lists to a single list

print("li\_token\_lists\_flat[:10]:", li\_token\_lists\_flat[:10])

di\_freq = nltk.FreqDist(li\_token\_lists\_flat)

del di\_freq['']

li\_freq\_sorted = sorted(di\_freq.items(), key=lambda x: x[1], reverse=True) # sorted list

print(li\_freq\_sorted)

di\_freq.plot(30, cumulative=False)

li\_lem\_words = df\_all\_words['lem'].tolist()

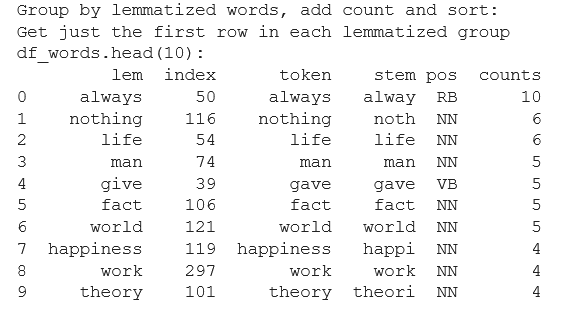
di\_freq2 = nltk.FreqDist(li\_lem\_words)

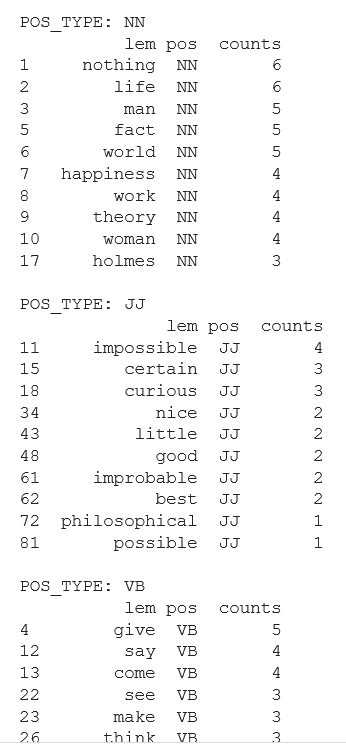
li\_freq\_sorted2 = sorted(di\_freq2.items(), key=lambda x: x[1], reverse=True) # sorted list

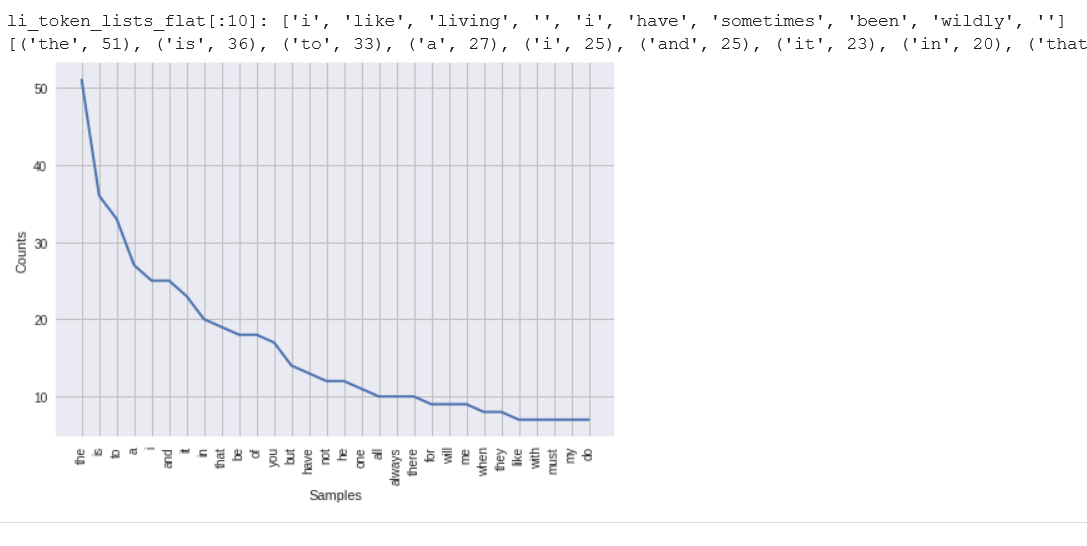
print(li\_freq\_sorted2)

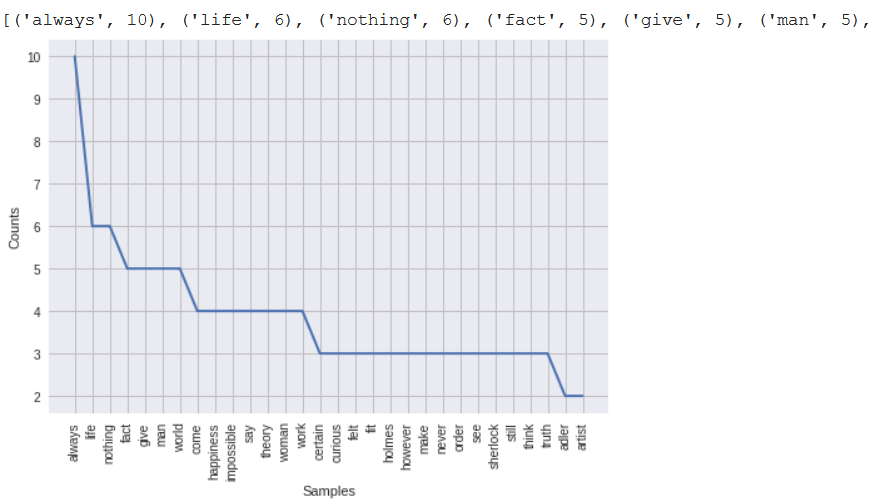
di\_freq2.plot(30, cumulative=False)

**Output:-**









**Result:-**Thus the NPL program was implemented

**Experiment No:-11**

**Date:-06-04-2022**

**Deep Learning AIM:** Implementation of Deep Learning

**CODE:**

**import** tensorflow **as** tf

**from** utils.DL\_utils **import** myCallback, build\_model, compile\_train\_model, plot\_loss\_acc

**from** itertools **import** product

accuracy\_desired **=** [0.85,0.9,0.95]

num\_neurons **=** [16,32,64,128]

cases **=** list(product(accuracy\_desired,num\_neurons)) print("So, the cases we are considering are as follows...\n")

**for** i,c **in** enumerate(cases):

print("Accuracy target {}, number of neurons:

{}"**.**format(c[0],c[1]))

**if** (i**+**1)**%4**==0 and (i+1)!=len(cases):

print("-"**\***50)

**for** c **in** cases:

*# Create a mycallback class with the specific accuracy target*

callbacks **=** myCallback(c[0], print\_msg**=False**)

*# Build a model with a specific number of neurons*

model **=** build\_model(num\_layers**=**1,architecture**=**[c[1]])

*# Compile and train the model passing on the callback class,choose suitable batch size and a max epoch limit*

model **=** compile\_train\_model(model, x\_train,y\_train,callbacks**=**callbacks,

batch\_size**=**32,epochs**=**30)

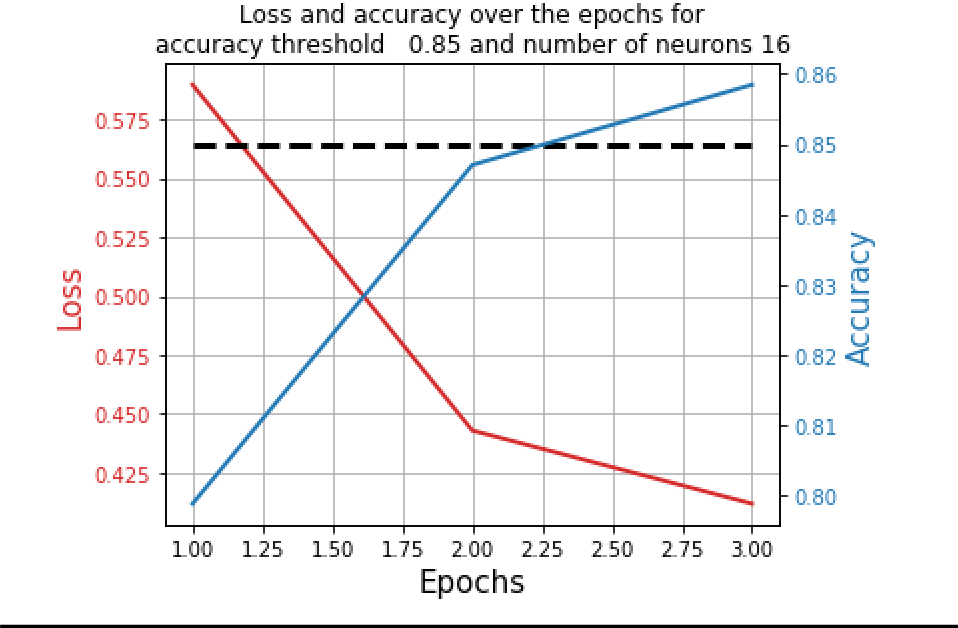
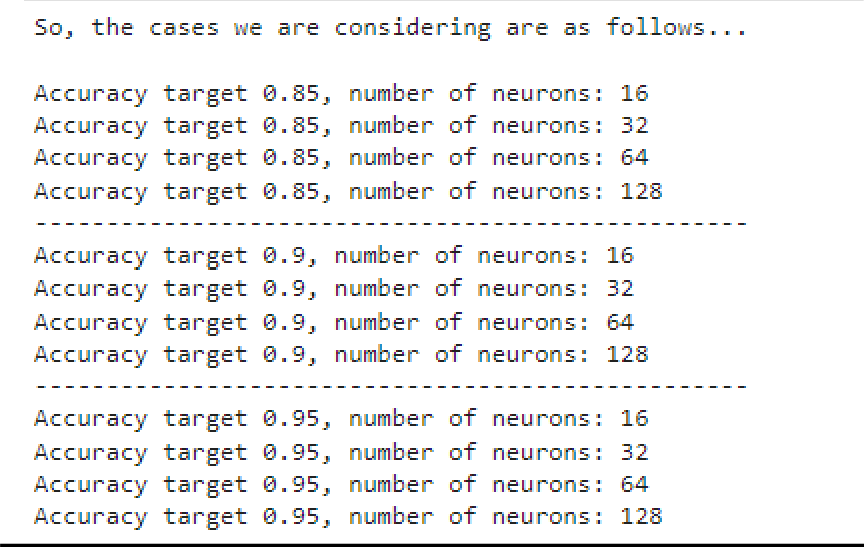
*# Construct a suitable title string for displaying the results properly*

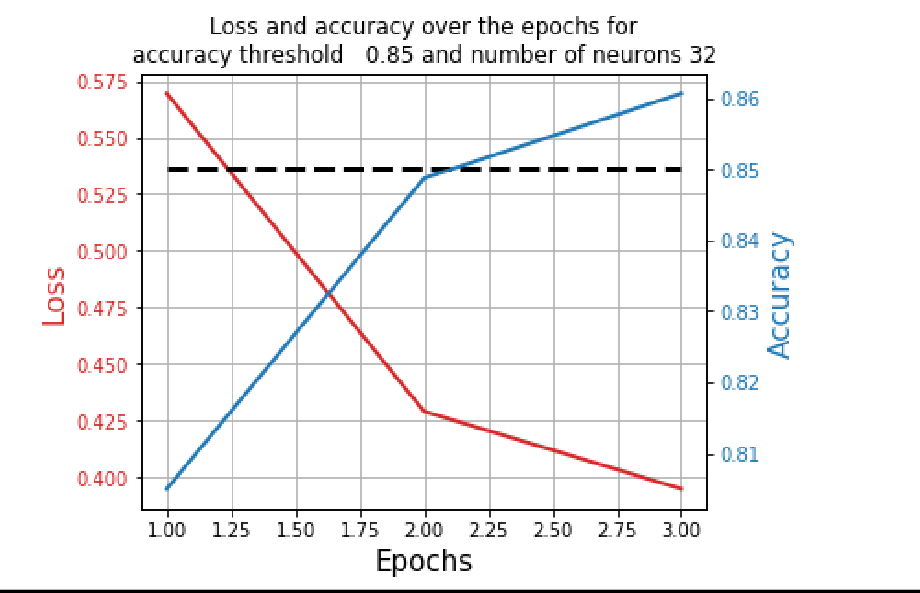
title **=** "Loss and accuracy over the epochs for\naccuracy threshold \

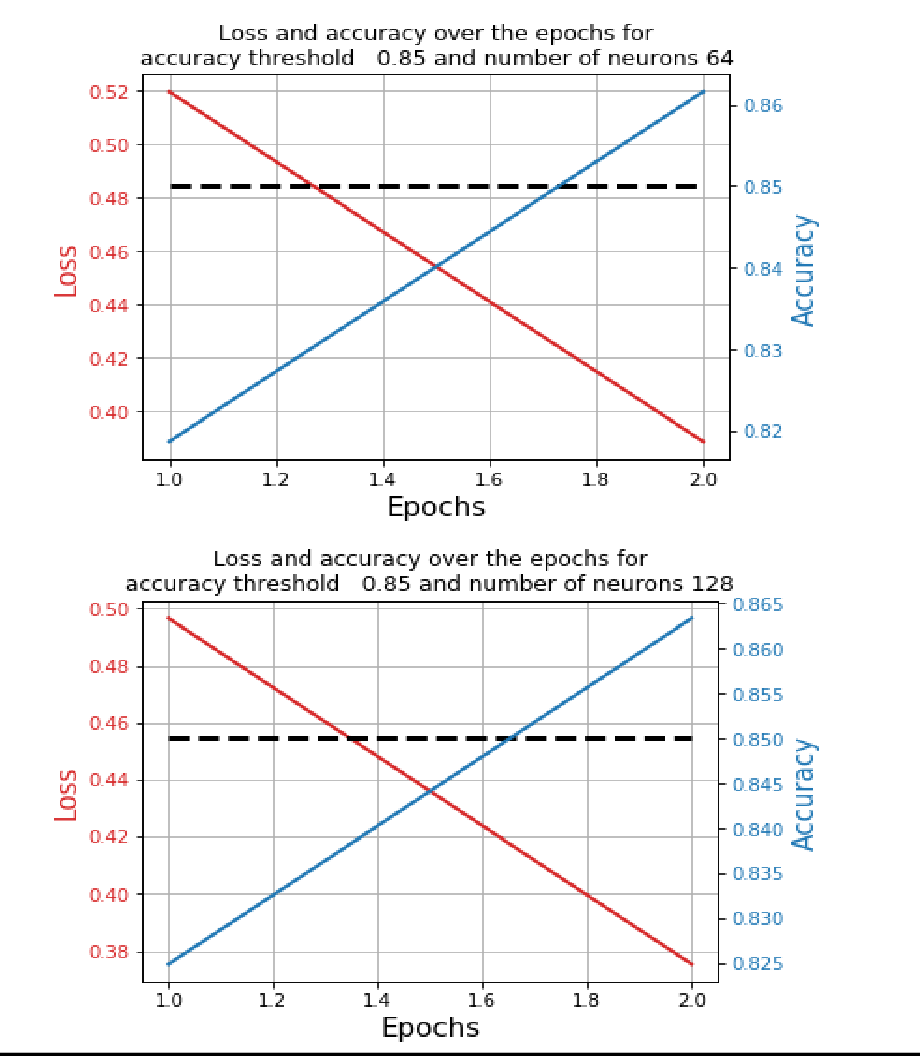
{} and number of neurons {}"**.**format(c[0],c[1])

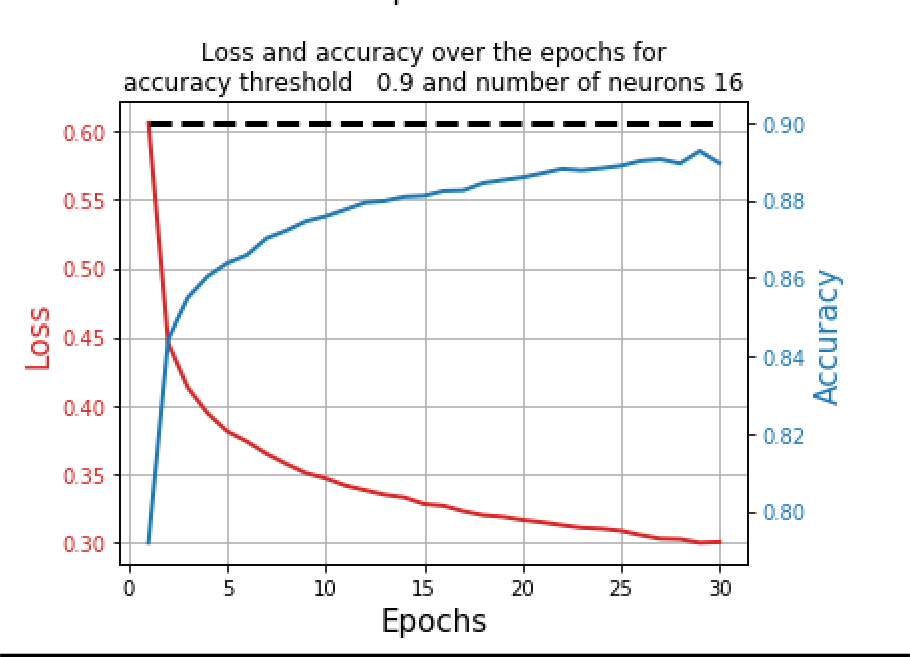
*# Use the plotting utility function, pass on the accuracy target, # trained model, and the custom title string* plot\_loss\_acc(model,target\_acc**=**c[0],title**=**title)

**OUTPUT:**









**RESULT:** Deep Learning Model Implemented