Part 1 – Game Problem statement: TicTacToe with X,T and O

Part 2 – Logic Problem statement: Decision Tree Prolog for Software Defect Prediction

Problem statement – 1

Simulate the working of TicTacToe game extended to three players with coins ‘X’, ‘O’ and ‘T’ with the below sample larger game board of dimension 6*6. The first player to match any four consecutive coins in the same row or same column or same diagonal wins. A sample state of the game board is given below for reference. Your program must start from empty configuration.

X				X	
	X	O		T	
		O	T	T	
		T	T	O	
		X			O

a) You are free to choose your own static evaluation function. Justify your choice of static evaluation value design and explain with a sample game state. Do not use any machine learning model for the evaluation function.

b) Similar to the virtual lab example, one of the players must be a human ie., it must get dynamic inputs from us. The other two players must be simulated using the program.

c) Implement Python code for the design under part a, using Minimax Algorithm.

Introduction

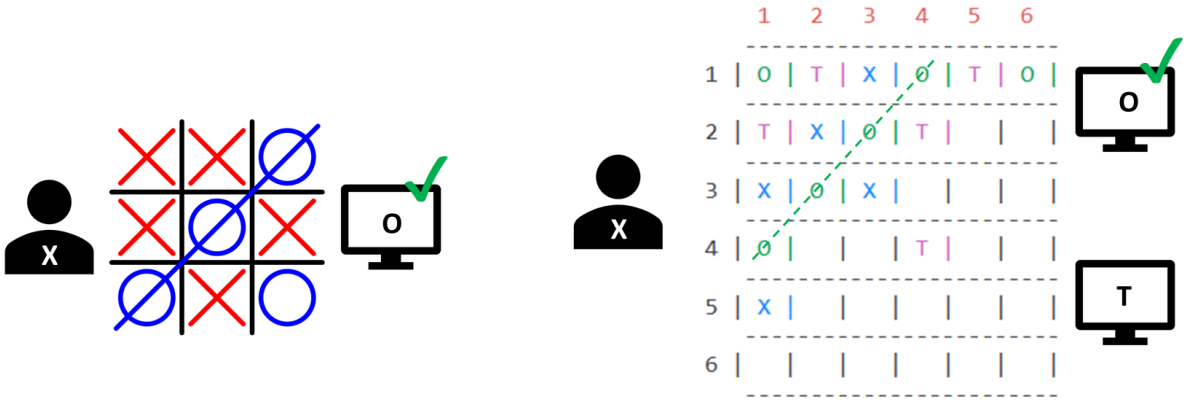
Tic-Tac-Toe

Traditionally Tic-tac-toe also known as '*Noughts and Crosses*', or '*Xs and Os*' is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with X or O. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner.

For 3 players, the game is played in a round-robin fashion, where each player takes turns in a predetermined order, and the game continues until one player wins by getting a certain number of marks in a row/column/diagonal or game is a tie with all spaces on the board filled without any winner.

As per the problem statement, the board is larger i.e. a 6x6 board. This poses a challenge for game-playing algorithms, as the number of possible moves and outcomes increases with the addition of another player and expansion of board size. Game tree search and minimax algorithms can still be used but require modification to account for the added complexity of the game.

Overall, the Tic-Tac-Toe problem provides an interesting and challenging problem for game theory and artificial intelligence research.



Traditional Game with 2 Players on 3x3 board

Game with 3 Players on 6x6 board as per Problem Statement

Figure 1:Tic-Tac-Toe game samples

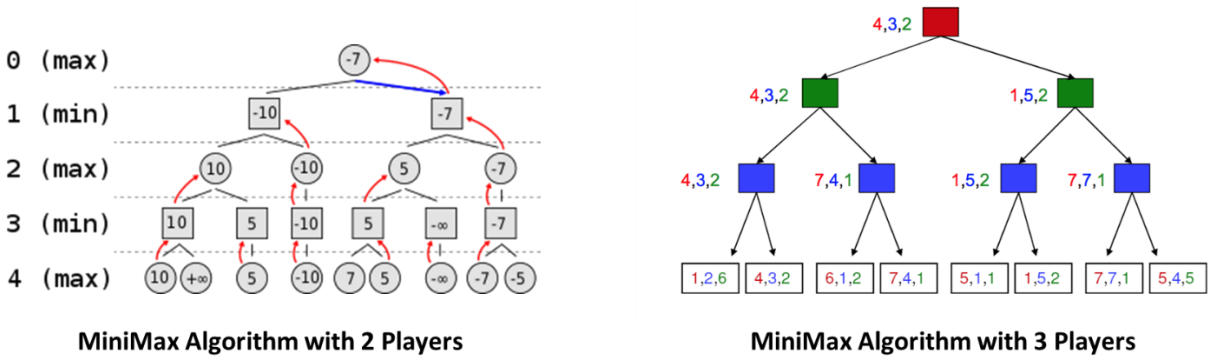
MiniMax Algorithm

Minimax is a decision-making algorithm used in game theory and artificial intelligence to find the optimal move in a game where two players take turns. Traditionally with 2 players, the basic idea of the algorithm is to assume that both players will play optimally and choose the move that leads to the best outcome for the player making the move, assuming that the opponent will choose the move that leads to the worst outcome for the player making the move.

However, it is possible to extend the algorithm to games with more than two players, although the complexity of the algorithm increases significantly. In a game with three players,

the minimax algorithm can be extended by considering all possible combinations of moves by the three players. At each level of the game tree, we assume that each player will make the optimal move based on the assumption that the other two players will also make optimal moves.

To apply the minimax algorithm to Tic-Tac-Toe, we start by creating a game tree that represents all the possible moves and outcomes for both players. Each node in the tree represents a board state, and the edges leading from the node represent the possible moves that can be made from that state.



MiniMax Algorithm with 2 Players **MiniMax Algorithm with 3 Players**

Figure 2: MiniMax Algorithm examples

Inputs

As per the problem statement, Tic-Tac-Toe game with 3 players is to be developed on a 6x6 board using python code. The 3 players' coins chosen are **X** for Human, **O** and **T** for AI agents. The first player to match any four consecutive coins in the same row or same column or same diagonal wins.

Minimax algorithm is to be implemented to solve the Tic-Tac-Toe problem.

X				X	
	X	O		T	
		O	T	T	
		T	T	O	
		X			O

Figure 3: Sample Tic-Tac-Toe game board as per given problem statement

Solution

Python Libraries used in Jupyter Notebook:

In [3]:
```
# Code block
# Initiate Python Libraries
import math
import random
from termcolor import colored, cprint
from colorama import Fore, Style
```

In [4]:
```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

Figure 4: Python Libraries used.

Class definition:

A class is defined as TicTacToe3, with 4 attributes – board, players, current_player, winner and DIAGONALS.

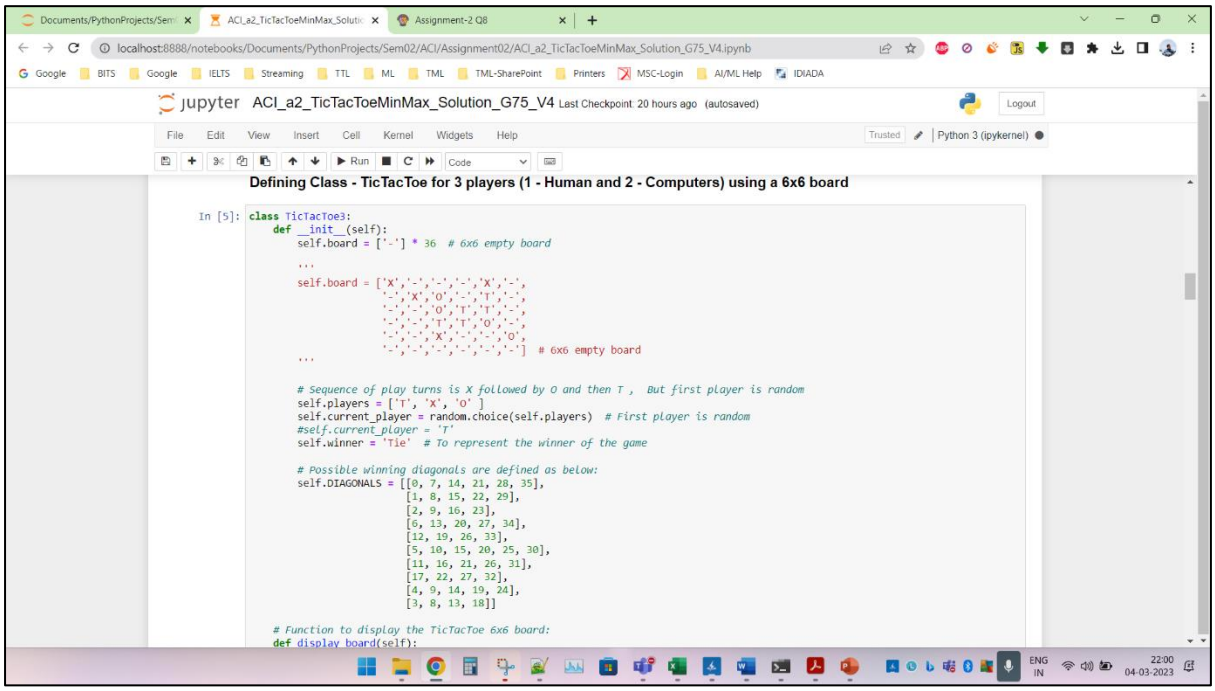


Figure 5: Class definition with initialisation block

Board: It is used to represent 36 cells for the game. The initial board is empty, and all cells are marked as '-' in the list defined. However, during display of board as output '-' is displayed as *blank* cell for better user experience.

Players: 3 Players in which, 1 Human and 2 AI Agents are modelled. The players of the game are represented by '**T**', '**X**' and '**O**' coins. The sequence of players' inputs is 'T' followed by 'X' followed by 'O'. The player to start the game is chosen randomly from the players list.

DIAGONALS: This constant is used to represent possible winning diagonals of the board as a list of lists (2D Array). We found 10 such possible wining diagonal combinations on the board. Considering winning combination requires at least 4 cells. Figure 6 highlights the possible diagonals.

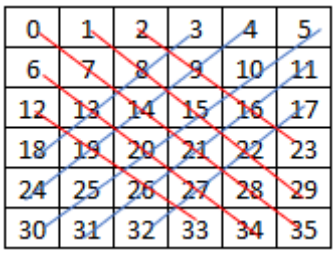
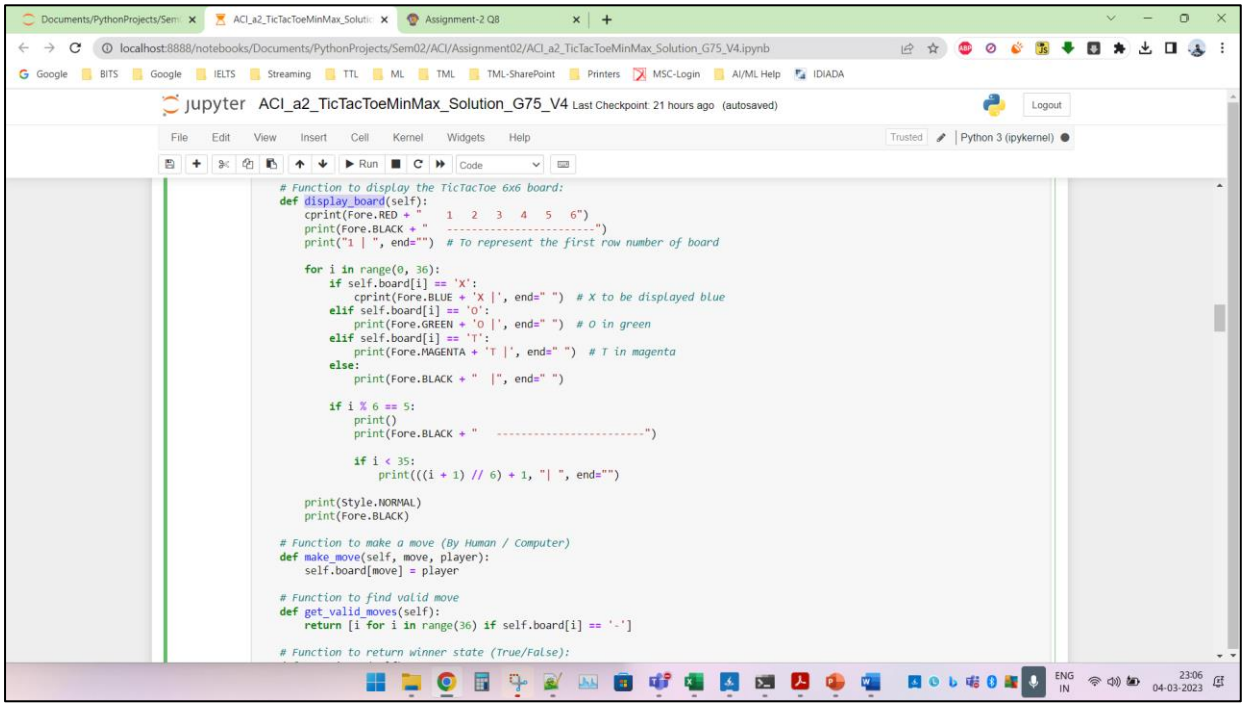


Figure 6: Possible winning diagonals

Winner: This attribute is initialized as a 'Tie', however it used to represent the Winning player as the game progresses.

CurrentPlayer: Initially this attribute holds a random player from the available players list. This is updated with the next player according to the sequence defined in the game algorithm.

Display_board Function:



```
# Function to display the TicTacToe 6x6 board:
def display_board(self):
    cprint(Fore.RED + "    1   2   3   4   5   6")
    print(Fore.BLACK + "   ------------------------")
    print("1 | ", end="")  # To represent the first row number of board

    for i in range(0, 36):
        if self.board[i] == 'X':
            cprint(Fore.BLUE + 'X |', end=" ")  # X to be displayed blue
        elif self.board[i] == 'O':
            print(Fore.GREEN + 'O |', end=" ")  # O in green
        elif self.board[i] == 'T':
            print(Fore.MAGENTA + 'T |', end=" ")  # T in magenta
        else:
            print(Fore.BLACK + "  |", end=" ")

        if i % 6 == 5:
            print()
            print(Fore.BLACK + "   ------------------------")

            if i < 35:
                print(((i + 1) // 6) + 1, "| ", end="")

    print(Style.NORMAL)
    print(Fore.BLACK)

# Function to make a move (By Human / Computer)
def make_move(self, move, player):
    self.board[move] = player

# Function to find valid move
def get_valid_moves(self):
    return [i for i in range(36) if self.board[i] == '-']

# Function to return winner state (True/False):
```

Figure 7: Function definition of display_board

This code defines a method called 'display_board' which is used to print the current state of the game board. The board is printed in the terminal using ASCII characters.

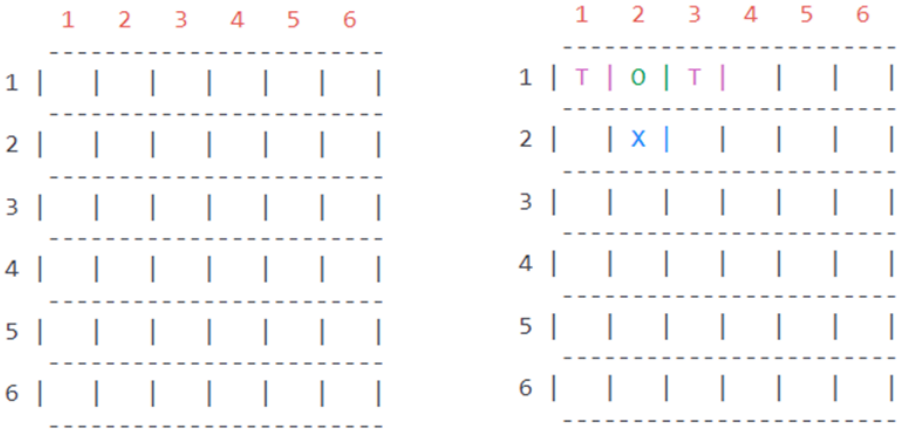


Figure 8: Display of 6x6 board

The method first prints the column numbers (1 to 6) in red color, using the 'cprint' method from the 'colorama' library. For each element of the board, the method checks if it is 'X', 'O', or 'T', and prints it in the corresponding color (blue for 'X', green for 'O', and magenta for 'T'). If the element is '-' (empty), it prints a blank space.

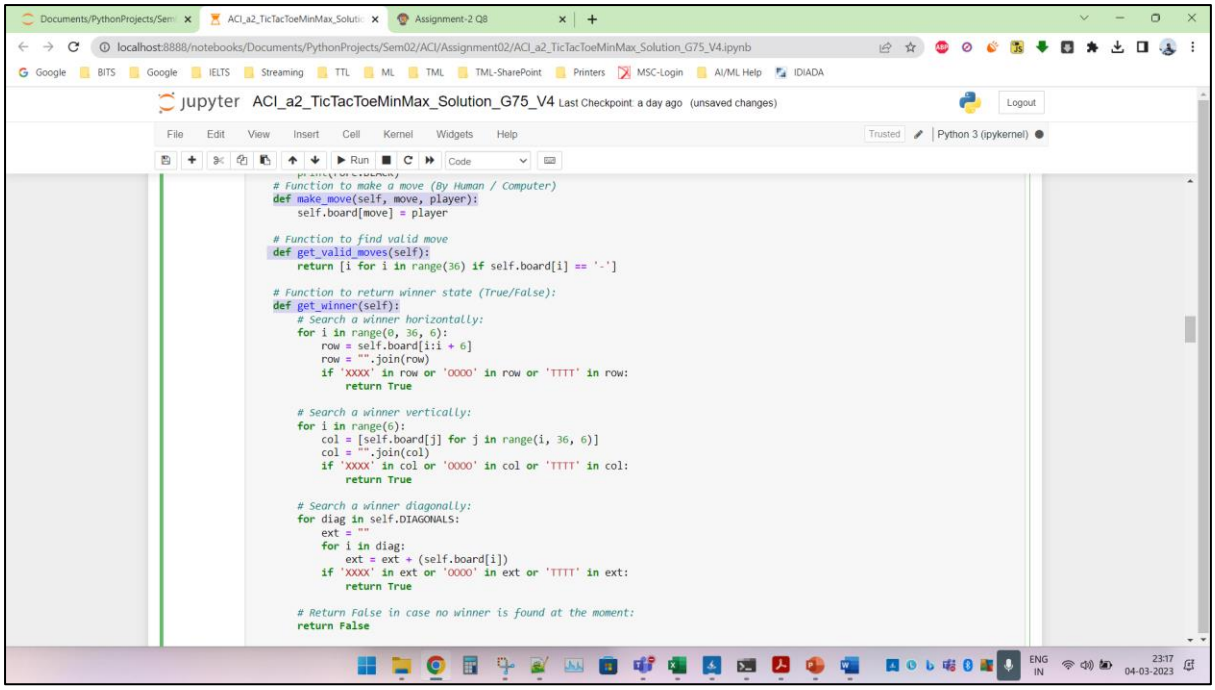


Figure 9: Make_move(), get_valid_moves() and get_winner() function definitions

Make_move Function:

This code defines a method called 'make_move' that updates the game board with a player's move. The method takes two arguments - *'move'* and *'player'*.

The *'move'* argument is an integer representing the index (0 to 35) of the element on the board where the player wants to make their move.

Get_valid_moves Function:

This code defines a method called 'get_valid_moves' that returns a list of valid moves which a player can make on the game board. A valid move is any position on the board which is currently empty (i.e., has a value of '-').

Get_winner Function:

This code defines a method called 'get_winner' that checks whether there is a winner in the current game state. The method returns True if there is a winner, and False otherwise.

The method first checks for a horizontal winner by iterating over the rows of the board (i.e., indices 0-5, 6-11, 12-17, 18-23, 24-29, 30-35) and concatenating the values in each row. If a row contains four consecutive 'X's, 'O's, or 'T's, then the method returns True to indicate that a winner is found.

In case, no winner is found during row-wise testing, the method then checks for a vertical winner by iterating over the columns of the board (i.e., indices [0, 6, 12, 18, 24,30], [1,7,13,19,25,31].....[5,11,17,23,29,35]) and concatenating the values in each column. If a column contains four consecutive 'X's, 'O's, or 'T's, the method returns True to indicate that a winner is found.

Finally, if column winner testing function also does not return True value, then the method checks for a diagonal winner by iterating over the diagonal combinations defined in the 'DIAGONALS' list. For each diagonal combination, the method concatenates the values in the corresponding positions on the board. . If a column contains four consecutive 'X's, 'O's, or 'T's, the method returns True to indicate that a winner is found.

If no winner is found after checking all possible combinations, then the method returns False to indicate that there is no winner at the current game state.

This method can be called during the game after each player makes a move to check if that move was a winning move. If the method returns True, the game can be ended and the winner can be declared. If the method returns False, the game can continue until a winner is found or all cells are occupied which leads to a TIE.

Minimax Function:

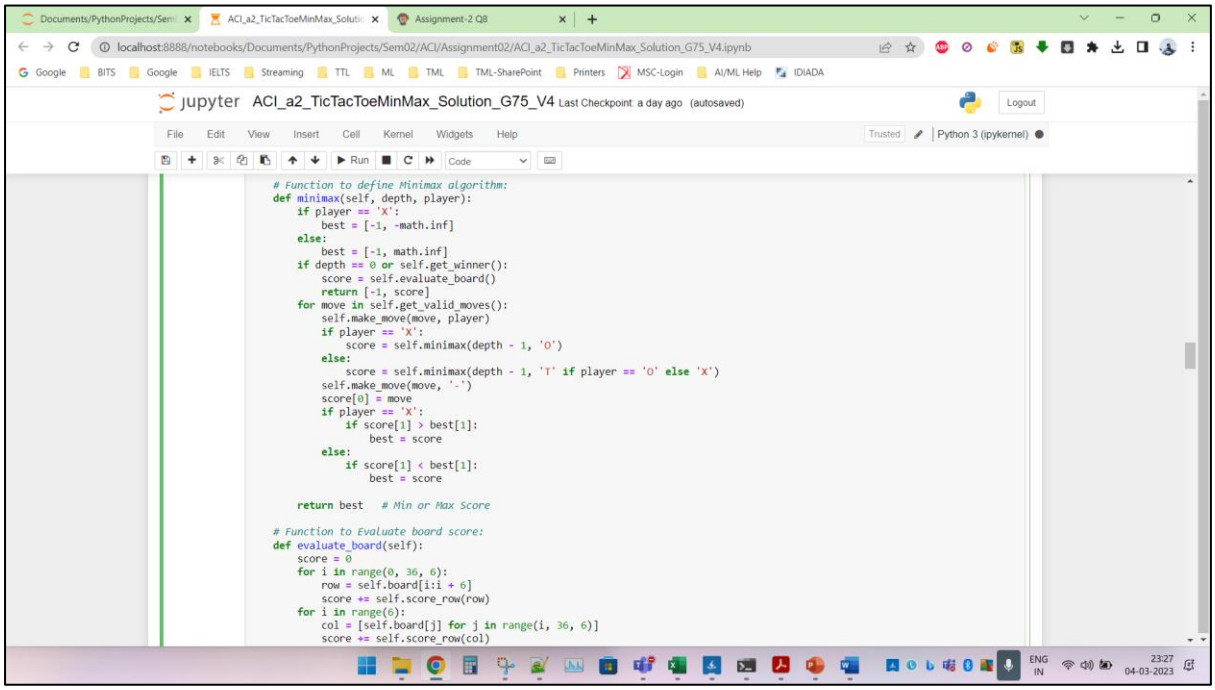


Figure 10: Minimax() function definition

The minimax function used in the Tic-Tac-Toe game program utilizes the minimax algorithm to find the best move for the current player. The algorithm recursively evaluates all possible future moves for each player until a terminal state is reached, i.e., either the game ends or the maximum depth of search is reached. At each level of recursion, the function evaluates the best move by calling itself with the opponent player and decreasing the depth by 1.

When a terminal state is reached or the maximum depth is reached, the function returns the score of the current board evaluation. The function then backtracks and returns the move with the highest or lowest score, depending on whether the current player is maximizing or minimizing. The move is returned as a list, with the move index as the first element and the score as the second element.

If it's the turn of the MAX_PLAYER, the function initializes the *'best_val'* variable to -INFINITY and loops over all legal moves on the board. For each move, it creates a new board with the move applied and recursively calls minimax with the new board, a reduced depth, and the MIN_PLAYER as the next player. It then updates the *'best_val'* variable with the maximum score obtained from any of the moves and returns it.

If it's the turn of the MIN_PLAYER, the function initializes the *'best_val'* variable to +INFINITY and loops over all legal moves on the board. For each move, it creates a new board with the move applied and recursively calls minimax with the new board, a reduced depth, and the MAX_PLAYER as the next player. It then updates the *'best_val'* variable with the minimum score obtained from any of the moves and returns it.

Evaluate_board Function:

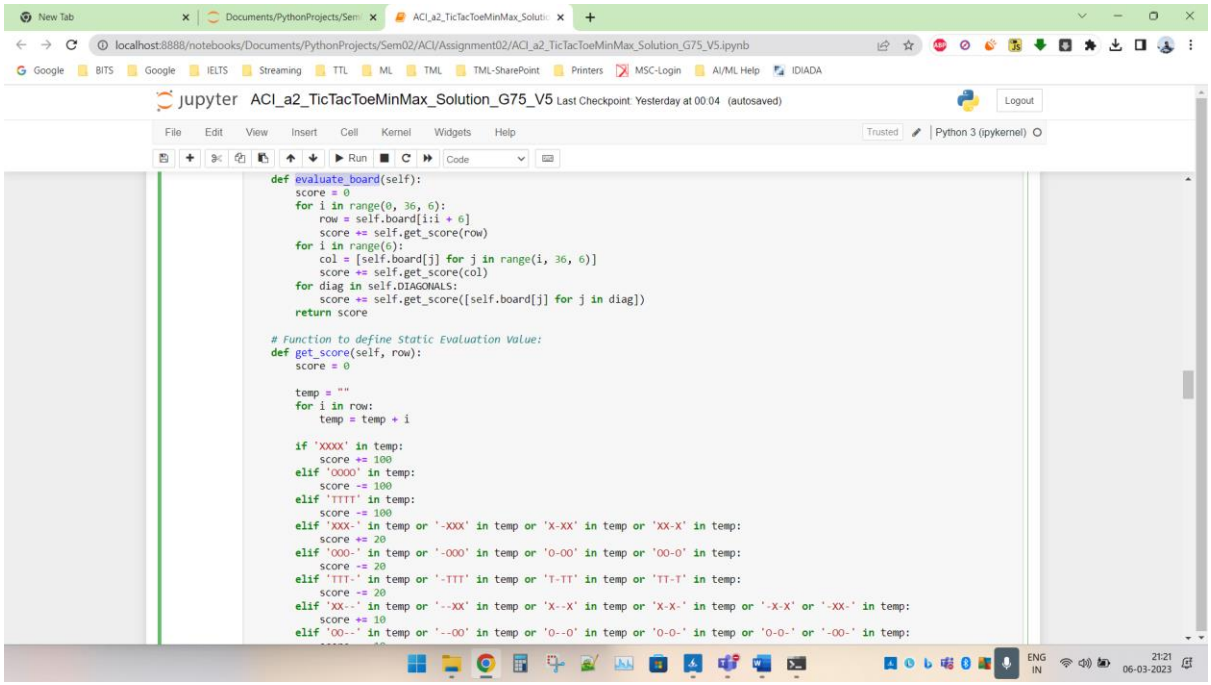


Figure 11: Evaluate_board function definition.

The evaluate_board function in a Tic-Tac-Toe game is a function that takes a board as input and returns a score indicating the desirability of the current board state for the current player. The score returned by the evaluate_board function is typically calculated based on the number of possible winning combinations for the current player on the board.

Get_score Function:

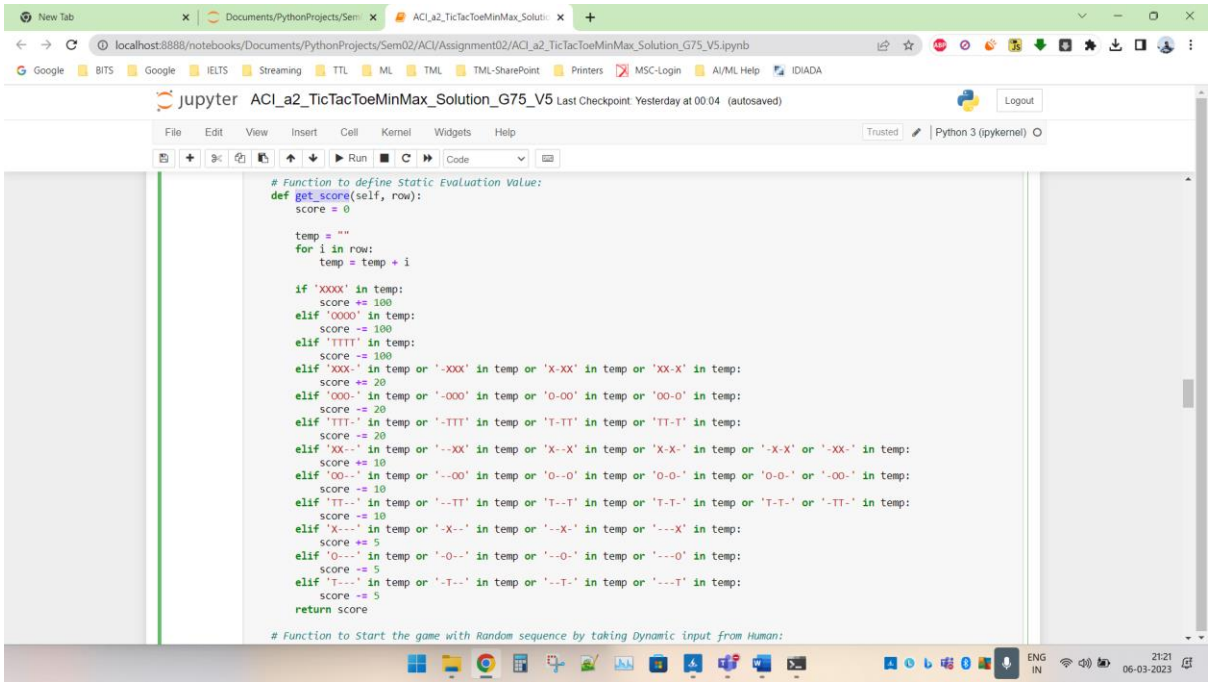


Figure 12: get_score() function definition.

Get_score() function is called through evaluate_board() function to estimate row wise, column wise and diagonal wise scores.

For Tic-Tac-Toe, the score metric could be as simple as returning +1 if the player wins, -1 if the computer wins, or 0 otherwise. However, simple evaluation function may require deeper search and with 6x6 Tic-Tac-Toe it may result in algorithm to get stuck for a long time.

A better evaluation function for Tic-Tac-Toe which is used in the algorithm is as follows:

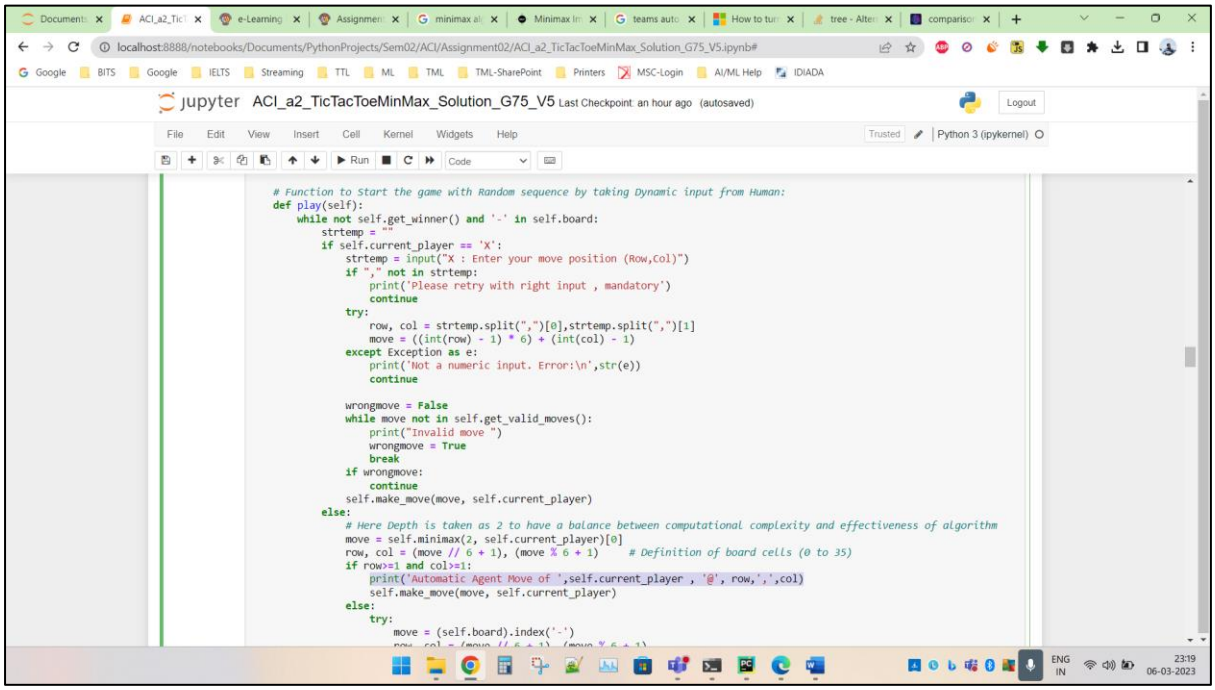
+100 for each 4-in-a-line for player (X)
-100 for each 4-in-a-line for computer (O,T)

+20 for each 3-in-a-line (with 1 empty cell) for player (X)
-20 for each 3-in-a-line (with 1 empty cell) for computer (O,T)

+10 for each 2-in-a-line (with 2 empty cells) for player (X)
-10 for each 2-in-a-line (with 2 empty cells) for computer (O,T)

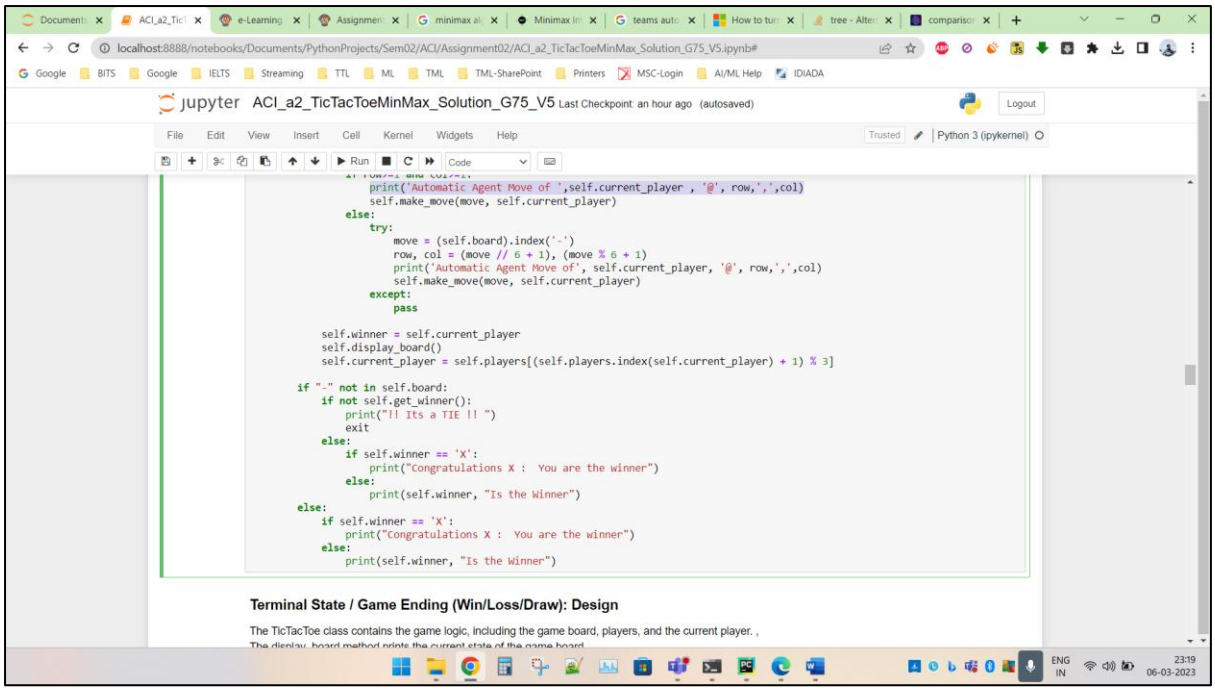
+5 for each 1-in-a-line (with 3 empty cells) for player (X)
-5 for each 1-in-a-line (with 3 empty cells) for computer (O,T)

Play Function:



```
# Function to Start the game with Random sequence by taking Dynamic input from Human:
def play(self):
    while not self.get_winner() and '-' in self.board:
        strtemp = ""
        if self.current_player == 'X':
            strtemp = input("X : Enter your move position (Row,Col)")
            if "," not in strtemp:
                print('Please retry with right input , mandatory')
                continue
            try:
                row, col = strtemp.split(",")[0],strtemp.split(",")[1]
                move = ((int(row) - 1) * 6) + (int(col) - 1)
            except Exception as e:
                print('Not a numeric input. Error:\n',str(e))
                continue

            wrongmove = False
            while move not in self.get_valid_moves():
                print("Invalid move ")
                wrongmove = True
                break
            if wrongmove:
                continue
            self.make_move(move, self.current_player)
        else:
            # Here Depth is taken as 2 to have a balance between computational complexity and effectiveness of algorithm
            move = self.minimax(2, self.current_player)[0]
            row, col = (move // 6 + 1), (move % 6 + 1)      # Definition of board cells (0 to 35)
            if row>=1 and col>=1:
                print('Automatic Agent Move of ',self.current_player , '@', row,',',col)
                self.make_move(move, self.current_player)
            else:
                try:
                    move = (self.board).index('-')
```



```
                print('Automatic Agent Move of ',self.current_player , '@', row,',',col)
                self.make_move(move, self.current_player)
            else:
                try:
                    move = (self.board).index('-')
                    row, col = (move // 6 + 1), (move % 6 + 1)
                    print('Automatic Agent Move of', self.current_player, '@', row,',',col)
                    self.make_move(move, self.current_player)
                except:
                    pass

        self.winner = self.current_player
        self.display_board()
        self.current_player = self.players[(self.players.index(self.current_player) + 1) % 3]

    if "-" not in self.board:
        if not self.get_winner():
            print("!! Its a TIE !! ")
            exit
        else:
            if self.winner == 'X':
                print("Congratulations X :  You are the winner")
            else:
                print(self.winner, "Is the Winner")
    else:
        if self.winner == 'X':
            print("Congratulations X :  You are the winner")
        else:
            print(self.winner, "Is the Winner")
```

Terminal State / Game Ending (Win/Loss/Draw): Design

The TicTacToe class contains the game logic, including the game board, players, and the current player. ,

Figure 13: Play() function definition (2 Part screenshot)

Play() function is called to start the Tic-Tac-Toe game. This function runs in a loop till a winner is found or the game ends in a tie. This function asks for dynamic input from Human (X coin) as per the sequence of players (With random start player). During each turn, the code prompts the player to enter their move as a row and column number separated by a comma. It then converts this input into a move index (0 to 35) and checks if it is a valid move by calling the get_valid_moves method. If the move is valid, it is made by calling the make_move method and the board is displayed by calling display_board() function.

In the algorithm, ***move = self.minimax(2, self.current_player)[0]*** calls the minimax function with depth = 2 and player = current player.

In Tic-Tac-Toe game as per the problem statement, the maximum depth of the game tree is 36, which represents the total number of cells on the board (6x6 board). At each level of the game tree, the algorithm considers all possible moves and evaluates the resulting game state by assigning a score to each outcome. The score is based on the assumption that both players will play optimally.

The significance of the depth in Tic-Tac-Toe Minimax game is that it determines the computational complexity and the effectiveness of the algorithm. A higher depth means that the algorithm considers more moves ahead, which results in a more accurate evaluation of the game state but also increases the computational complexity of the algorithm.

In practice, a depth of 2 or 3 is commonly used for Tic-Tac-Toe Minimax game, as it strikes a balance between computational complexity and effectiveness. A depth of 2 or 3 allows the algorithm to make good decisions while keeping the response time reasonable for real-time gameplay. However, the optimal depth may depend on factors such as the computer's processing power and the specific implementation of the algorithm.

If the board is full but no winner has been found, the game is a tie. If a winner is found, the game displays a congratulatory message for the winner.

Sample Input Sequences & Output:

Run 1: Random player to start is chosen as Human (X Coin). Empty board is as below:

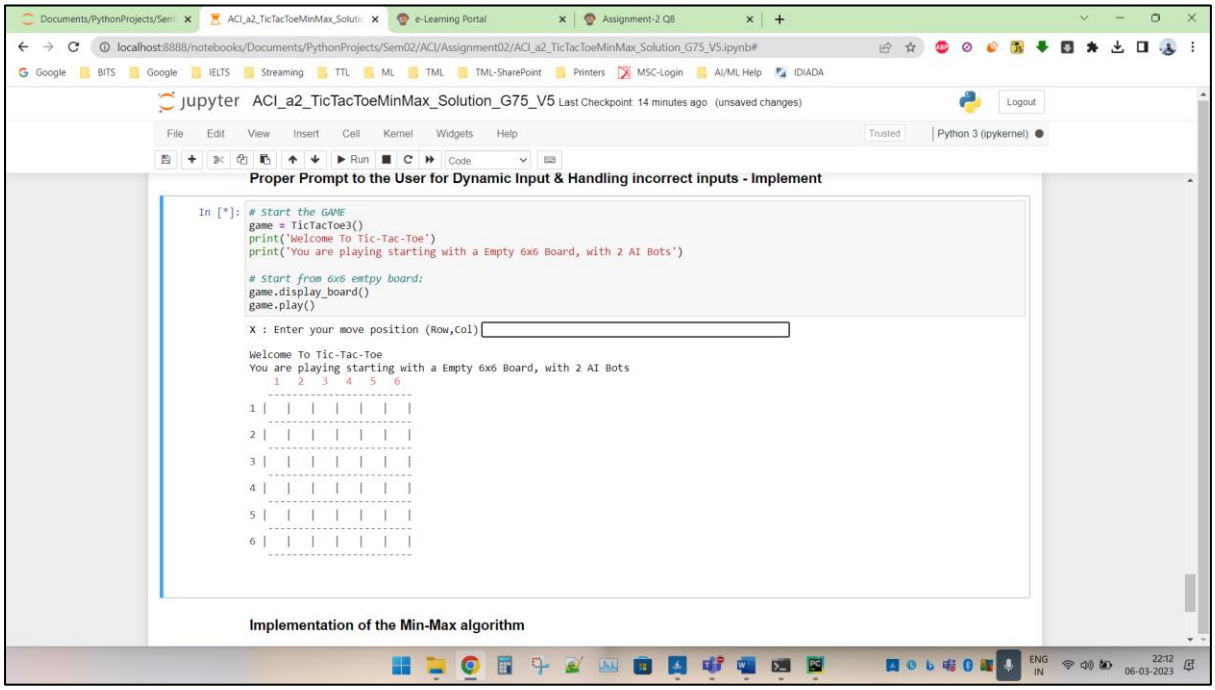


Figure 14: Empty 6x6 Board

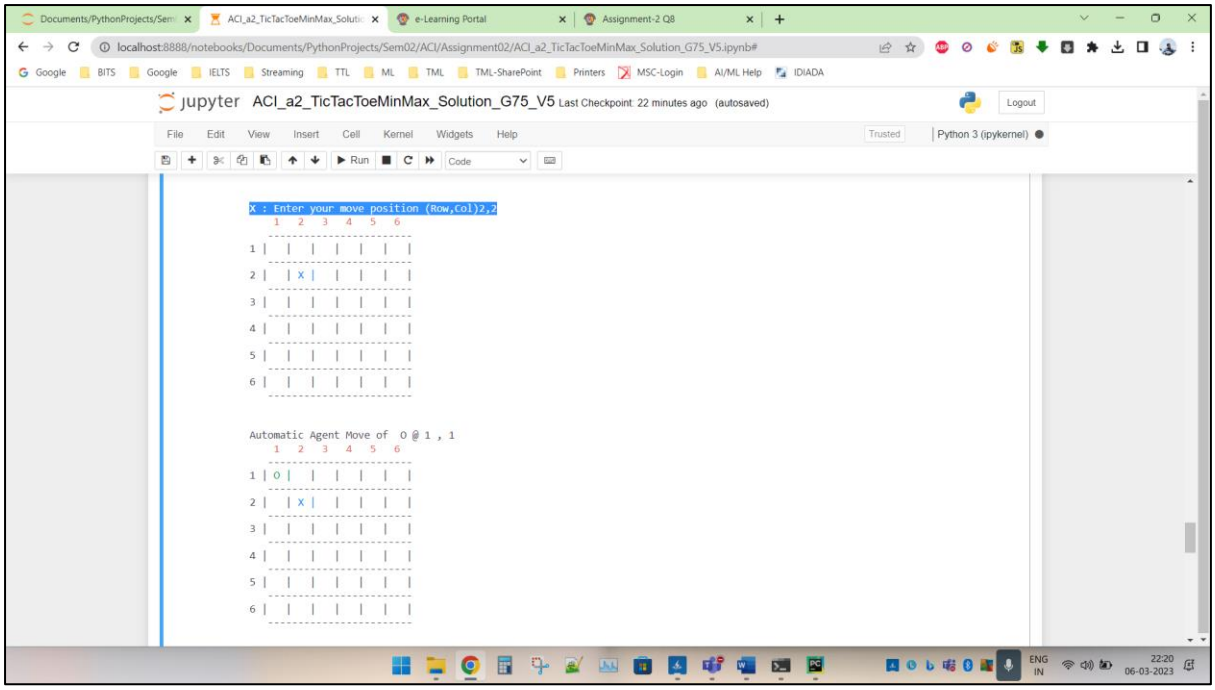


Figure 15: Human (X) input = 2,2 and AI Input (O) = 1,1

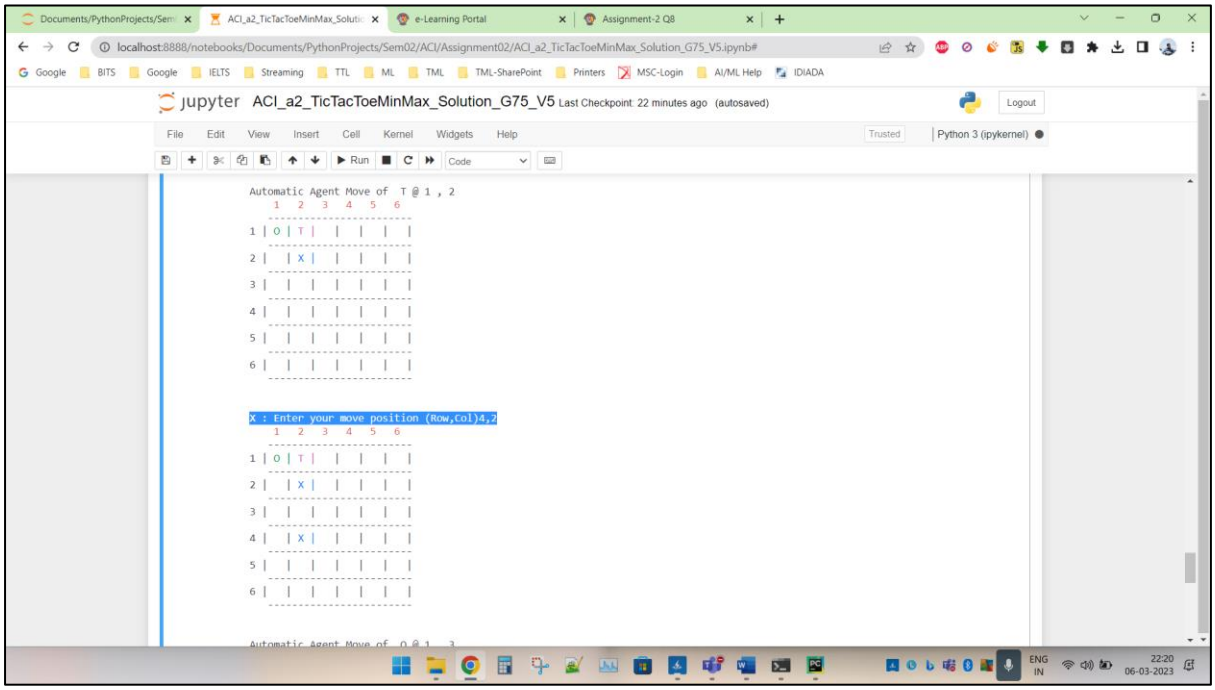


Figure 16: AI (T) Input = 1,2 and Human (X) input = 4,2

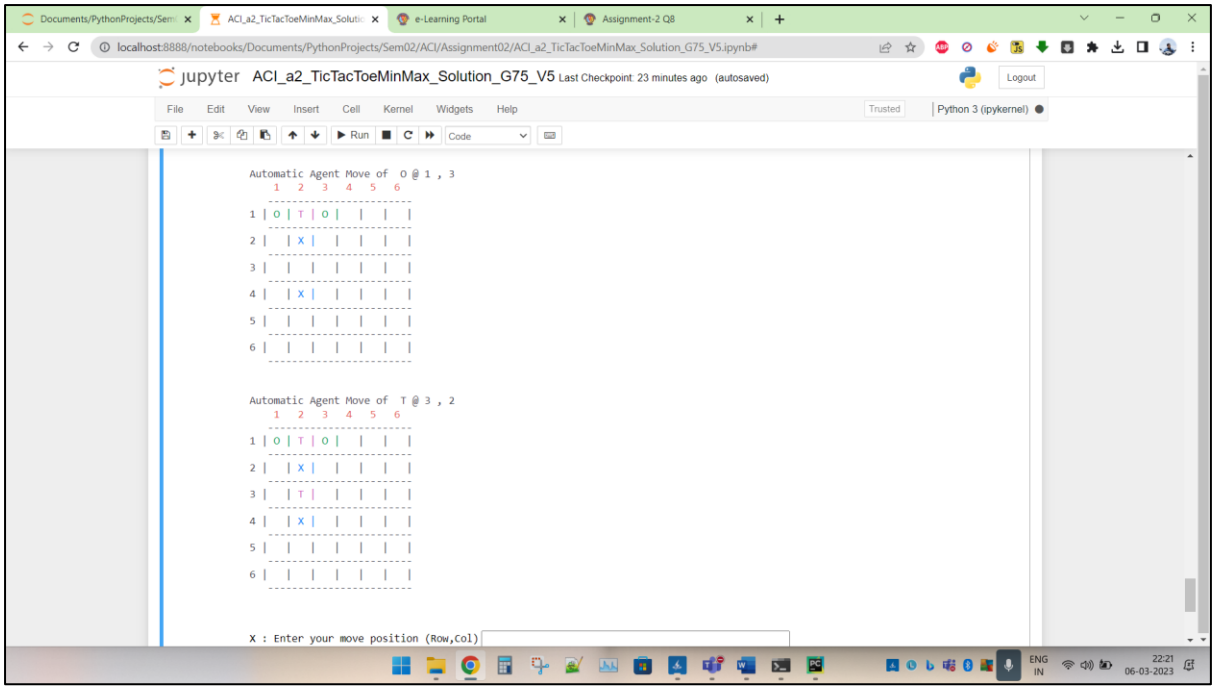


Figure 17: AI (O) input = 1,3 and AI (T) input = 3,2

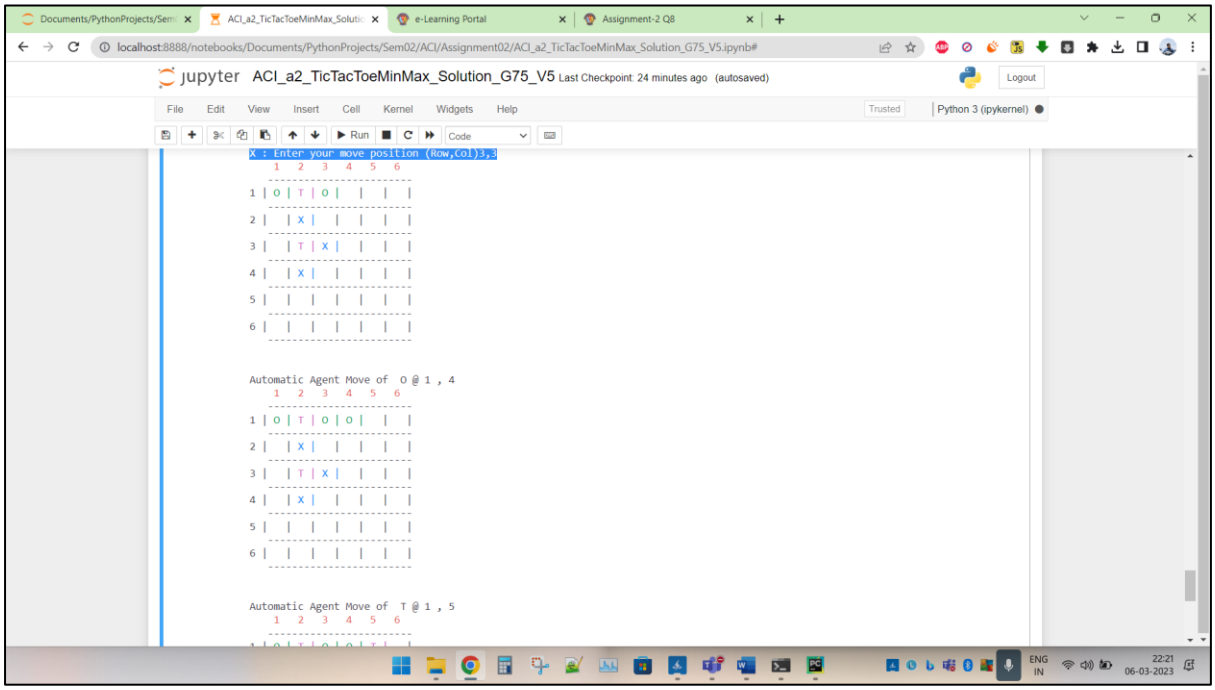


Figure 18: Human (X) input = 3,3 and AI (O) input = 1,4

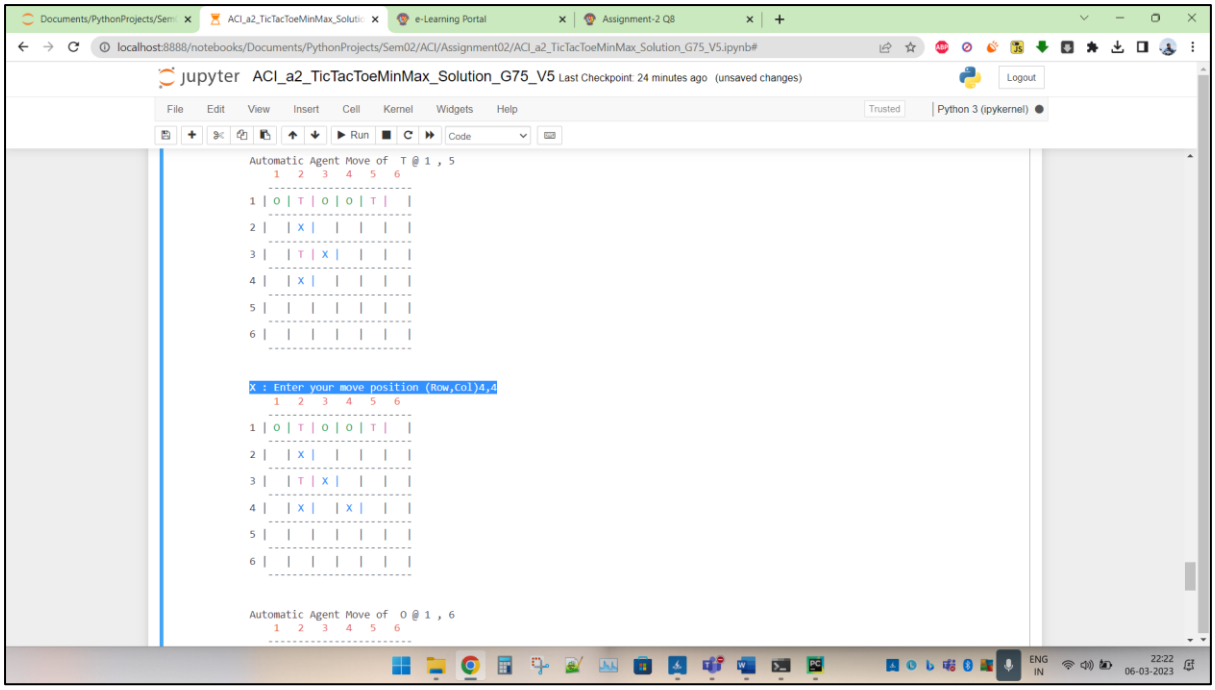


Figure 19: AI (T) input = 1,5 and Human (X) input = 4,4

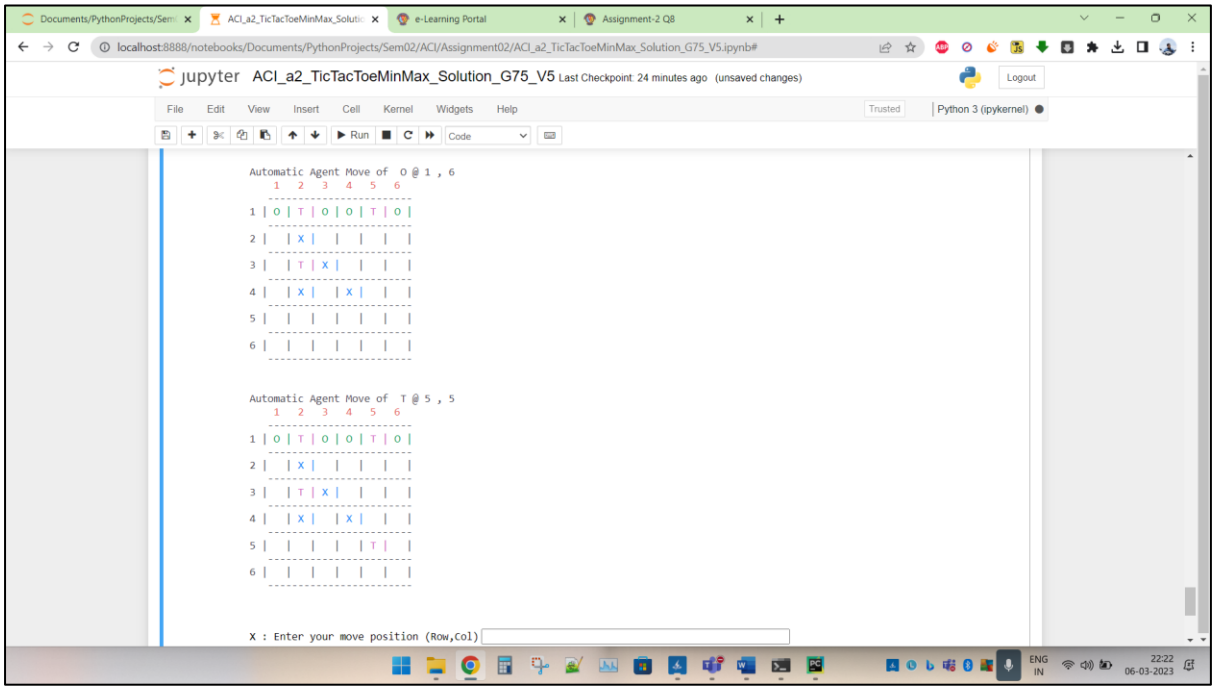


Figure 20: AI (O) input = 1,6 and AI (T) input = 5,5

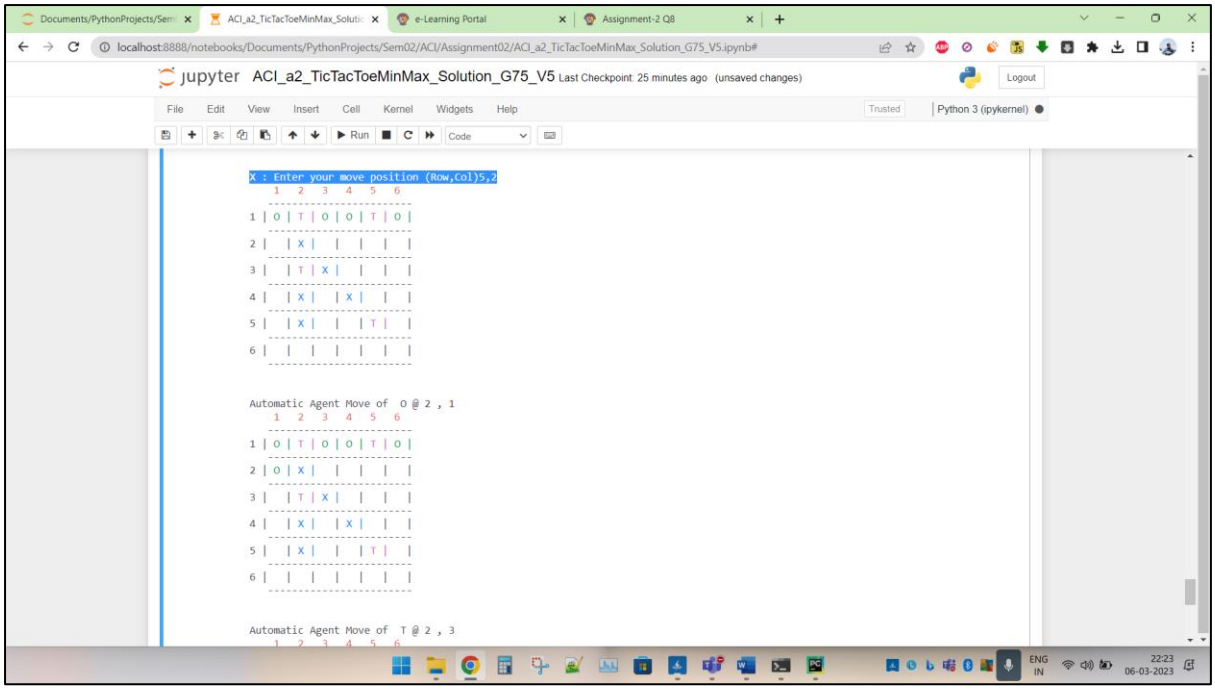


Figure 21: Human (X) input = 5,2 and AI (O) input = 2,1

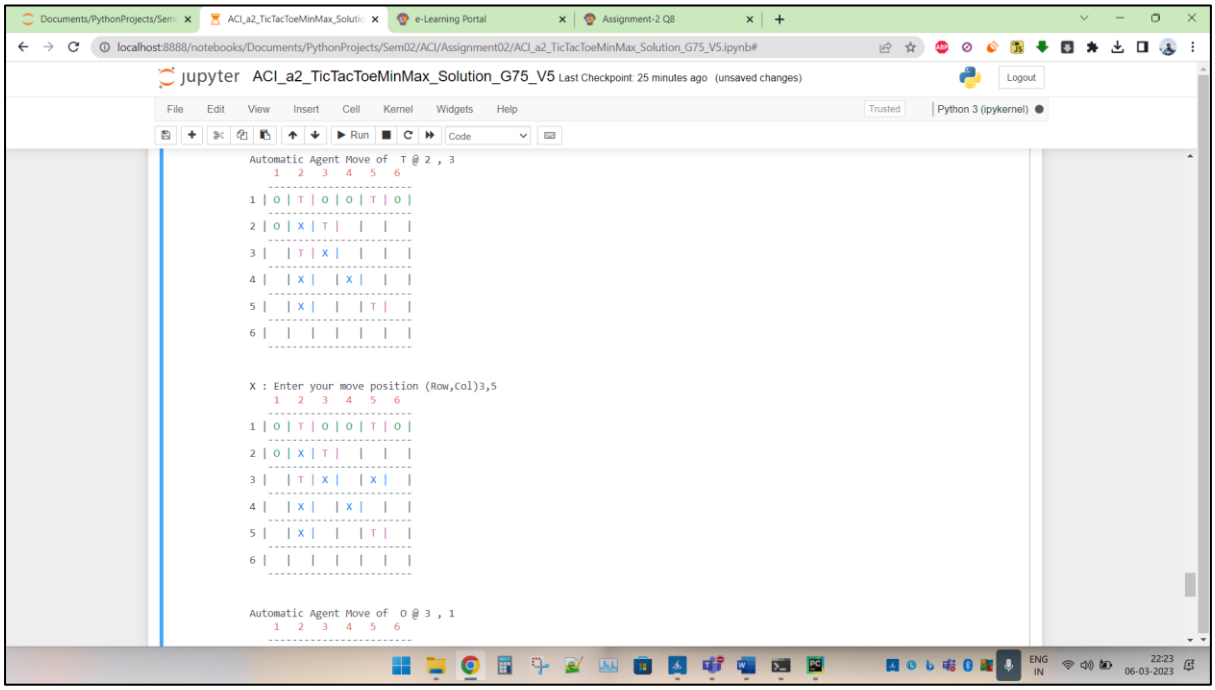


Figure 22: AI (T) input = 2,3 and Human (X) input = 3,5

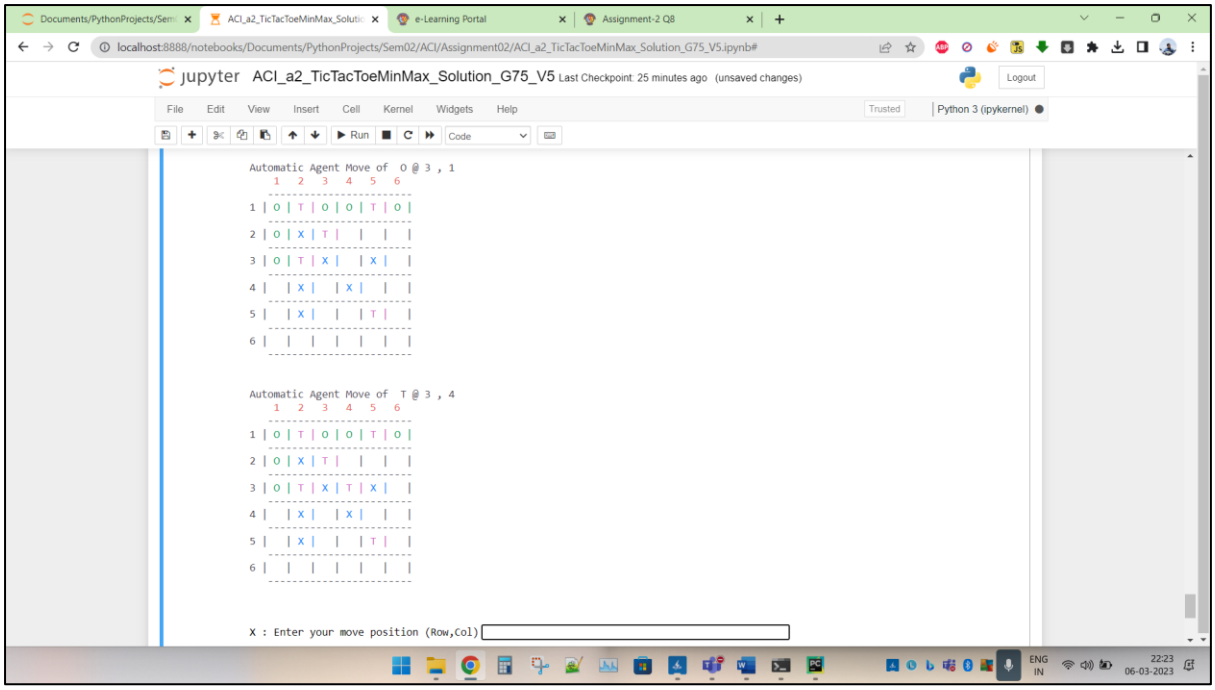


Figure 23: AI (O) input = 3,1 and AI (T) input = 3,4

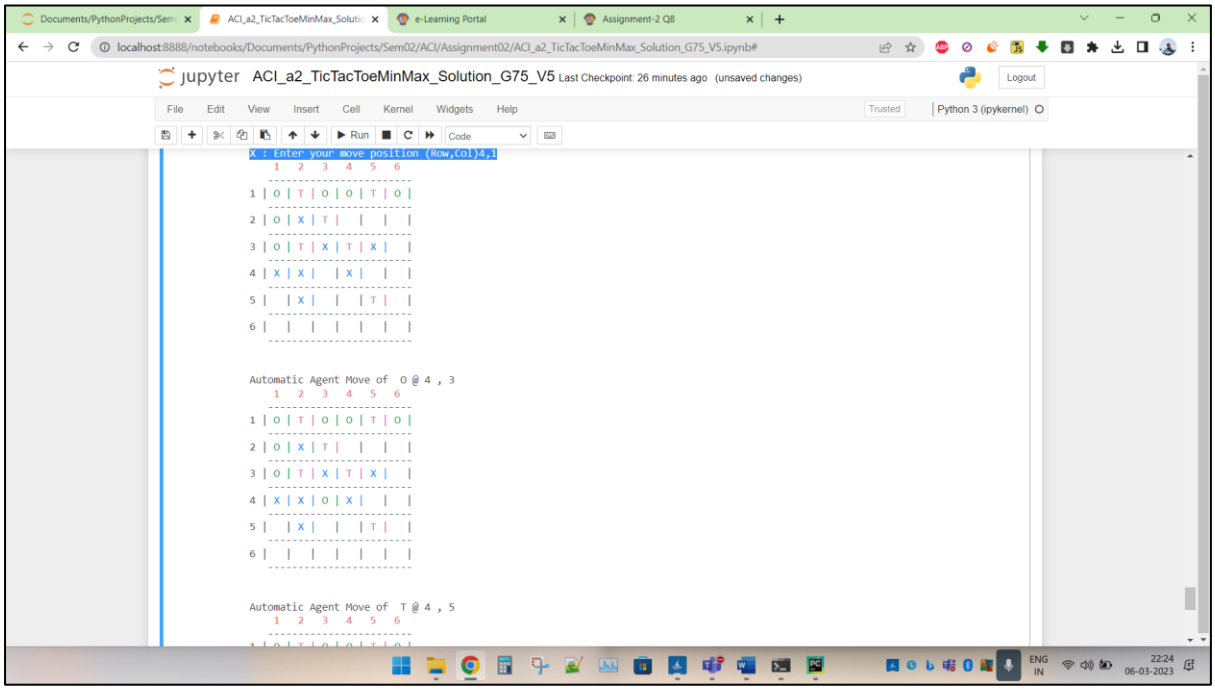


Figure 24: Human (X) input = 4,1 and AI (O) input = 4,3

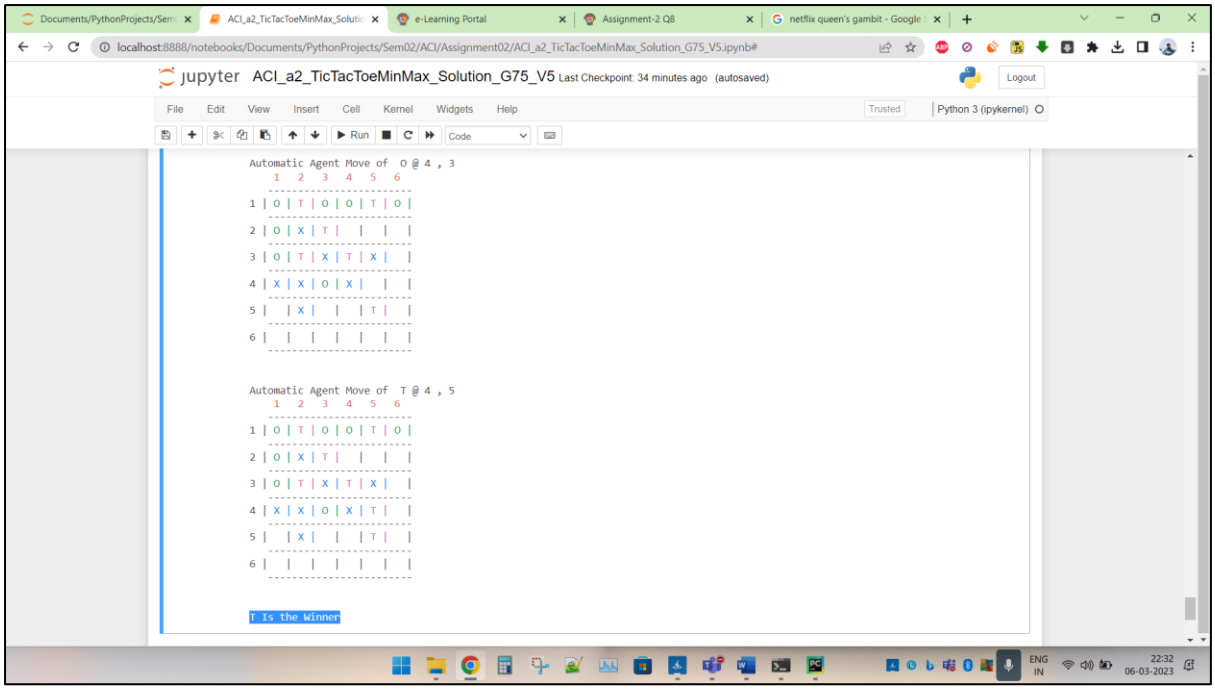


Figure 25: AI (T) input = 4,5 leads to T winning the game

Summary:

The 6x6 grid based Tic-Tac-Toe problem is implemented successfully in Python with 3 players and using minimax algorithm with a depth constraint of 2. The 6x6 grid allows for more potential winning combinations, making the game more challenging and exciting. Additionally, with three players, there is more strategy involved in trying to block opponents and create opportunities for oneself.

Future enhancements can be made in such a way that, we can give an option to the end user to choose the complexity level of AI agents and then play the game. This can be done by creating different scoring functions corresponding to the level of difficulty chosen by the user.

Minimax is a popular algorithm used to determine the optimal move for a player in games like Tic Tac Toe. However, there are alternative algorithms that can be used to achieve similar results. Here are some options:

Alpha-Beta pruning: This algorithm is similar to Minimax but adds a technique called pruning, which allows it to eliminate certain branches of the game tree that are guaranteed to be worse than other branches. This can result in faster and more efficient computation of the optimal move.

Monte Carlo Tree Search (MCTS): This algorithm uses random simulations to determine the optimal move. It works by building a tree of possible moves and outcomes and then simulating random games from each node to determine the best path to victory.

Neural Networks: Another approach is to use neural networks to learn the optimal move based on training data. This involves training a model on a large dataset of previous games to predict the best move based on the current state of the game.

Rule-Based Systems: Alternatively, one can also use a rule-based system where a set of rules are defined to make moves based on the current state of the game. This approach involves defining a set of heuristics and strategies to guide the decision-making process.

Overall, while Minimax is a powerful algorithm for determining optimal moves in Tic Tac Toe, there are many other techniques that can be used to achieve similar results. The choice of algorithm depends on factors such as performance, accuracy, and ease of implementation.

Problem statement – 2

A sample decision tree for classifying the given software module as having defects or no-defects is given. Derive if-then rules from this tree and code these rules as Prolog rules. The Prolog code must take the attribute values as input and classify whether the module has defects using the decision tree as reference. The data-set for software defect prediction and its associated variable names are available in the announcement section for reference.

```
n <= 1: TRUE (81.0/4.0)
n > 1
|  i <= 122.55
|  |  l <= 0.05
|  |  |  i <= 39.38
|  |  |  |  ev(g) <= 1.4
|  |  |  |  |  d <= 19.5
|  |  |  |  |  |  iv(g) <= 2: TRUE (3.0)
|  |  |  |  |  |  iv(g) > 2
|  |  |  |  |  |  |  loc <= 50: FALSE (6.0)
|  |  |  |  |  |  |  loc > 50: TRUE (3.0/1.0)
|  |  |  |  |  d > 19.5: TRUE (37.0/6.0)
|  |  |  |  ev(g) > 1.4: TRUE (73.0/6.0)
|  |  |  i > 39.38
|  |  |  |  v(g) <= 3: FALSE (6.0)
|  |  |  |  v(g) > 3
|  |  |  |  |  iv(g) <= 8
|  |  |  |  |  |  l <= 0.04
|  |  |  |  |  |  |  ev(g) <= 14: FALSE (46.0/17.0)
|  |  |  |  |  |  |  ev(g) > 14: TRUE (3.0)
|  |  |  |  |  |  l > 0.04
|  |  |  |  |  |  |  i <= 71.94
|  |  |  |  |  |  |  |  d <= 21: TRUE (11.0/1.0)
|  |  |  |  |  |  |  |  d > 21: FALSE (3.0/1.0)
|  |  |  |  |  |  |  i > 71.94: FALSE (3.0)
|  |  |  |  |  iv(g) > 8: TRUE (47.0/11.0)
|  |  l > 0.05: FALSE (436.0/218.0)
|  i > 122.55: TRUE (33.0)
```

Figure 26: ACI Assignment 2 - Question 2 Decision Tree

Introduction

Prolog

Prolog is a programming language that is primarily used for implementing logic-based systems. Prolog stands for "Programming in Logic", and it is based on the idea of defining relationships and rules between objects and using logical inference to answer queries.

Prolog is particularly well-suited for tasks that involve symbolic manipulation, such as natural language processing, expert systems, and knowledge representation. It is also commonly used in the field of artificial intelligence and automated reasoning, as it provides a powerful set of tools for dealing with complex logical problems.

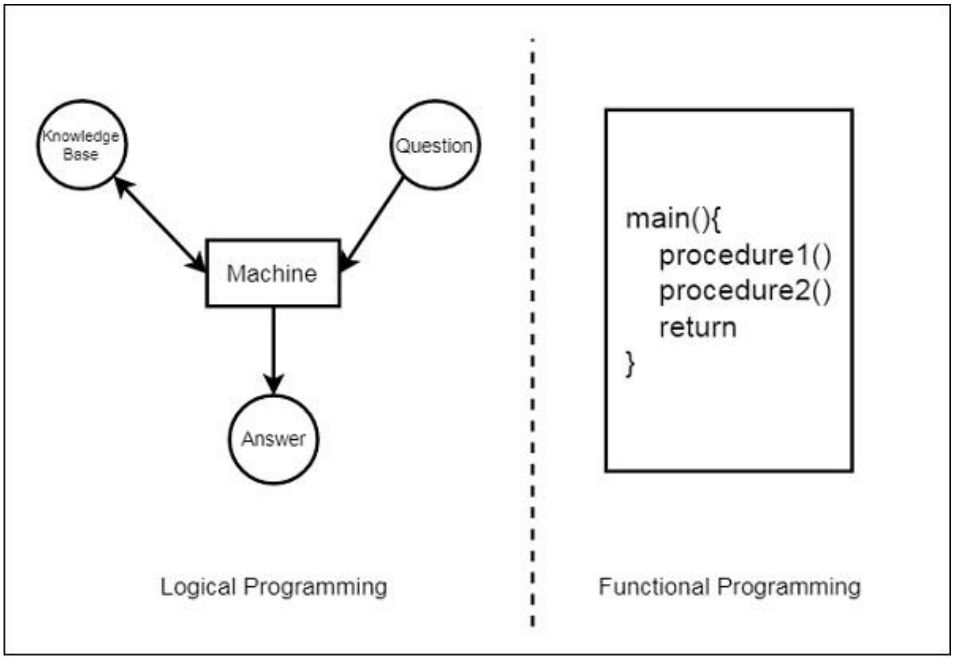


Figure 27: Logic and Functional Programming

Decision Tree

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

A decision tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. Decision tree learning employs a divide and conquer strategy by conducting a greedy search to identify the optimal split points within a tree. This process of splitting is then repeated in a top-down, recursive manner until all, or the majority of records have been classified under specific class labels.

In Prolog, decision trees can be used to implement decision-making systems that can reason about complex logical problems and provide advice or recommendations based on the available information.

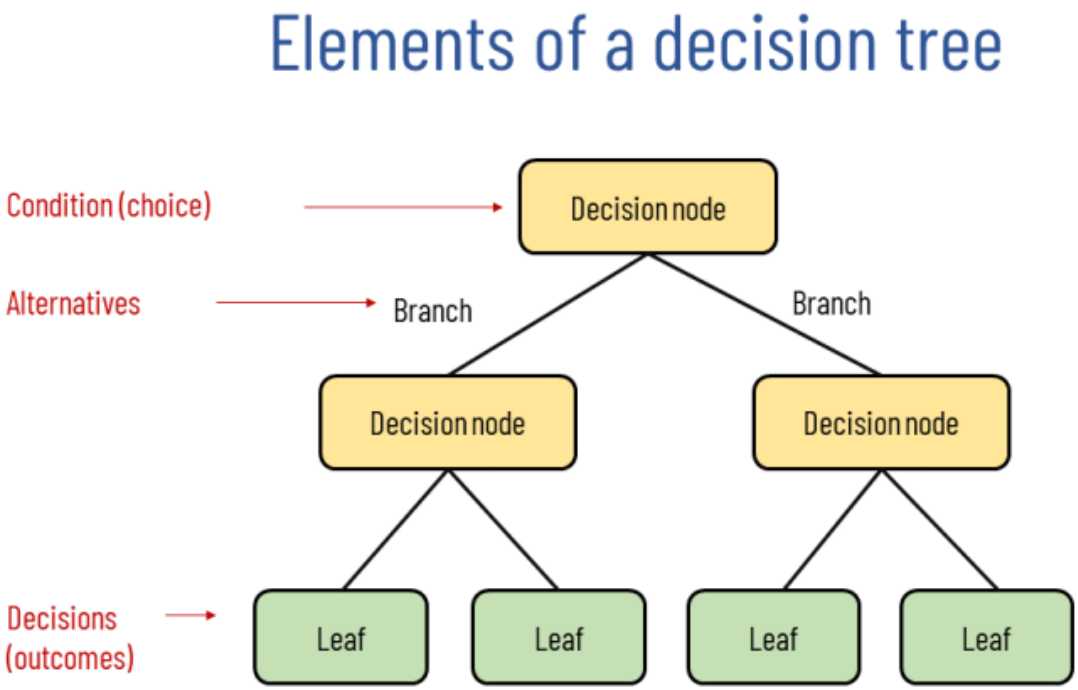


Figure 28 : Decision Tree Structure

Language/Tools Used

- SWI Prolog
- Excel

Inputs

The Decision tree for classifying the given software module as having defects or no-defects is given. Following 1 to 12 attributes are taken as input and 13th attribute is the output: defects = True/False

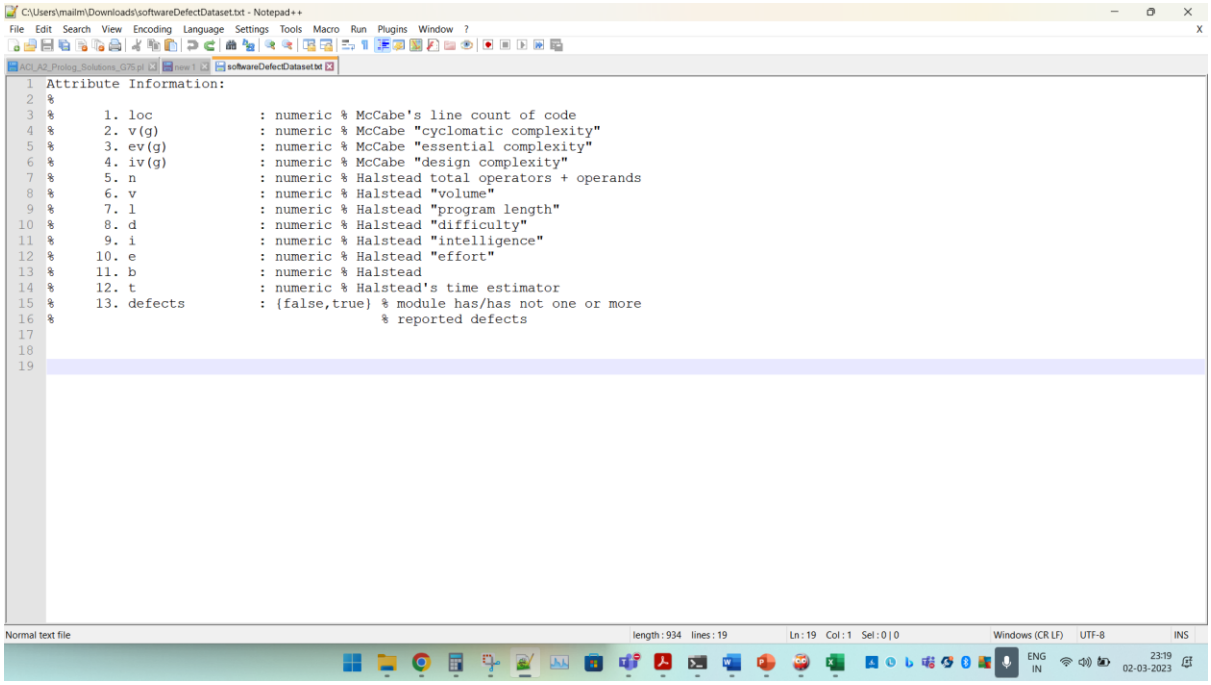


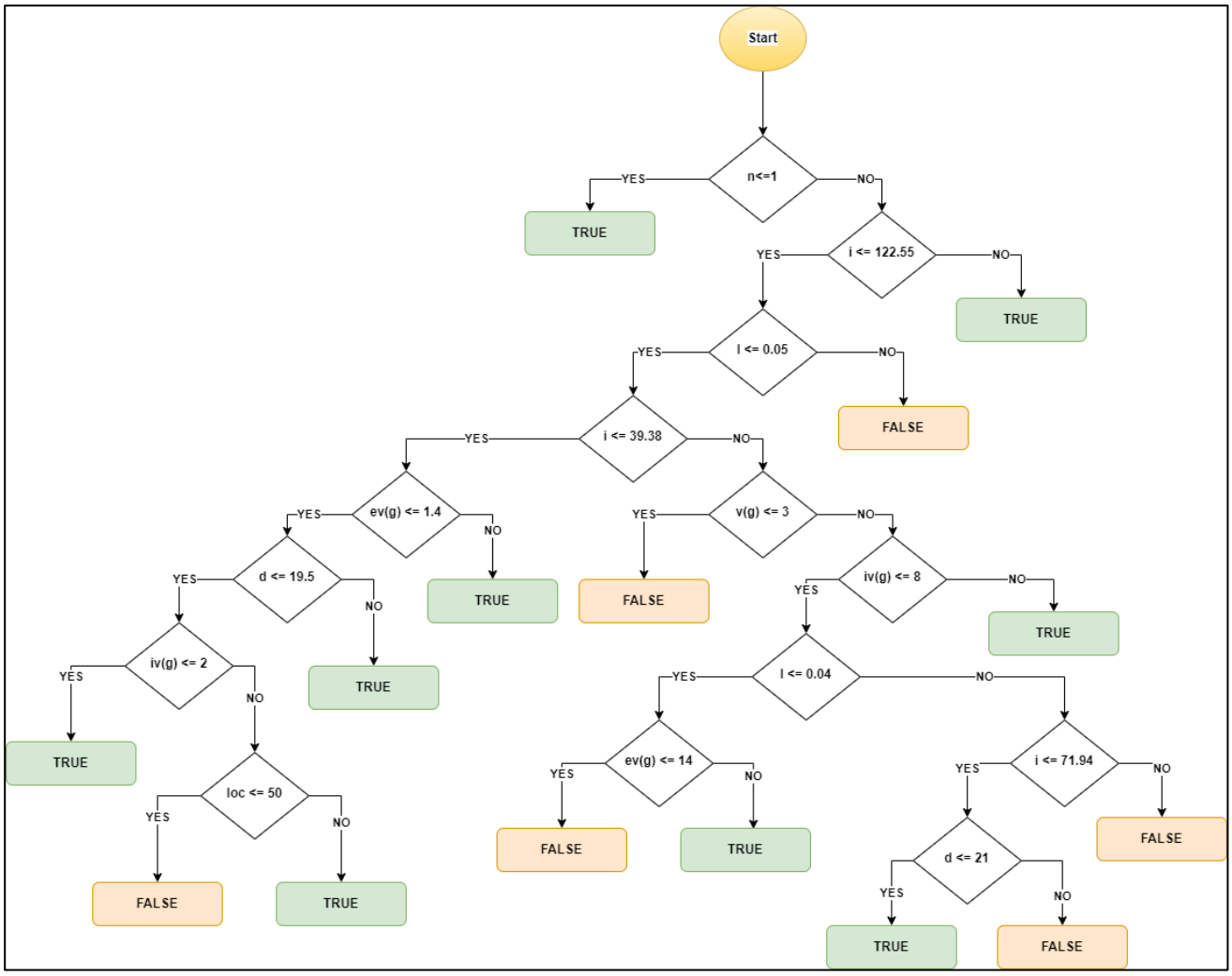
Figure 29: Attributes as per problem statement

```
n <= 1: TRUE (81.0/4.0)
n > 1
| i <= 122.55
| | l <= 0.05
| | | i <= 39.38
| | | | ev(g) <= 1.4
| | | | | d <= 19.5
| | | | | | iv(g) <= 2: TRUE (3.0)
| | | | | | iv(g) > 2
| | | | | | | loc <= 50: FALSE (6.0)
| | | | | | | loc > 50: TRUE (3.0/1.0)
| | | | | d > 19.5: TRUE (37.0/6.0)
| | | | ev(g) > 1.4: TRUE (73.0/6.0)
| | | i > 39.38
| | | | v(g) <= 3: FALSE (6.0)
| | | | v(g) > 3
| | | | | iv(g) <= 8
| | | | | | l <= 0.04
| | | | | | | ev(g) <= 14: FALSE (46.0/17.0)
| | | | | | | ev(g) > 14: TRUE (3.0)
| | | | | | l > 0.04
| | | | | | | i <= 71.94
| | | | | | | | d <= 21: TRUE (11.0/1.0)
| | | | | | | | d > 21: FALSE (3.0/1.0)
| | | | | | | i > 71.94: FALSE (3.0)
| | | | | iv(g) > 8: TRUE (47.0/11.0)
| | l > 0.05: FALSE (436.0/218.0)
| i > 122.55: TRUE (33.0)
```

Figure 30: Input Decision Tree as per problem statement

Solution

Step 1: Derived a flowchart corresponding if-then rules of the given decision tree.



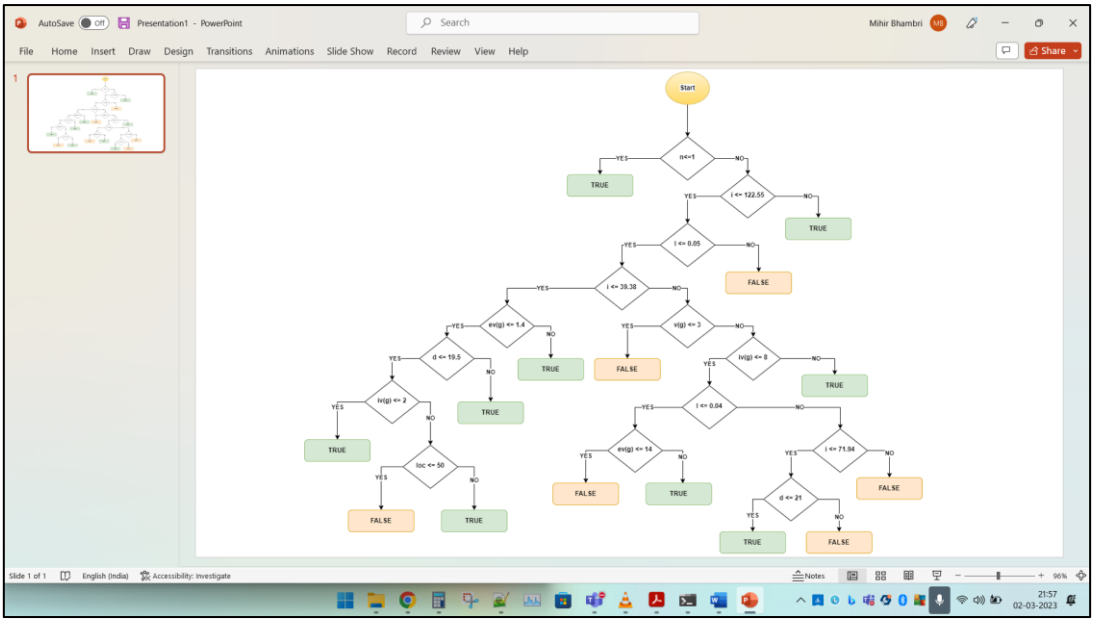


Figure 31: If-Then rules flowchart

Step 2: Prolog rules derived from the flowchart if-then rules is as below:

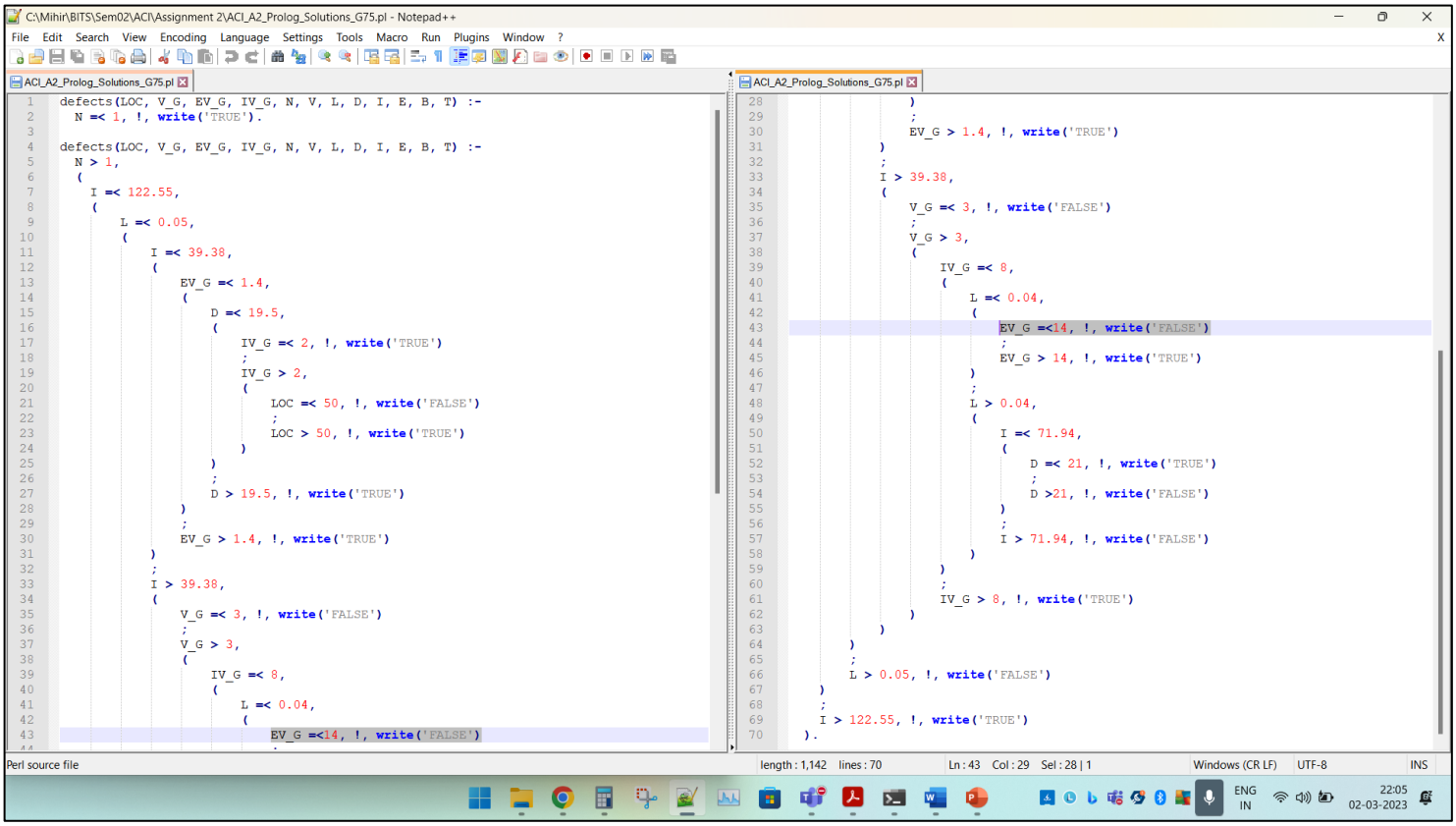


Figure 32: Prolog Rules as per given Decision Tree

For loading the .pl file, following syntax is used:

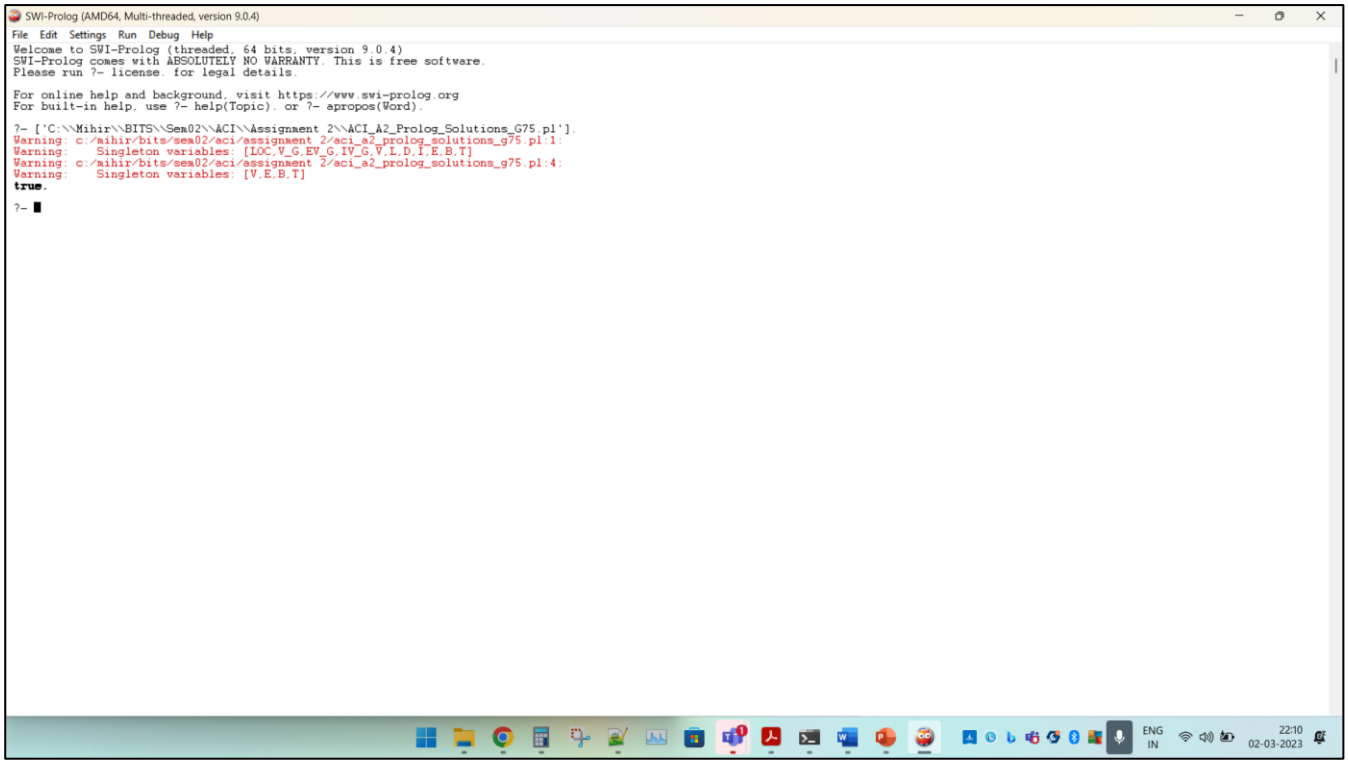


Figure 33: Loading .pl file in SWI Prolog

Output Test Cases:

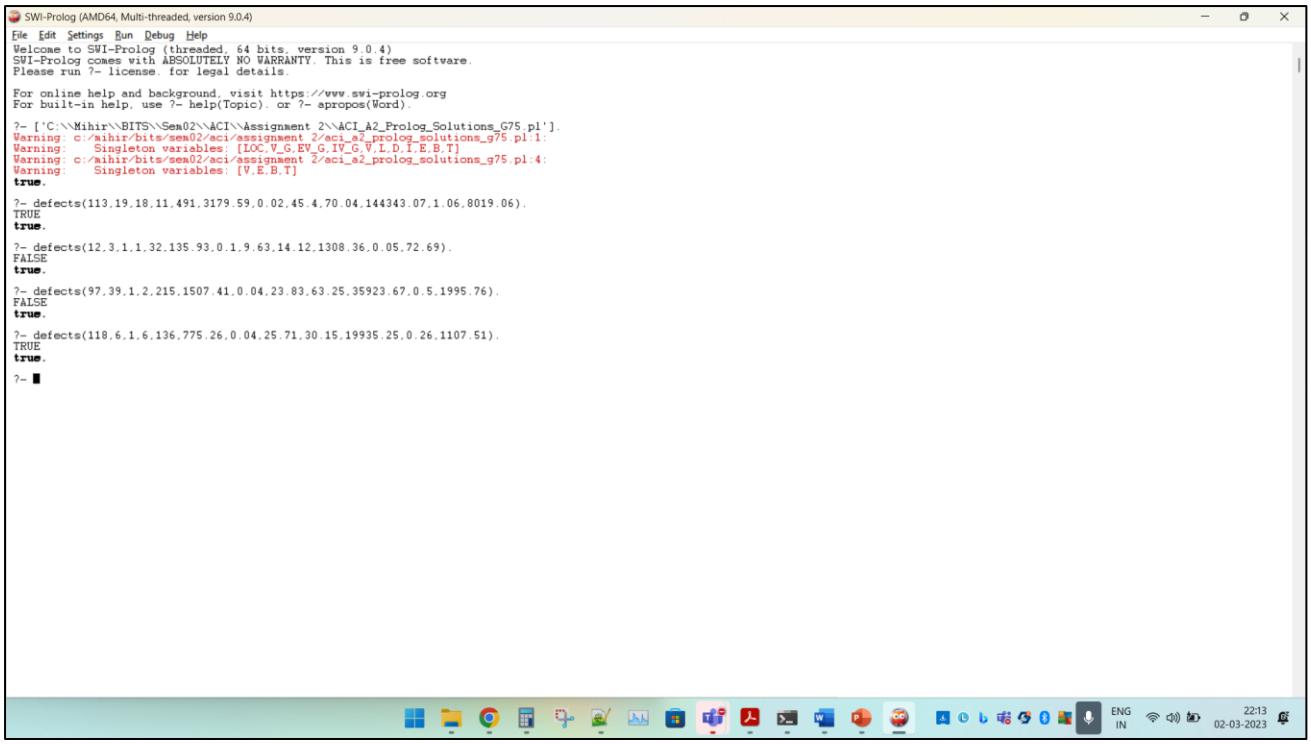


Figure 34: Prolog output examples

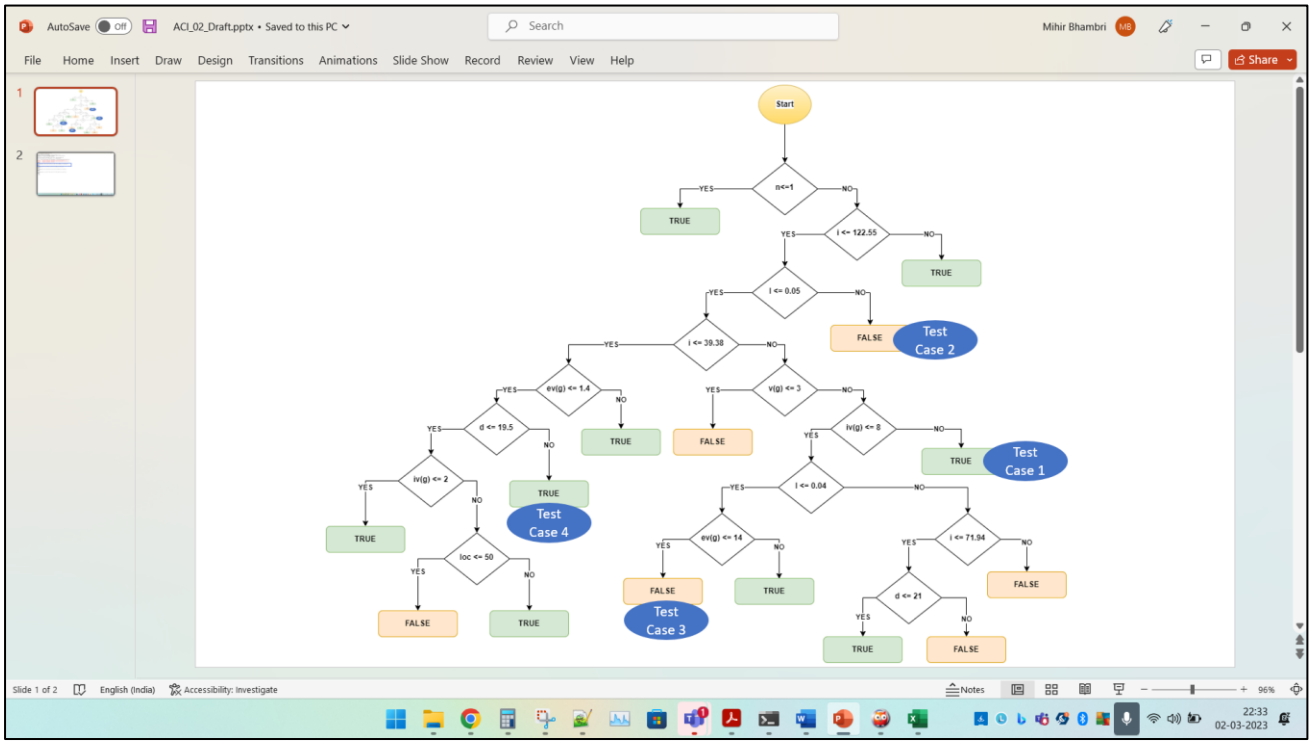
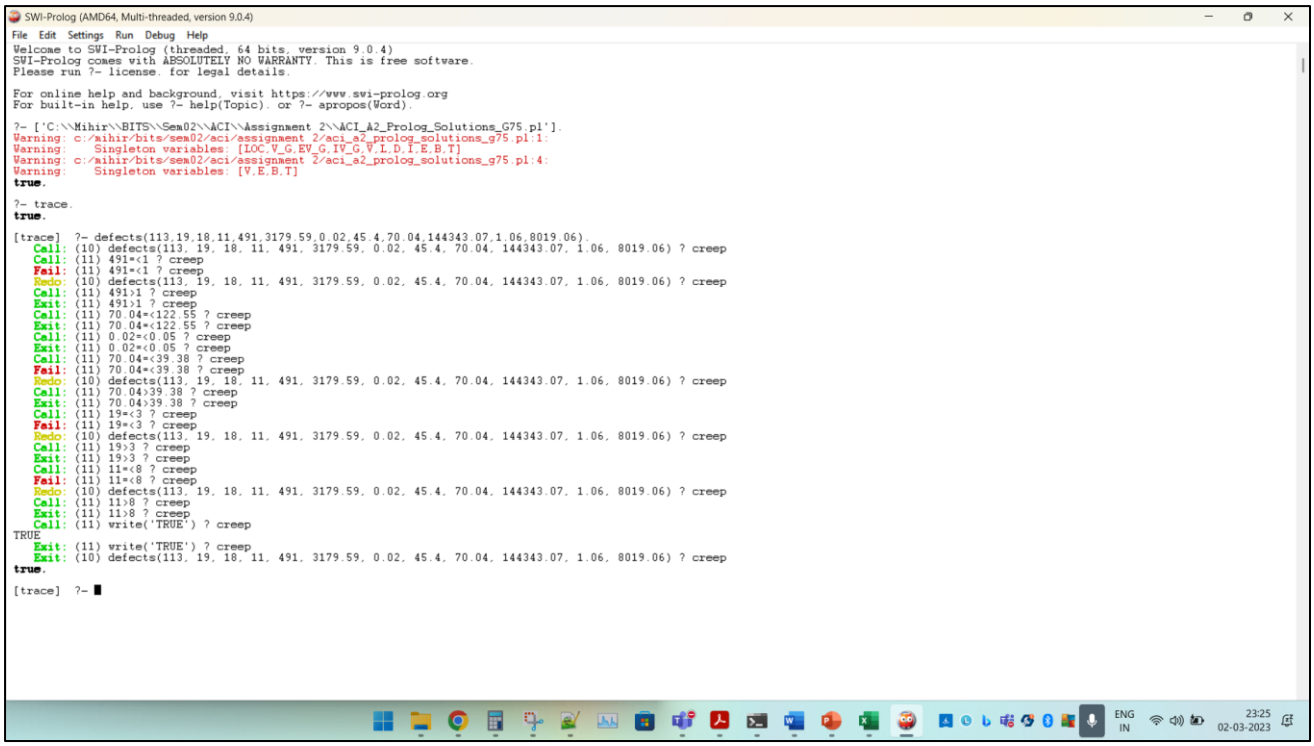


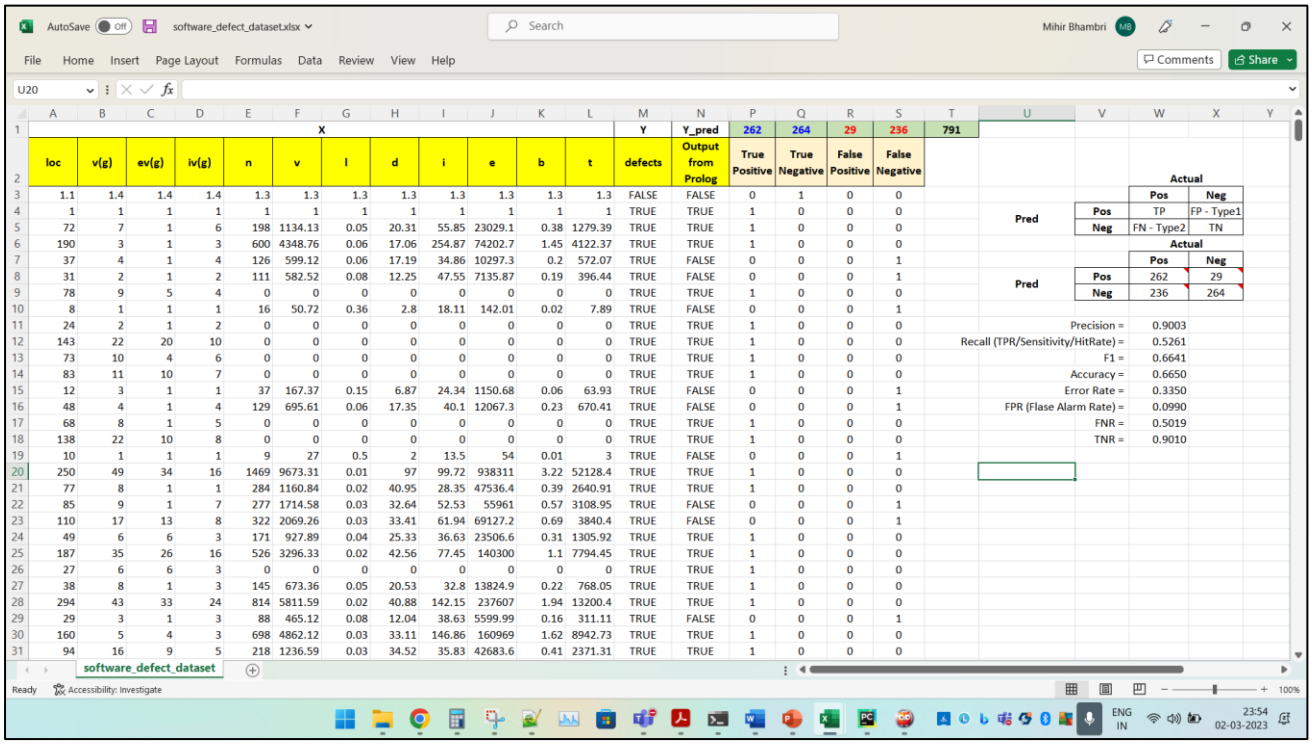
Figure 35: Prolog Output Examples - If-Then Rules Verification

Test Case 1: Output Traced for reference (Step-by-Step execution)



Summary

Classification as per Prolog rules is developed and executed. A confusion matrix is used to define the performance of this classification algorithm to visualize and summarize its performance. Excel is used for this purpose.



	X											Y	Y_pred	262	264	29	236	791
	loc	v(g)	ev(g)	iv(g)	n	v	l	d	i	e	b	t	defects	Output from Prolog	True Positive	True Negative	False Positive	False Negative	
3	1.1	1.4	1.4	1.4	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	FALSE	FALSE	0	1	0	0	
4	1	1	1	1	1	1	1	1	1	1	1	1	TRUE	TRUE	1	0	0	0	
5	72	7	1	6	198	1134.13	0.05	20.31	55.85	23029.1	0.38	1279.39	TRUE	TRUE	1	0	0	0	
6	190	3	1	3	600	4348.76	0.06	17.06	254.87	74202.7	1.45	4122.37	TRUE	TRUE	1	0	0	0	
7	37	4	1	4	126	599.12	0.06	17.19	34.86	10297.3	0.2	572.07	TRUE	FALSE	0	0	0	1	
8	31	2	1	2	111	582.52	0.08	12.25	47.55	7135.87	0.19	396.44	TRUE	FALSE	0	0	0	1	
9	78	9	5	4	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
10	8	1	1	1	16	50.72	0.36	2.8	18.11	142.01	0.02	7.89	TRUE	FALSE	0	0	0	1	
11	24	2	1	2	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
12	143	22	20	10	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
13	73	10	4	6	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
14	83	11	10	7	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
15	12	3	1	1	37	167.37	0.15	6.87	24.34	1150.68	0.06	63.93	TRUE	FALSE	0	0	0	1	
16	48	4	1	4	129	695.61	0.06	17.35	40.1	12067.3	0.23	670.41	TRUE	FALSE	0	0	0	1	
17	68	8	1	5	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
18	138	22	10	8	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
19	10	1	1	1	9	27	0.5	2	13.5	54	0.01	3	TRUE	FALSE	0	0	0	1	
20	250	49	34	16	1469	9673.31	0.01	97	99.72	938311	3.22	52128.4	TRUE	TRUE	1	0	0	0	
21	77	8	1	1	284	1160.84	0.02	40.95	28.35	47536.4	0.39	2640.91	TRUE	TRUE	1	0	0	0	
22	85	9	1	7	277	1714.58	0.03	32.64	52.53	55961	0.57	3108.95	TRUE	FALSE	0	0	0	1	
23	110	17	13	8	322	2069.26	0.03	33.41	61.94	69127.2	0.69	3840.4	TRUE	FALSE	0	0	0	1	
24	49	6	6	3	171	927.89	0.04	25.33	36.63	23506.6	0.31	1305.92	TRUE	TRUE	1	0	0	0	
25	187	35	26	16	526	3296.33	0.02	42.56	77.45	140300	1.1	7794.45	TRUE	TRUE	1	0	0	0	
26	27	6	6	3	0	0	0	0	0	0	0	0	TRUE	TRUE	1	0	0	0	
27	38	8	1	3	145	673.36	0.05	20.53	32.8	13824.9	0.22	768.05	TRUE	TRUE	1	0	0	0	
28	294	43	33	24	814	5811.59	0.02	40.88	142.15	237607	1.94	13200.4	TRUE	TRUE	1	0	0	0	
29	29	3	1	3	88	465.12	0.08	12.04	38.63	5599.99	0.16	311.11	TRUE	FALSE	0	0	0	1	
30	160	5	4	3	698	4862.12	0.03	33.11	146.86	160969	1.62	8942.73	TRUE	TRUE	1	0	0	0	
31	94	16	9	5	218	1236.59	0.03	34.52	35.83	42683.6	0.41	2371.31	TRUE	TRUE	1	0	0	0	

		Actual	
		Pos	Neg
Pred	Pos	TP	FP - Type1
	Neg	FN - Type2	TN

		Actual	
		Pos	Neg
Pred	Pos	262	29
	Neg	236	264

Precision =	0.9003
Recall (TPR/Sensitivity/HitRate) =	0.5261
F1 =	0.6641
Accuracy =	0.6650
Error Rate =	0.3350
FPR (Flase Alarm Rate) =	0.0990
FNR =	0.5019
TNR =	0.9010

Figure 36: Comparison of output generated by Prolog Decision Tree against the given csv output

Output as per given decision tree and prolog rules (Y_pred) is noted in an excel and compared against given dataset (Y) in "*software_defect_dataset.csv*".

True Positive (262), True Negative (264), False Positive (29) and False Negative (236) is estimated from Y and Y_pred and a confusion matrix is plotted in the same excel to analyse the accuracy, precision and F1 score. **Precision of 0.9003** along with **F1 score of 0.6641** is observed.

To develop better classification algorithm, python-based decision tree algorithm can be constructed using ‘Gini’ index / ‘Entropy’ / Max Depth hyper-parameters as shown in Figure 37

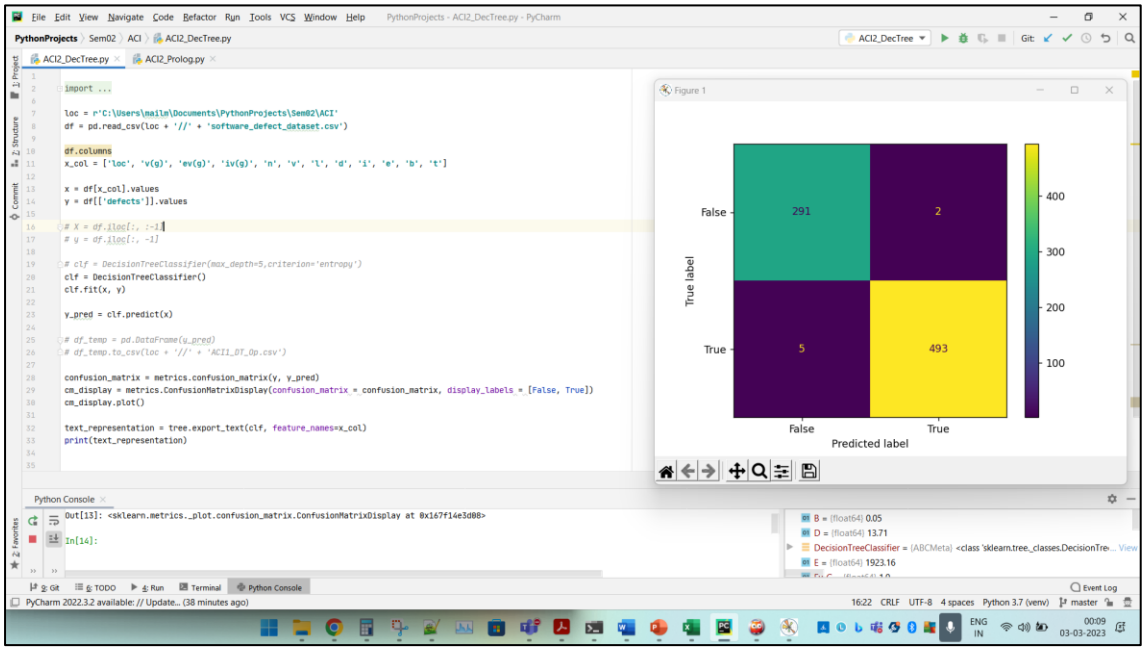


Figure 37: Decision Tree based on Python libraries

Applications of Prolog:

- Natural Language Processing: Prolog can be used to implement natural language processing systems that can analyze and interpret human language.
- Expert Systems: Prolog can be used to build expert systems that can reason about a particular domain of knowledge and provide advice or recommendations.
- Automated Reasoning: Prolog can be used to build automated reasoning systems that can prove or disprove logical statements and solve complex logical problems.
- Knowledge Representation: Prolog can be used to represent knowledge in a formal and logical way, making it easy to reason about and manipulate.

Overall, Prolog provides a powerful and flexible set of tools for solving complex logical problems, making it a valuable tool in many different fields.

Thank you