

**THE MENACE**  
**Reinforcement Learning based Tic-tac-toe Game**

**INFO6205 – Final Project**  
**Prof. Robin Hillyard**

**Shenckerr Gollapudi – 001569753**  
**Ganesh Dharani – 002191922**  
**Nithin Sai Guvvala – 001521107**

# **Index**

|                                      |                |
|--------------------------------------|----------------|
| ➤ <b>Introduction</b>                | <b>- 3</b>     |
| ➤ <b>Aim</b>                         | <b>- 3</b>     |
| ➤ <b>Algorithms and Invariants</b>   | <b>- 4-6</b>   |
| ➤ <b>Training and Implementation</b> | <b>- 7-10</b>  |
| ➤ <b>User Interface</b>              | <b>- 11-13</b> |
| ➤ <b>Logfile Observations</b>        | <b>- 14</b>    |
| ➤ <b>Test Cases</b>                  | <b>- 15</b>    |
| ➤ <b>Observation</b>                 | <b>- 16</b>    |
| ➤ <b>Conclusion</b>                  | <b>- 16</b>    |
| ➤ <b>References</b>                  | <b>- 17</b>    |

## **Introduction:**

The Menace is a reinforcement learning based famous Noughts and crosses' game. The original idea was to simplify the game into 304 matchboxes (with colored beads representing the moves in them) representing different states of the game. After each game played the matchboxes are punished or rewarded with the beads depending on win/loss of that game. At the end of each game, if menace loses, each bead menace used is removed from each box. If menace wins, three beads the same as the color used during each individual turn are added to their respective box. If the game resulted in a draw, one bead is added.

## **Aim:**

Implement a program to implement a menace that gets trained against human strategy. The menace should be able to refine its strategy after each training round. After a number of training rounds, the menace should be able to challenge a player and be able to win over or at least draw the match.

## **Data Structures and classes:**

There are majorly two classes named **Game** and **Model**.

**Game:** This deals with the basics of the Tic-Tac-Toe game like making a move(**Method: move()**), finding the winner of the game(**Method: getWinner()**) and determining when the game is over and to end the moves after winner is declared(**Method: isGameOver()**)

**Model:** This class deals with creating a menace model, updating the strategies after each game, training the menace accordingly. **intializeMatchBox()**, **getNextStates()**, **getRotations**, **train()** are the most important methods in this class which are clearly discussed in the **next pages**.

## Algorithms and Invariants:

As part of a game, menace should be able to know all the possible moves at each state of the game to decide which move to make. So, getNextStates() method (explained in **next pages**) uses **Breadth First Search(BFS)** algorithm to find out all the possible steps in each state of the game. To determine the weights of each possible move, we used **python dictionary(Hash Table)** that stores the next possible **moves as keys** and the **weight** of each move as **values**. More the value of weights, more is the probability for the menace to win the game if that move is made. (as shown in the image below, menace makes the move with highest weight)

```
{3: 1141, 5: 1141, 6: 1092, 8: 1092}
3
x|o|x
----
o|o|
----
|x|

{6: 1000, 8: 1356}
8
x|o|x
----
o|o|x
----
|x|o
```

## Model Implementation:

**Model Design:** Menace model has alpha, beta gamma, delta, matchbox attributes.

**Alpha:** Number of beads in each box at the start of game (1000 in our case).

**Beta:** Number of beads to add to each box (state) in event of win (+2 in our case).

**Gamma:** Number of beads to remove from the boxes in an event of loss (-1 in our case).

**Delta:** Number of beads to add to each box in an event of draw (+1 in our case).

**Matchbox:** A dictionary representing the state of each game with

**Key:** possible moves to be played in each state.

**Value:** Weight of each move (beads in each matchbox)

```
##### Designing a menace Model#####
class Model:
    def __init__(self, alpha, beta, gamma, delta):
        self.game = Game()
        self.matchBox = dict()
        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma
        self.delta = delta
        self.trainWins=0
        self.trainLoses=0
        self.trainDraws=0
        self.initializeMatchBox()

    def initializeMatchBox(self):
        q = [[0, 0, 0, 0, 0, 0, 0, 0, 0], 1]
        while len(q) != 0:
            u, turn = q.pop()
            if tuple(u) not in self.matchBox:
                self.matchBox[tuple(u)] = dict()
            vAll = self.getNextStates(u, turn)
            for v in vAll:
                boardCopy = u[:]
                boardCopy[v] = turn
                q.append([boardCopy, 1 if turn == 10 else 10])
                self.matchBox[tuple(u)][v] = self.alpha
```

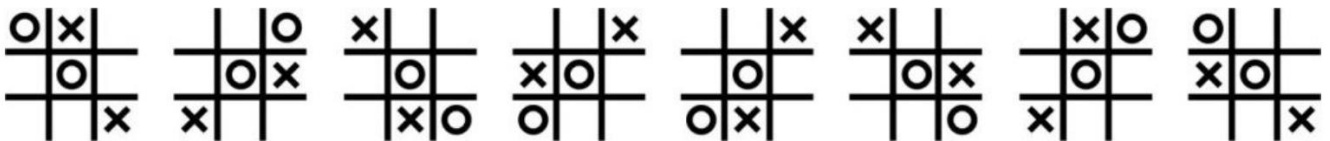
**Initialize Matchbox:** This is implemented using Breadth First Search (BFS). For every state of the game, the menace finds out all the possible next states using BFS and has the values(beads) for each move so that it can choose the best possible move.

**Get Next States:** After every move we are updating the board with values (1 for X, 10 for O and 0 for empty space). So, after every move the menace searches for the 0 valued box and the matchbox assigns values(beads) for that move.

```
def getNextStates(self, board = None, turn = None):
    if board == None:
        board = self.game.board
    if turn == None:
        turn = self.game.turn
    nextStates = []
    for i in range(9):
        if board[i] == 0:
            nextStates.append(i)
    return nextStates
```

## **Handling Symmetries and Rotations:**

For example, all these below states are similar and considered to be the same by the menace. So, these states need to be trained multiple times. If one of these is trained, all the other variants will have the same strategy. getRotations() method handles these.



```

def getRotations(board, move):
    rotations = [[0,1,2,3,4,5,6,7,8],
    [0,3,6,1,4,7,2,5,8],
    [6,3,0,7,4,1,8,5,2],
    [6,7,8,3,4,5,0,1,2],
    [8,7,6,5,4,3,2,1,0],
    [8,5,2,7,4,1,6,3,0],
    [2,5,8,1,4,7,0,3,6],
    [2,1,0,5,4,3,8,7,6]]

    ret = []

    for rotation in rotations:
        x = [0] * 9
        newMove = None
        for r in range(len(rotation)):
            x[r] = board[rotation[r]]
            if rotation[r] == move:
                newMove = r
        ret.append((tuple(x), newMove))

    return ret

```

## Training:

This is the most important part for the menace. `train()` method trains the Menace. `modelMoveHistory` maintains the records of all the modelMoves. Considering the same for all the rotations and symmetries of the state too.

Our implementation of menace trains itself against human strategy. i.e., for every move against the menace, the player move distributes the probability of the best move. For every move, the player picks the best possible move (from the `getNextStates()`) on a probability of 0.9 and the remaining 0.1 is shared among all the other possible moves. Considering humans not always plays the perfect move, menace is trained so.

```

If model.game.isGameOver():
    break

    playerChoices = model.getNextStates()
playerChoiceWeights = [.9 if i == maxPlayerWeight else .1/(len(playerChoices)-1) for i in
playerMove = random.choice(playerChoices)

```

After every move, `modelMoveHistory` is updated.

```

modelMoveHistory.append((tuple(model.game.board), modelMove - 1))

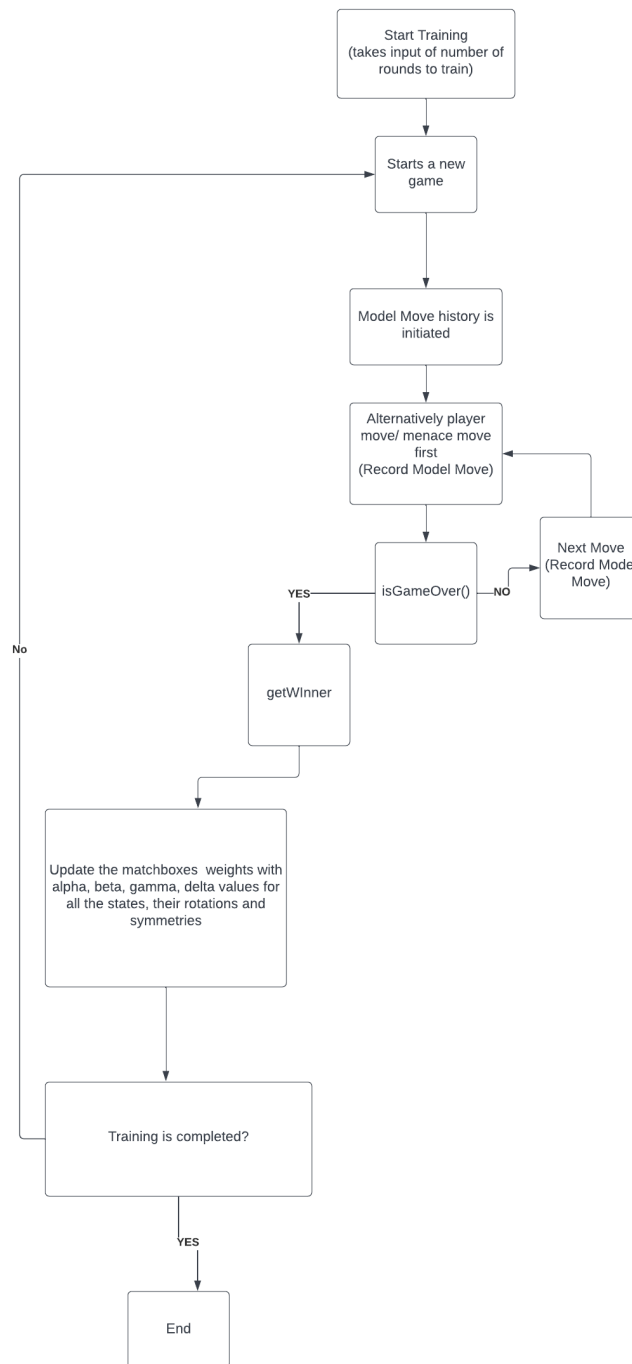
```

## **Training Process:**

1. `train()` method takes `n` as input where `n` is no. of times, we want to train the menace against human strategy.
2. This method alternatively plays the first move. i.e., once menace plays the first move, for the next game human strategy plays the first move.
3. Menace's move : Gets all the possible steps using `getNextStates()`. Chooses randomly between move with highest weight (highest number of beads) and any of the possible moves.
4. Player's move : Gets all the possible steps using `getNextStates()`. Chooses between best possible move(happens 90% of the times) and all other possibilities(remaining 10% shared probability among these).
5. The moves get played until the `isGameOver()` and each move is recorded in `modelMoveHistory`.
6. Once one training round is over, the matchbox is updated with beta, gamma, delta values for the states (and their symmetries and rotations too) occurred from `modelMoveHistory`. This is done after each training run.
7. After training for `n` times, for every state of the game, the menace is now able to understand which is the best move in that state.



# Training Flow Chart:



**Intelligent move:** Most important part of the training. After training is done for n times, for all the states that are touched in the training part, the menace now has the information about best possible move in the form of key-value pairs (no. of beads each color) in the matchbox(state). So, when it is trained and ready, while playing a real game, it chooses the getNextStates() with highest weight (maximum no of beads). We also have implemented a version that allows a player to play against the trained menace. In that version, the menace makes the intelligentMove() against a human player.

```
def intelligentMove(self):
    currentState = tuple(self.game.board)
    nextStates = self.getNextStates()
    nextStateWeights = [self.matchBox[currentState][i] for i in nextStates]
    move = None
    mx = -math.inf
    for i in nextStates:
        if mx < self.matchBox[currentState][i]:
            mx = self.matchBox[currentState][i]
            move = i
    print(move)
    self.move(move + 1)
    return move + 1
```

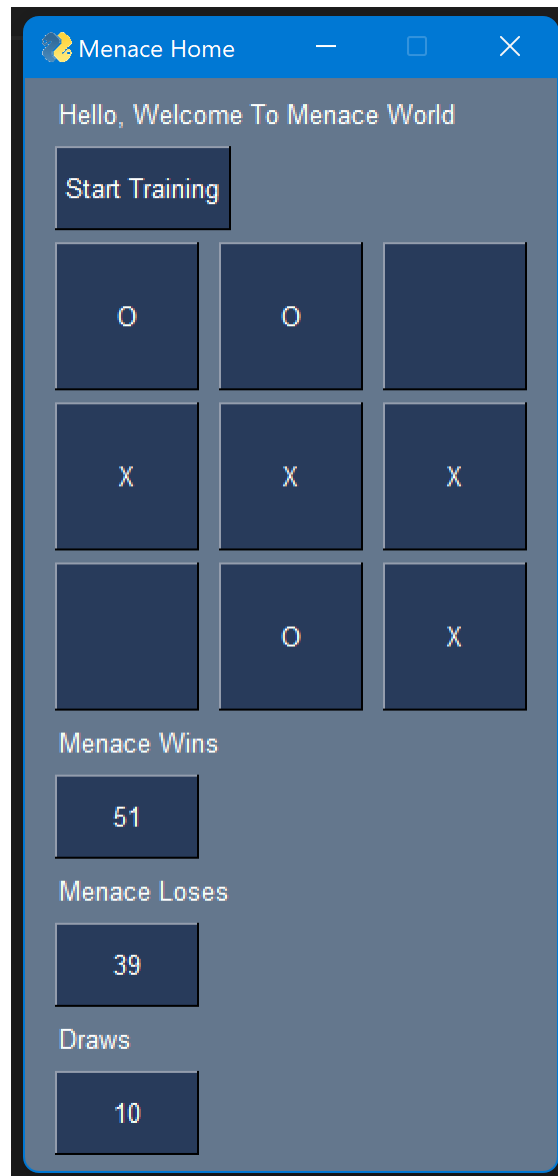
Below is the representation of the matchbox. In each state, for each move, the training method assigned weights(beads) and the intelligent move chooses the highest weighted from the matchbox. The key (below image) is next possible step and values are the weights for each move. Higher the weight, higher is the probability for the menace to win.

```
1
x|o|x
-----
|o|
-----
| |
{3: 1141, 5: 1141, 6: 1092, 8: 1092}
3
x|o|x
-----
o|o|
-----
|x|
{6: 1000, 8: 1356}
8
x|o|x
-----
o|o|x
-----
|x|o
```

## User Interface:

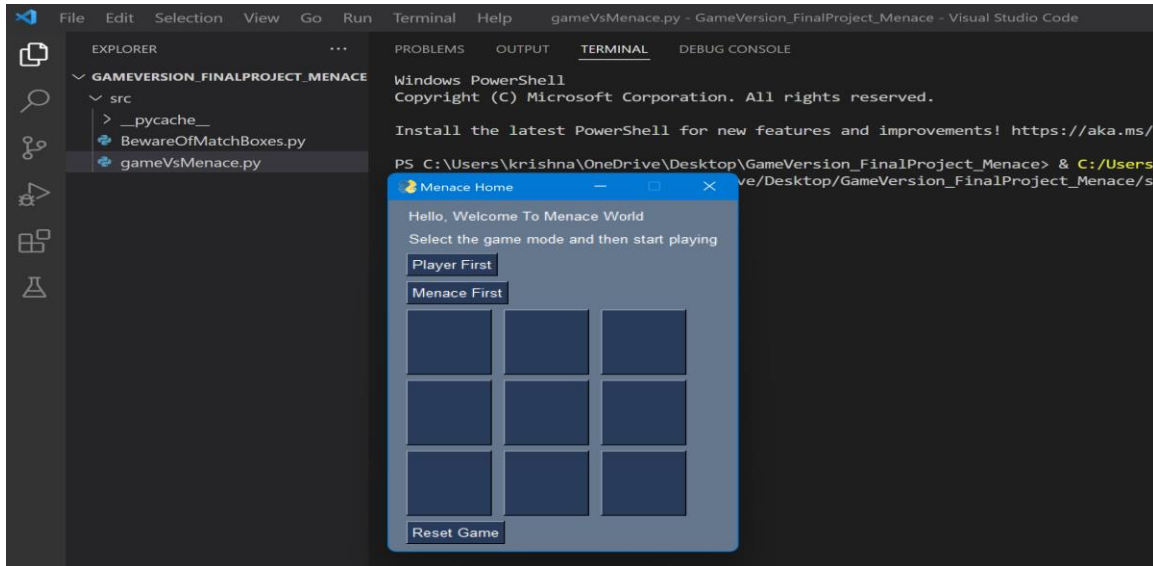
We have implemented the user interfaces for both training part and a Player vs Trained Menace separately using PySimpleGUI.

**Training:** This shows the progress of training process. Once the Start training button is clicked, the process starts and the UI dynamically gets updated with the moves, numbers of wins, losses and draws that menace gone through.

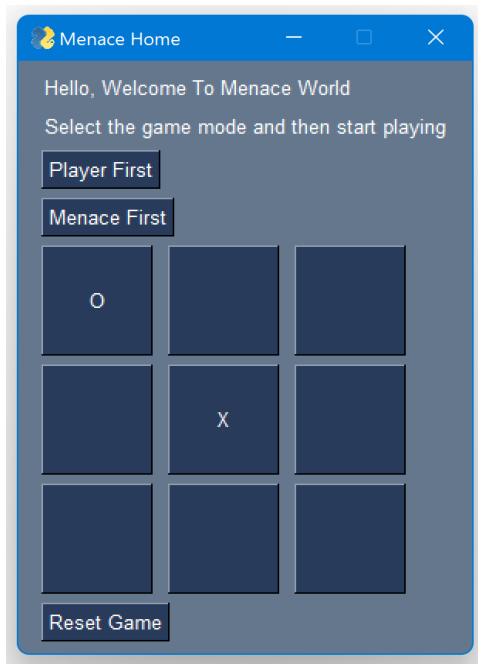


## Game (Player vs Trained Menace):

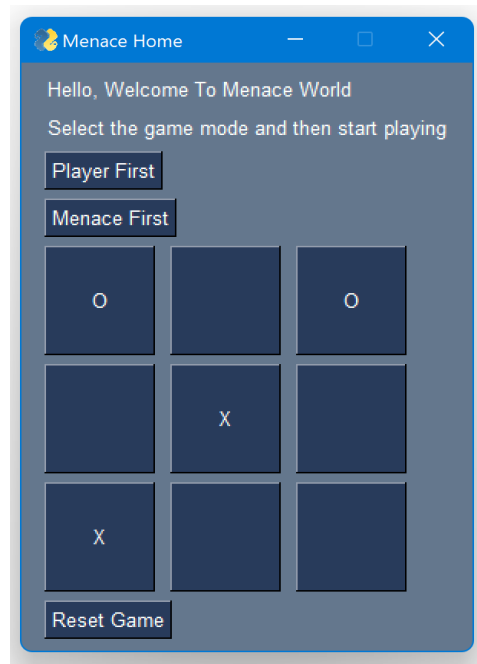
When the program is run, firstly the **menace gets trained with the above shown training method. You can challenge the menace and verify if the training algorithm is working.** A popup pops-up and then the menace is ready to challenge you. You can select the game mode. Player First/Menace first, once the game is over another popup occurs showing who won the Game.



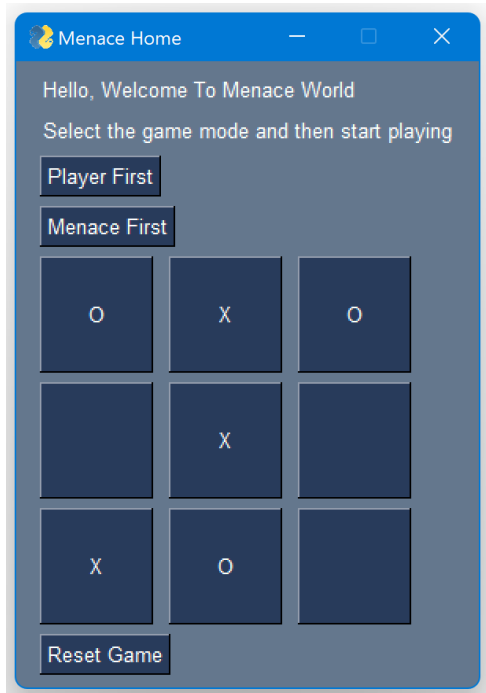
### Player Vs Menace: Player First (X), Menace (O)



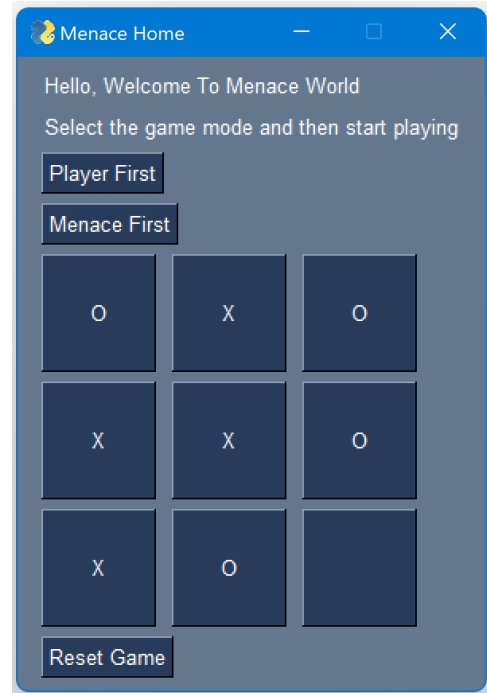
1<sup>st</sup> Move: X Player, O Menace



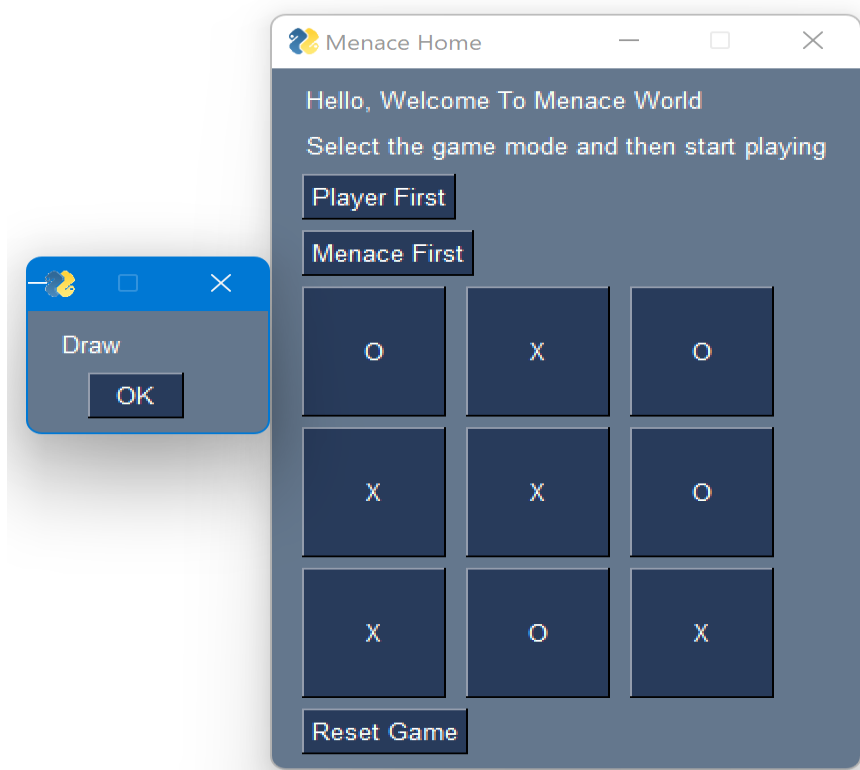
2<sup>nd</sup> Move: X Player, O Menace



3<sup>rd</sup> Move: X Player, O Menace



4<sup>th</sup> Move: X Player, O Menace



Final Move and Game got DRAW

## LogFile and Observations:

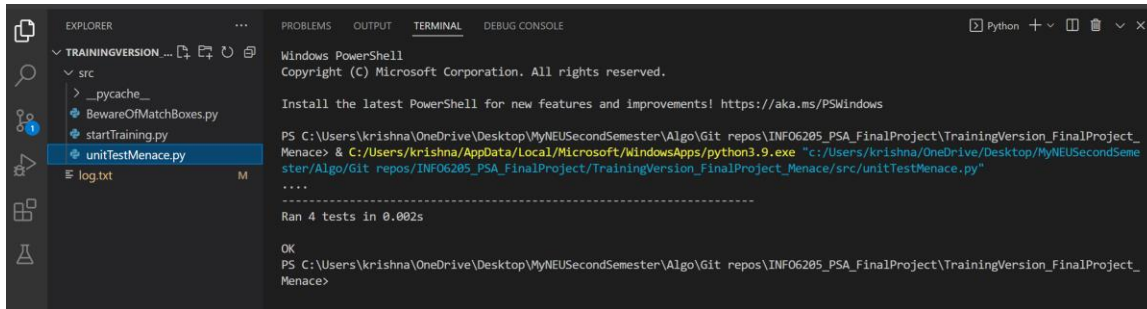
```

v TRAININGVERSION_FINALPROJECT_M...
  v src
    > __pycache__
    BewareOfMatchBoxes.py M
    startTraining.py
    unitTestMenace.py
    log.txt M

85 7 18:30:13,844 Training round 84 : X is the winner - Menace Wins : 33 Menace Loses : 41 Draws :10
86 7 18:30:14,772 Training round 85 : X is the winner - Menace Wins : 33 Menace Loses : 42 Draws :10
87 7 18:30:15,810 Training round 86 : X is the winner - Menace Wins : 34 Menace Loses : 42 Draws :10
88 7 18:30:16,739 Training round 87 : X is the winner - Menace Wins : 34 Menace Loses : 43 Draws :10
89 7 18:30:17,767 Training round 88 : X is the winner - Menace Wins : 35 Menace Loses : 43 Draws :10
90 7 18:30:18,587 Training round 89 : O is the winner - Menace Wins : 35 Menace Loses : 44 Draws :10
91 7 18:30:19,609 Training round 90 : X is the winner - Menace Wins : 35 Menace Loses : 45 Draws :10
92 7 18:30:20,646 Training round 91 : O is the winner - Menace Wins : 35 Menace Loses : 46 Draws :10
93 7 18:30:21,905 Training round 92 : X is the winner - Menace Wins : 35 Menace Loses : 47 Draws :10
94 7 18:30:22,832 Training round 93 : X is the winner - Menace Wins : 35 Menace Loses : 48 Draws :10
95 7 18:30:23,965 Training round 94 : O is the winner - Menace Wins : 36 Menace Loses : 48 Draws :10
96 7 18:30:24,781 Training round 95 : O is the winner - Menace Wins : 37 Menace Loses : 48 Draws :10
97 7 18:30:25,822 Training round 96 : X is the winner - Menace Wins : 37 Menace Loses : 49 Draws :10
98 7 18:30:26,862 Training round 97 : O is the winner - Menace Wins : 37 Menace Loses : 50 Draws :10
99 7 18:30:27,012 Last Training Game moves 4
100 7 18:30:27,223 MatchBox: {0: 1000, 2: 1000, 4: 1000, 5: 1000, 6: 1000, 7: 1000, 8: 1000}
101 7 18:30:27,223 Last Training Game moves 2
102 7 18:30:27,332 Last Training Game moves 7
103 7 18:30:27,443 MatchBox: {0: 1000, 2: 1000, 5: 1000, 7: 1000, 8: 1000}
104 7 18:30:27,443 Last Training Game moves 5
105 7 18:30:27,553 Last Training Game moves 1
106 7 18:30:27,664 Training round 98 : X is the winner - Menace Wins : 37 Menace Loses : 51 Draws :10
107 7 18:30:27,802 MatchBox: {0: 1000, 1: 1000, 2: 1000, 3: 1000, 4: 1000, 5: 1000, 6: 1000, 7: 1000}
108 7 18:30:27,802 Last Training Game moves 9
109 7 18:30:27,916 Last Training Game moves 1
110 7 18:30:28,027 MatchBox: {1: 1000, 3: 1000, 4: 1000, 5: 1000, 6: 1000, 7: 1000}
111 7 18:30:28,027 Last Training Game moves 3
112 7 18:30:28,136 Last Training Game moves 8
113 7 18:30:28,246 MatchBox: {1: 1000, 3: 1000, 4: 1000, 6: 1000}
114 7 18:30:28,246 Last Training Game moves 6
115 7 18:30:28,356 Training round 99 : X is the winner - Menace Wins : 37 Menace Loses : 52 Draws :10
116
```

# Test Cases:

The unitTestMenace.py has different test cases.

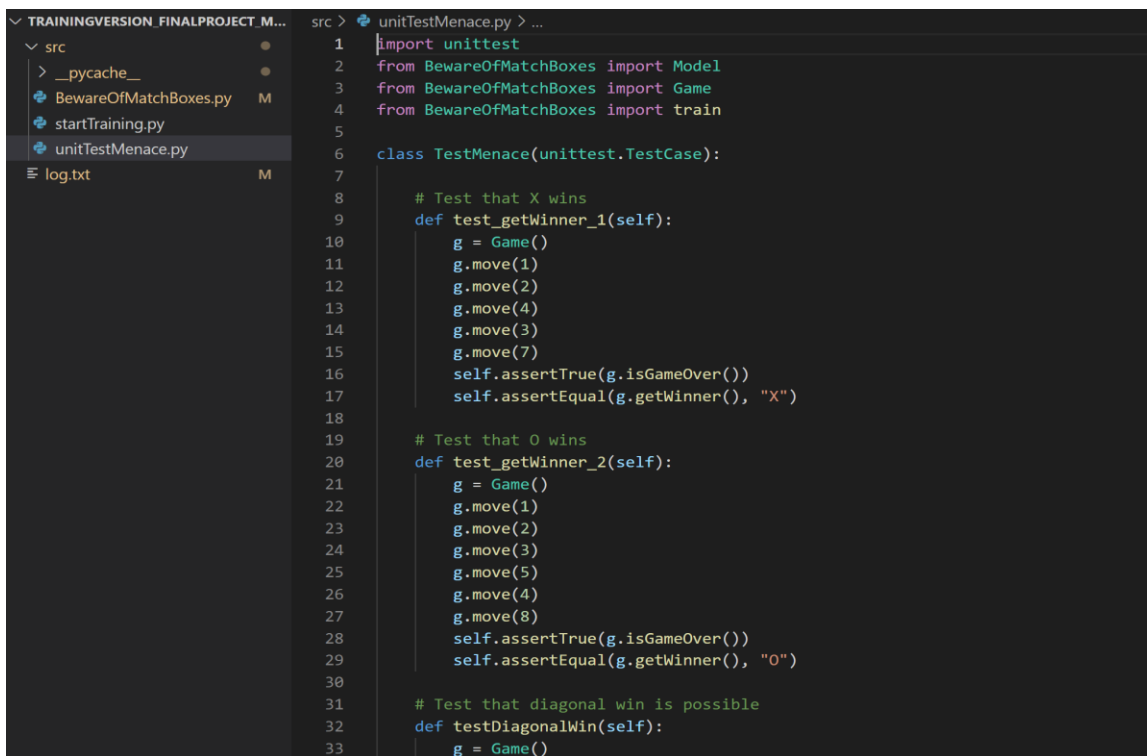


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\krishna\OneDrive\Desktop\MyNEUSecndSemester\Algo\Git repos\INFO6205_PSA_FinalProject\TrainingVersion_FinalProject_Menace> & c:/Users/krishna/AppData/Local/Microsoft/WindowsApps/python3.9.exe "c:/Users/krishna/OneDrive/Desktop/MyNEUSecndSemester/Algo/Git repos/INFO6205_PSA_FinalProject/TrainingVersion_FinalProject_Menace/src/unitTestMenace.py"
.....
Ran 4 tests in 0.0025s

OK
PS C:\Users\krishna\OneDrive\Desktop\MyNEUSecndSemester\Algo\Git repos\INFO6205_PSA_FinalProject\TrainingVersion_FinalProject_Menace>
```



```
src > unitTestMenace.py > ...
1 import unittest
2 from BewareOfMatchBoxes import Model
3 from BewareOfMatchBoxes import Game
4 from BewareOfMatchBoxes import train
5
6 class TestMenace(unittest.TestCase):
7
8     # Test that X wins
9     def test_getWinner_1(self):
10         g = Game()
11         g.move(1)
12         g.move(2)
13         g.move(4)
14         g.move(3)
15         g.move(7)
16         self.assertTrue(g.isGameOver())
17         self.assertEqual(g.getWinner(), "X")
18
19     # Test that O wins
20     def test_getWinner_2(self):
21         g = Game()
22         g.move(1)
23         g.move(2)
24         g.move(3)
25         g.move(5)
26         g.move(4)
27         g.move(8)
28         self.assertTrue(g.isGameOver())
29         self.assertEqual(g.getWinner(), "O")
30
31     # Test that diagonal win is possible
32     def testDiagonalWin(self):
33         g = Game()
```

1. **test\_getWinner\_1** to test if the program works when 'X' wins
2. **test\_getWinner\_2** to test if the program works when 'O' wins
3. **testDiagonalWin** to test if the program is recognizing when there is a win diagonally
4. **testDraw** to test if the program is recognizing when a game ends in a tie
5. **modelTrainCount** to find out if the total n is equal to sum of Losses, Wins and Draws
6. **highlyTrainedModelWinsOften** : Considering 2 different menace models, one trained higher than the other. When games are played, the model with highest training should have a higher Win/Loss ratio

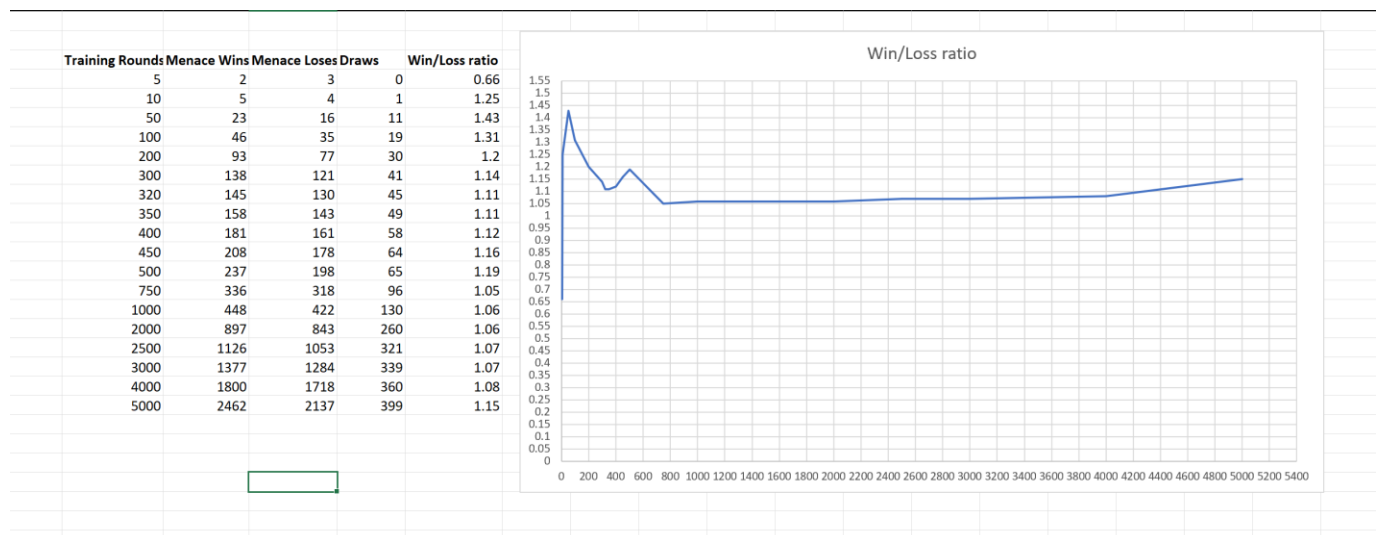
## Observations And Analysis:

After proper training, when menace plays first the game is tending to start at the middle of the board.

The win-to-loss ratio of the menace looks to be increasing after number of trainings runs even though it is a random in the beginning when it's not trained with all the possible states.

When trained heavily, the menace becomes unbeatable and if the player also plays perfectly all the games might end up draw, but menace would never lose.

## Win/Loss ratio vs Training Runs:



## Conclusion:

Reinforcement learning based algorithms are really very helpful and efficient in refining strategies and decision-making process. When after every iteration if an algorithm is getting better and more efficient in providing required solutions, it makes more smart and intelligent applications. In this case, the Menace is learning after each training run, refining its strategy and after sufficient runs it can choose the best strategy considering the next states.



## **References:**

1. [Python Developer's Guide](#)
2. [GUI Programming in Python - Python Geeks](#)
3. [Menace: the Machine Educable Noughts And Crosses Engine - Chalkdust \(chalkdustmagazine.com\)](#)
4. [AI exercise: Implementing MENACE - Department of Information Technology - Uppsala University \(uu.se\)](#)
5. [MENACE: the pile of matchboxes which can learn - YouTube](#)