The Menace

Program Structures & Algorithms

Section 8

Spring 2022

INFO 6205 Final Project

Team Members:

Raj Mehta (NUID): 001094914

Kanishk Bhatia (NUID): 001580259

Instructor:

Prog. Robin Hillyard

Date: 27th April 2022

Introduction:

The MENACE (Matchbox Educable Noughts And Crosses Engine) is a mechanical computer designed from 304 matchboxes, built by artificial intelligence researcher, Donald Michie in 1961. It was designed to play human opponents in a game of tic-tac-toe, by returning a move for any given state of play, and to improve its strategy through reinforcement learning.

The game board is a 3 x 3 grid, each box on the grid has a different color. The matchbox represents a single possible layout. Each matchbox consists of multiple-colored beads, and each color bead corresponds to a move the menace can make. In the initial phase of training, there are a random collection of beads in each matchbox, which makes every possible move equally likely, which makes the menace lose a lot of games against a human. Applying Reinforcement Learning, it keeps a track of each game, and if the menace wins a game, we give it more beads for the same color in the same boxes. If it loses, we take a bead away from that box. This makes it less likely to do that move again. Training it multiple times improves the menace after every game.

Aim:

1. To implement "The Menace" by replacing the matchboxes with values in a dictionary. Train the Menace by running games played against the "human" strategy, which is based upon the optical strategy. Choose the values for:
   - Alpha (the number of "beads" in each "matchbox" at the start of the game)
   - Beta (the number of "beads" to add to the "matchbox" in the event of a win)
   - Gamma (the number of "beads" to take out of the "matchbox" in the event of a loss)
   - Delta (the number of "beads" to add to the "matchbox" in the event of a draw)
2. Choose an optimal strategy with probability *p.*
3. Implement logging with date/time, win/loss/draw, and p.
4. Run Unit tests showing the date/time.

Approach:

- The approach was to first figure out the data structures which are efficient enough to use for training.
- The next step was to make the Game Board class for easy display of the game board and play games with it.
- We then needed to code the Tic Tac Toe Human Optimal Strategy. For that, we needed to log all the possible game conditions for coding the accurate conditions. For example: logging all the possible fork conditions.
- We then needed to code the brains of the Menace. We needed to code the win, lose, and draw conditions and the actions, i.e., insertion or deletion of available beads to choose from for a particular game state.
- We now needed a main class to orchestrate it all. We made two functions, one for training the Menace against the Human Optimal Strategy and the second function for the user to play against the Menace according to the learnings by Menace in the training state.
- We also needed a means of storing the learnings by the Menace. We opted to store the game states (as the key) and the beads available (as the value) in a JSON file for easy storage during training and retrieval during gameplay.
- We chose to store the training and gameplay logs as text files for easy readability.
- During the training phase, we realized that Python uses a single thread for processing. We inserted the for loop which calls the training function inside a multi-threading loop for faster processing. With this, we could bring down the training time by multiple times.

Program:

The program is coded in Python 3

Data Structures and Classes:

The data structures and classes are described as follows:

- **main.py:** The functions in main.py are:
    1. **logging:** Writes the training log in a text file with the date and time attached to it.
    2. **training:** Implements human optimal strategy.
    3. **gameplay:** Takes human input for the gameplay, which is to be used after the training function.
- **humany.py:** The functions in human.py are:
    1. **win:** If the player has two in a row, they can place a third to get three in a row.
    2. **block:** If the opponent has two in a row, the player must play the third themselves to block the opponent.
    3. **center:** A player marks the center.
    4. **opposite corner:** If the opponent is in the corner, the player plays in the opposite corner.
    5. **empty center:** The player plays in a corner square.
    6. **empty side:** The player plays in a middle square on any of the four sides.
    7. **fork:** Cause a scenario where the player has two ways to win.
    8. **block fork:** If there is only one possible fork for the opponent, the player should block it.
    9. **human move:** Checks if the move made by a human is valid or not.

- **menace.py:**
    1. **game states:** A dictionary of the game states
    2. **moves played:** An array of the moves played by the menace, which is used for training
    3. **moves to make:** Function to decide which move the menace will make
    4. **win result:** Adding beads to the elements in moves played array on winning a game.
    5. **draw result:** Adding beads to the elements in moves played array on drawing a game.
    6. **lose result:** Confiscating beads from the elements in moves played array on losing a game.
- **game board:**
    1. **game board:** An array to declare an empty game board.
    2. **display board:** Function to display the board on the terminal.
    3. **win condition:** Function to check if the current board state corresponds to a win.
    4. **draw condition:** Function to check if the current board state corresponds to a draw.
    5. **lose condition:** Function to check if the current board state corresponds to a loss.
    6. **make move on board:** Function to make the selected move on the board
    7. **is move valid:** Function to check if the selected is valid.
    8. **board string:** Convert the game board into a string format.

Algorithm:

Reinforcement Learning:

Step 1: Create an empty dictionary to store the game states and an empty array to store the moves made by the menace.

Step 2: Train the menace against the human optimal strategy. Reward the menace by adding beads if the menace wins, add some beads if the menace draws and remove beads if the menace loses.

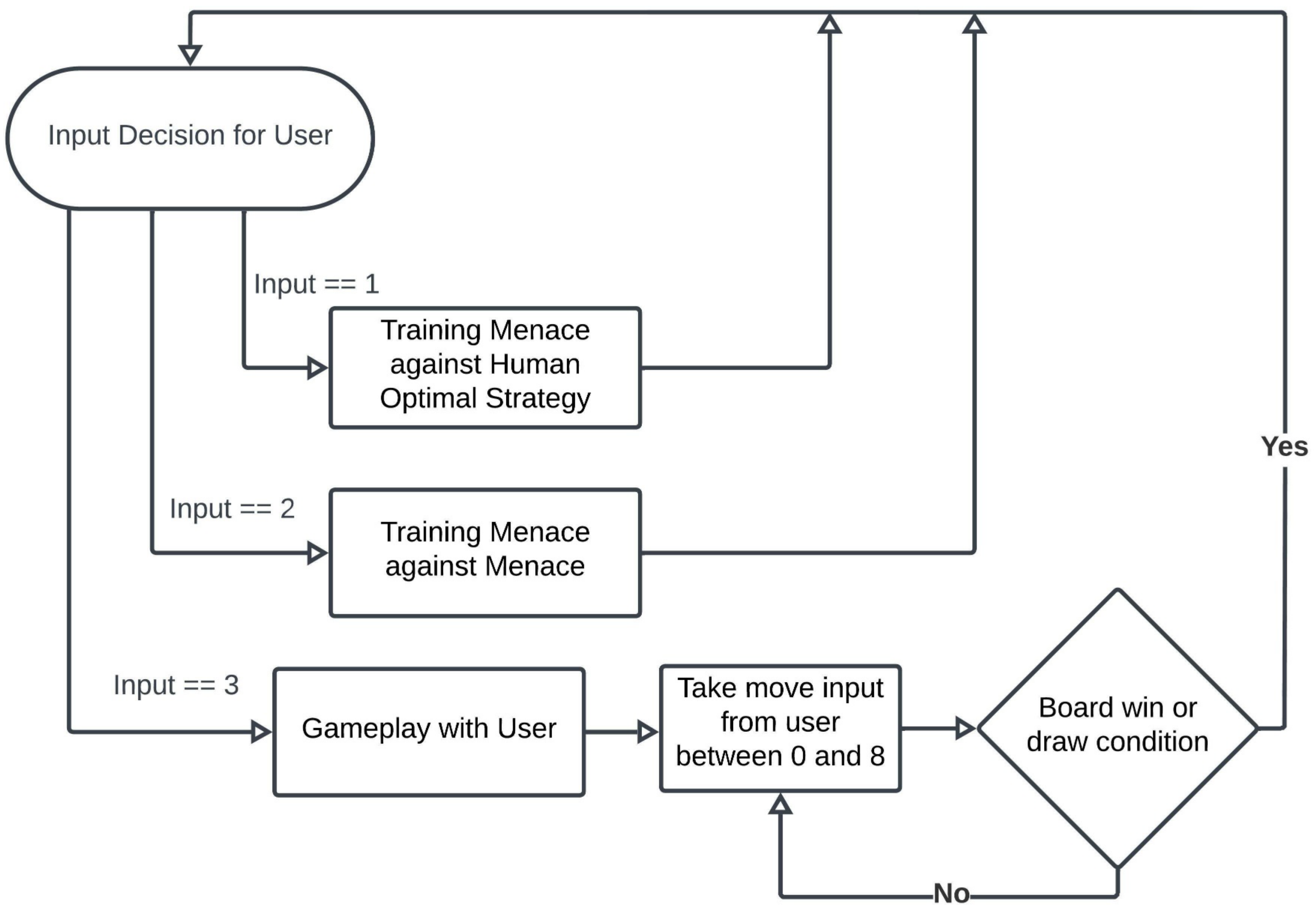
Step 3: Repeat Step 2 by training the menace against the menace itself and rewarding it in the same way.

Step 4: User plays against the menace

Invariants:

- A board will always have 9 cells. Hence, a move that is made will always be between 0 and 8.
- A move cannot be made on a non-empty cell.
- Three same symbols in a row will always lead to a win condition.
- A completely occupied board with no symbol three in a row will lead to a draw condition

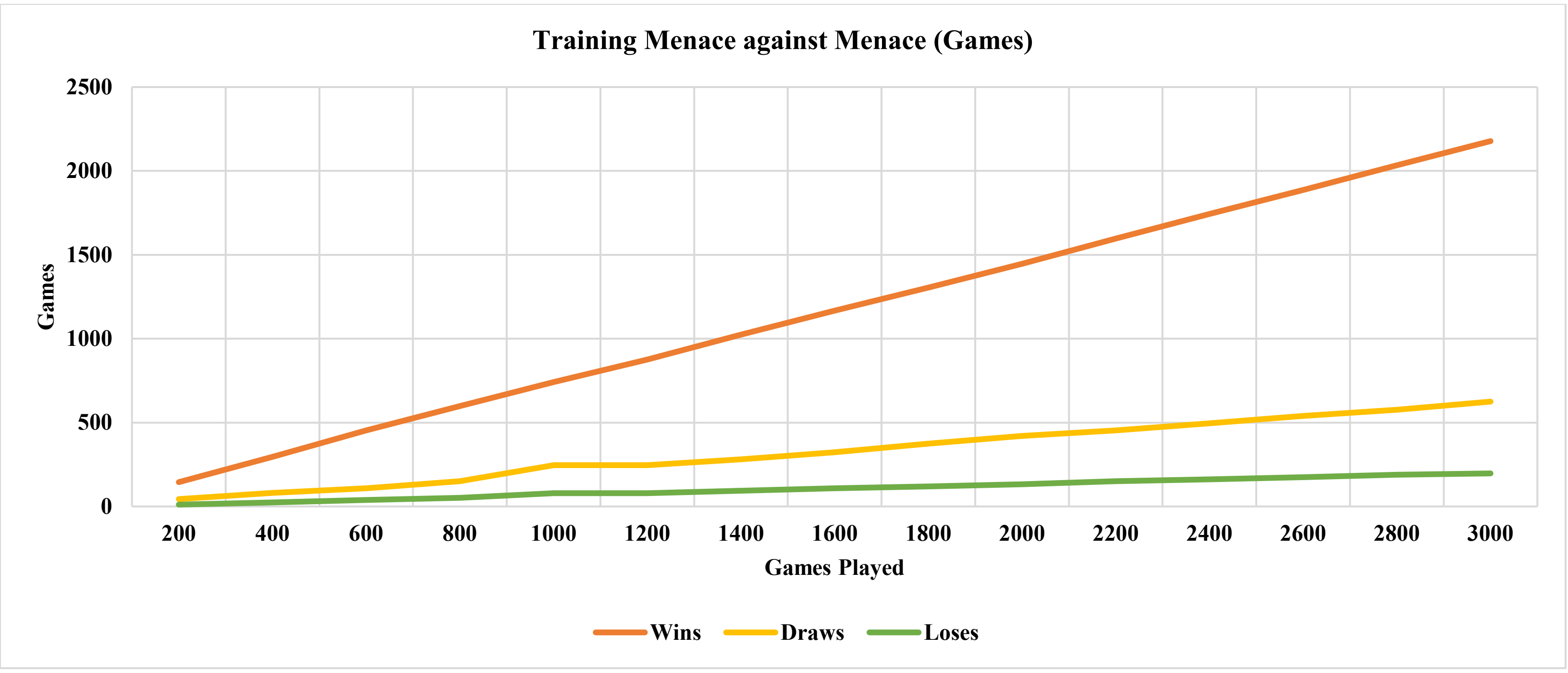
Flow Chart:

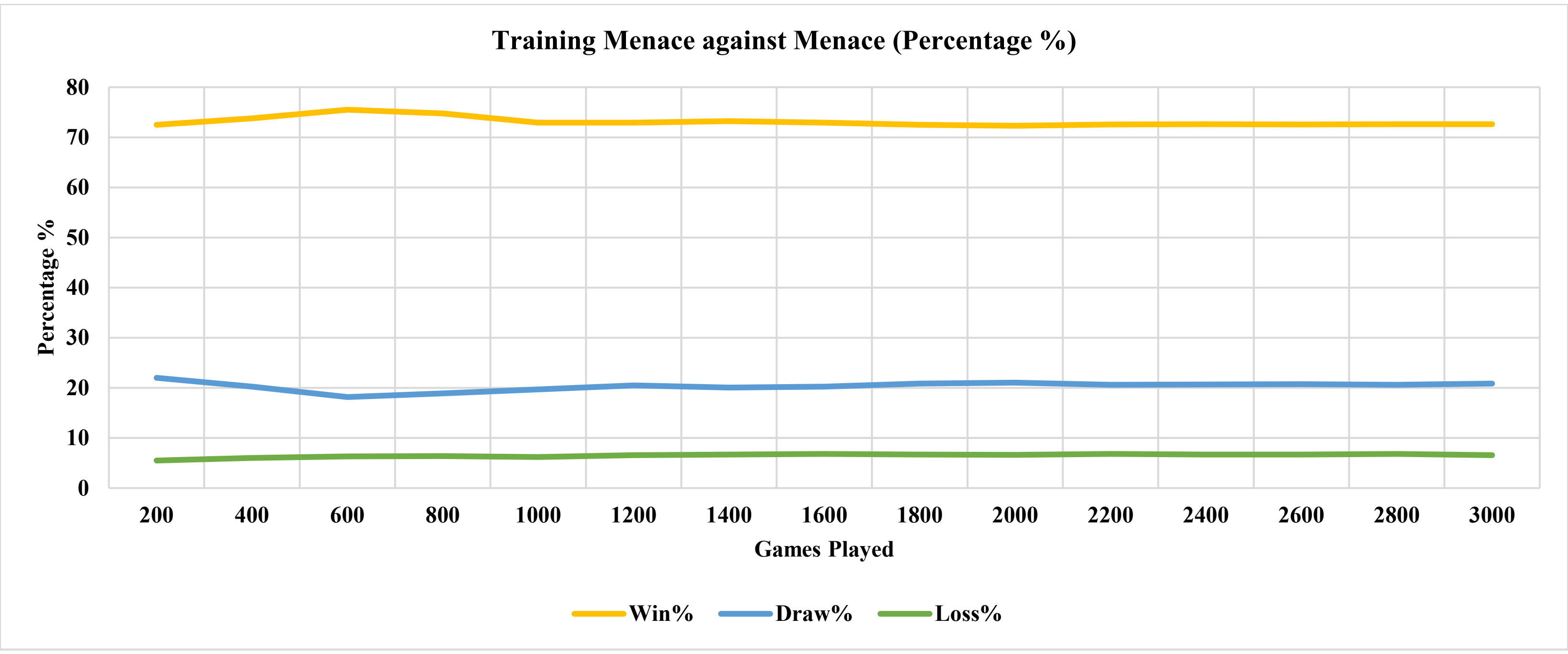

Input Decision for User
Input == 1
Training Menace against Human Optimal Strategy
Input == 2
Training Menace against Menace
Input == 3
Gameplay with User
Take move input from user between 0 and 8
Board win or draw condition
Yes
No

Observation:

There are 2 ways to train the menace:

1. Training against the menace: In this case, the menace is being trained against another menace, using random combinations to place the beads on the game board.

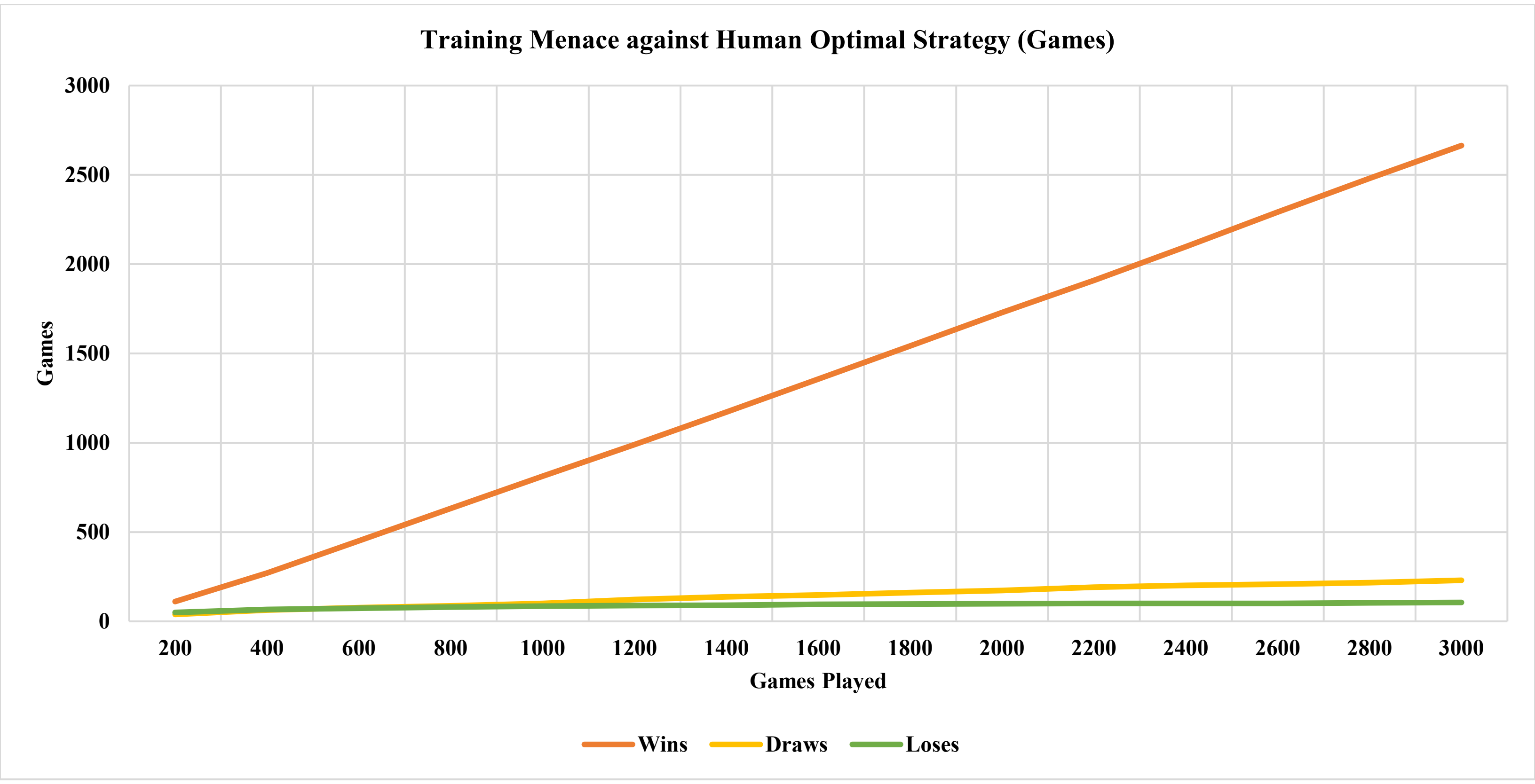


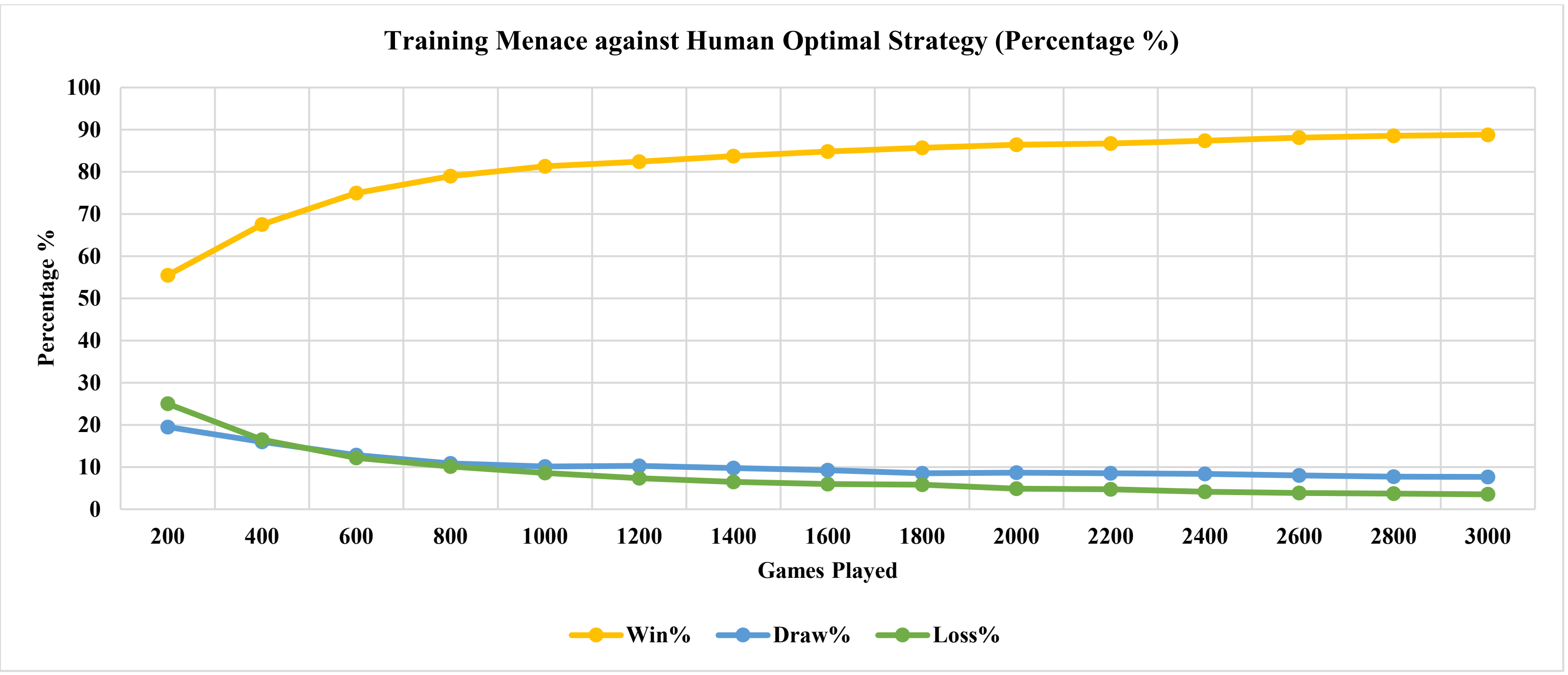


As observed from the graphs above, training the menace against another menace gives a linear increase in the number of games won as the number of games is increased. Similarly, the number of draws also increases but with a smaller gradient. However, the number of games lost is also increasing by a small amount, which is not ideal for the game.

The win, draw and loss percentage against the number of games played is almost a constant graph i.e., the lines are parallel to the x-axis. This states that the menace is not increasing its chances to win or draw even when the training is completed using 3000 games.

2. Training against human optimal strategy: In this case, the menace is being trained against the human optimal strategy, in the order win, block, fork, block opponent's fork, center, opposite corner, empty corner, and empty side.





As observed from the graphs above, while training the menace against the human optimal strategy also gives a linear increase in the number of games won as the number of games is increased. But for the number of draws and losses, the graph is almost constant. There is only a minor increase in the gradient, which is ideal for the game.

The win percentage steadily increases as the number of games is increased, and the draw and loss percentage slightly decreases as the number of games is increased. This states that as the number of games is increased, the win percentage will increase the draw and loss percentage.

Results:

We take 2 sets of variables for Alpha, Beta, Gamma, and Delta

1. Set 1: Alpha = 4, Beta = 2, Gamma = 1, Delta = 1:

   For Human Optimal Strategy:

   - Number of games: 3000
   - Number of wins: 2403
   - Number of draws: 368

- Number of losses: 229
- Win Percentage: 80.1
- Lost Percentage: 7.633333333333334
- Draw Percentage: 12.266666666666667

For Menace:

- Number of games: 3000
- Number of wins: 2178
- Number of draws: 628
- Number of losses: 194
- Win Percentage: 72.6
- Lost Percentage: 6.466666666666667
- Draw Percentage: 20.933333333333334

2. Set 2: Alpha = 5, Beta = 3, Gamma = 2, Delta = 1

   For Human Optimal Strategy:

   - Number of games: 3000
   - Number of wins: 2578
   - Number of draws: 235
   - Number of losses: 187
   - Win Percentage: 85.93333333333334
   - Lost Percentage: 6.233333333333333
   - Draw Percentage: 7.833333333333333

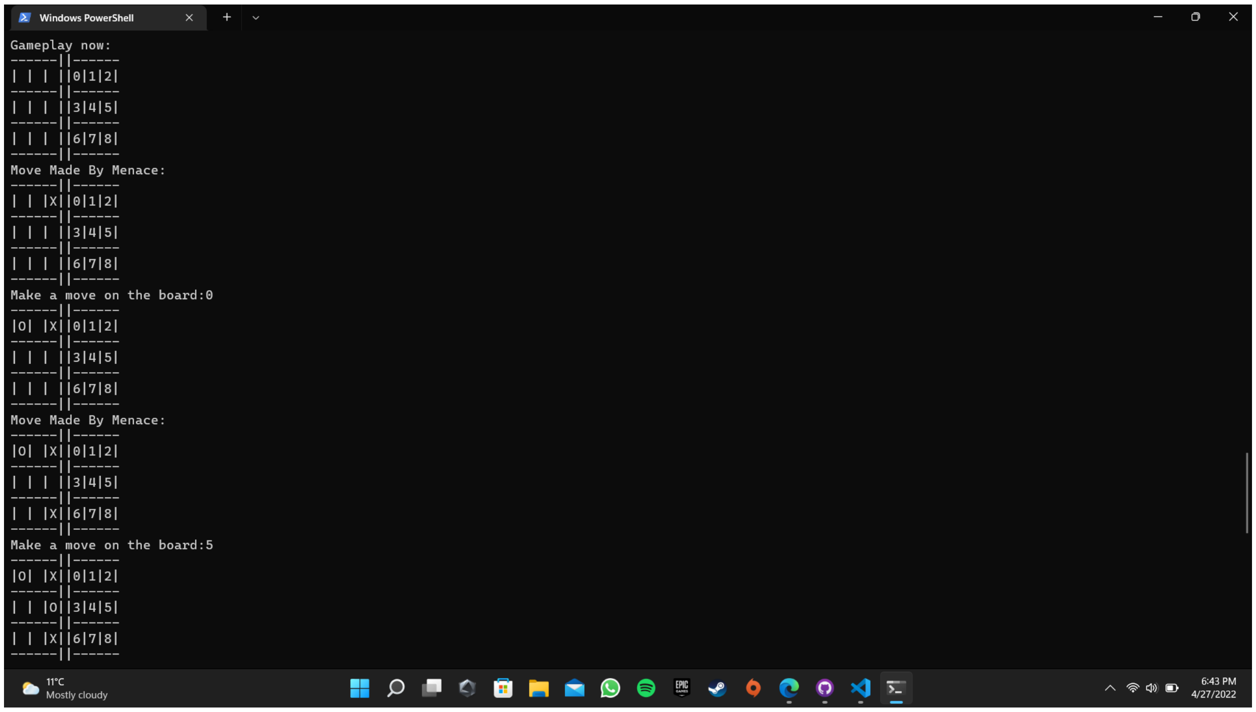
   For Menace:

   - Number of games: 3000
   - Number of wins: 2162
   - Number of draws: 647
   - Number of losses: 191
   - Win Percentage: 72.06666666666666

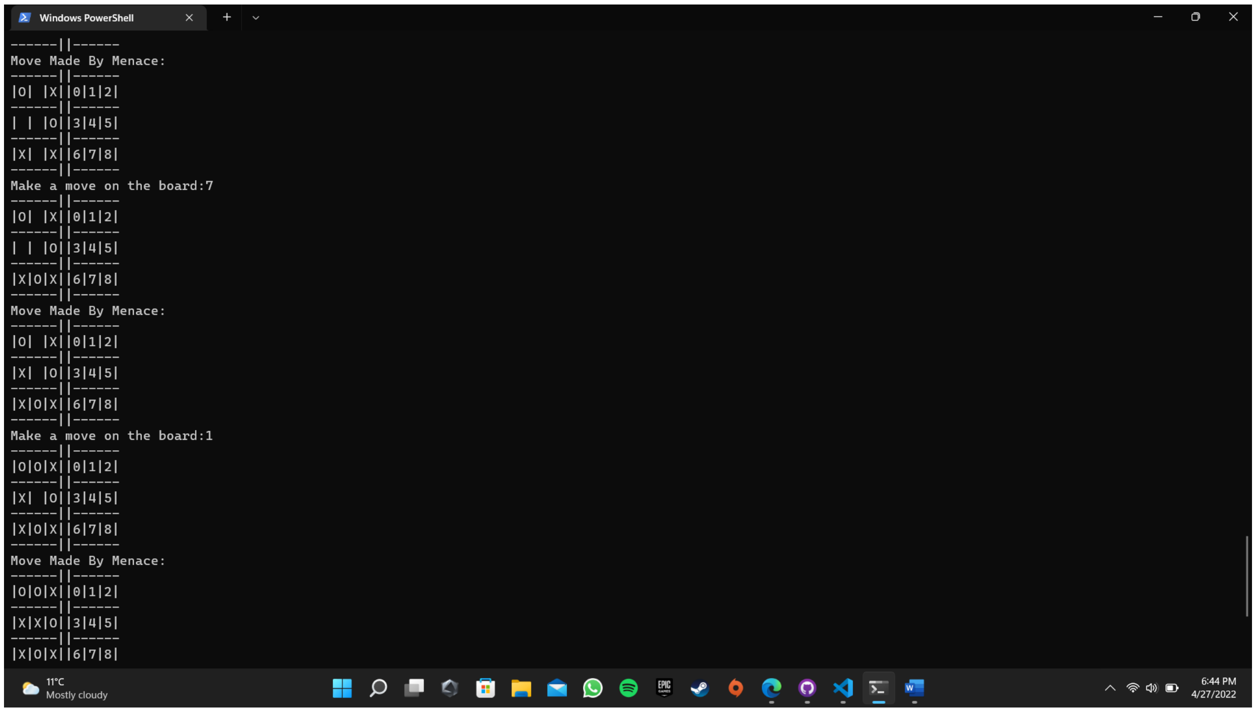
- Lost Percentage: 6.366666666666666
- Draw Percentage: 21.566666666666666

Comparing set 1 and set, we can conclude that set 2, when the menace is trained against the human optimal strategy, gives a better win percentage than set 1. But when the menace is trained against the menace itself, the win percentage is similar for both set 1 and set 2.

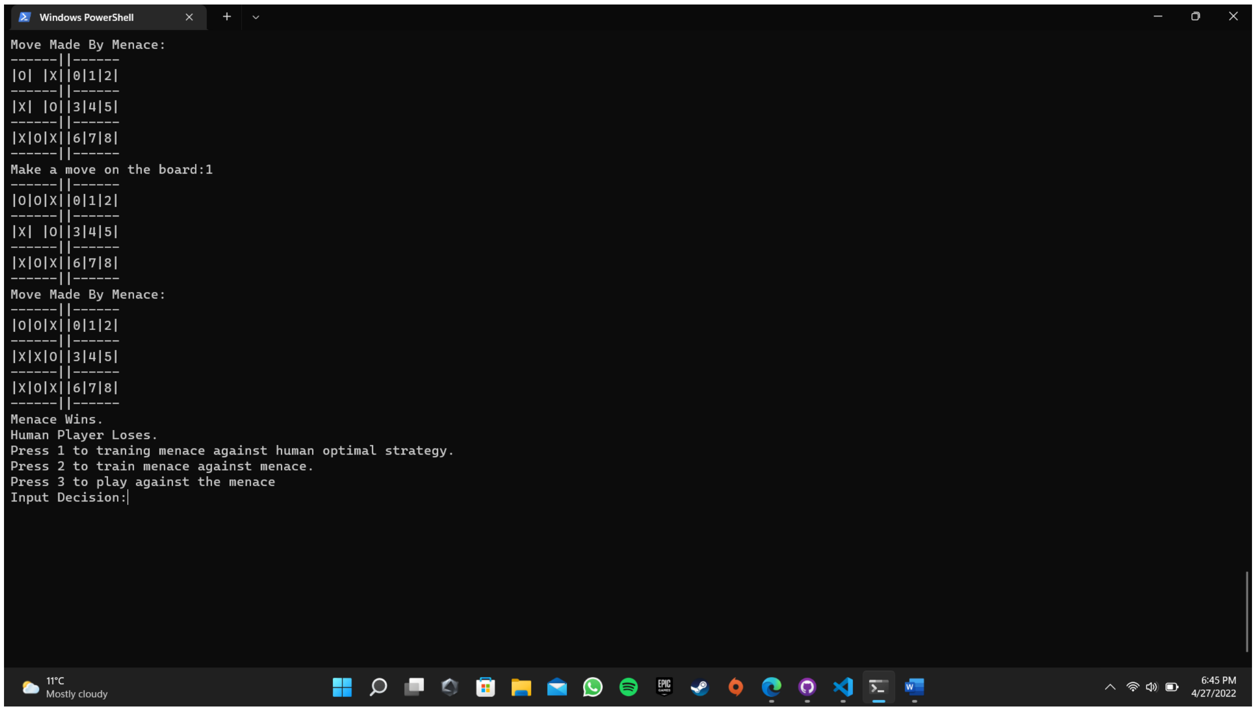
Gameplay:



```
Gameplay now:
------||------
| | | ||0|1|2|
------||------
| | | ||3|4|5|
------||------
| | | ||6|7|8|
------||------
Move Made By Menace:
------||------
| | |X||0|1|2|
------||------
| | | ||3|4|5|
------||------
| | | ||6|7|8|
------||------
Make a move on the board:0
------||------
|O| |X||0|1|2|
------||------
| | | ||3|4|5|
------||------
| | | ||6|7|8|
------||------
Move Made By Menace:
------||------
|O| |X||0|1|2|
------||------
| | | ||3|4|5|
------||------
| | |X||6|7|8|
------||------
Make a move on the board:5
------||------
|O| |X||0|1|2|
------||------
| | |O||3|4|5|
------||------
| | |X||6|7|8|
------||------
```

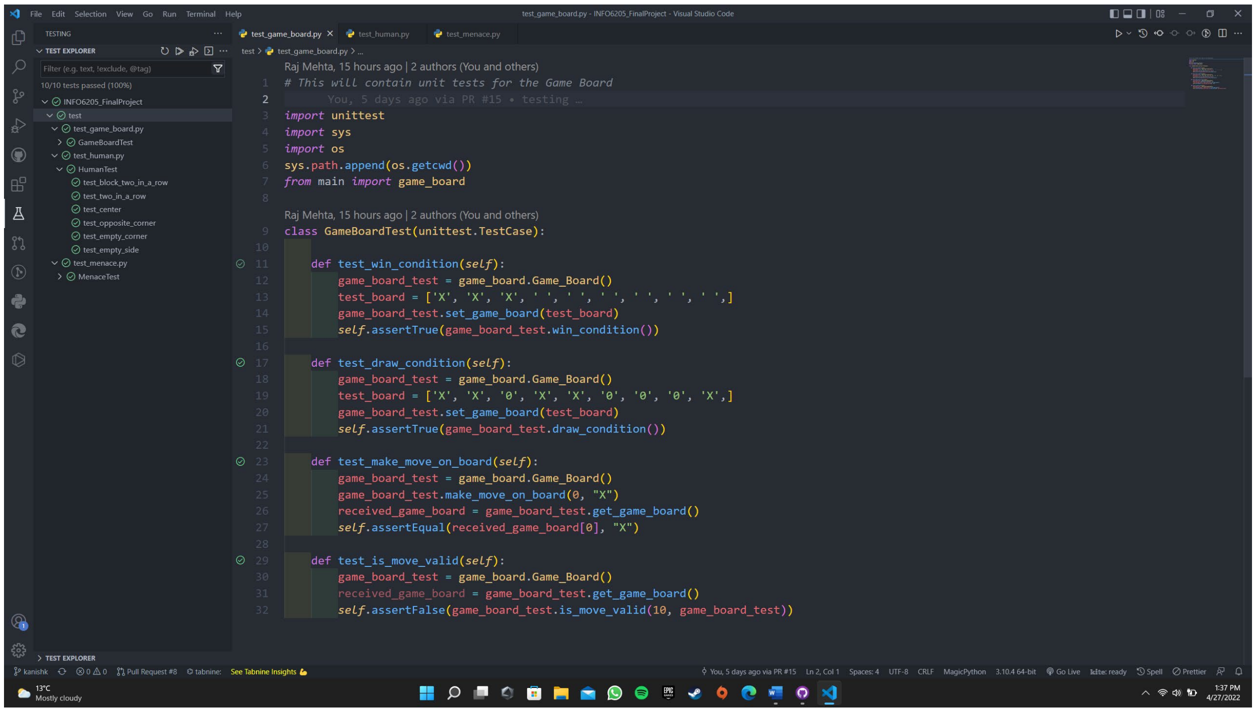


```
------||------
Move Made By Menace:
------||------
|O| |X||0|1|2|
------||------
| | |O||3|4|5|
------||------
|X| |X||6|7|8|
------||------
Make a move on the board:7
------||------
|O| |X||0|1|2|
------||------
| | |O||3|4|5|
------||------
|X|O|X||6|7|8|
------||------
Move Made By Menace:
------||------
|O| |X||0|1|2|
------||------
|X| |O||3|4|5|
------||------
|X|O|X||6|7|8|
------||------
Make a move on the board:1
------||------
|O|O|X||0|1|2|
------||------
|X| |O||3|4|5|
------||------
|X|O|X||6|7|8|
------||------
Move Made By Menace:
------||------
|O|O|X||0|1|2|
------||------
|X|X|O||3|4|5|
------||------
|X|O|X||6|7|8|
```



```
Move Made By Menace:
------||------
|O| |X||0|1|2|
------||------
|X| |O||3|4|5|
------||------
|X|O|X||6|7|8|
------||------
Make a move on the board:1
------||------
|O|O|X||0|1|2|
------||------
|X| |O||3|4|5|
------||------
|X|O|X||6|7|8|
------||------
Move Made By Menace:
------||------
|O|O|X||0|1|2|
------||------
|X|X|O||3|4|5|
------||------
|X|O|X||6|7|8|
------||------
Menace Wins.
Human Player Loses.
Press 1 to traning menace against human optimal strategy.
Press 2 to train menace against menace.
Press 3 to play against the menace
Input Decision:
```

Test Cases:



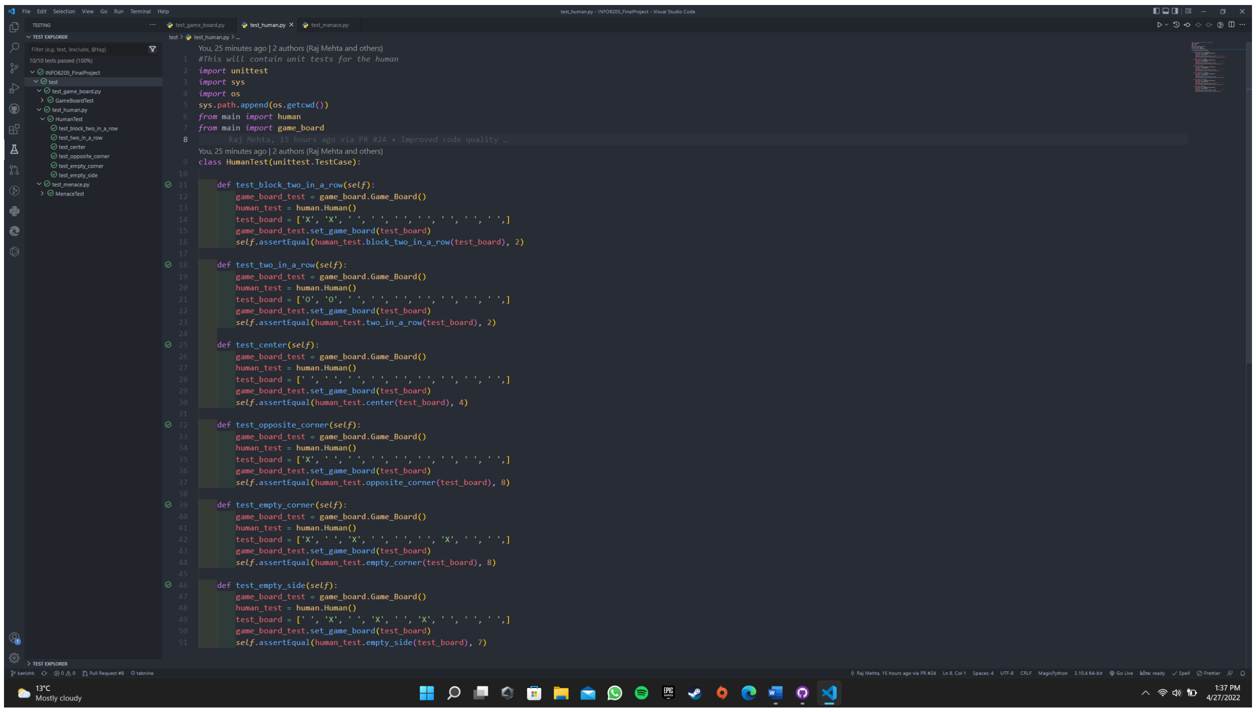
TEST EXPLORER

10/10 tests passed (100%)

- INFO6205_FinalProject
  - test
    - test_game_board.py
      - GameBoardTest
    - test_human.py
      - HumanTest
        - test_block_two_in_a_row
        - test_two_in_a_row
        - test_center
        - test_opposite_corner
        - test_empty_corner
        - test_empty_side
    - test_menace.py
      - MenaceTest

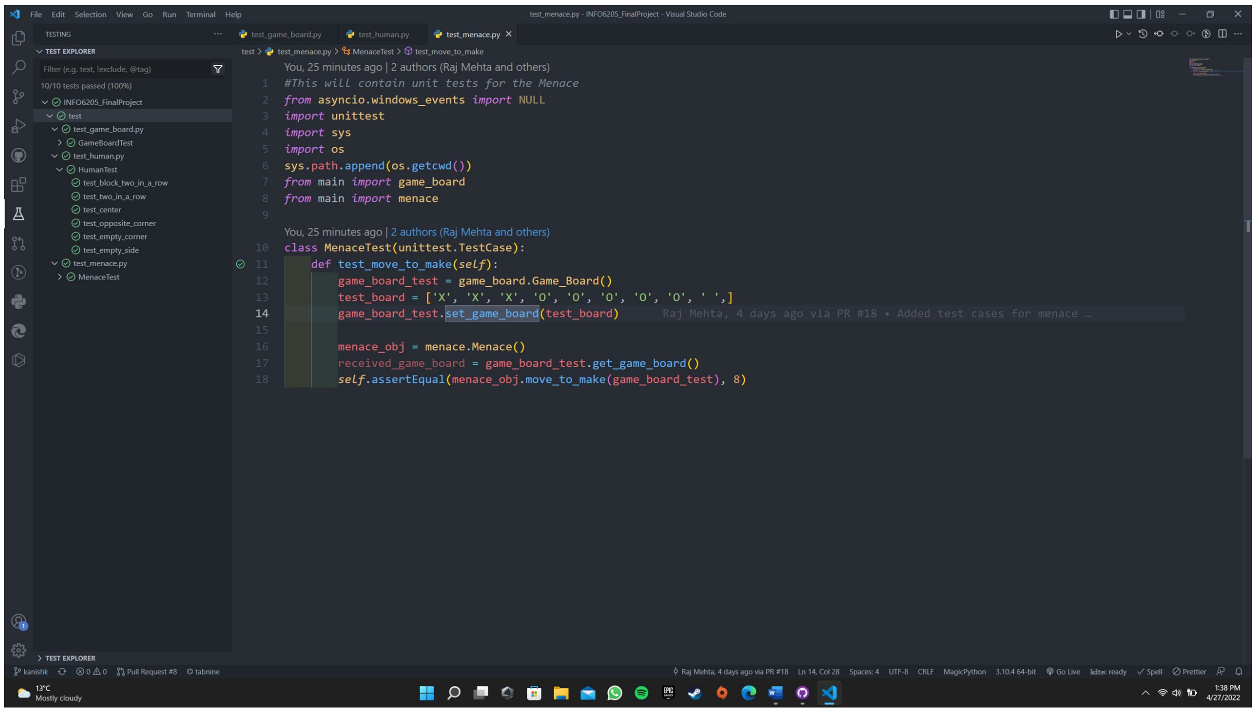
test_game_board.py

```python
# This will contain unit tests for the Game Board

import unittest
import sys
import os
sys.path.append(os.getcwd())
from main import game_board

class GameBoardTest(unittest.TestCase):

    def test_win_condition(self):
        game_board_test = game_board.Game_Board()
        test_board = ['X', 'X', 'X', ' ', ' ', ' ', ' ', ' ', ' ',]
        game_board_test.set_game_board(test_board)
        self.assertTrue(game_board_test.win_condition())

    def test_draw_condition(self):
        game_board_test = game_board.Game_Board()
        test_board = ['X', 'X', 'O', 'X', 'X', 'O', 'O', 'O', 'X',]
        game_board_test.set_game_board(test_board)
        self.assertTrue(game_board_test.draw_condition())

    def test_make_move_on_board(self):
        game_board_test = game_board.Game_Board()
        game_board_test.make_move_on_board(0, "X")
        received_game_board = game_board_test.get_game_board()
        self.assertEqual(received_game_board[0], "X")

    def test_is_move_valid(self):
        game_board_test = game_board.Game_Board()
        received_game_board = game_board_test.get_game_board()
        self.assertFalse(game_board_test.is_move_valid(10, game_board_test))
```

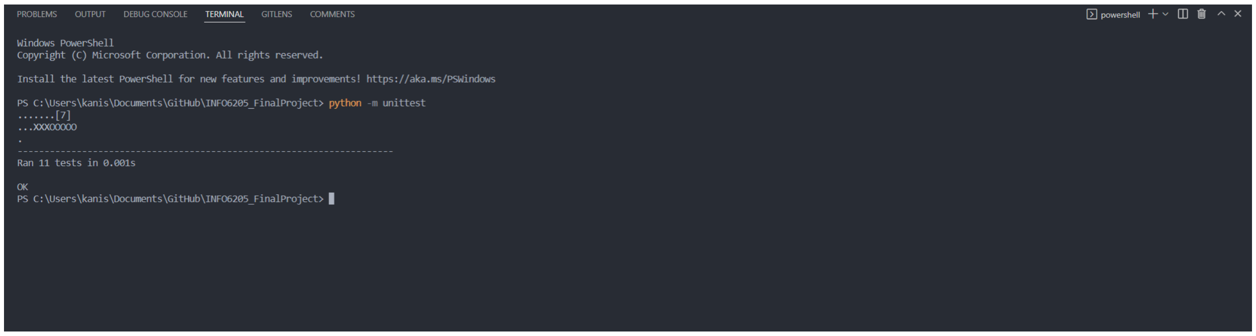


test_human.py

```python
#This will contain unit tests for the human
import unittest
import sys
import os
sys.path.append(os.getcwd())
from main import human
from main import game_board

class HumanTest(unittest.TestCase):

    def test_block_two_in_a_row(self):
        game_board_test = game_board.Game_Board()
        human_test = human.Human()
        test_board = ['X', 'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ',]
        game_board_test.set_game_board(test_board)
        self.assertEqual(human_test.block_two_in_a_row(test_board), 2)

    def test_two_in_a_row(self):
        game_board_test = game_board.Game_Board()
        human_test = human.Human()
        test_board = ['O', 'O', ' ', ' ', ' ', ' ', ' ', ' ', ' ',]
        game_board_test.set_game_board(test_board)
        self.assertEqual(human_test.two_in_a_row(test_board), 2)

    def test_center(self):
        game_board_test = game_board.Game_Board()
        human_test = human.Human()
        test_board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',]
        game_board_test.set_game_board(test_board)
        self.assertEqual(human_test.center(test_board), 4)

    def test_opposite_corner(self):
        game_board_test = game_board.Game_Board()
        human_test = human.Human()
        test_board = ['X', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',]
        game_board_test.set_game_board(test_board)
        self.assertEqual(human_test.opposite_corner(test_board), 8)

    def test_empty_corner(self):
        game_board_test = game_board.Game_Board()
        human_test = human.Human()
        test_board = ['X', ' ', 'X', ' ', ' ', ' ', 'X', ' ', ' ',]
        game_board_test.set_game_board(test_board)
        self.assertEqual(human_test.empty_corner(test_board), 8)

    def test_empty_side(self):
        game_board_test = game_board.Game_Board()
        human_test = human.Human()
        test_board = [' ', 'X', ' ', 'X', ' ', 'X', ' ', ' ', ' ',]
        game_board_test.set_game_board(test_board)
        self.assertEqual(human_test.empty_side(test_board), 7)
```



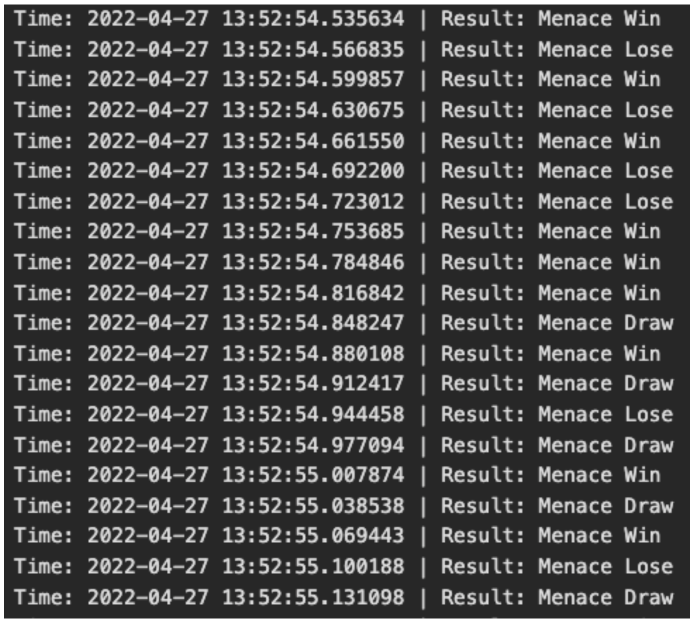
```python
#This will contain unit tests for the Menace
from asyncio.windows_events import NULL
import unittest
import sys
import os
sys.path.append(os.getcwd())
from main import game_board
from main import menace

class MenaceTest(unittest.TestCase):
    def test_move_to_make(self):
        game_board_test = game_board.Game_Board()
        test_board = ['X', 'X', 'X', 'O', 'O', 'O', 'O', 'O', ' ',]
        game_board_test.set_game_board(test_board)

        menace_obj = menace.Menace()
        received_game_board = game_board_test.get_game_board()
        self.assertEqual(menace_obj.move_to_make(game_board_test), 8)
```



```
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\kanis\Documents\GitHub\INFO6205_FinalProject> python -m unittest
.......[7]
...XXXOOOOO
.
----------------------------------------------------------------------
Ran 11 tests in 0.001s

OK
PS C:\Users\kanis\Documents\GitHub\INFO6205_FinalProject>
```

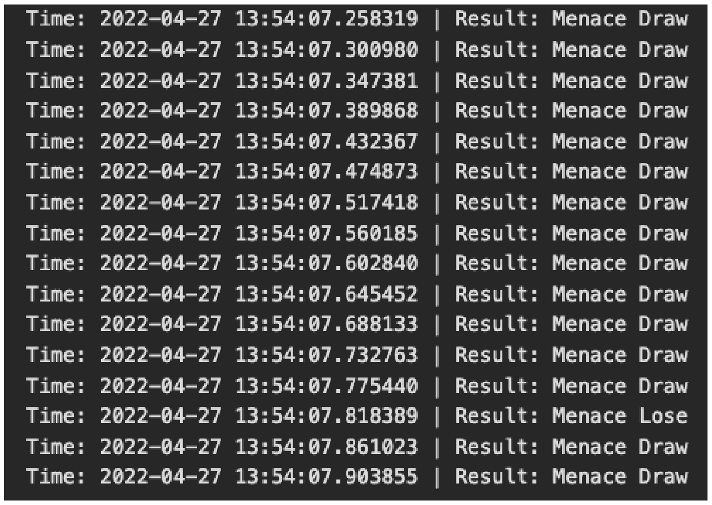
Conclusion:

From the results above, we can conclude that reinforcement learning in even a manageable implementation can be effective to train a machine to do tasks such as playing Noughts and Crosses. Our machine might take a very small number of iterations, say 5000 to train itself against the Human Optimal Strategy in a way where the Menace will be close to unbeatable. However, when playing against the user, we notice that Menace will at times not make the most obvious moves. Training it against another Menace, both of which are trying to make themselves stronger than the other with each passing game exposes it to game states it has not seen before. As shown in the screenshots in this section, training two Menaces continuously, we could notice that the games go from wins and losses to mostly draws. This, logically, occurs when the Menace has eventually learned almost all possible game states and knows what move to play to 'not lose'.



```
Time: 2022-04-27 13:52:54.535634 | Result: Menace Win
Time: 2022-04-27 13:52:54.566835 | Result: Menace Lose
Time: 2022-04-27 13:52:54.599857 | Result: Menace Win
Time: 2022-04-27 13:52:54.630675 | Result: Menace Lose
Time: 2022-04-27 13:52:54.661550 | Result: Menace Win
Time: 2022-04-27 13:52:54.692200 | Result: Menace Lose
Time: 2022-04-27 13:52:54.723012 | Result: Menace Lose
Time: 2022-04-27 13:52:54.753685 | Result: Menace Win
Time: 2022-04-27 13:52:54.784846 | Result: Menace Win
Time: 2022-04-27 13:52:54.816842 | Result: Menace Win
Time: 2022-04-27 13:52:54.848247 | Result: Menace Draw
Time: 2022-04-27 13:52:54.880108 | Result: Menace Win
Time: 2022-04-27 13:52:54.912417 | Result: Menace Draw
Time: 2022-04-27 13:52:54.944458 | Result: Menace Lose
Time: 2022-04-27 13:52:54.977094 | Result: Menace Draw
Time: 2022-04-27 13:52:55.007874 | Result: Menace Win
Time: 2022-04-27 13:52:55.038538 | Result: Menace Draw
Time: 2022-04-27 13:52:55.069443 | Result: Menace Win
Time: 2022-04-27 13:52:55.100188 | Result: Menace Lose
Time: 2022-04-27 13:52:55.131098 | Result: Menace Draw
```

Training Logs: Start of Training



```
Time: 2022-04-27 13:54:07.258319 | Result: Menace Draw
Time: 2022-04-27 13:54:07.300980 | Result: Menace Draw
Time: 2022-04-27 13:54:07.347381 | Result: Menace Draw
Time: 2022-04-27 13:54:07.389868 | Result: Menace Draw
Time: 2022-04-27 13:54:07.432367 | Result: Menace Draw
Time: 2022-04-27 13:54:07.474873 | Result: Menace Draw
Time: 2022-04-27 13:54:07.517418 | Result: Menace Draw
Time: 2022-04-27 13:54:07.560185 | Result: Menace Draw
Time: 2022-04-27 13:54:07.602840 | Result: Menace Draw
Time: 2022-04-27 13:54:07.645452 | Result: Menace Draw
Time: 2022-04-27 13:54:07.688133 | Result: Menace Draw
Time: 2022-04-27 13:54:07.732763 | Result: Menace Draw
Time: 2022-04-27 13:54:07.775440 | Result: Menace Draw
Time: 2022-04-27 13:54:07.818389 | Result: Menace Lose
Time: 2022-04-27 13:54:07.861023 | Result: Menace Draw
Time: 2022-04-27 13:54:07.903855 | Result: Menace Draw
```

Training Logs: End of Training

References:

1. Tic-Tac-Toe Strategy
   https://en.wikipedia.org/wiki/Tic-tac-toe

2. Reinforcement Learning Algorithm
   https://www.techtarget.com/searchenterpriseai/definition/reinforcement-learning#:~:text=Reinforcement%20learning%20is%20a%20machine,learn%20through%20trial%20and%20error.

3. Tic Tac Toe Fork Generator
   https://gabegamedev.com/2017/05/31/first-blog-post/

4. Unittest for Python Testing
   https://docs.python.org/3/library/unittest.html