Name: Karan Dua
Student Id: 21331391

Design Choices

Tic-Tac-Toe game

1. I have designed Semi-Intelligent player in tic-tac-toe game to perform following tasks:
   - If Semi-Intelligent player has a winning move, it will **execute that winning move**.
   - If Q-Learning or MinMax player has a winning move, it will **block that winning move**.
   - If above 2 conditions are not met, then it will play a **random valid move**.

2. To implement **Q-Learning algorithm**, I have made the following design choices:
   - I have trained Q-Learning model for **3,000,000 episodes** overall against a semi-intelligent player. It resulted in total **5020 states**. All the Q-Learning states were stored in a pickle file to be referenced later. I have tried training Q-Learning for more episodes but it was taking a lot of time and resulting in crashing of kernel.
   - Once model is trained, I **tested for 2000 games** against Semi-Intelligent player.
   - When playing against a Semi-Intelligent player, I have tried **3 variations** of the game:
     a) **Player is chosen randomly** between Semi-Intelligent player and Q-Learning player to take first move in the game.
     b) **Semi-Intelligent player always takes the first move** in the game.
     c) **Q-Learning player always takes the first move** in the game.

3. To implement **MinMax algorithm**, I have made the following design choices:
   - I have implemented MinMax algorithm **without Alpha-Beta pruning** and tested it against Semi-Intelligent player for **100 games**. I tried playing more games but it was taking a lot of time so I decide to restrict it to 100 games.
   - I have implemented MinMax algorithm **with Alpha-Beta pruning** and tested it against Semi-Intelligent player for **1000 games**.
   - When playing against a Semi-Intelligent player, I have tried **3 variations** of the game:
     a) **Player is chosen randomly** between Semi-Intelligent player. and MinMax player to take first move in the game.
     b) **Semi-Intelligent player always takes the first move** in the game.
     c) **MinMax player always takes the first move** in the game.

Connect4 game

1. All the games were played on **6X7 Connect4 board**.
2. I have designed Semi-Intelligent player in tic-tac-toe game to perform following tasks:
   - If Semi-Intelligent player has a winning move, it will **execute that winning move**.
   - If Q-Learning or MinMax player has a winning move, it will **block that winning move**.
   - If above 2 conditions are not met, then it will play a **random valid move**.

3. To implement **Q-Learning algorithm**, I have made the following design choices:
   - I have trained Q-Learning model for **3,000,000 episodes** overall against a semi-intelligent player. It resulted in total **11,822,743 states**. All the Q-Learning states were stored in a pickle file to be referenced later. I have tried training Q-Learning for more episodes but it was taking a lot of time and resulting in crashing of kernel.
   - Once model is trained, I **tested for 2000 games** against Semi-Intelligent player.
   - When playing against a Semi-Intelligent player, I have tried **3 variations** of the game:

a) **Player is chosen randomly** between Semi-Intelligent player and Q-Learning player to take first move in the game.
b) **Semi-Intelligent player always takes the first move** in the game.
c) **Q-Learning player always takes the first move** in the game.

4. To implement **MinMax algorithm**, I have made the following design choices:
   - I have implemented MinMax algorithm **with Alpha-Beta pruning for depth 8** and tested it against Semi-Intelligent player for **100 games**. I tried playing more games but it was taking a lot of time so I decide to restrict it to 100 games.
   - I have implemented MinMax algorithm **with Alpha-Beta pruning for depth 6** and tested it against Semi-Intelligent player for **100 games**.
   - I tried to test this **using higher depth option like 10 and 15**, but it was taking a lot of time and resulting in crashing of kernel.
   - I have also tried to test without Alpha-Beta pruning, but it was taking a lot of time and resulting in crashing of kernel.
   - When playing against a Semi-Intelligent player, I have tried **3 variations** of the game:
     d) **Player is chosen randomly** between Semi-Intelligent player. and MinMax player to take first move in the game.
     e) **Semi-Intelligent player always takes the first move** in the game.
     f) **MinMax player always takes the first move** in the game.

Name: Karan Dua
Student Id: 21331391

Comparisons of different algorithms

I have compared performance these algorithms on different parameters. The primary reason for this is that it will help us the gauge their performance under different scenarios and help us better understand their functioning.

Q-Learning implementation of Tic-Tac-Toe game

1. Training Q-Learning model

Below table summarises the results for training Q-Learning player against Semi-Intelligent player:

Tic-Tac-Toe GameType	Q-Learning Wins	Semi-Intelligent Wins	Draw
Training	2250425	586352	163223

Table 1: Training Performance of Q-Learning vs. Semi-Intelligent player

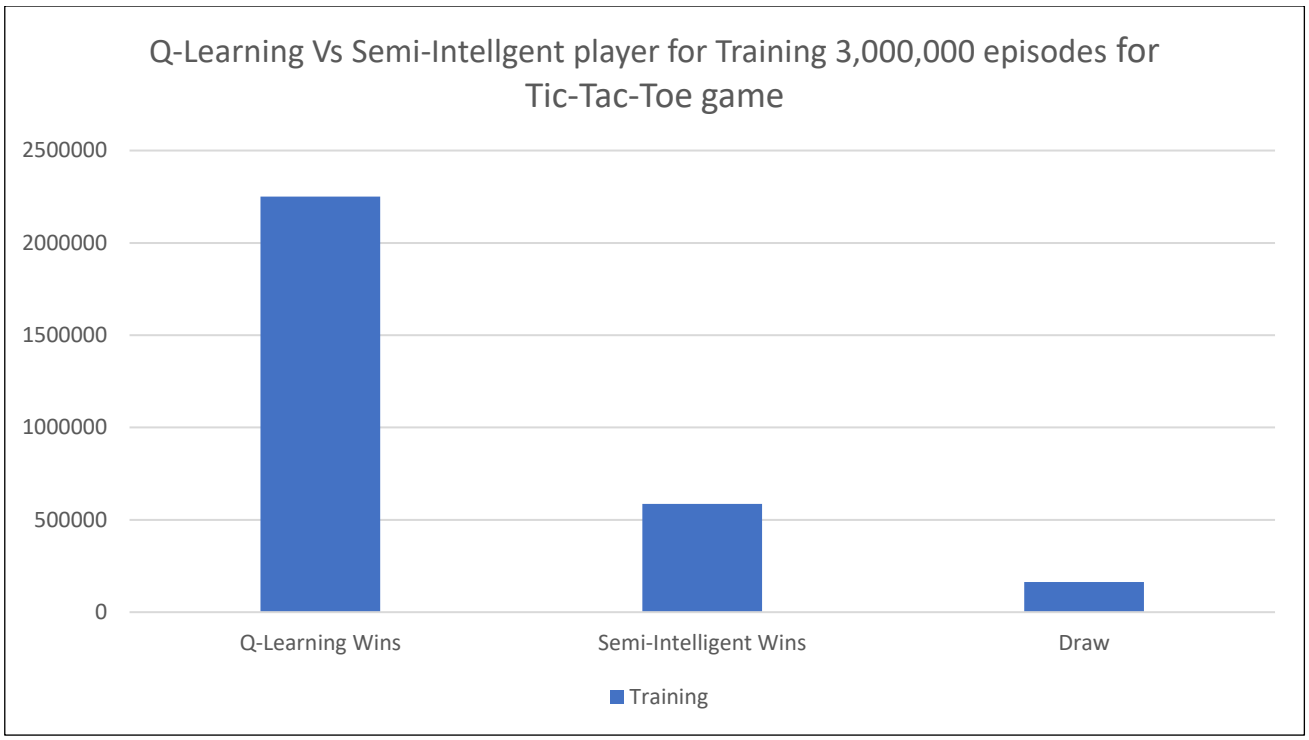

Figure 1: Graphs comparing Training performance of Q-Learning vs. Semi-Intelligent player

From above graphs I can conclude that Q-Learning player performs better than Semi-Intelligent player for majority of the games played. The primary reason for this behaviour is that during early phases of training, Q-Learning model starts to learn about states that lead to overall win and starts giving these states more weight. Once it has learned sufficiently, it starts to play smart moves that always lead to win or draw game.

2. Q-Learning model vs. Semi-Intelligent player

Below table summarises the results for training Q-Learning player against Semi-Intelligent player:

Tic-Tac-Toe GameType	Q-Learning Wins	Semi-Intelligent Wins	Draw
First Move: Random Player	869	862	269
First Move: Q-Learning Player	1183	572	245
First Move: Semi-Intelligent Player	599	1128	273

Table 2: Performance of Q-Learning vs. Semi-Intelligent player for 2000 games

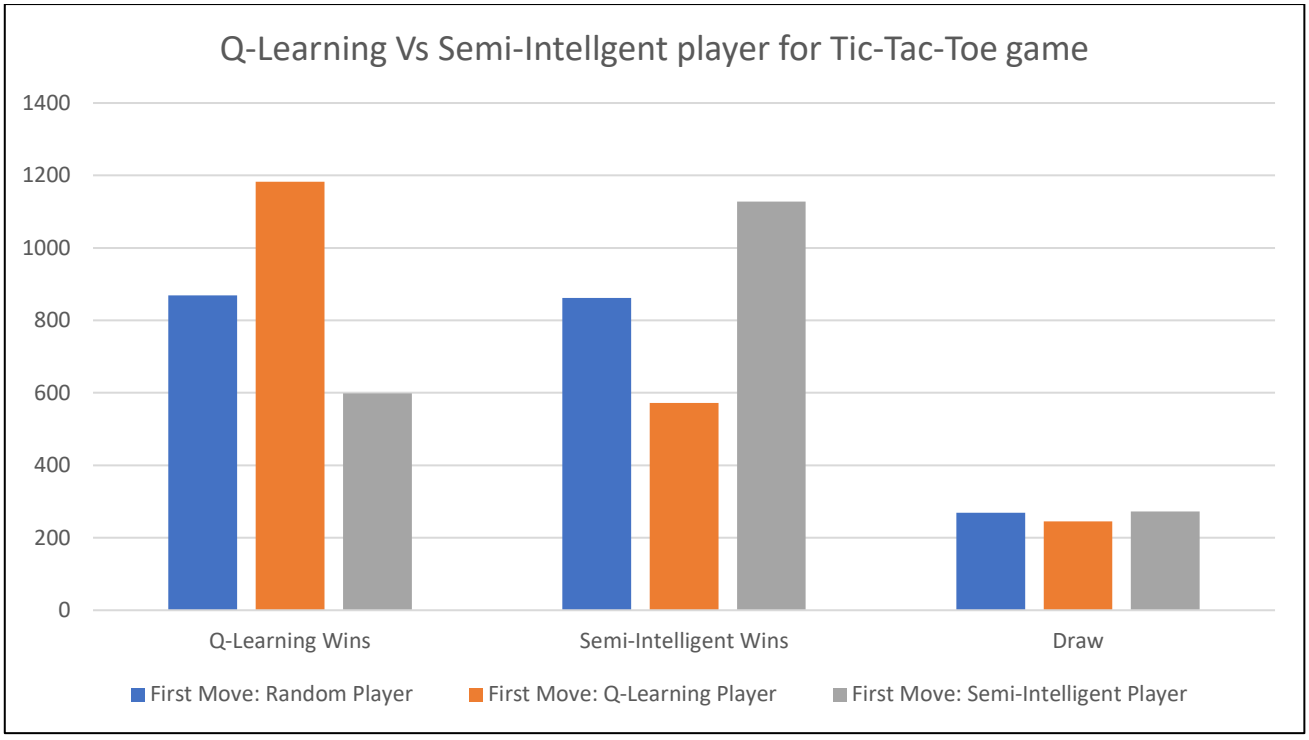

Figure 2: Graphs comparing performance of Q-Learning vs. Semi-Intelligent player

From above graphs I can conclude that when Q-Learning player take the first moves, it performs exceptionally well. However, when Semi-Intelligent player takes the first move, Q-Learning does not perform optimally. The primary reason for this behaviour could be that Q-Learning was trained for always taking the first move and hence, its performance degrades when it takes the second move in the game.

Likewise, when a random player is chosen to take the first move, both the players perform equally well.

Overall, I can conclude that to improve the performance of Q-Learning against Semi-Intelligent player, it has to be trained for playing both first and second move in the game.

MinMax implementation of Tic-Tac-Toe game

1. Time performance comparison for MinMax with Alpha-Beta pruning and without Alpha-Beta pruning

Below table summarises the results comparing the performance of different implementations of MinMax algorithm.

Tic-Tac-Toe GameType	Time taken in seconds (100 games): MinMax without Alpha-Beta pruning	Time taken in seconds (1000 games): MinMax without Alpha-Beta pruning
First Move: Random Player	448.515099	176.37148
First Move: MinMax Player	481.483409	193.123701
First Move: Semi-Intelligent Player	454.219773	169.496459

Table 3: Performance of different implementations MinMax algorithm

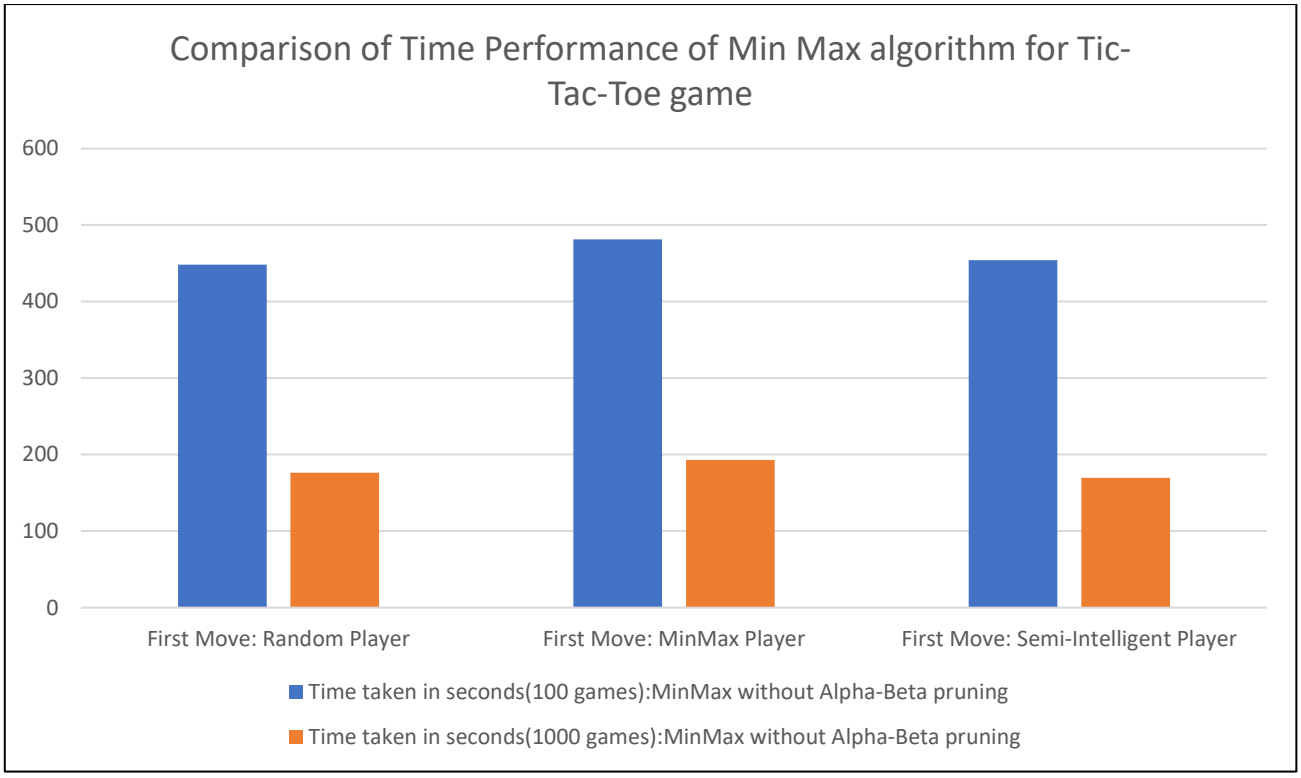

Figure 3: Graphs comparing performance of different implementations MinMax algorithm

From above graphs I can conclude that MinMax algorithm takes very less time when it is implemented with Alpha-Beta pruning. I played 100 games with MinMax without Alpha-Beta pruning and 1000 game with MinMax with Alpha-Beta pruning and playing 1000 games is taking less time than playing 100 games. Also, playing different variations of the game has not impact on the performance of the algorithm.

Thus, I can conclude that MinMax algorithm with Alpha-Beta pruning is the optimised way to implement MinMax algorithm.

2. MinMax without Alpha-Beta pruning vs. Semi-Intelligent player

Below table summarises the results for MinMax player against Semi-Intelligent player:

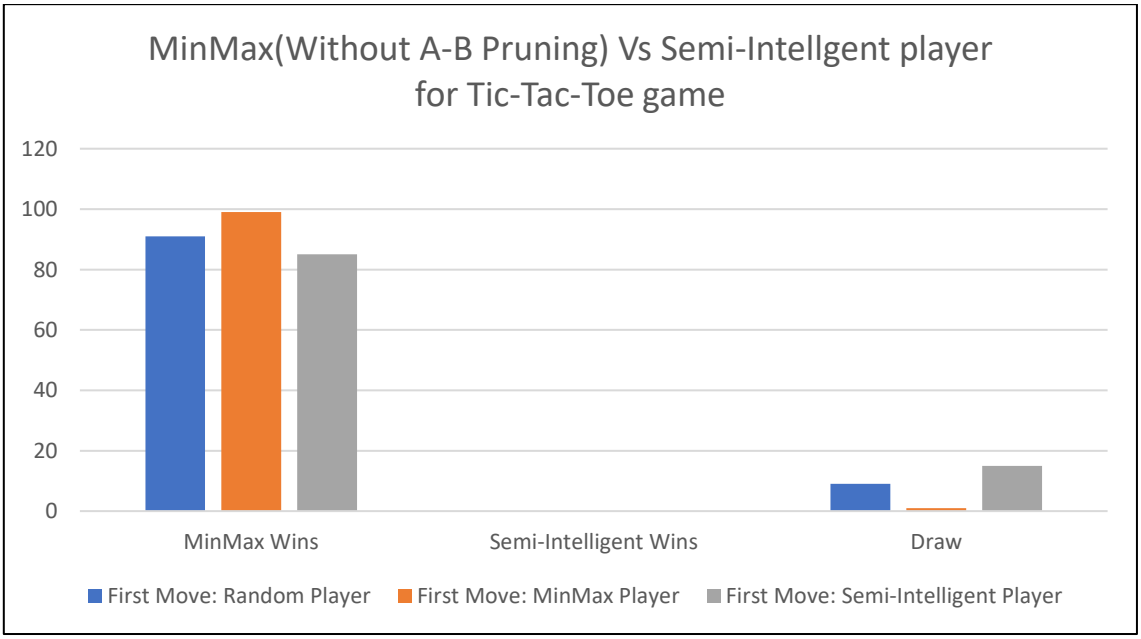

Figure 4: Graphs comparing performance of MinMax without Alpha-Beta pruning vs. Semi-Intelligent player

Tic-Tac-Toe GameType	MinMax Wins	Semi-Intelligent Wins	Draw
First Move: Random Player	91	0	9
First Move: MinMax Player	99	0	1
First Move: Semi-Intelligent Player	85	0	15

Table 4: Performance of MinMax without Alpha-Beta pruning vs. Semi-Intelligent player for 100 games

From above graphs I can conclude that MinMax player wins comprehensively against the Semi-Intelligent player for all variations of the game. Semi-Intelligent player did not win a single game and is only able to execute a few draw games when it goes first. The primary reason for this behaviour is that MinMax algorithm evaluates best move for every state of the board. Therefore, there is a high probability of MinMax player either winning or drawing the game.

3. MinMax with Alpha-Beta pruning vs. Semi-Intelligent player

Below table summarises the results for MinMax player against Semi-Intelligent player:

Tic-Tac-Toe GameType	MinMax Wins	Semi-Intelligent Wins	Draw
First Move: Random Player	899	0	101
First Move: MinMax Player	993	0	7
First Move: Semi-Intelligent Player	818	0	182

Table 5: Performance of MinMax with Alpha-Beta pruning vs. Semi-Intelligent player for 1000 games

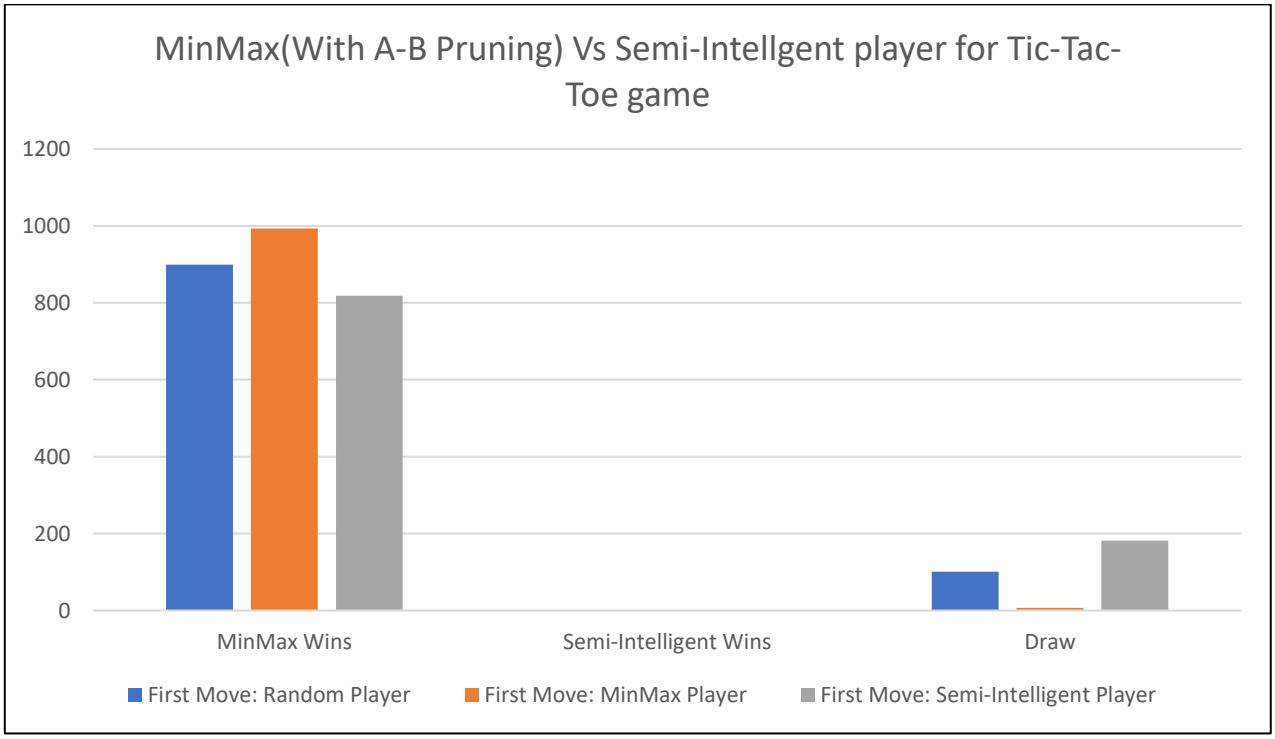

Figure 5: Graphs comparing performance of MinMax with Alpha-Beta pruning vs. Semi-Intelligent player

From above graphs I can conclude that MinMax player wins comprehensively against the Semi-Intelligent player for all variations of the game. Similar to previous results, Semi-Intelligent player did not win a single game and is only able to execute a few draw games when it goes first.

Overall, I can conclude that both implementations of MinMax player wins comprehensively against the Semi-Intelligent player. Furthermore, MinMax with Alpha-Beta pruning performs similar to Min-Max without Alpha-Beta pruning. Therefore, we should always prefer Min-Max with Alpha-Beta pruning for implementing this algorithm as it has better performance than the other.

MinMax vs. Q-Learning for Tic-Tac-Toe game

I have compared the performance of MinMax with Q-Learning for 2000 games. Below table summarises the results:

Tic-Tac-Toe GameType	Q-Learning Wins	MinMax Wins	Draw
First Move: Random Player	206	1196	598
First Move: Q-Learning Player	318	906	776
First Move: MinMax Player	0	1809	191

Table 6: Performance of MinMax with Alpha-Beta pruning vs. Q-Learning player for 100 games

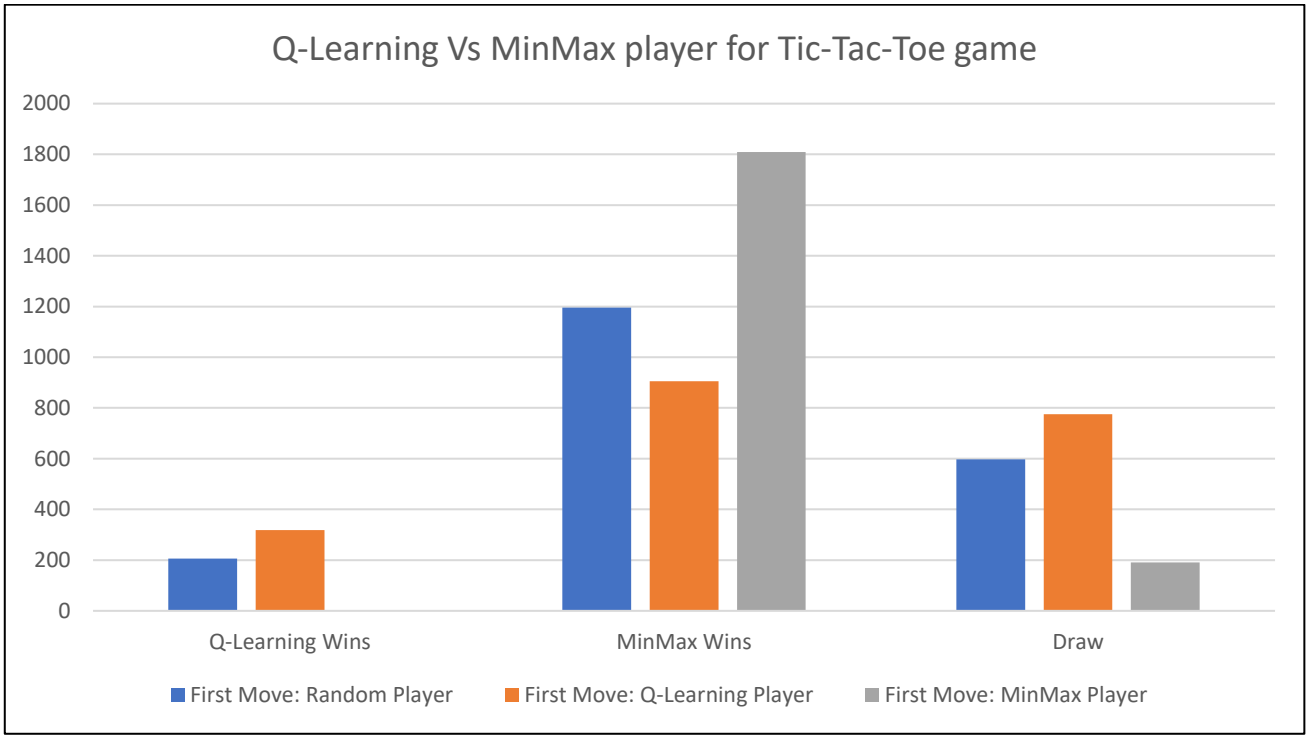

Figure 6: Graphs comparing performance of MinMax with Alpha-Beta pruning vs. Q-Learning player

From above graphs I can conclude that MinMax again plays comprehensively well against Q-Learning and wins most of the games irrespective of whether it plays first move or second move in the game. Furthermore, MinMax never loses the game when it plays first move in the game. It either wins the game or draws the game.

On the other hand, Q-Learning was able to win or draw games when it played first move in the game. It could not win a game when it played second move in the game.

Overall, I can conclude that MinMax algorithm with Alpha-Beta pruning is the ideal algorithm to play Tic-Tac-Toe game. It takes less time to train and performs better than Q-Learning. Furthermore, its performance is not significantly dependent on whether it plays first move or second move in the game.

Q-Learning implementation of Connect4 game

1. Training Q-Learning model

Below table summarises the results for training Q-Learning player against Semi-Intelligent player:

Connect4 GameType	Q-Learning Wins	Semi-Intelligent Wins	Draw
Training	2943920	55547	533

Table 7: Training Performance of Q-Learning vs. Semi-Intelligent player

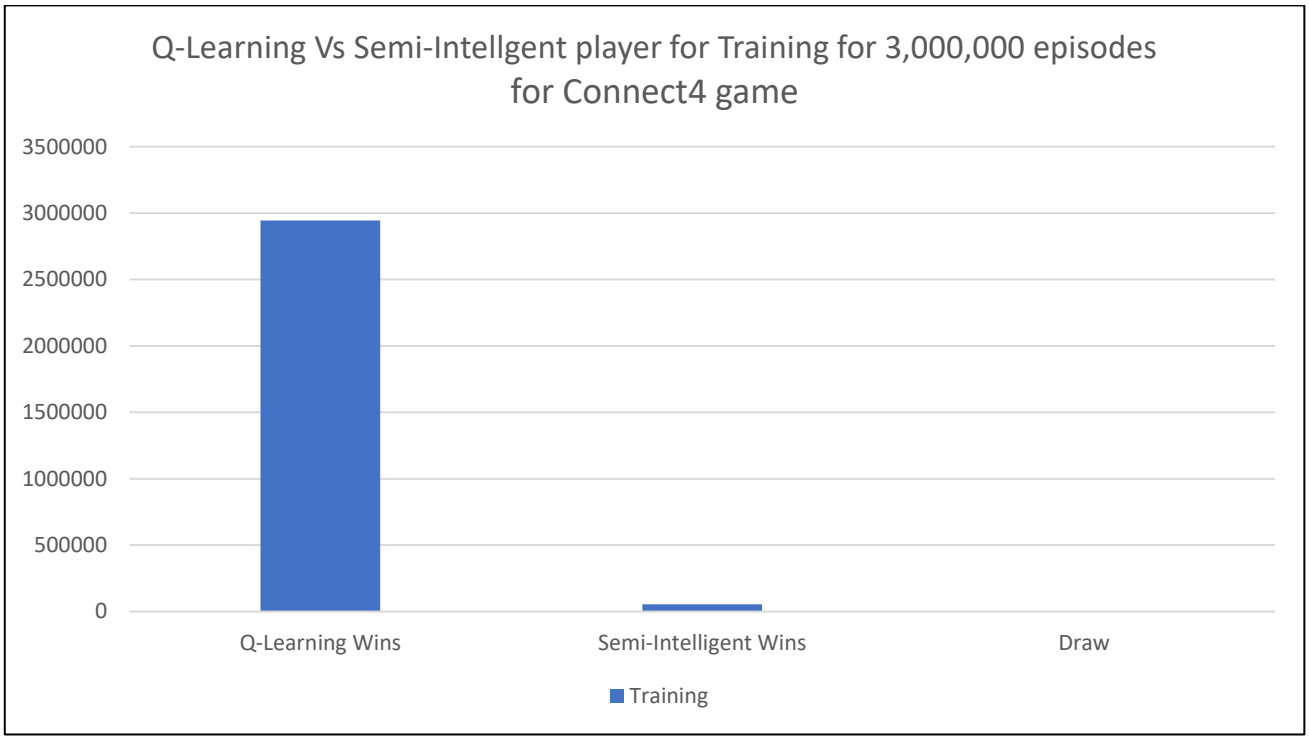

Figure 7: Graphs comparing Training performance of Q-Learning vs. Semi-Intelligent player

From above graphs I can conclude that Q-Learning player performs better than Semi-Intelligent player and wins majority of the games played. The primary reason for this behaviour is that during early phases of training, Q-Learning model starts to learn about states that lead to overall win and starts giving these states more weight. Once it has learned sufficiently, it starts to play smart moves that always lead to win or draw game.

2. Q-Learning model vs. Semi-Intelligent player

Below table summarises the results for training Q-Learning player against Semi-Intelligent player:

Connect 4 GameType	Q-Learning Wins	Semi-Intelligent Wins	Draw
First Move: Random Player	1277	694	29
First Move: Q-Learning Player	1322	646	32
First Move: Semi-Intelligent Player	1225	740	35

Table 8: Performance of Q-Learning vs. Semi-Intelligent player for 2000 games

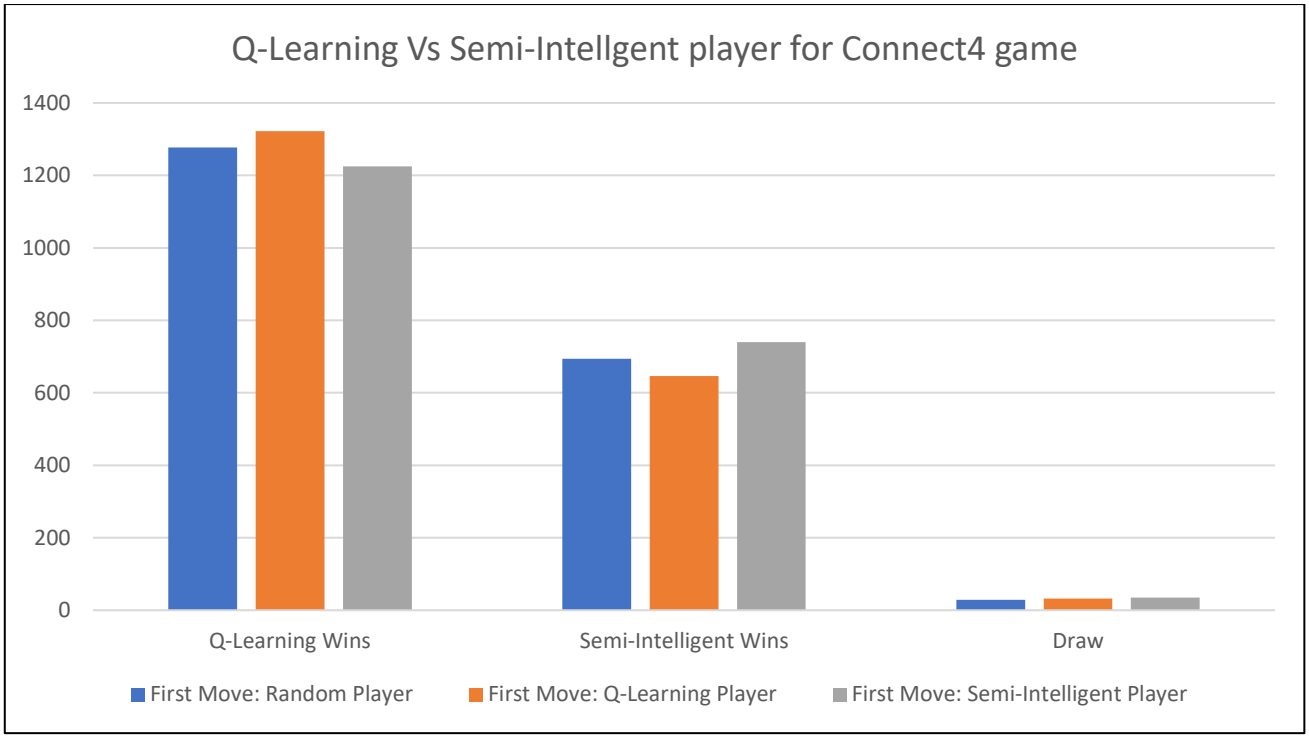

Figure 8: Graphs comparing performance of Q-Learning vs. Semi-Intelligent player

From above graphs I can conclude that when Q-Learning player performs well against Semi-Intelligent player and wins almost double then number of games irrespective of whether it goes first or second. However, when Semi-Intelligent player takes the first move, it wins slightly more games when compared when it goes second.

Overall, I can conclude that our Q-Learning agent has trained well and learned a lot of winning states during training process. Therefore, the Q-Learning player performs well against Semi-Intelligent player.

MinMax implementation of Connect4 game

1. Time performance comparison for MinMax with depth = 6 and depth = 8

Below table summarises the results comparing the performance of different implementations of MinMax algorithm.

Connect4 GameType	Time taken in seconds (100 games): MinMax with Depth = 8	Time taken in seconds (100 games): MinMax with Depth = 6
First Move: Random Player	956.59	133.962062
First Move: MinMax Player	929.250072	141.769968
First Move: Semi-Intelligent Player	937.176666	139.694996

Table 9: Performance of different implementations MinMax algorithm

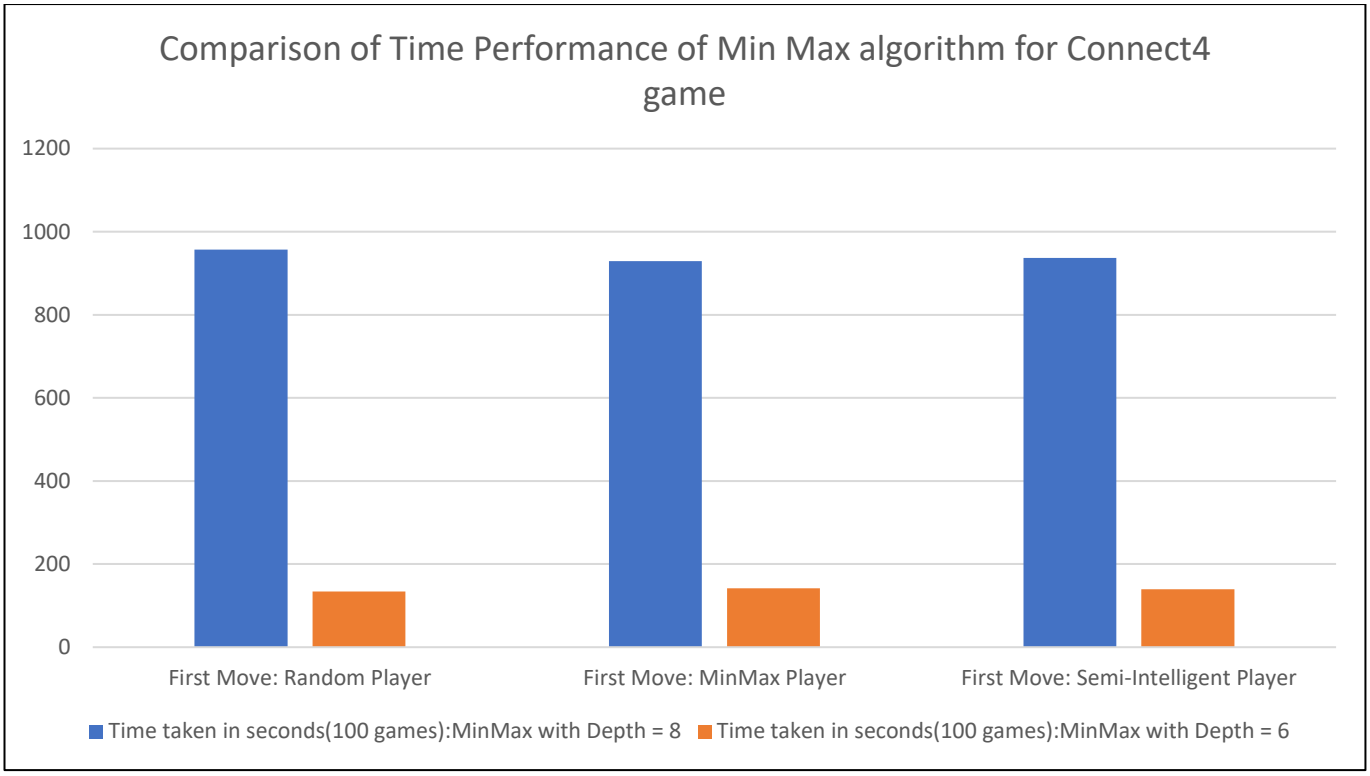

Figure 9: Graphs comparing performance of different implementations MinMax algorithm

From above graphs I can conclude that MinMax algorithm takes very less time when it is implemented with Depth=6. Also, playing different variations of the game has not impact on the performance of the algorithm. Please note that MinMax is implemented with Alpha-Beta pruning mechanism for both implementations.

2. MinMax with Alpha-Beta pruning and Depth = 8 vs. Semi-Intelligent player

Below table summarises the results for MinMax player against Semi-Intelligent player:

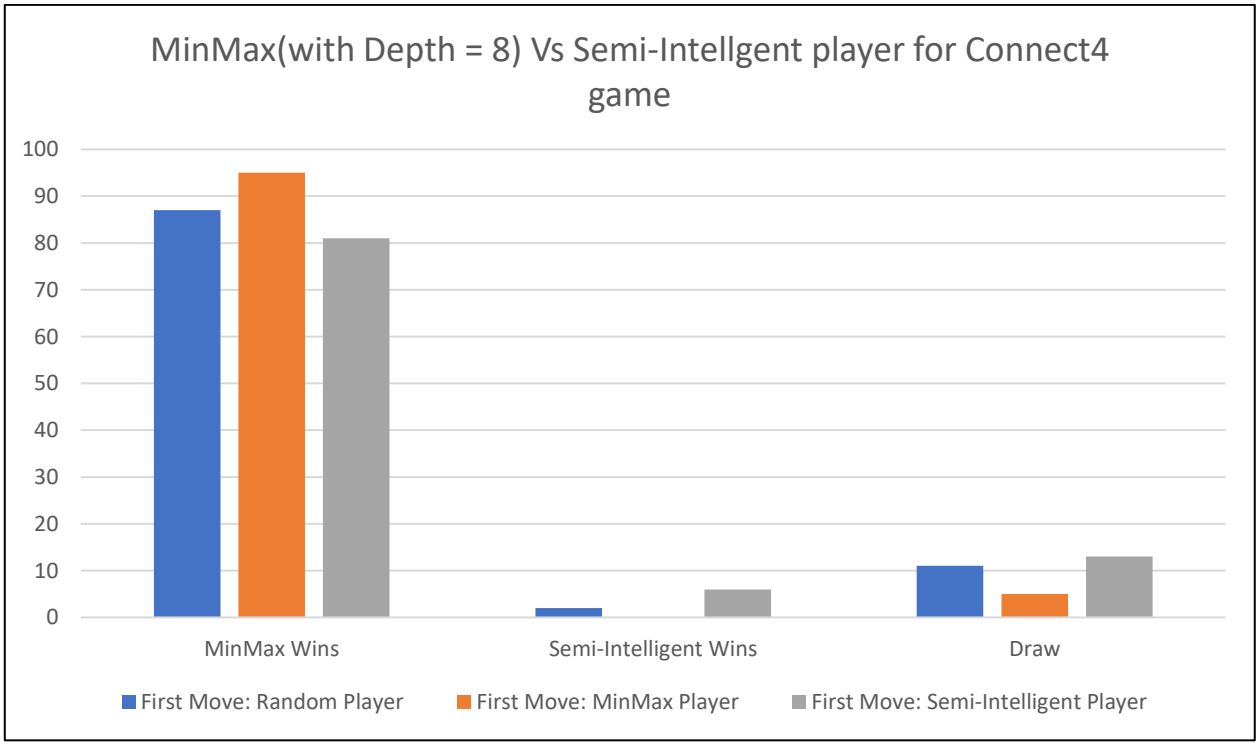

Figure 10: Graphs comparing performance of MinMax with Depth = 8 vs. Semi-Intelligent player

Connect4 GameType	MinMax Wins	Semi-Intelligent Wins	Draw
First Move: Random Player	87	2	11
First Move: MinMax Player	95	0	5
First Move: Semi-Intelligent Player	81	6	13

Table 10: Performance of MinMax with Depth = 8 vs. Semi-Intelligent player

From above graphs I can conclude that MinMax player wins comprehensively against the Semi-Intelligent player for all variations of the game. Semi-Intelligent player was able to win or draw a few games only when it played first move in the game. The primary reason for this behaviour is that MinMax algorithm evaluates best move for every state of the board. Therefore, there is a high probability of MinMax player either winning or drawing the game.

3. MinMax with Alpha-Beta pruning and Depth = 6 vs. Semi-Intelligent player

Below table summarises the results for MinMax player against Semi-Intelligent player:

Connect4 GameType	MinMax Wins	Semi-Intelligent Wins	Draw
First Move: Random Player	55	4	41
First Move: MinMax Player	67	1	32
First Move: Semi-Intelligent Player	44	2	53

Table 11: Performance of MinMax with Depth = 6 vs. Semi-Intelligent player

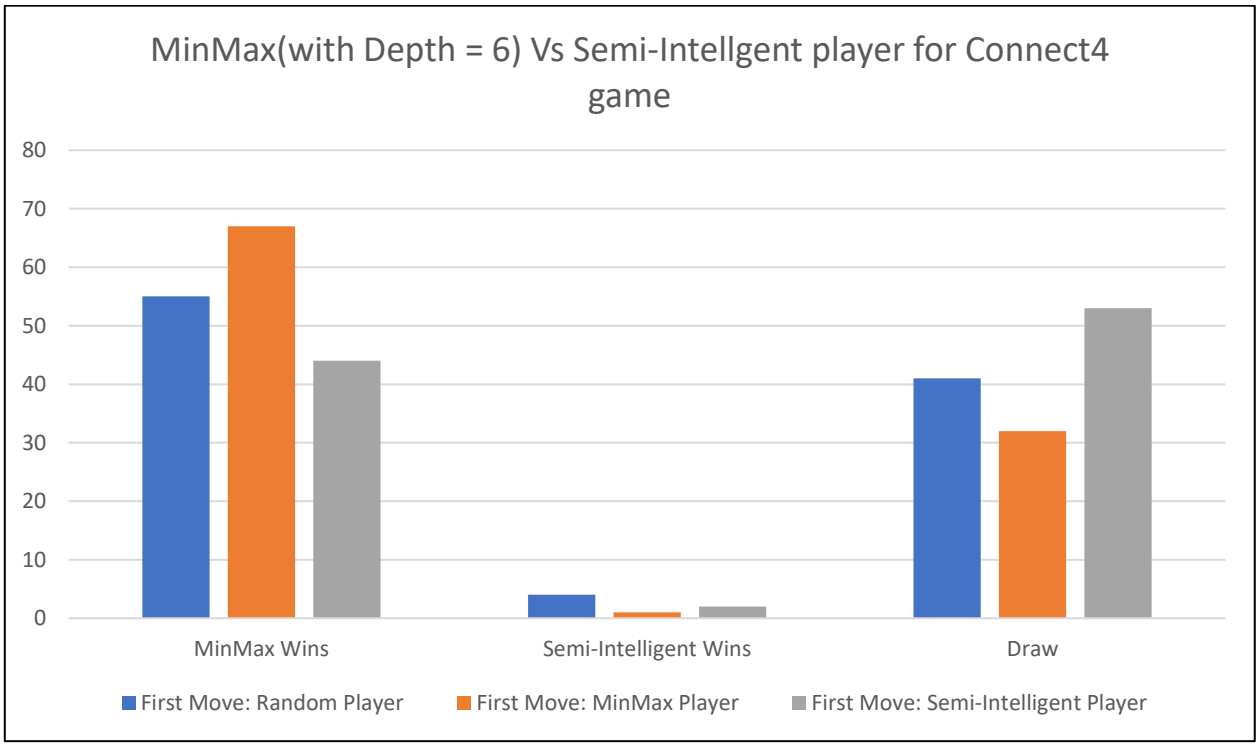

Figure 11: Graphs comparing performance of MinMax with Depth = 6 vs. Semi-Intelligent player

From above graphs I can conclude that performance of MinMax player has degraded when compared to previous version. It was able to win highest when it played first move in the game. However, it was able to draw most of the games when it played second move in the game. Overall, it lost a very few games but almost half of the games ended in a draw irrespective which player plays first move in the game.

Overall, I can conclude the depth of MinMax tree plays a very significant role in its overall performance. Its performance degrades when the depth of tree is reduced. On the other hand, its performance also decreases and it takes a lot more time to finish the game if we increase the depth of tree. Therefore, it's a trade-off between performance and winnability of the MinMax algorithm. I have chosen MinMax with Depth = 8 to increase its winnability chances.

MinMax vs. Q-Learning for Connect4 game

I have compared the performance of MinMax with Q-Learning for 2000 games. Below table summarises the results:

Connect4 GameType	Q-Learning Wins	MinMax Wins	Draw
First Move: Random Player	40	125	35
First Move: Q-Learning Player	69	98	33
First Move: MinMax Player	8	167	25

Table 12: Performance of MinMax with Depth = 8 vs. Q-Learning player

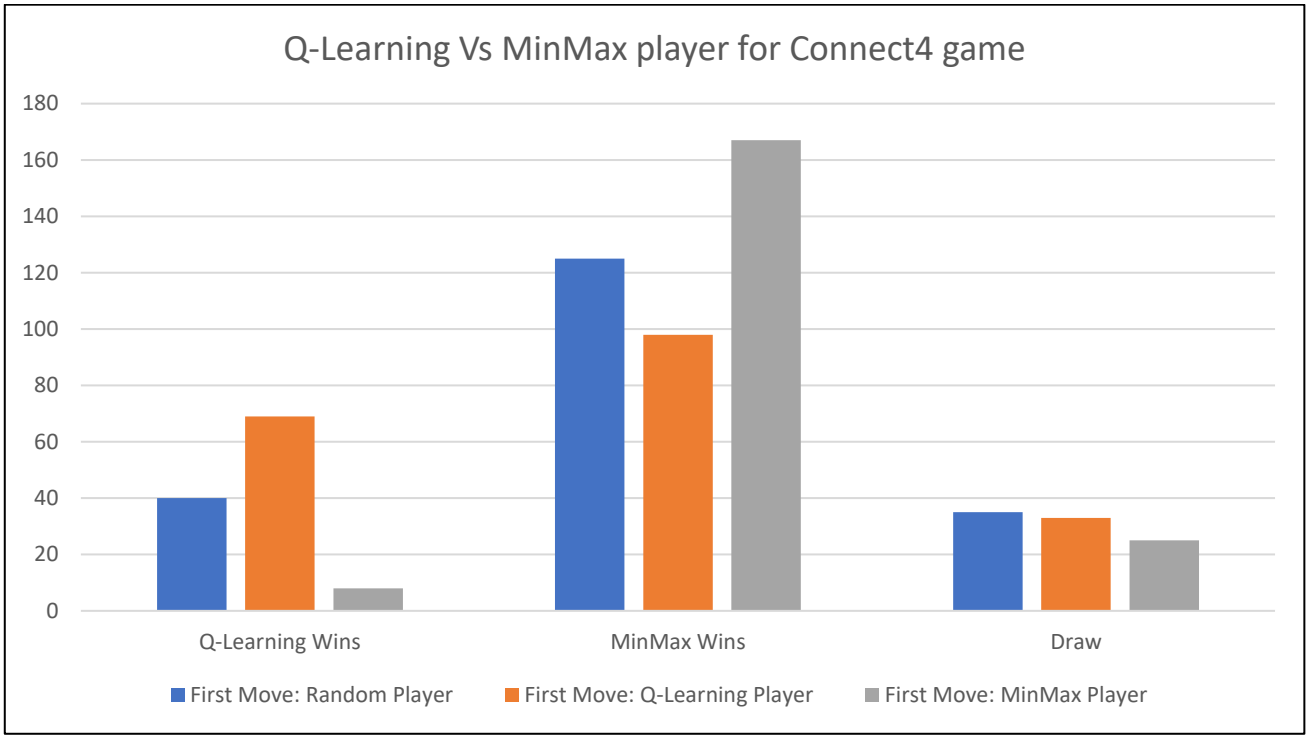

Figure 12: Graphs comparing performance of MinMax with Depth = 8 vs. Q-Learning player

From above graphs I can conclude that MinMax again plays comprehensively well against Q-Learning and wins most of the games irrespective of whether it plays first move or second move in the game.

On the other hand, Q-Learning was able to win or draw games when it played first move in the game. It could win a very few numbers of games when it played first move in the game.

Overall, I can conclude that MinMax algorithm with Alpha-Beta pruning and Depth = 8 is the ideal algorithm to play Tic-Tac-Toe game. It takes less time to train and performs better than Q-Learning. Furthermore, its performance is not significantly dependent on whether it plays first move or second move in the game.

Name: Karan Dua
Student Id: 21331391

References

i. How to Program a Connect 4 AI (implementing the minimax algorithm) - YouTube
ii. GitHub - KeithGalli/Connect4-Python: Connect 4 programmed in python using pygame
iii. GitHub - javacodingcommunity/TicTacToeAI-with-Minimax: Create a tic tac toe AI using minimax and python.
iv. Tic Tac Toe AI with MiniMax using Python | Part 1: Programming Tic Tac Toe - YouTube
v. Reinforcement Learning : Tic-Tac-Toe - YouTube
vi. Algorithms Explained – minimax and alpha-beta pruning - YouTube
vii. Alpha-beta pseudocode - Pastebin.com
viii. Minimax pseudocode - Pastebin.com

Code Execution Instructions

i. Before running this code, please download pickle file from below link**:**

https://drive.google.com/drive/folders/1pzqZonTMNfINttprf8QaAuLzS1O-cNoK?usp=sharing

ii. Unzip file code.zip
iii. Please ensure following pickle files are present in same directory as python notebooks
- Connect4QLearningModel.pickle
- TicTacToeQLearningModel.pickle

iv. Execute each of python notebooks provided in code folder implementing algorithms.
v. Please note that training Q-Learning model may take a lot of time. To avoid this, use pickle files provided for pre-trained models

Appendix: Code for: TicTacToe_QLearning

```
import random
import math
from IPython.display import display
import pandas as pd
from tqdm import tqdm
import pickle
import numpy as np
class TicTacToe_Game:

    def initialise_baord(self) :
        self.ttt_board = {
                        1: ' ', 2:' ', 3: ' ',
                        4: ' ', 5:' ', 6: ' ',
                        7: ' ', 8:' ', 9: ' '
                       }
    def display_board(self):
        print("\n")
        for row in range(3):
            for col in range(3):
                cell = row * 3 + col + 1
                print(self.ttt_board[cell], end="")
                if col < 2:
                    print(" | ", end="")
            print()
            if row < 2:
                print("---------")
        print()

    def tossForFirstMove(self):
        choices = [1,2]
        return random.choice(choices)

    def display_board(self):
        print("\n")
        print( self.ttt_board[1], '|', self.ttt_board[2], '|', self.ttt_board[3])
        print(' -+---+-')
        print(self.ttt_board[4], '|', self.ttt_board[5], '|', self.ttt_board[6])
        print(' -+---+-')
        print(self.ttt_board[7], '|', self.ttt_board[8], '|', self.ttt_board[9], "\n")

    def validateMove(self, move):
        return self.ttt_board[move] == ' '

    def validateDraw(self):
        return all(self.ttt_board[key] != ' ' for key in self.ttt_board.keys())


    def validateWin(self):
        win_combinations = [
            (1, 2, 3), (4, 5, 6), (7, 8, 9),
            (1, 4, 7), (2, 5, 8), (3, 6, 9),
            (1, 5, 9), (7, 5, 3)
        ]

        for combo in win_combinations:
            if (self.ttt_board[combo[0]] == self.ttt_board[combo[1]] == self.ttt_board[combo[2]] != ' '):
                return True

        return False

    def validateWinForLetter(self, mark):
        winning_positions = [
            (1, 2, 3), (4, 5, 6), (7, 8, 9),
            (1, 4, 7), (2, 5, 8), (3, 6, 9),
```

```
            (1, 5, 9), (7, 5, 3)
        ]
        for pos in winning_positions:
            if all(self.ttt_board[i] == mark for i in pos):
                return True
        return False

class QLearning:
    def __init__(self):
        self.epsilon = 1.0
        self.QLearningStates = {}

    getPosition = lambda self, current_board: tuple(tuple(current_board[i+j] for j in range(3)) for i in range(1, 10, 3))

    def getQLearningValue_For_Action(self, current_board, current_position):
        position = self.getPosition(current_board)
        if position not in self.QLearningStates:
            self.QLearningStates[position] = np.zeros((9,))
        return self.QLearningStates[position][current_position - 1]


    def getBestPositionFromQLearning(self, current_board, possible_positions):
        return random.choice(possible_positions) if random.random() < self.epsilon else max(possible_positions, key=lambda x:
self.getQLearningValue_For_Action(current_board, x))

    def updateQLearningModel(self, current_board, current_position, reward, successive_board, possible_positions):
        bestQValue = max([self.getQLearningValue_For_Action(successive_board, current_position) for next_action in possible_positions],
default=0)
        optimisedQVlaue = self.getQLearningValue_For_Action(current_board, current_position) + 0.1 * ((reward + 0.99 * bestQValue) -
self.getQLearningValue_For_Action(current_board, current_position))
        position = self.getPosition(current_board)
        self.QLearningStates[position][current_position - 1] = optimisedQVlaue

    def update_epsilon(self):
        self.epsilon = max(self.epsilon * 0.999, 0.1)

    def saveQLearningModel(self):
        with open("TicTacToeQLearningModel.pickle", "wb") as file:
            pickle.dump(self.QLearningStates, file)

    def loadQLearningModel(self):
        with open("TicTacToeQLearningModel.pickle", "rb") as file:
            self.QLearningStates = pickle.load(file)

    def trainQLearningModel(self):
        QLearningWin = SIAgentWin = Draw = 0
        total_episodes = 3000000
        for episode in tqdm(range(total_episodes)):
            ttt_game = TicTacToe_Game()
            ttt_game.initialise_baord()
            current_board = ttt_game.ttt_board

            while True:
                QLearningPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

                if len(QLearningPossible_Positions) == 0:
                    break

                QLearningPosition = self.getBestPositionFromQLearning(current_board, QLearningPossible_Positions)

                if ttt_game.validateMove(QLearningPosition):
                    ttt_game.ttt_board[QLearningPosition] = 'X'

                isQLearningWinner = ttt_game.validateWinForLetter('X')
                isSIAgentWinner = ttt_game.validateWinForLetter('O')
                possibleMoves = [i for i in range(1, 10) if ttt_game.validateMove(i)]
```

```
            if isQLearningWinner:
                QLearningWin += 1
                self.updateQLearningModel(current_board, QLearningPosition, 1, ttt_game.ttt_board, [])
                break

            elif isSIAgentWinner:
                SIAgentWin += 1
                self.updateQLearningModel(current_board, QLearningPosition, -1, ttt_game.ttt_board, [])
                break

            elif ttt_game.validateDraw():
                Draw += 1
                self.updateQLearningModel(current_board, QLearningPosition, 0, ttt_game.ttt_board, [])
                break
            else:
                self.updateQLearningModel(current_board, QLearningPosition, 0, ttt_game.ttt_board, possibleMoves)

            SIAgentPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]
            SIAgentPosition = SIAgentPossible_Positions[random.randint(0, len(SIAgentPossible_Positions)-1)]

            if ttt_game.validateMove(SIAgentPosition):
                ttt_game.ttt_board[SIAgentPosition] = 'O'

            isQLearningWinner = ttt_game.validateWinForLetter('X')
            isSIAgentWinner = ttt_game.validateWinForLetter('O')
            possibleMoves = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if isQLearningWinner:
                QLearningWin += 1
                self.updateQLearningModel(current_board, SIAgentPosition, 1, ttt_game.ttt_board, [])
                break

            elif isSIAgentWinner:
                SIAgentWin += 1
                self.updateQLearningModel(current_board, SIAgentPosition, -1, ttt_game.ttt_board, [])
                break

            elif ttt_game.validateDraw():
                Draw += 1
                self.updateQLearningModel(current_board, SIAgentPosition, 0, ttt_game.ttt_board, [])
                break
            else:
                self.updateQLearningModel(current_board, SIAgentPosition, 0, ttt_game.ttt_board, possibleMoves)

            current_board = ttt_game.ttt_board
        self.update_epsilon()

    return QLearningWin, SIAgentWin, Draw, total_episodes

def play_tic_tac_toe(self, SIAgent_plays_first, ttt_game):
    SI_Agent_Letter = 'O'
    QLearning_Letter = 'X'

    while True:
        if SIAgent_plays_first:
            SIAgentPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(SIAgentPossible_Positions) == 0:
                return "Draw"

            SIAgentPosition = SIAgentPossible_Positions[random.randint(0, len(SIAgentPossible_Positions)-1)]

            if ttt_game.validateMove(SIAgentPosition):
                ttt_game.ttt_board[SIAgentPosition] = SI_Agent_Letter
```

```
            if ttt_game.validateWinForLetter(SI_Agent_Letter) :
                return "SIAgentWon"

            if ttt_game.validateDraw():
                return "Draw"

            QLearningPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(QLearningPossible_Positions) == 0:
                "Draw"

            QLearningPosition = self.getBestPositionFromQLearning(ttt_game.ttt_board, QLearningPossible_Positions)

            if ttt_game.validateMove(QLearningPosition):
                ttt_game.ttt_board[QLearningPosition] = QLearning_Letter

            if ttt_game.validateWinForLetter(QLearning_Letter) :
                return "QLearningWon"

            if ttt_game.validateDraw():
                return "Draw"

        else:
            QLearningPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(QLearningPossible_Positions) == 0:
                break

            QLearningPosition = self.getBestPositionFromQLearning(ttt_game.ttt_board, QLearningPossible_Positions)

            if ttt_game.validateMove(QLearningPosition):
                ttt_game.ttt_board[QLearningPosition] = QLearning_Letter

            if ttt_game.validateWinForLetter(QLearning_Letter) :
                return "QLearningWon"

            if ttt_game.validateDraw():
                return "Draw"


            SIAgentPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(SIAgentPossible_Positions) == 0:
                return "Draw"

            SIAgentPosition = SIAgentPossible_Positions[random.randint(0, len(SIAgentPossible_Positions)-1)]

            if ttt_game.validateMove(SIAgentPosition):
                ttt_game.ttt_board[SIAgentPosition] = SI_Agent_Letter

            if ttt_game.validateWinForLetter(SI_Agent_Letter) :
                return "SIAgentWon"

            if ttt_game.validateDraw():
                return "Draw"

qLearning = QLearning()

QLearningWin, SIAgentWin, Draw, total_episodes = qLearning.trainQLearningModel()

qLearning.saveQLearningModel()

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games Qlearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'Training'
```

```
statistics_dict['Total Number of Games'] = total_episodes
statistics_dict['Number of Games Qlearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw

statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 2000
SIAgentWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    ttt_game = TicTacToe_Game()
    ttt_game.initialise_baord()

    SIAgent_plays_first = False
    if ttt_game.tossForFirstMove() == 1 :
        SIAgent_plays_first = True
    else :
        SIAgent_plays_first = False

    winner = qLearningPlayer.play_tic_tac_toe(SIAgent_plays_first, ttt_game)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Random'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 2000
SIAgentWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    ttt_game = TicTacToe_Game()
    ttt_game.initialise_baord()

    SIAgent_plays_first = True

    winner = qLearningPlayer.play_tic_tac_toe(SIAgent_plays_first, ttt_game)

    if winner == 'QLearningWon':
        QLearningWin += 1
```

```
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: First Move: Semi Intelligent Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 2000
SIAgentWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    ttt_game = TicTacToe_Game()
    ttt_game.initialise_baord()

    SIAgent_plays_first = False

    winner = qLearningPlayer.play_tic_tac_toe(SIAgent_plays_first, ttt_game)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Q-Learning Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)
```

Appendix: Code for: TicTacToe_MinMax

```
import random
import math
from IPython.display import display
import pandas as pd
from tqdm import tqdm
import time

class TicTacToe_MinMax:

    def initialise_baord_set_letter(self) :
        self.ttt_board = {
                        1: ' ', 2:' ', 3: ' ',
                        4: ' ', 5:' ', 6: ' ',
                        7: ' ', 8:' ', 9: ' '
                       }
        self.SI_Agent_Letter = 'X'
        self.MinMax_Letter = 'O'

    def display_board(self):
        print("\n")
        for row in range(3):
            for col in range(3):
                cell = row * 3 + col + 1
                print(self.ttt_board[cell], end="")
                if col < 2:
                    print(" | ", end="")
            print()
            if row < 2:
                print("---------")
        print()

    def tossForFirstMove(self):
        choices = [1,2]
        return random.choice(choices)

    def validateMove(self, move):
        return self.ttt_board[move] == ' '

    def validateDraw(self):
        return all(self.ttt_board[key] != ' ' for key in self.ttt_board.keys())


    def validateWin(self):
        win_combinations = [
            (1, 2, 3), (4, 5, 6), (7, 8, 9),
            (1, 4, 7), (2, 5, 8), (3, 6, 9),
            (1, 5, 9), (7, 5, 3)
        ]

        for combo in win_combinations:
            if (self.ttt_board[combo[0]] == self.ttt_board[combo[1]] == self.ttt_board[combo[2]] != ' '):
                return True

        return False

    def validateWinForLetter(self, mark):
        winning_positions = [
            (1, 2, 3), (4, 5, 6), (7, 8, 9),
            (1, 4, 7), (2, 5, 8), (3, 6, 9),
            (1, 5, 9), (7, 5, 3)
        ]
        for pos in winning_positions:
            if all(self.ttt_board[i] == mark for i in pos):
                return True
```

```
        return False

    def get_random_generated_move(self):
        position = random.randint(1, 9)
        if self.validateMove(position):
            return position
        else:
            position = self.get_random_generated_move()
            return position

    def play_tic_tac_toe_with_alpha_beta_pruning(self, SIAgent_plays_first):
        while True:
            if SIAgent_plays_first:
                self.Semi_Intelligent_Agent_Move()

                if self.validateWinForLetter(self.SI_Agent_Letter) :
                    return "SIAgentWon"

                if self.validateDraw():
                    return "Draw"

                self.Min_Max_Move_with_alpha_beta_pruning()

                if self.validateWinForLetter(self.MinMax_Letter) :
                    return "MinMaxWon"


            else:

                self.Min_Max_Move_with_alpha_beta_pruning()

                if self.validateWinForLetter(self.MinMax_Letter) :
                    return "MinMaxWon"

                self.Semi_Intelligent_Agent_Move()

                if self.validateWinForLetter(self.SI_Agent_Letter) :
                    return "SIAgentWon"

                if self.validateDraw():
                    return "Draw"



    def Semi_Intelligent_Agent_Move(self) :
        for possible_position in self.ttt_board.keys():
            if self.ttt_board[possible_position] == ' ':

                self.ttt_board[possible_position] = self.SI_Agent_Letter

                if self.validateWin() :
                    self.ttt_board[possible_position] = ' '
                    position = possible_position
                    break

                elif self.validateDraw():
                    self.ttt_board[possible_position] = ' '
                    position = possible_position
                    break

                else:
                    self.ttt_board[possible_position] = ' '
                    position = self.get_random_generated_move()

        self.ttt_board[position] = self.SI_Agent_Letter
        return
```

```
def Min_Max_Move_with_alpha_beta_pruning(self):
    optimised_score =  -math.inf
    optimised_position = self.get_random_generated_move()

    for possible_position in self.ttt_board.keys() :

        if self.ttt_board[possible_position] == ' ' :
            self.ttt_board[possible_position] = self.MinMax_Letter
            current_score = self.evaluate_MinMax_score_with_alpha_beta_pruning(False, -math.inf, math.inf)
            self.ttt_board[possible_position] = ' '

            if current_score > optimised_score :
                optimised_score = current_score
                optimised_position = possible_position

    self.ttt_board[optimised_position] = self.MinMax_Letter
    return


def evaluate_MinMax_score_with_alpha_beta_pruning(self, isMinMaxMove, alpha, beta):
    if self.validateWinForLetter(self.MinMax_Letter) :
        return 1
    elif self.validateWinForLetter(self.SI_Agent_Letter) :
        return -1
    elif self.validateDraw() :
        return 0

    if isMinMaxMove :
        optimisedScore = -math.inf

        for possible_position in self.ttt_board.keys():

            if self.ttt_board[possible_position] == ' ' :
                self.ttt_board[possible_position] = self.MinMax_Letter
                current_score = self.evaluate_MinMax_score_with_alpha_beta_pruning(False, alpha, beta)
                self.ttt_board[possible_position] = ' '

                optimisedScore = max(optimisedScore, current_score)
                alpha = max(alpha, optimisedScore)

                if alpha >= beta :
                    break

        return optimisedScore

    else:
        optimisedScore = math.inf
        for possible_position in self.ttt_board.keys():
            if self.ttt_board[possible_position] == ' ':
                self.ttt_board[possible_position] = self.SI_Agent_Letter
                current_score = self.evaluate_MinMax_score_with_alpha_beta_pruning(True, alpha, beta)
                self.ttt_board[possible_position] = ' '

                optimisedScore = min(optimisedScore, current_score)
                beta = min(beta, optimisedScore)

                if alpha >= beta :
                    break

        return optimisedScore

def play_tic_tac_toe(self, SIAgent_plays_first):
    while True:
        if SIAgent_plays_first:
            self.Semi_Intelligent_Agent_Move()
```

```
            if self.validateWinForLetter(self.SI_Agent_Letter) :
                return "SIAgentWon"

            if self.validateDraw():
                return "Draw"

            self.Min_Max_Move()

            if self.validateWinForLetter(self.MinMax_Letter) :
                return "MinMaxWon"


        else:

            self.Min_Max_Move()

            if self.validateWinForLetter(self.MinMax_Letter) :
                return "MinMaxWon"

            self.Semi_Intelligent_Agent_Move()

            if self.validateWinForLetter(self.SI_Agent_Letter) :
                return "SIAgentWon"

            if self.validateDraw():
                return "Draw"

    def Min_Max_Move(self):
        optimised_score =  -math.inf
        optimised_position = self.get_random_generated_move()

        for possible_position in self.ttt_board.keys() :

            if self.ttt_board[possible_position] == ' ' :
                self.ttt_board[possible_position] = self.MinMax_Letter
                current_score = self.evaluate_MinMax_score(False)
                self.ttt_board[possible_position] = ' '

                if current_score > optimised_score :
                    optimised_score = current_score
                    optimised_position = possible_position

        self.ttt_board[optimised_position] = self.MinMax_Letter
        return

    def evaluate_MinMax_score(self, isMinMaxMove):
        if self.validateWinForLetter(self.MinMax_Letter) :
            return 1
        elif self.validateWinForLetter(self.SI_Agent_Letter) :
            return -1
        elif self.validateDraw() :
            return 0

        if isMinMaxMove :
            optimisedScore = -math.inf

            for possible_position in self.ttt_board.keys():

                if self.ttt_board[possible_position] == ' ' :
                    self.ttt_board[possible_position] = self.MinMax_Letter
                    current_score = self.evaluate_MinMax_score(False)
                    self.ttt_board[possible_position] = ' '

                    optimisedScore = max(optimisedScore, current_score)
```

```
            return optimisedScore

        else:
            optimisedScore = math.inf
            for possible_position in self.ttt_board.keys():
                if self.ttt_board[possible_position] == ' ':
                    self.ttt_board[possible_position] = self.SI_Agent_Letter
                    current_score = self.evaluate_MinMax_score(True)
                    self.ttt_board[possible_position] = ' '

                    optimisedScore = min(optimisedScore, current_score)

            return optimisedScore

games = 100
SIAgentWin = MinMaxWin = Draw = 0

startTime = time.time()
for _ in tqdm(range(games)):
    ttt_min_max = TicTacToe_MinMax()
    ttt_min_max.initialise_baord_set_letter()

    SIAgent_plays_first = False
    if ttt_min_max.tossForFirstMove() == 1 :
        SIAgent_plays_first = True
    else :
        SIAgent_plays_first = False
    try:
        winner = ttt_min_max.play_tic_tac_toe(SIAgent_plays_first)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1
totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Time taken without Alpha Beta Pruning (in seconds)', 'Total Number of Games',
'Number of Games MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Random'
statistics_dict['Time taken without Alpha Beta Pruning (in seconds)'] = totalTime
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 100
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

startTime = time.time()
for _ in tqdm(range(games)):
    ttt_min_max = TicTacToe_MinMax()
    ttt_min_max.initialise_baord_set_letter()

    SIAgent_plays_first = False

    try:
```

```
        winner = ttt_min_max.play_tic_tac_toe(SIAgent_plays_first)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Time taken without Alpha Beta Pruning (in seconds)', 'Total Number of Games',
'Number of Games MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: MinMax Player'
statistics_dict['Time taken without Alpha Beta Pruning (in seconds)'] = totalTime
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 100
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

startTime = time.time()
for _ in tqdm(range(games)):
    ttt_min_max = TicTacToe_MinMax()
    ttt_min_max.initialise_baord_set_letter()

    SIAgent_plays_first = True

    try:
        winner = ttt_min_max.play_tic_tac_toe(SIAgent_plays_first)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Time taken without Alpha Beta Pruning (in seconds)', 'Total Number of Games',
'Number of Games MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Semi Intelligent Player'
statistics_dict['Time taken without Alpha Beta Pruning (in seconds)'] = totalTime
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 1000
```

```
SIAgentWin = MinMaxWin = Draw = 0

startTime = time.time()
for _ in tqdm(range(games)):
    ttt_min_max = TicTacToe_MinMax()
    ttt_min_max.initialise_baord_set_letter()

    SIAgent_plays_first = False
    if ttt_min_max.tossForFirstMove() == 1 :
        SIAgent_plays_first = True
    else :
        SIAgent_plays_first = False

    try:
        winner = ttt_min_max.play_tic_tac_toe_with_alpha_beta_pruning(SIAgent_plays_first)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1
totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Time taken with Alpha Beta Pruning (in seconds)', 'Total Number of Games', 'Number
of Games MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Random Player'
statistics_dict['Time taken with Alpha Beta Pruning (in seconds)'] = totalTime
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 1000
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

startTime = time.time()
for _ in tqdm(range(games)):
    ttt_min_max = TicTacToe_MinMax()
    ttt_min_max.initialise_baord_set_letter()

    SIAgent_plays_first = False

    try:
        winner = ttt_min_max.play_tic_tac_toe_with_alpha_beta_pruning(SIAgent_plays_first)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime
```

```
statistics_df = pd.DataFrame(columns=['Game Type', 'Time taken with Alpha Beta Pruning (in seconds)', 'Total Number of Games', 'Number of Games MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: MinMax Player'
statistics_dict['Time taken with Alpha Beta Pruning (in seconds)'] = totalTime
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 1000
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

startTime = time.time()
for _ in tqdm(range(games)):
    ttt_min_max = TicTacToe_MinMax()
    ttt_min_max.initialise_baord_set_letter()

    SIAgent_plays_first = True
    try:
        winner = ttt_min_max.play_tic_tac_toe_with_alpha_beta_pruning(SIAgent_plays_first)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Time taken with Alpha Beta Pruning (in seconds)', 'Total Number of Games', 'Number of Games MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Semi Intelligent Player'
statistics_dict['Time taken with Alpha Beta Pruning (in seconds)'] = totalTime
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)
```

Name: Karan Dua
Student Id: 21331391

Appendix: Code for: TicTacToe_QLearningVsMinMax

```
import random
import math
from IPython.display import display
import pandas as pd
from tqdm import tqdm
import pickle
import numpy as np

class TicTacToe_Game:

    def initialise_baord(self) :
        self.ttt_board = {
                        1: ' ', 2:' ', 3: ' ',
                        4: ' ', 5:' ', 6: ' ',
                        7: ' ', 8:' ', 9: ' '
                       }
    def display_board(self):
        print("\n")
        for row in range(3):
            for col in range(3):
                cell = row * 3 + col + 1
                print(self.ttt_board[cell], end="")
                if col < 2:
                    print(" | ", end="")
            print()
            if row < 2:
                print("---------")
        print()


    def tossForFirstMove(self):
        choices = [1,2]
        return random.choice(choices)

    def get_random_generated_move(self):
        position = random.randint(1, 9)
        if self.validateMove(position):
            return position
        else:
            position = self.get_random_generated_move()
            return position

    def display_board(self):
        print("\n")
        print( self.ttt_board[1], '|', self.ttt_board[2], '|', self.ttt_board[3])
        print(' -+---+-')
        print(self.ttt_board[4], '|', self.ttt_board[5], '|', self.ttt_board[6])
        print(' -+---+-')
        print(self.ttt_board[7], '|', self.ttt_board[8], '|', self.ttt_board[9], "\n")

    def validateMove(self, move):
        return self.ttt_board[move] == ' '

    def validateDraw(self):
        return all(self.ttt_board[key] != ' ' for key in self.ttt_board.keys())


    def validateWin(self):
        win_combinations = [
            (1, 2, 3), (4, 5, 6), (7, 8, 9),
            (1, 4, 7), (2, 5, 8), (3, 6, 9),
            (1, 5, 9), (7, 5, 3)
        ]
```

```
        for combo in win_combinations:
            if (self.ttt_board[combo[0]] == self.ttt_board[combo[1]] == self.ttt_board[combo[2]] != ' '):
                return True

        return False

    def validateWinForLetter(self, mark):
        winning_positions = [
            (1, 2, 3), (4, 5, 6), (7, 8, 9),
            (1, 4, 7), (2, 5, 8), (3, 6, 9),
            (1, 5, 9), (7, 5, 3)
        ]
        for pos in winning_positions:
            if all(self.ttt_board[i] == mark for i in pos):
                return True
        return False

class QLearning:
    def __init__(self):
        self.epsilon = 1.0
        self.QLearningStates = {}

    getPosition = lambda self, current_board: tuple(tuple(current_board[i+j] for j in range(3)) for i in range(1, 10, 3))

    def getQLearningValue_For_Action(self, current_board, current_position):
        position = self.getPosition(current_board)
        if position not in self.QLearningStates:
            self.QLearningStates[position] = np.zeros((9,))
        return self.QLearningStates[position][current_position - 1]


    def getBestPositionFromQLearning(self, current_board, possible_positions):
        return random.choice(possible_positions) if random.random() < self.epsilon else max(possible_positions, key=lambda x:
self.getQLearningValue_For_Action(current_board, x))

    def loadQLearningModel(self):
        with open("TicTacToeQLearningModel.pickle", "rb") as file:
            self.QLearningStates = pickle.load(file)

class TicTacToe_MinMax:

    def Min_Max_Move_with_alpha_beta_pruning(self, ttt_game, MinMax_Letter, QLearing_Letter):
        optimised_score =  -math.inf
        optimised_position = ttt_game.get_random_generated_move()

        for possible_position in ttt_game.ttt_board.keys() :

            if ttt_game.ttt_board[possible_position] == ' ' :
                ttt_game.ttt_board[possible_position] = MinMax_Letter
                current_score = self.evaluate_MinMax_score_with_alpha_beta_pruning(ttt_game, MinMax_Letter, QLearing_Letter, False, -
math.inf, math.inf)
                ttt_game.ttt_board[possible_position] = ' '

                if current_score > optimised_score :
                    optimised_score = current_score
                    optimised_position = possible_position

        return optimised_position


    def evaluate_MinMax_score_with_alpha_beta_pruning(self, ttt_game, MinMax_Letter, QLearing_Letter, isMinMaxMove, alpha, beta):
        if ttt_game.validateWinForLetter(MinMax_Letter) :
            return 1
        elif ttt_game.validateWinForLetter(QLearing_Letter) :
            return -1
        elif ttt_game.validateDraw() :
```

```
        return 0

    if isMinMaxMove :
        optimisedScore = -math.inf

        for possible_position in ttt_game.ttt_board.keys():

            if ttt_game.ttt_board[possible_position] == ' ' :
                ttt_game.ttt_board[possible_position] = MinMax_Letter
                current_score = self.evaluate_MinMax_score_with_alpha_beta_pruning(ttt_game, MinMax_Letter, QLearing_Letter, False,
alpha, beta)
                ttt_game.ttt_board[possible_position] = ' '

                optimisedScore = max(optimisedScore, current_score)
                alpha = max(alpha, optimisedScore)

                if alpha >= beta :
                    break

        return optimisedScore

    else:
        optimisedScore = math.inf

        for possible_position in ttt_game.ttt_board.keys():
            if ttt_game.ttt_board[possible_position] == ' ':
                ttt_game.ttt_board[possible_position] = QLearing_Letter
                current_score = self.evaluate_MinMax_score_with_alpha_beta_pruning(ttt_game, MinMax_Letter, QLearing_Letter, True,
alpha, beta)
                ttt_game.ttt_board[possible_position] = ' '

                optimisedScore = min(optimisedScore, current_score)
                beta = min(beta, optimisedScore)

                if alpha >= beta :
                    break

        return optimisedScore

def play_tic_tac_toe(MinMaxPlaysFirst, QLearning, MinMax, ttt_game):
    MinMaxLetter = 'O'
    QLearning_Letter = 'X'

    while True:
        if MinMaxPlaysFirst:

            MinMaxPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(MinMaxPossible_Positions) == 0:
                return "Draw"

            MinMaxPosition = MinMax.Min_Max_Move_with_alpha_beta_pruning(ttt_game, MinMaxLetter, QLearning_Letter)

            if ttt_game.validateMove(MinMaxPosition):
                ttt_game.ttt_board[MinMaxPosition] = MinMaxLetter

            if ttt_game.validateWinForLetter(MinMaxLetter) :
                return "MinMaxWon"

            if ttt_game.validateDraw():
                return "Draw"

            QLearningPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(QLearningPossible_Positions) == 0:
                break
```

```
            QLearningPosition = QLearning.getBestPositionFromQLearning(ttt_game.ttt_board, QLearningPossible_Positions)

            if ttt_game.validateMove(QLearningPosition):
                ttt_game.ttt_board[QLearningPosition] = QLearning_Letter

            if ttt_game.validateWinForLetter(QLearning_Letter) :
                return "QLearningWon"

            if ttt_game.validateDraw():
                return "Draw"

        else:
            QLearningPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(QLearningPossible_Positions) == 0:
                break

            QLearningPosition = QLearning.getBestPositionFromQLearning(ttt_game.ttt_board, QLearningPossible_Positions)

            if ttt_game.validateMove(QLearningPosition):
                ttt_game.ttt_board[QLearningPosition] = QLearning_Letter

            if ttt_game.validateWinForLetter(QLearning_Letter) :
                return "QLearningWon"

            if ttt_game.validateDraw():
                return "Draw"


            MinMaxPossible_Positions = [i for i in range(1, 10) if ttt_game.validateMove(i)]

            if len(MinMaxPossible_Positions) == 0:
                return "Draw"

            MinMaxPosition = MinMax.Min_Max_Move_with_alpha_beta_pruning(ttt_game, MinMaxLetter, QLearning_Letter)

            if ttt_game.validateMove(MinMaxPosition):
                ttt_game.ttt_board[MinMaxPosition] = MinMaxLetter

            if ttt_game.validateWinForLetter(MinMaxLetter) :
                return "MinMaxWon"

            if ttt_game.validateDraw():
                return "Draw"

games = 2000
MinMaxWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

ttt_min_max = TicTacToe_MinMax()
print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    ttt_game = TicTacToe_Game()
    ttt_game.initialise_baord()

    MinMaxPlaysFirst = False
    if ttt_game.tossForFirstMove() == 1 :
        MinMaxPlaysFirst = True
    else :
        MinMaxPlaysFirst = False
```

```
    winner = play_tic_tac_toe(MinMaxPlaysFirst, qLearningPlayer, ttt_min_max, ttt_game)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'MinMaxWon':
        MinMaxWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
MinMax Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Random '
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 2000
MinMaxWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

ttt_min_max = TicTacToe_MinMax()
print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    ttt_game = TicTacToe_Game()
    ttt_game.initialise_baord()

    MinMaxPlaysFirst = False

    winner = play_tic_tac_toe(MinMaxPlaysFirst, qLearningPlayer, ttt_min_max, ttt_game)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'MinMaxWon':
        MinMaxWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
MinMax Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Q-Learning Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 2000
MinMaxWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

ttt_min_max = TicTacToe_MinMax()
print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")
```

```
for _ in tqdm(range(games)):
    ttt_game = TicTacToe_Game()
    ttt_game.initialise_baord()

    MinMaxPlaysFirst = True

    winner = play_tic_tac_toe(MinMaxPlaysFirst, qLearningPlayer, ttt_min_max, ttt_game)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'MinMaxWon':
        MinMaxWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
MinMax Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Min-Max Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)
```

Appendix: Code for: Connect4_QLearning

```
import random
import math
from IPython.display import display
import pandas as pd
from tqdm import tqdm
import pickle
import numpy as np
import time

class Connect4_Game:

   def initialise_board(self) :
      self.rows = 6
      self.columns = 7
      self.connect4_board = np.zeros((self.rows, self.columns))

   validateMove = lambda self, column: self.connect4_board[len(self.connect4_board)-1][column] == 0

   getNextAvailableRow = lambda self, column: next((row for row in range(len(self.connect4_board)) if self.connect4_board[row][column]
== 0), None)

   getValidMove = lambda self: [column for column in range(self.columns) if self.validateMove(column)]

   def getNextAvailablePosition(self, letter):
      rows, cols = self.rows, self.columns
      for row, row_vals in enumerate(self.connect4_board):
         for col, col_val in enumerate(row_vals[:-3]):
            if all(elem == letter for elem in row_vals[col:col+4]):
               return row, col
         for col, col_vals in zip(range(cols), (self.connect4_board[r][col] for r in range(row, min(row+4, rows)))):
            if all(elem == letter for elem in col_vals):
               return row, col
         for col, col_vals in enumerate(row_vals[:-3]):
            if row < rows-3 and col < cols-3:
               diag_vals = [self.connect4_board[row+i][col+i] for i in range(4)]
               if all(elem == letter for elem in diag_vals):
                  return row, col
         for col, col_vals in enumerate(row_vals[:-3]):
            if row >= 3 and col < cols-3:
               diag_vals = [self.connect4_board[row-i][col+i] for i in range(4)]
               if all(elem == letter for elem in diag_vals):
                  return row, col
      else:
         return -1, -1



   def validateWin(self, letter):
      for row in range(self.rows):
         for col in range(self.columns - 3):
            if all(self.connect4_board[row][col + i] == letter for i in range(4)):
               return True

      for row in range(self.rows - 3):
         for col in range(self.columns):
            if all(self.connect4_board[row + i][col] == letter for i in range(4)):
               return True

      for row in range(self.rows - 3):
         for col in range(self.columns - 3):
            if all(self.connect4_board[row + i][col + i] == letter for i in range(4)):
               return True

      for row in range(3, self.rows):
```

```
        for col in range(self.columns - 3):
            if all(self.connect4_board[row - i][col + i] == letter for i in range(4)):
                return True

        return False



    def tossForFirstMove(self):
        choices = [1,2]
        return random.choice(choices)

    def validateFinalMove(self, SI_Agent_Letter, MinMax_Letter):
        return any(self.validateWin(letter) for letter in (SI_Agent_Letter, MinMax_Letter)) or not self.getValidMove()

class SI_Agent :

    def Semi_Intelligent_Agent_Move(self, c4_game, SIAgentLetter, MinMaxLetter) :
        if c4_game.validateFinalMove(SIAgentLetter, MinMaxLetter):
            siagent_row, siagent_col = c4_game.getNextAvailablePosotion(SIAgentLetter)
            if siagent_row != -1:
                return siagent_row, siagent_col
            else:
                minmax_row, minmax_col = c4_game.getNextAvailablePosotion(MinMaxLetter)
                if minmax_row != -1:
                    return minmax_row, minmax_col
                else:
                    possible_positions = c4_game.getValidMove()
                    random_row = c4_game.getNextAvailableRow(random.choice(possible_positions))
                    random_col = random.choice(possible_positions)
                    return random_row, random_col
        else:
            possible_positions = c4_game.getValidMove()
            random_row = c4_game.getNextAvailableRow(random.choice(possible_positions))
            random_col = random.choice(possible_positions)

            return random_row, random_col

class QLearning:
    def __init__(self):
        self.epsilon = 1.0
        self.QLearningStates = {}

    getPosition = lambda self, positions: int(''.join([str(int(position)) for position in positions.flatten()]))

    def getQLearningValue_For_Action(self, current_board, current_position):
        position = self.getPosition(current_board)
        if position not in self.QLearningStates:
            self.QLearningStates[(position, current_position)] = 0
        return self.QLearningStates[(position, current_position)]

    def getBestPositionFromQLearning(self, current_board, possible_positions):
        return random.choice(possible_positions) if random.random() < self.epsilon else
max([(self.getQLearningValue_For_Action(current_board, position), position) for position in possible_positions], key=lambda x: x[0])[1]

    def updateQLearningModel(self, current_board, current_position, reward, successive_board, possible_positions):
        bestQValue = max([self.getQLearningValue_For_Action(successive_board, next_position) for next_position in possible_positions],
default=0)
        optimisedQValue = self.getQLearningValue_For_Action(current_board, current_position) + 0.1 * ((reward + 0.99 * bestQValue) -
self.getQLearningValue_For_Action(current_board, current_position))
        position = self.getPosition(current_board)
        self.QLearningStates[(position, current_position)] = optimisedQValue

    def update_epsilon(self):
        self.epsilon = max(self.epsilon * 0.999, 0.1)
```

```
def saveQLearningModel(self):
    with open("Connect4QLearningModel.pickle", "wb") as file:
        pickle.dump(self.QLearningStates, file)

def loadQLearningModel(self):
    with open("Connect4QLearningModel.pickle", "rb") as file:
        self.QLearningStates = pickle.load(file)

def trainQLearningModel(self):
    QLearningWin = SIAgentWin = Draw = 0
    QLearningLetter = 1
    SIAgentLetter = 2
    total_episodes = 3000000
    si_agent = SI_Agent()

    for episode in tqdm(range(total_episodes)):
        c4Game = Connect4_Game()
        c4Game.initialise_board()
        current_board = c4Game.connect4_board

        while True:

            QLearningPossible_Positions = c4Game.getValidMove()

            if len(QLearningPossible_Positions) == 0:
                break

            QLearning_chosen_column = self.getBestPositionFromQLearning(current_board, QLearningPossible_Positions)
            QLearning_chosen_row = c4Game.getNextAvailableRow(QLearning_chosen_column)
            c4Game.connect4_board[QLearning_chosen_row][QLearning_chosen_column] = QLearningLetter

            possibleMoves = c4Game.getValidMove()

            if c4Game.validateWin(QLearningLetter):
                QLearningWin += 1
                self.updateQLearningModel(current_board, QLearning_chosen_column, 1, c4Game.connect4_board, [])
                break

            elif c4Game.validateWin(SIAgentLetter):
                SIAgentWin += 1
                self.updateQLearningModel(current_board, QLearning_chosen_column, -1, c4Game.connect4_board, [])
                break

            elif len(possibleMoves) == 0:
                Draw += 1
                self.updateQLearningModel(current_board, QLearning_chosen_column, 0, c4Game.connect4_board, [])
                break

            else:
                self.updateQLearningModel(current_board, QLearning_chosen_column, 0, c4Game.connect4_board, possibleMoves)


            SIAgent_chosen_row, SIAgent_chosen_column = si_agent.Semi_Intelligent_Agent_Move(c4Game, SIAgentLetter, QLearningLetter)
            c4Game.connect4_board[SIAgent_chosen_row][SIAgent_chosen_column] = SIAgentLetter

            possibleMoves = c4Game.getValidMove()

            if c4Game.validateWin(QLearningLetter):
                QLearningWin += 1
                self.updateQLearningModel(current_board, SIAgent_chosen_column, 1, c4Game.connect4_board, [])
                break

            elif c4Game.validateWin(SIAgentLetter):
                SIAgentWin += 1
                self.updateQLearningModel(current_board, SIAgent_chosen_column, -1, c4Game.connect4_board, [])
                break
```

```
            elif len(possibleMoves) == 0:
                Draw += 1
                self.updateQLearningModel(current_board, SIAgent_chosen_column, 0, c4Game.connect4_board, [])
                break

            else:
                self.updateQLearningModel(current_board, SIAgent_chosen_column, 0, c4Game.connect4_board, possibleMoves)

            current_board = c4Game.connect4_board
        self.update_epsilon()

    return QLearningWin, SIAgentWin, Draw, total_episodes

qLearning = QLearning()

QLearningWin, SIAgentWin, Draw, total_episodes = qLearning.trainQLearningModel()

qLearning.saveQLearningModel()

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games Qlearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'Training'
statistics_dict['Total Number of Games'] = total_episodes
statistics_dict['Number of Games Qlearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw

statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

def play_connect4(SIAgent_plays_first, c4Game, si_agent, qLearningPlayer):
    QLearningLetter = 1
    SIAgentLetter = 2

    while True:
        if SIAgent_plays_first:

            SIAgentPossible_Positions = c4Game.getValidMove()

            if len(SIAgentPossible_Positions) == 0:
                return "Draw"

            SIAgent_chosen_row, SIAgent_chosen_column = si_agent.Semi_Intelligent_Agent_Move(c4Game, SIAgentLetter, QLearningLetter)
            c4Game.connect4_board[SIAgent_chosen_row][SIAgent_chosen_column] = SIAgentLetter

            if c4Game.validateWin(SIAgentLetter) :
                return "SIAgentWon"

            if c4Game.validateWin(QLearningLetter):
                return "QLearningWon"

            if len(c4Game.getValidMove()) == 0 :
                return "Draw"

            QLearningPossible_Positions = c4Game.getValidMove()

            if len(QLearningPossible_Positions) == 0:
                return "Draw"

            QLearning_chosen_column = qLearningPlayer.getBestPositionFromQLearning(c4Game.connect4_board,
QLearningPossible_Positions)
            QLearning_chosen_row = c4Game.getNextAvailableRow(QLearning_chosen_column)
            c4Game.connect4_board[QLearning_chosen_row][QLearning_chosen_column] = QLearningLetter
```

```
        if c4Game.validateWin(SIAgentLetter) :
            return "SIAgentWon"

        if c4Game.validateWin(QLearningLetter):
            return "QLearningWon"

        if len(c4Game.getValidMove()) == 0 :
            return "Draw"

    else:
        QLearningPossible_Positions = c4Game.getValidMove()

        if len(QLearningPossible_Positions) == 0:
            return "Draw"

        QLearning_chosen_column = qLearningPlayer.getBestPositionFromQLearning(c4Game.connect4_board,
QLearningPossible_Positions)
        QLearning_chosen_row = c4Game.getNextAvailableRow(QLearning_chosen_column)
        c4Game.connect4_board[QLearning_chosen_row][QLearning_chosen_column] = QLearningLetter

        if c4Game.validateWin(SIAgentLetter) :
            return "SIAgentWon"

        if c4Game.validateWin(QLearningLetter):
            return "QLearningWon"

        if len(c4Game.getValidMove()) == 0 :
            return "Draw"


        SIAgentPossible_Positions = c4Game.getValidMove()

        if len(SIAgentPossible_Positions) == 0:
            return "Draw"

        SIAgent_chosen_row, SIAgent_chosen_column = si_agent.Semi_Intelligent_Agent_Move(c4Game, SIAgentLetter, QLearningLetter)
        c4Game.connect4_board[SIAgent_chosen_row][SIAgent_chosen_column] = SIAgentLetter

        if c4Game.validateWin(SIAgentLetter) :
            return "SIAgentWon"

        if c4Game.validateWin(QLearningLetter):
            return "QLearningWon"

        if len(c4Game.getValidMove()) == 0 :
            return "Draw"

games = 2000
SIAgentWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

si_agent = SI_Agent()


print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    c4Game = Connect4_Game()
    c4Game.initialise_board()

    SIAgent_plays_first = False
    if c4Game.tossForFirstMove() == 1 :
        SIAgent_plays_first = True
```

```
    else :
        SIAgent_plays_first = False

    winner = play_connect4(SIAgent_plays_first, c4Game, si_agent, qLearningPlayer)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Random'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 2000
SIAgentWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

si_agent = SI_Agent()

print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    c4Game = Connect4_Game()
    c4Game.initialise_board()

    SIAgent_plays_first = True

    winner = play_connect4(SIAgent_plays_first, c4Game, si_agent, qLearningPlayer)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: First Move: Semi Intelligent Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 2000
SIAgentWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()
```

```
si_agent = SI_Agent()

print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    c4Game = Connect4_Game()
    c4Game.initialise_board()

    SIAgent_plays_first = False

    winner = play_connect4(SIAgent_plays_first, c4Game, si_agent, qLearningPlayer)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Q-Learning Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)
```

Appendix: Code for: Connect4_MinMax

```
import random
import math
from IPython.display import display
import pandas as pd
from tqdm import tqdm
import pickle
import numpy as np
import time

class Connect4_Game:

    def initialise_board(self) :
        self.rows = 6
        self.columns = 7
        self.connect4_board = np.zeros((self.rows, self.columns))

    validateMove = lambda self, column: self.connect4_board[len(self.connect4_board)-1][column] == 0

    getNextAvailableRow = lambda self, column: next((row for row in range(len(self.connect4_board)) if self.connect4_board[row][column]
== 0), None)

    getValidMove = lambda self: [column for column in range(self.columns) if self.validateMove(column)]

    def getNextAvailablePosition(self, letter):
        rows, cols = self.rows, self.columns
        for row, row_vals in enumerate(self.connect4_board):
            for col, col_val in enumerate(row_vals[:-3]):
                if all(elem == letter for elem in row_vals[col:col+4]):
                    return row, col
            for col, col_vals in zip(range(cols), (self.connect4_board[r][col] for r in range(row, min(row+4, rows)))):
                if all(elem == letter for elem in col_vals):
                    return row, col
            for col, col_vals in enumerate(row_vals[:-3]):
                if row < rows-3 and col < cols-3:
                    diag_vals = [self.connect4_board[row+i][col+i] for i in range(4)]
                    if all(elem == letter for elem in diag_vals):
                        return row, col
            for col, col_vals in enumerate(row_vals[:-3]):
                if row >= 3 and col < cols-3:
                    diag_vals = [self.connect4_board[row-i][col+i] for i in range(4)]
                    if all(elem == letter for elem in diag_vals):
                        return row, col
        else:
            return -1, -1



    def validateWin(self, letter):
        for row in range(self.rows):
            for col in range(self.columns - 3):
                if all(self.connect4_board[row][col + i] == letter for i in range(4)):
                    return True

        for row in range(self.rows - 3):
            for col in range(self.columns):
                if all(self.connect4_board[row + i][col] == letter for i in range(4)):
                    return True

        for row in range(self.rows - 3):
            for col in range(self.columns - 3):
                if all(self.connect4_board[row + i][col + i] == letter for i in range(4)):
                    return True

        for row in range(3, self.rows):
```

```
        for col in range(self.columns - 3):
            if all(self.connect4_board[row - i][col + i] == letter for i in range(4)):
                return True

        return False



    def tossForFirstMove(self):
        choices = [1,2]
        return random.choice(choices)

    def validateFinalMove(self, SI_Agent_Letter, MinMax_Letter):
        return any(self.validateWin(letter) for letter in (SI_Agent_Letter, MinMax_Letter)) or not self.getValidMove()

class MinMax :

    def evaluate_MinMax_score(self, c4Game, letter, SIAgentLetter, MinMaxLetter):
        score = 0
        OtherPlayerLetter = MinMaxLetter if letter == SIAgentLetter else SIAgentLetter

        for i in range(c4Game.rows):
            row_array = [int(x) for x in list(c4Game.connect4_board[i,:])]
            col_array = [int(x) for x in list(c4Game.connect4_board[:,i])]
            for j in range(c4Game.columns-3):
                sub_row = row_array[j:j+4]
                sub_col = col_array[j:j+4]
                if sub_row.count(letter) == 4:
                    score += 1000
                elif sub_row.count(letter) == 3 and sub_row.count(0) == 1:
                    score += 100
                elif sub_row.count(letter) == 2 and sub_row.count(0) == 2:
                    score += 10
                if sub_row.count(OtherPlayerLetter) == 3 and sub_row.count(0) == 1:
                    score -= 10
                if sub_col.count(letter) == 4:
                    score += 1000
                elif sub_col.count(letter) == 3 and sub_col.count(0) == 1:
                    score += 100
                elif sub_col.count(letter) == 2 and sub_col.count(0) == 2:
                    score += 10
                if sub_col.count(OtherPlayerLetter) == 3 and sub_col.count(0) == 1:
                    score -= 10

        for i in range(c4Game.rows-3):
            for j in range(c4Game.columns-3):
                sub_diagonal1 = [c4Game.connect4_board[i+k][j+k] for k in range(4)]
                sub_diagonal2 = [c4Game.connect4_board[i+3-k][j+k] for k in range(4)]
                if sub_diagonal1.count(letter) == 4:
                    score += 1000
                elif sub_diagonal1.count(letter) == 3 and sub_diagonal1.count(0) == 1:
                    score += 100
                elif sub_diagonal1.count(letter) == 2 and sub_diagonal1.count(0) == 2:
                    score += 10
                if sub_diagonal1.count(OtherPlayerLetter) == 3 and sub_diagonal1.count(0) == 1:
                    score -= 10
                if sub_diagonal2.count(letter) == 4:
                    score += 1000
                elif sub_diagonal2.count(letter) == 3 and sub_diagonal2.count(0) == 1:
                    score += 100
                elif sub_diagonal2.count(letter) == 2 and sub_diagonal2.count(0) == 2:
                    score += 10
                if sub_diagonal2.count(OtherPlayerLetter) == 3 and sub_diagonal2.count(0) == 1:
                    score -= 10

        return score
```

```
    def Min_Max_Move_with_alpha_beta_pruning_and_depth(self, c4Game, connect4_board, current_depth, isMinMaxMove,
MinMaxLetter, SIAgentLetter, alpha, beta):

        if c4Game.validateFinalMove(SIAgentLetter, MinMaxLetter):

            if c4Game.validateWin(MinMaxLetter) :
                return (None, 10000000)

            elif c4Game.validateWin(SIAgentLetter) :
                return (None, -10000000)

            else:
                return (None, 0)

        if current_depth == 0 :
            return (None, self.evaluate_MinMax_score(c4Game, MinMaxLetter, SIAgentLetter, MinMaxLetter))


        possible_positions = c4Game.getValidMove()

        if isMinMaxMove:
            optimisedScore = -math.inf
            optimisedPosition = random.choice(possible_positions)

            for position in possible_positions:
                random_row = c4Game.getNextAvailableRow(position)
                connect4_board = c4Game.connect4_board.copy()
                connect4_board[random_row][position] = MinMaxLetter
                current_minmax_score = self.Min_Max_Move_with_alpha_beta_pruning_and_depth(c4Game, connect4_board, current_depth -
1, False, MinMaxLetter, SIAgentLetter, alpha, beta)[1]

                if current_minmax_score > optimisedScore:
                    optimisedScore = current_minmax_score
                    optimisedPosition = position

                alpha = max(optimisedScore, alpha)

                if alpha >= beta:
                    break

            return optimisedPosition, optimisedScore

        else:
            optimisedScore = math.inf
            optimisedPosition = random.choice(possible_positions)

            for position in possible_positions:
                random_row = c4Game.getNextAvailableRow(position)
                connect4_board = c4Game.connect4_board.copy()
                connect4_board[random_row][position] = MinMaxLetter
                current_minmax_score = self.Min_Max_Move_with_alpha_beta_pruning_and_depth(c4Game, connect4_board, current_depth -
1, True, MinMaxLetter, SIAgentLetter, alpha, beta)[1]

                if current_minmax_score < optimisedScore:
                    optimisedScore = current_minmax_score
                    optimisedPosition = position

                beta = min(beta, optimisedScore)

                if alpha >= beta:
                    break

        return optimisedPosition, optimisedScore

class SI_Agent :
```

```
    def Semi_Intelligent_Agent_Move(self, c4_game, SIAgentLetter, MinMaxLetter) :
        if c4_game.validateFinalMove(SIAgentLetter, MinMaxLetter):
            siagent_row, siagent_col = c4_game.getNextAvailablePosotion(SIAgentLetter)
            if siagent_row != -1:
                return siagent_row, siagent_col
            else:
                minmax_row, minmax_col = c4_game.getNextAvailablePosotion(MinMaxLetter)
                if minmax_row != -1:
                    return minmax_row, minmax_col
                else:
                    possible_positions = c4_game.getValidMove()
                    random_row = c4_game.getNextAvailableRow(random.choice(possible_positions))
                    random_col = random.choice(possible_positions)
                    return random_row, random_col
        else:
            possible_positions = c4_game.getValidMove()
            random_row = c4_game.getNextAvailableRow(random.choice(possible_positions))
            random_col = random.choice(possible_positions)

            return random_row, random_col

def play_connect4(SIAgent_plays_first, minmax_agent, si_agent, c4_game) :
    MinMaxLetter = 1
    SIAgentLetter = 2
    isGameOver = False
    gameWinner = ''
    while not isGameOver :

        if SIAgent_plays_first :

            si_chosen_row, si_chosen_column = si_agent.Semi_Intelligent_Agent_Move(c4_game, SIAgentLetter, MinMaxLetter)

            if c4_game.validateMove(si_chosen_column-1):
                SIAgent_plays_first = False
                c4_game.connect4_board[si_chosen_row][si_chosen_column] = SIAgentLetter

                if c4_game.validateWin(SIAgentLetter):
                    isGameOver = True
                    gameWinner = 'SIAgentWon'

            else:
                continue
        else:

            minmax_chosen_column, _ = minmax_agent.Min_Max_Move_with_alpha_beta_pruning_and_depth(c4_game,
c4_game.connect4_board,
                                6, True, MinMaxLetter, SIAgentLetter, -math.inf, math.inf)


            if c4_game.validateMove(minmax_chosen_column):
                SIAgent_plays_first = True
                minmax_chosen_row = c4_game.getNextAvailableRow(minmax_chosen_column)
                c4_game.connect4_board[minmax_chosen_row][minmax_chosen_column] = MinMaxLetter

                if c4_game.validateWin(MinMaxLetter):
                    isGameOver = True
                    gameWinner = 'MinMaxWon'

            else:
                continue

    return gameWinner if gameWinner != '' else 'Draw'

games = 100
SIAgentWin = 0
```

```
MinMaxWin = 0
Draw = 0

minmax_agent = MinMax()
si_agent = SI_Agent()

startTime = time.time()

for _ in tqdm(range(games)):
    c4_game = Connect4_Game()
    c4_game.initialise_board()

    SIAgent_plays_first = False
    if c4_game.tossForFirstMove() == 1 :
        SIAgent_plays_first = True
    else :
        SIAgent_plays_first = False

    try:
        winner = play_connect4(SIAgent_plays_first, minmax_agent, si_agent, c4_game)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Time taken (in seconds) with Depth = 8', 'Number of Games
MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Random'
statistics_dict['Total Number of Games'] = games
statistics_dict['Time taken (in seconds) with Depth = 8'] = totalTime
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 100
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

minmax_agent = MinMax()
si_agent = SI_Agent()

startTime = time.time()

for _ in tqdm(range(games)):
    c4_game = Connect4_Game()
    c4_game.initialise_board()

    SIAgent_plays_first = False

    try:
        winner = play_connect4(SIAgent_plays_first, minmax_agent, si_agent, c4_game)
    except:
        continue
```

```
    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Time taken (in seconds) with Depth = 8', 'Number of Games
MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: MinMax Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Time taken (in seconds) with Depth = 8'] = totalTime
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 100
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

minmax_agent = MinMax()
si_agent = SI_Agent()

startTime = time.time()

for _ in tqdm(range(games)):
    c4_game = Connect4_Game()
    c4_game.initialise_board()

    SIAgent_plays_first = True

    try:
        winner = play_connect4(SIAgent_plays_first, minmax_agent, si_agent, c4_game)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Time taken (in seconds) with Depth = 8', 'Number of Games
MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Semi Intelligent Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Time taken (in seconds) with Depth = 8'] = totalTime
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 100
```

```
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

minmax_agent = MinMax()
si_agent = SI_Agent()

startTime = time.time()

for _ in tqdm(range(games)):
  c4_game = Connect4_Game()
  c4_game.initialise_board()

  SIAgent_plays_first = False
  if c4_game.tossForFirstMove() == 1 :
    SIAgent_plays_first = True
  else :
    SIAgent_plays_first = False

  try:
    winner = play_connect4(SIAgent_plays_first, minmax_agent, si_agent, c4_game)
  except:
    continue

  if winner == 'MinMaxWon':
    MinMaxWin += 1
  elif winner == 'SIAgentWon':
    SIAgentWin += 1
  else:
    Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Time taken (in seconds) with Depth = 6', 'Number of Games
MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Semi Intelligent Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Time taken (in seconds) with Depth = 6'] = 133.962062
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 100
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

minmax_agent = MinMax()
si_agent = SI_Agent()

startTime = time.time()

for _ in tqdm(range(games)):
  c4_game = Connect4_Game()
  c4_game.initialise_board()

  SIAgent_plays_first = False

  try:
    winner = play_connect4(SIAgent_plays_first, minmax_agent, si_agent, c4_game)
  except:
    continue
```

```
    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Time taken (in seconds) with Depth = 6', 'Number of Games
MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: MinMax Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Time taken (in seconds) with Depth = 6'] = totalTime
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 100
SIAgentWin = 0
MinMaxWin = 0
Draw = 0

minmax_agent = MinMax()
si_agent = SI_Agent()

startTime = time.time()

for _ in tqdm(range(games)):
    c4_game = Connect4_Game()
    c4_game.initialise_board()

    SIAgent_plays_first = True

    try:
        winner = play_connect4(SIAgent_plays_first, minmax_agent, si_agent, c4_game)
    except:
        continue

    if winner == 'MinMaxWon':
        MinMaxWin += 1
    elif winner == 'SIAgentWon':
        SIAgentWin += 1
    else:
        Draw += 1

totalTime = time.time()-startTime

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Time taken (in seconds) with Depth = 6', 'Number of Games
MinMax Won', 'Number of Games Semi-Intelligent player Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Semi Intelligent Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Time taken (in seconds) with Depth = 6'] = 139.694996
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Semi-Intelligent player Won'] = SIAgentWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)
```

Appendix: Code for: Connect4_QLearningVsMinMax

```
import random
import math
from IPython.display import display
import pandas as pd
from tqdm import tqdm
import pickle
import numpy as np
import time

class Connect4_Game:

    def initialise_board(self) :
        self.rows = 6
        self.columns = 7
        self.connect4_board = np.zeros((self.rows, self.columns))

    validateMove = lambda self, column: self.connect4_board[len(self.connect4_board)-1][column] == 0

    getNextAvailableRow = lambda self, column: next((row for row in range(len(self.connect4_board)) if self.connect4_board[row][column]
== 0), None)

    getValidMove = lambda self: [column for column in range(self.columns) if self.validateMove(column)]

    def getNextAvailablePosition(self, letter):
        rows, cols = self.rows, self.columns
        for row, row_vals in enumerate(self.connect4_board):
            for col, col_val in enumerate(row_vals[:-3]):
                if all(elem == letter for elem in row_vals[col:col+4]):
                    return row, col
            for col, col_vals in zip(range(cols), (self.connect4_board[r][col] for r in range(row, min(row+4, rows)))):
                if all(elem == letter for elem in col_vals):
                    return row, col
            for col, col_vals in enumerate(row_vals[:-3]):
                if row < rows-3 and col < cols-3:
                    diag_vals = [self.connect4_board[row+i][col+i] for i in range(4)]
                    if all(elem == letter for elem in diag_vals):
                        return row, col
            for col, col_vals in enumerate(row_vals[:-3]):
                if row >= 3 and col < cols-3:
                    diag_vals = [self.connect4_board[row-i][col+i] for i in range(4)]
                    if all(elem == letter for elem in diag_vals):
                        return row, col
        else:
            return -1, -1



    def validateWin(self, letter):
        for row in range(self.rows):
            for col in range(self.columns - 3):
                if all(self.connect4_board[row][col + i] == letter for i in range(4)):
                    return True

        for row in range(self.rows - 3):
            for col in range(self.columns):
                if all(self.connect4_board[row + i][col] == letter for i in range(4)):
                    return True

        for row in range(self.rows - 3):
            for col in range(self.columns - 3):
                if all(self.connect4_board[row + i][col + i] == letter for i in range(4)):
                    return True

        for row in range(3, self.rows):
```

```
            for col in range(self.columns - 3):
                if all(self.connect4_board[row - i][col + i] == letter for i in range(4)):
                    return True

        return False



    def tossForFirstMove(self):
        choices = [1,2]
        return random.choice(choices)

    def validateFinalMove(self, SI_Agent_Letter, MinMax_Letter):
        return any(self.validateWin(letter) for letter in (SI_Agent_Letter, MinMax_Letter)) or not self.getValidMove()

class QLearning:
    def __init__(self):
        self.epsilon = 1.0
        self.QLearningStates = {}

    getPosition = lambda self, positions: int(''.join([str(int(position)) for position in positions.flatten()]))

    def getQLearningValue_For_Action(self, current_board, current_position):
        position = self.getPosition(current_board)
        if position not in self.QLearningStates:
            self.QLearningStates[(position, current_position)] = 0
        return self.QLearningStates[(position, current_position)]

    def getBestPositionFromQLearning(self, current_board, possible_positions):
        return random.choice(possible_positions) if random.random() < self.epsilon else
max([(self.getQLearningValue_For_Action(current_board, position), position) for position in possible_positions], key=lambda x: x[0])[1]

    def loadQLearningModel(self):
        with open("Connect4QLearningModel.pickle", "rb") as file:
            self.QLearningStates = pickle.load(file)

class MinMax :

    def evaluate_MinMax_score(self, c4Game, letter, SIAgentLetter, MinMaxLetter):
        score = 0
        OtherPlayerLetter = MinMaxLetter if letter == SIAgentLetter else SIAgentLetter

        for i in range(c4Game.rows):
            row_array = [int(x) for x in list(c4Game.connect4_board[i,:])]
            col_array = [int(x) for x in list(c4Game.connect4_board[:,i])]
            for j in range(c4Game.columns-3):
                sub_row = row_array[j:j+4]
                sub_col = col_array[j:j+4]
                if sub_row.count(letter) == 4:
                    score += 1000
                elif sub_row.count(letter) == 3 and sub_row.count(0) == 1:
                    score += 100
                elif sub_row.count(letter) == 2 and sub_row.count(0) == 2:
                    score += 10
                if sub_row.count(OtherPlayerLetter) == 3 and sub_row.count(0) == 1:
                    score -= 10
                if sub_col.count(letter) == 4:
                    score += 1000
                elif sub_col.count(letter) == 3 and sub_col.count(0) == 1:
                    score += 100
                elif sub_col.count(letter) == 2 and sub_col.count(0) == 2:
                    score += 10
                if sub_col.count(OtherPlayerLetter) == 3 and sub_col.count(0) == 1:
                    score -= 10

        for i in range(c4Game.rows-3):
```

```
        for j in range(c4Game.columns-3):
            sub_diagonal1 = [c4Game.connect4_board[i+k][j+k] for k in range(4)]
            sub_diagonal2 = [c4Game.connect4_board[i+3-k][j+k] for k in range(4)]
            if sub_diagonal1.count(letter) == 4:
                score += 1000
            elif sub_diagonal1.count(letter) == 3 and sub_diagonal1.count(0) == 1:
                score += 100
            elif sub_diagonal1.count(letter) == 2 and sub_diagonal1.count(0) == 2:
                score += 10
            if sub_diagonal1.count(OtherPlayerLetter) == 3 and sub_diagonal1.count(0) == 1:
                score -= 10
            if sub_diagonal2.count(letter) == 4:
                score += 1000
            elif sub_diagonal2.count(letter) == 3 and sub_diagonal2.count(0) == 1:
                score += 100
            elif sub_diagonal2.count(letter) == 2 and sub_diagonal2.count(0) == 2:
                score += 10
            if sub_diagonal2.count(OtherPlayerLetter) == 3 and sub_diagonal2.count(0) == 1:
                score -= 10

    return score

  def Min_Max_Move_with_alpha_beta_pruning_and_depth(self, c4Game, connect4_board, current_depth, isMinMaxMove,
MinMaxLetter, SIAgentLetter, alpha, beta):

    if c4Game.validateFinalMove(SIAgentLetter, MinMaxLetter):

        if c4Game.validateWin(MinMaxLetter) :
            return (None, 10000000)

        elif c4Game.validateWin(SIAgentLetter) :
            return (None, -10000000)

        else:
            return (None, 0)

    if current_depth == 0 :
        return (None, self.evaluate_MinMax_score(c4Game, MinMaxLetter, SIAgentLetter, MinMaxLetter))


    possible_positions = c4Game.getValidMove()

    if isMinMaxMove:
        optimisedScore = -math.inf
        optimisedPosition = random.choice(possible_positions)

        for position in possible_positions:
            random_row = c4Game.getNextAvailableRow(position)
            connect4_board = c4Game.connect4_board.copy()
            connect4_board[random_row][position] = MinMaxLetter
            current_minmax_score = self.Min_Max_Move_with_alpha_beta_pruning_and_depth(c4Game, connect4_board, current_depth -
1, False, MinMaxLetter, SIAgentLetter, alpha, beta)[1]

            if current_minmax_score > optimisedScore:
                optimisedScore = current_minmax_score
                optimisedPosition = position

            alpha = max(optimisedScore, alpha)

            if alpha >= beta:
                break

        return optimisedPosition, optimisedScore

    else:
        optimisedScore = math.inf
```

```
        optimisedPosition = random.choice(possible_positions)

        for position in possible_positions:
            random_row = c4Game.getNextAvailableRow(position)
            connect4_board = c4Game.connect4_board.copy()
            connect4_board[random_row][position] = MinMaxLetter
            current_minmax_score = self.Min_Max_Move_with_alpha_beta_pruning_and_depth(c4Game, connect4_board, current_depth -
1, True, MinMaxLetter, SIAgentLetter, alpha, beta)[1]

            if current_minmax_score < optimisedScore:
                optimisedScore = current_minmax_score
                optimisedPosition = position

            beta = min(beta, optimisedScore)

            if alpha >= beta:
                break

    return optimisedPosition, optimisedScore

def play_connect4(MinMaxPlaysFirst, qLearningPlayer, minmaxPlayer, c4Game):

    QLearningLetter = 1
    MinMaxLetter = 2

    while True:
        if MinMaxPlaysFirst:

            MinMaxPossible_Positions = c4Game.getValidMove()

            if len(MinMaxPossible_Positions) == 0:
                return "Draw"

            minmax_chosen_column, _ = minmaxPlayer.Min_Max_Move_with_alpha_beta_pruning_and_depth(c4Game,
c4Game.connect4_board,
                        6, True, MinMaxLetter, QLearningLetter, -math.inf, math.inf)

            minmax_chosen_row = c4Game.getNextAvailableRow(minmax_chosen_column)
            c4Game.connect4_board[minmax_chosen_row][minmax_chosen_column] = MinMaxLetter

            if c4Game.validateWin(MinMaxLetter) :
                return "MinMaxWon"

            if c4Game.validateWin(QLearningLetter):
                return "QLearningWon"

            if len(c4Game.getValidMove()) == 0 :
                return "Draw"

            QLearningPossible_Positions = c4Game.getValidMove()

            if len(QLearningPossible_Positions) == 0:
                return "Draw"

            QLearning_chosen_column = qLearningPlayer.getBestPositionFromQLearning(c4Game.connect4_board,
QLearningPossible_Positions)
            QLearning_chosen_row = c4Game.getNextAvailableRow(QLearning_chosen_column)
            c4Game.connect4_board[QLearning_chosen_row][QLearning_chosen_column] = QLearningLetter

            if c4Game.validateWin(QLearningLetter):
                return "QLearningWon"

            if c4Game.validateWin(MinMaxLetter) :
                return "MinMaxWon"

            if len(c4Game.getValidMove()) == 0 :
```

```
            return "Draw"

        else:
            QLearningPossible_Positions = c4Game.getValidMove()

            if len(QLearningPossible_Positions) == 0:
                return "Draw"

            QLearning_chosen_column = qLearningPlayer.getBestPositionFromQLearning(c4Game.connect4_board,
QLearningPossible_Positions)
            QLearning_chosen_row = c4Game.getNextAvailableRow(QLearning_chosen_column)
            c4Game.connect4_board[QLearning_chosen_row][QLearning_chosen_column] = QLearningLetter

            if c4Game.validateWin(QLearningLetter):
                return "QLearningWon"

            if c4Game.validateWin(MinMaxLetter) :
                return "MinMaxWon"

            if len(c4Game.getValidMove()) == 0 :
                return "Draw"


            MinMaxPossible_Positions = c4Game.getValidMove()

            if len(MinMaxPossible_Positions) == 0:
                return "Draw"

            minmax_chosen_column, _ = minmaxPlayer.Min_Max_Move_with_alpha_beta_pruning_and_depth(c4Game,
c4Game.connect4_board,
                                6, True, MinMaxLetter, QLearningLetter, -math.inf, math.inf)

            minmax_chosen_row = c4Game.getNextAvailableRow(minmax_chosen_column)
            c4Game.connect4_board[minmax_chosen_row][minmax_chosen_column] = MinMaxLetter

            if c4Game.validateWin(MinMaxLetter) :
                return "MinMaxWon"

            if c4Game.validateWin(QLearningLetter):
                return "QLearningWon"

            if len(c4Game.getValidMove()) == 0 :
                return "Draw"

games = 200
MinMaxWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

minmaxPlayer = MinMax()


print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    c4Game = Connect4_Game()
    c4Game.initialise_board()

    MinMaxPlaysFirst = False
    if c4Game.tossForFirstMove() == 1 :
        MinMaxPlaysFirst = True
    else :
        MinMaxPlaysFirst = False

    winner = play_connect4(MinMaxPlaysFirst, qLearningPlayer, minmaxPlayer, c4Game)
```

```
    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'MinMaxWon':
        MinMaxWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
MinMax Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Random '
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 200
MinMaxWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

minmaxPlayer = MinMax()


print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    c4Game = Connect4_Game()
    c4Game.initialise_board()

    MinMaxPlaysFirst = False

    winner = play_connect4(MinMaxPlaysFirst, qLearningPlayer, minmaxPlayer, c4Game)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'MinMaxWon':
        MinMaxWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
MinMax Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Q-Learning Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)

games = 200
MinMaxWin = QLearningWin = Draw = 0

qLearningPlayer = QLearning()
qLearningPlayer.loadQLearningModel()

minmaxPlayer = MinMax()
```

```
print(f"Current Q Learning model has {len(qLearningPlayer.QLearningStates)} states")

for _ in tqdm(range(games)):
    c4Game = Connect4_Game()
    c4Game.initialise_board()

    MinMaxPlaysFirst = True

    winner = play_connect4(MinMaxPlaysFirst, qLearningPlayer, minmaxPlayer, c4Game)

    if winner == 'QLearningWon':
        QLearningWin += 1
    elif winner == 'MinMaxWon':
        MinMaxWin += 1
    else:
        Draw += 1

statistics_df = pd.DataFrame(columns=['Game Type', 'Total Number of Games', 'Number of Games QLearning Won', 'Number of Games
MinMax Won', 'Number of Games Drawn'])
statistics_dict = {}
statistics_dict['Game Type'] = 'First Move: Min-Max Player'
statistics_dict['Total Number of Games'] = games
statistics_dict['Number of Games QLearning Won'] = QLearningWin
statistics_dict['Number of Games MinMax Won'] = MinMaxWin
statistics_dict['Number of Games Drawn'] = Draw
statistics_df = statistics_df.append(statistics_dict, ignore_index = True)
statistics_df = statistics_df.style.applymap(lambda x:'white-space:nowrap')
display(statistics_df)
```