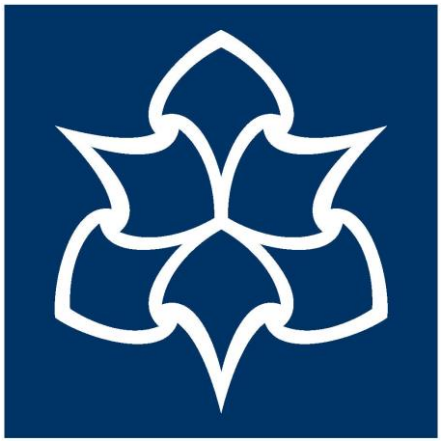
An Online Learning Game to Demonstrate How Machines Can be Trained to Win Noughts and Crosses (MENACE)

A DISSERTATION SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING



2020

By
Jan Clare
Department of Computing and Mathematics

Table of Contents

List of Figures

2. Literature Survey

3. Problem Analysis

4. Design and Implementation

5. Evaluation

Abstract

This project attempts to make Artificial Intelligence more accessible to newcomers in the field by creating a website that demonstrates how machines can learn using MENACE as an example.

Matchbox Educable Noughts and Crosses Engine (MENACE) is a Reinforcement Learning algorithm developed by Donald Michie in 1961 (Michie, 1961). This project implements MENACE and trains it against various opponent types such as the optimal and random players. It was found that MENACE made the most learning progress against the optimal player when using moderate reinforcements.

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures and has received ethical approval number Your EthOS Number.

Signed: Jan Clare

Date: 30/09/2022

Acknowledgements

I would like to thank my family and friends for their support.

Abbreviations

MENACE	Matchbox Educable Noughts and Crosses Engine
AI	Artificial Intelligence
KNN	K-Nearest Neighbour
SVM	Support Vector Machines
SHAP	Shapley Additive explanations
MDP	Markov Decision Process
MVC	Model View Controller
SQL	Structured Query Language
EF	Entity Framework

1 Introduction

1.1 Background and Motivation

Many academics in the field of Artificial Intelligence, such as Russell and Norvig and Kai-Fu Lee, acknowledge the benefits AI technology has today. In addition, they recognise the potential for AI to significantly improve civilization and society (J.Russell & Norvig, 2021). However, AI technology has many risks and ethical problems (See section 2.1). The motivation of this project is to contribute to mitigating these risks by helping improve societal awareness of AI.

The aspiration and vision of this project is to build an informative website which demonstrates how machines can learn. This will be aimed at newcomers to the field of AI and therefore constitute a relatively simple AI problem i.e., Noughts and Crosses. The AI model chosen for this problem is a Reinforcement Learning algorithm constructed by Donald Michie in 1961 called MENACE (Matchbox Educable Noughts and Crosses Engine) (Michie, 1961).

MENACE was originally built physically out of matchboxes and beads and without the use of a computer. In a nutshell, MENACE learns by playing Noughts and Crosses games and positively reinforcing moves which led to victory and negatively reinforcing moves which lead to defeat (Michie, 1961). When compared with modern AI technology such as neural networks, MENACE is a relatively simple model. This should make it accessible and understandable for those completely unfamiliar with how machines can learn.

It is hoped the website will allow users to interact with their own instance of MENACE and train it using various methods. They will be able to clearly see the internal workings of MENACE and watch MENACE learn as it plays games. In addition to this, the website will present information acquired by the research conducted in this project.

1.2 Aims and Objectives

1.2.1 Aim

Design and build an explainable AI model (MENACE) that learns to play a game of noughts and crosses.

1.2.2 Objectives

1. Conduct a Literature Survey contextualising the project.
2. Design and develop a system allowing users to play noughts and crosses against a computer.
3. Create a training dataset.
4. Implement, train, and test MENACE and various algorithms.
5. Evaluate and compare all algorithm performances.
6. Design and develop a user interface.
7. Conduct a user evaluation of the user interface.

1.3 Overview

A literature survey has been conducted in Chapter 2 which researches and contextualises the topics and issues relevant for this project. Chapter 3 is a problem analysis of MENACE and how to implement it. To achieve the aim and objectives a C# class library implementing MENACE, and a website demonstrating MENACE has been developed and explained thoroughly in Chapter 4. Chapter 5 evaluates the performance of MENACE and the success of the website.

2 Literature Survey

This chapter explores the literature contextualising MENACE and its relevance. The need for mitigating AI risks and making the study of it as accessible as possible is discussed in section 2.1 by highlighting the Risks and Ethical Issues involved with AI. Section 2.2, 2.3, 2.4, 2.5 discusses the contemporary research of the fields relevant to MENACE. This includes Artificial Intelligence, Machine Learning and its challenges, and Reinforcement Learning. Section 2.6. describes MENACE and its origins.

2.1 The Risks and Ethical Issues of Artificial Intelligence and The Importance of AI Awareness

Artificial Intelligence benefits society in many ways. One of the most respected experts on AI, Kai-Fu Lee predicts that AI will soon enable self-driving cars, be managing personal records and manufacturing a large proportion of consumer goods (Lee, 2018). Russell and Norvig, in their popular AI textbook entitled AI a modern approach, go further, asserting that AI will save lives through new medical breakthroughs, enhanced medical diagnosis, and improved forecasting of dangerous weather events (J.Russell & Norvig, 2021). Though these are exciting prospects, both Lee and Russell and Norvig as well as many other academics acknowledge the potential risks and ethical issues that developments in AI technology will incur. This section discusses some of these concerns and argues that increasing the general awareness of the basics of AI is one way to mitigate these concerns.

One of the main dangers of Artificial Intelligence is its potential for misuse in surveillance as well as influencing society. Russell and Norvig point out that our increasing reliance on computers through use of mobile phones and the internet etc., allows governments and corporations to gather substantial data on us. AI technologies such as speech recognition, computer vision and natural language processing make this possible (J.Russell & Norvig, 2021). These technologies make it possible to tailor information flows on social media individually for every user (J.Russell & Norvig, 2021). This can be used by governments to influence society for political benefits or by organisations for capital gain. An affirming example of this is the 2016 Cambridge Analytica scandal in which the consultancy firm used psychological profiles built from the personal data of millions of nonconsenting Facebook users to influence the 2016 US presidential election (Hinds, et al., 2020). In the West there are measures to mitigate the risk of organisations misusing AI technology. For example, the European Union's General Data Protection Act (GDPR) aims to protect individuals control of their personal data (Zaeem & Barber, n.d.). Jinghan Zeng's article on Artificial Intelligence and China's Authoritarian Government also highlights that the USA and Europe have slowed the development of AI surveillance technologies such as facial recognition due to ethical concerns (Zeng, 2020). However, he also points out that China is now leading the way in this domain and that it has largely contributed to enabling the CCP (Chinese Communist Party) to bolster governmental authority through digital means (Zeng, 2020). Zeng also partly attributes this to a weak civil awareness of the dangers of AI within Chinese society (Zeng, 2020). Therefore, the danger AI poses in influencing society creates the necessity for people to be aware of AI technology and its uses.

Another danger of AI is the impact it may have on employment. Russell and Norvig explain that fears of new technologies automating jobs and replacing workers have been around for most of human history (J.Russell & Norvig, 2021). They explain that new technologies can sometimes cause significant social changes and even major redistributions of wealth, and it is likely that continued

development of AI technology will have this effect (J.Russell & Norvig, 2021). An example of this in the past is the industrial revolution when the nations of Europe underwent transitions from pre-industrial agricultural societies to modern industrialised economies. These rapid changes caused significant civil unrest as well as demand for social reform (Price, 1988).

AI algorithms are also susceptible to biases which could cause unfairness in high stake situations such as employment and loaning (Mehrabi, et al., 2021). The more reliant society becomes on AI for decision making, the more important it is to ensure these risks are addressed (J.Russell & Norvig, 2021). Given that AI technology is already beginning to impact society on a scale comparable with the industrial revolution, with the technology said to be worth an expected $15 trillion by 2030 (Rao, et al., 2017), Russell and Norvig's point is a plausible concern. One way to reduce this risk is to increase the general civil understanding of AI technologies so we are better prepared for the changes to society it may bring.

Lack of understanding and awareness of how AI systems work increases all the risks discussed in this section. Many AI systems used in the industry today are black boxes meaning their architectures can be explained from a high level, but the internal workings are obscured by complexity (Carabantes, 2020). For example, the YouTube recommendation algorithm is a large-scale AI system with immense complexity comprised of two neural networks (Covington, et al., 2016). Though the architecture is explainable, it is impossible to explain, exactly, how the algorithm derives a specific instance of video recommendation. The Defence Advanced Research Projects Agency highlights the commonly observed trade-off in AI models between performance and explain-ability (DARPA , 2016). In addition to this, academics and professionals in the field now have access to powerful coding libraries, allowing them to apply a great variety of statistical, Machine Learning and Neural Network models to a myriad of scenarios. The use of these libraries is possible even with a limited understanding of the computation and theory behind them. These tools have made Artificial Intelligence more accessible to people who may be less informed about the risks of AI. This is a particularly concerning issue when it comes to safety critical applications of AI e.g., self-driving cars. Russell and Norvig stress the importance of developing technical and ethical standards which are as rigorous as those in other engineering and healthcare disciplines where safety is critical (J.Russell & Norvig, 2021). It is also therefore important that those concerned with AI have a good knowledge of the field to reduce the risk of deploying bad systems which may lead to disastrous consequences (Yampolskiy, 2016).

2.2 Defining Artificial Intelligence

Artificial Intelligence is notoriously difficult to define succinctly because it is a widely researched topic with many different subfields (J.Russell & Norvig, 2021). This section discusses the ideas of a few academics and their attempts to define AI.

In 1950 Alan Turing proposed the Imitation Game. This was a theorised game concerned with determining if "machines can think" (Turing, 1950). In a nutshell, a computer player attempts to imitate a human player while the human player attempts to assist an interrogator in making the correct distinction between which player is which. If the interrogator demonstrates they are consistently incapable of distinguishing between human and machine, then there is convincing evidence that the machine participant is intelligent (Turing, 1950).

With the benefit of seeing 70 years of development in the field since Turing's Imitation Game, and the hindsight that researchers have committed few resources to passing the Turing Test (J.Russell & Norvig, 2021), Russell and Norvig offer a more informed explanation of Artificial Intelligence. They

categorise Turing's Imitation Game approach as AI that endeavours to act humanly (J.Russell & Norvig, 2021). Russell and Norvig recognise the important AI skills a machine must possess to perform well in the Imitation Game, namely: natural language processing, knowledge representation, automated reasoning, machine learning, computer vision and robotics (J.Russell & Norvig, 2021). However, they point out that human intelligence and imitating it, is only one form of Artificial Intelligence of which there are others. Human intelligence does not always follow perfect mathematical rationality and imitating human intelligence differs from possessing it. These ideas are the basis for Russell and Norvig's other forms of AI. This includes AI that acts rationally which they define as an agent that endeavours to achieve the optimal outcome (J.Russell & Norvig, 2021). They also assert that historically this has been the most successfully developed form of AI. This is because acting rationally can be achieved using methods that other forms of AI use. For example, the skills needed to act humanly can also lead an AI to act rationally. Furthermore, it is easier to measure whether an agent achieves the best outcome in a well-defined environment than it is to measure how humanly or rationally an AI agent thinks (J.Russell & Norvig, 2021).

Russell and Norvig's modern way of thinking about Artificial Intelligence is somewhat consistent with Vasant Honavar's overview of AI which attributes the difficulty in defining AI to the difficulty in defining intelligence. Despite the many observable examples of intelligence that humans and animals can demonstrate, it is challenging to quantify or measure. Honavar suggests that it is helpful to think of intelligence as a set of attributes (Honavar, 2016) including perception, action, reasoning, adaptation and learning, communication, autonomy etc. (Honavar, 2016). Honavar asserts that it is currently unlikely a single AI system possesses all these attributes combined but recognises, akin to Russell and Norvig, that combinations of them are exhibited in various disciplines of AI. Honavar eventually summarises Artificial Intelligence as the process of researching and creating intelligent systems (Honavar, 2016).

A lot of contemporary AI work can be categorised into various disciplines such as Computer Vision, Robotics, Pattern Recognition, Stock Market analysis (Honavar, 2016) etc. Fundamental to many of these disciplines is Machine Learning.

2.3 Machine Learning

Russell and Norvig define Machine Learning as computer agents that improve their performance by making observations about the world (J.Russell & Norvig, 2021). In other words, Machine Learning is the subject in AI concerned with giving machines the ability to think and learn independently (Alzubi, et al., 2018) without being explicitly programmed (Samuel, 1959). Machine Learning has many practical uses such as image reconstruction or beating chess masters. This is possible because modern technology facilitates rapid processing speeds over vast quantities of data (Alzubi, et al., 2018). Russell and Norvig explain that generally Machine Learning is good for variable problems when designers cannot foresee all contingencies, or when the problem is too complex for someone to program a direct solution (J.Russell & Norvig, 2021).

Machine Learning problems can be categorised into different types. Problems in which an agent labels something based on its attributes is called classification. For example, classifying the weather as sunny, cloudy, or rainy based on temperature and humidity (J.Russell & Norvig, 2021). When an agent quantifies something based on its attributes, such as predicting the price of a car based on mileage, age and car make, it is a regression problem (Alzubi, et al., 2018).

2.3.1 Supervised Learning

In addition to the various types of Machine Learning problems, there are different ways to conduct the learning process. In supervised learning, an agent learns a function mapping input sets to an output value. This is done by looking at examples of inputs and the outputs they produce (J.Russell & Norvig, 2021). A typical example of this is training an agent to recognise handwritten digits. This works by giving the agent example images of handwritten digits with their label. The agent uses some Machine Learning model such as K-Nearest Neighbour or a Neural Network to associate pixel values with different digit labels (Bottou, 1994).

2.3.2 Unsupervised Learning

Whereas supervised learning entails the use of labelled datasets for mapping inputs to outputs, unsupervised learning attempts to discover patterns in unlabelled data (IBM Cloud Education, 2020). IBM's website on Machine Learning technology asserts that unsupervised learning is useful for use cases such as exploratory data analysis because of its ability to discover relationships that are difficult for humans to detect (IBM Cloud Education, 2020). According to Russell and Norvig, a typical unsupervised learning task is to cluster similar data together, such as images on the internet which contain cats (J.Russell & Norvig, 2021).

2.3.3 Machine Learning Models

There are many machine learning algorithms used for the problem and learning types discussed above. This section will summarise some of the commonly used algorithms.

Regression is typically used in supervised learning for predicting continuous variables (Uysal & Guvenir, 319-340). In its simplest form this means calculating a line of best fit between an independent and dependant variable. However, complex models may have multivariate regression analysis which plots a line through multiple variables (Ray, 2019). Regression also does not need to be linear as it is possible to plot polynomial lines and attempt to fit them to data.

Support Vector Machines (SVM) work by drawing decision boundaries between clusters of data based on how they are classified in training data (Salcedo-Sanz, et al., 2014). For example, on a graph which plots the relationship between car price and mileage, an SVM may calculate a decision boundary predicting data values on one side of a boundary as expensive and cheap on the other. With complicated datasets calculating decision boundaries can be tricky. For example, one cluster could be completely engulfed by another. SVMs can solve this by adding dimensions to a graph and applying some function to all data points which maps them to the new graph. This can help distinguish clusters and make it easier to draw hyperplanes cutting between them (Cournapeau & Brucher, 2022).

K nearest neighbour (KNN) is an algorithm which classifies data (Ray, 2019). It works by plotting unseen data on a graph with the training data. It then finds the closest k points (from the training set). From these, it looks at their target value and predicts the most frequently occurring one as the classification for the unknown data. The value of k can be experiment with.

Neural networks are Machine Learning algorithms inspired by the human brain. They are layers of connected nodes which are designed to mimic the behaviour of neurons (Zou, et al., 2008). Nodes receive weighted signals from all connected nodes in the previous layer. If the sum of these values surpasses some limit it passes a signal to all the nodes in the next layer. Neural networks learn by viewing many example inputs and attempting to predict their output. In supervised learning, the desired outputs are provided, and the neural network aims to minimise the difference between this

and its own prediction. This happens through a process called back propagation which modifies the weights in node connections (Zou, et al., 2008).

Another model type is ensembles. Ensembles take multiple models and combine their individual predictions into a global prediction. For classification problems this can be done by voting or in regression typically the mean of the predictions is taken as the global model's prediction (Opitz & Maclin, 1999).

2.4 Challenges of Machine Learning

Developing a Machine Learning model can involve various challenges. A typical development process has the following steps (Alzubi, et al., 2018):

1. Collection and preparation of Data
2. Feature selection
3. Choice of algorithm
4. Selection of model parameters
5. Training
6. Performance evaluation

2.4.1 Collection and Preparation of Data and Feature Selection

Before an AI model can be developed it is necessary to gather data which encapsulates the problem to be solved (Brownlee, 2020). To build a good AI model it must be trained on reliable data. However, it is not uncommon for datasets to be flawed, contain discrepancies, or be bloated by irrelevant data (Alzubi, et al., 2018). These issues must be amended before it can be used in a model. The data must also be suitable and compatible with whichever algorithm is going to be used. Data does not always contain direct information about the problem, and it often needs to be modified extensively and features of the dataset need to be selected or created (Brownlee, 2020).

There are common challenges with collecting and preparing data including missing values, erroneous data, mixed formats, duplicate data etc. (Chu, et al., 2016). Missing data values can be resolved by creating substitute values or, if data is abundant, deleting the rows in which the missing values occurred. Sometimes features in a dataset contain only one value or very low variance which means it does contain much information. These features are normally removed. Datasets sometimes contain duplicate data; this can also be removed. Another problem is detecting and removing outliers which might skew or influence a Machine Learning algorithms accuracy (Brownlee, 2020).

An important concept in Machine Learning is overfitting vs generalization. If a model is trained on a dataset which contains too much noise, then it will struggle to learn anything meaningful (Ying, 2018). Conversely if a dataset has been stripped of too much information, to avoid erroneous data being used, then there is a risk the model trained on it will only learn how to cope with the data provided. It will be useless when deployed in the real world. In other words, it will have overfitted to the data provided (Ying, 2018).

2.4.2 Choosing an Algorithm and Selecting Model Parameters for Training

Machine Learning algorithms have properties which make them suitable for different problems (Alzubi, et al., 2018). Given a Machine Learning problem, it is important to select the most suitable algorithm. This usually involves some experimentation. In addition to selecting the right algorithm, it

is important to determine appropriate parameters. For example, a polynomial regression model can have any number of turning points, or a decision tree can have any number of nodes.

Regression algorithms have the benefit that they are explainable, and easy to avoid overfitting (Ray, 2019). This makes them suitable for predicting values in obvious relationships, where a complex model would be overkill. However, regression models tend to be relatively simplistic and ineffective in complex applications or real-world issues (Ray, 2019).

Support vector machines are good for managing organised and semi-structured data. They also benefit from being less susceptible to overfitting as well as being scalable to data with many dimensions (Ray, 2019). However, for large datasets the time complexity of the algorithm can become undesirable especially when used on limited computing resources. Furthermore, the performance of SVMs often suffers when used with noisy datasets or when there are many outliers (Ray, 2019).

K Nearest Neighbour is a straightforward algorithm which is versatile and easy to build (Ray, 2019). In addition, it offers the benefit that no training is necessary. However, it is a memory intensive algorithm (Bottou, 1994) so does not scale to excessively large datasets. When selecting parameters for a K Nearest Neighbour model, a large k value can lead to over fitting whilst a value of k that is too small can lead to a simplistic and erroneous models. K nearest neighbour can also be weighted. If set to ‘uniform’ the closest k points will all be weighted equally. Conversely, if set to ‘distance’, the model will consider the distance of the k nearest data points to the unseen data to be predicted (Cournapeau & Brucher, 2022).

Neural networks are excellent for fitting to data that is not linear and in situations where the data has high dimensionality (Livingstone, et al., 1997). Another advantage of neural networks is that their general architecture is highly customisable and useful in a wide range of problems. For example, neural networks designed for image recognition often make use of convolutional layers which transform input images before being fed to the standard dense layers. The idea is that convolutional layers help to identify patterns in the input which significantly improves the accuracy of the following network (Albawi, et al., 2017). However, a hindrance of neural networks is that they can become extremely complex and unwieldy. They often require a lot of computing power. Furthermore, they can often be regarded as black boxes (Ray, 2019). This means in a lot of cases it is difficult or impossible to explain exactly how a neural network reached some conclusion given an input, because of its complexity (Ray, 2019). This can cause ethical problems when neural networks are relied on for decision making such as propagating biases and discrimination.

2.4.3 Evaluation

After a model has been trained on some data, its performance needs to be evaluated. This is typically done by reserving some training data for testing later (Alzubi, et al., 2018). There are various metrics for analysing the performance of a model on test data.

In classification problems the common metrics are micro and macro accuracy. Micro accuracy is the proportion of predictions a model got right across the whole test dataset. Macro accuracy takes the average of proportions of correct predictions in each classification group (Pedregosa, et al., 2011).

In regression problems the commonly used metrics are mean absolute error and root mean squared error. Both use the concept of residuals which is the difference between the model’s predicted value and the true value. Mean absolute error is the mean of the absolute values of the residuals whereas root mean squared error is the square root of the mean of the residuals squared. Root mean

squared error avoids the need to change the polarity of residuals (e.g., -20 becomes +20), however, since outliers will have their distance to non-outliers inflated as a result of squaring their values, this metric is affected more by, and assigns more weight to, the influence of outliers (Pedregosa, et al., 2011).

Another important challenge is ensuring the performance of a model on a test dataset is a reliable result. This can be achieved using a technique called cross validation. Cross validation splits the full dataset into segments the size of a test dataset. It then trains the model several times such that each segment has a turn to be the test data while the rest is used for training. This facilitates the comparison of multiple evaluations of the model over different training and test data without sacrificing any of the available data (Pedregosa, et al., 2011).

Another challenge is evaluating complex models such as neural networks where it is difficult to precisely explain how predictions were made. This can be done using a tool called SHAP (Shapely Additive Explanations). SHAP automatically experiments with input data and notes the output from the model. Through exhaustive experimentation, it can determine what aspects of the input data influence aspects of the model's output (Lundberg, 2022).

2.5 Reinforcement Learning

MENACE is an example of a Reinforcement Learning algorithm. Pierre Yves Glorennec's paper titled "Reinforcement Learning: An Overview" defines Reinforcement Learning as "a family of problems in which an agent evolves while analysing the consequences of its actions, through a simple scalar signal given by the environment" (Glorennec, 2000). This aligns with Russell and Norvig's assessment that Reinforcement Learning is when an agent interacts with an environment and regularly receives feedback reflecting its performance (J.Russell & Norvig, 2021).

In contrast to other Machine Learning paradigms such as supervised learning, where a model is adjusted to produce a desired output which is known at all stages of the models training, Reinforcement Learning aims to develop an agent's skill through trial and error. As Glorennec puts it: an agent is given reinforcement signals indicating: "what you have to do without saying how to do it" (Glorennec, 2000).

Russell and Norvig express that Reinforcement Learning is particularly useful in situations where the number of possible scenarios a model must prepare for is impractically large (J.Russell & Norvig, 2021). This is because typical supervised or unsupervised learning methods normally depend on a dataset being prepared and provided upfront. Examples affirming this is Donald Michie's MENACE which learned to play Noughts and Crosses without the use of computer resources (Michie, 1961) or Arthur Samuels Checkers Reinforcement algorithm which learned to play checkers despite there being 10^{40} possible games (Samuel, 1959).

2.5.1 Markov Decision Processes

An important concept for understanding Reinforcement Learning is Markov Decision Processes (MDPs). MDPs are designed to be an understandable framework for formally discussing sequential decision making (Sutton & Barto, 2020). MDPs are a widely understood topic and its definition is not contested among academics. The following section will mostly be based on Richard Sutton and Andrew Barto's explanation of MDPs from their textbook entitled Reinforcement Learning an Introduction (Sutton & Barto, 2020). MDPs have the following components:

- Environment

- Agent
- States – a finite set of states in the environment denoted as S.
- Actions – a finite set of actions an agent can take denoted as A.
- Rewards –a finite set of rewards an agent can receive denoted as R.

At each time step t, the agent has a representation of the environment state $s_t \in S$. From a state s_t the agent has a set of actions it can take $A_t \in A$ (Kaelbling, et al., 1996). The action the agent chooses A_t, changes the environment state in the next time step, denoted as $s_{t+1} \in S$. The change from environment state s_t to s_{t+1} causes the agent to receive a reward signal $r_{t+1} \in R$. The reward signal modifies the agent's behaviour (Sutton & Barto, 2020).

An MDP agent aims to maximise the total sum of rewards it receives over its interactions with the environment (Glorennec, 2000). In MDPs this is encapsulated by the concept of an expected return of the rewards from a given time step denoted as G_t (Sutton & Barto, 2020). This is defined as the total of future rewards:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T \quad (2.1)$$

Where r_T is the reward given to the agent at the final time step T (Sutton & Barto, 2020). The expected return dictates the actions agents decide to take from the states they are in.

2.5.2 Policies and Value Functions

In MDPs, an agent follows a policy denoted as π. A policy is the strategy an agent is currently using based on its experience in the past. More formally, they are functions mapping the actions an agent can take from a given state s_t to the probability of the agent choosing the action (Sutton & Barto, 2020). If an agent is following a policy π, then from state s_t the probability of the agent choosing an action A_t is given by $\pi(a|s)$ (Sutton & Barto, 2020).

Action value functions of a policy estimate how good an action is from a given state. An action value function of a policy π is denoted as q_π. Action value functions are measures of expected return. The value of an action A_t from state s_t when following a policy π is the expected return from continuing to follow π after taking the action (Sutton & Barto, 2020).

Reinforcement Learning agents try to learn a policy which yields the highest reward. Policies are compared according to their expected return. The policy that yields the highest reward is called the optimal policy (Sutton & Barto, 2020). Optimal policies have optimal action value functions denoted as q_*. This gives the greatest expected return possible, following any policy, for all possible state action pairs.

2.5.3 Episodic Tasks

For games like Noughts and Crosses, where there is a natural end to the game, it is logical to consider the idea of a final time step (Sutton & Barto, 2020). When a Noughts and Crosses game ends, a new grid is drawn, and the game begins again, independently of the old game. However, the players remember what they learned from the old game and take this into the new one. In an MDP this is encapsulated by the concept of episodic tasks. When an environment and agent reach the final time step, the environment is reset, and the agent starts again from a standard starting state (Sutton & Barto, 2020).

2.6 MENACE

MENACE is the subject of Donald Michie's 1961 paper entitled "Experiments on the Mechanization of Game Learning Part 1: Characterization of the Model and its Parameters". MENACE (Matchbox Educable Noughts and Crosses Engine) is a reinforcement learning algorithm capable of learning to play Noughts and Crosses. It was originally built using matchboxes and beads but later simulated on a computer (Michie, 1961).

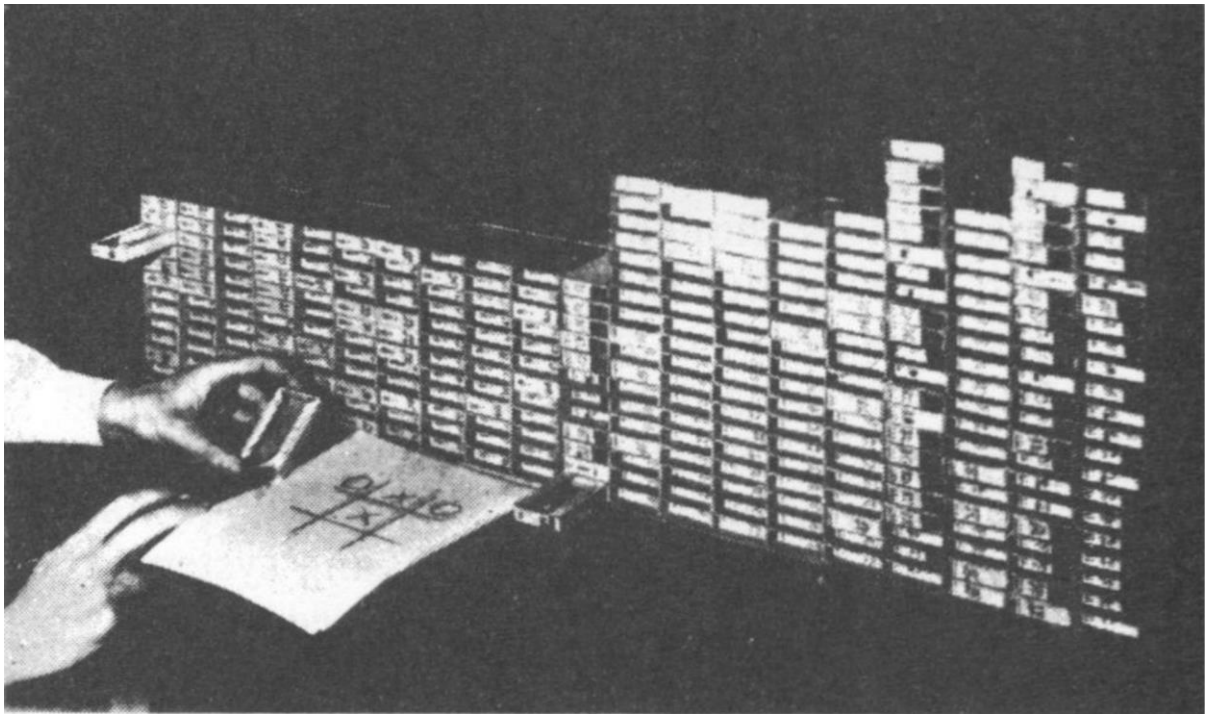


Figure 2.2 The original MENACE (Michie, 1961).

MENACE consists of matchboxes each representing a Noughts and Crosses board position. Matchboxes contain beads which are colour coded to represent the moves MENACE can play from the matchboxes' board position. During a turn, the matchbox representing the current board position of the game will be found and a random bead is drawn, dictating the move MENACE plays. This continues until the game has finished. The outcome of the game then determines the reinforcement MENACE receives. If MENACE won the game, the beads chosen from the matchboxes used in the game, are added to. If MENACE lost the game, they are decremented (Michie, 1961).

Michie successfully trained his MENACE models to play as competently as human players. When training against a human player, MENACE was able to draw most of the time after about 250 games (Michie, 1961). Michie also experimented with different methods for training including putting two MENACE machines against each other which resulted in both learning the optimal strategy at around 600 games (Michie, 1961). MENACE was able to learn how to consistently beat a player always choosing random moves after about 300 games (Michie, 1961).

3 Problem Analysis

Chapter 3 explores MENACE in more detail. Section 3.1 reaffirms the suitability of MENACE for fulfilling the goal of this project. Section 3.2 describes MENACE's environment. Section 3.3 formally defines MENACE by showing how it can be expressed as an MDP. Section 3.4 describes the optimal policy in Noughts and Crosses. Section 3.5 discusses the various approaches to implementing and training MENACE.

3.1 Suitability of MENACE

3.1.1 Noughts and Crosses

Given the aspiration of this project is to build an informative website aimed at newcomers to the field of AI, Noughts and Crosses is a suitable topic to build a Machine Learning model for. This is because Noughts and Crosses is a widely understood game played by many school children (Gibson, 1994). An advantage of studying simple games is they can be a platform for easily exploring and communicating Artificial Intelligence theory. Another advantage is that testing theorised AI solutions for basic games can be done using limited hardware as Donald Michie proved by building MENACE initially from matchboxes (Michie, 1961). Therefore, a Machine Learning model trained to play Noughts and Crosses should be accessible and easy to demonstrate and understand.

3.1.2 MENACE

Around the same time as Michie's MENACE, Arthur Samuel wrote a similarly ground-breaking program to play Checkers "capable of easily defeating novice players" (Schaffer, 1997). However, Checkers is a large game with a speculated "10^{40} choices of moves" (Samuel, 1959) and Samuel's Machine Learning algorithm is not straightforward for those unfamiliar with the subject. In 1961, MENACE was a demonstration of the capabilities and possibilities of Machine Learning which used extremely limited hardware and worked on a comparatively simple game. This makes MENACE more suitable for the purposes of this project which aspires to explain Machine Learning principles to newcomers in the field.

3.2 Noughts and Crosses

Noughts and Crosses is the game where two players take turns to fill a 3 by 3 grid with noughts or crosses until either player has achieved 3 in a row (Clarkson, 2008). In MDP terms, Noughts and Crosses can be thought of as an environment where the states consist of a 3 by 3 grid of tiles. Tiles can be empty or contain a nought or a cross. The set of all environment states is every possible board position in a Noughts and Crosses game. Noughts and Crosses games start with the empty board position i.e., all tiles in the grid are empty. The players can be thought of as agents. One player has the symbol cross and their set of actions from a given state or board position is to place a cross in any of the tiles in the Noughts and Crosses grid that are empty. The noughts player is the same, except they play noughts instead of crosses and their turn occurs after the cross player. To win the game and receive a return, a player must achieve a board position in which all three tiles from the configurations described below contain that players symbol before their opponent wins.

1. Top row
2. Middle row
3. Bottom row
4. Left column
5. Middle column

6. Right column
7. Diagonal line from top left corner to bottom right corner
8. Diagonal line from top right corner to bottom left corner

3.3 MENACE as a Markov Decision Process

MENACE can be thought of as the agent in a MDP (discussed in section 2.5). MENACE's environment is a Noughts and Crosses board.

- In any given game the set of all board positions that MENACE can possibly be presented with is the set of environment states S. Across two games where MENACE starts as the first player in the first game and then the second player in the second game, this is the set of all Noughts and Crosses Board Positions.
- The set of moves that MENACE can play is the set of actions the agent can take A.
- The set of rewards MENACE can receive via incrementing or decrementing beads when a Noughts and Crosses game ends is the set of rewards R.

Noughts and Crosses is an episodic task and MENACE receives reinforcements retrospectively when each game has finished. In Donald Michie's first iteration of MENACE, three beads were awarded to each matchbox used in the game for a win and one for a draw (Michie, 1961). In an MDP this can be expressed as a reward of $r_t = 3$, in games that MENACE wins. The expected return would be:

$$G_{t=0} = 3T \tag{3.1}$$

Where T is the number of moves required to win a game ($1 < T < 5$ for a game MENACE starts; $1 < T < 4$ where MENACE is the second player). $G_{t=0}$ is the expected return from the first move that MENACE plays.

Every matchbox is associated with a single board position. Matchboxes contain beads of different colours (types). Each bead colour represents one valid move that can be made from the matchbox's associated position. Therefore, a given matchbox will contain as many different bead colours as there are legal moves from the associated board position. More formally, each bead type (colour) represents one of the actions, a, that can be made from state s_t. All possible bead types in a given matchbox comprise the set A_t for state s_t.

To make a move from a given state, MENACE selects one bead at random from the matchbox. Once the reinforcement (reward) process begins, there will start to be a different number of beads of each type (colour) in each matchbox, and this dictates the probability of any given bead type (move / action) being chosen in future games. The more beads of a given type exist, the more likely the corresponding move will be made. In this way, MENACE's matchboxes and beads define its policy π.

The probability of action $a \in A_t$ given state s_t is:

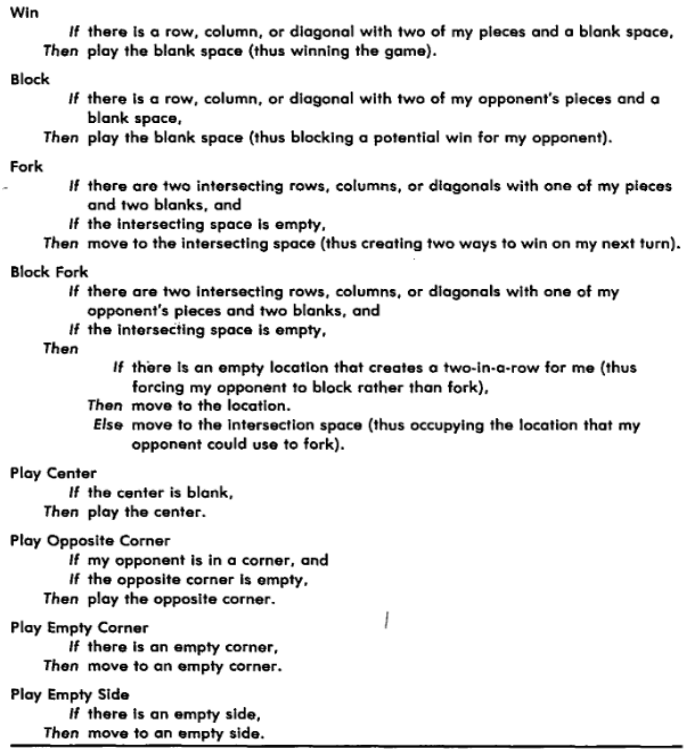
$$\pi(a \in A_t | s_t) = b_a / n \tag{3.2}$$

where b_a is the number of beads of a specific colour in a matchbox and n is the total number of beads in the matchbox.

3.4 Noughts and Crosses Optimal Strategy

Noughts and Crosses is a simple game. After considering board symmetry there are 23,129 unique games (Schaefer, 2002). Furthermore, if both players use the optimal strategy, the game always results in a draw (Weisstein, 2022). In fact, it is relatively easy to produce a computer program which

follows a set of rules dictating expert play (Crowley & Siegler, 1993). In an MDP, this would be the optimal policy. The aim of MENACE is to learn this policy without it being explicitly programmed. The following diagram is Crowley and Siegler's optimal algorithm:



Win
If there is a row, column, or diagonal with two of my pieces and a blank space,
Then play the blank space (thus winning the game).

Block
If there is a row, column, or diagonal with two of my opponent's pieces and a blank space,
Then play the blank space (thus blocking a potential win for my opponent).

Fork
If there are two intersecting rows, columns, or diagonals with one of my pieces and two blanks, and
If the intersecting space is empty,
Then move to the intersecting space (thus creating two ways to win on my next turn).

Block Fork
If there are two intersecting rows, columns, or diagonals with one of my opponent's pieces and two blanks, and
If the intersecting space is empty,
Then
If there is an empty location that creates a two-in-a-row for me (thus forcing my opponent to block rather than fork),
Then move to the location.
Else move to the intersection space (thus occupying the location that my opponent could use to fork).

Play Center
If the center is blank,
Then play the center.

Play Opposite Corner
If my opponent is in a corner, and
If the opposite corner is empty,
Then play the opposite corner.

Play Empty Corner
If there is an empty corner,
Then move to an empty corner.

Play Empty Side
If there is an empty side,
Then move to an empty side.

Figure 3.3 Crowley and Siegler's algorithm for expert Noughts and Crosses play (Crowley & Siegler, 1993).

It is important to mention that the best opening move is to play a corner as this minimises the opponent's chance to avoid losing if the opening player continues to play optimally (Gardner, 1988). Crowley and Siegler's algorithm favours playing the centre because if the opponent is not an optimal player, then the centre gives more options (Kutschera, 2022).

3.5 Data and Training MENACE

3.5.1 Typical Machine Learning Approach

Section 2.4 outlines the typical machine learning model development process whereby some data is gathered and prepared for training and testing various algorithms. By the end of this process, the best model will be chosen as the final product and deployed to real world scenarios (Alzubi, et al., 2018). This is a viable direction for this project. There are plenty of Noughts and Crosses datasets online which could be adapted to fit with an implementation of MENACE. In addition to this, there are a myriad of machine learning models capable of learning to play Noughts and Crosses (e.g., Artificial Neural Networks (Morley, 2009)) which, given the same dataset, could be compared with MENACE.

3.5.2 Computing All Games

Another approach for training MENACE is that it would be possible to create a dataset containing every Noughts and Crosses game. Arthur Samuel's Checkers reinforcement learning program took thousands of games to become proficient against a human player (Samuel, 1959). This training strategy was suitable for Checkers because of the impractical number of unique possible games

there are. In contrast, Noughts and Crosses is relatively simple with a computationally manageable number of unique games. A dataset containing every Noughts and Crosses game could be fed to MENACE several times. In theory this would teach MENACE to be an extremely competent player.

3.5.3 Reinforcement Learning Approach

Reinforcement Learning algorithms learn by receiving reinforcement as they interact with the environment. In episodic tasks, agents play many games and over time learn a policy which maximises an expected return (Sutton & Barto, 2020). This is the training process which MENACE will undergo.

In Noughts and Crosses, the optimal strategy is well understood. As MENACE trains, it will gradually learn the policy which matches this strategy.

3.5.4 Michie's Approach

When Michie built MENACE out of matchboxes, he trained it against a human opponent (Michie, 1961). This resulted in MENACE learning to consistently draw against a human opponent in about 220 games for its maiden tournament. However, this training process took 16 hours and is not practical for humans (Scroggs, 2015).

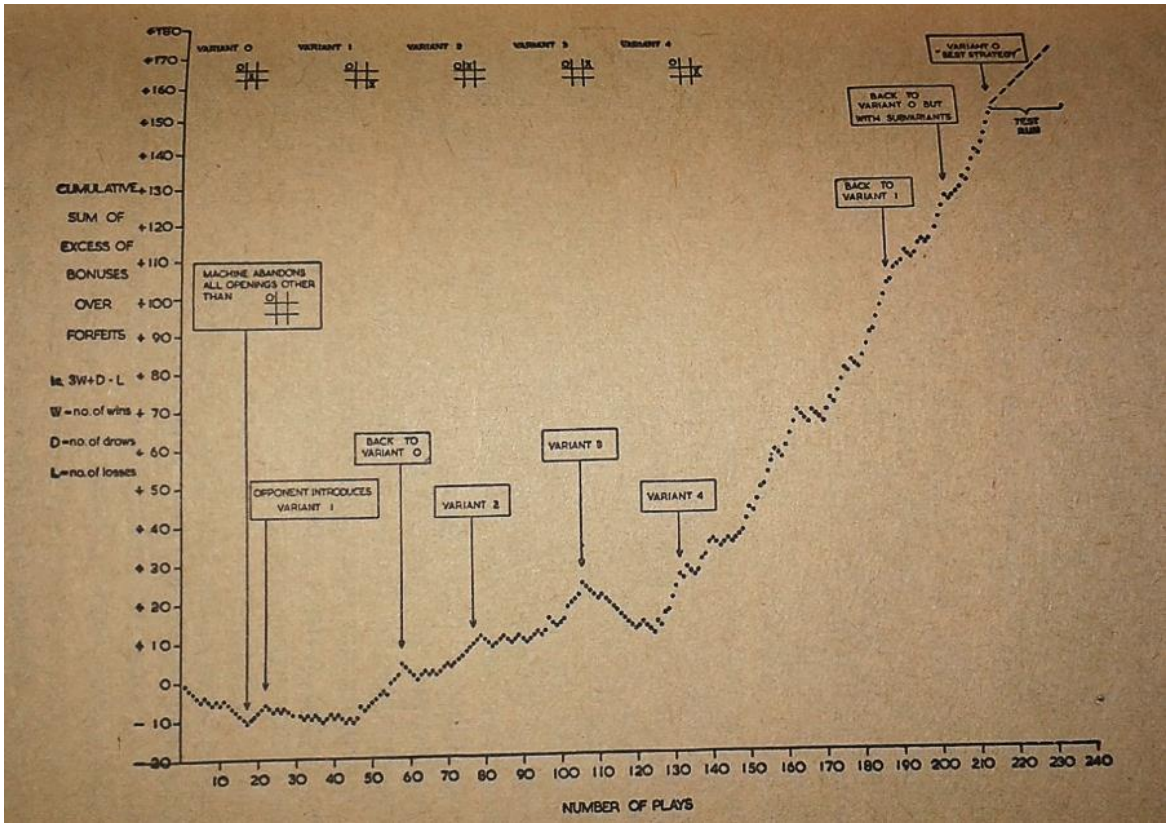


Figure 3.4 The progress of MENACE'S maiden tournament against a human opponent (Michie, 1961). Image retrieved from "Mr Scroggs Blog" (Scroggs, 2015).

Michie later simulated MENACE on a Pegasus 2 computer and automated its training by programming new players for MENACE to play against. This included a random move player and an expert player (Michie, 1961). Michie experimented with training MENACE against these various player types. As a benchmark performance to measure MENACE against, Michie noted that when both players in a Noughts and Crosses game play random moves, the opening player wins roughly two games in three (Michie, 1961). If MENACE can perform better than this, it is a sign of progress.

Michie observed that when MENACE trained against a random move player, it took a great deal of time for MENACE to achieve an improved skill level. After this process MENACE was still easy for a human to beat. He attributed this to MENACE jumping to premature conclusions and learning suboptimal behaviours which were difficult to unlearn (Michie, 1961). Michie also tried training MENACE against an expert player. However, he explains that a drawback of this is MENACE only explores a restricted subtree of the whole game tree (Michie, 1961). Michie's experiments conclude by asserting that playing two MENACE instances against each other yields near perfect play by both after a few hundred games (Michie, 1961).

The reinforcement function is also an aspect of training that can be experimented with. Michie found that a sensible function reinforced moves that are closer to the end of the game more dramatically than those at the start (Michie, 1961). This ensures that MENACE learns how to finish a game when in a winning position or block an opponent's winning position more quickly.

4 Design and Implementation

4.1 Overview

Two products were developed for this project. The first is a C# class library implementing MENACE. This library comes with Noughts and Crosses board representation, Noughts and Crosses game playing capability and other AI types, namely: random move player and optimal move player. Importantly, it also comes with a method of training MENACE and evaluating its performance. It also comes with an entry point for running in the console. This is primarily for debugging and was not intended to be seen by users. The class library's main purpose is to be used for the second product: the ASP.NET MVC web application.

The web application is designed to be a user interface for the class library. It allows users to create, interact with, and train MENACE instances. It also persists user's MENACE models in a database for comparison with other models made on the website. The website also provides information about how MENACE works to show how machines can be trained to win Noughts and Crosses.

4.2 Scope and Requirements

The scope of this project is defined by the aims and objectives in section 1.2. Both products are built to fulfil the aim of designing and building an explainable AI model (MENACE) that learns to play a game of noughts and crosses. The class library covers the designing and building of MENACE while the web application explains and presents the model.

The fulfilment of objectives from section 1.2 is also spread across both products. The class library covers the following objectives:

2. Design and develop a system allowing users to play Noughts and Crosses against a computer.
3. Create training dataset.
4. Implement, train, and test MENACE and various algorithms.
5. Evaluate and compare all algorithm performances.

The web application covers the following objectives:

2. Design and develop a system allowing users to play noughts and crosses against a computer.
6. Design and develop a user interface.
7. Conduct a user evaluation of the user interface.

4.2.1 Functional Requirements

To ensure the best chance of covering the full scope of the project, here are the functional requirements for both products.

1. Users can play Noughts and Crosses against MENACE.
2. Users can choose whether MENACE goes first or second.
3. Users can train MENACE against random players, optimal players, and other MENACE players.
4. The matchbox MENACE used to play its turn is displayed for users to see.
5. Users can play any existing MENACE instance or create a new one.
6. There are many reinforcement methods and users can choose which one MENACE uses.

7. Users can see all the MENACE instances in the database with how many games they won, drew, or lost.
8. Users can view the details of any existing MENACE.
9. Users can see the full list of matchboxes in any existing MENACE from a details page.
10. Users can see a training history of any existing MENACE from a details page.
11. Users can see a visual representation of any existing MENACE's learning progress.
12. Users can train any existing MENACE using a random, optimal, or MENACE player from a details page.
13. Research and Information about MENACE is displayed on an info page.
14. Users can navigate to all pages from any page.
15. The website is hosted and can be accessed by many users concurrently.

4.2.2 Non-Functional Requirements

To ensure the best chance of covering the full scope of the project, here are the non-functional requirements for both products.

1. The MENACE implementation is efficient with no component of it exceeding a time complexity of $O(n^2)$.
2. The library is easy to incorporate in the web application i.e., easy to map to a database, easy to load and manipulate objects etc.
3. The library is reliable.
4. The library is easily modifiable.
5. The website is intuitive enough for users to comfortably navigate the web application.
6. The website is informative enough that users with limited AI knowledge are aware of how MENACE works.
7. The visual style of the website is exciting and interesting.
8. The website has a reasonable response time.
9. The website does not store any personal data.

4.3 Class Library Design

4.3.1 Training Approach

When it came to designing a method for training MENACE, several approaches were considered. Firstly, the approaches mentioned in Donald Michie's paper included training MENACE against a human opponent as well as various automated opponents. This approach has many advantages for fulfilling the aspiration of this project which is to build a website designed to be informative for newcomers to AI. A website allowing users to play against MENACE themselves would enable real time observation of how MENACE is learning. Furthermore, there can be options to pit MENACE against user chosen opponents over some number of iterations for rapid training. This training method would be highly interactive and informative as well as conceptually easy to understand.

Another approach is to use a ready-made dataset. There are numerous datasets available online such as Sizhe Yuen's dataset used for a Noughts and Crosses Neural Network, available on GitHub (Yuen, 2017). This dataset contains thousands of Noughts and Crosses games. It could be modified for use in MENACE and each game could be iteratively fed to MENACE for training. This approach would also make it easy to compare MENACE with other Machine Learning models on the same dataset (see section 2.4 and 3.5). This method is perfectly viable for training MENACE, but it is not ideal for use in a website designed to demonstrate how machines can learn. This is because adapting a dataset to give to MENACE and playing against the finished product goes against the spirit of being

able to observe and interact with a Reinforcement Learning algorithm learning in real time. Furthermore, contemporary Machine Learning algorithms are advanced and difficult to present in a user-friendly way and therefore unsuitable for newcomers to the field of AI.

Another approach is to train MENACE over several iterations of a dataset containing every unique Noughts and Crosses game. This would be good for ensuring MENACE learns to cope with every contingency, making it a competent Noughts and Crosses player. However, this approach has the same shortcomings as using a ready-made dataset i.e., it would be less informative in a website. Furthermore, building a model that is only capable of learning by exploring the entire game tree is extraordinarily inefficient, especially given Michie's MENACE learnt to play competently in 220 games in its first tournament (Michie, 1961). This approach could not be scaled to more complex problems such as Checkers with a computationally impractical "10^{40} choices of moves" (Samuel, 1959). This renders the approach somewhat irrelevant. In addition, Reinforcement Algorithms are often used when it is too difficult to provide an adequate dataset for the problem. This is because providing reinforcement signals which teach an agent how to behave is generally easier than providing a dataset of labelled examples (J.Russell & Norvig, 2021). This training approach would counteract this advantage.

It was decided that replicating Michie's approach as closely as possible was the best course of action. This is because it is ideal for use in an informative website, as it is conceptually straight forward and would allow users to engage with MENACE's training. Furthermore, this approach makes the most sense for training a Reinforcement Learning algorithm.

4.3.2 Technologies

The aspiration of this project is to build MENACE in an explainable and presentable way. The technologies used for this task were chosen with this in mind.

4.3.2.1 Object Oriented Programming and C#

C# is an object-oriented programming language used for creating a wide range of applications (Microsoft, 2022). It has been chosen for implementing MENACE in a class library for the following reasons:

1. Object oriented programming languages have many useful features for implementing something like MENACE. This includes classes for encapsulating MENACE objects, board positions, reinforcement algorithms, etc. Furthermore, polymorphism is useful for grouping similar objects in an inheritance structure such as the various player types i.e., human, MENACE, etc.
2. Object oriented architectures make it easy to abstract and compartmentalize (Booch, 1986). This is ideal for implementing MENACE because it will be easy to present on a website and for allowing users to experiment with components and parameters of MENACE.
3. C# is part of the ASP.NET ecosystem which means a MENACE class library written in C# will be compatible with many tools for building the type of application this project aspires to build. Namely these tools are the MVC Web Application Framework, Entity Framework, Razor, Microsoft SQL etc.

4.3.3 Class Diagram

The implementation of MENACE as well as its training and game playing capabilities has many components to it. Figure 4.1 below depicts the structure of the class library which contains the following key classes:

1. **AIMenace**
2. **AIOptimalMove**
3. **AIRandomMove**
4. **Bead**
5. **BoardPosition**
6. **Coordinate**
7. **Game**
8. **GameHistory**
9. **Matchbox**
10. **Player**
11. **PlayerHuman**
12. **PlayerMenace**
13. **PlayerOptimal**
14. **PlayerRandom**
15. **Reinforcement-Incremental**
16. **Turn**

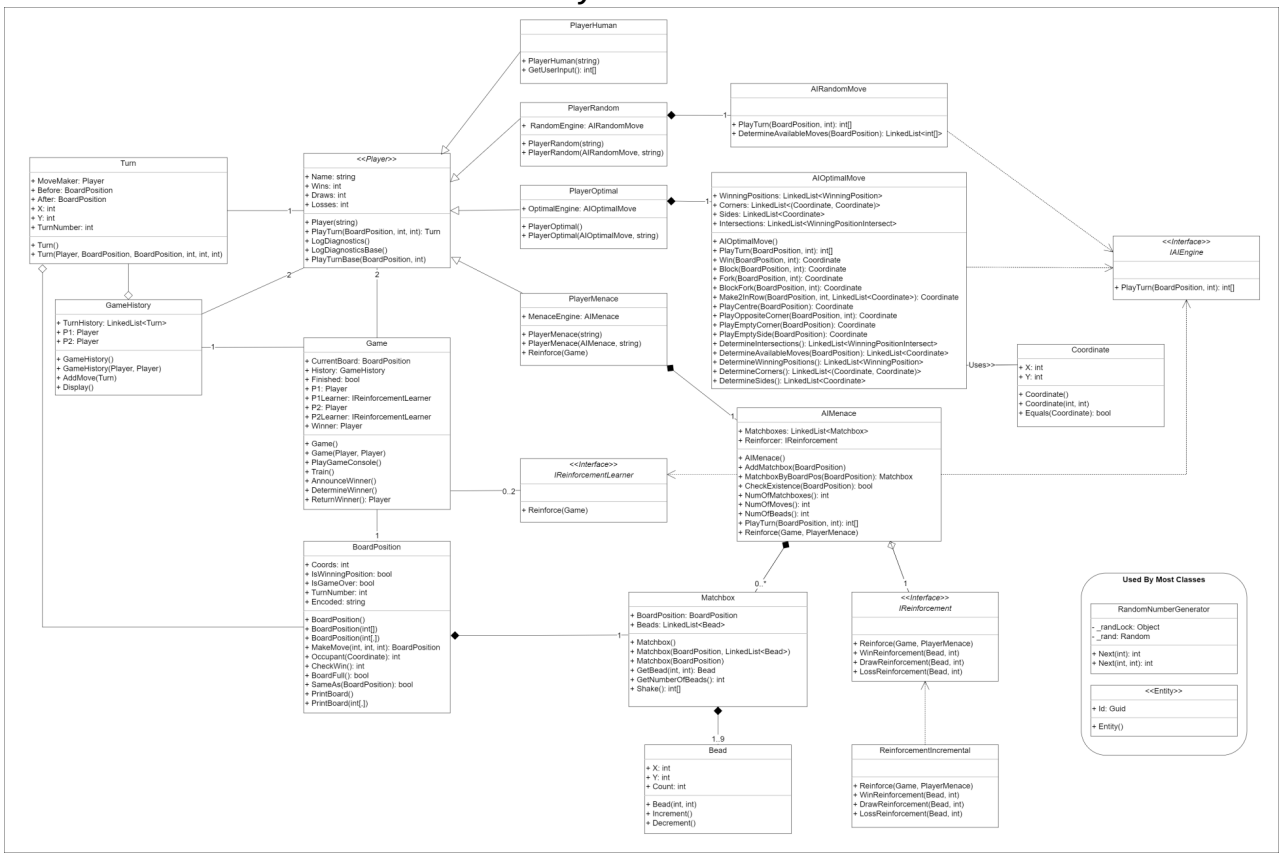


Figure 4.1 Class Structure of Menace Class Library (See appendix A for a better view).

Reinforcement Learning Algorithms, need an environment to interact with. **BoardPosition** represents a Noughts and Crosses game state. It contains a 3x3 two-dimensional array mimicking a Noughts and Crosses grid. At each index of the array, there is an integer denoting the occupant of the position (-1 for Crosses, 0 for vacant, 1 for Noughts).

The **Game** class enables two **Player**s to play Noughts and Crosses. It uses **BoardPosition** instances to keep track of game state. When a **Player** plays their turn, the **Game** class updates the game state with a new **BoardPosition**. This process repeats until there is a winning **Player** or the current **BoardPosition** instance is full i.e., a draw.

To avoid the **Game** class being concerned with **Player** types, an abstract **Player** class is used. The **Player** class specifies that a **PlayTurn()** method must be implemented. The **PlayerMenace** class inherits from the Player class and has an **AIMenace** instance dictating the moves it plays for given **BoardPosition**s.

AIMenace is the implementation of MENACE. It has a list of **Matchbox** instances as well as a reinforcement algorithm. The library supports the interchanging of reinforcement algorithms by having an **IReinforcement** interface. At this point **ReinforcementIncremental** handles reinforcing a **PlayerMenace** which can have one of two reward functions: **ReinforceWinPlusTurnLossMinusTurn** or **ReinforceThreePerWinOnePerDraw**.

Matchbox instances are identifiable by a **BoardPosition**. They also have a list of **Bead** objects. **Bead** objects have **X** and **Y** coordinates representing an available move that can be played. In addition to this, **Bead**s have a count to represent the abundance of that **Bead** in its **Matchbox**. This is later used for calculating which move should be played. When a **Matchbox** is initialised, it calculates which moves are available from its **BoardPosition** and creates a **Bead** for each one. When a **Bead** is initialised, it has a count of 1.

When an **AIMenace** plays a turn, it finds the **Matchbox** with the **BoardPosition** matching the **BoardPosition** of the current game state. If the **AIMenace** does not have the **Matchbox** representing the games current **BoardPosition** it creates a new one. It then determines what move it should play by calling the **Matchbox.Shake()** method. This method randomly selects a **Bead** from the **Matchbox** and plays the move represented by the **Bead**s coordinates. **Bead**s with higher counts are more likely to be chosen than **Bead**s with lower counts.

When a **Game** ends and one of the **Player**s is a **PlayerMenace**, the **PlayerMenace**'s **AIMenace** calls upon its reinforcement algorithm. If the **PlayerMenace** won the **Game**, the reinforcement algorithm increases the **Bead.Count**s of the **Bead**s that were chosen to dictate **AIMenace**'s moves during the **Game**. If the **AIMenace** is confronted with these **BoardPosition**s again it is more likely to select these **Bead**s. If the **PlayerMenace** lost the **Game**, then the **Bead.Count**s are decreased.

4.4 Class Library Implementation

4.4.1 Game Class

Game is a class designed to allow two players to play a game of Noughts and Crosses. It has two ways of doing this. Firstly, the **PlayGame()** method is designed for when there is a human player involved. After every move by a player, the board is printed out in console so human players can see the new board state. The other way of playing a game is by using the **Train()** method which is designed for games between two AI. It does not print any stage of the game out except from the result at the end. This is useful for training Machine Learning algorithms such as MENACE because the console won't clog up with unimportant information and should run quicker.

```
79      public void Train()
80      {
81          if (Finished) throw new Exception("Cannot play a game that has already Finished!");
82
83          // Game Loop
84          while (true)
85          {
86              // Player 1 turn
87              Turn p1Turn = P1.PlayTurn(CurrentBoard, GameSymbol.MapSymbolToInt("X"), CurrentBoard.TurnNumber);
88
89              CurrentBoard = p1Turn.After;
90
91              History.AddMove(p1Turn);
92
93              // Check Game End
94              if (p1Turn.After.IsGameOver) break;
95
96              // Player 2 Turn
97              Turn p2Turn = P2.PlayTurn(p1Turn.After, GameSymbol.MapSymbolToInt("O"), p1Turn.After.TurnNumber);
98
99              CurrentBoard = p2Turn.After;
100
101             History.AddMove(p2Turn);
102
103             // Check Game End
104             if (p2Turn.After.IsGameOver) break;
105         }
106
107         Finished = true;
108
109         DetermineWinner();
110
111         //AnnounceWinner();
112
113         // Apply Reinforcements
114         if (P1Learner != null) P1Learner.Reinforce(this);
115
116         if (P2Learner != null) P2Learner.Reinforce(this);
```

Figure 4.2 **Game.Train()**

4.4.2 Board Position Class

BoardPosition is a class designed to encapsulate board positions so they can be manipulated by other components of the program. The internal representation of the board is a 3 by 3 array storing -1s to represent crosses, 1s to represent noughts and 0s to represent empty board positions. This format was chosen because it makes it easier to refer to positions on the board with coordinates and it is also a minimalist data structure. It was also thought that this was probably a common representation for Noughts and Crosses.

The most important method in the **BoardPosition** class is the **MakeMove()** method which takes an **X** and **Y** coordinate as well as the **Player** who is making a move. It then creates and returns a new **BoardPosition** instance which has the same board state except with the added move. Since Board States are not static objects, it was also important to include the **SameAs()** method to check if two **BoardPosition** instances refer to the same board state. Other methods in **BoardPosition** include methods for printing the board in the console, as well as checking if the board is full or if the board is a winning position. **Occupant()** is a method that returns the player at a given coordinate.

```
51
52      // To make a move create a new board position and return it to whoever wants it...
        3 references
53      public BoardPosition MakeMove(int x, int y, int turn)
54      {
55          int[,] newposition = new int[,] { { Coords[0, 0], Coords[0, 1], Coords[0, 2] }, { C
56          newposition[x, y] = turn;
57          return new BoardPosition(newposition);
58      }
```

Figure 4.3 **BoardPosition.MakeMove()**

4.4.3 Game History Class

The **GameHistory** class is designed to store information about a Noughts and Crosses game's history. It stores the **Player**s involved with the game as well as the **TurnHistory** which is a list of **Turn** objects. It also includes methods to add turns to the **TurnHistory** as well as display all the moves played.

```
        3 references
 9      public class GameHistory
10      {
            5 references
11          public LinkedList<Turn> TurnHistory { get; }
            2 references
12          public Player P1 { get; }
            2 references
13          public Player P2 { get; }
14
            1 reference
15          public GameHistory(Player p1, Player p2)
16          {
17              TurnHistory = new LinkedList<Turn>();
18              P1 = p1;
19              P2 = p2;
20          }
```

Figure 4.4 **GameHistory***: properties and constructor*

The **Turn** class is designed to store information about a single turn in a Noughts and Crosses game. It stores the **Player** who played the turn, the **BoardPosition** before the move and after, the **X** and **Y** coordinate of the move played, and the turn number.

```
7   namespace Noughts_and_Crosses
8   {
        18 references
9       public class Turn
10      {
            4 references
11          public Player MoveMaker { get; }
            3 references
12          public BoardPosition Before { get; }
            6 references
13          public BoardPosition After { get; }
            4 references
14          public int X { get; }
            4 references
15          public int Y { get; }
            7 references
16          public int TurnNumber { get; }
17
```

Figure 4.5 **Turn** *class*

4.4.4 Player Classes

Originally the **Game** class had methods for games where there were two humans involved, or a human and AI etc. However, it became apparent that it would be useful to abstract MENACE and human players into **Player** objects. This meant the **Game** class no longer needed to be concerned with which type of players were involved. Another advantage of abstracting players is that they could track their progress in diagnostics such as the number of wins, draws and losses. For MENACE players it could also track the number of matchboxes it has as well as the number of **BoardPositions** it recognises and overall, how many beads it had at any given time.

```
9       public abstract class Player
10      {

11          public string Name { get; }

12          public int Wins { get; set; }
            5 references
13          public int Draws { get; set; }
            5 references
14          public int Losses { get; set; }
15
```

Figure 4.6 **Player** *class*

The class inheritance structure of **Player**s is as follows. All **Player**s have a name and properties to track the number of games they have won, drawn, or lost. In addition, all **Player**s must implement methods to play a turn, output diagnostics to the console and apply some reinforcement method to themselves. The **PlayerHuman** class inherits from the **Player** base class. **Reinforcement** is irrelevant for human players so this is left unimplemented. All AI **Player**s inherit from the **Player** base class and in addition to the base properties, each AI **Player** has an AI engine. Some AI engines do not learn such as **AIRandomMove**, and **AIOptimalMove**. Therefore, the reinforcement methods for

these classes are left unimplemented. Finally, the **PlayerMenace** class has an AI engine called **AIMenace** which implements MENACE. **PlayerMenace** implements the reinforcement method.

4.4.5 AI Classes

It became necessary to build an inheritance structure for the various Noughts and Crosses AIs so that **Game**s did not need to be concerned with what type of AI they were using. As long as all AI types implemented the **PlayTurn()** method, the internal components of AIs could vary significantly.

The most basic type of AI is the **AIRandomMove** class. This simply determines the available moves from a **BoardPosition** and randomly chooses one. This was useful for MENACE development because there was an issue during training where it would only learn how to play against the type of opponents it was exposed to. When it trained against the **AIOptimalMove** player it would learn how to deal with competent opponents but play terribly against opponents who play erratically. Ironically at this point in development it was a better strategy to play badly against MENACE than it was to play well.

```
1 reference
9   public class AIRandomMove : AI
10  {
        3 references
11      public override int[] PlayTurn(BoardPosition boardPos, int turn)
12      {
13          LinkedList<int[]> available = DetermineAvailableMoves(boardPos);
14
15          // return a random move from those available
16          return available.ElementAt(RandomNumberGenerator.Next(available.Count));
17      }
```

Figure 4.7 **AIRandomMove** *Class*

4.4.6 Optimal Move Player

This type of AI became important for training MENACE because it taught MENACE how to play well. The optimal algorithm is based on Crowley and Siegler's algorithm for expert Noughts and Crosses play (Crowley & Siegler, 1993). It attempts to play the first available move from the following list in order.

1. Win by completing three in a row.
2. Block a win by blocking an opponent from getting three in a row in the next turn.
3. Create a fork i.e., a position which creates two opportunities to complete three in a row in the next turn. The opponent can only block one of them on their turn.
4. Block a fork in the most advantageous way i.e., block the opponent from creating two opportunities to complete three in a row in the next turn. If possible, do this in a way that makes progress towards making three in a row.
5. Play the centre position of the board. If both players are playing optimally, it is actually more advantageous to play a corner position as the opening move (Kutschera, 2022). However, Crowley and Siegler determined that against players who don't play optimally, the centre position gives more options long term (Crowley & Siegler, 1993).
6. Play the opposite corner to the opponent's corner.
7. Play any empty corner.
8. Play any empty side.

(Crowley & Siegler, 1993)

This algorithm was surprisingly tricky to implement because determining how to code instructions for forking or blocking a fork in the most advantageous way is not straight forward. It relied on calculating all the ways winning positions could intersect with each other as well as calculating opposite corners etc.

```
175
176     // FORK
177     // If there are two intersecting rows, columns, or diagonals with one of my pieces and two blanks, and
178     //  If the intersecting space Is empty,
179     //      Then move to the intersecting space (thus creating two ways to win on my next turn).
        1 reference
180     public LinkedList<Coordinate> Fork(BoardPosition boardPos, int player)
181     {
182         LinkedList<Coordinate> possibleMoves = new LinkedList<Coordinate>();
183
184         foreach (var Intersect in Intersections)
185         {
186             // check if pos1 meets criteria
187             int empty1 = 0;
188             int filled1 = 0;
189             foreach (Coordinate i in Intersect.WinPos1.Coordinates)
190             {
191                 if (boardPos.Occupant(i) == player) filled1++;
192                 if (boardPos.Occupant(i) == 0) empty1++;
193             }
194
195             // check if pos2 meets criteria
196             int empty2 = 0;
197             int filled2 = 0;
198             foreach (Coordinate i in Intersect.WinPos2.Coordinates)
199             {
200                 if (boardPos.Occupant(i) == player) filled2++;
201                 if (boardPos.Occupant(i) == 0) empty2++;
202             }
203
204             // if criteria met, and intersecting coord empty return it
205             if (empty1 == 2 && filled1 == 1 && empty2 == 2 && filled2 == 1 && boardPos.Occupant(Intersect.IntersectCoord) == 0)
206             {
207                 possibleMoves.AddLast(Intersect.IntersectCoord);
208             }
209         }
210
211         if (possibleMoves.Count > 0) return possibleMoves;
212         return null;
213     }
```

Figure 4.8 **`AIOptimalMove.Fork()`**

4.4.7 MENACE

The **`AIMenace`** class is an implementation of MENACE based on Donald Michie's paper from 1961. It has a list of **`Matchbox`** objects as well as a **`Reinforcement`** object. It also has methods to Add **`Matchbox`**es, fetch and return **`Matchbox`** instances from a given **`BoardPosition`**, check the existence of **`Matchbox`**es, and return the number of **`Matchbox`**es, moves, and **`Bead`**s the instance has. MENACE's implementation of the **`PlayTurn()`** method (inherited from its parent class) checks to see if it has any **`Matchbox`**es for the given board position. If the **`Matchbox`** exists, it uses the **`Matchbox`**es **`Shake()`** method to obtain the coordinate it should play.

```
86      public override int[] PlayTurn(BoardPosition boardPos, int turn)
87      {
88          // If Menace hasn't encountered this position before
89          if (!CheckExistence(boardPos))
90          {
91              AddMatchbox(boardPos);
92              Console.WriteLine("Adding new position to MENACE");
93              boardPos.PrintBoard();
94          }
95
96          Matchbox box = MatchboxByBoardPos(boardPos);
97
98          return box.Shake();
```

Figure 4.9 **`AIMenace.PlayTurn()`**

4.4.8 Matchbox Class

One of the defining components of MENACE is its **Matchbox**es. This has been implemented as a class with an identifying **BoardPosition** object as well as a list of **Bead** objects representing the available moves that can be played and their probability of MENACE choosing them. The **Matchbox** class also has the method **GetBead()** which returns a **Bead** instance from a given coordinate. There is also a **GetNumberOfBeads()** method which returns the number of beads in the matchbox.

The most important method is **Shake()** which returns a coordinate to play. This relies on the list of **Bead** objects which store a coordinate to play as well as the abundance of this **Bead** in the **Matchbox** (its count). When selecting a move to play, the **Matchbox** chooses a random number between 1 and the sum of bead counts from all **Bead**s in the **Matchbox**. This number then dictates which **Bead** is selected. If the **Bead** has a count significantly higher than the rest, it is significantly more likely the coordinates this bead represents will be played.

```
1 reference
62  public int[] Shake()
63  {
64      // Generate random number
65      int randomNumber = RandomNumberGenerator.Next(GetNumberOfBeads());
66
67      // Determine which move the random number points to
68      int count = 0;
69
70      foreach (Bead move in Beads)
71      {
72          count += move.Count;
73
74          if (randomNumber < count) return new int[] {move.X, move.Y};
75      }
76
77      throw new Exception("Something went wrong while MENACE was picking it's next move.");
78  }
```

Figure 4.10 ***Matchbox.Shake()***

4.4.9 Reinforcement Classes

The other defining component of MENACE is the reinforcement algorithm. This algorithm determines how MENACE learns to play Noughts and Crosses. This is done by changing the **Bead** counts in the **Matchbox**es used to play a **Game** based on whether MENACE won, drew, or lost. Donald Michie's paper on MENACE suggests various ways of doing this. For example, he mentions that turns played towards the end of a game could be rewarded or punished more than turns played at the start (Michie, 1961). This makes sense because it quickly reinforces winning moves and punishes moves that immediately lose MENACE the game.

Since reinforcement was an aspect of MENACE that was likely to be experimented with it was important to make it interchangeable. This is so various reinforcements could be implemented and swapped into MENACE. **ReinforcementIncremental** handles the reinforcement of a **PlayerMenace** given a **GameHistory**, using a reward function. It does this by iterating through the **Turn**s in the **GameHistory** and calling on a reward function to change the number of **Bead**s in the **Matchbox** used by the **PlayerMenace** during that turn based on whether the **PlayerMenace** won, drew, or lost.

Currently two reward functions have been implemented. **ReinforceWinPlusTurnLossMinusTurn** adds or subtracts the number of **Bead**s equal to the turn number from which the **Bead** was played in. **ReinforceThreePerWinOnePerDraw** adds three **Bead**s a win, one for a draw and zero for a loss.

```
public class ReinforcementIncremental
{
    4 references
    public static void Reinforce(GameHistory g, PlayerMenace menace)
    {
        // Ensure menace played in the game given
        if (g.P1 != menace && g.P2 != menace) throw new Exception("Reinforcement Error: it would seem MENACE did not play in the game given");

        var rewardFunction = ReinforcementRewardFunction.GetRewardFunction(menace.ReinforcementType);

        // for each turn in the game's history where menace played a turn
        foreach (Turn t in g.Turns)
        {
            if (t.TurnPlayer == menace)
            {
                // Fetch the box Menace used to play the turn
                Matchbox boxUsed = menace.MenaceEngine.MatchboxByBoardPos(t.Before);

                // Fetch the bead that Menace selected from the boxUsed by checking coordinates played that turn
                Bead beadUsed = boxUsed.GetBead(t.X, t.Y);

                // Determine reinforcement type based on the game outcome

                if (g.Winner == null)
                {
                    boxUsed.Draws++;

                    rewardFunction.RewardDraw(t, boxUsed, beadUsed);

                }
                else if (g.Winner == menace)
                {
                    boxUsed.Wins++;

                    rewardFunction.RewardWin(t, boxUsed, beadUsed);
                }
                else if (g.Winner != menace)
                {
                    boxUsed.Losses++;
```

Figure 4.11 **`ReinforcementIncremental`** *Class*

```
public class ReinforceWinPlusTurnLossMinusTurn : IReinforceRewardFunction
{
    2 references
    public void RewardDraw(Turn t, Matchbox box, Bead bead)
    {
    }

    2 references
    public void RewardLoss(Turn t, Matchbox box, Bead bead)
    {
        bead.Count = bead.Count - t.TurnNumber;


        if (bead.Count < 1)
        {
            var deficit = -bead.Count;

            foreach (Bead b in box.Beads)
            {
                b.Count += deficit + 1;
            }
        }
    }

    2 references
    public void RewardWin(Turn t, Matchbox box, Bead bead)
    {
        bead.Count += t.TurnNumber;
```

Figure 4.12 **`ReinforceWinPlusTurnLossMinusTurn`** *Class*

```
public class ReinforceThreePerWinOnePerDraw : IReinforceRewardFunction
{
    2 references
    public void RewardDraw(Turn t, Matchbox box, Bead bead)
    {
        bead.Count += 1;
    }

    2 references
    public void RewardLoss(Turn t, Matchbox box, Bead bead)
    {
    }

    2 references
    public void RewardWin(Turn t, Matchbox box, Bead bead)
    {
        bead.Count += 3;
    }
```

Figure 4.13 **`ReinforceThreePerWinOnePerDraw`** *Class*

4.4.10 Training Method

As discussed in section 4.3, it was decided to train MENACE by replicating Michie's approach. Section 3.5 explains the details of this approach. For training MENACE, Michie implemented automated opponents for playing random moves, as well as expert play. The class library encapsulates these with **`AIRandomMove`** and **`AIOptimalMove`**.

It was observed that when **`AIMenace`** is trained by playing **`AIRandomMove`** it is easily beatable by a human. This is most likely because, compared with the set of all Noughts and Crosses games, games where both players play expertly is small. **`AIRandomMove`** only plays expertly by accident, and this rarely occurs. Most games **`AIMenace`** plays against **`AIRandomMove`** are erratic. This results in **`AIMenace`** reinforcing basic strategies which would not hold up against a human or expert player. Because of the scarceness of games where **`AIRandomMove`** plays expertly, it is difficult for MENACE to unlearn the basic strategies and learn good ones. As discussed in section 3.5, this is consistent with Michie's observations.

When **`AIMenace`** trains against an **`AIOptimalMove`** it learns how to play against expert players but falters against erratic ones. This is likely because **`AIMenace`** overfits to expert play. Random players are likely to expose **`AIMenace`** to **`BoardPositions`** that it's never seen before and therefore unequipped to deal with.

Section 3.5 discusses how Michie determined that training MENACE against another instance of MENACE was the best training method (Michie, 1961). However, this project has found (by playing MENACE from a console) that this can often result in both MENACE instances learning to play the same game over and over, to avoid negative reinforcement. It is possible that this is due to an inadequate reinforcement method however it was found that a training program involving many games against all player types yields the best training results.

```
19
20      // Training for MENACE Start
21      for (int i = 0; i < 10000; i++)
22      {
23          //Console.WriteLine("training");
24          Game g = new Game(Menace1, Menace2);
25          g.Train();
26          for (int k = 0; k < 5; k++)
27          {
28              Game optiTrain1 = new Game(Menace1, Optimo1);
29              Game optiTrain2 = new Game(Optimo2, Menace2);
30              optiTrain1.Train();
31              optiTrain2.Train();
32          }
33          for (int j = 0; j < 20; j++)
34          {
35              Game rand1 = new Game(Menace1, Rando);
36              Game rand2 = new Game(Rando, Menace2);
37              rand1.Train();
38              rand2.Train();
39          }
40      }
41      // Training for MENACE End
```

Figure 4.14 Example training program for a **`PlayerMenace`**

4.5 Web Application Design

4.5.1 User Interface Design

Figures 4.15 and 4.16 show the design of the website's user interface. The website consists of a home page, a Build Menace page, and a Player Leader board page. All pages include the navigation bar at the top. This allows a user to navigate to all other pages from anywhere on the website. The home page currently lacks anything exciting, but it is intended that this will contain introductory information. For now, there is a button navigating users to the Build Menace page.

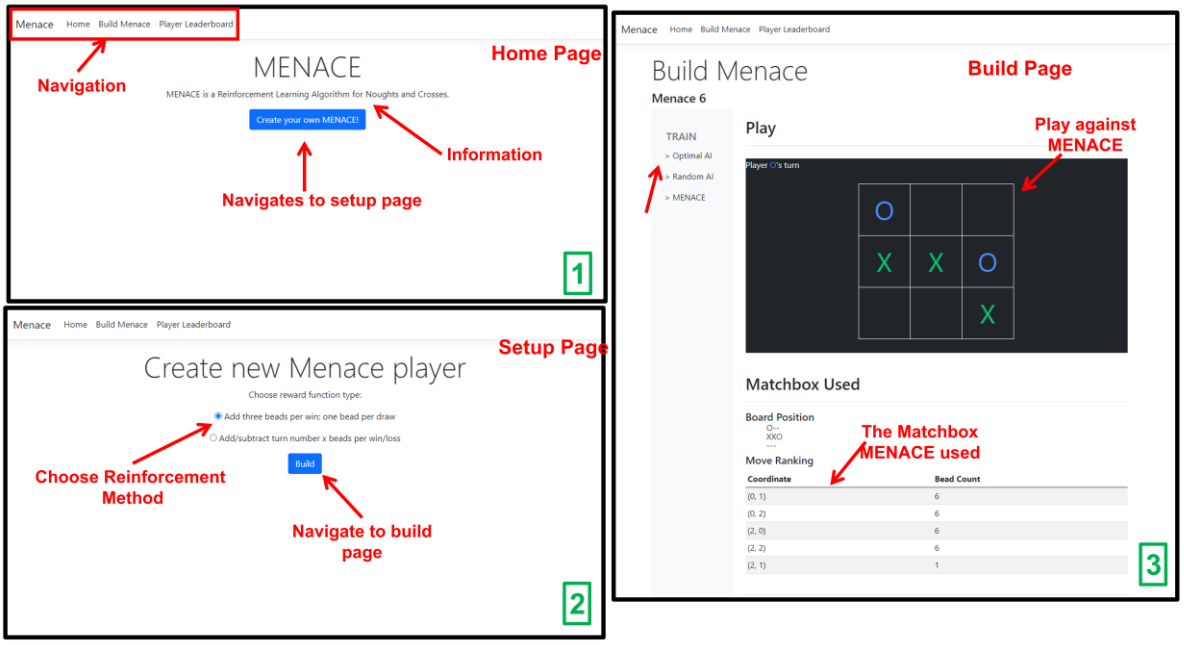


Figure 4.15 Home page and Build Menace pages

The Player Leader board page (shown in figure 4.16) displays all **`PlayerMenace`**s currently in the database with how many games they've played, and how many of those were won, drawn, or lost. There is also the option to play any of the **`PlayerMenace`**s listed as well as view the details.

The Player details page displays **`PlayerMenace`** properties such as name, games, wins etc. There are also tables displaying the **`PlayerMenace`**'s training history as well as its **`Matchbox`**es. Furthermore, there is the option to train a **`PlayerMenace`** from this screen with a user specified number of iterations. The results of the training can immediately be viewed.

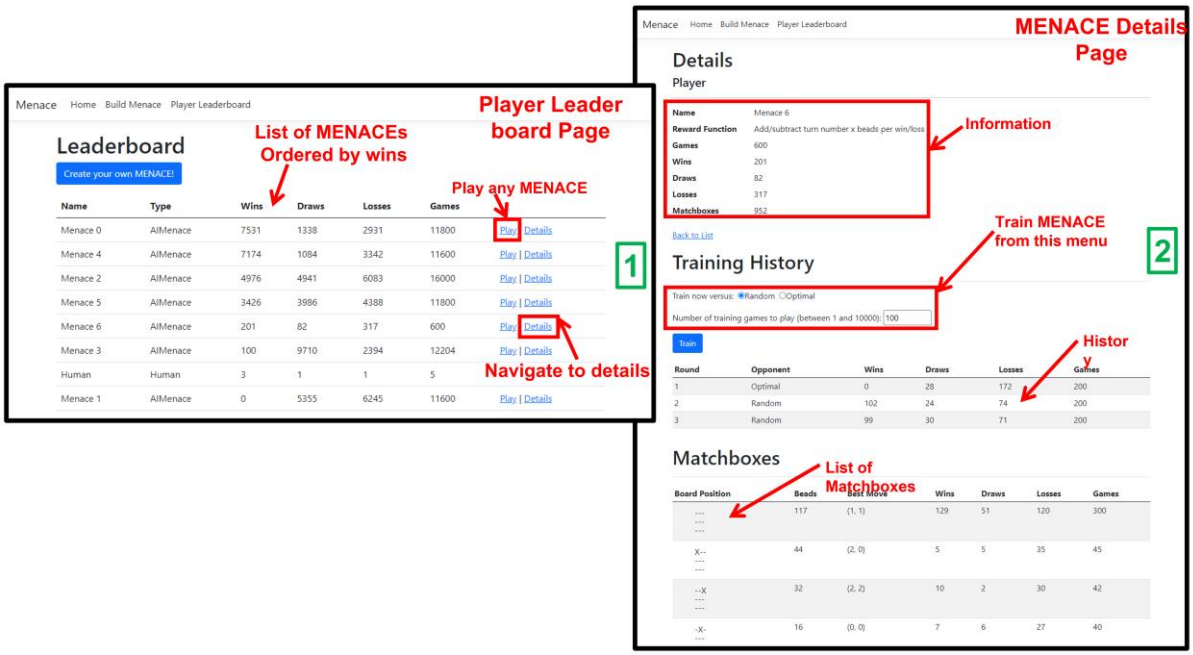


Figure 4.16 Leader board and Details pages

4.5.2 Technologies

The aspiration of this project is to build MENACE in an explainable and presentable way. The technologies used for this task were chosen with this in mind.

4.5.2.1 ASP.NET MVC Web Application Framework

ASP.NET MVC is a lightweight framework for building web applications. This is ideal for a small application such as playing and experimenting with a relatively basic Noughts and Crosses AI (Microsoft, 2022).

ASP.NET MVC uses the Model View Controller design pattern. This pattern is typically used for user interfaces. It splits an application into three parts: model, view, and controller (Microsoft, 2022). The model implements the logic for the data which is typically stored in an SQL server. The model performs CRUD operations such as retrieving information from the database, changing it in some way and updating the database accordingly (Microsoft, 2022). For MENACE, its matchboxes will be stored in a database which will be retrieved by the model, used for playing a game, changed at the end of a game, and put back in the database.

The view is the display the user sees. For an application to be interactive, the view needs to represent the model for the user and the user needs to be able to interact with the model from the view (Microsoft, 2022). For MENACE, this will allow users to play Noughts and Crosses against it as well as see the Matchboxes and Beads. This interaction between the view and the model is handled by the controller.

The controller receives requests from the view and interacts with the model to determine a response (Microsoft, 2022). For example, in MENACE, a user will be able to play Noughts and Crosses. When a user clicks a tile, the information will be sent back to the controller which will load an instance of MENACE from the model, work out the move MENACE plays in response and update the view accordingly.

4.5.2.2 Microsoft SQL Server

Microsoft SQL Server is Microsoft's relational database management system.

4.5.2.3 Entity Framework

Entity Framework is a code first Object Relational Mapper (ORM). ORMs are designed for mapping C# object instances to SQL tables (Microsoft, 2022). Code first means that the class library of MENACE can be designed and implemented before much thought is given to how it will work in a database which introduces constraints such as primary and foreign keys (Tutorials Point, 2022). Entity Framework is an ASP.NET tool which makes it straight forward to store C# objects such as MENACE, Matchboxes, Beads etc. in a database.

4.5.2.4 ASP.NET Razor

Razor is a syntax which enables the embedding of C# code into web pages (W3 Schools, 2022). Practically this means C# content can be inserted into HTML files. When the page is loaded in the browser, the content of the C# is converted into html. This makes coding views and controllers easy because content from the controller can be passed to the view which will handle how the content is presented.

4.5.2.5 HTML, CSS, JavaScript

The view is written using HTML, CSS, and JavaScript. The HTML handles the layout of web pages while CSS handles the styling and appearance. JavaScript makes the web pages interactive (Cox, 2022) and can also be used to submit forms to the controller. For example, JavaScript handles the interactive Noughts and Crosses applet.

4.5.3 System Architecture

Figure 4.17 below depicts an overview of the system architecture with all the technologies mentioned from section 4.3.2.

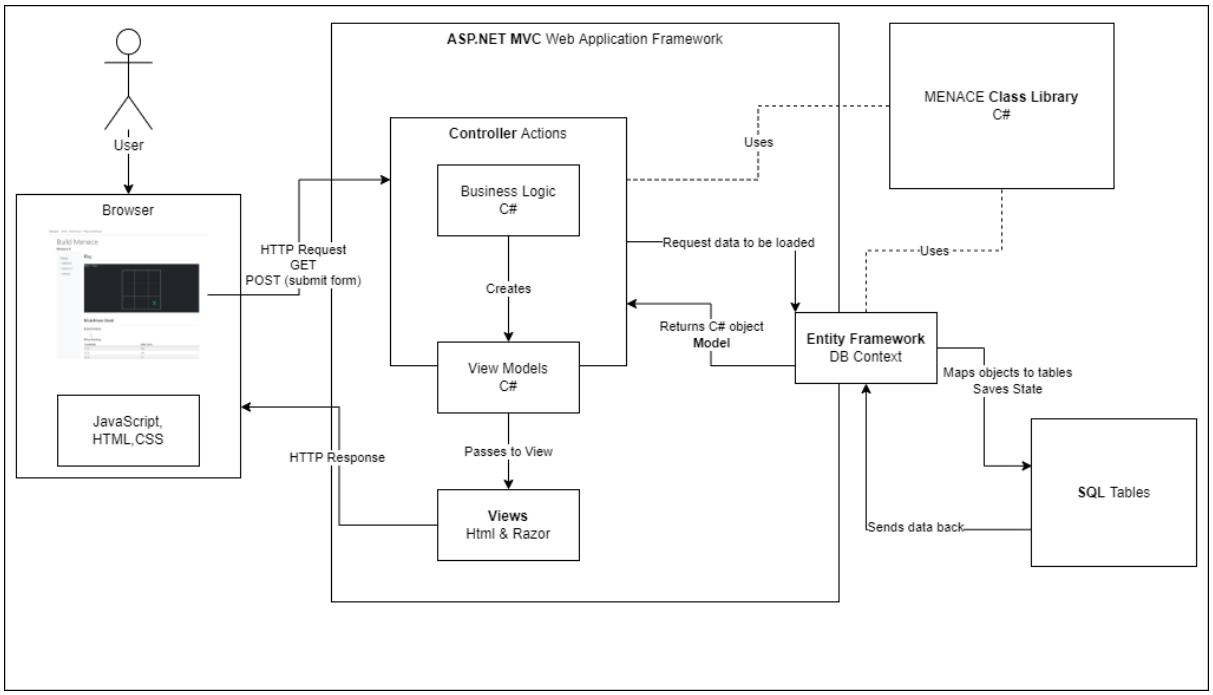


Figure 4.17 System Architecture

When the user interacts with the view from the browser, HTTP requests are made. This takes the information provided by an HTML form, submitted by an event listener in a JavaScript script, and sends it to the controller. The controller contains many C# methods which deal with requests from the view. The ASP.NET MVC framework mostly takes care of routing requests to specific controller

methods, provided the intended method is specified in an **asp-action** tag from the HTML and naming conventions are adhered to.

An example of this process is when a user is playing Noughts and Crosses against a **PlayerMenace**. When a user clicks a Noughts and Crosses tile on the web page, a **POST** request is made to the controller, carrying the coordinates the user played as well as other important game information. The controller will contain a C# method which handles this request. This method uses an Entity Framework **DbContext** instance to fetch the C# objects it needs from the database. Following our example, this would be the **PlayerMenace** with all its **Matchboxe**s and **Bead**s. From here, the controller method uses the game information provided by the view to retrieve the turn **PlayerMenace** plays in response. At this stage a **GameHistory** is updated and sent back to the database through the **DbContext** instance. The controller then returns a view and passes it a view model which is the information that needs to be sent back to the view bundled into one C# object.

Views are Razor files. These are HTML files with C# embedded in them. This lets the controller pass C# objects to the view which the view formats nicely and then sends back to the browser for the user to see.

4.6 Web Application Implementation

4.6.1 MVC Entry Point

Figure 4.18 shows the general file structure of an ASP.NET MVC project. When an MVC project is setup for the first time, this structure is mostly automatically generated, including the application entry point **Program.cs**.

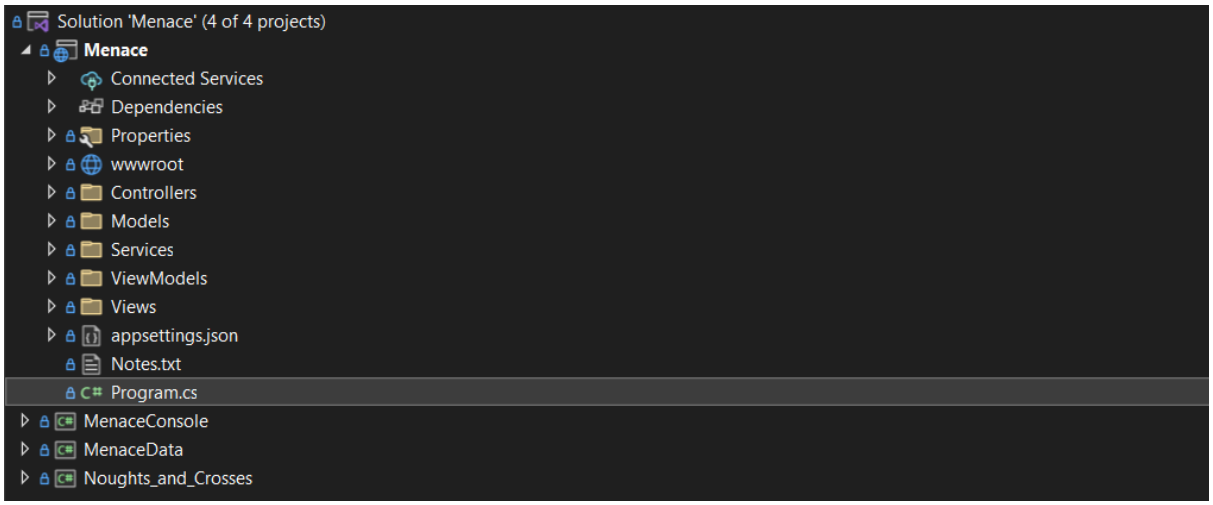


Figure 4.18 File structure of Menace Web App

Program.cs is mostly implemented when a new MVC project is started. Most lines of code here were not messed with nor necessary to understand because MVC takes care of it. However, it was important to specify a connection string for connecting to the correct database which can be seen highlighted on line 13 in figure 4.19.

```
using MenaceData;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;


var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddMvc();
builder.Services.AddDbContext<MenaceContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("MenaceConnectionSql"));
});

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

```

Figure 4.19 Menace Web App entry point

4.6.2 Configuring Entity Framework

To use Entity Framework from code in a controller or elsewhere, a **`DbContext`** object is used. To make Entity Framework work with a custom class library you can create a class which inherits from **`DbContext`** that specifies how to map your class library objects to a database. In this project this class is called **`MenaceContext`**.

`MenaceContext` consists of **`DbSet`**s of the MENACE class library objects. This tells Entity Framework which objects need to be tracked and persisted in the database. Entity Framework uses this to build a database when the first data migration is made. Every object that Entity Framework was told to persist is turned into a SQL table. Objects which have other objects as properties is represented in SQL as relationships. For example, all **`Matchbox`**es have a **`BoardPosition`**. **`Matchbox`**es from different **`AIMenace`**s may refer to the same **`BoardPosition`**. Therefore, many **`Matchbox`**es can refer to one **`BoardPosition`**. In the database this is represented as a one-to-many relationship between the **`BoardPosition`** and **`Matchbox`** tables where **`Matchbox`** entities have a foreign key containing a **`BoardPositionId`**.

```
namespace MenaceData
{
    20 references
    public partial class MenaceContext : DbContext
    {
        8 references
        public DbSet<BoardPosition> BoardPosition { get; set; }

        3 references
        public DbSet<Bead> Bead { get; set; }

        15 references
        public DbSet<Matchbox> Matchbox { get; set; }

        2 references
        public DbSet<AIMenace> AIMenace { get; set; }
```

Figure 4.20 Setting up an Entity Framework **`DbContext`** *class: adding* **`DbSets`**

In addition to **DbSet**s, **MenaceContext** lets you customise how the database tables and relations are setup by overriding the **OnModelCreating()** method. This includes specifying what entities should do with their foreign keys when they are deleted. By default, the delete cascades to the entities the foreign keys point to. This was causing major issues because for some entities this behaviour was incorrect. For example, when a **Matchbox** is deleted, the deletion should not cascade to a **BoardPosition** because many other entities still rely on it such as **GameHistory**. **OnModelCreating()** also allows you to seed the database with objects. For this project the empty **BoardPosition** and a **PlayerHuman** (used for all humans interacting with the website) is seeded.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // BoardPosition
    modelBuilder.Entity<BoardPosition>()
        .HasKey(b => b.BoardPositionId);

    modelBuilder.Entity<BoardPosition>()
        .HasData(new BoardPosition { });

    // Game
    modelBuilder.Entity<Game>()
        .HasOne(g => g.P1)
        .WithMany()
        .OnDelete(DeleteBehavior.NoAction);

    modelBuilder.Entity<Game>()
        .HasOne(g => g.P2)
        .WithMany()
        .OnDelete(DeleteBehavior.NoAction);

    modelBuilder.Entity<Game>()
        .HasOne(g => g.CurrentBoard)
        .WithMany()
        .OnDelete(DeleteBehavior.NoAction);
```

Figure 4.21 Setting up an Entity Framework **DbContext** *class: Overriding* **OnModelCreating()**

Once **MenaceContext** was configured, a data migration was made. This can be done from the NuGet package manager console using the commands: **add-migration vN** and **update-database**. This automatically generates a file which specifies how to create the SQL tables for the database based on **MenaceContext**.

```
migrationBuilder.CreateTable(
    name: "Matchbox",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        BoardPositionId = table.Column<string>(type: "nvarchar(450)", nullable: false),
        Wins = table.Column<int>(type: "int", nullable: false),
        Draws = table.Column<int>(type: "int", nullable: false),
        Losses = table.Column<int>(type: "int", nullable: false),
        AIMenaceId = table.Column<Guid>(type: "uniqueidentifier", nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Matchbox", x => x.Id);
        table.ForeignKey(
            name: "FK_Matchbox_AIMenace_AIMenaceId",
            column: x => x.AIMenaceId,
            principalTable: "AIMenace",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_Matchbox_BoardPosition_BoardPositionId",
            column: x => x.BoardPositionId,
            principalTable: "BoardPosition",
            principalColumn: "BoardPositionId");
    });
```

Figure 4.22 Data Migration

4.6.3 Adapting the MENACE Class Library

Despite Entity Framework being a code first ORM, it was still necessary to make changes to the MENACE class library so that it was more compatible with the website. For example, making **`PlayerOptimal`** and **`PlayerRandom`** static classes so they would not need to be persisted in the database.

The most noteworthy change was that board positions in the frontend are represented as strings as this is easier to work with for building interactive views. Since the old representation was referenced all over the place in the class library and time was short it was decided to make a method in **`BoardPosition`** for converting both representations. In the future, **`BoardPosition`** will use one representation to avoid confusion.

```
50 references
public string BoardPositionId
{
    get
    {
        var value = "";
        for (int i = 0; i < Coords.GetLength(0); i++)
        {
            for (int j = 0; j < Coords.GetLength(1); j++)
            {
                value += GameSymbol.MapIntToSymbol(Coords[i, j]);
            }
        }
        return value;
    }
    set
    {
        for (int i = 0; i < value.Length; i++)
        {
            Coords[i / 3, i % 3] = GameSymbol.MapSymbolToInt(value[i]);
        }
    }
}
```

Figure 4.23 Converting **`BoardPosition`** *coordinates into strings and vice versa*

4.6.4 Database Design

Spending too much time designing the database was avoided because this is something Entity Framework should mostly be able to handle. However, there are some noteworthy decisions that had to be made for Entity Framework to work properly. Firstly, to make it easy to compare **`BoardPosition`**s in the database, the **`BoardPosition`** primary key **`BoardPositionId`** was made to be the actual board position the **`BoardPosition`** object represents.

There were also several specifications made in the **`MenaceContext`** **`OnModelCreating()`** method. These were as follows:

1. When a **`Game`** entity is deleted, it does not delete the **`Player`** and **`BoardPosition`** entities which its **`P1`**, **`P2`** and **`CurrentBoard`** foreign keys point to.
2. When a **`GameHistory`** entity is deleted, it does not delete the **`Player`** entities which its **`P1`**, **`P2`**, and **`Winner`** foreign keys point to.
3. When a **`Matchbox`** entity is deleted, it does not delete the **`BoardPosition`** entity which its **`BoardPosition`** foreign key points to.
4. When a Turn entity is deleted, it does not delete the **`BoardPosition`** and **`Player`** entities which its **`Before`**, **`After`**, and **`TurnPlayer`** foreign keys point to.
5. Seeding the database with an entity for the empty **`BoardPosition`**.
6. Seeding the database with an entity for the default **`PlayerHumanOnWeb`**.

Figure 4.24 Entity Relationship Diagram for MENACE Web Application

4.6.5 Build Menace

With ASP.NET MVC, and Entity Framework and the Database setup it was possible to move on to creating views and controllers. The main idea of the website was to allow users to play a game of Noughts and Crosses against their own instance of MENACE. This was realized by creating the Build Menace webpage which has the added features of buttons allowing MENACE to train against an **`AIOptimal`** or **`AIRandom`** for a number of iterations as well as displaying the **`Matchbox`** MENACE used to play its turn with the **`Bead`**s and **`Bead`** counts. This turned out to be the most challenging aspect of implementing the web application.

As a starting point, MVC has a feature which automatically generates views and a controller for any entity in your model. These views and controller methods include templates allowing users to create, delete, edit, and view the details of an entity in the model from the view. This auto-generation was applied to the **`Players`** entity so that the templates could be referred to and copied when building the Build Menace views and controller which incorporates aspects from all of these features. Figure 4.25 shows a modified example of one of these templates, though it has not changed much from the original template.

```
@using Menace.ViewModels
@using Noughts_and_Crosses

@model IEnumerable<Noughts_and_Crosses.Player>

@{
    ViewData["Title"] = "Players";
}

<h1>Leaderboard</h1>

<p>
    <a class="btn btn-primary" asp-area="" asp-controller="BuildMenace" asp-action="BuildSetup">Create your own MENACE!</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                Type
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Wins)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Draws)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Losses)
            </th>
            <th>
                Games
            </th>
            <th></th>
        </tr>
    </thead>
```

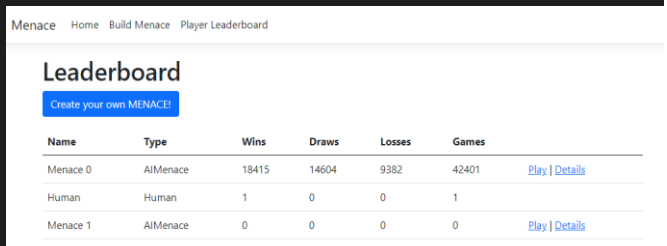


Figure 4.25 Player index view with how it appears in the browser

4.6.5.1 Implementing the View

Views are **`cshtml`** files which use the razor syntax. Firstly, the layout of the webpage was built using HTML and CSS. It was stitched together by copying bits from a Bootstrap template online (Bootstrap, 2022) as well as bits from the auto-generated **`Player`** views and controller. In addition to this, a Noughts and Crosses applet inspired by an online tutorial (JavaScript Academy, 2022) was added in the centre of the screen. The view is depicted in figure 4.26.

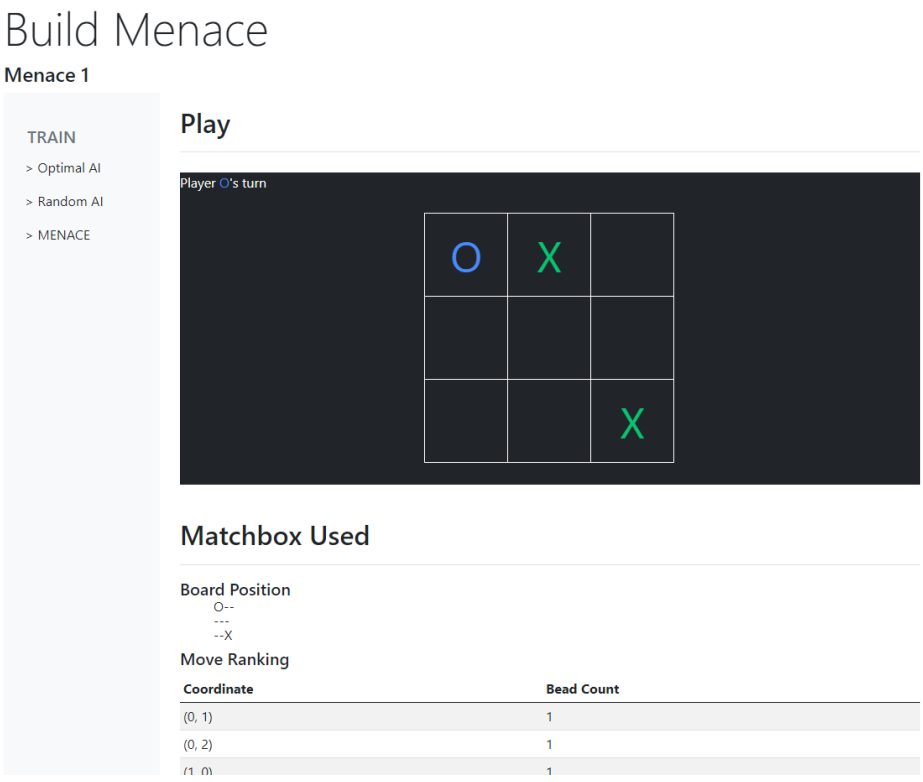


Figure 4.26 Build Menace View

4.6.5.2 Implementing the Noughts and Crosses Applet

To make the view interactive it uses a script called **NandCApplet.js**. When a tile in the Noughts and crosses board is clicked, it triggers an event listener in the JavaScript (as seen in figure 4.27). The JavaScript then updates its own representation of the board as well as submits the HTML form containing information about the game state (as seen in figure 4.28). MVC routes this to the **Build(GameState)** method with the attribute labelled **[HttpPost]** from the **BuildMenaceController** class (as seen in figure 4.29).

```
if (isGameActive) {
    tiles.forEach((tile, index) => {
        tile.addEventListener('click', () => userAction(tile, index));
    });
}
const userAction = (tile, index) => {
    if (isValidAction(tile) && isGameActive) {
        tile.innerText = currentPlayer;
        tile.classList.add(`player${currentPlayer}`);
        updateBoard(index);
        handleResultValidation();
        changePlayer();

        saveState();
        submitForm.submit();
    }
    else {
        alert('invalid');
    }
}
```

Figure 4.27 ***NandCApplet.js****: JavaScript tile click event listener.*

```
{
    <form asp-action="Build" id="submitForm">
        <input type="hidden" asp-for="BoardBeforeInput" id="hiddenBoardBeforeInput" />
        <input type="hidden" asp-for="BoardAfterInput" id="hiddenBoardAfterInput" />
        <input type="hidden" asp-for="CurrentPlayerSymbol" id="hiddenCurrentPlayerSymbol" />
        <input type="hidden" asp-for="GameHistoryId" id="hiddenGameHistoryId" />
        <input type="hidden" asp-for="IsGameActive" id="hiddenIsGameActive" />
        <input type="hidden" asp-for="GameType" id="hiddenGameType" />
    </form>
}
```

Figure 4.28 Build view: HTML form

```
[HttpPost]
[ValidateAntiForgeryToken]
3 references
public IActionResult Build([Bind("BoardBeforeInput, BoardAfterInput, CurrentPlayerSymbol, GameHistoryId, GameType")] GamePlayState gameState)
{
    if (ModelState.IsValid)
    {
```

Figure 4.29 **BuildMenaceController***:* **[HttpPost] Build** *method*

4.6.5.3 Implementing the Controller

It is important to mention that the **BuildMenaceController** class has a **MenaceContext** object as a property. This is used by all methods in the **BuildMenaceController**.

The **[HttpPost] BuildMenaceController.Build()** takes a **GamePlayState** as parameter. **GamePlayState** is a view model which is just a C# class containing all the information the **Build** method needs. It includes the board before and after a user input, the current player's symbol, the ID of the **GameHistory**, whether the game is active, the type of game (i.e. is MENACE player 1 or 2), the ID of player 1 and player 2 and more.

The controller needs to handle interactions between the view and the model. This means fetching entities from the database and instantiating them as C# objects so some logic can be applied to them. Using the **GameHistoryID** from the **GamePlayState** passed to it from the view, the Build method loads the full game history from the database. It then instantiates the **humanPlayer** and **aiPlayer** from the database as C# objects. By default, Entity Framework does this with lazy loading. This means it does not load all of an entities' properties unless specifically asked for e.g. MENACE's **Matchbox**es and **Bead**s. This is to improve performance by minimising interaction with the database which is usually slow. However, for the purpose of the **Build** method this is problematic because all of MENACE's properties need to be loaded into the **aiPlayer** object so it can play its turn. This is done with a static **PlayerFactory** class which explicitly includes all the **PlayerMenace** properties needed upon loading (seen in figure 4.30). Note the use of the **MenaceContext** class.

```
9 references
public static Player GetPlayer(MenaceContext context, Guid playerId, PlayerType playerType)
{
    switch (playerType)
    {
        case PlayerType.Human:
            return context.Player.Where(p => p.Id == playerId).Single();
        case PlayerType.AIOptimal:
            throw new NotImplementedException();
        case PlayerType.AIRandom:
            throw new NotImplementedException();
        case PlayerType.AIMenace:
            var menace = context.PlayerMenace
                .Where(p => p.Id == playerId)
                .Include("MenaceEngine.Matchboxes")
                .Include("MenaceEngine.Matchboxes.Beads")
                .Include("MenaceEngine.Matchboxes.BoardPosition")
                .Include("TrainingHistory.Rounds")
                .Single();
            return menace;
        default:
            throw new Exception($"Unexpected player type {playerType}");
    }
```

Figure 4.30 **PlayerFactory.GetPlayer()**

With this loaded, the Build method works out the turn the **humanPlayer** just played and records it in the history. If the **humanPlayer** played the winning turn a method is called to handle the end of the game (discussed later). Otherwise, the **aiPlayer** or MENACE can play its turn. If **aiPlayer** does not play the winning turn, then the database is updated with the new turns played and a new **GamePlayState** is created with the updated game state. This is then passed to the **Build.cshtml** view which has embedded C# telling the view and JavaScript how to render the new game state.

```
// Load game history with players
var game = _context.GameHistory.Where(g => g.Id == gameState.GameHistoryId)
    .Include(g => g.P1)
    .Include(g => g.P2)
    .Include(g => g.Turns)
    .Single();

var humanPlayer = game.P1 is PlayerMenace ? game.P2 : game.P1;

var nonHumanPlayer = game.P1 is PlayerMenace ? game.P1 : game.P2;

var aiPlayer = PlayerFactory.GetPlayer(_context, nonHumanPlayer.Id, PlayerType.AIMenace) as PlayerMenace;

var humanSymbol = humanPlayer == game.P1 ? "X" : "O";

var aiSymbol = nonHumanPlayer == game.P1 ? "X" : "O";

// Human player's turn
var humanTurn = RecordHumanTurn(game, humanPlayer, boardBeforeInput, boardAfterInput);

// Check win
if (humanTurn.After.IsGameOver)
{
    return HandleEndOfGame(game, humanTurn, humanSymbol, aiPlayer, gameState.GameType);
}

// AI player's turn
var aiTurn = PlayMenaceTurn(game, aiPlayer, boardAfterInput, aiSymbol, boardAfterInput.TurnNumber);

// Check win
if (aiTurn.After.IsGameOver)
{
    return HandleEndOfGame(game, aiTurn, aiSymbol, aiPlayer, gameState.GameType);
}
_context.SaveChanges();

// Reload UI with new game state
ModelState.Clear();

var matchbox = aiPlayer.MenaceEngine.MatchboxByBoardPos(aiTurn.Before);

var newGameState = new GamePlayState
{
    BoardBeforeInput = GamePlayState.WrapBoard(aiTurn.After.BoardPositionId),
    GameHistoryId = gameState.GameHistoryId,
    IsGameActive = true,
    CurrentPlayerSymbol = humanSymbol,
    GameType = gameState.GameType,
    Player1Id = humanPlayer.Id,
    Player2Id = aiPlayer.Id,
    Beads = matchbox?.Beads,
    Matchbox = matchbox,
    MenaceName = aiPlayer.Name
};
return View(newGameState);
```

Figure 4.31 **`BuildMenaceController.PlayMenaceTurn()`**

This part of the build webpage was particularly challenging to implement. In a lot of cases Entity Framework needed to be explicitly told to do tasks which it was thought Entity Framework could handle. For example, if an **`AIMenace`** creates a new **`Matchbox`**, Entity Framework needs to be told to add it to the database. Furthermore, if the new Matchbox contained a **`BoardPosition`** which already existed in the database, Entity Framework had to be told not to make the **`Matchbox`** point to the existing **`BoardPosition`** entity in the database instead of creating a new one (see figure 4.32). Examples like this made working with Entity Framework tedious because it was not clear what Entity Framework could or couldn't do.

```
2 references
private Turn PlayMenaceTurn(GameHistory game, PlayerMenace aiPlayer, BoardPosition initialBoardPosition, string playerSymbol, int turnNumber)
{
    initialBoardPosition = _context.BoardPosition.GetOrAddIfNotExists(initialBoardPosition, b => b.BoardPositionId == initialBoardPosition.BoardPositionId);

    var aiTurn = aiPlayer.PlayTurn(initialBoardPosition, GameSymbol.MapSymbolToInt(playerSymbol), turnNumber);

    aiTurn.After = _context.BoardPosition.GetOrAddIfNotExists(aiTurn.After, b => b.BoardPositionId == aiTurn.After.BoardPositionId);

    _context.Turn.Add(aiTurn);

    game.AddMove(aiTurn);

    var matchbox = aiPlayer.MenaceEngine.Matchboxes.Single(m => m.BoardPosition.BoardPositionId == initialBoardPosition.BoardPositionId);

    if (_context.Matchbox.AddIfNotExists(matchbox, m => m.Id == matchbox.Id))
    {
        _context.Entry(matchbox).Reference(m => m.BoardPosition).IsModified = false;

        foreach (var bead in matchbox.Beads)
        {
            _context.Bead.Add(bead);
        }
    }

    return aiTurn;
}
```

Figure 4.32 **`BuildMenaceController.PlayMenaceTurn()`**

4.6.5.5 Handling the End of a Game

If the game has ended either by a player winning or the board being full, the **`HandEndOfGame`** method is called. This adds the appropriate win, loss, or draw to the players and reinforces the MENACE player. The database is then updated, and the view is returned with the final **`GamePlayState`** which importantly marks the game as being inactive. This is then used by the view to show the end of the game as well as showing a button which lets the user play again.

```
2 references
private IActionResult HandleEndOfGame(GameHistory game, Turn lastTurn, string currentPlayerSymbol, PlayerMenace aiPlayer, GameType gameType)
{
    // Record final state
    if (lastTurn.After.IsWinningPosition)
    {
        game.Winner.Wins++;
        if (game.P1 == game.Winner)
        {
            game.P2.Losses++;
        }
        else
        {
            game.P1.Losses++;
        }
        // Reinforce aiPlayer
        if (aiPlayer != null)
        {
            ReinforcementIncremental.Reinforce(game, aiPlayer);
        }
    }
    else if (lastTurn.After.IsBoardFull)
    {
        game.P1.Draws++;
        game.P2.Draws++;
        // Reinforce aiPlayer
        if (aiPlayer != null)
        {
            ReinforcementIncremental.Reinforce(game, aiPlayer);
        }
    }
    _context.SaveChanges();
    // Display end of game UI
    ModelState.Clear();
    var finalState = new GamePlayState
    {
        BoardBeforeInput = GamePlayState.WrapBoard(lastTurn.After.BoardPositionId),
        CurrentPlayerSymbol = currentPlayerSymbol,
        IsGameActive = false,
        GameType = gameType == GameType.MenaceP1 ? GameType.MenaceP2 : GameType.MenaceP1,
        Player1Id = game.P2.Id,
        Player2Id = game.P1.Id,
        Beads = aiPlayer.MenaceEngine.MatchboxByBoardPos(lastTurn.Before)?.Beads,
        Matchbox = aiPlayer.MenaceEngine.MatchboxByBoardPos(lastTurn.Before),
        MenaceName = aiPlayer.Name
    };
    return View(finalState);
}
```

Figure 4.33 **`BuildMenaceController.HandleEndOfGame()`**

4.6.5.6 Displaying the Matchbox Used

The **`GamePlayState`** view model includes a Matchbox property which is used by the view to make a table representing the matchbox used. The table is ordered by the **`Bead.Count`** property and also displays the coordinate each bead represents.

Matchbox Used

Board Position

-X-

Move Ranking

Coordinate	Bead Count
(2, 0)	27
(2, 2)	21
(0, 0)	19
(2, 1)	19
(0, 2)	7
(0, 1)	5
(1, 0)	1
(1, 2)	1

Figure 4.34 Matchbox Display

4.6.5.7 Training Buttons

The view includes buttons allowing users to train their MENACE instance for 1000 iterations against an **`AIRandomMove`**, **`AIOptimal`** or **`AIMenace`**. For this to work, the class library had to be adapted so that **`PlayerRandom`** and **`PlayerOptimal`** as well as **`AIRandomMove`** and **`AIOptimal`** were static classes. This was possible because these classes never change their internal state. It was necessary so that they did not need to be persisted in the database and using Entity Framework could be avoided.

Since these methods were static, implementing the controller methods for these buttons meant loading the **`PlayerMenace`** from the database using the **`PlayerId`** passed by the view, creating a game object, and training the **`PlayerMenace`** against a static **`PlayerOptimal`** for 1000 iterations. The **`MenaceContext`** object was then used to update the database with any new **`Matchbox`**es and **`Bead`**s MENACE has.

```
public IActionResult TrainOptimal(GameCreate createGameInput)
{
    // Setup players
    PlayerMenace playerMenace;
    PlayerOptimal playerOptimal;
    if (createGameInput.Type == GameType.MenaceP1)
    {
        playerMenace = PlayerFactory.GetPlayer(_context, createGameInput.Player1Id, PlayerType.AIMenace) as PlayerMenace;
        playerOptimal = new PlayerOptimal("Optimal Trainer");
    }
    else if (createGameInput.Type == GameType.MenaceP2)
    {
        playerOptimal = new PlayerOptimal("Optimal Trainer");
        playerMenace = PlayerFactory.GetPlayer(_context, createGameInput.Player2Id, PlayerType.AIMenace) as PlayerMenace;
    }
    else { throw new Exception("Invalid input when choosing if Menace is P1 or P2"); }


    // train Menace
    for (int i=0; i < 1000; i++)
    {
        var game1 = new Game(playerMenace, playerOptimal);
        var game2 = new Game(playerOptimal, playerMenace);

        game1.Train();
        game2.Train();
    }

    // Add new matchboxes and beads.
    foreach (Matchbox matchbox in playerMenace.MenaceEngine.Matchboxes)
    {
        matchbox.BoardPosition = _context.BoardPosition.GetOrAddIfNotExists(matchbox.BoardPosition, b => b.BoardPositionId == matchbox.BoardPosition.BoardPositionId);

        if (_context.Matchbox.AddIfNotExists(matchbox, m => m.Id == matchbox.Id))
        {
            _context.Entry(matchbox).Reference(m => m.BoardPosition).IsModified = false;

            foreach (var bead in matchbox.Beads)
            {
                _context.Bead.Add(bead);
            }
        }
    }

    _context.SaveChanges();

    // Display end of game UI
    ModelState.Clear();

    return RedirectToAction(nameof(Build), createGameInput);
}
```

Figure 4.35 **`BuildMenaceController.TrainOptimal()`**

4.6.5.8 How the Build Webpage is Initialised

When a user navigates to build webpage, a **`HTTP GET`** request is routed to the **`BuildMenaceController Build(GameCreate)`** method. This method takes a **`GameCreate`** object which is the view model containing information needed to create a new game. This includes an ID for a **`PlayerMenace`**, an ID for a **`PlayerHumanOnWeb`**, the **`GameType`** which indicates whether **`PlayerMenace`** is player 1 or 2, and the **`RewardFunctionType`** which indicates which Reinforcement type **`PlayerMenace`** is using. With this information, the appropriate players are loaded from the database or created and added to the database. If MENACE is player 1, it needs to play it's opening move. A new **`GameHistory`** is initialised, and MENACE's turn is added. A **`GamePlayState`** is then created, and the view is redirected to the **`[HttpPost] Build`** method which takes a **`GamePlayState`** object. This method handles the rest of the game.

```
94          [HttpGet]
            3 references
95          public IActionResult Build(GameCreate createGameInput)
96          {
97              // Set-up game
98              Player player1;
99              Player player2;
100
101             if (createGameInput.GameType == GameType.MenaceP1)
102             {
103                 player1 = PlayerFactory.GetPlayer(_context, createGameInput.Player1Id, PlayerType.AIMenace);
104                 player2 = PlayerFactory.GetPlayer(_context, createGameInput.Player2Id, PlayerType.Human);
105
106             }
107             else if (createGameInput.GameType == GameType.MenaceP2)
108             {
109                 player1 = PlayerFactory.GetPlayer(_context, createGameInput.Player1Id, PlayerType.Human);
110                 player2 = PlayerFactory.GetPlayer(_context, createGameInput.Player2Id, PlayerType.AIMenace);
111             }
112             else { throw new Exception("Invalid input when choosing if Menace is P1 or P2"); }
113
114             var aiPlayer = player1 as PlayerMenace ?? player2 as PlayerMenace;
115
116             var newGame = new GameHistory(player1, player2);
117
118             _context.Add(newGame);
119
120             var boardPosition = new BoardPosition();
121
122             var playerSymbol = "X";
123
124             if (player1 is PlayerMenace)
125             {
126                 PlayMenaceTurn(newGame, player1 as PlayerMenace, boardPosition, playerSymbol, 1);
127
128                 boardPosition = newGame.Turns.Last().After;
129
130                 playerSymbol = "O";
131             }
132
133             _context.SaveChanges();
134
135             // Set-up UI
136             ModelState.Clear();
137
138             var gameState = new GamePlayState
139             {
140                 BoardBeforeInput = GamePlayState.WrapBoard(boardPosition.BoardPositionId),
141                 GameHistoryId = newGame.Id,
                    IsGameActive = true
```

Figure 4.36 **`[HttpGet]` `Build`** *method*

5 Evaluation

This chapter aims to evaluate the performance of MENACE as well as how effective it is for demonstrating Machine Learning. This comes in two parts. Section 5.1 and 5.2 evaluates MENACE's performance whereas section 5.3 evaluates the success of the website.

5.1 Model Evaluation

This section discusses the methods for evaluating MENACE, the experiments that were conducted and an analysis of the results.

5.1.1 Approach

There are two main ways to evaluate a Reinforcement Learning algorithm such as MENACE. The first is to consider the rewards an agent received while learning (David L. Pool, 2017). This was the approach Michie took for MENACE's maiden tournament against a human as he plotted the cumulative reward MENACE received over each game (Michie, 1961) (see figure 3.4). This approach is mostly used when an agent needs to learn while it is being deployed or if it is unlikely there will ever be a point at which the agent should stop exploring (David L. Pool, 2017). This evaluation method reveals how fast or effective the agent is at learning which may be the primary interest. This approach is ideal because it is intended that users will be involved in the training process so they can see MENACE improve as they play more games with it.

The second way to evaluate a Reinforcement Learning algorithm is to consider how good the policy is that it found (David L. Pool, 2017). Given that Noughts and Crosses has a well-defined optimal policy (see section 3.4) it is possible to value the policy MENACE learns by how much it deviates from the optimal policy. This approach is typically used when the agent has time to learn before being deployed and the primary interest is how effective the agent is in a real-world scenario (David L. Pool, 2017).

Evaluating MENACE by considering its rewards over the games it plays was the main evaluation method used for this project. This is because this approach is most effective for demonstrating how machines can improve their performance over time through learning. However, measuring the policy MENACE uses compared with the optimal policy is also of great interest for this project. A good metric which would combine both approaches would be to measure the difference between MENACE's policy and the optimal policy as it plays Noughts and Crosses games. This would be something for future work.

5.1.2 Experiments

There are several components of MENACE to experiment with. The first is the training method. MENACE was trained on the following AI types and the results are analysed in section 5.2:

1. **`PlayerRandom`**
2. **`PlayerOptimal`**
3. Alternating between **`PlayerRandom`** and **`PlayerOptimal`**

It is intended that the website will facilitate the training of a **`PlayerMenace`** against another instance of **`PlayerMenace`** in the future. This was something Michie alluded to in his paper as the best training method (Michie, 1961), so it is of high interest for this project. However, implementing this on the website is considerably more challenging than **`PlayerRandom`** and **`PlayerOptimal`** because **`PlayerMenace`** is not a static class and needs to persist its state in the database. This introduces

complications which, with the interest of finishing a deliverable product on time, was side lined as non-essential. However, it is possible to play against a **`PlayerMenace`** trained this way in from a console. The observations from this are briefly discussed in section 4.4.10.

The second component of MENACE to experiment with is the reinforcement method. Michie experimented with various reinforcement functions. The functions implemented so far in this project are the following:

1. Rewarding three beads for a win, one for a draw and none for a loss. This is based on Michie's first experiments (Michie, 1961). In the code this is denoted as: **`ReinforceThreePerWinOnePerDraw`**. For the rest of this document, it will be abbreviated to ***`ThreeBead`***.
2. Rewarding or confiscating the number of beads equal to the turn number in which a matchbox was used for a win or loss. This is based on the Michie's suggestions for experimenting with the reinforcement function (Michie, 1961). In the code this is denoted as: **`ReinforceWinPlusTurnLossMinusTurn`**. For the rest of this document, it will be abbreviated to ***`TurnNumber`***.

The experiments were conducted by playing MENACE against the various AI types in steps of 200 iterations and measuring the ratios of wins and draws to losses. Graphs were then plotted to show MENACE's progress. Each reinforcement method was trained on the AI types mentioned above.

It is important to note that the following training programs occur over thousands of training iterations. This may seem strange when compared with Michie's MENACE which learned over hundreds of iterations. The reason for this is that Michie's MENACE was programmed to recognise equivalent board positions (Michie, 1961). An example of this is depicted in figure 5.1 where four seemingly distinct positions are actually variants of the same position. This is something this project will eventually implement in future work as it makes MENACE more efficient. However, given the speed at which computers operate today, this feature was considered not critical for making a deliverable product in time.

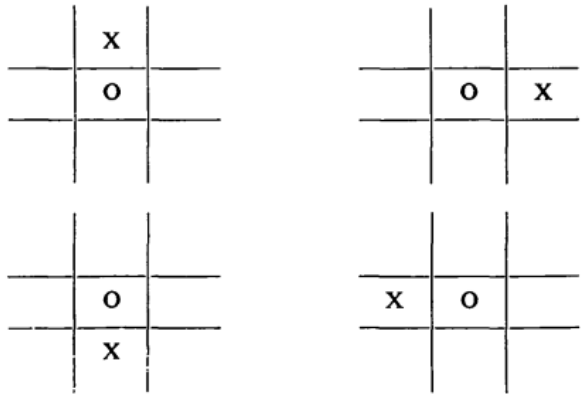


Figure 5.1 Recognising equivalent board positions, taken from Michie's original paper (Michie, 1961).

5.2 Results

5.2.1 Three per Win & One per Draw

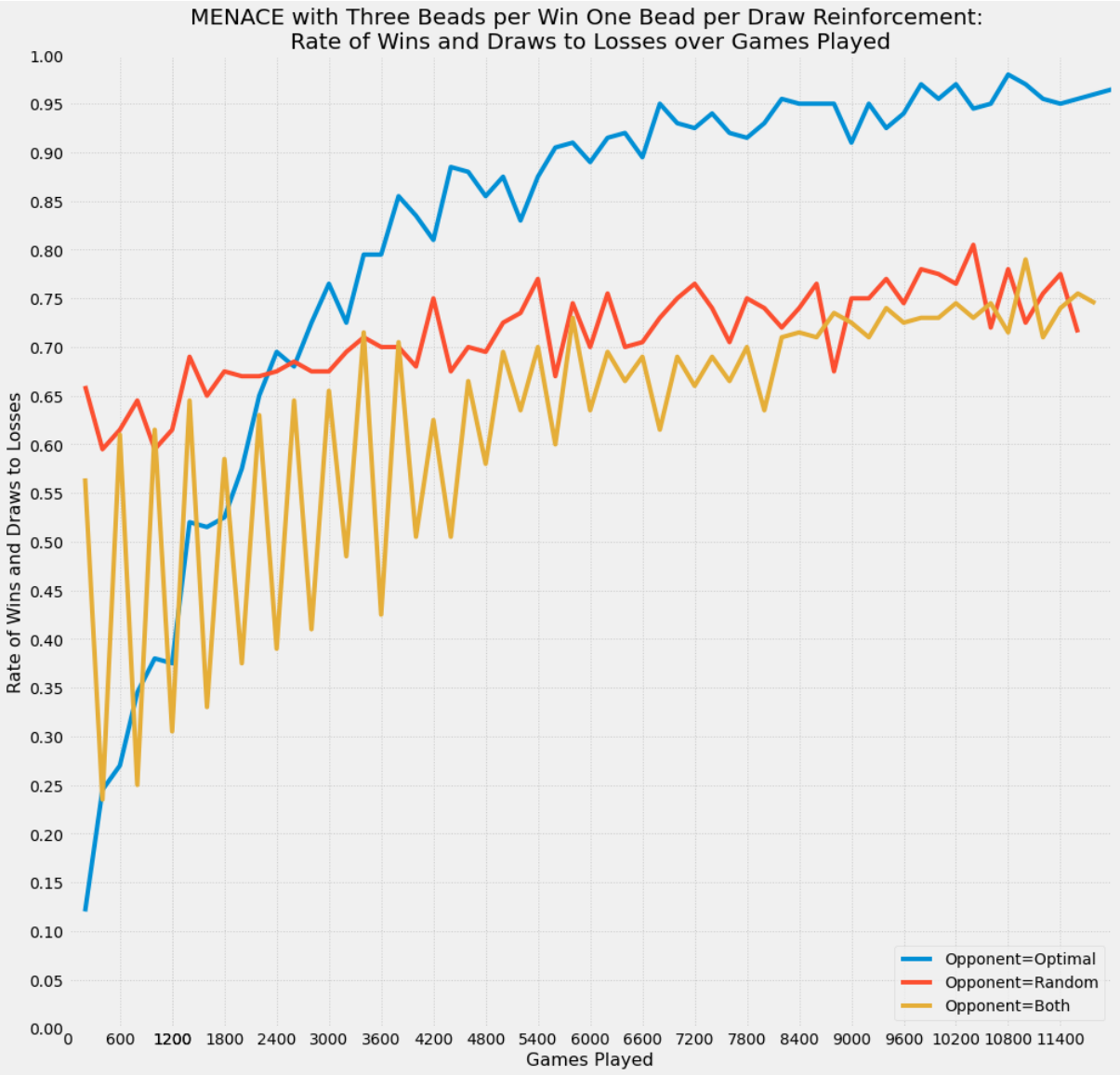


Figure 5.2 Menace with Three Beads per Win One Bead per Draw Reinforcement: Rate of Winds and Draws to Losses over Games Player

Figure 5.2 shows the progress of three MENACE instances using the ***ThreeBead*** reward function. The red line shows the progress of MENACE when trained against a **`PlayerRandom`**. It seems that this was not an effective training method as it was unable to improve its rate of wins and draws to losses past 75%.

On the other hand, when playing against a **`PlayerOptimal`**, MENACE performed miraculously. At one point it reached a win draw rate of 96%. This was a significant improvement from its initial 200 games in which it only drew 12% of its games. This shows that under the right circumstances MENACE is capable of learning.

When training MENACE on a program that alternated between 200 games with a **`PlayerRandom`** and 200 games with a **`PlayerOptimal`**, it performed worse than both other experiments. There seems to be a large difference between MENACE's performance when switching to games against

PlayerRandom and games against **PlayerOptimal**. This is evident by the consistent zig zagging in its first 6000 games. An explanation for this might be that MENACE learns a separate policy for each player type. Interestingly, by 7000 games MENACE appears to learn a policy which it can apply to both AI types.

5.2.2 Reward by Turn Number

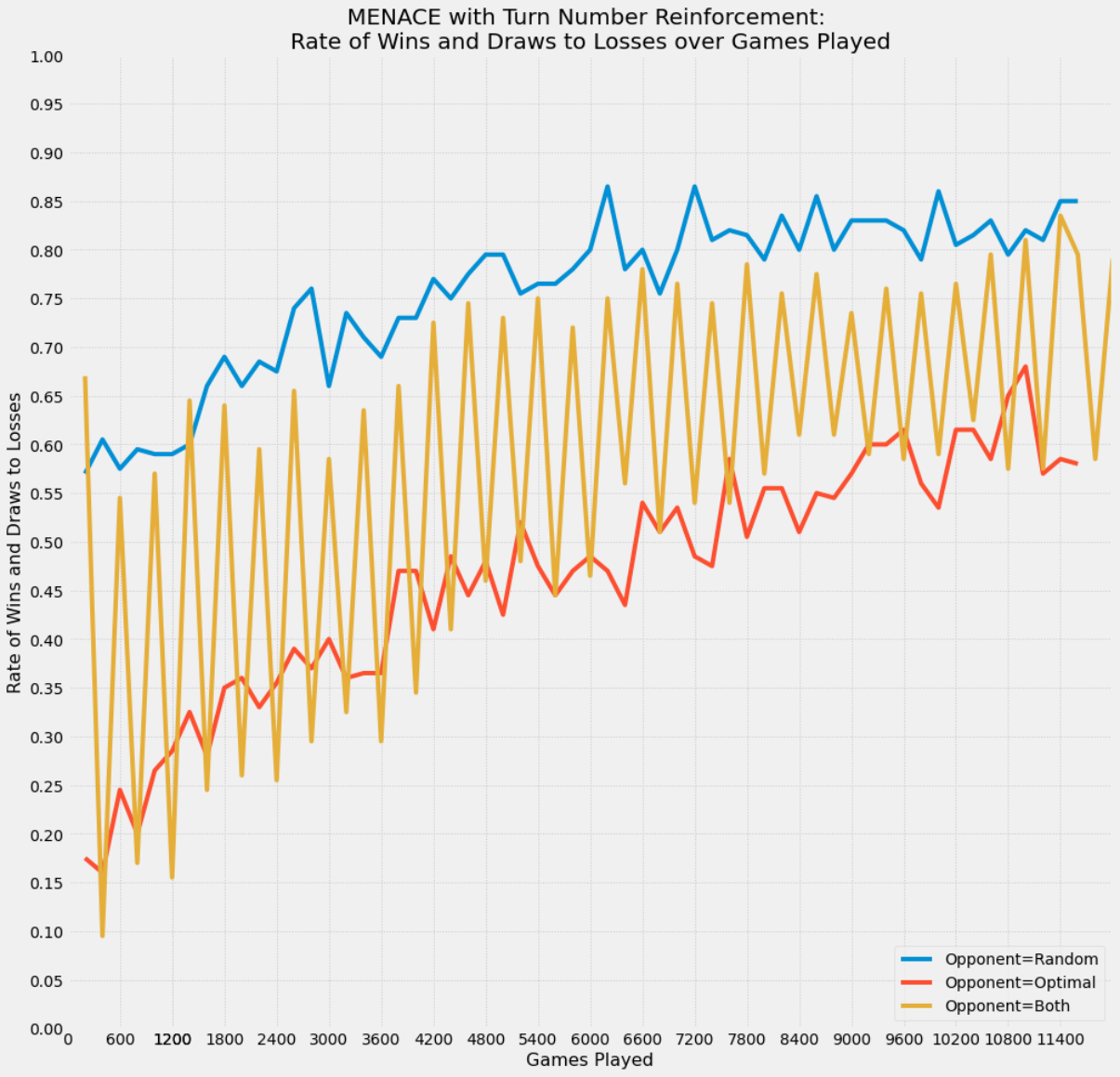


Figure 5.3 Menace with Turn Number Reinforcement: Rate of Winds and Draws to Losses over Games Player

Figure 5.3 shows the progress of three MENACE instances using the ***TurnNumber*** reward function. MENACE vs **PlayerOptimal** (depicted in red above) only reached a maximum win draw rate of roughly 60%. This is not much better than if MENACE picked a random move every time. MENACE fared better against **PlayerRandom**, with its best win draw rate of 85%. Interestingly, MENACE against both AI types appears to learn policies for dealing with them separately without converging the strategies into one policy like in figure 5.2.

5.2.3 Comparing Both Reinforcement Types

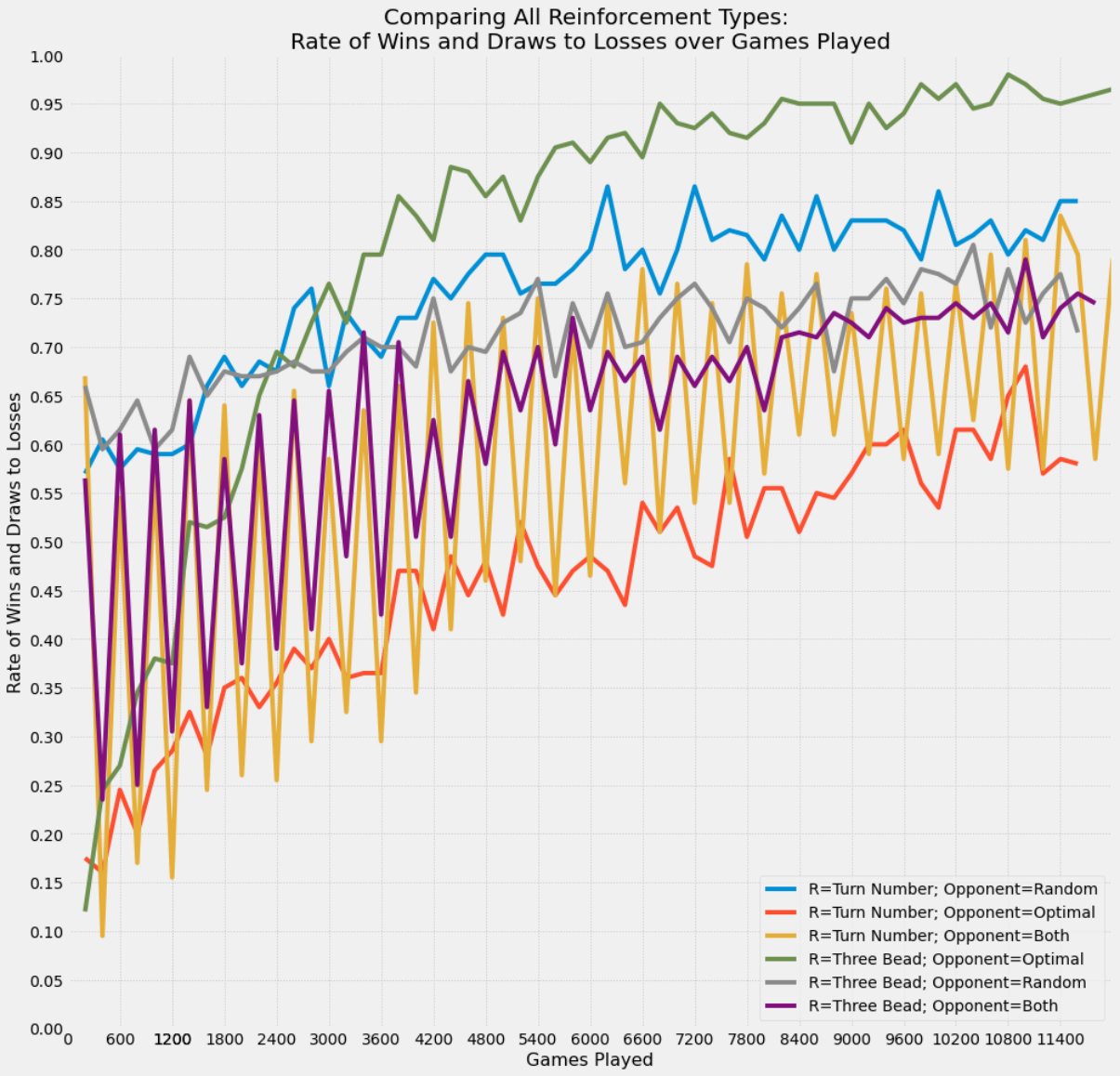


Figure 5.4 Comparing All Reinforcement Types: Rate of Wins and Draws to Losses over Games Played

Figure 5.4 shows how the two reinforcement types compare with each other. Overall, it appears that ***ThreeBead*** was a better reward system than ***TurnNumber***. A possible explanation for this is that rewarding per turn number is too much reinforcement. It is probable that MENACE quickly learns a policy that is sub optimal but difficult to unlearn because the policy yields adequate reward to satisfy MENACE. **ThreeBead** rewards MENACE significantly less and therefore gives MENACE time to explore more board positions before converging on one policy.

5.2.4 Limitations

All experiments depicted in figure 5.4 unanimously demonstrate that MENACE improves its performance over time. This proves that MENACE works. However, these experiments are limited because figure 5.4 does not show how good the policy is that MENACE learns. This would require comparing MENACE's policy against the optimal policy as discussed in section 5.1.1.

It is most probable that the policies MENACE learned in the experiments are overfitted to the opponents used in the experiments and would not be effective against other player types e.g., humans. In fact, this is somewhat observable on the website. The best MENACE from figure 5.4 "R=Three Bead; Opponent=Optimal" (depicted in green) is also shown in figure 5.5 making poor move choices against a human and performing noticeably worse against a **`PlayerRandom`** after being trained only against **`PlayerOptimal`**.

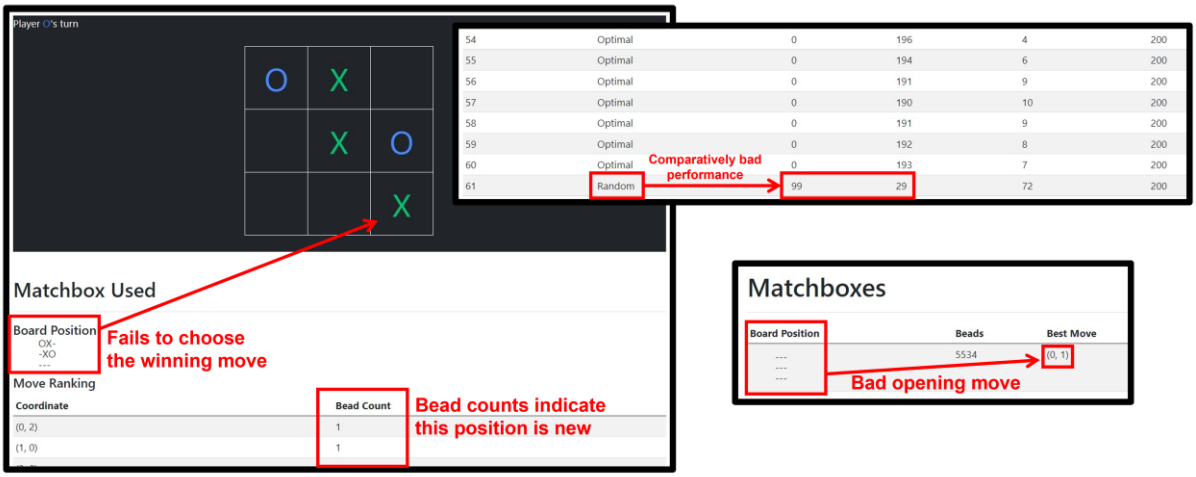


Figure 5.5 MENACE behaving badly

Moving forward, improvements that can be made to MENACE include:

1. Giving MENACE a more diverse range of opponents to play against. Michie noted that playing MENACE against MENACE yields good results (Michie, 1961).
2. Implementing a system to evaluate MENACE's policy at any given time.
3. Adding more reinforcement methods to experiment with.

5.3 Web Application Evaluation

The success of the website can be evaluated by measuring how successful it is at implementing the functional and non-functional requirements.

5.3.1 Functional Evaluation

Currently the features that have been implemented are as follows:

1. Users can play Noughts and Crosses against MENACE.
2. Users can choose whether MENACE goes first or second.
3. Users can train MENACE against random players and optimal players.
4. The matchbox MENACE used to play its turn is displayed for users to see.
5. Users can play any existing MENACE instance or create a new one.
6. There are *two* reinforcement methods and users can choose which one MENACE uses.
7. Users can see all the MENACE instances in the database with how many games they won, drew, or lost.
8. Users can view the details of any existing MENACE.
9. Users can see the full list of matchboxes in any existing MENACE from a details page.
10. Users can see a training history of any existing MENACE from a details page.
11. Users can train any existing MENACE using a random or optimal player from a details page.
12. Users can navigate to all pages from any page.

This fulfils most requirements. However, some important features are missing these are as follows:

1. Users cannot train MENACE against MENACE
2. Users cannot see a visual representation of any existing MENACE's learning progress.
3. An info page presenting research and information about MENACE and Machine Learning has not been implemented, though a lot of this research has been conducted in the literature survey.
4. The website is not hosted and therefore cannot be accessed by any users except on this author's local machine.

These features are to be implemented in future work. In addition to this, it is intended that the visual style of the website will be made more exciting by playing with the CSS and HTML.

5.3.2 Non-Functional Evaluation

Non-functional requirements take more justification to explain how they have been implemented. In addition, some requirements are subjective e.g., user friendliness. The following non-functional requirements have been implemented:

1. The MENACE implementation is efficient with no component of it exceeding a time complexity of $O(n^2)$.
    > This was achieved using good programming practices and avoiding nested iterations where possible.
2. The library is reliable.
    > It successfully implements MENACE
3. The library is easily modifiable.
    > This is evident because it was successfully modified for the website.
4. The website has a reasonable response time.
    > This is mostly true, but it was observed that requests for MENACE to play many games when it has large number of matchboxes cause the website to slow down.
5. The website does not store any personal data.
    > Users are given a human object for playing games against MENACE. No data about the user is recorded.
6. The library is easy to incorporate in the web application i.e., easy to map to a database, easy to load and manipulate objects etc.
    > This was achieved by using Entity Framework.

Some non-functional requirements are tricky to evaluate because they are subjective. The following requirements can only be evaluated to a limited extent based on this authors experience.

1. The website is intuitive enough for users to comfortably navigate the web application.
    > Users can navigate to all pages from any page on the website using the navigation bar at the top of the view.
2. The website is informative enough that users with limited AI knowledge are aware of how MENACE works.
    > The website lacks an abundance of written information such as research findings, as well as visual representations of MENACE's progress. However, the website does allow users to interact and train MENACE as well as see its internal state change as it learns.
3. The visual style of the website is exciting and interesting.

> The website views are heavily inspired by Bootstrap and Microsoft view templates. These companies are known for good user interfaces. However, a task for future work will be to update the visual style so it is more original.

To properly measure the success of implementing these requirements, a user evaluation would need to be conducted. This would firstly require hosting the website and sharing the link with volunteers. They would then spend some time using the website and answer a questionnaire. This would contain questions about how user friendly the interface was, how informative it was and whether it is successful in achieving its goal. This task was not prioritised because it was felt that there are still some interesting features to add before the website is shown to users.

6 Conclusion

6.1 Project Summary

The following section discusses how the aims and objectives have been achieved.

6.1.1 Objectives

1. Conduct a Literature Survey contextualising the project.

The literature survey (chapter 2) explores a range of topics to contextualises MENACE.

2. Design and develop a system allowing users to play noughts and crosses against a computer.

Two systems have been developed which allow users to play Noughts and Crosses against three AI types. System one is the class library from which it is possible to play Noughts and Crosses against a random player, an optimal player or MENACE from a console. However, this is not a user-friendly interface.

System two is the website. This is a user-friendly interface for the class library. It enables users to play against MENACE as well as train MENACE against a random player or an optimal player. There is still a lot that can be added to the website, but the core functionality is there, and it demonstrates how MENACE is capable of learning. This fulfils objective 2 as well as satisfies the vision and aspiration of this project.

3. Create a training dataset.

Though a dataset for training MENACE has not been created, a training method was developed and experimented with. This has also been thoroughly discussed in sections 3.5, 4.3.1, 4.4.10 and 5.1.1.

4. Implement, train, and test MENACE and various algorithms.

MENACE has been implemented and trained in the class library and the website. Testing how MENACE learns has been discussed in section 5.1.1. Testing the quality of the policy MENACE learns is something that will be implemented in the future.

Other Noughts and Crosses algorithms have been developed i.e., random player and optimal player. It was intended at the start of the project that other Machine Learning algorithms would be imported or implemented for comparison with MENACE. This is another item for future work.

5. Evaluate and compare all algorithm performances.

There were limitations to the experiments conducted because the websites current stage of development only facilitates comparison between MENACE trained against different opponents. However, MENACE trained against random players, optimal players and both has been compared in chapter 5 In addition to this, two different reinforcement methods for MENACE have been tested and compared.

6. Design and develop a user interface.

A user interface has been developed in the form of views from the website. This allows you to play and train MENACE, see its decision-making process evolve over time, and compare other MENACE instances. This accomplishes the objective. However, an evaluation of its quality has not been conducted as this would require feedback from users.

7. Conduct a user evaluation of the user interface.

Currently the website is not hosted and therefore inaccessible for users to review. However, the website mostly achieves what it set out to do. Furthermore, the user interface adapts bootstrap and Microsoft ASP.NET MVC templates whose interfaces are proven to be successful e.g., Microsoft's website, Twitter.

6.1.2 Aim

MENACE has been implemented in the class library and demonstrated in the website. This achieves the aim of designing and building an explainable AI model (MENACE) that learns to play noughts and crosses.

6.2 Review

This project built and trained a Machine Learning algorithm without the use of popular data science libraries such as Scikit-learn and TensorFlow. This was done to gain a practical understanding of how Machine Learning algorithms work and to make it explainable for beginners in the field of AI. Though the policies MENACE learned were far from optimal, this project demonstrates that MENACE is capable of learning. This achieves the aim of the project as well as most of the objectives.

In addition, a website was built to present an implementation of MENACE. This was a fantastic software engineering experience because it covered a broad range of topics. This included persisting data in a database, mapping data to useable objects, MVC design pattern, building website views to name a few. This helped achieve the aim and objectives and is a realisation of the vision and aspiration of the project.

6.3 Future Work

Though some features of the website are missing, implementing a Machine Learning algorithm from scratch and then in addition building a website is an ambitious endeavour. Some features have not yet been added because they are not deemed essential for achieving the goals of this project in a reasonable timescale. In many ways this document is a snapshot of where the project currently is. There are still many exciting parts of the website to add to and a motivation to do so.

Firstly, users will soon be able to train their instance of MENACE against another MENACE instance. Visual representations of MENACE's learning progress, like those seen in chapter 5, will also be added to help illustrate how MENACE learns. Information to help guide the user through the website will also be added based on research conducted during this project.

Another feature to be added is a system that evaluates the quality of MENACE's policy by comparing it with the optimal policy. It is also intended that more MENACE reinforcement methods will be added as well as new AI types. This will allow more experiments with MENACE as well as comparison with other algorithms.

Finally, it is intended that the website will eventually be hosted and shared. At this point a user evaluation will be conducted to gauge how successful the website is with its design.

6.4 Personal Reflection

The most challenging part of this project was deciding what to prioritise so that a viable product was completed on time. This is a Data Science project with an element of research, but it is also largely a Software Engineering project. I'm glad I stuck to making the website because it fulfils the vision and

aspiration of the project. However, there were frequent occurrences where some part of the website broke, and I would spend many hours finding the solution. Given the website was barely off the ground for many of these occurrences, it was easy to feel that I should have been improving the class library instead e.g., adding new AI models. This is testament to my lack of experience with web development before this project began.

There are many positives to take from these struggles. During my degree programme I learned about the theory of numerous AI models and used many coding libraries to experiment with them. However, I felt there was not much chance to see how they were implemented. I wanted to do this project because there was an opportunity to implement my own machine learning model. From this I have learned about Reinforcement Learning and have a deeper understanding of how Machine Learning algorithms are coded and evaluated.

In addition, I now have experience working with all aspects of a website from the database to the view in the browser. I learned about C#, MVC, Entity Framework, Microsoft SQL, razor, HTML, CSS, JavaScript, version control with git and software engineering. I now also have a way of presenting MENACE that can be added to in the future.

My favourite aspect of the project is how the code is simultaneously an implementation of MENACE and a history of my progress as a developer. There are many things I wish I knew when I started which would have reduced the modifications needed for adapting the class library to the website. Due to time constraints, many of these modifications are rushed and the code is analogous to a battle between the old approaches and the new. My takeaways are that static classes are great where possible because they don't have any state to persist in the database. Board position representations are better as strings for the frontend.

I've enjoyed working on this project and I'm excited to add more to it in the future.

References

Albawi, S., Mohammed, T. A. & Al-Zawi, S., 2017 . *Understanding of a convolutional neural network.* s.l., IEEE, pp. 1-6.

Alzubi, J., Nayyar, A. & Kumar, A., 2018. *Machine Learning from Theory to Algorithms: An Overview.* s.l., IOP Publishing.

Booch, G., 1986. Object-Oriented Development. *Transactions on Software Engineering,* pp. 211-221.

Bootstrap, 2022. *Build fast, responsive sites with Bootstrap.* [Online]
Available at: https://getbootstrap.com/docs/5.2/examples/dashboard/

Bottou, L., 1994. *Comparison of Classifier Methods: A Case Study in Handwritten Digit Recognition.* Zurich, Switzerland, Institute of Electrical and Electronics Engineers, pp. 77-82.

Botvinick, M. et al., 2019. Reinforcement Learning, Fast and Slow. *Trends in Cognitive Sciences Vol.23, No.5,* pp. 408-422.

Bowling, M., Furnkranz, J., Graepel, T. & Musick, R., 2006. Machine Learning and Games. *Mach Learn .*

Brownlee, J., 2020. *Data Preparation for.* s.l.:s.n.

Burgsteiner, H., Kandlhofer, M. & Steinbauer, G., 2016. *IRobot: Teaching the Basics of Artificial Intelligence in High Schools.* s.l., Association for the Advancement of Artificial Intelligence.

Carabantes, M., 2020. Black-box artifcial intelligence: an epistemological and critical analysis. *AI & SOCIETY,* pp. 309-317.

Chu, X., Krishnan, S. & Wang, J., 2016. *Data Cleaning: Overview and Emerging Challenges.* San Francisco, SIGMOD '16, p. 2201–2206.

Clarkson, P. C., 2008. Exploring the possibilities of using 'ticktacktoe' to think and communicate about mathematics. *Australian Mathematics Teacher,* pp. 28-35.

Cournapeau, D. & Brucher, M., 2022. *1.4. Support Vector Machines.* [Online]
Available at: https://scikit-learn.org/stable/modules/svm.html

Cournapeau, D. & Brucher, M., 2022. *sklearn.neighbors.KNeighborsClassifier.* [Online]
Available at: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Covington, P., Adams, J. & Sargin, E., 2016. *Deep Neural Networks for YouTube Recommendations,* Boston: Association for Computing Machinery.

Cox, L. K., 2022. *Web Design 101: How HTML, CSS, and JavaScript Work.* [Online]
Available at: https://blog.hubspot.com/marketing/web-design-html-css-javascript

Crowley, K. & Siegler, R. S., 1993. Flexible Strategy Use in Young Children's Tic-Tac-Toe. *Cognitive Science A Multidisciplinary Journal,* pp. 531-561.

DARPA , 2016. *Explainable Artifcial Intelligence,* Arlington, VA: Defense Advanced Research Projects Agency.

David L. Pool, A. K. M., 2017. 12.6 Evaluating Reinforcement Learning Algorithms. In: *Artificial Intelligence: Foundations of Computational Agents.* s.l.:Cambridge University Press.

Gardner, M., 1988. *Hexaflexagons.* Chicago and London: The University of Chicago Press.

Gibson, J. P., 1994. A Noughts and Crosses Java Applet to Teach Programming to Primary School Children. *New Frontiers. In PPIG,* p. 9.

Glorennec, P. Y., 2000. *Reinforcement Learning: an Overview.* Aachen, Germany, INSA de Rennes / IRISA, pp. 14-15.

Hinds, J., Williams, E. J. & Joinson, A. N., 2020. "It wouldn't happen to me": Privacy concerns and perspectives following the. *International Journal of Human-Computer Studies.*

Honavar, V., 2016. *Artificial Intelligence: An Overview.* s.l.:Pennsylvania State University.

IBM Cloud Education, 2020. *What is unsupervised learning?.* [Online]
Available at: https://www.ibm.com/cloud/learn/unsupervised-learning

J.Russell, S. & Norvig, P., 2021. *Artificial Intelligence A Modern Approach.* 4th Edition ed. s.l.:Pearson
.

JavaScript Academy, 2022. *javascriptacademy-stash tic-tac-toe.* [Online]
Available at: https://github.com/javascriptacademy-stash/tic-tac-toe

Kaelbling, L., Littman, M. & Moore, A., 1996. Reinforcement Learning: A Survey. *JournalofArti cialIntelligenceResearch,* pp. 237-285.

Kutschera, A., 2022. *The best opening move in a game of tic-tac-toe.* [Online]
Available at: https://www.maxant.ch/2018/04/07/1523086680000/

Lee, K.-F., 2018. *AI Superpowers: China, Silicon Valley, and the New World Order.* Boston; New York: Houghton Mifflin Harcourt.

Livingstone, D., Manallack, D. & Tetko, I., 1997. Data modelling with neural networks: Advantages and limitations. *Journal of Computer-Aided Molecular Design,* p. 135–142.

Lundberg, S., 2022. *SHAP.* [Online]
Available at: https://shap.readthedocs.io/en/latest/index.html

Mehrabi, N. et al., 2021. A Survey on Bias and Fairness in Machine Learning. *ACM Computing Surveys.*

Michie, D., 1961. Experiments on the mechanization of game-learning Part I. Characterization of the model and its parameters. *"Trial and Error," Science Survey,* pp. 129-145.

Microsoft, 2022. *A tour of the C# language.* [Online]
Available at: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Microsoft, 2022. *ASP.NET MVC Overview.* [Online]
Available at: https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/overview/asp-net-mvc-overview

Microsoft, 2022. *Entity Framework.* [Online]
Available at: https://learn.microsoft.com/en-us/aspnet/entity-framework

Morley, P., 2009. Evolving Neural Networks to Play Noughts & Crosses. *Intelligent Decision Making Systems.*

Opitz, D. & Maclin, R., 1999. Popular Ensemble Methods: An Empirical Study. *Journal of Artificial Intelligence Research ,* pp. 169-198.

Pedregosa, F. et al., 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research,* pp. 2825--2830.

Price, R., 1988. *The Revolutions of 1848.* s.l.:Macmillan Education LTD.

Rao, A., Verwij, G. & Cameron, E., 2017. *Sizing the prize What's the real value of AI for your business and how can you capitalise?,* s.l.: PwC.

Ray, S., 2019. *A Quick Review of Machine Learning Algorithms.* India, IEEE, pp. 35-39.

Salcedo-Sanz, S., Rojo-Álvarez, J. L., Martínez-Ramón, M. & Camps-Valls, G., 2014. Support vector machines in engineering: an overview. *WIREs Data Mining and Knowledge Discovery,* p. 234–267.

Samuel, A. L., 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development,* Volume VOL. 3, pp. 206-226.

Sauble, D., 2019. *Tic-Tac-Toe and Reinforcement Learning.* [Online]
Available at: https://medium.com/swlh/tic-tac-toe-and-deep-neural-networks-ea600bc53f51

Schaefer, S., 2002. *Tic-Tac-Toe (Naughts and Crosses, Cheese and Crackers, etc.) (January 2002).* [Online]
Available at:
https://web.archive.org/web/20200224170428/http://www.mathrec.org/old/2002jan/solutions.html

Schaffer, J., 1997. *One Jump Ahead: Challenging Human Supremacy in Checkers.* s.l.:Springer.

Scroggs, M., 2015. *MENACE: Machine Educable Noughts And Crosses Engine.* [Online]
Available at: https://www.mscroggs.co.uk/blog/19
[Accessed 15 09 2022].

Sutton, R. S. & Barto, A. G., 2020. *Reinforcement Learning: An Introduction.* Cambridge, Massachusetts; London, England: The MIT Press.

Turing, A., 1950. Computing Machinery and Intelligence. *MIND A Quarterly Review of Psychology and Philosophy,* pp. 433-460.

Tutorials Point, 2022. *Entity Framework - Code First Approach.* [Online]
Available at:
https://www.tutorialspoint.com/entity_framework/entity_framework_code_first_approach.htm

Uysal, İ. & Guvenir, H. A., 319-340. An overview of regression techniques for knowledge discovery. *The Knowledge Engineering Review,* p. 1999.

W3 Schools, 2022. *ASP.NET Razor - Markup.* [Online]
Available at: https://www.w3schools.com/asp/razor_intro.asp

Weisstein, E. W., 2022. *Tic-Tac-Toe.* [Online]
Available at: https://mathworld.wolfram.com/Tic-Tac-Toe.html

Yampolskiy, R. V., 2016. *Taxonomy of Pathways to Dangerous Artificial Intelligence.* s.l., Association for the Advancement of Artificial Intelligence.

Ying, X., 2018. An Overview of Overfitting and its Solutions. *Journal of Physics: Conference Series.*

Yuen, S., 2017. *Neural Net for learning Tic Tac Toe.* [Online]
Available at: https://github.com/12yuens2/neural-net-tic-tac-toe

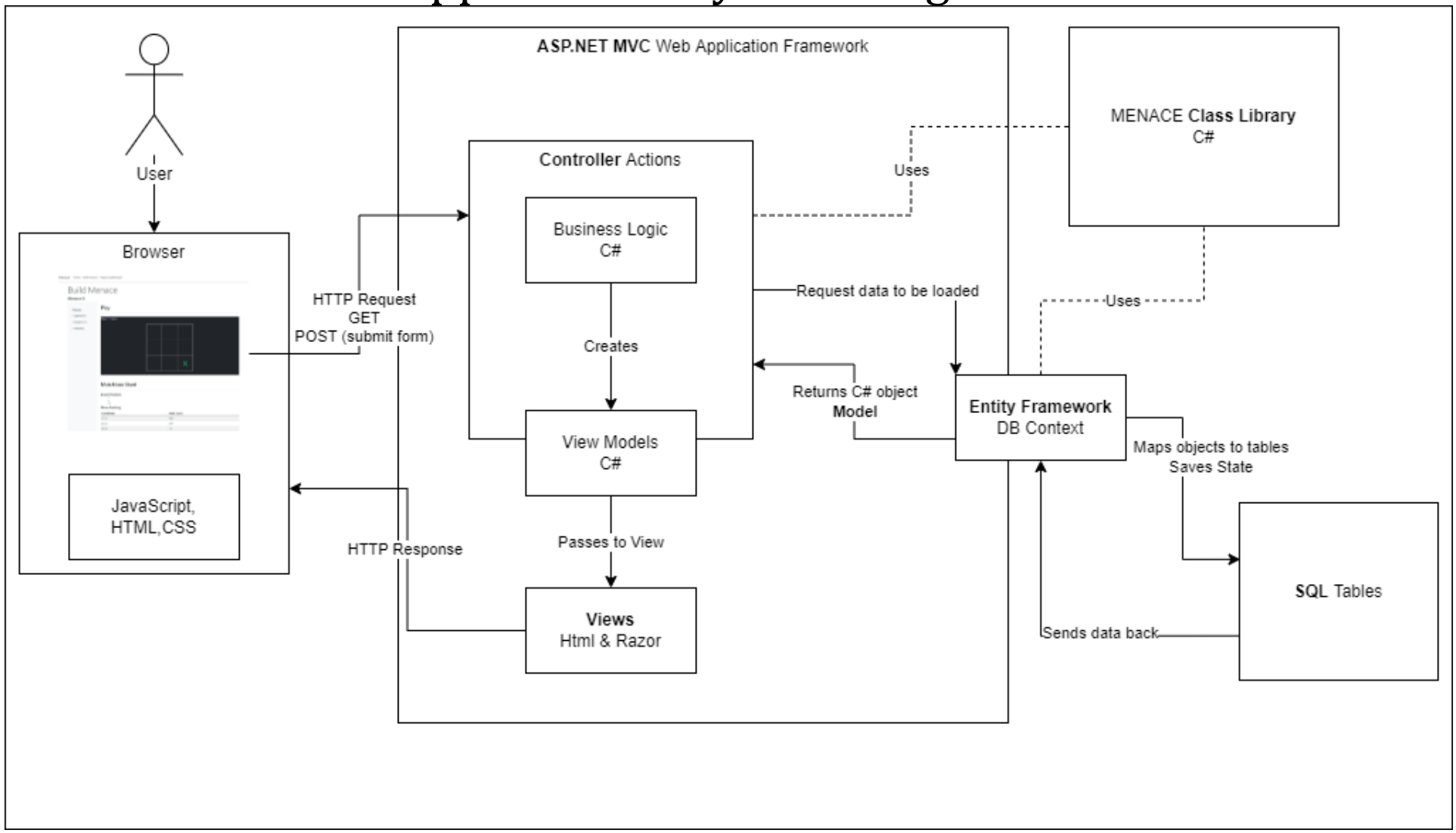
Zaeem, R. N. & Barber, K. S., n.d. The Effect of the GDPR on Privacy Policies: Recent Progress. *ACM Transactions on Management Information Systems,* p. 2020.

Zaslavsky, C., 1982. *Tic Tac Toe: And Other Three-In-A Row Games from Ancient Egypt to the Modern Computer.* s.l.:Ty Crowell Co.

Zeng, J., 2020. Artificial Intelligence and China's Authoritarian Governance. *International Affairs 96,* pp. 1441-1459.

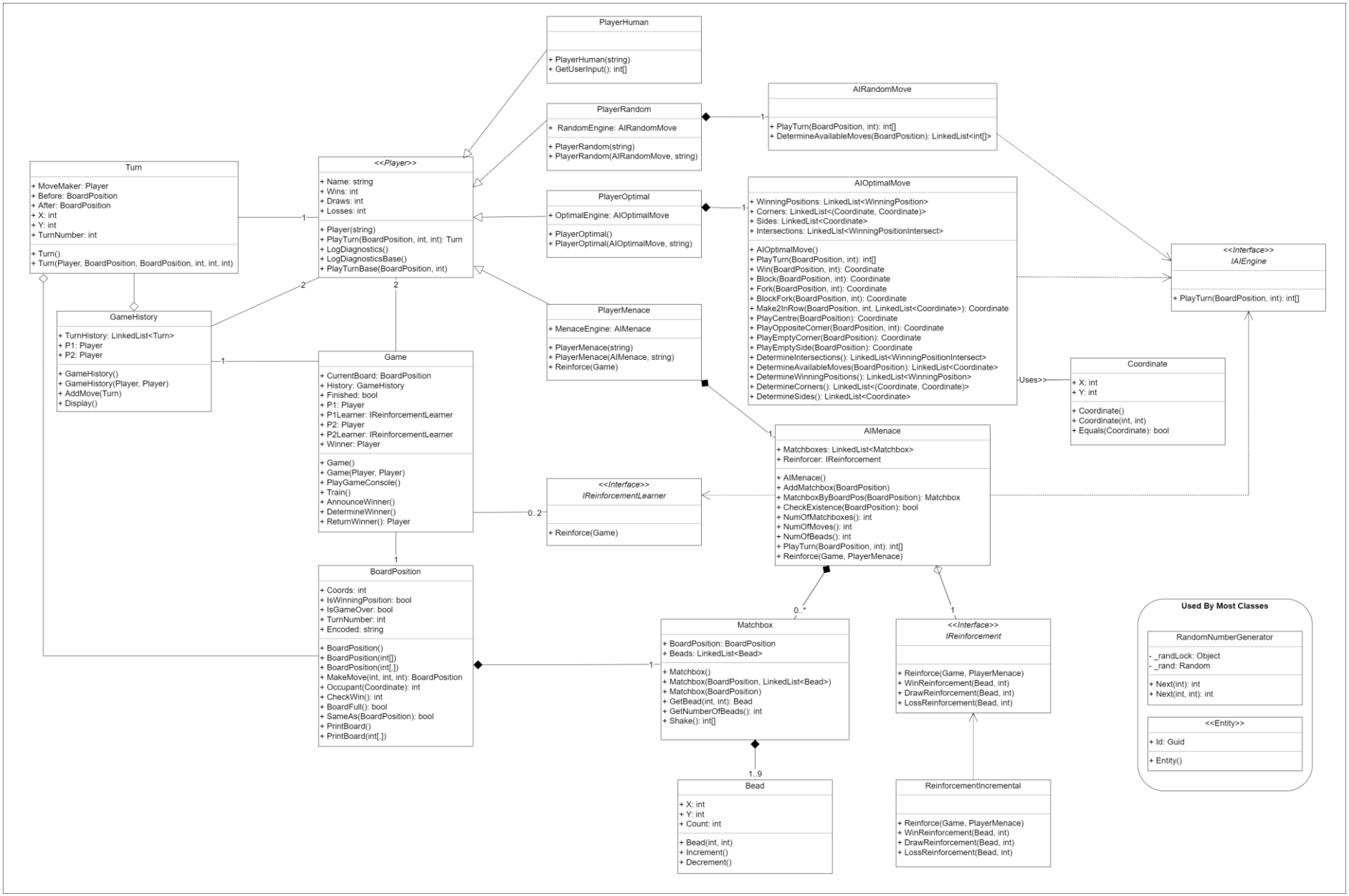
Zou, J., Han, Y. & So, S.-S., 2008. Overview of Artificial Neural Networks. *Methods in Molecular Biology,* Volume 458, pp. 15-25.

Appendix A - System Diagrams



Website System Architecture

Entity Relationship Diagram for MENACE Web Application



Class Structure of Menace Class Library

Appendix B – Terms of Reference

Department of Computing and Mathematics Computing and Digital Technology Postgraduate Programmes Terms of Reference Coversheet	
Student name:	Jan Clare
University I.D.:	21415004
Academic supervisor:	Annabel Latham
External collaborator (optional):	
Project title:	An online learning game to show how machines are trained to win noughts and crosses (MENACE)
Degree title:	MSc Data Science
Project unit code:	6G7V0007_2122_9F
Credit rating:	60
Start date:	30/06/2022
ToR date:	30/06/2022
Intended submission date:	23/09/2022
Signature and date student:	Jan Clare 23/06/2022
Signature and date external collaborator (if involved):	

This sheet should be attached to the front of the completed ToR and uploaded with it to Moodle.

Terms of Reference

7 Project Aims & Objectives

Artificial Intelligence, Machine Learning and Neural Networks usually involve building complex models trained on vast datasets to solve difficult problems. This project takes a simple problem i.e., Noughts and Crosses, and endeavours to build an explainable AI model (MENACE) with the aim of gaining insight into complex models and educating users unfamiliar with the topic of AI. This project also has a research aspect. In addition to MENACE, alternative models for playing Noughts and Crosses will be investigated and compared. Finally, an interactive application will be developed with the aim of demonstrating how MENACE works, presenting the comparisons between models, and presenting any research findings. Ultimately, a user should be able to interact with this application and it should teach them how AI algorithms can learn to solve problems.

8 Aim

Design and build an explainable AI model (MENACE) that learns to play a game of noughts and crosses.

9 Objectives

8. Conduct a Literature Review into:
    - MENACE.
    - Existing Noughts and Crosses AI models.
    - Machine Learning Algorithms.
    - Generating training data.
    - Contribution of Noughts and Crosses AI to modern AI problems.
9. Design and develop a system allowing users to play noughts and crosses against a computer.
10. Create training dataset.
11. Implement, train, and test MENACE and various algorithms.
12. Evaluate and compare all algorithm performances.
13. Design and develop a user interface.
14. Conduct a user evaluation of the user interface.

10 Learning Outcomes

LO1: "Choose, customise, and integrate core techniques to build sophisticated solutions for real-world Data Science problems."

LO3: "Process and analyse, effectively and efficiently, data of varying scales and from heterogeneous sources, formats, and systems, using a range of suitable languages, tools, and environments."

There are several ways this learning outcome will be met.

1. Noughts and Crosses is a simple game. There are 26,830 unique games and when the game is played perfectly by both players, it always ends in a draw (Sauble, 2019). 26,830 is relatively small in computational terms. Generating random training data for noughts and crosses games should be straight forward.
2. By building an online user interface users will be able to take part in training an instance of MENACE. Users will anonymously play against a MENACE model being trained and the data from the game will be used as training data.
3. Noughts and Crosses game data can be generated without human interaction by putting two AI models against each other.
4. Online sources may contain datasets with noughts and crosses game histories. This can be used to train MENACE and other models.

These methods of obtaining data will involve using various tools to generate, process and analyse data from different sources and formats.

LO4: "Build data science products with good software practices such as in code reuse, separation of concerns, modularity, testing, and documentation."

11 Project Description

12 Context

Modern AI problems often have complicated solutions e.g. Neural Networks, Decision Trees etc. Understanding them and what they can do normally relies on specialist knowledge. Consequently, many people have misconceptions about AI which are often encouraged by popular media such as Hollywood Sci-Fi movies.

Noughts and Crosses is a simple and widely understood game. This makes it an excellent tool for education. This project will ultimately present an explainable solution to an accessible problem with the goal of demonstrating key AI concepts.

Furthermore, solutions for small problems often yield ideas that scale to larger problems. During the development of this project there is the potential for discovering new ideas. This will benefit users unfamiliar with the topic of AI.

13 Literature Review

Rigorous research of MENACE will be conducted to acquire an intimate understanding of how it works. This will be necessary to implement MENACE as well as properly explain it in a dissertation. Additionally, research on other Noughts and Crosses AI will be conducted and compared with MENACE.

MENACE was capable of learning to play a perfect game of Noughts and Crosses before computers were readily available. The literature review will also include an explanation of how MENACE has contributed to the field of AI and how AI methods devised on smaller problems scale to larger ones.

14 Building a Noughts and Crosses Console App

C# will most likely be used to build a console app allowing users to play Noughts and Crosses. This will involve using Object Oriented Programming to build classes representing the board and players. Class can also be built to capture game histories which can later be used for training AI models etc.

15 Building and Training MENACE and AI Models

MENACE will be built for the console app. This will most likely be done using C# and Object-Oriented Programming. A MENACE class could store lots of matchbox objects. Users could play against a MENACE instantiation and MENACE objects could save their learning progress when the user is done.

Using research from the Literature Review, experiments will be conducted with other types of Noughts and Crosses AI. They will either be implemented completely like MENACE or instantiated from whatever libraries are found during the literature review.

Training will not be limited to user participation. Ways of training MENACE and other models will be developed as they are likely to be much faster. This will be done using several methods:

- Automatically generating random training data for noughts and crosses games which can be fed to MENACE.
- Putting AI models against each other for rapid training or to build a repository of game histories.
- Looking for repositories of game history data from online sources.

Finally, ways of measuring the performance of the various models will be developed for comparison. This will most likely be done by counting the number of wins and draws as opposed to losses against each other or (anonymous human volunteers) after they have been trained on the same data.

16 Building a Frontend Application

This will entail building an application that uses the console app to present the work from this project. It has not yet been decided what form this will take but it will be determined based on this authors skillset at the time of creation. Current ideas include:

- Build an interactive website.
- Build an application using Unity.
- Build an application using Streamlit.

Whatever form the application takes, several requirements have been determined which summarise the current ideas for the application. The application should:

- Allow users to play against prebuilt models.
- Allow users to view the contents of every matchbox for the prebuilt MENACE model.
- Allow users to contribute to the training of a MENACE model by anonymously playing against a working model. (This will require ethical considerations.)
- Allow users to view the contents of every matchbox in the working MENACE model as well as infographics for its progress over time.
- Present my research on a separate page.

17 References

Albawi, S., Mohammed, T. A. & Al-Zawi, S., 2017 . *Understanding of a convolutional neural network.* s.l., IEEE, pp. 1-6.

Alzubi, J., Nayyar, A. & Kumar, A., 2018. *Machine Learning from Theory to Algorithms: An Overview.* s.l., IOP Publishing.

Booch, G., 1986. Object-Oriented Development. *Transactions on Software Engineering,* pp. 211-221.

Bootstrap, 2022. *Build fast, responsive sites with Bootstrap.* [Online]
Available at: https://getbootstrap.com/docs/5.2/examples/dashboard/

Bottou, L., 1994. *Comparison of Classifier Methods: A Case Study in Handwritten Digit Recognition.* Zurich, Switzerland, Institute of Electrical and Electronics Engineers, pp. 77-82.

Botvinick, M. et al., 2019. Reinforcement Learning, Fast and Slow. *Trends in Cognitive Sciences Vol.23, No.5,* pp. 408-422.

Bowling, M., Furnkranz, J., Graepel, T. & Musick, R., 2006. Machine Learning and Games. *Mach Learn .*

Brownlee, J., 2020. *Data Preparation for.* s.l.:s.n.

Burgsteiner, H., Kandlhofer, M. & Steinbauer, G., 2016. *IRobot: Teaching the Basics of Artificial Intelligence in High Schools.* s.l., Association for the Advancement of Artificial Intelligence.

Carabantes, M., 2020. Black-box artifcial intelligence: an epistemological and critical analysis. *AI & SOCIETY,* pp. 309-317.

Chu, X., Krishnan, S. & Wang, J., 2016. *Data Cleaning: Overview and Emerging Challenges.* San Francisco, SIGMOD '16, p. 2201–2206.

Clarkson, P. C., 2008. Exploring the possibilities of using 'ticktacktoe' to think and communicate about mathematics. *Australian Mathematics Teacher,* pp. 28-35.

Cournapeau, D. & Brucher, M., 2022. *1.4. Support Vector Machines.* [Online]
Available at: https://scikit-learn.org/stable/modules/svm.html

Cournapeau, D. & Brucher, M., 2022. *sklearn.neighbors.KNeighborsClassifier.* [Online]
Available at: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Covington, P., Adams, J. & Sargin, E., 2016. *Deep Neural Networks for YouTube Recommendations,* Boston: Association for Computing Machinery.

Cox, L. K., 2022. *Web Design 101: How HTML, CSS, and JavaScript Work.* [Online]
Available at: https://blog.hubspot.com/marketing/web-design-html-css-javascript

Crowley, K. & Siegler, R. S., 1993. Flexible Strategy Use in Young Children's Tic-Tac-Toe. *Cognitive Science A Multidisciplinary Journal,* pp. 531-561.

DARPA , 2016. *Explainable Artifcial Intelligence,* Arlington, VA: Defense Advanced Research Projects Agency.

David L. Pool, A. K. M., 2017. 12.6 Evaluating Reinforcement Learning Algorithms. In: *Artificial Intelligence: Foundations of Computational Agents.* s.l.:Cambridge University Press.

Gardner, M., 1988. *Hexaflexagons.* Chicago and London: The University of Chicago Press.

Gibson, J. P., 1994. A Noughts and Crosses Java Applet to Teach Programming to Primary School Children. *New Frontiers. In PPIG,* p. 9.

Glorennec, P. Y., 2000. *Reinforcement Learning: an Overview.* Aachen, Germany, INSA de Rennes / IRISA, pp. 14-15.

Hinds, J., Williams, E. J. & Joinson, A. N., 2020. "It wouldn't happen to me": Privacy concerns and perspectives following the. *International Journal of Human-Computer Studies.*

Honavar, V., 2016. *Artificial Intelligence: An Overview.* s.l.:Pennsylvania State University.

IBM Cloud Education, 2020. *What is unsupervised learning?.* [Online]
Available at: https://www.ibm.com/cloud/learn/unsupervised-learning

J.Russell, S. & Norvig, P., 2021. *Artificial Intelligence A Modern Approach.* 4th Edition ed. s.l.:Pearson .

JavaScript Academy, 2022. *javascriptacademy-stash tic-tac-toe.* [Online]
Available at: https://github.com/javascriptacademy-stash/tic-tac-toe

Kaelbling, L., Littman, M. & Moore, A., 1996. Reinforcement Learning: A Survey. *JournalofArti cialIntelligenceResearch,* pp. 237-285.

Kutschera, A., 2022. *The best opening move in a game of tic-tac-toe.* [Online]
Available at: https://www.maxant.ch/2018/04/07/1523086680000/

Lee, K.-F., 2018. *AI Superpowers: China, Silicon Valley, and the New World Order.* Boston; New York: Houghton Mifflin Harcourt.

Livingstone, D., Manallack, D. & Tetko, I., 1997. Data modelling with neural networks: Advantages and limitations. *Journal of Computer-Aided Molecular Design,* p. 135–142.

Lundberg, S., 2022. *SHAP.* [Online]
Available at: https://shap.readthedocs.io/en/latest/index.html

Mehrabi, N. et al., 2021. A Survey on Bias and Fairness in Machine Learning. *ACM Computing Surveys.*

Michie, D., 1961. Experiments on the mechanization of game-learning Part I. Characterization of the model and its parameters. *"Trial and Error," Science Survey,* pp. 129-145.

Microsoft, 2022. *A tour of the C# language.* [Online]
Available at: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Microsoft, 2022. *ASP.NET MVC Overview.* [Online]
Available at: https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/overview/asp-net-mvc-overview

Microsoft, 2022. *Entity Framework.* [Online]
Available at: https://learn.microsoft.com/en-us/aspnet/entity-framework

Morley, P., 2009. Evolving Neural Networks to Play Noughts & Crosses. *Intelligent Decision Making Systems.*

Opitz, D. & Maclin, R., 1999. Popular Ensemble Methods: An Empirical Study. *Journal of Artificial Intelligence Research ,* pp. 169-198.

Pedregosa, F. et al., 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research,* pp. 2825--2830.

Price, R., 1988. *The Revolutions of 1848.* s.l.:Macmillan Education LTD.

Rao, A., Verwij, G. & Cameron, E., 2017. *Sizing the prize What's the real value of AI for your business and how can you capitalise?,* s.l.: PwC.

Ray, S., 2019. *A Quick Review of Machine Learning Algorithms.* India, IEEE, pp. 35-39.

Salcedo-Sanz, S., Rojo-Álvarez, J. L., Martínez-Ramón, M. & Camps-Valls, G., 2014. Support vector machines in engineering: an overview. *WIREs Data Mining and Knowledge Discovery,* p. 234–267.

Samuel, A. L., 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development,* Volume VOL. 3, pp. 206-226.

Sauble, D., 2019. *Tic-Tac-Toe and Reinforcement Learning.* [Online]
Available at: https://medium.com/swlh/tic-tac-toe-and-deep-neural-networks-ea600bc53f51

Schaefer, S., 2002. *Tic-Tac-Toe (Naughts and Crosses, Cheese and Crackers, etc.) (January 2002).* [Online]
Available at:
https://web.archive.org/web/20200224170428/http://www.mathrec.org/old/2002jan/solutions.html

Schaffer, J., 1997. *One Jump Ahead: Challenging Human Supremacy in Checkers.* s.l.:Springer.

Scroggs, M., 2015. *MENACE: Machine Educable Noughts And Crosses Engine.* [Online]
Available at: https://www.mscroggs.co.uk/blog/19
[Accessed 15 09 2022].

Sutton, R. S. & Barto, A. G., 2020. *Reinforcement Learning: An Introduction.* Cambridge, Massachusetts; London, England: The MIT Press.

Turing, A., 1950. Computing Machinery and Intelligence. *MIND A Quarterly Review of Psychology and Philosophy,* pp. 433-460.

Tutorials Point, 2022. *Entity Framework - Code First Approach.* [Online]
Available at:
https://www.tutorialspoint.com/entity_framework/entity_framework_code_first_approach.htm

Uysal, İ. & Guvenir, H. A., 319-340. An overview of regression techniques for knowledge discovery. *The Knowledge Engineering Review,* p. 1999.

W3 Schools, 2022. *ASP.NET Razor - Markup.* [Online]
Available at: https://www.w3schools.com/asp/razor_intro.asp

Weisstein, E. W., 2022. *Tic-Tac-Toe.* [Online]
Available at: https://mathworld.wolfram.com/Tic-Tac-Toe.html

Yampolskiy, R. V., 2016. *Taxonomy of Pathways to Dangerous Artificial Intelligence.* s.l., Association for the Advancement of Artificial Intelligence.

Ying, X., 2018. An Overview of Overfitting and its Solutions. *Journal of Physics: Conference Series.*

Yuen, S., 2017. *Neural Net for learning Tic Tac Toe.* [Online]
Available at: https://github.com/12yuens2/neural-net-tic-tac-toe

Zaeem, R. N. & Barber, K. S., n.d. The Effect of the GDPR on Privacy Policies: Recent Progress. *ACM Transactions on Management Information Systems,* p. 2020.

Zaslavsky, C., 1982. *Tic Tac Toe: And Other Three-In-A Row Games from Ancient Egypt to the Modern Computer.* s.l.:Ty Crowell Co.

Zeng, J., 2020. Artificial Intelligence and China's Authoritarian Governance. *International Affairs 96,* pp. 1441-1459.

Zou, J., Han, Y. & So, S.-S., 2008. Overview of Artificial Neural Networks. *Methods in Molecular Biology,* Volume 458, pp. 15-25.

18 Evaluation Plan

The primary measure of success will be how many of the aims and objectives are complete by the end. The success of the AI models can be quantified by measuring their performance. The success of the frontend application can be determined by conducting a user evaluation.

19 MENACE and Other AI Models

All AI models can be evaluated and compared by training and testing them on the same data. If a perfect game is played by both players in Noughts and Crosses, the result will always be a draw. Therefore, to measure a model's performance the number of wins and draws it achieves vs the number of losses can be counted. In fact, a single loss would indicate a model is incapable of always playing a perfect game.

In addition to this, an interesting evaluation would be to measure the performance of different models playing against each other when trained on the same data.

20 Frontend Application

The success of the frontend application can be determined by conducting a user evaluation. Anonymous volunteers could be asked to rate aspects of the application such as user friendliness and informativeness. They could also anonymously contribute to the training of a MENACE instantiation and rate how interactive and informative it was.

21 Activity Schedule

The following is a rough Gantt Chart outlining the plan for the project.