Using AI to Detect Computer Assisted Play in Chess

Third Year Project Final Report

James Burnham

2151141

Supervisor:

Dr. Long Tran-Thanh

30th April 2024

Contents

Abstract

Deep Blue was the first computer to beat world champion Garry Kasparov in a chess match in 1997, it was the turning point where computers started becoming better at the game than even the best humans.

Today, over 25 years later, we now have engines like Stockfish which can beat any grandmaster running only on someone's mobile phone. Given how powerful engines have become and the widening skill gap between them and humans, it begs the question of how effectively we can distinguish games played by humans and games played by computers. Can we spot instances where players have referred to an engine to cheat during games through analysis and by comparing them to similar games played before?

With this project, I will be answering these questions by creating a program which takes a series of games as an input and analyses them alongside a large database of other games using AI to identify whether those games were played by a computer or a human.

Acknowledgements

The data plots in this report were plotted using the Matplotlib Python library [24].

Chess board figures were created using the xskak package [20].

Many thanks to my supervisor Long Tran-Thanh for his encouragement and feedback throughout the project and his idea about analysing engine move distributions.

Keywords

Chess; Chess Engine; Cheating;

Detection; AI; Machine Learning

1 Introduction

1.1 Motivation

In recent years, chess has become massively popular. Especially online, Chess.com is hosting over ten million games every day [6] with an increase in monthly users of 550% since January 2020 [9]. This is great for the community as people all around the world now have the opportunity to play against each other, whereas thirty years ago they would only play at their local club. Chess.com itself was included in TIME magazine's list of 100 most influential companies in 2023 [29] demonstrating the profound effect the game has had on the world in recent times, especially during lockdown giving people something new to learn while being stuck at home. In competitive chess, there have now been some major tournaments held online such as the Champions Chess Tour in 2023 with a $2 million US dollar prize fund [4].

However, this influx of new players also brings about an influx of cheaters. For traditional in-person tournaments it is more difficult to cheat during a game, with arbiters[1] patrolling the playing hall and metal detectors scanning for any devices. When playing online, it is completely unmoderated. Anyone who can play a game of chess online has the ability to cheat. This becomes even more of a concern when prize money can be won as a result.

There are two main types of cheating:

- **Naïve Cheating** - A player who doesn't play any of their own moves, they use a computer program to analyse the board and give them the best moves to play in every single game. This is easier to detect, as the player's performance will be almost perfect and all of their games will show highly accurate play.
- **Intermittent Cheating** - A player who plays legitimately and only cheats for some games, or only cheats for certain parts in games when they're losing. This is a much more intricate problem, analysing a player's performance isn't necessarily enough and some games will naturally show no signs of cheating. For a player who is skilled without cheating, it can be extremely difficult to find evidence if they cheat sparingly.

[1] A moderator who resolves issues and ensures the rules are followed.

1.2 Background

1.2.1 The Game of Chess

Chess is a two player board game played on an eight by eight checkerboard. Each player has eight pawns ♙, two rooks ♖, two knights ♘, two bishops ♗, a queen ♕, and a king ♔. They are arranged on the chessboard in the starting position shown in Figure 1.

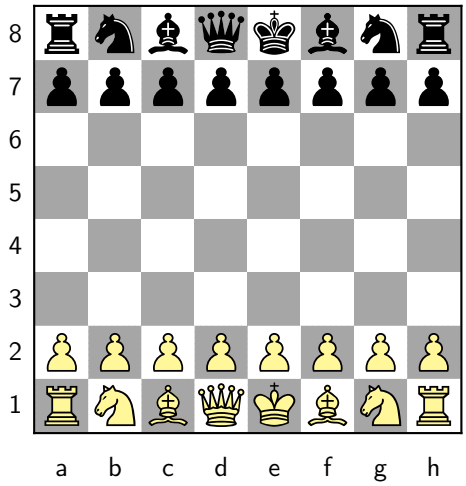


Figure 1: The starting position of a chess game.

One player plays with the white pieces and makes the first move, the other plays as black. Each piece has a unique moveset, they can capture by moving ontop of an enemy piece.

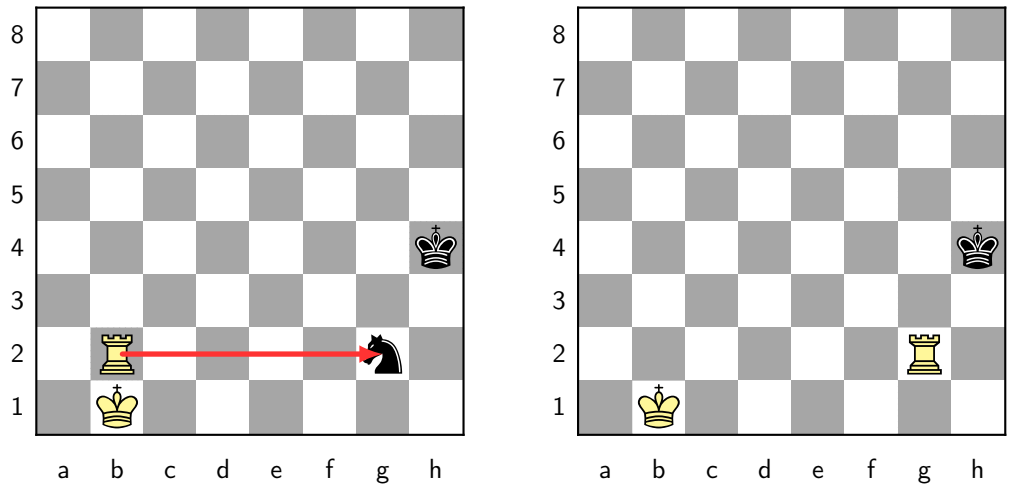


Figure 2: A rook (which moves orthogonally) capturing a knight.

Players take turns, after moving a piece their opponent moves a piece. When a piece is captured, it is removed from the board. Additionally, if a pawn reaches the end of the board, it is promoted to any piece of the player's choosing which in some cases may be seen as 'reviving' a captured piece. If a player is threatening to capture the opponent's king on their next move, then the opponent is said to be in 'check'. If the opponent cannot stop the player from capturing their king on the next turn no matter what move they make, they are in 'checkmate'. A player wins the game by checkmating their opponent. A player can also resign the game if they know they are going to lose.

There are many ways for the game to be drawn too. If the same position is reached three times in the same game, a player can claim a draw. Similarly, if 50 moves have been played by both players and no captures were made and no pawns moved forwards, a player can claim a draw. Additionally, if enough captures were made it is possible for neither player to have enough pieces left to deliver checkmate in which case the game is a draw. At lower levels, a common draw is by 'stalemate'. This occurs when a player is unable to make a move on their turn, typically because their king is trapped, but not actually in check.

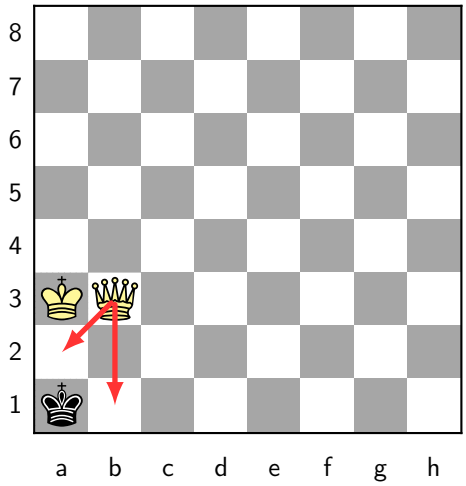


Figure 3: White stalemates black.

The board is described in terms of files (columns labelled 'a' to 'h') and ranks (rows labelled '1' to '8'). We can see in Figure 3 the black king is on the square 'a1', it is stalemate because it is black's turn to move and moving into check is not a valid move.

1.2.2 Game Phases

A game of chess can be divided into three phases, which may need special consideration in the context of identifying instances of computer assistance.

- **Opening** - The start of the game, it consists of moves and positions which have all been played and seen many times previously. In top level chess, this portion of the game may last for ten or even fifteen moves depending on how many moves they memorise before the game.
- **Middlegame** - The turning point where the game diverges from known positions, at this point the game has reached a unique position that has never been seen before. This is where players need to rely on their own knowledge and intuition and come up with their own ideas.
- **Endgame** - Typically after many pieces have been captured and taken off the board, usually when the queens have been removed as they are the most powerful pieces. This can be a very theoretical stage of the game, and like the opening some common endgame positions may have been seen or studied by the players before.

These phases are important to consider, attempting to detect computer assistance in the opening stage is difficult as the positions have been studied in great depth and all of the best moves have been documented already. Similarly we may not be able to treat a middlegame position with 26 pieces on the board the same as an endgame with only six pieces.

1.2.3 Elo Ratings

When organising tournaments and even when match-making online, it is useful to pair players of a similar skill level together. For example, a tournament may have multiple sections and award prizes for the best in each of the sections. A minor section for beginners, a major section for intermediate players, and an open section that anyone can join. The international chess federation FIDE [17] handles this by assigning players numerical ratings. Specifically, these are calculated using the Elo algorithm created by Arpad Elo [16].

The Elo algorithm rates a player based on their skill relative to the general population. Let a player's current rating be denoted by R_c and their opponent's rating be denoted R_o. The idea is to calculate the expected points P_e earned from the game, where a player earns 1 point for a win, 0.5 for a draw, and 0 for a loss.

$$P_e = \frac{1}{1 + 10^d} \tag{1}$$

Where $d = \frac{R_c - R_o}{400}$ in Equation 1. Once the game has finished, we know the actual points earned P_a. The player's new rating is updated based on the difference between the actual result and the expected result, weighted by a factor k to determine how volatile the rating is. For example, young players may be given higher k factors as they typically learn quickly and their ratings should increase rapidly to reflect their fast improvement.

$$R'_c = R_c + k \cdot (P_a - P_e) \tag{2}$$

For example, if a 2200 player faced a 2400 opponent with $k = 10$ they would be expected to lose with $P_e = 0.24$. However, if they won they'd gain $10 \cdot (1 - 0.24) = 7.6$ rating and become 2207.6. If that same 2200 player lost against the 2400 player however, they would lose only 2.4 rating. If they managed to draw, they'd still gain 2.6 rating. This system allows players to be rated based on their performance, rather than simply the number of games they win.

Notably, this means players can't inflate their rating by targeting low rated opponents because they gain less from each win and lose lots for each loss. Similarly, facing strong opposition isn't harmful because even a single win can restore all the rating taken away from five or six losses.

1.2.4 Portable Game Notation (PGN)

To record and store chess games, portable game notation (PGN) is used [15]. This is a file format and allows for a number of headings containing information about where the game

was played, the players, the result, and finally a list of all the moves played. Each move starts with a letter denoting the piece being moved, B for bishop, R for rook, Q for queen, K for king, and N for knight. The pawn has no symbol, unless it is capturing, in which case the letter of the file it was originally on is used. If the move is a capture, an 'x' is appended after the piece symbol. Finally, the coordinates of the square the piece lands on comes after.

For example, moving a knight to the square 'f3' would be denoted 'Nf3'. Capturing a pawn on 'd5' with a pawn on 'c4' would be denoted 'cxd5'. If the move is a check, a '+' is appended to it. If the move delivers checkmate, '#' is appended. Long and short castling moves are denoted 'O-O-O' and 'O-O' respectively. Moves can also be annotated with symbols, such as '??' representing a mistake, or '!!' representing a good move.

A full move is written with the move number, then white's move then black's move separated by a space. Comments can be inserted inside braces as shown in Listing 1. Additionally, variations can be written in parentheses showing what could have happened in the game.

```
1 [White "OsoPleitas"]
2 [Black "James"]
3 [Result "0-1"]
4 [Termination "Time forfeit"]
5
6 1. c4 e5 2. g3 c6 3. Nc3 Nf6 4. Bg2 d5 5. cxd5 cxd5 6. e3 Nc6 7. Nge2 e4 8. d4 Bb4 9. O-O
7 O-O?? { A mistake. } 10. a3 Ba5!! { A good move. } 11. b4 Bc7 12. Qb3 a5 (12. Qc2 a6)
8 13. Bd2 axb4 14. axb4 Rxa1 { Black wins on time. } 0-1
```

Listing 1: An example of a PGN file.

This is what the system will take as an input, a series of PGN files representing chess games to be analysed.

1.2.5 Chess Engines

Chess engines are computer programs which play chess. For many years, engines have been better at playing chess than humans. Notably in 1997 IBM's Deep Blue supercomputer beat world champion Garry Kasparov [23], this marked the turning point where they started to become superior. According to Kenneth Regan, he estimates the strongest chess engines nowadays to be 3600 Elo [36]. The highest ever human rating achieved was by Magnus

Carlsen at 2882 Elo [12]. Using the Elo formula, this would give him an expected score of $P_e = 0.016$ against engines implying that at his peak strength he would be expected to lose virtually every game.

Following Deep Blue, many chess engines have been created by many different groups of developers and are continually being improved and updated. Stockfish [41] is an open source engine which has won the 'Top Chess Engine Championship' [11] multiple times, it is currently regarded as the strongest available chess engine despite Leela Chess Zero [27] being a strong contender.

The majority of cheating in chess occurs through the use of engines, they take positions as inputs and are able to evaluate them returning the best move in that position. This evaluation is measured in 'centipawns', one centipawn represents a relative advantage of having one extra hundredth of a pawn. If the advantage is white's, this number is positive. If the advantage is black's, it is negative. Therefore, a position where white has two extra pawns may be evaluated as being roughly +200, or +2.00 as it is common to represent the evaluation as a decimal with one centipawn being 0.01. Each piece can be given an estimated value in pawns too:

♙= 1 ♘= 3 ♗= 3 ♖= 5 ♕= 9 ♔= ∞

This evaluation is not based purely on material advantage however, if white has fewer pieces but is launching a strong attack which black cannot defend, the position may still have a large positive evaluation like +5.00. This ability to numerically score positions is naturally very useful when attempting to recognise engine moves, especially when we consider the fact that we can score a move by assigning it the evaluation of the position it leads to.

It is also important to realise that this evaluation is not calculated using a deterministic algorithm or a formula using set of metrics, it involves randomness and is often implemented to be multi-threaded. If the same position is evaluated twice using the same engine on running on the same computer, it is possible for it to return a different best move on the second run. It may even suggest the same move but with a slightly different evaluation. This

is why we can't detect naïve cheating by simply comparing every move a player makes with the engine's top choice, because even if we use the same engine the moves might not match. The opposite is also true, if a player does play the top engine choice several times in a row, it is not an indication of cheating.

1.3 Related Publications

Before approaching a task such as detecting engine play, it is important to examine previous attempts. We can recognise why they may not have worked so we can avoid failing due to the same issues, or look at which parts were successful to find out what features might make a cheat detection system work.

1.3.1 Ken Regan's system

Kenneth Regan developed a system to detect cheating in chess which has become quite well known [36], especially after the lawsuit between Hans Niemann and Magnus Carlsen which eventually got dismissed [3].

In a given position, Regan's system relies on the use of an engine to obtain the evaluation of every legal move. Using these evaluations, the moves are ordered from best to worst and each move is assigned what he calls a partial credit score. By definition, the partial credit scores of all legal moves in a position sum to 1. The idea is that we not only have a measure of how good a move is based on the absolute evaluation difference between the best move's evaluation and the played move's evaluation, but we also have a measure of how good a move is relative to the other available moves in the position. For example, if a move had an evaluation difference of 0.0 and a partial credit of 0.9, it implies that the move was the best move but also the only available good move. If it had an evaluation difference of 0.0 but a partial credit of 0.2, it implies that while the move was as good as the best move, there were many other good choices available too.

Regan shows how plotting each move's partial credit against its evaluation difference produces an L-shaped curve due to engines being far superior to humans, because any played move is either as good as or worse than the engine's best move. An example is shown in Figure 4.

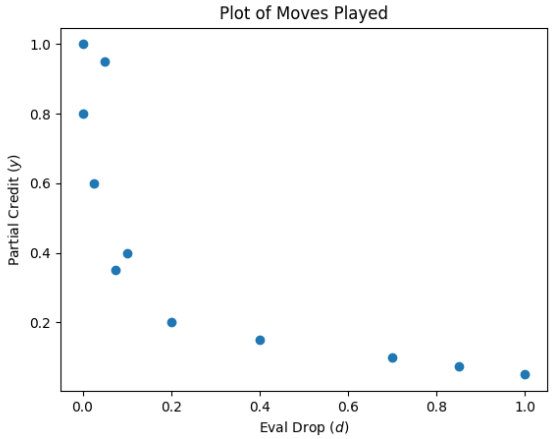


Figure 4: A similar plot to the one shown in Regan's article [36] depicting an L-shape.

After plotting several of a player's moves like in Figure 4, Regan then fits a curve to the data points using the general formula given in Equation 3 by finding values of s and c.

$$y = e^{(\frac{d}{s})^c} \tag{3}$$

Regan calls these s and c values the sensitivity and consistency respectively. Having a higher c indicates that the player makes less devastating mistakes and having a low s indicates that the player is able to tell the difference between moves even if they have a small evaluation difference. Importantly, given that the Elo rating algorithm is objective and numerical, Regan is able to calculate typical s and c values for various Elo ratings and compare them to the suspect player's s and c values. This is done by taking the z-score, which refers to the number of standard deviations above the mean the player sits at. Regan estimates that a single instance of a player having a z-score over 5.0 should be enough statistical evidence to consider them a cheater [36].

The ideas employed here are very interesting, creating a mapping between Elo and actual moves on the board, determining which moves certain Elo players should and shouldn't be able to find. However, it relies on the assumption that the engine used will identify the best move in any given position, which is not the case. Stockfish 16 [41] is considered the best

engine yet it has previously lost against Torch [7] which is a new engine being developed. Likewise Torch has lost against Stockfish 16, showing that even our best engines cannot play perfectly yet. As long as this is the case, we cannot be certain that the engine's evaluations are entirely accurate. If an engine incorrectly identifies the best move, it will score every other move as being worse, resulting in the true best move having a lower partial credit score and a greater evaluation difference. This means that a cheater using a strong engine on a powerful computer may come up with a move which is better than Regan's engine's top choice, but it would be seen as an inaccuracy and under this system it would not be suspicious.

Despite this, the system would likely work very well for detecting naïve cheating at intermediate Elo levels, as the lack of severe mistakes from engines will give the cheater a very high z-score for the c value.

1.3.2 Using Convolutional and Dense Neural Networks

In 2021, a group from the Bina Nusantara University computer science department published a paper on cheat detection in chess using neural networks [33]. Here they explore whether convolutional or dense neural networks perform better and whether analysed or unanalysed games are more effective. An analysed game is one where an engine is used to insert comments into the PGN file stating the evaluation of every position.

They show how the neural networks were set up with each input having a shape of $80 \times 14 \times 8 \times 8$. This is derived from allowing up to 80 moves in each game, then for each move there is an array of shape 8×8 to show the positions of the pieces of every type on the board. This is because there are 12 piece types (♙♗♘♖♕♔, for white and black) and they added two boards to show the positions of all white pieces together and all black pieces together. The output is either 'nobody cheats', 'white cheats' or 'black cheats'.

This is an interesting approach to the problem, with such a large number of features in the input the neural networks should be able to identify complex patterns within the games. In the end, the paper shows that convolutional neural networks performed better on the test data. The analysed and unanalysed convolutional neural networks, on the test data set, had

the same accuracy of 57.2%. While this does show signs of being able to detect cheating, the accuracy is lower than what we would aim for. If we imagine a seven round tournament, if a player cheated in every single game, we see that there would be a 64% chance of them having a 'not cheated' result in three or more games if we use the binomial formula:

$$P(x \geq 3) = 1 - \sum_{k=0}^{2} \binom{7}{k} \cdot 0.428^{k} \cdot 0.572^{7-k} = 0.6395 \approx 64\%$$

The authors of the paper also speculated as to why their accuracy on the test data is so much lower than the training data (which was 97%). They concluded that it is because while 'the models learned to recognize patterns of board positions from the training data' [33] this pattern recognition isn't as effective on unseen positions. This isn't ideal, as almost all chess games reach unique positions.

We can learn from this while designing our own cheat detection system, our system is unlikely to perform well if it is heavily reliant on the arrangement of the pieces in positions. This is particularly relevant if our system doesn't have any understanding of chess positions. For example, Figure 5 shows a very precise endgame. If it is white's turn in this position, they can forcefully checkmate the black king in 15 moves. If it is black's turn in the same position, they can push their pawn forwards and the game will be a draw.

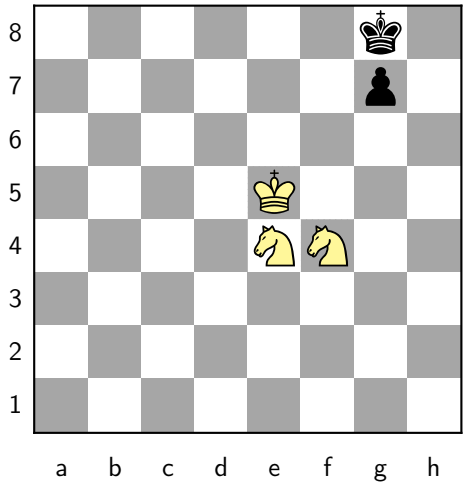


Figure 5: If it is white's move, white wins. If it is black's move, the game is a draw.

As we can see, developing pattern recognition on positions alone is dangerous as the same position can be reached through poor play or great play. To avoid succumbing to the same problems discussed in the paper, we could focus on more general features obtained through the use of engines, allowing them to deal with the details of the board. In particular, using the evaluation of a position (in the paper, adding evaluation as a feature improved the dense neural network's test accuracy by 4.65% [33]) would resolve situations like these for us and helps us to develop pattern recognition that will still be effective on unseen positions.

1.3.3 The Limits of Analytical Cheat Detection

Detecting computer assistance is a difficult problem on its own, it is not helped by the fact that false positives are potentially going to be very frequent. After all, there's a finite set of legal moves in each position and a subset of those are going to be engine top choices. Even the most tactical, genius, engine-like move could be played by a complete beginner choosing it at random. This is particularly relevant in positions where there are only a handful of available moves. A paper was published in 2015 on this topic, discussing the limits of using engine analysis for cheat detection [2].

The paper focuses on using an engine to calculate two metrics for games, CV and AE. CV stands for 'Coincidence Value' and is the proportion of moves in a game which have an evaluation difference of 0 to the best move (moves which are as good as the engine's top choice). AE on the other hand is the 'Average Error', it is the average evaluation difference measured in centipawns between the best move and the played move in each position. Therefore, if we took a game played by an engine and also analysed it with the same engine we would expect to see $CV \approx 1.0$ and $AE \approx 0$.

This is clearly a very simple method to detecting cheating, nonetheless the paper explores its effectiveness. The conclusion they came to was that it is not possible to detect cheating by simply comparing a player's moves to those played by an engine over the course of a single game. Due to the variance in the evaluations given by engines, amplified in multi-threaded implementations, they could not calculate accurate CV or AE values for games [2]. Interestingly, they also show examples of games played by Paul Morphy from the 1800's back

before chess engines existed which had $CV = 1.0$ and $AE = 0$. If these metrics were being used to detect cheating, these games would be strong positive results yet we know for a fact that Paul Morphy wasn't cheating.

The conclusions drawn by this paper should be kept in mind when designing a new system for detecting cheating, it should not be centred around comparisons to engine suggestions as these metrics would be too volatile and unpredictable. The fact that the evaluation also changes when Stockfish is set to a different depth should also be taken into consideration [41][2]. Based on this, it may make more sense to limit the extent to which Stockfish should evaluate a position by the maximum depth the search tree is allowed to reach rather than an amount of time to help reduce the variance in evaluations which may improve the system's reliability.

2 Planning and Design

2.1 Requirements and Objectives

The system will primarily analyse the attributes of moves being played, aiming to help us identify instances of cheating and potentially give an insight into how engines think differently to humans. There are many different features we would like to have, it would be nice to have a system which explains to us why a certain move is an engine move, or a live system with an API interface which inspects tournament games for cheating as they are happening, but attempting to develop a system which can do everything is unrealistic. Therefore, five requirements for the project were outlined as shown below. The three base objectives are required, whereas the two stretch objectives are designed as extensions.

BASE 1. The program must allow the user to input games and have them analysed, by highlighting moves and sequences which are likely to have been suggested by a computer.

BASE 2. Research must be conducted to find an appropriate machine learning model to implement to analyse the games and recognise instances of computer assistance. This will involve data mining from a large database of games to generate test data and training data for a machine learning model.

BASE 3. The system must involve an algorithmic component, using an open source engine to help gather data about the games and generate metrics to be used by the machine learning model as well as process the opening and endgame stages of a game.

STRETCH 4. It would be nice to implement multiple machine learning models. Possibly combining them if it is effective in order to improve the accuracy of the program.

STRETCH 5. It would be nice to spend time after making the program to study the analysis it performs and attempt to decode some of the meta-features the machine learning model creates, to gain some insights into the differences in playing styles between humans and computers.

2.2 Risks

As well as the main objectives and requirements of the project, it is important to consider any risks which may result in the project failing. The four main risks for the project were outlined at the beginning while creating the project specification, with mitigations specified for all of them.

1. RISK: The device the project is stored on and developed on may fail, progress would be lost and development would be halted.

MITIGATION: Use GitHub to effectively use version control and cloud storage. It allows work to be committed before signing off and picked up on another device such as the DCS computers. All work would still need to be done on campus however.

2. RISK: The project may require more computing power for tasks such as data mining than a laptop or personal computer can supply. This may lead to devices running for hours, during this time no other work can be done.

MITIGATION: DCS provides compute nodes which can be used. Since the most likely part to require this is data mining, which is done in the middle of term 1, the compute nodes being occupied by other students is not a significant risk as deadlines are usually towards the end of term.

3. RISK: The chosen machine learning model may unexpectedly perform badly leading to the system as a whole being ineffective. This could also be due to the dataset not being ideal or the metrics chosen being poor.

MITIGATION: Dedicate a generous amount of time to researching, experimenting with and making prototypes of multiple machine learning models. Therefore it is more likely that the initial choice will succeed but in the event that it doesn't there will be backup options as well. Defining a range of metrics will help to introduce redundancy, in case certain metrics confuse the model and need to be swapped out.

4. RISK: Encountering bugs and general programming issues which take a long time to fix and push the project behind schedule. This is especially concerning given the lack of prior experience with data mining and machine learning.

MITIGATION: The project has been scheduled such that there is a two-week section in term 2 dedicated to the extension research task. This can be used for programming if necessary instead. Additionally, development can be done throughout the Christmas break too.

Finally, we can evaluate these risks (with their mitigations) and the likelihood of them occurring. This is shown in the table below.

Risk	**Initial Severity**	**Likelihood**	**Mitigated Severity**
1. Personal device failure	Very high	Low	Moderate
2. Insufficient computing power	Moderate	Moderate	Low
3. ML model performs poorly	High	Moderate	Low
4. Programming bugs halt progress	Moderate	High	Low

Table 1: Risks listed with their severities and likelihoods.

2.3 Approach

The system has been designed to analyse each move individually and independently, this recognises the fact that intermittent cheating can take place. A player can cheat for some parts of a game (or even single moves), so a legitimate move does not necessarily mean that the previous move was an engine move and vice versa. This is more effective than analysing all moves together as one full game, because the engine moves won't be diluted by all the human moves.

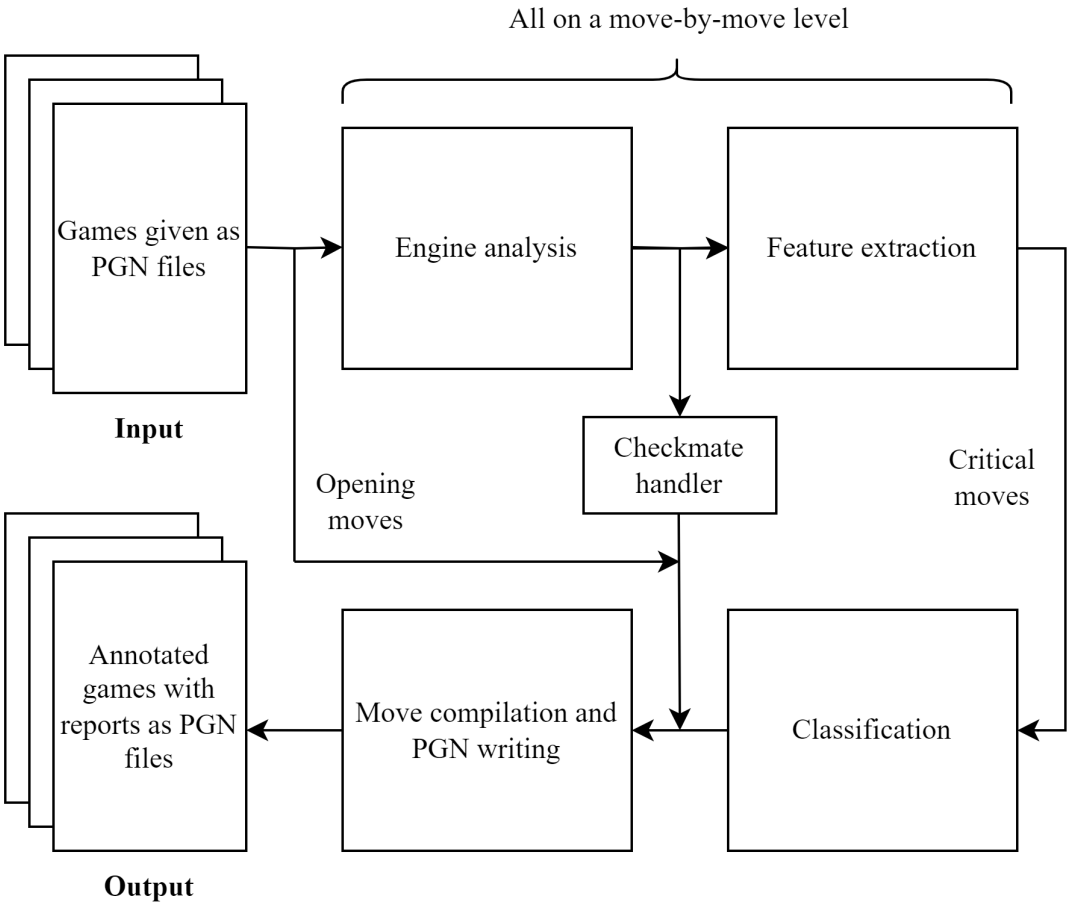


Figure 6: General overview of the pipeline.

The idea is to annotate each game by highlighting which moves are likely to be engine moves, then provide a summary of the game as a comment in the PGN file. This summary

will include statistics such as how many engine moves were detected for each player and what percentage of moves played were engine moves. However, as discussed previously there are different stages to each game which benefit from being handled differently.

For the opening stage of the game, there isn't much analysis which can be done effectively. There are too many legitimate ways that an engine-like move could have been played. It is possible that every move in the position was good, and the player happened to pick the best move at random. Even if a beginner played an extremely technical and engine-like opening, tricking their opponent with a brilliant tactic, it is possible that they simply learnt that specific opening before from watching a video covering it online. At the top level, it is also possible for grandmasters to study opening positions with engines and memorise the best moves. This requires a great memory, but when they encounter the same positions in real games they can use their prepared knowledge. These moves could be flagged as engine moves, as they were suggested by an engine, but this practice is not cheating. For these reasons, opening moves are filtered out through the use of opening books, which are collections of known opening positions. If a position which is about to be analysed is in this database, it is then ignored.

Additionally, we must consider checkmate positions. There are certain situations where one player can force checkmate in a few moves. For example, in Figure 7 we see that white can play the highlighted move 'Rc8+'. Black can only respond by blocking the check with their own rook, which white can promptly take (delivering checkmate). There is no way for black to avoid this, so the position's evaluation is M2 meaning 'White can checkmate in 2 moves'. If the colours were swapped, we would call it -M2. This evaluation is given by engines whenever checkmate can be forced and isn't compatible with the numerical evaluation in centipawns that the rest of the system would be

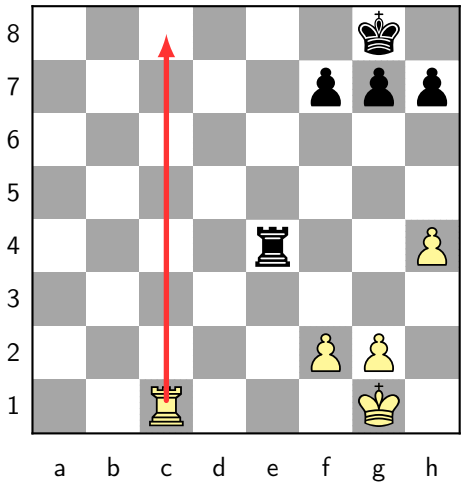


Figure 7: The evaluation is M2.

working with.

Therefore, it makes sense to take an algorithmic approach to handling these checkmate positions. Engines always prefer the best move, in this instance, it is the move that leads to the quickest checkmate. We can form a tree with the root being the current position, then each arrow is a move and each node is the position after a move is played. Each node is labelled with its evaluation. We can also assign an arbitrary 'risk' to each move, roughly indicating what probability the opponent would have of winning if the player made a mistake. For example in Figure 8 below, the current position is M3. There are two candidate moves, one leads to M2 and the other to M6. The best move is the one leading to M2 (which is what an engine would play) but the risk is very high. This could be because the move involves sacrificing an important piece, which means if the player miscalculates and can't checkmate their opponent, they will lose. On the other hand, the worse move leading to M6 has no risk. The move could involve capturing an opponent's piece for free, ensuring the player's victory even if they don't play perfectly despite taking more moves.

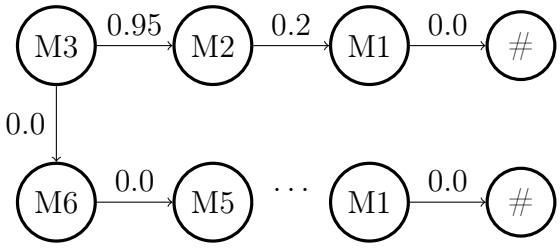


Figure 8: Representation of a position, each move (arrow) is labelled with its associated risk.

There is no benefit to winning in fewer moves, so we would expect someone to take the path of least risk if the best move is very complicated. We can also see that these situations are much simpler to work with, there are only a handful of possible evaluations and we have a clear understanding of how an engine would behave differently to a human, which is why an algorithmic approach is suitable for this component.

During the middlegame, we don't have the discrete evaluations like we do for checkmate positions, and there are also many more reasons why a position can have a high evaluation. It is not just the distance to checkmate, it may be the activity of a player's pieces or the

number of extra pieces they have, or even just the lack of safety of the opponent's king. Additionally, the evaluations are more volatile. If an engine finds M7 once, it is unlikely to change to M6 or M8 if the same position is evaluated again. When working in centipawns, the evaluations can vary more. This is the part of the system where machine learning is employed, extracting features from each move using information about the position and information from analysing it with Stockfish [41] to classify it as either an engine move or a human move.

The final special consideration is for positions which lie inbetween middlegame and checkmate positions, a 'conversion stage' where one player is converting a significant advantage into checkmate. If a position had an evaluation of +9.00 for example, even if black played perfectly it is likely that the evaluation would increase each move to +23.00, +67.00, +90.00 up until it switched to a checkmate evaluation like M15, because the game will progress and the evaluation must eventually tend towards either checkmate for white, a draw, or checkmate for black. This is also a problematic stage of the game, as the engine evaluation (a critical piece of information) starts to become unstable. Almost every move is good because it leads to a winning position, which means the deeper the analysis is the greater the evaluation will be. This stage of the game would therefore likely contain many false positives if it were analysed by the system using the same machine learning model applied to the middlegame. Finally, if an intermittent cheater was attempting to use an engine as little as possible to avoid being caught, it would make sense for them to play this conversion stage legitimately because it is easy to play when almost every move is a good move. For these reasons, we restrict the machine learning analysis to only the 'critical moves' played while the magnitude of the evaluation is less than 3.00, a threshold chosen to describe a significant advantage for one player.

With these edge cases being handled, the number of false positives the system produces should be reduced and the outputs it gives should be more meaningful.

2.4 Project Schedule

After writing a progress report for the project, the schedule was refined in more detail and is shown below.

Term 1	
Weeks 1-2	Write project specification
Weeks 3-4	Research features of a move
Weeks 5-6	Database setup and data mining
Week 7	ML model research and experimentation
Week 8	Work on other modules
Week 9	Write progress report
Week 10	Other deadlines: CS342 coursework, CS352 presentation, CS352 essay
Christmas	
Week 1	Research ML models for multi-depth analysis and other features
Week 2	Build dataset of games and run multi-depth analysis for training/testing
Weeks 3-4	Implement and refine model for multi-depth analysis specifically
Term 2	
Week 1	Implement handling for endgames and positions with forced checkmate
Week 2	Integrate the other move features into analysis using a researched model
Week 3	Implement algorithmic component and apply analysis to multiple games
Week 4	Decide the output format of the program and implement it
Weeks 5-6	Begin final report - development
Weeks 7-8	Extension: analysis of system outputs on additional games if time permits
Weeks 9-10	Presentation
Easter	Continue final report - evaluation
Term 3	
Weeks 1-2	Finish and submit final report

Table 2: Project timetable.

This schedule was designed to provide contingency for each task, with the minimum time allocated to a given component being one full week. As a high level overview, most of the research and preparation work was planned to be done in the first term whereas the bulk of the development and implementation was planned for term two.

Additionally, there were two weeks allocated to additional testing and analysis of the system outputs in the second term. While there is time to test each component and the system itself within the week-long sprints, this time can be used to refine the system and fulfil the extension requirement of actually developing some understanding of what some of the characteristics of engine moves are. However, this work is not critical for the project's success and as such this time acts as a period to catch up if development falls behind.

2.5 Programming Language

For this project, the chosen language for development was Python [21]. This choice was made due to availability of the libraries python-chess [19] and scikit-learn [34] and my large amount of experience with the language. Python-chess has a variety of functions and data structures allowing PGN files to be read and written, as well as an interface for communicating with Stockfish [41]. Scikit-learn provides implementations of many machine learning models and utility functions, for example splitting a dataset into testing and training data and performing a grid search to optimise hyper parameters. The grid search function can also be used with the XGBoost [5] library, this is the main machine learning model used in this system which will be discussed in the implementation section.

The main downside of Python is that as an interpreted language, it may be considered slower than compiled alternatives. However, for this project the bulk of the processing time will be spent by the engine analysing the games. Therefore, even if a compiled language was used the system would not be noticeably faster.

2.6 Data Sourcing

To build this system, there will need to be a large dataset of games to be analysed for training and testing. This will involve sourcing both engine games and human games.

For games played between two engines, we can look towards Chess.com's Computer Chess Championship [10]. This has a huge database of games free to download, played between a wide variety of engines running on powerful hardware. Additionally, the openings played in each game are chosen at random from an opening book to ensure the engines don't attempt to play the same moves in every game. With a variety of results including wins, losses, and draws for Stockfish this is a suitable dataset to sample from for examples of engine moves.

For games between two humans, PGN files of FIDE world championship matches are ideal. We can be almost certain that the players in these games were not cheating (unless they went to extreme lengths to do so) due to the strict anti-cheating measures in place at world championship events imposed by FIDE [18]. Not only are these match PGN files publicly available to download, but they also represent the highest quality games that humans have played. This is useful because the core of this problem is trying to differentiate between excellent human moves and engine moves, we don't want games with many poor quality moves because we can already observe that engines don't make obvious mistakes.

Unfortunately, arranged matches between chess masters and engines are difficult to find. An organiser wouldn't pay to host a match like this since neither the audience nor the player would enjoy the computer winning every single game.

> *"Would you participate in a public game against a computer in the near future?"*
>
> *"I personally never wanted that. I find it much more interesting to play humans. And also, of course, now that they have become so strong in a game like that, I wouldn't stand a chance."*
>
> - Klaudia Prevezanos interviewing grandmaster Magnus Carlsen [35].

The reason the matches with Deep Blue in 1997 [23] were famous was because Deep Blue

wasn't massively stronger than Kasparov, so the games were exciting and the match was relatively close. The final score was 3.5-2.5 (Deep Blue won) not 6-0 as it would be if an engine-human match was played today.

Finally, it is useful to see games played by cheaters who have had their accounts closed on Chess.com [6] for violating their fair play policy. Chess.com provides an 'accuracy' score for each game similar to the CV and AE values discussed previously, so we can observe that if a banned player regularly has accuracies above 90 it is likely they were using an engine to cheat intermittently. This is similar to a human-engine match in some ways, however we can't assume that an engine was used for every game. Regardless, this is a good source of games to test the system on as we still expect to see a higher number of engine moves in games played by a cheater.

2.7 Machine Learning Application

There will be two primary applications of machine learning in the program. The first application will be using Support Vector Classification (SVC) on 'evaluation levels' vectors. This is based on how Stockfish uses iterative deepening, using a complex 21-step search method called when the depth increases [37]. On each iteration, an evaluation and at least one suggested best line[2] will be returned. Some lines will be extensions of previously evaluated lines, so compiling them would form a search tree such as the one in Figure 9 below, where each arrow is a half-move and each node is a position which will have an evaluation in centipawns:

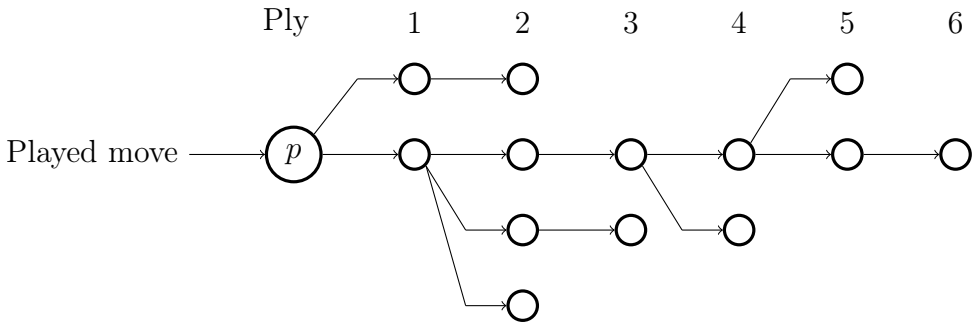


Figure 9: A theoretical evaluation search tree from an engine of position p.

[2] A 'line' is a sequence of moves.

This provides evaluations of many positions and gives us lots of information about how the engine arrived at the best move, which can be compiled into a single vector. Each entry in this 'evaluation levels' vector L is the evaluation of the resulting position in the best suggested line at depth d. It is important to note that the depth is not the same as ply. Depth is a value representing what iteration the iterative deepening search is on, it is not necessarily the number of half-moves (ply) ahead that the engine is analysing. Instead of strictly using the raw depth or ply, we use 'selective depth', it can be thought of as combination of depth and ply. It is defined as the length of the longest line calculated by the engine in that iteration. Therefore, if we were to analyse until reaching seldepth 3 such as in Figure 10, we might see these three lines returned (each line would be the path from p to the highlighted node).

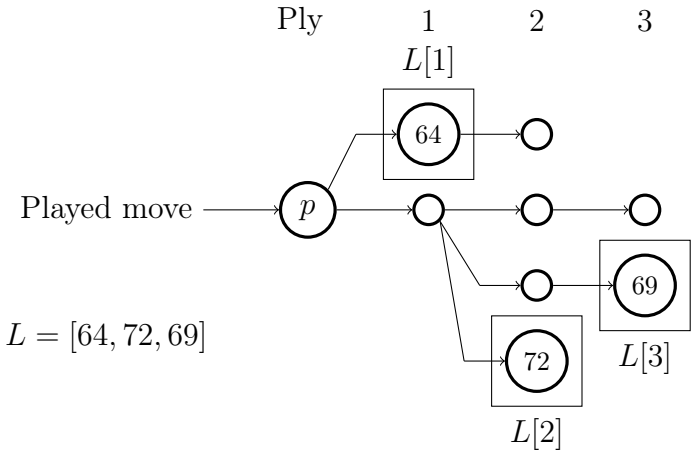


Figure 10: An example of what a search to seldepth 3 might return.

We also observe that the values in L are not strictly increasing, as the engine analyses deeper the move it finds may not change. It is possible that it may analyse more moves of same line and encounter new issues in the position, decreasing the evaluation.

Intuitively the value of $L[d]$ is the engine's opinion of the position p evaluated to depth d, as we increase d we increase the relative strength of the engine and increase the amount of consideration it gives each move. This gives us a range of evaluations emulating many playing strengths, and analysing the trend in the evaluations is very informative. A move that enters a position which is seen as losing for one player at low or medium depths but

winning for them at high depths is likely a move that only an engine would play, because a human would probably stop calculating after reaching a bad position in their mind and look for a different move.

In this program, we will limit the engine to a depth of 20, so for every position p we can compute a vector L of fixed size. Then, SVC can be applied to each feature vector L for every move classifying the trend in the values as either 'ascending', 'descending', or 'other'. For example, Figure 11 shows a move played in a game between Torch and Stockfish, two chess engines. The move was played by black, and was analysed in particular due to how irregular it was (sacrificing a full rook for no obvious reason). In Figure 12, we can see how up until depth 10, this move gives a substantial advantage to white and is evaluated by the engine to be a game losing mistake. But after this, the engine begins to like the move and considers the position roughly equal (within ±50). This is an excellent example of an engine move that even a strong player would likely avoid, especially since the obvious move '♕c2' is evaluated by Stockfish to be almost just as good.

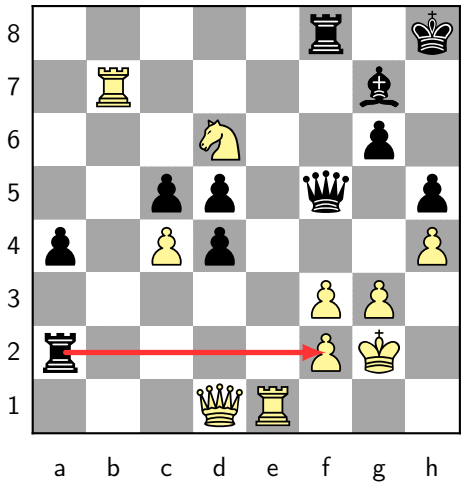


Figure 11: Move played by engine.

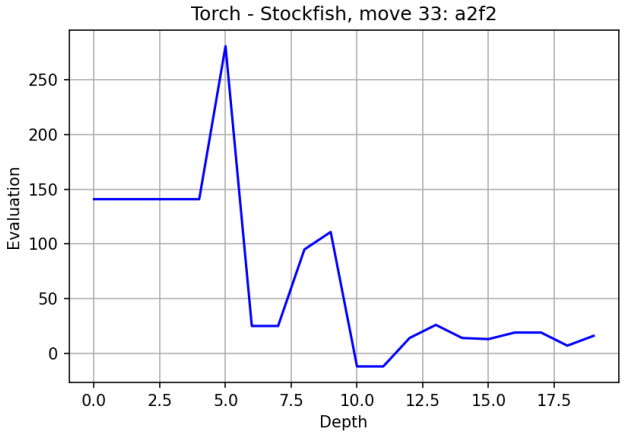


Figure 12: Visualisation of L.

The main model being used will be extreme gradient boosting (XGBoost) [5] on several features discussed in more detail in the implementation (Section 3), many of these features are extracted from this method we will refer to as "multi depth analysis", including the classification from the SVC model.

2.8 Class Structure

Overall, the structure of the program will be quite simple. At the highest level, upon being invoked with a list of PGN filenames as an input, it will instantiate a `GameAnalyser` object which will analyse all of them. They will be annotated and given reports which are written to PGN files, then the program will exit.

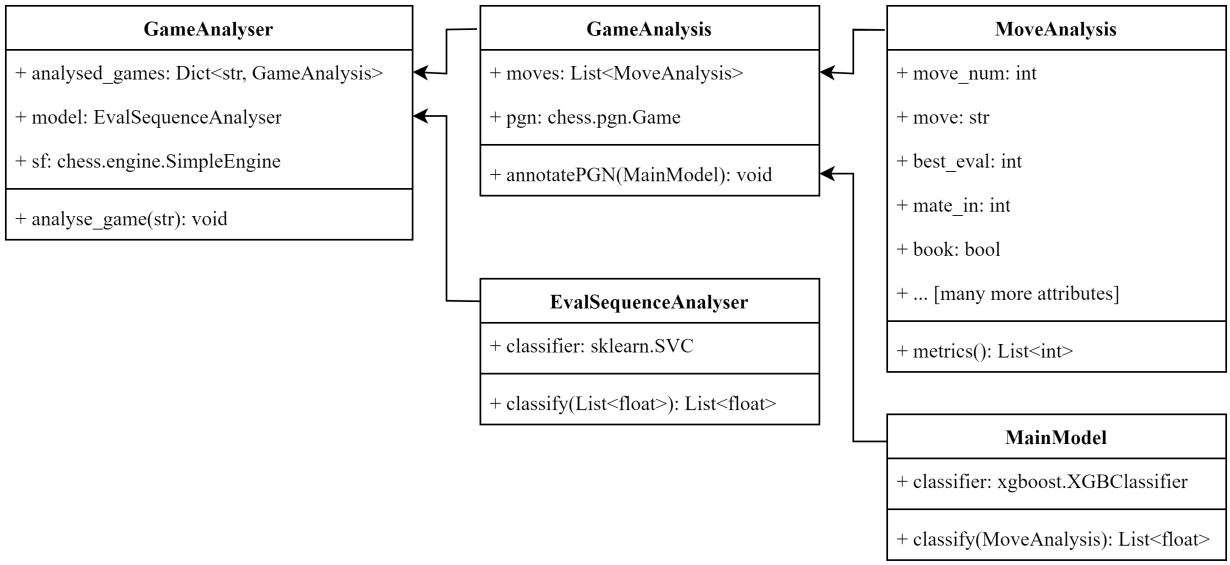


Figure 13: A class diagram of the system. Some `MoveAnalysis` attributes are hidden.

Each analysed game will be stored as a `GameAnalysis` object, containing both the original parsed PGN file as a python-chess `Game` object and a list of `MoveAnalysis` objects containing all the information extracted from the moves. The `MoveAnalysis` object has many attributes (too many to display in a diagram), including the evaluation of the position, multi-depth analysis data and results, data about the best move, and also data about the actual position such as the difference in the number of pieces (in centipawns) between both sides. The method `annotatePGN` will annotate the stored python-chess `Game` which can then be later written to an actual PGN file.

3 Implementation

3.1 Parsing Games

The reading and parsing of the PGN files is handled by the python-chess library [19]. This is very simple to do, as shown by the code snippet in Listing 2 below.

```
# Python 3.11.9
import chess.pgn
with open(f"{gamefile}.pgn") as f:
    game = chess.pgn.read_game(f)
```

Listing 2: Reading a PGN file with python-chess.

This generates a `Game` object which is the root of a tree structure containing all of the header information and all of the moves played. Each move is stored as a `ChildNode` object, with attributes allowing a `Board` (the position) to be extracted alongside the text representation of the move. The system will only be analysing the moves played in the game ignoring any variations, so we can treat this as a linked list. The `Board` object can be passed to an engine to analyse the position to determine the evaluation, as well as providing useful functions to count the number of pieces on the board. Finally, each `ChildNode` has a function which traverses back to the root `Game` node.

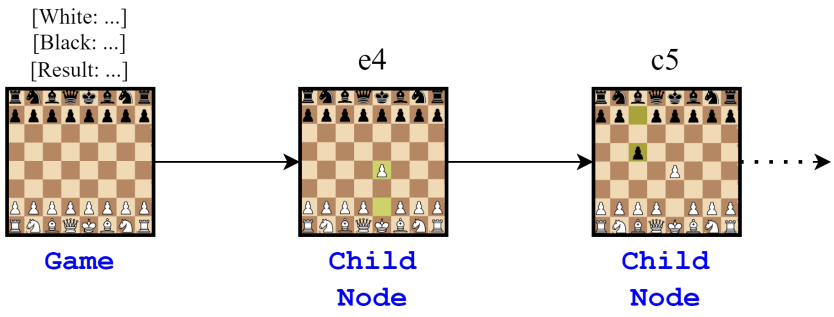


Figure 14: How parsed games are stored in memory. Board images are from lichess.org [28].

Each `ChildNode` also has a NAG (numerical annotation glyph) and a comment attribute, allowing moves such as '♘f3' to be annotated as '♘f3!? {Engine move.}' when written to a

PGN file. The report will be included in the comment of the final move of the game. The code for an annotation like this is shown in Listing 3 below.

```
# position is a ChildNode object (associated with a move)
position.comment = "Engine move."
position.nags.add(chess.pgn.NAG_SPECULATIVE_MOVE)
```

Listing 3: Annotating a move in a game.

3.2 Universal Chess Interface

Modern engines communicate using the Universal Chess Interface (UCI) protocol [30]. The python-chess library provides functions to interface with engines using this protocol, which includes Stockfish. UCI defines various commands which can be sent to engines and various responses they can give, all of which are text sent through standard input/output streams.

This is useful, as the public documentation allows us to browse through all of an engine's features and what outputs we have access to. Importantly in the UCI protocol, moves are not given in the standard algebraic notation as seen in PGN files. Instead, they are provided in long algebraic notation. Each move is written as the square the piece moves from concatenated with the square the piece lands on. So, instead of 'Nf3', we would write 'g1f3' if the knight came from g1. Listing 4 below shows an example of how Stockfish 16 can be instructed to analyse a position using UCI commands.

```
isready
    readyok
position startpos moves e2e4 c7c5
go infinite
    info string NNUE evaluation using nn-5af11540bbfe.nnue enabled
    info depth 1 seldepth 1 multipv 1 score cp 19 nodes 33 pv g1f3
    info depth 2 seldepth 2 multipv 1 score cp 19 nodes 63 pv g1f3
    info depth 3 seldepth 2 multipv 1 score cp 24 nodes 128 pv h2h3
    info depth 4 seldepth 2 multipv 1 score cp 33 nodes 170 pv a2a4
    info depth 5 seldepth 3 multipv 1 score cp 44 nodes 216 pv h2h3
    info depth 6 seldepth 4 multipv 1 score cp 59 nodes 411 pv g1f3
    info depth 7 seldepth 4 multipv 1 score cp 52 nodes 1079 pv g1f3 e7e6 b1c3
stop
    bestmove g1f3 ponder e7e6
```

Listing 4: Communicating with Stockfish 16 using UCI commands, inputs are highlighted.

This is primarily the output we are looking for from the engine, every time it reaches a new depth it sends an output automatically until a command to halt is given. Each output contains the evaluation of the position (`score cp 52`, used to construct L for multi-depth analysis) and we also have access to the 'principal variation' (`pv g1f3 e7e6 b1c3`), which consists of the engine's best suggested moves for each player. We can see how the `seldepth` is different to the actual depth too, it refers to the number of ply in the deepest PV line considered during evaluation. It is also interesting to note how the engine's top choice changed many times from g1f3 → h2h3 → a2a4 → h2h3 → g1f3 even over the course of just a few depths, we can monitor this metric (the number of changes) as it indicates that the best move is not obvious and there are a few options to consider.

Fortunately, python-chess has a handler for this so UCI commands do not need to be constructed manually and the outputs don't need to be raw text. We can achieve the above analysis using Stockfish using the Python code shown below in Listing 5.

```
sf = chess.engine.SimpleEngine.popen_uci("engine\stockfish.exe")

board = chess.Board()
board.push(chess.Move(chess.E2, chess.E4))
board.push(chess.Move(chess.C7, chess.C5))

info = sf.analyse(board, chess.engine.Limit(depth=7))
evaluation = info["score"] # cp 52
bestmove = info["pv"][0] # g1f3

sf.quit()
```

Listing 5: Performing the same analysis to depth 7 with Python code.

Additionally, because engines share this same protocol, to switch from using one engine such as Stockfish to another like Leela Chess Zero [27] we simply need to specify a different engine executable. With this we can experiment with different engines and potentially combinations of engines to see which ones are most effective, an example is shown in Listing 6.

```
sf = chess.engine.SimpleEngine.popen_uci("engine\stockfish.exe")
lc0 = chess.engine.SimpleEngine.popen_uci("engine\lc0.exe")

# Evaluation from Stockfish
stockfish_result = sf.analyse(position)

# Evaluation from Leela Chess Zero (Lc0)
leela_result = lc0.analyse(position)

sf.quit()
lc0.quit()
```

Listing 6: An example using multiple engines to analyse a position.

3.3 Engine Choice

With the ability to switch between engines easily, the choice of engine can be decided by both research and also experimentation seeing which engine is more efficient and works best for the project.

We want to use a strong engine to evaluate positions as accurately as possible, a weak engine will see a brilliant move as a mistake if it can't calculate well enough. There are a wide range of engines available, but Stockfish is the strongest engine available winning the last eight seasons of the Top Engine Chess Championship, with Leela Chess Zero [27] being the runner-up for seven of those seasons [11]. There is an interesting new engine being developed, Torch [7], however it is not available to download and use yet. This was something which was monitored in case it did release during the project's development as it may have ended up being faster or stronger than Leela Chess Zero or potentially even Stockfish. Therefore, the choice was between Stockfish and Leela Chess Zero.

Stockfish and Leela Chess Zero were both used to analyse three games to selective depth 30. The first game was short (20 moves = 40 ply) and the second game was average length (40 moves = 80 ply) and the final game was slightly longer (50 moves = 100 ply).

	40 ply	80 ply	100 ply
Sf	112s	141s	192s
Lc0	265s	478s	638s

Table 3: Engine analysis speeds in seconds.

The average length of a game of chess is 40 moves (80 ply) [12], so these measurements should serve as good upper and lower bounds for timing. We can see that Stockfish takes roughly two or three minutes to analyse a game, this is a reasonable speed and acts as a benchmark figure as Stockfish is currently the best choice based on its strength. Changing nothing except switching the executable from 'stockfish.exe' to 'lc0.exe', the timings increase dramatically. Across the three measurements, Leela Chess Zero's timings are 2.37×, 3.39×, and 3.32× slower than Stockfish's respectively. If Leela Chess Zero was faster than Stockfish it could be considered, but with Stockfish being both faster and stronger it is the clear choice for this project.

3.4 Secondary Engine

While Stockfish is a better option than Leela Chess Zero for a solitary engine, this does not mean that we should disregard the idea of using it entirely. We can analyse positions with both engines and compute a 'dispute' metric between the two. The idea here is that intuitively, engines agree on obvious moves and will likely disagree in complex positions where the best move is unclear. They may even suggest the same move, but give different evaluations.

Below, Figure 15 shows games analysed by both Stockfish and Leela Chess Zero. Both games involve some engine assistance, to different extents, so we would hope to see some disagreement between the engines to reflect the complexities in the position.

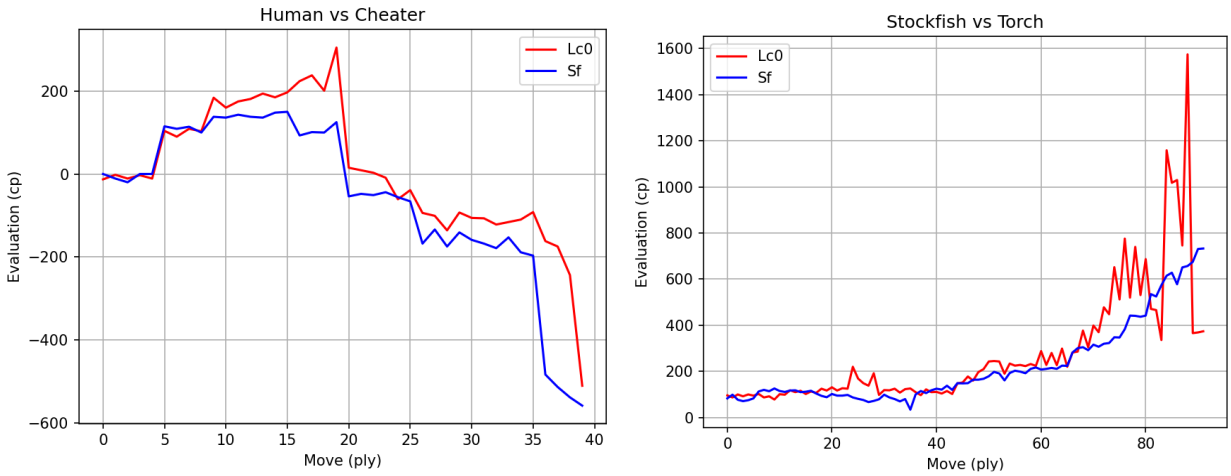


Figure 15: Stockfish's evaluation compared to Leela Chess Zero's.

In the first game, both engines seem to give roughly the same evaluations. We see in some parts, one of the two might exaggerate the advantage slightly compared to the other, but overall, one engine evaluating the position as +150 and the other evaluating it as +200 is not meaningful.

In the second game, we see the same thing up until the evaluation breaks out above +400. Leela Chess Zero wildly fluctuates between +500 to +800 for a few moves, then spikes massively at the end. Meanwhile, Stockfish has a gradual increase which is likely more accurate, because two engines as strong as Torch and Stockfish are unlikely to make back-to-back blunders as Leela Chess Zero is implying. If this pattern was seen in the middle of the game when the evaluation was within ±100, perhaps it would be interesting to use this dispute metric as a feature. However, this all occurs when the evaluation is indicating a decisive advantage to one player. At this stage evaluation differences are less significant, analysing the difference between +500 and +1000 is analogous to analysing the difference between a 'huge' advantage and a 'massive' advantage, we don't benefit from it.

Therefore, given the lack of any promising findings and the fact that analysing positions with both engines could increase the runtime by more than 300% based on the tests in Section 3.2, a secondary engine has not been used to calculate a dispute metric in this system.

3.5 Engine Limits

In this program we are using an engine (Stockfish) to analyse games and we are using the data collected to detect cheating. However, it is important to consider the limitations that engines have.

If we take the set of possible moves M, then an engine will internally assign an evaluation $e(m)$ for all $m \in M$. Without loss of generality, we assume that a greater evaluation is better for the player. It will assign each move an index such that $e(m_1) \geq e(m_2) \geq \ldots \geq e(m_n)$ for a position with n legal moves, where m_1 will be chosen as the best move. As we know engines aren't perfect, they won't always find the best move. However, this means that a better move might be given a lower evaluation.

This can be problematic, in cases such as Ken Regan's system [36] we see how it can cause issues because cheaters may have some of their moves classed as inaccuracies. Metrics such as average error or coincidence value [2] would be dampened by this too, so it must be taken into consideration.

To mitigate this, for each position we evaluate it as-is and determine the best move (with its evaluation). Then, instead of simply taking the evaluation of the move played from that analysis, we take the position after playing the move in question and evaluate it. This has a key difference, in the first instance the engine is looking at all possible moves and determining which one looks most favourable. In the second instance, the engine has been given the first move to play and is told to examine all of the possible options, which leads to a slightly deeper analysis. This essentially forces the engine to analyse the played move as deeply as (or perhaps slightly more than) the best move even if it didn't choose it.

If we say that the played move was m_p, the engine's top choice was m_1, and the new evaluation is $e'(m_p)$, then we observe that not only is it possible that $e'(m_p) \neq e(m_p)$, but also it is possible that $e'(m_p) > e(m_1)$. This negative evaluation drop is interesting to observe, it implies that the engine only found the benefit to playing the move once it was prompted and given the idea to analyse it. We can therefore use this to detect some instances of 'better than perfect' play, and reduce the impact of this limitation of the engine.

3.6 Multi Depth Analysis

3.6.1 Computing Evaluation Levels

The idea behind multi depth analysis was developed to replace the failed dispute metric. Instead of using a different engine, we use snapshots of Stockfish's evaluation as it is analysing a position. This builds up a list of evaluations for each move, ranging from a value representing the engine's initial reaction to the position to a figure backed by deep consideration.

We could use ply for this component, generating an evaluation levels vector with each entry $L[p]$ obtained by analysing to ply p, based on the intuition that analysing more moves ahead gives a more accurate result. However, it is possible to calculate ahead while simply verifying that a move works, which may lead to finding a flaw in the idea and result in a low evaluation even for a high ply count.

Equally, we could have each entry $L[r]$ as the evaluation obtained by analysing to raw depth r. But the raw depth is more of an arbitrary counter representing how much the engine has analysed already. It is possible for the engine to explore a different line at a low ply while on a high depth, but this goes against what we are aiming for. We want the first values in L to represent evaluations that humans agree with, and later ones to be evaluations that humans potentially disagree with. If a high depth is searching a lower ply, then it is still feasible for a human to be calculating the same line.

Selective depth is a value which mitigates these two issues. Firstly, because it is partially dependent on ply, it is likely that as we go from $L[1]$ to $L[20]$ the ply the engine is analysing at will be increasing. But also, because the selective depth is based on the length of the principal variation, it will only increase when the engine has more confidence in the longer lines it is calculating, which stabilises the evaluation.

Listing 7 below shows a code snippet for computing L using the python-chess interface. The key point here is that we not simply giving a command to "analyse to depth d", instead we are analysing until the seldepth of a result is greater than our threshold, `sf_depth`. This is done through the use of the method `sf.analysis(...)` which returns a stream of results. A

time limit of 10 seconds has been placed to ensure the engine doesn't spend an unreasonable amount of time on any given position.

```python
def iterative_analysis(self, played):
        raw_levels = np.zeros(sf_depth) # L
        pos_eval = 0 # The final evaluation
        pos_move = None # The best move
        changes = -1 # Number of times pos_move is updated

        results = self.sf.analysis(played, limit=chess.engine.Limit(time=10))
        prev_depth = 0

        # Return default values if the game is over
        if played.is_game_over():
            return np.ones(sf_depth), 0, played, 0

        # Each 'r' is equivalent to an 'info' line from the engine as shown in Listing 4
        for r in results:
            depth = r.get('seldepth')
            # Some results are empty as the engine is not done analysing
            if depth is not None:
                if depth > sf_depth:
                    depth = sf_depth
                # Convert evaluation to an int, M1 becomes 9999, ..., Mn becomes 10000-n
                pos_eval = r['score'].white().score(mate_score=10000)
                if depth > prev_depth:
                    if r['pv'][0] != pos_move:
                        changes += 1
                    pos_move = r['pv'][0]
                    # It's possible that the seldepth skips a value, so fill any missing
                    # ones with this evaluation
                    raw_levels[prev_depth:depth] = pos_eval
                    prev_depth = depth
                else:
                    raw_levels[depth-1] = pos_eval

        # If evaluation is interrupted due to time, extrapolate
        if depth != sf_depth:
            raw_levels[depth:] = pos_eval

        return raw_levels, pos_eval, pos_move, changes
```

Listing 7: Python code snippet for computing L, a method of the `GameAnalyser` class.

This function takes one argument, `played`, which is the resulting position of the board after playing the move we are analysing. Using this and attributes from the `GameAnalyser` object such as the engine and the target seldepth, it will compute the evaluation levels vector L. Alongside this, it also tracks the number of times the first move in the principal variation

changes as this is also an interesting metric. The final evaluation of the position is returned too.

3.6.2 Further Processing

After testing we can see that this function performs as we expect, allows us to obtain an evaluation vector L for each move. We can now move onto processing them further. However, when examining these vectors as displayed in Figure 16, we can see that they have a tendency to be very jagged.

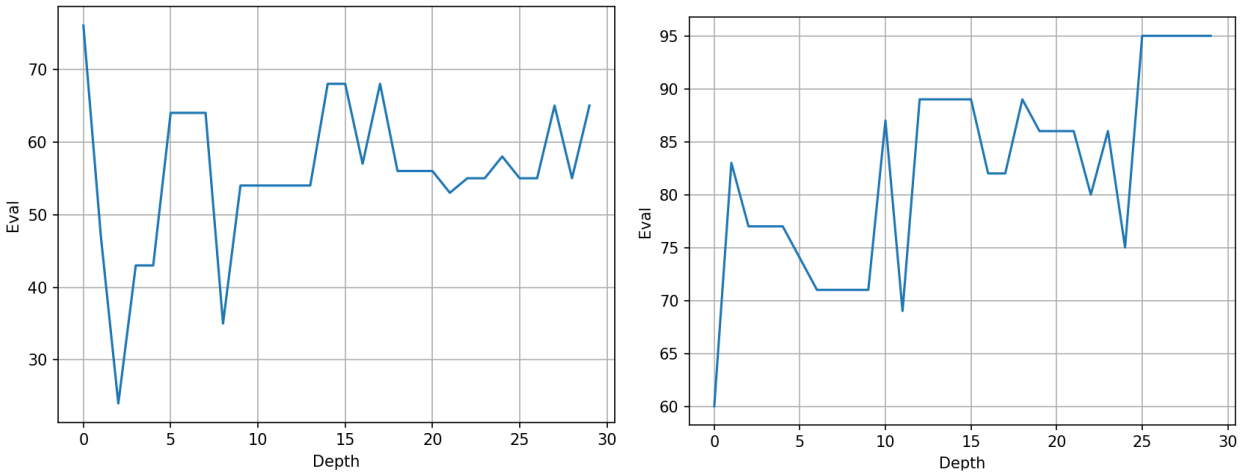


Figure 16: Visualisations of L for two moves.

When trying to analyse the trend in the values, this noise is unhelpful. Perhaps if the noise was caused by some aspect of the position, it would be interesting to use this to our advantage. Unfortunately, it is likely a result of the horizon effect. This occurs when a search tree is explored up until a certain point, and the position is evaluated returning a great advantage for the player. But when explored just one node further, the advantage flips and the player is losing. Therefore, we could encounter situations where a move is considered brilliant on depth n, but poor on depths $n+1$ and $n-1$. We don't want these anomalies to confuse the model used to classify the trends, so it is useful to filter them out.

Additionally, we can see that the scales of these two graphs are different and they are centred around different evaluations. Therefore, it is useful to separate the amplitude of the graph

from the shape and also subtract the evaluation of the best move from every value. This gives more context to the graph, after subtracting the best move's evaluation we can see that if the values are still above zero, the move was likely very good. Exactly zero, the move was the engine's top choice. Below zero, the move was a mistake. For black, the inverse is true, below zero would be good.

Taking the example before from Section 2.7 (Figure 12), after filtering the graph and normalising the values by subtracting the evaluation of the best move and dividing by the greatest magnitude of any value, we gain a much smoother curve as shown below.

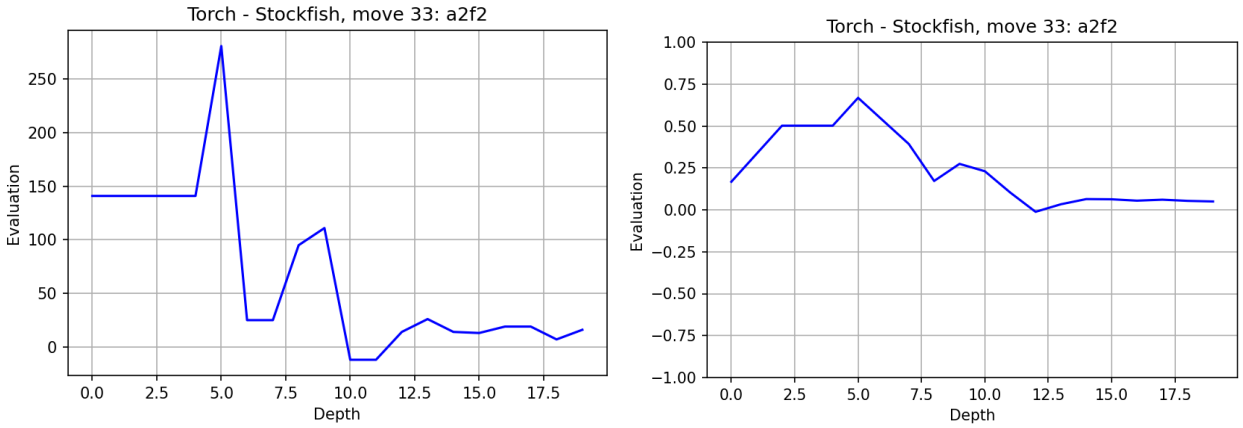


Figure 17: Filtering a normalising L.

Preserving the difference between the values in L and the evaluation of the best move is important, because two graphs may have identical trends showing increasing values, but one may be going from terrible to good while the other goes from good to amazing and it could be beneficial to classify them differently. A code snippet demonstrating how this is done is shown below in Listing 8, where `self` refers to an instance of the `GameAnalyser` class.

```python
from scipy.signal import filtfilt

# Evaluate the current position to find the best move and current evaluation
best_levels, best_eval, best_move, _ = self.iterative_analysis(board)
# Evaluate the move that was actually played
raw_levels, pos_eval, future_best_move, changes = self.iterative_analysis(played)

# Offset the evaluation such that zero means 'best_eval'
levels = raw_levels - best_eval
# Flip evaluation for black (if ply count is odd)
if not ply%2:
    levels *= -1

# Scipy filtfilt function to smooth the values
filtered = filtfilt([1/3 for _ in range(3)], 3, levels)
# Normalise by dividing by the value with the greatest magnitude
max_mag = max(np.abs(filtered))
# Don't divide if max_mag is zero (which is possible, and sometimes common!)
if max_mag:
    filtered = [l/max_mag for l in filtered]
```

Listing 8: Python code snippet for normalising and smoothing L.

The smoothing of the curve is done using a function from the Python library scipy [39] called `filtfilt`. This applies an IIR (Infinite Impulse Response) filter both forwards and backwards across L, essentially smoothing out any sharp peaks or troughs slightly and removing noise from the sequence of values.

With this code in place, L is now split into a 'shape' describing the trend in the values relative to the best evaluation, and an 'amplitude' which represents how far away from the best evaluation the values drifted (this is `max_mag` in the code).

3.6.3 Support Vector Classification

It is tempting to use a naïve classification method, by using the gradient between the first and last points. For example:

$$grad(L) = \frac{L[20] - L[1]}{19}$$

We might then use $m < 0$ to mean 'descending' and $m > 0$ to be 'ascending', and $m = 0$ otherwise. However, this does not take into account the variety of the trends we encounter, the point of analysing the position at so many depths is to capture the shape in the middle.

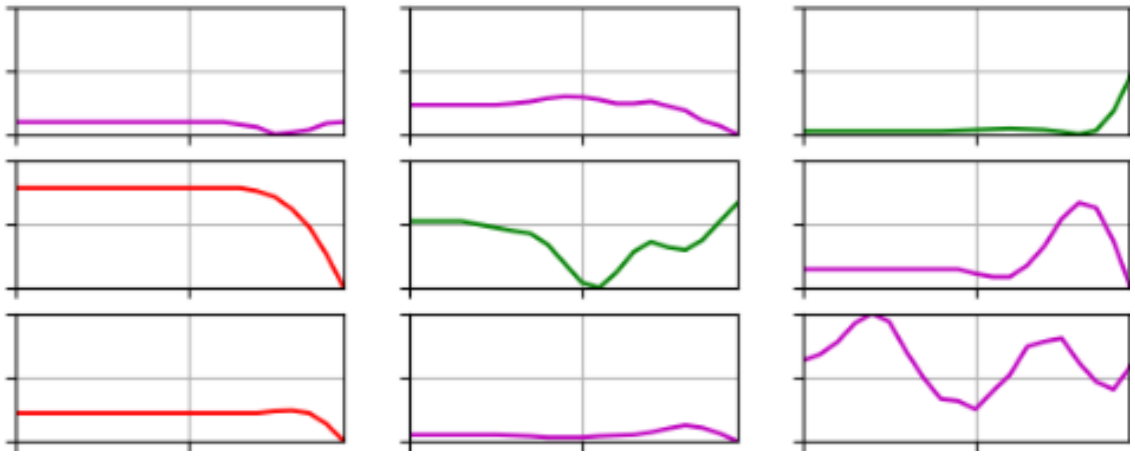


Figure 18: Different types of trends encountered. Red is 'descending', green is 'ascending', magenta is 'noisy' and 'flat'.

Some trends go up and down wildly in the middle, and some are mostly flat with an extremely subtle flick at the end going up or down. These would all be mis-classified using the gradient method, and wouldn't capture details such as the dip in the middle of the central plot in Figure 18. It was classified as ascending because despite being good at the beginning, it reached a very poor evaluation at a reasonable low depth (which a human may have been able to realise), but then continually improved at high depths.

Instead, to classify these trends, support vector classification is being used. The idea here is to map the input to a high dimensional feature space, with a non-linear function, so that a linear boundary can then be drawn [14]. The boundary is drawn to maximise the margin between the boundary and the closest data points in each class. This is because if we get a new data point which is an outlier from the rest of the training points in a class (but should still be in it), we want the decision boundary to give it as much room as possible before it crosses over into the wrong class.

This method of classification is suitable for the task, mainly due to a combination of there being a relatively small amount of training data available alongside support vector machines' ability to generalise well from mapping to a high dimensional feature space. We only have a limited amount of training data because in order to use supervised learning, all samples must be labelled by hand which takes time. Equally, it would take a large amount of time

to generate enough data for unsupervised learning which did not show promising results (as discussed in Section 3.6.5).

3.6.4 Training and Testing Data

The data samples were selected from four games, two of which were between engines and two of which were between humans to ensure variety. The results of these games were varied too, not all were won by white. Additionally, one of the engine games included a weaker engine and one of the human games was a strong player against a weak player. There was also a human game between two weak players to give examples of poor quality play. A breakdown of the 331 labelled data points is shown in Table 4. Each data point is an evaluation levels vector L of size 20, assigned one of the three classes 'ascending', 'descending' or 'other'.

Game	**Asc**	**Dsc**	**Oth**	**Total**
Engine vs Engine	32	26	38	96
Engine vs Weak Engine	27	45	36	108
Human vs Weak Human	14	11	14	39
Weak Human vs Weak Human	24	30	34	88
Total	97	112	122	331

Table 4: Dataset for the classification task sourced from the four games.

This is the overall data set, providing a near equal number of samples of all three classes. There are more engine samples than human samples, this is because engines will never resign and won't make severe mistakes, which drags out the games. However, this isn't an issue because we are more interested in getting samples of lots of different trends, we aren't training this model to differentiate human and engine moves so a 50:50 ratio between them is not necessary. It was split into both training and testing data with a test to train ratio of 1:3, ensuring that the model's accuracy could be measured with reasonable confidence while also not wasting the limited dataset on testing as opposed to training.

3.6.5 Performance and Testing

Splitting the testing and training data was done using the scikit-learn library's `test_train_split` method [34] as shown in Listing 9. The support vector classification method itself is implemented using scikit-learn's `svm.SVC` class, this allows us to specify a kernel and also gives the probabilities of the predictions it outputs. Overall, this gives our `EvalSequenceAnalyser` class a relatively simple definition after the parameters are tuned.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# Read a CSV file, splitting the first column from the rest
def loadgame(name):
    db = np.genfromtxt(f"{name}.csv", delimiter=",")
    return db[:,1:], db[:,0]

class EvalSequenceAnalyser():
    def __init__(self):
        # X contains the data, Y contains the labels
        X = []
        Y = []
        # Load the four games containing the 331 samples
        for game in ("sf_torch", "sf_weiss", "me_ege", "me_random"):
            X_temp, Y_temp = loadgame("prototype\\training_data\\" + game)
            X.extend(X_temp)
            Y.extend(Y_temp)

        # The default split ratio is 1:3 (test:train)
        XTr, XTe, YTr, YTe = train_test_split(X, Y)

        # Fit the model
        self.model = SVC(kernel = 'linear',
                         C = 1.0,
                         probability=True).fit(XTr, YTr)

    # Takes an evaluations levels vector 'eval_sequence' and returns
    # the classification (along with the probabilities)
    def classify(self, eval_sequence):
        # predict() requires a list of inputs and returns a list too, so we index with [0]
        prediction = self.model.predict([eval_sequence])[0]
        prediction_proba = self.model.predict_proba([eval_sequence])[0]
        return prediction, prediction_proba
```

Listing 9: Python code snippet for implementing the support vector classification class.

In this implementation, a linear kernel has been used with a regularisation parameter C=1.

These parameters were derived from a grid search, with 5-fold cross validation for each set of parameters.

Kernel	C	Extra Params	Validation (%)	Test (%)
Linear	0.5, 1, 2, 5	N/A	88.74	91.57
Polynomial	0.5, 1, 2, 5	d: 2, 3, 4	76.23	77.11
Radial Basis Function	0.5, 1, 2, 5	γ: scale, auto	87.12	90.36

Table 5: Grid search parameters, with best validation and test accuracies.

For the radial basis function kernel, the parameters for γ are either 'scale' or 'auto'. These are the options provided in the sci-kit learn library's `SVC` class, they correspond to $1/(n \cdot \sigma^2)$ and $1/n$ respectively where n is the number of features and σ^2 is the variance of the data.

Visualising the test data and looking at the classifications given, we can see that the model performs very well. The error is almost entirely from edge cases, where a trend could be feasibly put into either class, but the model chose the 'incorrect' class. For example, in Figure 19 below there are two shaded plots (mis-classified trends). One is coloured green for 'ascending' because the trend goes up, but the original label was 'other' because the increase is small. The other is red for 'descending' because it does down slightly, where it was originally labelled 'other' due to it being mostly flat.

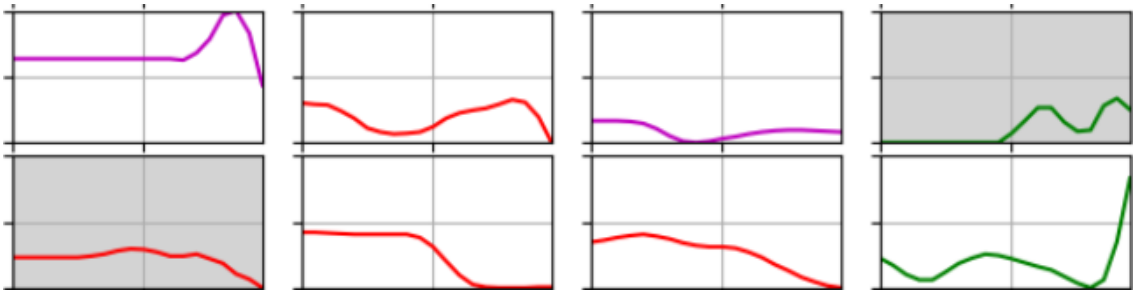


Figure 19: The two 'incorrect' classifications are shaded.

These three classes were chosen based on the trends in the data samples collected, however, it is possible to use unsupervised learning and allow the model to create its own classes.

Experimenting with this idea, K-means clustering was used to determine classes for the data. This involves finding K centroids in the dataset and forming clusters around them, such that the variance of each cluster (class) is minimised [40]. Unfortunately, this did not produce good results. Similar trends were present across different classes which indicates that these classes are not being formed based on the trends in the data, which is the primary function of this model.

Additionally, the idea of using four classes was explored too. Instead of having 'other', it was split into two classes 'noisy' and 'flat'. However, this performed worse, with the model struggling to differentiate the two. It is also not necessary to distinguish between these two classes, as both of them indicate no interesting trend in the data. If we wanted to distinguish these two classes in other parts of the code, we could simply look at the amplitude and see if its small (indicating flat as opposed to noisy).

3.7 Caching Results

In addition to generating a report, we want to save all the details of the analysis too. Given that analysing all of the games in a tournament for even one player could take a long time, we don't want to waste time re-analysing them with Stockfish every time we try using different or new feaures for the main machine learning model.

By using the Pickle library [38], we are able to efficiently serialise and store our custom `MoveAnalysis` and `GameAnalysis` objects preserving all of the data. This is stored in a `.pkl` file alongisde the PGN files of the game and its annotated version. Below in Listing 10 we can see the process for reading games when the program is invoked. The list of games will be passed as command line arguments (found in `sys.argv`, where the first index is the name of the file). If one of these arguments is `-force`, then we ignore any existing `.pkl` files. This allows us to re-run analysis if, for example, Stockfish releases a new update or if analysing with a different engine like Torch is more effective in the future.

```python
# Argv contains the list of game files to analyse
from sys import argv
import pickle
from game_analyser import GameAnalyser

analyser = GameAnalyser()
# If --force is given, we ignore the pickle file
force = "--force" in argv

for gamefile in argv[1:]:
    if gamefile == "--force":
        continue

    # Assume there is a .pkl file, update flag if there is not
    notfound = False
    try:
        with open(gamefile + ".pkl", "rb") as f:
            # This is a GameAnalysis object
            game = pickle.load(f)
    except FileNotFoundError:
        notfound = True

    # Do a full analysis if there is no .pkl file, or it is forced
    if notfound or force:
        analyser.analyse(gamefile, show=show)
        game = analyser.analysed_games[gamefile]
        # Update or create the .pkl file when done
        with open(gamefile + ".pkl", "wb") as f:
            pickle.dump(game, f, pickle.HIGHEST_PROTOCOL)
```

Listing 10: Python code snippet for reading and analysing games.

3.8 Opening Moves

Analysing the middlegame is the critical part of this project, it is the portion of the game where the majority of intermittent cheating will occur (and where any kind of cheating is most noticable). However, we still need to recognise the opening moves.

This is done by using a large collection of Polyglot opening book files which is an open-source format, the format has been summarised below [31]. A book contains a list of 8-byte Zobrist keys, each with key representing a position. This allows for millions of positions to be stored using a relatively small amount of storage. A Zobrist key is obtained by indexing lookup tables containing 781 random values in total (each value being an 8-byte number) and using the bitwise XOR operator to combine them into a new value representing the position.

For example, the main lookup table is for pieces, which has 768 entries. It can be seen as a flattened 3-dimensional array of shape $12 \times 8 \times 8$, representing 12 piece types, 8 board files, and 8 board ranks. Each of the 12 piece types is given a value from 0 to 11, and each file is remapped from A-H to 0-7. Therefore, for each piece in the position we take the random value R located in index '$(64 \times \text{type}) + (8 \times \text{rank}) + \text{file}$' in the lookup table, and update our key K with this value by letting our new key $K' = K \oplus R$. The table is pre-defined by Polyglot, so the same piece on the same square will always give the same R in any implementation.

We XOR this key with a few more random values representing details such as which colour moves next or if either side can castle. The result is that we get a unique Zobrist key for every possible position which can be efficiently computed. Due to the symmetry of the XOR operation this key can be efficiently updated too, when moving a piece we XOR the key with the two random values corresponding to where the piece was, and where it is moving to.

As such, this opening lookup system adds little to no overhead to the program as a whole which is useful, as the collection of Polyglot books being used contains opening moves from approximately three million games [13]. We use the python-chess library's functions to read these books and lookup board positions as shown in Listing 11 below.

```python
import chess.polyglot

# List of opening books being used
book_list = [
    r'openings\baron30.bin', r'openings\Book.bin', r'openings\codekiddy.bin',
    r'openings\DCbook_large.bin', r'openings\Elo2400.bin', r'openings\final-book.bin',
    r'openings\gm2600.bin', r'openings\komodo.bin', r'openings\KomodoVariety.bin',
    r'openings\Performance.bin', r'openings\varied.bin',
]
books = [chess.polyglot.open_reader(book) for book in book_list]

# Not an opening move only if all books return False
opening_move = False
for book in books:
    # Game is a ChildNode representing the move being analysed
    if book.get(board=game.board()):
        opening_move = True
        break
```

Listing 11: Python code snippet for reading Polyglot opening books and looking up board positions.

3.9 Checkmate Sequences

As discussed previously, we need to handle checkmate positions carefully. This is done by calculating a 'risk' metric for each move and using the idea that a human player would usually pick the move which minimises risk, while an engine simply picks the move which leads to the quickest checkmate.

To achieve this, we look at the material imbalance. This uses the standard piece values, so if white had an extra knight that black didn't have, the imbalance would be +3. If black had a queen which white didn't, it would be -9. We sum the values of all pieces in a position to get the imbalance, which we then compare with the range of moves that can be played. To weight these relative to each other, we use the softmax formula.

$$risk = \frac{e^p}{\sum_{i \in I} e^i} \tag{4}$$

In this equation, I represents the list of imbalances at the resulting positions of all analysed lines, and p is the imbalance at the end of the line played. This gives us a normalised risk factor for each move, assigning high risk to lines which involve material sacrifices for quick wins.

It would not be safe to brand these risky moves as engine moves just based on this, after all, strong players may see past the temptation of taking free pieces and calculate the faster checkmates anyway. We can still flag them as unusual though, if the move played was not the move with the lowest risk. Below in Listing 12 the implementation of this risk comparison function is shown. It is also important to note that we don't analyse all moves in the position, only the top 10. This is because beyond the 10^{th} best move it is likely in most positions that the checkmate sequence would have been lost. We benefit more from spending time analysing the top 10 moves deeper and calculating accurate risks for them, rather than diluting our analysis with the remaining 20 or 30 moves which are unlikely to be useful.

```python
def mate_complexity(sf, best_eval, board, move, played_move):
    piece_values = { # Each piece has a value in centipawns, black pieces are negative
        "K": 0, "k": 0,
        "Q": 9, "q":-9,
        "R": 5, "r":-5,
        "B": 3, "b":-3,
        "N": 3, "n":-3,
        "P": 1, "p":-1,
    }
    current_pieces = board.piece_map()
    current_imbalance = 0 # Find the current imbalance by summing all piece values
    for square in current_pieces:
        current_imbalance += piece_values[current_pieces[square].symbol()]
    # Only check if the winning player has a checkmate sequence
    good = 1 if move%2 else -1
    if best_eval*good*-1 < 9000:
        return None, current_imbalance, None
    mate_in = 10000 - best_eval
    # Flip the imbalance so that negative is good (less risk)
    current_imbalance *= good
    # Analyse the top ten lines
    all_moves = sf.analyse(board, chess.engine.Limit(time=1), multipv=10)
    imbalances = []
    for mv in all_moves:
        future_board = board.copy()
        count = 0
        if len(mv['pv']) >= 2:
            for pv_move in mv['pv']:
                # Calculate to the end of the line
                future_board.push(pv_move)
        else:
            imbalances.append(current_imbalance)
        # Look at the imbalance at the end
        pieces = future_board.piece_map()
        for square in pieces:
            count += piece_values[pieces[square].symbol()]*good
        imbalances.append(count)

    # Use softmax to weight moves by risk
    exp_total = sum([exp(i) for i in imbalances])
    risks = [exp(i)/exp_total for i in imbalances]

    lowest_risk = (1, 0, 0)
    played_risk = None
    for m,mv in enumerate(all_moves):
        if risks[m] < lowest_risk[0] and score(mv) > 9000:
            lowest_risk = (risks[m], score(mv), mv['pv'][0].uci())
        if mv['pv'][0].uci() == played_move.uci():
            played_risk = (risks[m], score(mv))
    # Return the difference between played risk and lowest risk if applicable
    if played_risk is None:
        return mate_in, current_imbalance, None
    if played_risk[0] > lowest_risk[0] and played_risk[1] > lowest_risk[1]:
        return mate_in, current_imbalance, played_risk[0]-lowest_risk[0]
    else:
        return mate_in, current_imbalance, None
```

Listing 12: Python code snippet for calculating move risk.

3.10 Linked Moves

We are analysing each game on a move-by-move level, however it is also useful to observe tactics. These are sequences of moves where the opponent has only one good response available to them, then the player has only one good move, and the chain goes on. This results in a situation where if a player has played the starting move in the sequence, it is highly likely that (if it works) they have calculated all the way to the end because each move is essentially forced.

This is useful to highlight during the game. While playing a tactic involving several moves is not proof of engine play, if the original move is flagged as an engine move by the main machine learning component then being highlighted as a very long and specific tactic by this component can increase the confidence the reviewer has in the existing 'engine move' classification.

To highlight these tactics we can take a recursive approach. For each position, we evaluate the top two best moves in the position. If the best move's evaluation is greater than the second best move's evaluation by a certain threshold, then we determine that the best move is the only good move. In this case, we play the best move and repeat the process until the top two moves are both given a similar evaluation (meaning there are multiple good moves so we stop). We can then return the depth we reached during this process, representing the number of moves in the forced line.

```python
def getLength(sf, board):
    # Analyse the top two moves and find the evaluation difference
    result = sf.analyse(board, chess.engine.Limit(time=0.5), multipv=2)
    if len(result) >= 2:
        diff = score(result[0]) - score(result[1])
    else:
        diff = 30
    # Check if the difference between the best and next best moves is above the threshold
    if abs(diff) >= 30:
        board.push(result[0]['pv'][0])
        val = 1 + getLength(sf, board) # If so, increase depth and repeat process
        board.pop()
        return val
    else:
        return 0
```

Listing 13: Python code snippet for calculating the depth of tactical sequences.

In Listing 13 above, the implementation is shown. The threshold for a significant evaluation difference was set to be 30 centipawns (or 0.3), as an engine playing a move with an evaluation drop this large in a game as little as two or three times could be enough to lose if the opposing engine is strong enough. Additionally, if there is only one legal move in the position, we automatically increase the depth and continue by setting the difference to equal the threshold.

3.11 Time as a Feature

Another interesting consideration goes to time controls. Chess games can have a variety of time controls ranging from 'classical chess' where each player has several hours to make their moves and games can last all day, to 'blitz chess' where each player has three minutes to scramble through the entire game. The amount of time a player spends on a move will clearly have an impact on its quality if the position is complex. However, for a player using an engine, there will be no correlation between time spent and quality as they will simply let the engine think for a few seconds and play the top suggested move. This was a feature which was planned to be utilised in the design stage, however, during implementation there were issues with this idea which led to it being removed.

Unfortunately, the PGN specification mentions nothing about comments relating to the time spent on each move, there are headers defined to specify the time control used for the game, and NAG symbols to indicate general time pressure, but nothing we can use on a move-by-move basis [15]. Because of this, when we examine PGN files obtained from Chess.com [6] and Lichess.org [28], we notice that they are both free to represent the time spent on each move in a unique way.

```
1 [Event "Rated Rapid game"]
2 [Site "https://lichess.org/gc7XCcKq"]
3 [Date "2023.09.07"]
4 [TimeControl "600+0"]
5
6 1. d4 { [%clk 0:10:00] } 1... e6 { [%clk 0:10:00] } { A40 Horwitz Defense }
7 2. c4 { [%clk 0:09:59] } 2... c5 { [%clk 0:09:58] }
8 3. d5 { [%clk 0:09:56] } 3... d6 { [%clk 0:09:52] }
9 4. Nc3 { [%clk 0:09:55] } 4... Nf6 { [%clk 0:09:46] }
```

Listing 14: Part of a PGN file downloaded from Lichess.org.

```
[Event "Live Chess"]
[Site "Chess.com"]
[Date "2023.11.07"]
[TimeControl "600"]

1. d4 {[%timestamp 55]} 1... Nf6 {[%timestamp 16]}
2. Bg5 {[%timestamp 11]} 2... e6 {[%timestamp 23]}
3. e4 {[%timestamp 8]} 3... Be7 {[%timestamp 33]}
4. e5 {[%timestamp 8]} 4... Nd5 {[%timestamp 26]}
```

Listing 15: Part of a PGN file downloaded from Chess.com.

Not only are the `TimeControl` headers in different formats in Listings 14 and 15, but the time comments are using different keys (`%clk` and `%timestamp`) and are even in different formats too. Without a standardised way to specify this, incorporating this as a feature in the system would be difficult and would run into issues if a new format is introduced.

Finally, there isn't a strict association between the move played and the time spent thinking about it. For example, if a player is thinking ahead then they might already be certain of what they are going to do in advance. As such they will play their brilliant move in under a second, going against the expected correlation between time and quality. This makes it very difficult to create an algorithm to incorporate time, and also makes it unlikely that a machine learning component would be able to use it either due to questionable relevance of the timings to the moves they are associated with.

3.12 Main Machine Learning Model

3.12.1 XGBoost

The main machine learning model being used for the classification of 'engine moves' and 'human moves' during the middlegame is XGBoost [5]. Unlike the evaluation depth levels data previously which needed to be manually labelled, we can obtain labelled data automatically from analysing both human and engine games, and using moves which were evaluated as being good as data points (ignoring any mistakes, as these are obviously human). This means we can obtain a large number of data samples which XGBoost benefits from, being an ensemble method.

The idea behind XGBoost is summarised below, we fit many decision trees and use their combined output to make a prediction. First we take the training feature vectors X and training labels Y and fit a decision tree t_1 with them. Then, we can predict the output using our t_1 predictor when given X as an input:

$$P_1 = predict(t_1, X) \tag{5}$$

Currently our prediction P is simply just P_1. However there will be some difference between Y and P_1, this is an error we can define as $E = Y - P$. We can fit a second tree, t_2 on the data X with labels E which will give us a prediction P_2. Essentially, t_2 is aiming to amend the predictions that t_1 got wrong, so we can now make an improved prediction using both $P = P_1 + P_2$. We can repeat this process many times, fitting t_3 using X and our new error $E = Y - (P_1 + P_2)$ to further improve our prediction. We can choose to stop at some tree t_n, then define our overall prediction as follows:

$$P = \sum_{i=1}^{n} P_i \tag{6}$$

Notice how if it were deep enough, we could fit a single decision tree to perfectly predict every feature vector in X. To avoid over fitting like this, we have a hyper parameter to specify the maximum depth of each tree. Equally, we could over fit by having too many trees to the point where our error $E = 0$, so there is a second hyper parameter to specify the maximum number of trees.

Determining whether a move is a human move or an engine move involves many features and is a very complex task. Therefore the ideal decision boundary in the n-dimensional feature space separating the human moves from the engine moves is likely to be a very complex shape, which many models like logistic regression would struggle to fit. However, if we allow the decision trees to be deep enough and use enough of them (with validation to avoid over fitting), XGBoost will be able to do a good job of estimating it.

3.12.2 Training Data

To train the model, 16 long engine games were fully analysed and a total of 513 moves were extracted which were evaluated to be as good as (or better than) Stockfish 16's top recommended move. For human moves, 22 world championship games taken between 2013 through to 2023 were analysed, along with four games played by myself, giving a total of 457 high quality moves.

We cannot guarantee that every move taken from an engine game is inhuman, nor can we say that every move taken from a human game is something that wouldn't be considered an engine move. There will be an overlap between the two classes which will affect the accuracy of the model, however without having a team manually labelling hundreds of moves as either 'feasible for a human to find' or 'impossible for a human to find', this is the best dataset we can obtain to train on.

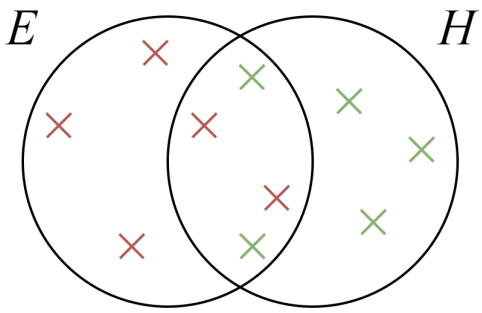


Figure 20: Some moves from engine games are human, and vice versa.

For each move analysed, 17 features were extracted. Many of these have been discussed previously, including the multi depth analysis trend classification, amplitude, and change counter. In addition to this, there are basic features such as whether the turn is white's or black's, the evaluation drop from the best move, the evaluation of the current position, and the evaluation drops of the next best four moves (a similar idea to partial credit [36], measuring the quality of other available options).

There are also some new features, taking the average difference between the current evaluation

and the evaluation of the best move for every entry in the evaluation levels vector. Equally, coefficients of a cubic curve fitted to an extended evaluation levels vector (to depth 30). For the trend classification, we stopped at depth 20 because this is deeper than humans can calculate, but for this model we will benefit from the extra data.

Finally, the last feature observes the difference between the played move and the best move in a literal sense. We compare the long algebraic notation forms of both moves, and count the number of increments or decrements to the ASCII value of each character we would need to make the two strings equal. This means that moving the same piece as the best move or moving to the same square as the best move will both give a low difference score. This is shown in Listing 16 below where the features are extracted from each `MoveAnalysis` object.

```
def metrics(self):
    x0 = self.level_amplitude
    # Fit a cubic curve and use coefficients as features
    coefs, x5, _, _, _ = np.polyfit(np.arange(len(self.levels)), self.levels, 3, full=True)
    x1, x2, x3, x4 = coefs
    x5 = x5[0]
    # Multi depth analysis trend classification
    x6 = self.shape
    # How divergent the evaluation was from the best move
    x7 = np.sum(self.levels)/len(self.levels)
    # How many times the principal variation root was updated
    x8 = self.changes
    good = 1 if self.white else -1
    # Evaluation of the current position
    x9 = self.best_eval*good/300
    # Evaluation of next best moves
    x10 = abs(self.next_best_evals[0] - self.next_best_evals[1:])/300

    # String similarity between best move and played move
    x11 = 0
    for c in range(min(len(self.move), len(self.best_move))):
        x11 += abs(ord(self.move[c])-ord(self.best_move[c]))

    # Current turn and quality of move (raw evaluation drop)
    x12 = 1 if self.white else 0
    x13 = self.eval_drop*good/300

    return np.array([x0, x1, x2, x3, x4, x5,
                     x6, x7, x8, x9, *x10, x11, x12, x13])
```

Listing 16: A method of the `MoveAnalysis` class.

With a few of these features, we take care to ensure that there are roughly equal counts

of each value in either class. For example, we wouldn't want to accidentally sample mostly white moves from engine games and mostly black moves from human games, as our classifier would simply use this as the main predictor, despite it having nothing to do with engine play. Below in Figure 21 we can see histograms showing these observations, the human counts are shown in blue and engine counts are shown in red, and the overlap between the two is purple. We see that there are roughly equal numbers of moves played by white and black for both human and engine classes, and additionally there are roughly equal counts for human and engine moves in terms of evaluation drop too (0.00 being equal to the best move, and negative being even better).

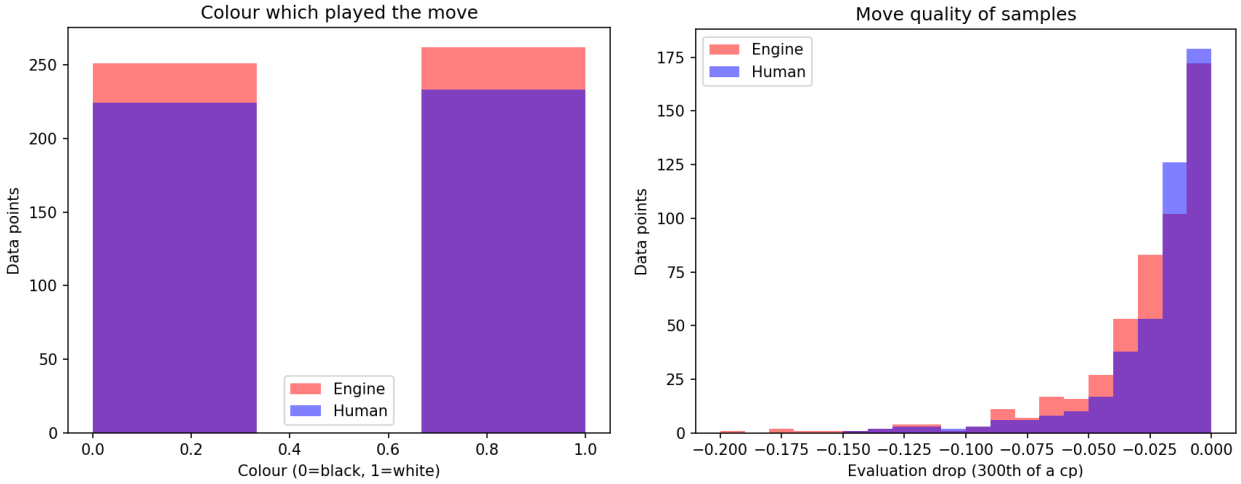


Figure 21: Ensuring there is no bias in the data collected.

This data set of 970 feature vectors was split into 20% test data and 80% training data. While fitting the models, 5-fold cross validation was used too to avoid over fitting.

3.12.3 Other Models

Other models were experimented with for this task, including support vector classification which performed well for the previous task of trend classification. Below Table 6 shows the performance of these models, along with the area under their receiver-operating-characteristic curves. This is a plot of true positive rates against false positive rates as the threshold for a positive classification ranges from 0 to 1, therefore a better classifier has a greater area [1].

Model	Validation (%)	Test (%)	RoC curve area
SVC - Radial Basis Function	54.50	58.76	0.5689
K-nearest neighbours	58.37	56.70	0.6058
SVC - Polynomial	51.93	56.19	0.5278
Naïve Bayes	51.29	53.09	0.6664
SVC - Linear	55.41	52.06	0.5415
Logistic Regression	54.64	51.54	0.5671

Table 6: Validation and test accuracies of other models, ordered by test data accuracy.

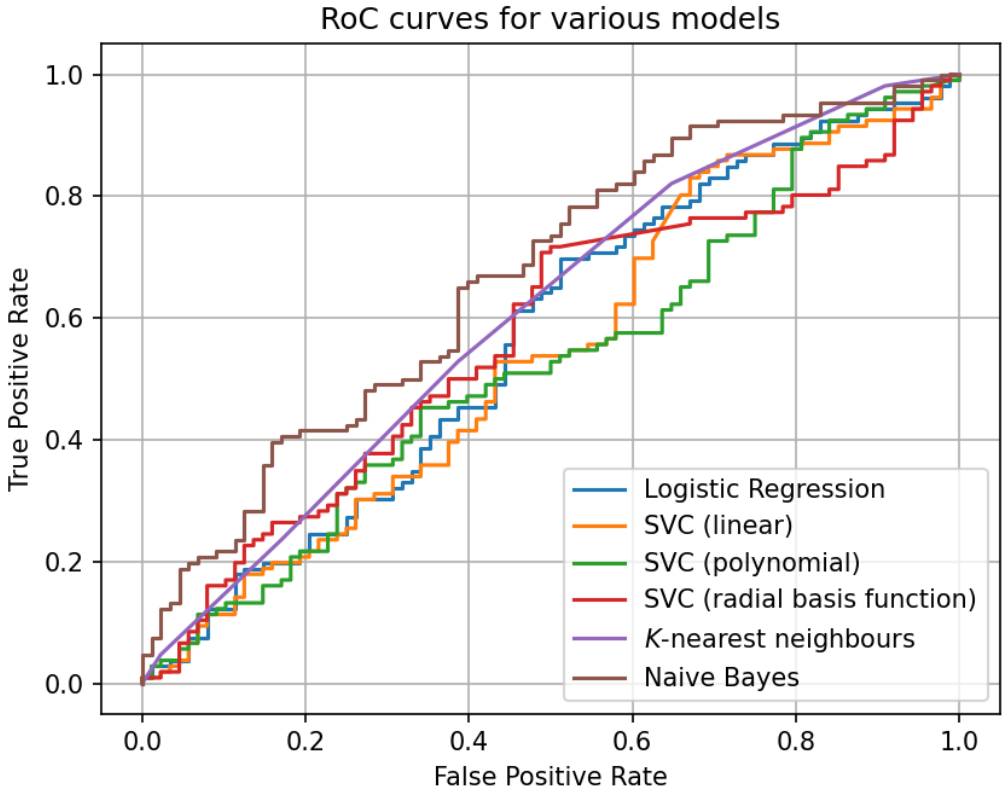


Figure 22: Plot of the RoC curves from Table 6.

As we can see none of these six models performed very well, most barely achieved better than a random classifier (50% accuracy). Support vector classification with the radial basis function kernel performed best with nearly 60% accuracy, however this is not close to the performance of XGBoost with the right hyper parameters.

3.12.4 XGBoost Tuning and Performance

The first grid search was performed to determine the number of estimators and the maximum tree depth of the model. We allow the maximum value for the depth parameter to be 10 and the maximum value for the number of estimators to be 50 to avoid over fitting. These were fixed at their optimal values (7 and 10 respectively) and another grid search was performed to find the optimal learning rate and minimum child weight.

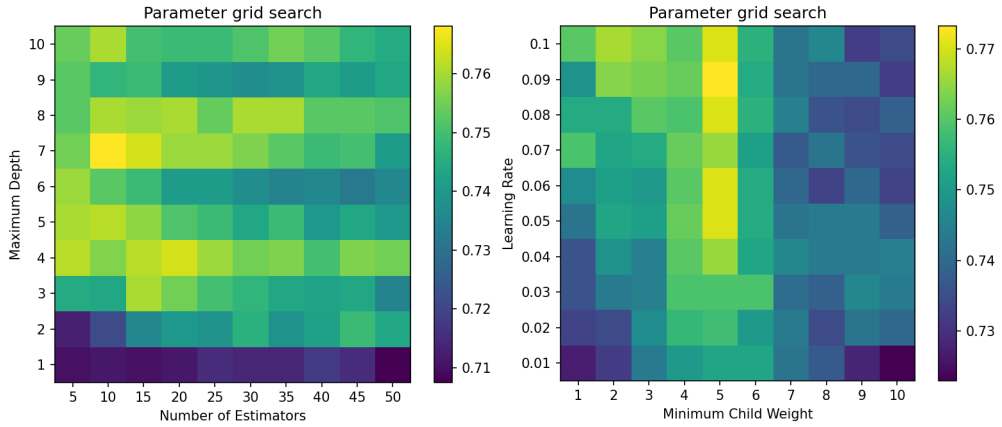


Figure 23: Two grid searches for the four hyper parameters.

The optimal minimum child weight was clearly five and we see that increasing the learning rate slightly helped the model to converge. Finally we can tune the lambda and gamma parameters to improve the model slightly. Both the lambda and gamma parameters make the algorithm more conservative [5], the optimal values were gamma=0.4 and lambda=4.

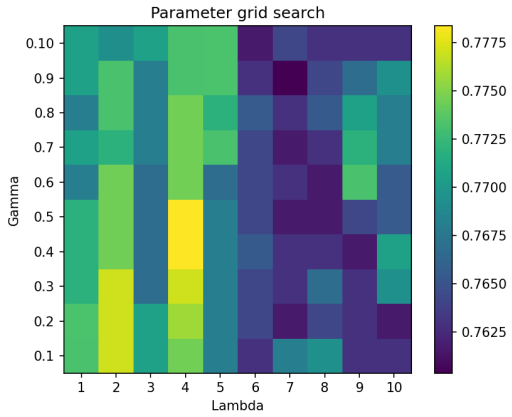


Figure 24: Tuning the final two hyper parameters.

Table 7 below shows the different accuracies and RoC areas after each grid search (keeping the optimal values from the previous search each time).

Tuning Stage	Validation (%)	Test (%)	RoC curve area
Starting Point	76.42	74.74	0.8168
Max Depth + Estimators	76.80	76.80	0.8216
Learning Rate + Child Weight	77.32	75.77	0.8291
Gamma + Lambda	77.84	77.32	0.8315

Table 7: Validation and test accuracy scores after each grid search.

Differentiating engine moves from human moves with 77% accuracy is an excellent result, however we cannot simply take this number and use it as a measure of how accurate our system is. This model was trained on games strictly between two engines with controlled openings and on world championship games, which are both very different to casual online chess and players cheating intermittently. More practical tests on games played by real intermittent cheaters and other human games from less serious tournaments will be showcased in Section 4 to understand the performance of the system.

3.13 Report Generation

The report for each PGN file contains various statistics about the game. An example is shown below in Listing 17, some moves of the game have been removed as they are not annotated.

Moves which had negative evaluation drops, better than Stockfish's top choice, are commented with a note saying this with the NAG `$3` corresponding to '!!' which represents a brilliant move. Engine moves are commented with a percentage confidence (from the XGBoost model prediction) alongside the NAG `$5` corresponding to '!?' which represents an interesting move. They will also have a note if the move is part of a tactic, stating the length of the idea being executed.

```
[Event "Live Chess"]
[Site "Chess.com"]
[Date "2022.12.19"]
[White "Grandmaster"]
[Black "Cheater"]
[Result "0-1"]
[TimeControl "900+10"]
[Annotator "Cheat Detector"]

1. e4 c6 2. d4 d5 3. f3 e6 4. Nc3 Bb4 5. a3 Bxc3+ 6. bxc3 dxe4
7. Nh3 Qa5 $3 { Stockfish likes this better than its original best move. } 8. Bb2 Nf6 9. fxe4 Nxe4
10. Bd3 $5 { This is an engine move, 61.8% confidence. This is the first move in a 3-move idea. }
10... Nf6 $5 { This is an engine move, 73.2% confidence. This is the first move in a 2-move idea. }
20... cxd4 $5 { This is an engine move, 53.2% confidence. This is the first move in a 12-move idea. }
21. Bxd4 {
Total non-book moves: 40

Report for Grandmaster:
Moves: 20
Good moves: 14 (70.0% of all moves)
Brilliant moves: 1 (5.0% of all moves)
Engine moves: 5 (27.8% of analysed, 35.7% of good moves,
                 38.5% of good moves made during critical period)
Average confidence of engine moves: 61.50%

Report for Cheater:
Moves: 20
Good moves: 15 (75.0% of all moves)
Brilliant moves: 4 (20.0% of all moves)
Engine moves: 9 (50.0% of analysed, 60.0% of good moves,
                 69.2% of good moves made during critical period)
Average confidence of engine moves: 62.95% } 0-1
```

Listing 17: Part of an annotated PGN of a game between a grandmaster and a cheater.

The report comment at the bottom states the number of moves played, along with details about the proportion of engine moves. One of these is the proportion 'of analysed', it is stated because not all moves are passed to the XGBoost model to be analysed. This statistic, along with the other two percentages about the proportions of engine moves, are the most important figures to understand in the report and will be covered in more detail in Section 4 with examples. Full reports generated by the system can be found under Appendix B.

4 Testing and Results

4.1 Engine Versus Engine Games

A good benchmark test is to took at games from Chess.com's Computer Chess Championship [10] and determine if the system is capable of recognising completely inhuman play. Below Table 8 shows the results from these analysed games, where the 'positive rate' refers to the proportion of moves the XGBoost model classed as engine moves out of the selection which were analysed by it. None of these games were included in the test or training data sets.

Engines	**Total Moves**	**Engine Moves**	**Positive Rate (%)**
Torch - Stockfish	102	59	73.75
Dragon - Torch	146	50	62.50
Lc0 - Stockfish	143	88	61.54
Stockfish - Torch	158	80	66.67
Lc0 - Torch	129	66	51.16
Stockfish - Torch	134	46	63.89

Table 8: Results from analysing engine games.

The average positive rate is 63.25%, this is a value we can use to represent significant signs of engine usage. In other games, between humans and engines or with intermittent cheaters, we would expect a much lower positive rate.

4.2 Top Level Human Games

We would then like to determine a figure representing a 'not cheating' result, by analysing several high level human games. The set of games used here are from the 2016 world championship match.

Game	Total Moves	Engine Moves	Positive Rate (%)
2016 WCC Game 1	73	0	0.00
2016 WCC Game 2	45	1	2.22
2016 WCC Game 3	138	28	20.29
2016 WCC Game 4	165	24	14.55
2016 WCC Game 5	87	19	21.84
2016 WCC Game 6	40	0	0.00
2016 WCC Game 7	47	4	8.51
2016 WCC Game 8	90	4	4.65
2016 WCC Game 9	113	26	23.01
2016 WCC Game 10	135	45	41.28
2016 WCC Game 11	46	2	4.35
2016 WCC Game 12	33	0	0.00

Table 9: Results from analysing the 2016 world championship games.

The average positive rate across all 12 games is 11.73% which is significantly lower than the average of the engine games. We do see a handful of games with higher rates of engine moves, this is potentially due to the fact that they are slightly longer, indicating that the players were trying to push for a win and avoid a draw, leading to more complex positions and as such more engine-like moves. In particular, game 10 has a significantly higher positive rate of 41.28%. This is an anomaly, as we know neither of the players were cheating. Despite this, the result is still less than the lowest engine game positive rate of 51.16%.

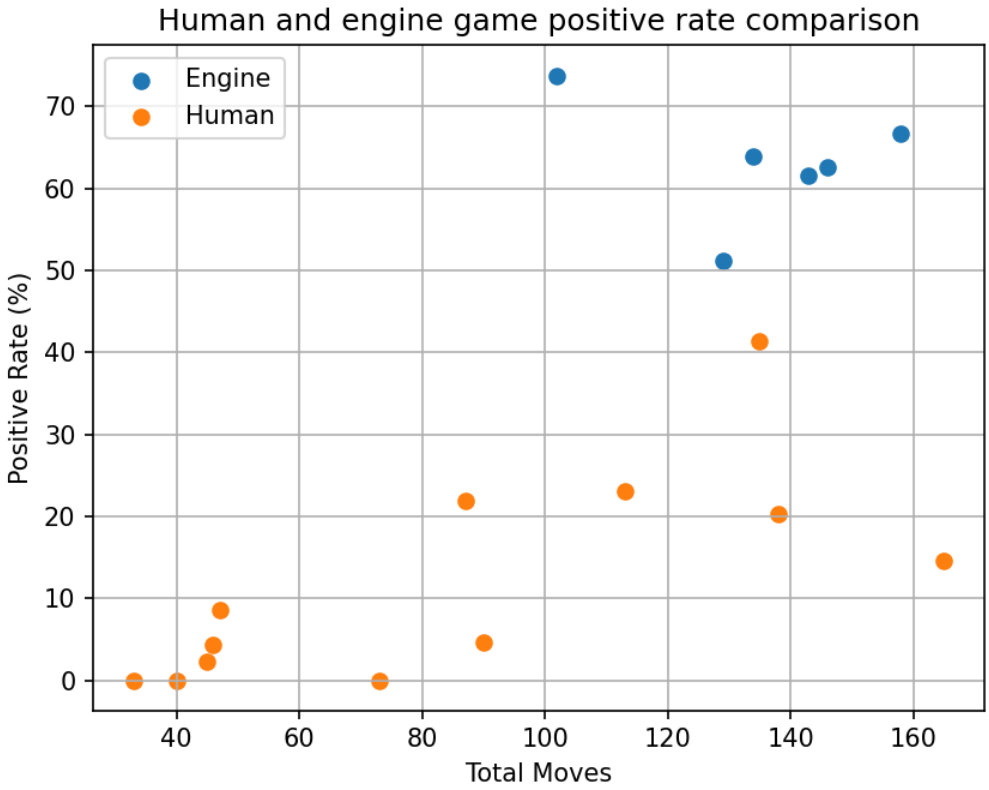


Figure 25: Positive rate compared to total moves for each game.

As we can see from Figure 25 above, we can quite clearly distinguish human games from engine games by observing the positive rate (PR). A threshold of $PR > 50\%$ appears to be a reasonable boundary to draw.

4.3 Opening Phase as a Control Variable

Clearly the system is capable of distinguishing human games from engine games, but these results are from many games playing different openings. We now perform a test to ensure this is not affecting the performance of the system. Three games played by chess masters were chosen and three corresponding engine games which share the same opening moves as their counterparts. All six games were analysed and their positive rates were compared.

Game	Total Moves	Engine Moves	PR (%)
Stockfish - Lc0	151	60	74.07
Tripoteau - Rapport	64	13	30.23

Table 10: Test one. The first 16 opening moves in both games were identical.

Game	Total Moves	Engine Moves	PR (%)
Torch - Stockfish	124	53	77.94
Amin - Carlsen	100	7	7.00

Table 11: Test two. The first 14 opening moves in both games were identical.

Game	Total Moves	Engine Moves	PR (%)
Stockfish - Torch	167	64	60.34
Braun - Ostl	63	11	26.19

Table 12: Test three. The first 17 opening moves in both games were identical.

As we can see from the above results, even given two games with identical openings, we can still use the system to correctly distinguish human games from engine games.

4.4 Tournament Analysis

Early in 2024, Optiver hosted an online UK chess championship [32]. Being hosted on Lichess [28] this gives us access to the PGNs of all the games. Given the large number of games, and the processing power required to analyse each one, a sample of 138 of the hundreds of games in the tournament were covered.

As shown in Figure 26 below, 136 of the the 138 games had a positive rate below 50%, the chosen threshold for a ‘human’ game. This is a great result, as we see that the system does not have a high false positive rate when classifying ‘human-human’ or ‘engine-engine’ games.

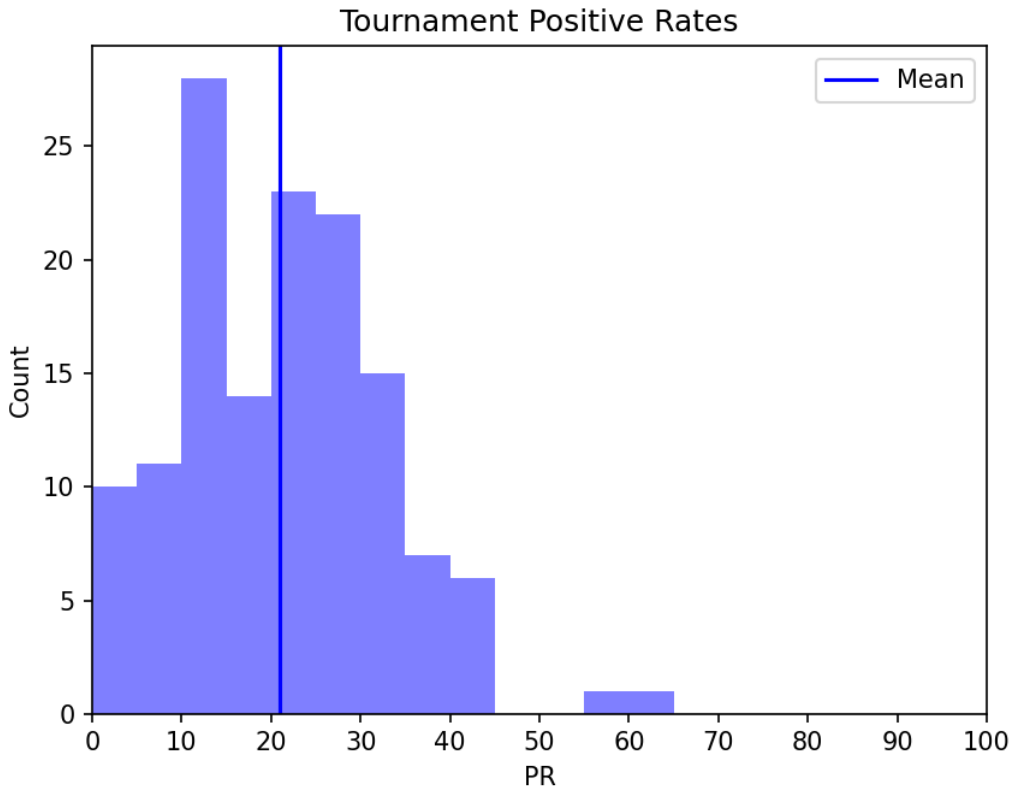


Figure 26: Positive rates of all 138 games. Mean PR is shown, at 21%.

4.5 Real Cheating Examples

The most obvious type of cheating to try and detect is naïve cheating. We established that there aren't many examples of human versus computer matches, however one game was found between grandmaster Hikaru Nakamura and Stockfish. Clearly, Stockfish won. However, the report below shown in Listing 18 gives us an interesting result.

```
Report for Hikaru:
Moves: 38
Good moves: 15 (39.5% of all moves)
Brilliant moves: 4 (10.5% of all moves)
Engine moves: 1 (3.4% of analysed (29),
                 6.7% of good moves, 10.0% of good moves made during critical period)
Average confidence of engine moves: 61.46%

Report for Stockfish:
Moves: 39
Good moves: 26 (66.7% of all moves)
Brilliant moves: 8 (20.5% of all moves)
Engine moves: 10 (33.3% of analysed (30),
                  38.5% of good moves, 55.6% of good moves made during critical period)
Average confidence of engine moves: 54.26%
Total positive rate: 18.64%
```

Listing 18: Report of a game between Stockfish and a top-level player.

We see a large difference in the positive rate between the two players (3.4% versus 33.3%) where Stockfish has considerably more engine moves, this was not present in the other anomalous results before. Game 10 of the 2016 WCC had a positive rate of 41.28% yet this was because both players had high positive rates. The same is true for the result in the Optiver tournament with positive rate 61.36%.

This difference is a good indication that the system is capable of recognising naïve cheating, especially given the second example of a grandmaster playing a cheater in Listing 17 where the positive rate difference between the two players was 22.2%.

The final performance measure is the ability to detect intermittent cheating. This was done by finding a player banned from Chess.com for violating the fair play policy and examining the accuracy of their games to confirm that they were banned for engine use (see Appendix C). Inspecting the games closely shows that they did not play perfectly which means the player was cheating intermittently. The last 28 games played by this player were analysed and the positive rate for the cheater in each game was measured. Of these 28 games, five contained opponents who also cheated which were removed for this experiment. The time control was 'rapid' with 10 minutes for the entire game, and the cheater's Elo was around 2000.

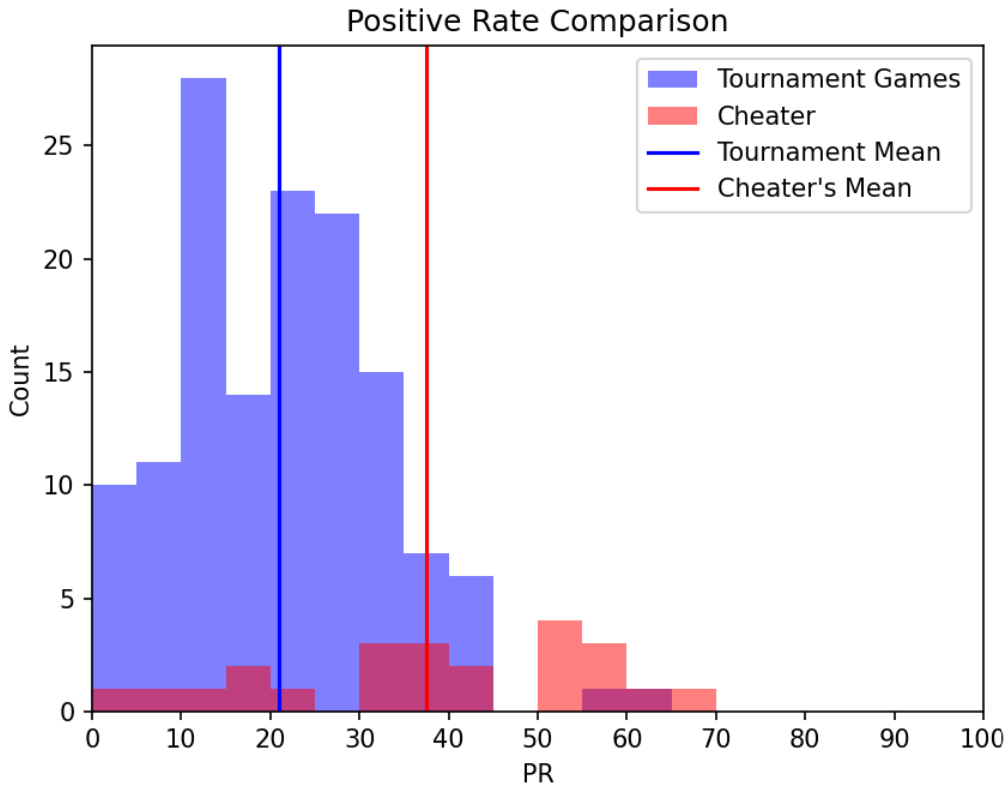


Figure 27: The intermittent cheater has a much higher positive rate than other games.

In Figure 27 above we see the histogram of the cheater's positive rates overlayed ontop of the histogram of the tournament games' postive rates. The mean of the cheater's positive rates is significantly higher, not far off being double. We even see that nine of the 23 games were above 50% positive rate.

We can also compare the cheater's positive rates to the positive rates of the opponents in the same games, this is shown below in Figure 28. This shows an even greater difference in the mean positive rates, with the opponents having a collective positive rate of 15% and the cheater having a positive rate of 38%. Also note that the cheater did not win every single game, the cheater won 11 out of 23, less than half. This is also an interesting observation, it implies that the system still identifies engine moves even when masked by poor ones that cause the player to lose.

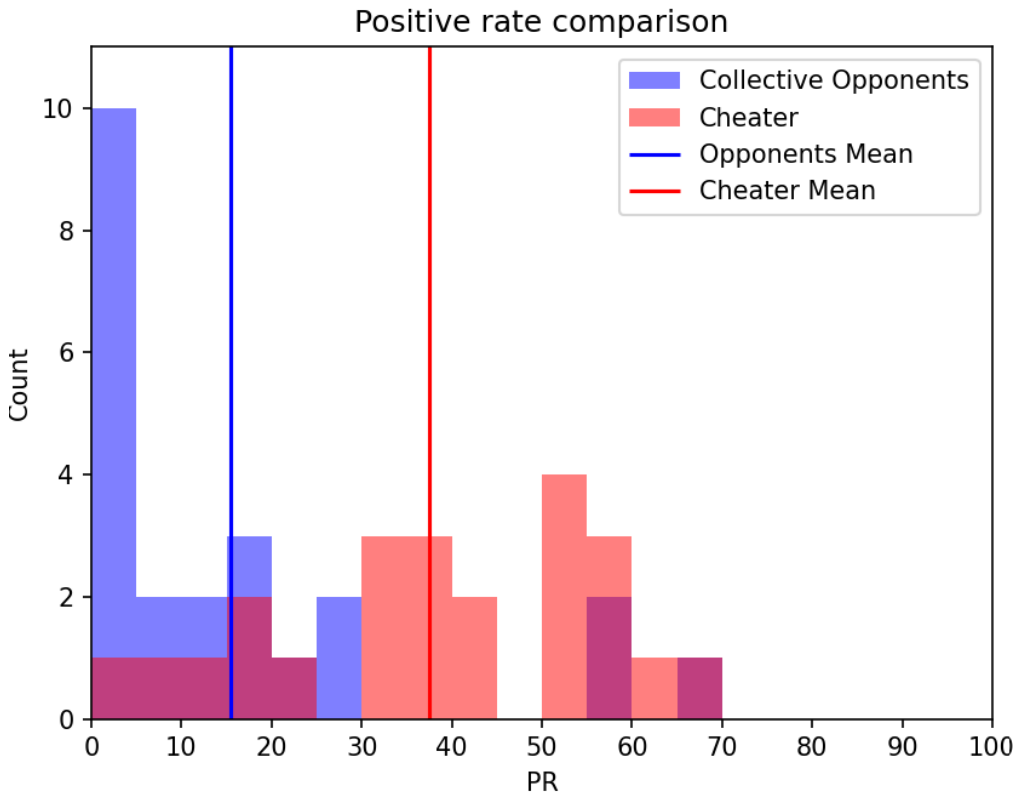


Figure 28: Comparison between the 2000 Elo cheater and their opponents.

A more difficult example was analysed too with another player (see Appendix D). The same process was used to deduce that the player was an intermittent cheater, but this time the games were 'blitz chess' with only three minutes for the entire game. This means that the cheater may not have had time to consult their engine in certain situations, or their engine may have suggested worse moves than usual as it could not analyse for as long.

Additionally, this player's Elo was 2800 which meant they were facing significantly stronger opposition, increasing the quality of the games overall and making it a harder example to perform well on. This time 32 games were analysed, and the cheater scored 24/32. Figure 29 below shows the comparison of their positive rate to the positive rates of their opponents.

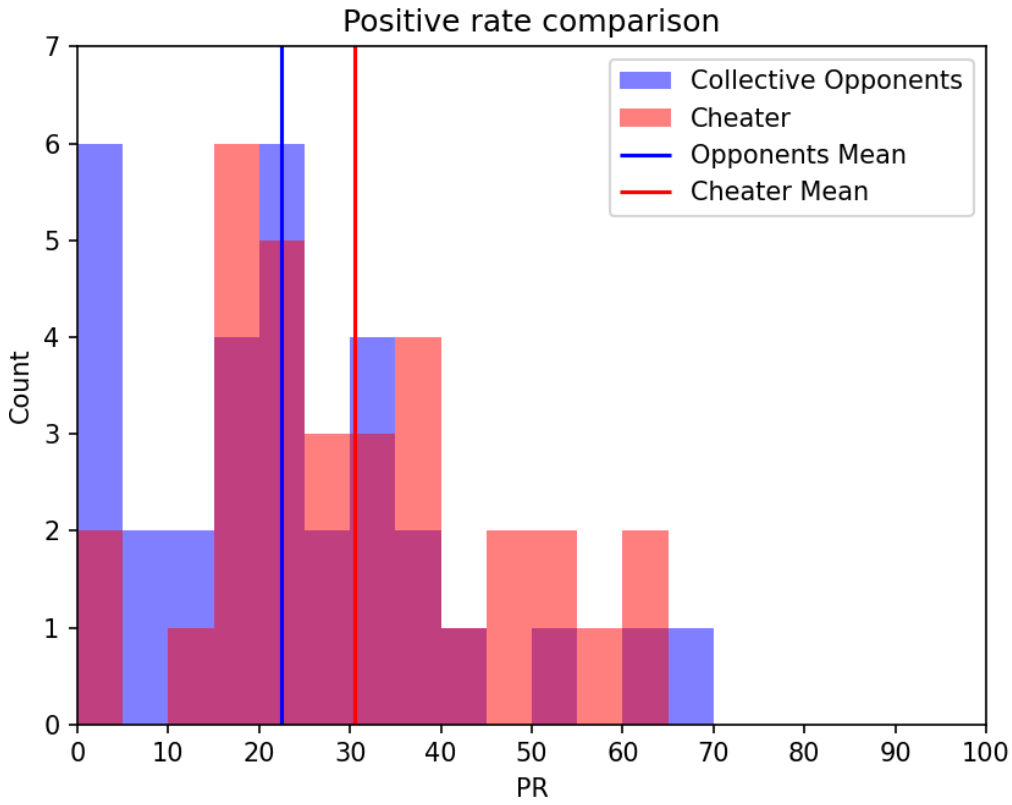


Figure 29: Comparison between the 2800 Elo cheater and their opponents.

The cheater's mean positive rate was still higher than the average of the opponents, but by a smaller margin this time. We also observe that the cheater had more than double the number of games with $PR \geq 45$ and less than a third of the number of games with $PR < 15$ compared to their opponents. Even in this more challenging case, the results still show the system detecting more instances of computer assistance from the cheater, but this is much less convincing than the '2000 Elo cheater' case shown in Figure 28.

4.6 Analysing Legitimate Games

After seeing results from players banned for receiving computer assistance, we perform the same analysis on a legitimate player's online games. While we did analyse world championship games, those are in person events and are the result of months of preparation from both players. We can't necessarily use them as a fair comparison to the casual online rapid and blitz games of the cheaters.

To be certain that the chosen player was not a cheater, for this experiment I am the legitimate player (see Appendix E). The same process as before was used to select the games, resulting in 42 analysed games.

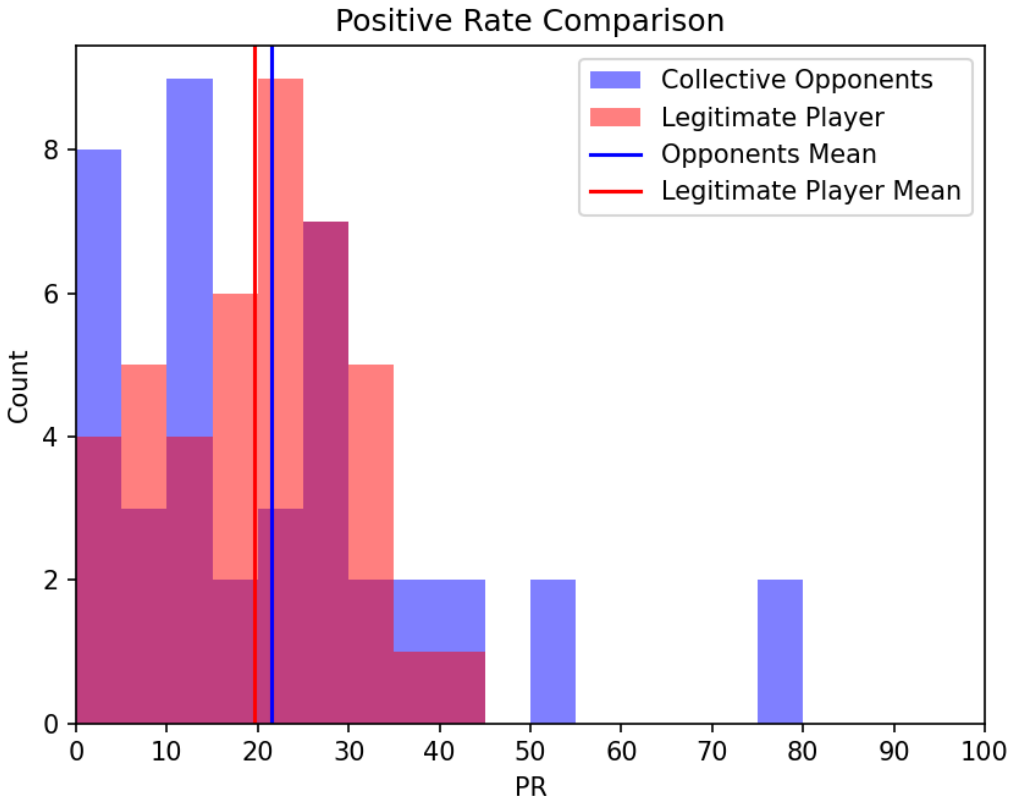


Figure 30: Comparison between the legitimate player and the opponents.

In Figure 30 above, we can see a difference from the two plots of the other intermittent cheaters. The average positive rate of the legitimate player is marginally below that of the opponents. It is also important to observe that the average positive rates here of the opponents and of the legitimate player are around 21%, almost exactly equal to the average positive rate of the 138 games from the Optiver tournament, which is equally very similar to the average positive rate of the 2800 Elo cheater's opponents.

To be more precise with our expected positive rate calculation for a legitimate player we can collate all of the games analysed from the Optiver tournament, both cheaters, and the legitimate player. Then we can calculate the mean positive rate for each individual player

excluding the two known cheaters. To remove outliers we can remove any data points based on the average of less than three games. This fits a normal distribution, plotted below in Figure 31. This time each data point represents a player, not an individual game.

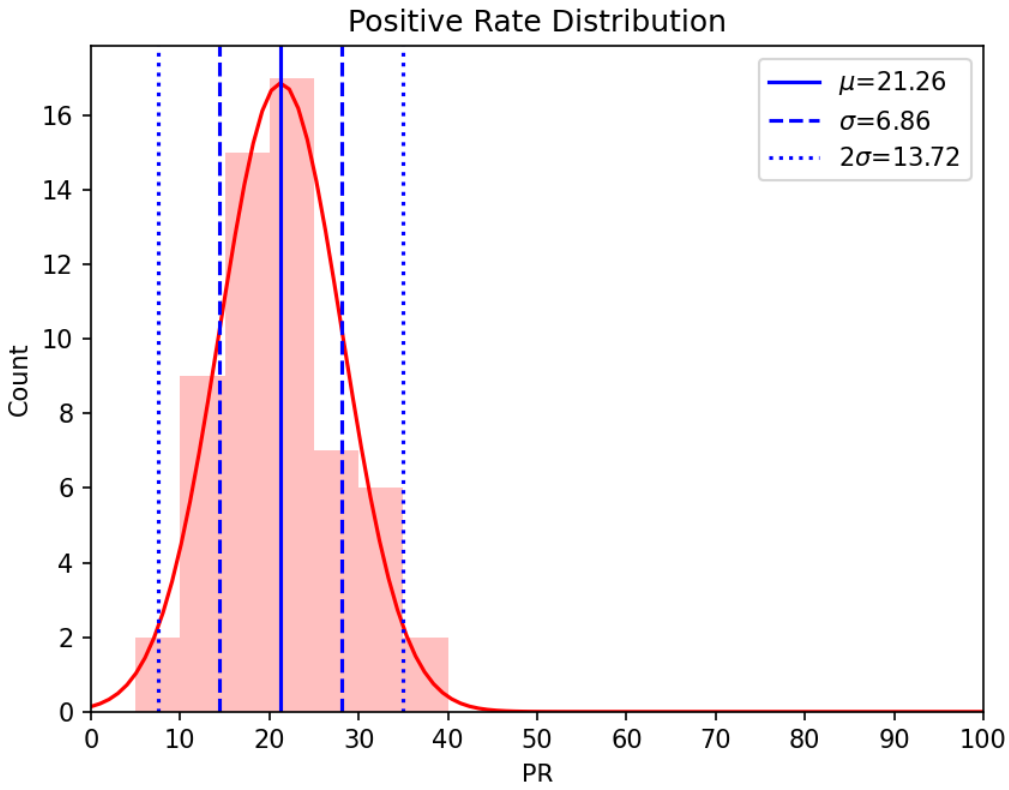


Figure 31: Distribution of average positive rates of legitimate players.

We can see that the average positive rate is very close to 21%, but we also observe that the standard deviation $\sigma = 6.86$ is small. With this data in mind, and our previously observed positive rates of intermittent cheaters, it would be reasonable to begin to suspect players who consistently maintain positive rates above 30% over long periods of time. This value exceeds slightly less than 90% of the positive rates in this sample. This also accounts for the fact that this sample is based on three game averages, as players continually play games we would expect their rolling average to dip closer to the mean (under 30%) at some point if they were legitimate. While it may not be enough evidence to accuse a player, it can be used to pick which players may benefit from further analysis or investigation to explain their results.

4.7 Consistency

While these tests are promising, and show the system performing well, these are the results obtained from running Stockfish's analysis just once. To measure the consistency of these results, the same game (Listing 17) was analysed from scratch 12 times. For white, the standard deviation of engine moves was 0.72 (with a range of 2) and for black, the standard deviation was 1.04 (with a range of 3). Therefore, the majority of the time we can expect a second run to be within ± 1 engine move of the first run. This is a low enough variation to have reasonable confidence in the results generated even from running the system just once.

4.8 Move Distribution Analysis

After the project presentation, an idea was suggested by the project supervisor Long Tran-Thanh to investigate the distributions of engine moves present in games.

A script was written to visualise games with each row representing a game and each pixel representing a move. Yellow pixels on the left are opening moves (on the right they simply fill the space after the end of the game). Purple pixels represent the middlegame, where the evaluation is within ±300 centipawns. Orange pixels show sections where one player is clearly winning. Finally, dark blue pixels are engine moves. An example is shown in Figure 32 below.

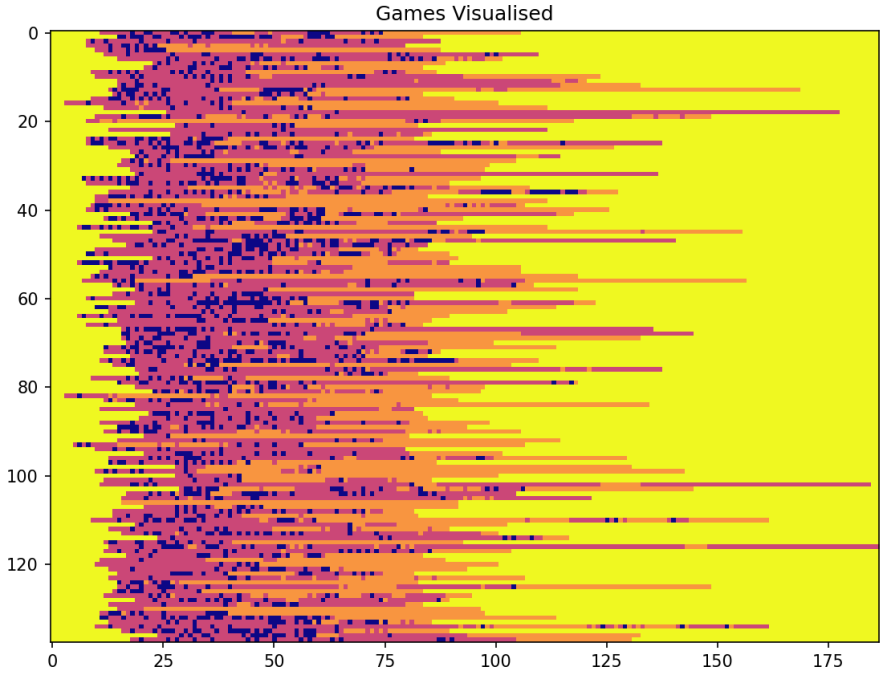


Figure 32: All 138 Optiver tournament games visualised.

As we can see, the majority of games have few engine moves flagged. There are some games with clusters of engine moves, which is expected because adjacent moves in the game will have been played in almost identical positions so a single false positive may sometimes 'carry over' to the next few moves. We see in some cases this effect is seen but with an alternating pattern, where one player has two or three engine moves yet the opponent's moves in between

are not flagged. However, overall there is no obvious pattern to be observed here.

We can look for vertical patterns in the visualisation too, even though games are independent from each other there may be common points where more engine moves are flagged. Figure 33 below shows the counts of the number of engine moves (dark blue pixels) found in each column of the visualisation.

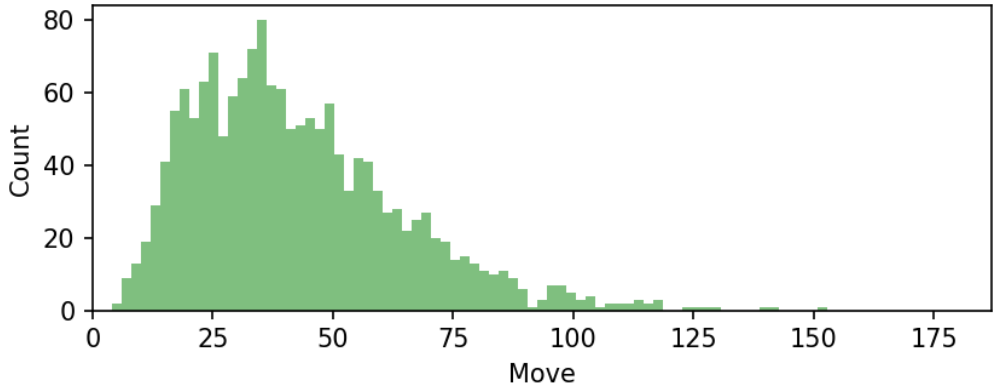


Figure 33: Number of engine moves flagged on each turn of the 138 games.

The fact that some games stay in opening theory for longer explains the gradual increase from 0 to 25 and the fact that some games end earlier explains the tail off towards the end. We can visualise the two sets of games involving the 2000 Elo and 2800 Elo intermittent cheaters too, as demonstrated in Figure 34 below.

However, we see similar distributions but just with a higher positive rate. This is expected from intermittent cheating, because potentially a significant portion of moves will be legitimate. Therefore the moves where an engine has been used in each game will be scattered and spread across all of them, and it is unlikely that we would spot a difference.

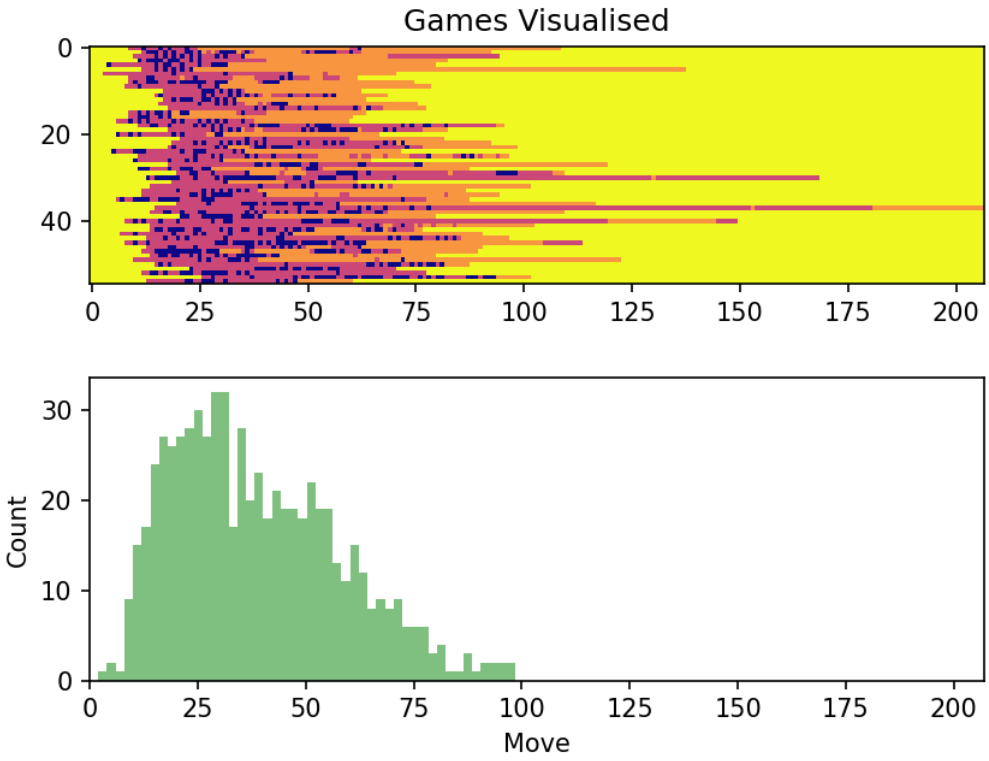


Figure 34: Visualising games involving intermittent cheaters.

These will be useful plots to keep track of if the system is expanded in the future. We would not want to see distributions of engine moves in human games diverge significantly from the bell curve displayed here, it would indicate over fitting as the model would be learning to detect engine play based on specific move numbers.

5 Evaluation

5.1 Overall Success of the Project

Overall the project has been successful. The system is capable of classifying 'human-human' games and 'engine-engine' games with high accuracy which is an excellent result. Additionally, on the few games tested, the system shows that it can highlight naïve and to an extent intermittent cheating through the positive rate imbalance between the two players in the games.

It is also important to refer back to the original objectives listed in Section 2.1, when evaluated on these objectives the project is still successful. The only objective which was not fully met was the fifth objective. This was an extension task, completing it fully would have only resulted in slightly more content in this report. Therefore, it did not impact the development and final state of the system developed.

BASE 1. The program allows users to input multiple games which can then be analysed, sequences of moves are highlighted and engine moves likely to have been suggested by computers are annotated with comments too. Achieved ✓

BASE 2. Many machine learning models were tested and considered, with XGBoost being chosen as the best option due to its ability to fit jagged and complex decision boundaries and its suitability with regards to the amount of training data available. Over 200 total games were analysed to form a large database for training and testing the model. Achieved ✓

BASE 3. Stockfish, an open source engine, was used to generate evaluation data for moves to build features for the machine learning model. The opening stage of the game was specifically handled by a different component consulting opening books. Checkmate positions at the end of games were handled by a risk assessment algorithm, determining how risk-averse (human) the moves were. Achieved ✓

STRETCH 4. Two machine learning models, support vector classification and XGBoost, were em-

ployed and combined to generate the final classification for each move. Achieved ✓

STRETCH 5. Many games were analysed after the model was trained, and the results were plotted and studied. Some hypotheses were drawn about what attributes engine moves have (they are more risky, they tend to try to complicate positions) however to gain deeper insights into the playing style of engines as defined in the original objective, moves would need to be visually considered in the context of their games and properly understood on a game-by-game basis. This was the only task which was too much work, at the time of writing the specification it was an interesting idea and seemed reasonable but after completing development it became clear that it was too specific. It could still be an interesting extension if more time was granted for the project. Mostly Achieved ✓

5.2 Limitations and Difficulties

The system itself is capable of analysing a game in only a few minutes, on average only two to three with particularly long games being up to six minutes. However this is still a relatively large amount of time when considering the number of games in some tournaments. It is quite possible for a tournament to have over 1,000 games, this could easily take 50 hours of processing running on a single machine. If we consider online chess, sites have millions of games per day, these definitely cannot all be analysed. Ideally each game would be analysed in just a handful of seconds, this would be very difficult to achieve though.

This is made worse by the fact that Stockfish's evaluations are not always the same. Running the system on a game may produce different results and positive rates, if we were to attempt to build evidence to make a cheating accusation against someone we would likely need to run the system multiple times to get more confident results. This takes time, and wastes processing power.

The biggest obstacle in this project was Stockfish not evaluating positions perfectly. When analysing games played by Stockfish (on very strong hardware) using Stockfish (on a laptop), some moves were labelled as minor inaccuracies. This is inconvenient, as these 'inaccuracies' would not be included in the training data. A move played by a strong engine that even

another engine cannot appreciate would be a great example of a move that a human would not play, and would have been beneficial to include if it was evaluated properly. The idea of re-evaluating a position after being given the idea to play it helped to mitigate this, but not entirely.

Finally, a natural limitation of the system is the noticeable false positive rate. In almost every human game, there are at least some engine moves spotted. This is a limitation of all analytical cheat detection systems. We are able to mitigate the effects by increasing the sample size of analysed games from suspect players and averaging across them. But ideally we would like to strengthen our conditions for what an 'engine move' really is. This would reduce the minimum number of games required to have confidence in the results, which is an important metric too.

5.3 Future Expansions

There are many opportunities for this project to be expanded upon in the future, or even if the deadline for the project was extended.

It would be interesting to explore the use of a different engine, such as Torch [7] if it ever gets released for the public to download. Games could be re-analysed using Torch by simply swapping the engine executable, then all of the same features can be extracted from moves and the machine learning models can be re-trained. It is difficult to predict whether this would perform better than Stockfish, but it would certainly be worth experimenting with.

Another interesting extension would be to optimise the program and create a lightweight version. For example, limiting Stockfish's evaluations and training the machine learning models on lower depths. This would involve re-labelling new evaluation levels vectors, but observing how this affects performance would be useful information. If the system performed sufficiently well while analysing to seldepth 20 instead of seldepth 30 then in most cases this could be used, reducing the processing time for each game from minutes to seconds potentially. The inverse is also true of this extension, with access to more powerful hardware analysing to seldepth 40 may produce even better results.

The dataset being used to train the XGBoost model could also be refined. Currently the model is being trained on games which are either 'engine versus engine' or 'human versus human'. A more thoughtful approach to gathering data might help. By taking more time to develop a filter removing any human-like moves, instead of treating every move from a known engine game as an engine move, the accuracy of the model could be improved.

Finally, engineering more features could improve the system too. Stockfish is an open source engine, therefore the code itself can be inspected and some more data could be extracted during the analysis process. Using this extra data, new features for the machine learning model could be created. This might improve the system, or make it more efficient if the new features are much better predictors than the old ones (and require analysis to a lower depth).

5.4 Research Opportunity

This is just one approach to detecting cheating, there are many other methods which have not been explored yet. Detecting cheating in chess only recently became a popular topic after the Carlsen and Niemann lawsuit [3] and as such there is a lot of research potential here.

As new techniques are developed in the fields of artificial intelligence, machine learning, and even neural networks, we can assess how suitable they are for this task and see if they perform better than what we have currently. Eventually, it is possible that cheat detection would become almost developed enough to automatically catch and punish online players for naïve cheating, and flag intermittent cheaters for investigation accurately without waiting for them to be reported by other players.

6 Project Management

6.1 Methodology Employed

The project followed a SCRUM approach by splitting the workload into week-long sprints and assigning story point estimates to each of the tasks, maintaining a kanban board with a backlog and organising the tasks into “to do”, “in progress” and “done” with Jira [26]. This complimented the project schedule given in Table 2, with each week or two week period being given one large task. These big tasks were then split into smaller work packages and added to the backlog to be completed throughout the week. This was an individual project however, so nobody else was acting as the SCRUM master or product manager and daily stand up meetings were not held.

For many tasks, this worked very well. For example, researching the features of chess moves for the model involved looking at related papers, setting up the Stockfish engine, and looking at UCI documentation to see what data can be extracted. Laying out all of these steps at the start of the allocated time slot and estimating which ones would take the longest was very useful, and helped the project to get started smoothly.

There were points where the project deviated from the original schedule however. After the first two sprints, the plan diverged from the intended data mining sprints, instead working on multi-depth analysis focused sprints for the next weeks. This was not something that could have been predicted at the start of the project, but ultimately this improved the overall system and became one of the most interesting parts. Additionally, thanks to the flexibility in the original timetable, the tasks which were replaced by the suddenly inserted multi depth analysis sprints were completed over the Christmas break instead.

Finally, it was decided during the project development that the final report would not require as long as initially expected to write and that the time would be better spent on refining and testing the main XGBoost model. This change to the schedule is reflected in the final project timeline shown in Appendix A, with a list of all major project components and their sub-task breakdowns listed.

6.2 SCRUM Sprints

Across the whole project, there was one sprint each week apart from the two scheduled breaks in term one totalling 26 work sprints. Almost all sprints were completed without letting tasks return to the backlog incomplete, however a few tasks in the 'Main Machine Learning Model' component were moved to a later sprint as their story point values were slightly under estimated and required more time as can be seen in the second half of the Gantt chart in Appendix A. Despite this, the project did not fall behind.

6.3 Jira

As discussed above, Jira supports kanban boards with custom columns. Below Figure 35 shows the kanban board structure used for the sprints. The 'testing' column was removed so as to simplify the board as there is no dedicated tester. All coding components were still tested before being moved to 'done'.

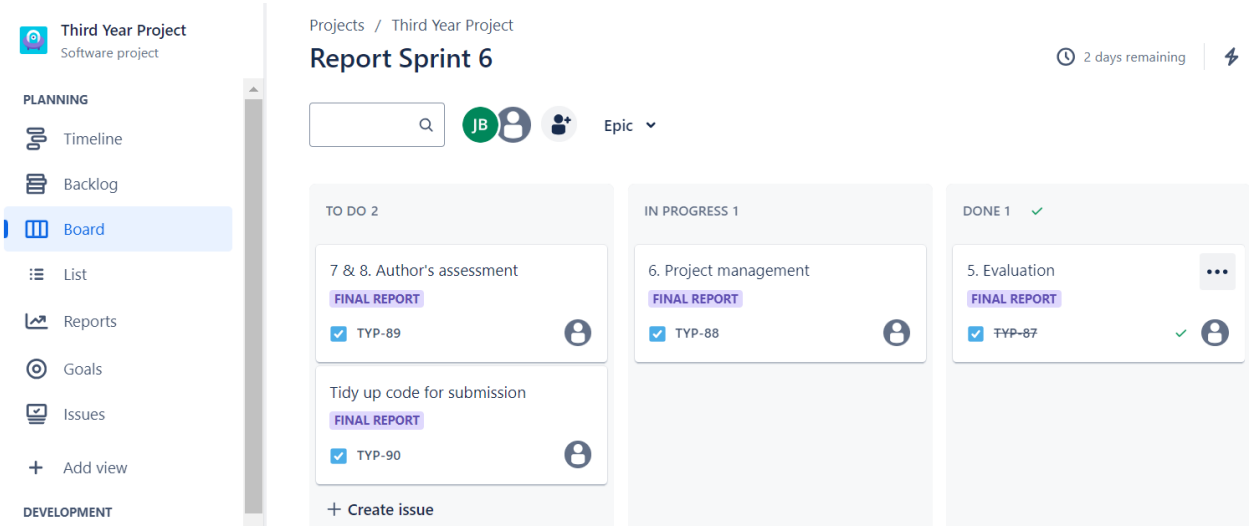


Figure 35: Screenshot of the current sprint's kanban board from Jira.

6.4 GitHub

To organise the codebase for the project, GitHub was used [22]. Code was committed to GitHub for version control and as a cloud backup utility in case the local version of the code

was lost.

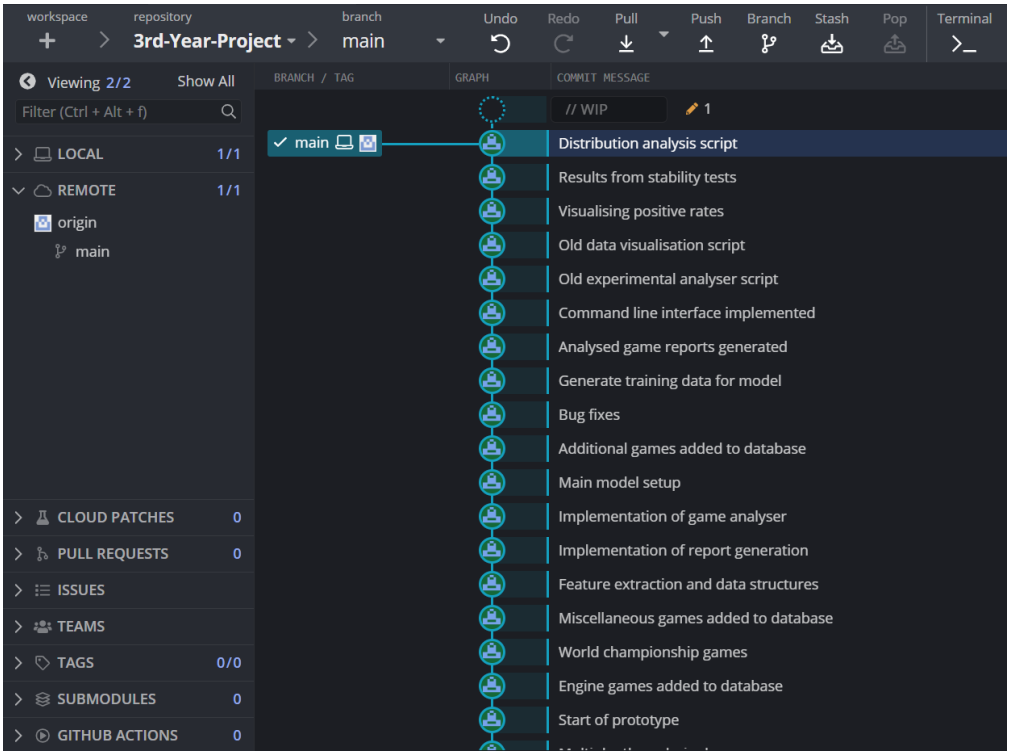


Figure 36: Various code commits, linked to GitHub.

6.5 Supervisor Meetings

Aside from working on the sprints, during term one supervisor progress update emails were sent weekly detailing the work completed during the week and some of the work planned to be done next. There were also a few supervisor meetings throughout term to keep in touch and discuss the project. Weekly updates continued throughout the Christmas break, but meetings were not necessary during this time as development was already planned and being completed without issues.

At the start of the year, meetings were planned to be almost weekly in term two. However, upon reaching term two it made more sense to simply meet after completing components and give quick progress updates, discussing the next steps and roughly planning when the next meeting would be. This was a better approach than meeting just to state that work had been done, which could be conveyed through an email instead.

7 Author's Assessment of the Project

This project designed and implemented a new method of purely analytical cheat detection in chess. Starting from raw transcripts of moves and finishing with data showing evidence of being able to recognise characteristics of computer moves, something which is becoming widely sought after as cheating becomes a more serious issue.

However, as we know, the accuracy is limited. Some examples show false positives, and in the case of an intermittent cheater at a high Elo the evidence for cheating was weaker. There is still room for improvement within the system in terms of optimisation too.

Putting computer science theory into practice in a game people can play is a great way to learn. Chess specifically allows many aspects of computer science to be applied, using reinforcement learning to build engines, or applying neural networks to evaluate positions like Stockfish's NNUE [25]. The project adds to this list, demonstrating even more techniques put into practice to create something useful.

There is a demand for effective cheat detection methods from sites like Chess.com [6] and lichess [28], where prize money from tournaments can be effectively stolen by cheaters using engines. Not only can the project assist here, but it can be expanded upon too. Once established, others can experiment with different engines, datasets, and features to improve it even further.

Being able to detect cheating in this way, particularly intermittent cheating, is a very relevant problem. If we were able to do this with high accuracy, it would be employed by all major online chess sites. With this in mind, the fact that the project has evidence of being able to detect cheating is a very exciting result.

References

[1] Anthony K Akobeng. "Understanding diagnostic tests 3: receiver operating characteristic curves". In: *Acta Paediatrica* 96.5 (2007), pp. 644–647. DOI: https://doi.org/10.1111/j.1651-2227.2006.00178.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1651-2227.2006.00178.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1651-2227.2006.00178.x.

[2] David J. Barnes and Julio Hernandez-Castro. "On the limits of engine analysis for cheating detection in chess". In: *Computers & Security* 48 (2015), pp. 58–73. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2014.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0167404814001485.

[3] Andrew Beaton and Joshua Robinson. *Hans Niemann's $100 Million Lawsuit Over Chess Cheating Allegations Is Dismissed*. 2023. URL: https://www.wsj.com/sports/chess-cheating-scandal-hans-niemann-magnus-carlsen-7fcbdb56 (visited on 29/11/2023).

[4] *Champions Chess Tour 2023: Magnus Carlsen scoops third title in a row*. URL: https://www.fide.com/news/2805 (visited on 23/03/2024).

[5] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[6] CHESScom. *About Chess.com*. 2023. URL: https://www.chess.com/about (visited on 23/03/2024).

[7] CHESScom. *Announcing Torch: New #2 Chess Engine*. 2023. URL: https://www.chess.com/news/view/torch-chess-engine (visited on 01/12/2023).

[8] CHESScom. *Computer Championship Rating Brawl: Fall*. 2023. URL: https://www.chess.com/computer-chess-championship#event=rating-brawl-fall-2023&game=34 (visited on 29/11/2023).

[9] CHESScom. *It's TIME To Acknowledge Chess: Chess.com Named In Prestigious 100 Most Influential Companies List*. 2023. URL: https://www.chess.com/news/view/chesscom-among-100-most-influential-companies-2023 (visited on 20/03/2024).

[10] CHESScom. *Live Now: The New Computer Chess Championship*. 2018. URL: https://www.chess.com/news/view/announcing-the-new-computer-chess-championship (visited on 06/04/2024).

[11] Chessdom and Chessdom Arena. *TCEC (Top Chess Engine Championship)*. URL: https://tcec-chess.com/ (visited on 01/12/2023).

[12] *Chessgames Statistics Page*. URL: https://www.chessgames.com/chessstats.html (visited on 27/04/2024).

[13] Codekiddy2. *15 Million Games Chess Database Files*. 2015. URL: https://sourceforge.net/projects/codekiddy-chess/ (visited on 29/04/2024).

[14] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[15] Steven J. Edwards. *Portable Game Notation Specification and Implementation Guide*. 1994. URL: https://www.chessclub.com/help/PGN-spec (visited on 28/03/2024).

[16] Arpad E. Elo and Sam Sloan. *The rating of chessplayers : past and present*. Ishi Press International, 2008. URL: https://cir.nii.ac.jp/crid/1130282270181653248.

[17] *FIDE*. URL: https://www.fide.com/fide/about-fide (visited on 23/03/2024).

[18] FIDE. *Regulations for the FIDE World Championship Match 2021*. 2021. URL: https://handbook.fide.com/files/handbook/FWCM2021.pdf (visited on 06/04/2024).

[19] Niklas Fiekas. *A chess library for Python*. 2023. URL: https://pypi.org/project/chess/ (visited on 01/12/2023).

[20] Ulrike Fischer. *xskak: An extension to the package skak*. 2019. URL: https://texdoc.org/serve/xskak/0 (visited on 23/04/2024).

[21] Python Software Foundation. *Python documentation*. 2024. URL: https://docs.python.org/3/ (visited on 05/04/2024).

[22] *GitHub*. URL: https://github.com/about (visited on 20/04/2024).

[23] Feng-Hsiung Hsu. "IBM's Deep Blue Chess grandmaster chips". In: *IEEE Micro* 19.2 (1999), pp. 70–81. DOI: 10.1109/40.755469.

[24] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[25] *Introducing NNUE Evaluation*. 2020. URL: https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/ (visited on 01/12/2023).

[26] *Jira | Issue & Project Tracking Software*. URL: https://www.atlassian.com/software/jira (visited on 03/12/2023).

[27] *Leela Chess Zero - Open Source Neural Network Based Chess Engine*. URL: https://lczero.org/ (visited on 01/12/2023).

[28] *Lichess*. URL: https://lichess.org/about (visited on 20/04/2024).

[29] TIME Magazine. *TIME 100 Most Influential Companies*. 2023. URL: https://time.com/collection/time100-companies-2023/ (visited on 23/03/2024).

[30] Stefan Meyer-Kahlen. *Universal Chess Interface (UCI)*. 2000. URL: https://www.shredderchess.com/chess-features/uci-universal-chess-interface.html (visited on 01/12/2023).

[31] Harm Geert Muller. *Polyglot book format*. URL: http://hgm.nubati.net/book_format.html (visited on 19/04/2024).

[32] Optiver. *Optiver Chess Championship 2024*. URL: https://optiver.com/recruitment-events/optiver-chess-championship-2024/ (visited on 23/04/2024).

[33] Reyhan Patria et al. "Cheat Detection on Online Chess Games using Convolutional and Dense Neural Network". In: *2021 4th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. 2021, pp. 389–395. DOI: 10.1109/ISRITI54043.2021.9702792.

[34] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[35] Klaudia Prevezanos. *World chess champion: 'The computer never was an opponent'*. 2016. URL: https://www.dw.com/en/world-chess-champion-magnus-carlsen-the-computer-never-has-been-an-opponent/a-19186058 (visited on 06/04/2024).

[36] Ken Regan. *Catching Chess Cheaters*. 2014. URL: https://cse.buffalo.edu/~regan/personal/JuneCLarticleKWR.pdf (visited on 29/11/2023).

[37] Marco Costalba Tord Romstad and Gary Linscott Joona Kiiski. *Stockfish source code*. URL: https://github.com/official-stockfish/Stockfish/blob/master/src/search.cpp (visited on 23/04/2024).

[38] Guido Van Rossum. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.

[39] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

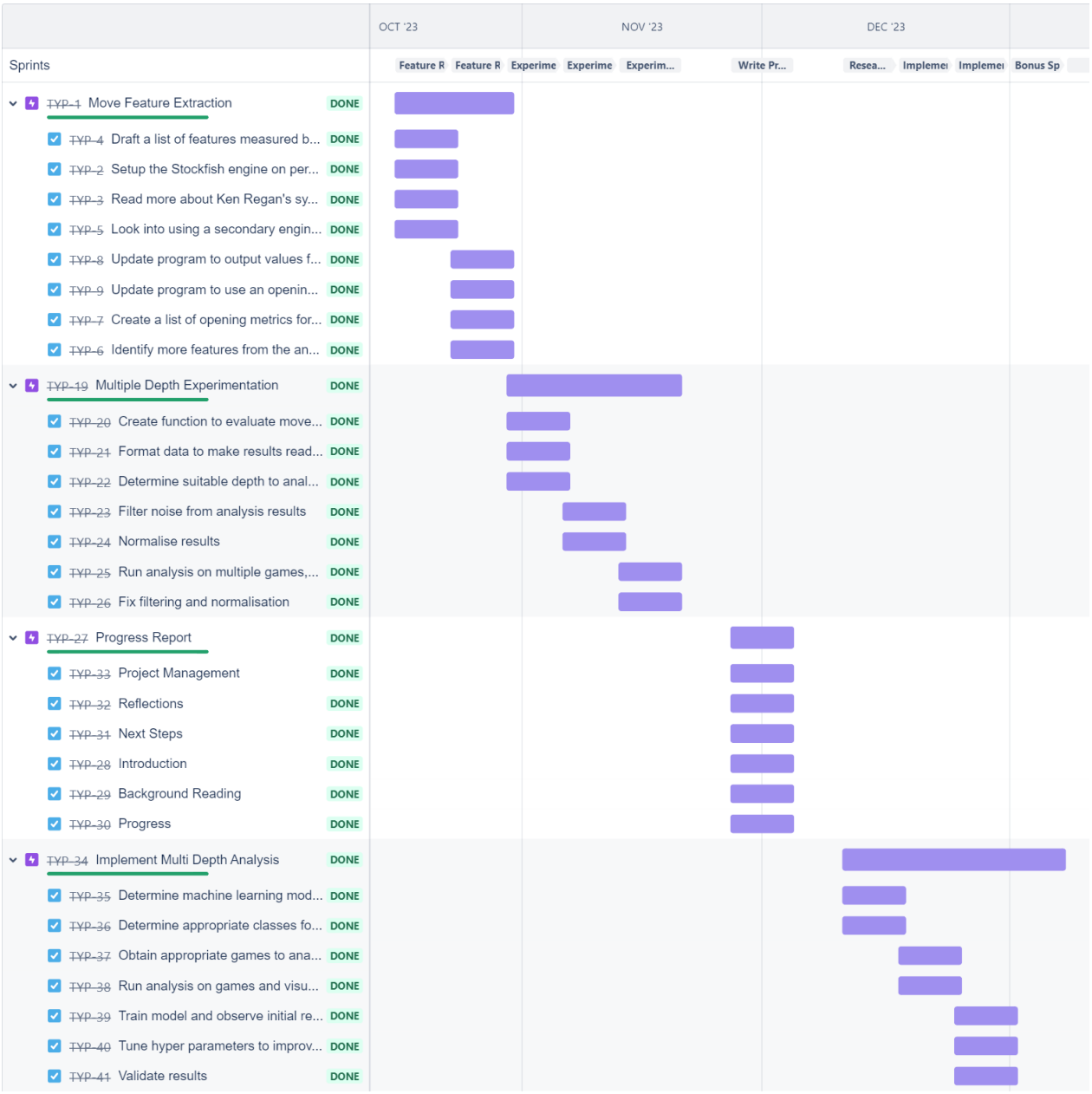
[40] Junjie Wu. *Advances in K-means clustering: a data mining thinking*. Springer Science & Business Media, 2012.

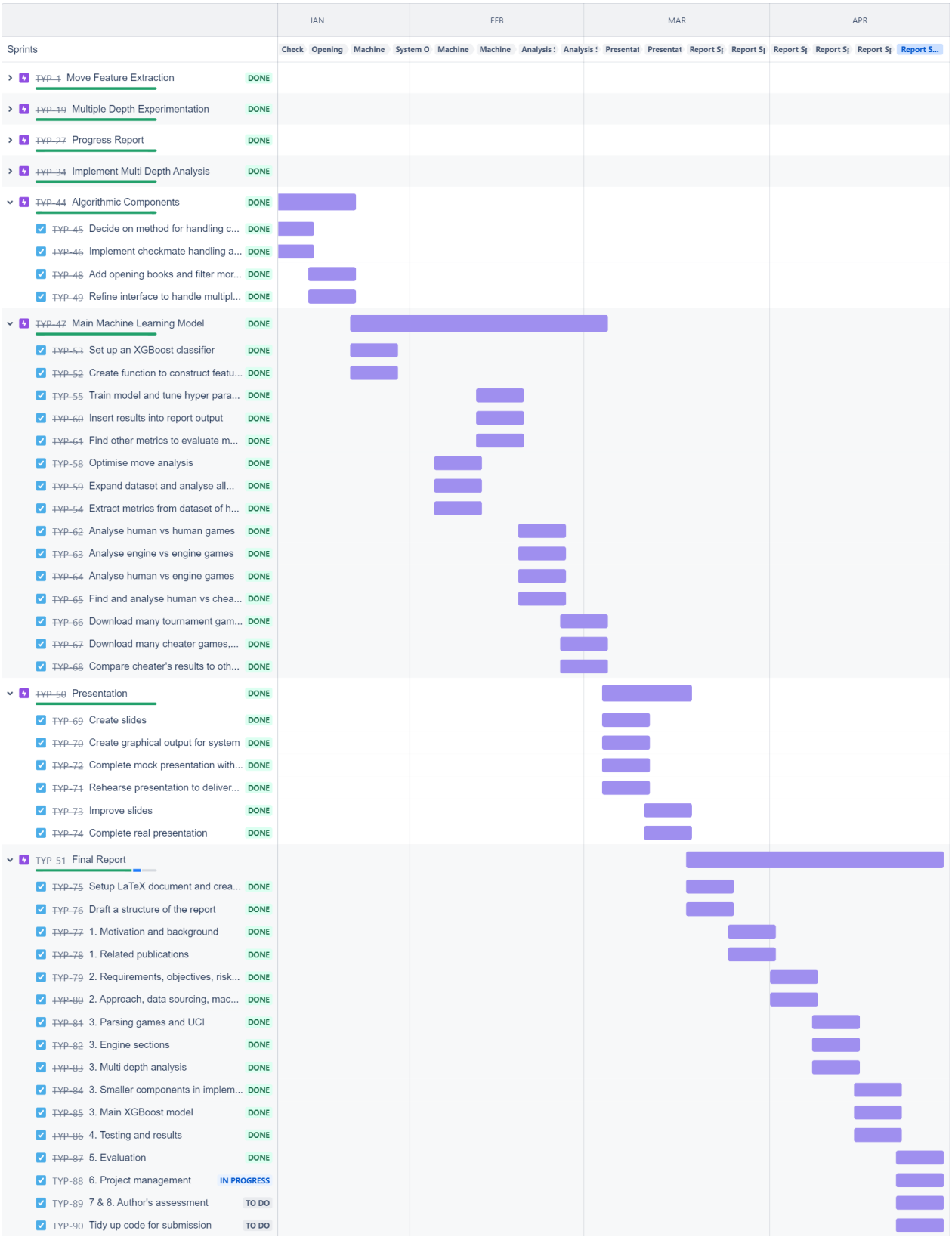
[41] Daylen Yang. *Stockfish - Open Source Chess Engine*. URL: https://stockfishchess.org/ (visited on 01/12/2023).

[42] V. B. Zakharov, M. G. Mal'kovskii and A.I. Mostyaev. "On Solving the Problem of 7-Piece Chess Endgames". In: *Programming and Computer Software* 45 (2019), pp. 96–98. ISSN: 1608-3261. DOI: https://doi.org/10.1134/S036176881903006X. URL: https://link.springer.com/article/10.1134/S036176881903006X.

Appendices

A Gantt chart of project timeline from Jira




JAN
FEB
MAR
APR
Sprints
Check
Opening
Machine
System O
Machine
Machine
Analysis !
Analysis !
Presentat
Presentat
Report S
Report S
Report S
Report S
Report S
Report S...
TYP-1 Move Feature Extraction DONE
TYP-19 Multiple Depth Experimentation DONE
TYP-27 Progress Report DONE
TYP-34 Implement Multi Depth Analysis DONE
TYP-44 Algorithmic Components DONE
TYP-45 Decide on method for handling c... DONE
TYP-46 Implement checkmate handling a... DONE
TYP-48 Add opening books and filter mor... DONE
TYP-49 Refine interface to handle multipl... DONE
TYP-47 Main Machine Learning Model DONE
TYP-53 Set up an XGBoost classifier DONE
TYP-52 Create function to construct featu... DONE
TYP-55 Train model and tune hyper para... DONE
TYP-60 Insert results into report output DONE
TYP-61 Find other metrics to evaluate m... DONE
TYP-58 Optimise move analysis DONE
TYP-59 Expand dataset and analyse all... DONE
TYP-54 Extract metrics from dataset of h... DONE
TYP-62 Analyse human vs human games DONE
TYP-63 Analyse engine vs engine games DONE
TYP-64 Analyse human vs engine games DONE
TYP-65 Find and analyse human vs chea... DONE
TYP-66 Download many tournament gam... DONE
TYP-67 Download many cheater games,... DONE
TYP-68 Compare cheater's results to oth... DONE
TYP-50 Presentation DONE
TYP-69 Create slides DONE
TYP-70 Create graphical output for system DONE
TYP-72 Complete mock presentation with... DONE
TYP-71 Rehearse presentation to deliver... DONE
TYP-73 Improve slides DONE
TYP-74 Complete real presentation DONE
TYP-51 Final Report
TYP-75 Setup LaTeX document and crea... DONE
TYP-76 Draft a structure of the report DONE
TYP-77 1. Motivation and background DONE
TYP-78 1. Related publications DONE
TYP-79 2. Requirements, objectives, risk... DONE
TYP-80 2. Approach, data sourcing, mac... DONE
TYP-81 3. Parsing games and UCI DONE
TYP-82 3. Engine sections DONE
TYP-83 3. Multi depth analysis DONE
TYP-84 3. Smaller components in implem... DONE
TYP-85 3. Main XGBoost model DONE
TYP-86 4. Testing and results DONE
TYP-87 5. Evaluation DONE
TYP-88 6. Project management IN PROGRESS
TYP-89 7 & 8. Author's assessment TO DO
TYP-90 Tidy up code for submission TO DO

B Generated Reports

Below a series of links can be found containing reports generated during the various experiments and tests of the system, each link provides access to up to 64 generated reports. The games are contained within unlisted studies on Lichess [28], they can only be accessed through these links.

On Lichess' interface, annotated engine moves will be highlighted and the games can be scrolled through easily while browsing the reports. The raw PGN files of the reports (the output of the system) can also be downloaded from there, either in bulk or individually.

Finally, the game used in the project presentation's demonstration is included under the miscellaneous collection as the first chapter.

Ha312 Reports	`https://lichess.org/study/5fQuVEnA`
Vartender Reports	`https://lichess.org/study/zbD90d1l`
Haku28866 Reports	`https://lichess.org/study/X4ro5usm`
Optiver Reports (1)	`https://lichess.org/study/1XASG25w`
Optiver Reports (2)	`https://lichess.org/study/fTpRO5rz`
Optiver Reports (3)	`https://lichess.org/study/K0T0N4VJ`
2016 WCC Reports	`https://lichess.org/study/oQEWf4Lt`
Engine vs Engine Reports	`https://lichess.org/study/65qGlqbE`
Demo + Miscellaneous Reports	`https://lichess.org/study/nWt9qSNO`

C Games From Ha312

Chess.com profile: `https://www.chess.com/member/ha312`
Link to games: `https://www.chess.com/games/archive/ha312`
Games which Ha312 won are green, games which they lost are red, and drawn games are grey.

Opponent (Elo)	Total Moves	Ha312 PR (%)	Opponent PR (%)
Isaackristal (2040)	100	50.0	0.0
Isaackristal (2049)	81	40.0	58.8
aKo168167 (2018)	85	34.6	0.0
Lenudit (1987)	72	31.2	28.6
BrainAche55 (2117)	76	42.9	21.4
Mihai1969 (1928)	126	50.0	0.0
Zlatafap (2215)	68	9.5	4.2
shashipalsharma (2064)	53	19.0	5.3
chessgrm (1990)	66	60.0	69.2
agnikitin (2055)	71	16.7	0.0
LivelyClaim (2215)	46	55.6	10.0
motigers (2055)	54	55.0	0.0
galacticism (2243)	47	55.6	12.5
Thomassina (2188)	60	35.7	16.7
RUSTAM693 (2014)	60	24.0	4.2
Luggomyeuggo (2318)	55	36.4	25.0
forkytry (2035)	52	66.7	0.0
Momcilozmaj (2003)	54	52.9	0.0
Hysenj_Hysa (1963)	85	31.7	16.7
baikal_53 (2062)	65	35.7	18.5
adamsbey (2081)	61	50.0	9.1
Hazem_1980 (2053)	61	10.5	0.0
Hazem_1980 (2063)	74	0.0	55.6

Table 13: Games played by Ha312, with positive rates analysed.

We can also observe how the positive rate changes with respect to the number of moves analysed, to verify that our positive rates are not based off tiny samples of moves. For example, situations where two moves were analysed and one was an engine move hence a 50% positive rate. We see that this is not the case by examining Figure 37 below.

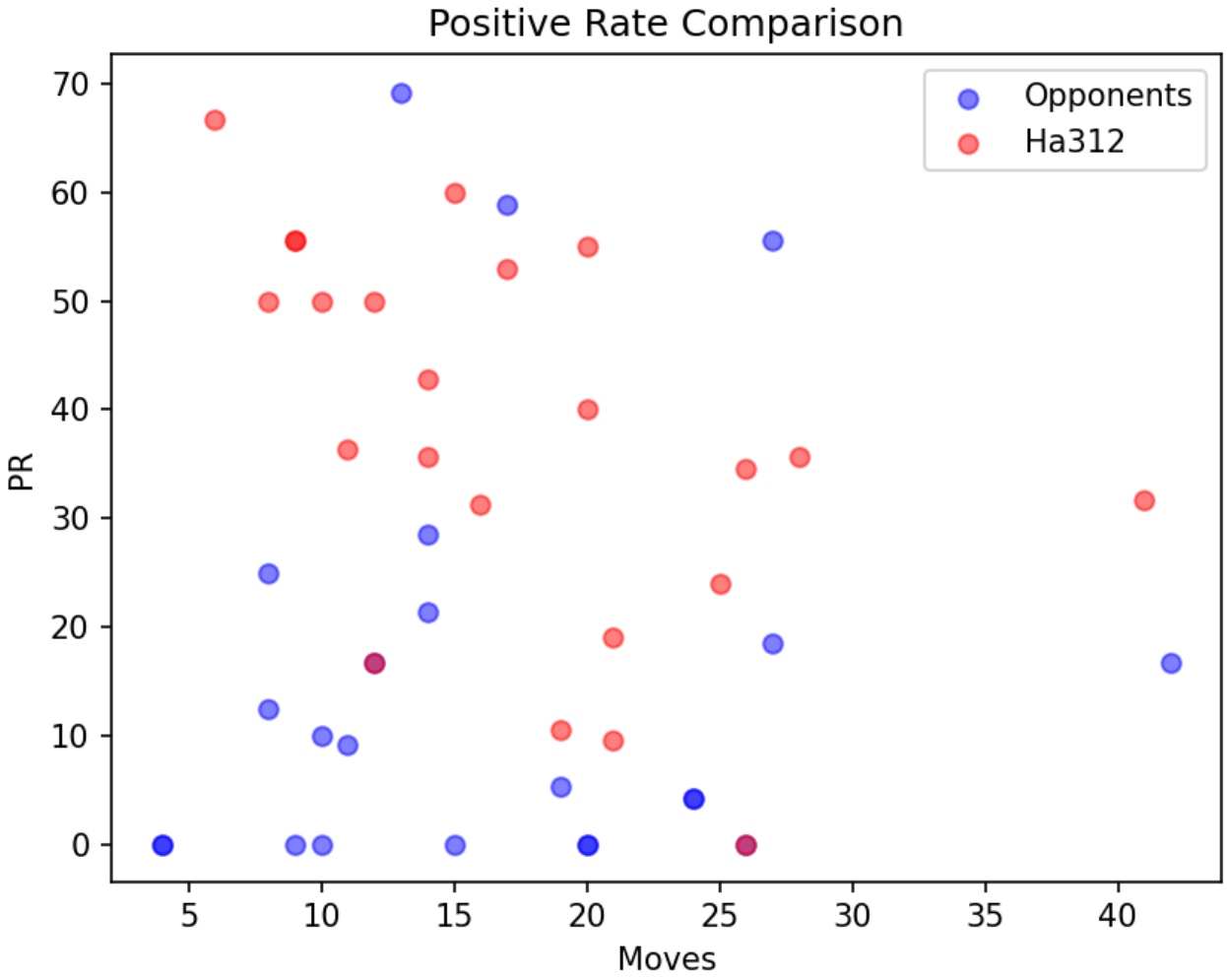


Figure 37: Positive rates plotted against number of moves analysed.

D Games From Vartender

Chess.com profile: `https://www.chess.com/member/vartender`
Link to games: `https://www.chess.com/games/archive/vartender`
Games which Vartender won are green, games which they lost are red, and drawn games are grey.

Opponent (Elo)	Total Moves	Vartender PR (%)	Opponent PR (%)
vogelj (2473)	81	22.2	22.2
CM_UlaneoM (2413)	67	37.5	26.7
BillieJean123 (2693)	86	28.1	0.0
Notime4bu (2016)	55	41.7	0.0
ElliotAldersonTwitch (2760)	102	24.1	20.0
ElliotAldersonTwitch (2769)	77	60.0	30.8
CarlosjPineror (2696)	90	17.1	23.1
alexrustemov (2931)	156	14.5	15.2
Mops_2004 (2709)	43	16.7	0.0
Edgar_Karagyozyan (2716)	88	27.6	0.0
mr_gustavo (2720)	75	50.0	0.0
fireemblem7 (2732)	83	31.6	21.4
Angry_Twin (2898)	82	18.9	17.1
Angry_Twin (2888)	97	36.8	36.8
Angry_Twin (2886)	192	2.4	6.0
EdmundDantes777 (2296)	96	29.6	42.9
fish_15 (2645)	59	63.6	62.5
fish_15 (2649)	142	17.2	10.2
JeRoma71 (2796)	89	0.0	3.1
JeRoma71 (2812)	63	36.0	54.2
stefanbusch (2253)	84	30.0	10.0
Evandro_Barbosa (2700)	84	22.2	18.9
Hoshimjon199 (2691)	106	45.5	28.1
bakifront (2726)	78	36.4	9.1
sumopork (2820)	78	45.5	65.0
superchthonic (2758)	66	56.2	33.3
Durarbayli (2929)	113	23.5	21.4
Durarbayli (2920)	73	20.6	18.2
snowlord (2981)	48	34.8	32.0
NewBornNow (2846)	66	18.2	24.2
baki83 (2911)	76	50.0	32.4
rychessmaster1 (2780)	48	16.7	35.3

Table 14: Games played by Vartender, with positive rates analysed.

Figure 38: Positive rates plotted against number of moves analysed.

E Games From Myself

Chess.com profile: `https://www.chess.com/member/haku28866`
Link to games: `https://www.chess.com/games/archive/haku28866`

Opponent (Elo)	Total Moves	Haku28866 PR (%)	Opponent PR (%)
Greeceacho (1779)	71	22.2	13.3
jhunnerio (1978)	70	7.1	40.0
jedjackoway (1780)	44	23.5	21.1
Brilliant90Gambit (1881)	110	0.0	0.0
procolarum (1886)	69	18.2	11.8
philkar2110 (1847)	73	40.0	41.7
Wazz1y (1857)	112	25.0	0.0
fitz451 (1875)	160	26.2	25.6
ypeeri (1773)	51	9.5	11.8
Nexur (1855)	136	22.7	27.3
AdamMcChess (1855)	72	20.0	14.3
SpaceOddity (1865)	55	30.8	75.0
darma12345 (1844)	74	25.0	4.5
ksrtiger (1890)	56	25.0	33.3
Doker1110 (1892)	107	34.3	8.8
klestaqerimi (1861)	97	24.0	4.0
Ljube77 (1847)	157	13.2	29.5
o1ram (1793)	58	11.5	11.1
alexmatiel (1818)	76	19.0	33.3
Aramsa (1781)	119	36.0	15.4
bojanles (1835)	159	18.4	2.6
Guindil (1778)	104	0.0	0.0
castuloromero (1838)	110	8.5	10.6
Rheged (1819)	86	0.0	75.0
alex100358 (1852)	63	33.3	23.5
tahsinkaanb (1862)	63	17.9	4.2
Toni20041992 (1830)	71	11.1	8.8
Kulimulimuh (1879)	53	16.7	12.0
fernandope1 (1818)	63	3.2	3.1
janek86 (1925)	76	21.7	50.0
furioussnail (1830)	103	17.6	5.3
GlobalResolution (1914)	91	30.8	36.0
Youll_NeverKnow (1901)	87	27.8	35.3
jbhakti (1936)	76	33.3	50.0
wanto84 (1977)	52	23.5	25.0
iBob_42 (1905)	53	13.0	26.1
JS5010 (1835)	161	7.5	11.7
kushkotjpr (1852)	116	6.8	13.6

BorceJovanovski (1888)	100	24.3	28.6
ligh567 (1852)	64	26.1	15.4
Mylly44 (1948)	79	21.6	20.5
Lordlemon333 (1834)	57	28.6	26.3

Table 15: Games played by Haku28866, with positive rates analysed.

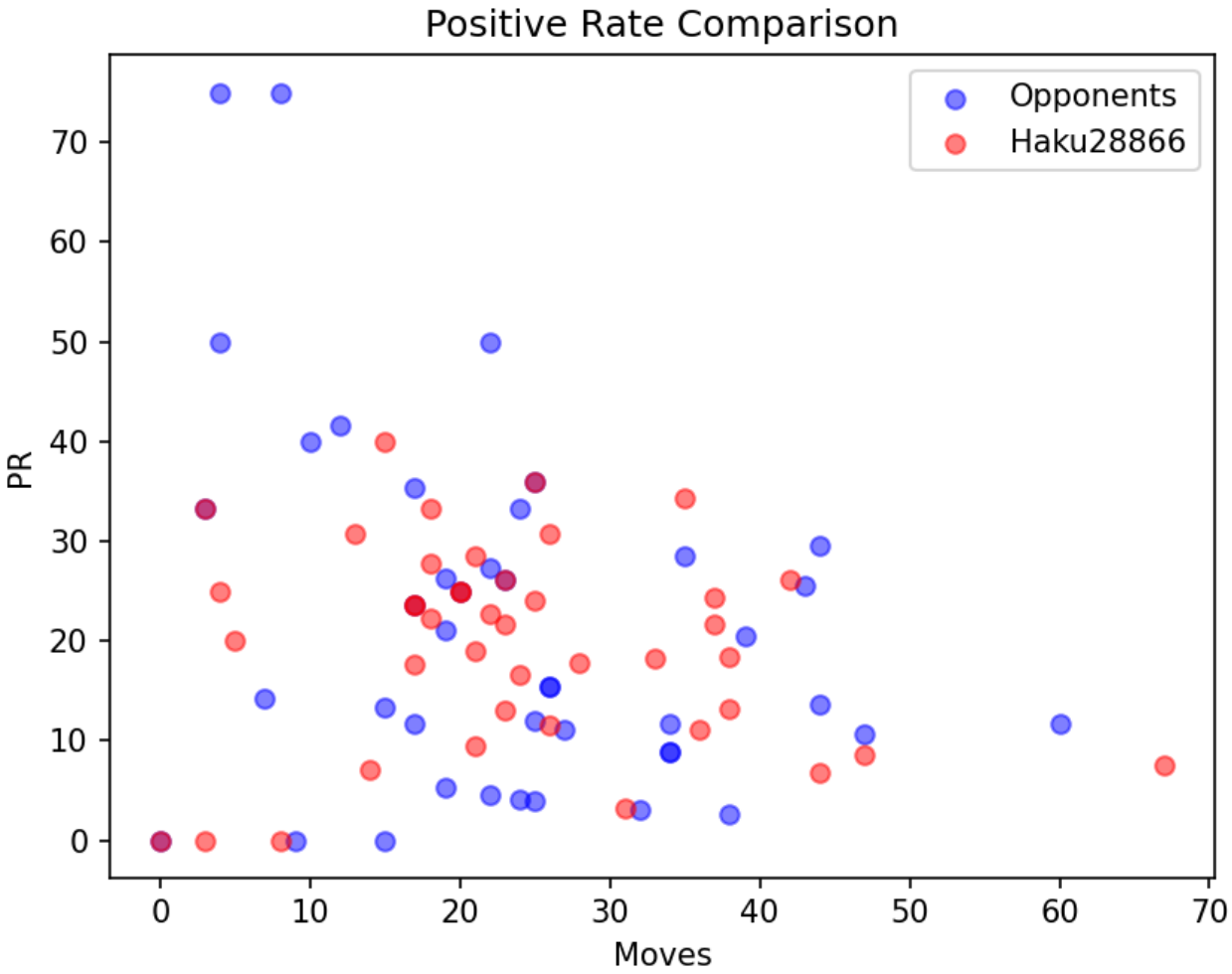


Figure 39: Positive rates plotted against number of moves analysed.