Naive Bayes Classifier in a Chess Engine

Final Project Report

Author: Mohammad Ibrahim Khan

Supervisor: Jeffery Raphael

Student ID: k22013981

April 2, 2025

Abstract

TODO: The abstract is a very brief summary of the report's contents. It should be about half-a-page long. Somebody unfamiliar with your project should have a good idea of what your work is about by reading the abstract alone.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Mohammad Ibrahim Khan

April 2, 2025

Contents

Chapter 1

Introduction

Since the creation of chess over 1500 years ago [1], it is widely acknowledged among researchers and players that the most influential moment in chess history was IBM's Deep Blue defeating Kasparov [2], the world champion at the time. This was a turning point for chess engines and AI in general. Since then we have seen the rise of chess engines the likes of Stockfish and AlphaZero. Despite our progress in this field, the game is still unsolved. Shannon (1950) mentions that there are 10^{120} possible positions in chess [3] which is more than the number of atoms in the observable universe. Therefore, brute-forcing the game remains infeasible. As a result, since 1997, attempts have been made towards creating a perfect chess engine. Achieving more efficient approaches to solving chess could provide meaningful insights in other problems in computer science like optimisation and decision-making.

Machine Learning is at the heart of modern chess engines like AlphaZero and Stockfish. The aim of this project is to create a chess engine that uses machine learning techniques that are not popular within the chess engine field, to explore the potential of these techniques but also to explore what processes can be shared between different machine learning techniques. Chess engines are usually used as a benchmark for advancements in AI, so this project could be a stepping stone for future research.

Minimax, especially when combined with Alpha-Beta pruning, is fundamental to modern chess engines. Alpha-Beta pruning optimises the algorithm by discarding unnecessary branches which decreases the computational demand considerably. Despite this, even optimised minimax algorithms cannot fully explore an entire tree due to exponential growth. This is where machine

learning is used to predict optimal moves more efficiently. Many techniques have been used from Neural Networks [4] to Natural Language Processing (NLP) [5].

This research focusses on how a Naive Bayes Classifier can be implemented in a search algorithm like Minimax to improve positional evaluation in a chess engine. Naive Bayes has proven effective in other contexts like spam detection and due to its simplicity and efficiency, it could be ideal for this application. Popular machine learning algorithms used in chess engines are usually very complex and require a lot of computational power, however Occam's Razor states that simplicity is usually the best option. This research examines whether this principle holds in this domain.

Traditional minimax algorithms implement simple evaluation functions, primarily based on material advantage. However, the complexity of chess involves intricacies that are not reflected in the number of pieces alone like piece positioning as well as structural weaknesses. This is where a Naive Bayes classifier could be utilised to try to learn these intricacies better than a standard evaluation function.

The primary objective of this research is to explore the potential of implementing a Naive Bayes Classifier into a minimax based chess engine to improve the evaluation. Specifically, this project investigates if a simpler and more efficient machine learning algorithm reduce complexity but achieve the performance of more complex machine learning techniques. This research will also explore a variety of implementations of the classifier, whether it should fully replace traditional evaluation functions or support them.

1.1 Report Structure

Chapter 2

Background

2.1 Chess

Modern chess is a game that has its origins in India, dating back to the 6th century as a way of devising strategy and tactics in war. Today, this game is perceived as a benchmark for skill and intelligence, played by millions. Given these roots in strategy, chess presents unique opportunities for AI research as it is a fully observable game, as both players can see everything related to the game state and the rules are well defined. Also there is no component of chance in the game, therefore the game state is only determined by each player's moves.

2.2 Search Algorithms

The defeat of Garry Kasparov against IBM's Deep Blue marked a major turning point, demonstrating the effectiveness search algorithms in chess and accelerating research into more sophisticated search algorithms and evaluation techniques. Search algorithms are important in chess engines as they systematically explore potential future moves, allowing the engine to determine the most optimal move. Commonly used is what is known as minimax. The concept of minimax was first proposed by Shannon in 1950 [3]. It is used for zero-sum games which are games where if one player wins, the other player loses. The algorithm recursively alternates between the maximising player and the minimising player, until it reaches a terminal node. Then the

algorithm backtracks to find the best moves for each player. The algorithm is shown below:

Algorithm 1 Minimax Algorithm

```
function MINIMAX(Node, Depth, MaximizingPlayer)
    if Depth = 0 or Node = Leaf then
        return EVAL(Node)
    end if
    if MaximizingPlayer then
        Value ← −∞
        for each in Node do
            Value ← max(value, MINIMAX(child, depth − 1, false))
        end for
        return Value
    else
        Value ← ∞
        for each Child in Node do
            Value ← min(value, MINIMAX(child, depth − 1, false))
        end for
        return Value
    end if
end function
```

The issue with this algorithm is that the search tree it creates grows exponentially, so techniques to decrease the search space are used. Alpha-Beta pruning is one such technique that is used to reduce the number of nodes that need to be evaluated. It does this by ignoring nodes that would not affect the final outcome of the algorithm. It introduces two new values, α and β, where α represents the maximum value that can be attained and β represents the minimum value that can be attained. If the value of a node is less than α or greater than β, then the node is pruned. In the best case scenario the algorithm only needs to evaluate $O(b^{m/2})$ nodes [6], where b is the branching factor and d is the depth of the tree compared to $O(b^m)$ nodes with normal minimax. However, in the worst case scenario it doesn't help improve minimax at all. Since alpha-beta pruning always returns the same value as minimax and is much faster, it is always used over minimax alone.

However, even with alpha-beta pruning, the search space for chess is still too large to evaluate in a reasonable amount of time. This is why a Heuristic Alpha-Beta Tree Search is used, which is where the search is cut off early and apply a heuristic evaluation function to estimate which player is in the winning position. In chess engines what is usually used for this evaluation function is calculating the material balance, where each piece is given a value and the player with the higher value is currently "winning".

The way to improve chess is one of two ways. The first is to increase the depth of the search tree, however this is generally dependent on the computational power of the machine so over

time as computational power increases (if Moore's Law still holds) the depth of the search tree can increase. The second way is to improve the evaluation function, which is what this research paper focusses on. The primary way this is done is by using machine learning techniques.

2.3 Naive Bayes

Modern chess engines, such as AlphaZero and Stockfish, generally use complex neural networks and reinforcement learning algorithms. These methods are very effective however they require substantial computing power, large training data and long training times. On the other hand, Naive Bayes, due to its simple algorithm, provides a promising alternative that is more efficient. Its ability to generalise well even with small datasets and predict efficiently, makes it favourable for applications where fast evaluation and decision making is required. This paper explores whether the simplicity and efficiency of Naive Bayes can achieve comparable effectiveness to other machine learning techniques or whether this simplicity will not allow it to understand the intricacies of the game.

Naive Bayes, sometimes also known as Idiot Bayes or Simple Bayes, is a simple classification algorithm that is based upon Bayes' theorem, which is a fundamental principle that describes how new evidence can change the probability of an event (Equation 2.1) .[7].

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{2.1}$$

Where:

- $P(A|B)$: Posterior Probability
- $P(B|A)$: Likelihood
- $P(A)$: Prior Probability
- $P(B)$: Evidence Probability

Posterior probability is the probability of event A occurring after knowing new evidence B. Likelihood is the probability of evidence B happening given that event A is true. Prior

probability is the probability of event A happening based on information we initially know before any new evidence. The prior probability of a class is given by the formula:

$$P(C) = \frac{N_c}{N} \tag{2.2}$$

Where:

- $P(C)$: Prior probability of class C
- N_c: Number of instances of class C
- N: Total number of instances

After calculating the priors for each class, the likelihoods of each feature need to be calculated. There are two main types of Naive Bayes classifiers, Multinomial and Gaussian. Multinomial Naive Bayes is primary used for situations where the features used are discrete which calculates the likelihood based on the count of each feature.

$$P(X|C) = \frac{N_{X,C} + 1}{N_C + V} \tag{2.3}$$

Where:

- $N_{X,C}$: Number of instances of the feature X in class C
- 1: Laplace smoothing constant
- N_C: Number of instances in class C
- V: Number of possible values for X

Where continuous features are used, like within this project, Gaussian Naive Bayes is more suitable. Gaussian Naive Bayes calculates the likelihood, assuming the features are normally distributed. This is done by applying the following formula.

$$P(X|C) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.4}$$

Where:

- x: Feature value
- σ: Standard deviation of the feature
- μ: Mean of the feature

The posterior probability for a class given a set of features $X_1, X_2, ..., X_n$ is calculated using Bayes' theorem. Since Naive Bayes assumes that the features are conditionally independent given the class, allows the equation to be simplified to the product of the likelihoods [7].

$$P(c|X) = \frac{P(c) \prod_{i=1}^{n} P(x_i|c)}{P(X)} \tag{2.5}$$

Where:

- $P(c|X)$: Posterior probability of class c given features X
- $P(c)$: Prior probability of class c
- $P(x_i|c)$: Likelihood of feature x_i given class c
- $P(X)$: Evidence probability

Naive Bayes classifies new instances by choosing the class with the highest posterior probability for that given feature. Since $P(X)$ is the same for each, it can be removed when comparing the posterior probabilities.

$$P(c|X) \propto P(c) \prod_{i=1}^{n} P(x_i|c) \tag{2.6}$$

So, the predicted class of an instance is chosen by the following formula:

$$\hat{y} = \arg\max_{c} P(c) \prod_{i=1}^{n} P(x_i|c) \tag{2.7}$$

For this project, since many features will be used, when multiplying probabilities, it can result in very small numbers, causing underflow errors. To prevent this, the logarithm of the

probabilities can be used, which allows the multiplications to be converted to additions.

$$\log(P(c|X)) \propto \log(P(c)) + \sum_{i=1}^{n} \log(P(x_i|c)) \tag{2.8}$$

A small constant ϵ is added to the likelihoods to prevent taking the logarithm of 0.

As mentioned, Naive Bayes makes an unrealistic assumption that features are conditionally independent. In practice, especially in chess, this assumption rarely hold since the game states involves complex interactions between positions and pieces. A pieces value on the board is highly dependent on its position and other pieces on the board. Due to this impractical assumption, the model's accuracy could be weakened.

Chapter 3

Literature Review

The application of machine learning in chess has seen significant progress in recent years. Modern machine learning implementation in chess is generally monopolised by neural networks or traditional methods like alpha-beta pruning. Naive Bayes has been demonstrated to be effective in text classification yet its application in chess remains unexplored. This literature review aims to explore the existing research on Naive Bayes and chess engines and whether this classifier can offer a unique perspective in this field. This research aims to contribute to the development of more diverse and efficient chess engines by exploring the feasibility of Naive Bayes as an alternative to other traditional methods.

3.1 Naive Bayes in other domains

There has been extensive research on Naive Bayes and where it can be applied. While many researchers have explored the use of Naive Bayes in different contexts, the success of Naive Bayes has varied between different domains.

The most popular use of Naive Bayes is in spam detection. In the paper 'An Evaluation of Naive Bayesian Anti-Spam Filtering' Androutsopoulos et al [8] evaluate the performance of Naive Bayes in spam filtering. They demonstrate that Naive Bayes performs surprisingly well in text classification tasks like spam filtering. Sahami et al. (1998) [9] showed that Naive Bayes was very successful in classifying spam detection. They conducted a number of experiments,

the first of which considered attributes as only word-attributes. The second experiment also considered 35 hand-crafted phrases such as "only $" and "FREE!". The third experiment considered non-textual features such as attachments and email domains (e.g. spam is rarely sent from .edu domains).

These findings highlight Naive Bayes' strength despite its conditional independence assumption. This justifies the use of Naive Bayes in contexts where the features are not necessarily independent, like this project where chess features are certainly very linked. What is worth highlighting, is the improved performance with additional features such that it was able to achieve 100% precision and 98.3% recall. This indicated the fact that feature selection impacts classification effectiveness, which is very relevant in chess. Androutsopoulos et al. [8] built upon this work to investigate the effect of attribute selection, training size and lemmatisation on the performance of the Naive Bayes model. Similarly to Sahami et al. [9] , they found that the model performed better when more features were used. This can translate well to chess since the selection of meaningful features may affect the quality of predictions. The paper also investigates the idea of how harmful it is to misclassify a legitimate email as spam compared to classifying a spam email as legitimate. Sahami et al. assumed that the cost of misclassifying a legitimate email as spam is as harmful as letting 999 spam emails through. in [8], the authors considered different contexts where this threshold could be different. This is also relevant to chess, where misclassifying a winning move could be more harmful than misclassifying a losing move however in certain contexts, the cost of both may be similar. The paper concludes that while Naive Bayes performs well in practise, it requires "safety nets" to be reliable in practise. In the context of emails, instead of deleting the email, the system could re-send it to a private email address. In the context of chess, the probabilistic classifier could help prioritise move evaluations and minimax is used to ensure strategic accuracy of decisions.

While Androutsopoulos et al.'s and Sahami et al.'s studies show Naive Bayes's strengths in textual domains, it is important to note the difference in domain. Chess, unlike textual data, is inherently highly interdependent

Naive Bayes has been shown to be very effective in text-based domains including spam detection as discussed before as well as in anti-cyber bullying systems [10]. Despite these proven strengths in textual domains, it is important to note the difference in domain. Chess, unlike textual data, is inherently highly interdependent. There has been some researches where it has been seen to fail compared to other classifiers. Hassan, Khan and Shah (2018) [11]

investigated a variety of classification algorithms in classifying Heart Disease and Hepatisis. The algorithms they evaluated were Logistic Regression, Decision Trees, Naive Bayes, K-Nearest Neighbours, Support Vector Machines and Random Forests. Their findings consistently showed that Naive Bayes was the worst performing algorithm in both cases. For Heart Disease, Naive Bayes achieved an accuracy of 50% whereas Random Forests achieved an accuracy of 83%. For Hepatisis, Naive Bayes achieved an accuracy of 68% compared to Random Forests which achieved an accuracy of 85%. This result contrasts what was found in the previous research on spam detection. These findings reinforce the idea that Naive Bayes is not a one-size-fits-all algorithm and it can excel in certain domains but its effectiveness is not guaranteed in all applications.

3.2 Machine Learning in Chess

The first machine that is mentioned in the history books that played chess against humans was the Turk [12]. This 1770s mechanical automaton was able to not only play chess but was able to beat human opponents. It was then later revealed that this machine was actually operated by a human making the moves.

The biggest milestone to date in the world of chess is undoubtedly Deep Blue vs Kasparov in 1997. Deep Blue was a chess engine created by IBM and in 1997, it was able to beat the reigning world champion Garry Kasparov. The first match between Kasparov and Deep Blue was in 1996 where Kasparov won. After this defeat, IBM hired grandmaster Joel Benjamin to improve the evaluation function of the engine. The rematch in 1997 then surprised the world where Deep Blue was able to beat Kasparov 3.5-2.5. This story emphasises the importance of the evaluation function in chess engines and that the just pure processing power is not enough.

Stockfish is a free, open-source chess engine that is widely regarded as one of the strongest chess engines in the world. It uses alpha-beta pruning and a variety of other techniques to evaluate positions.

In 2017, Google's DeepMind released AlphaZero, a chess engine that was able to beat Stockfish. What is notable in this chess engine is it's reliance on machine learning techniques. AlphaZero uses a Monte Carlo Tree Search algorithm combined with a deep neural network. Unlike Stockfish, it learns from self play, where it plays games against itself and learns from

its mistakes [4]. This was the most prominent engine that used machine learning techniques to play chess. One benefit of this technique is that it examines fewer positions than Stockfish but spends more time on evaluating each one. This mimics human-like pattern recognition by prioritising positional understanding over brute force. The core of AlphaZero is a deep neural network that takes as input the board state and outputs the probability of winning and it is trained using reinforcement learning. AlphaZero's design is general and can be adaptable to other two-player, determinstic games. It has been successfully applied to Shogi and also Go with AlphaGo, which relies upon five neural networks [4].

AlphaZero represented a paradigm shift in the world of chess engines. It showed that machine learning techniques can outperform traditional methods like alpha-beta pruning. Even stockfish, which is known for its brute force approach, has integrated efficient neural networks (NNUE) with its traditional search methods. An important note is that these well-known chess engines rely mainly on neural networks. This project aims to investigate if other techinques can yield the same result, specifically Naive Bayes.

3.3 Naive Bayes in Chess

It has been proven that machine learning has been highly effective in chess, like AlphaZero, therfore it is necessary to assess whether simpler classifiers could offer a viable alternative. Research on applying Naive Bayes specifically to chess engines is very limited, creating an important research gap. Given the inherent complexity and interdependence of chess features, exploring this gap critically evaluates whether Naive Bayes' simplicity and efficiency can compensate for its strong assumption of feature independence. One of the most relevant research on this topic is by DeCredico (2024) [13]. In this recent paper, DeCredico explores the use of a number of machine learning algorithms to predict the outcome of a chess game using player data from 'The Week In Chess' database. This work focused on utilising player statistics like win rate and rating as features to classify the outcome of a game. The algorithms used included, Naive Bayes, Decision Trees and Random Forests. DeCredico highlights the importance of feature selection in the performance of the algorithm, comparing different combinations of features. However, DeCredico focused on pre-game, player-centric statistics whereas this project explores the use of in-game positional features with Naive Bayes.

The norm for chess engines is usually complex neural networks but DeCredio's approach

is much simpler. Another benefit of this approach is that it is more interpretable. Neural Networks are usually considered black boxes [REFERENCE???] within literature so it is hard to understand why it makes certain decisions and to extract insights that can be applied by human players. Naive Bayes, on the other hand, is much easier to understand which features are important in making a decision. The results of DeCredico's work showed that Naive Bayes achieved a maximum accuracy of 63%. This is not very high but it is a promising result and shows that there is some potential in using Naive Bayes in chess engines.

The research of Naive Bayes and chess engines separately is well documented but there is a gap in the literature where the two are combined. This project aims to fill this gap by exploring the viability of Naive Bayes to improve the performance of chess engines.

Chapter 4

Methodology

4.1 Introduction

The focus of this project is to explore the uses of Naive Bayes in chess and whether it is a viable alternative to current techniques. This chapter will mention the methodology that was used to implement the Naive Bayes classifier in the chess engine. For this project, the python-chess library was highly relied upon. Many different python scripts were used during the project. The main scripts included were `data_prep.py` where the data was preprocessed, `training.py` where the model was trained and evaluated, `features.py` where the features were calculated, `game.py` where the game was played and the most important ones `minimax_NB_XXX.py` where the Naive Bayes Classifier was applied to the minimax algorithm.

4.2 Integration of Naive Bayes with Minimax

4.2.1 Data Preparation

The dataset used for this project was obtained from Kaggle [14]. The dataset contained over 6.2 million chess games that were played on lichess.com in July 2016. The dataset was in CSV format, making it easy to extract information and analyse. The dataset contained many features, however the only features relevant to this project was the result of the game and the

sequence of moves in Algebraic Notation form.

This project wants to explore the how the classification of chess positions into wins and loses can be used to improve the minimax algorithm. For this reason, all games which resulted in a draw were not considered as well as games where one of the players resigned The data was split into 3 different groups to explore how player expetise affects the model's learning and generalisation ability. The first group, named `master`, was all games where one of the players had an Elo rating of 2200 and above as defined by the Federation International des Echecs (FIDE). The second group, named `beginner`, were games were both players had an Elo lower than 2200. The last group, named `random` were games where players with any Elo were considered. The dataset only had 300,000 instances of master games, therefore only 300,000 instances of beginner and random games were used. This is to ensure that the experiments are fair when comparing the effect of group on the model's performance.

The moves in Algebraic Notation would not provide the Naive Bayes classifier with enough information to classify as it would not provide context of the board state. For this reason, the python-chess library was used to simulate the games. The moves would be extracted from the csv file, then each move would be played on the board. The csv file would be read by using the pandas library and every 6 moves, the features of the board would be extracted and stored, this was to reduce the amount of data to train on as consecutive moves are highly correlated so would not provide more insight for the model. However, end game moves have a bigger affect on the result of the game so in this phase, every other move was used. The last move was also included since this is the move and game state that determined the result of the game. This was also when the results of the games were converted from `1-0` or `0-1` to a binary classification where 1 represented a win and 0 represented a loss. This was done to make the model easier to train and evaluate.

4.2.2 Feature Selection

Feature selection is a crucial part of how well the Naive Bayes will perform and generalise. Limited use of features can lead to underfitting and too many features can lead to overfitting. This project also wants to explore the impact different features can have on the model's performance. There were 4 feature sets used in this project.

The first feature set, considered 4 features: material balance, piece mobility, king attack

balance and positional value. Material balance is a very simple feature that considers the difference in number of pieces between the two players, where a positive value indicates a piece advantage for white and a negative value indicates a piece advantage for black. However, each piece is not of equal value in the game, for example a queen has much more power than 2 pawns have. For this reason, the values in Table 4.1 were used [3].

Piece	**Value**
Pawn	100
Knight	300
Bishop	300
Rook	500
Queen	900
King	0

Table 4.1: Values of Chess Pieces

Piece mobility is the difference in number of legal moves between the two players. The more moves available to a player suggests that they have more control over the board which could give them a tactical advantage. King attack balance is the difference in number of pieces attacking the king. Most of these features are simple and don't consider the game state as whole, like the position of pieces on the board. Positional value was a feature used where the location of a particular piece on the board can affect how effective it is. An example of a positional value table for a knight is shown in Table 4.2.

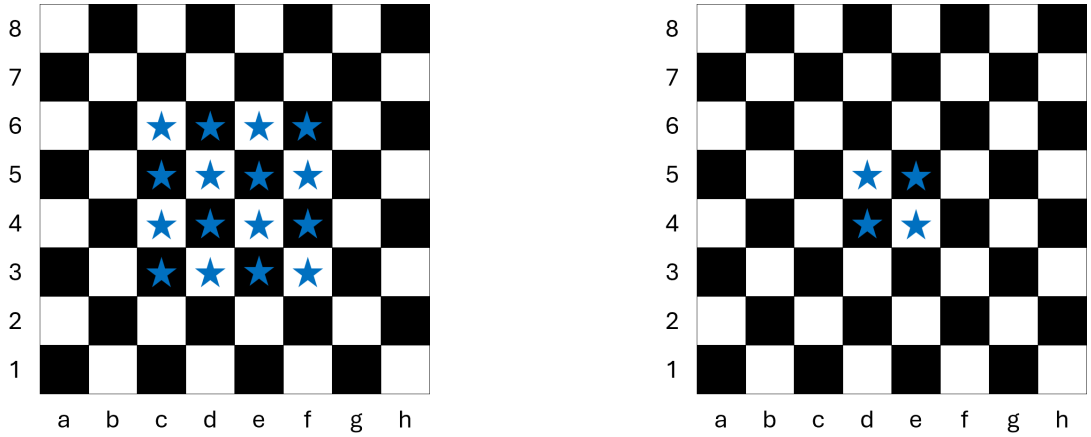
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
8	-50	-40	-30	-30	-30	-30	-40	-50
7	-40	-20	0	5	5	0	-20	-40
6	-30	0	10	15	15	10	0	-30
5	-30	5	15	20	20	15	5	-30
4	-30	0	15	20	20	15	0	-30
3	-30	5	10	15	15	10	5	-30
2	-40	-20	0	5	5	0	-20	-40
1	-50	-40	-30	-30	-30	-30	-40	-50

Table 4.2: Positional Value Table for Knight

This table favours the knight to be in the centre of the board rather than the edge. This is because when the knight is is in the centre, it can control more squares so has more opportunity to attack and defend, whereas when it is near the edge, the knight is more restricted, especially the corners where it only has 2 possible moves.

The second feature set considered the same features as the first set but also included the control of the centre. This is defined by the number of pieces within the centre. Again, it calculates the difference between the white and black pieces in the middle. This feature was

implemented as two separate features, one for the 2x2 square in the middle and one for the 4x4 square in the middle as shown in Figure 4.1.



(a) Big centre of the Board

(b) Small centre of the Board

Figure 4.1: Comparison of the small and big centres of the board.

The third feature set explored in this research considers all of the features mentioned previously but also more complex features, specifically pawn structure. The structure of pawns can determine how much control the player has, defending its pieces and preventing advancements from the enemy. Two structures were used for this project, isolated pawns and doubled pawns. Isolated pawns are pawns that do not have any friendly pawns on adjacent files. Usually this is considered as a weak structure since they can't be defended by other pawns and also can be easily blocked by the opponent pieces. However, some times it could be a powerful structure as it can have more control over the board and also some openings use isolated pawns in order to allow more movement for rooks and bishops. Doubled pawns are pawns that are on the same file. Generally this is a weak structure since they are limiting each other's mobility and can generally become isolated. However creating doubled pawns can be used to open up files or diagonals for rooks and bishops.

The last feature set used included all the features in the third feature set but also more complex features. This included the castling rights of the player, king safety and game phase. Castling is a move that allows the king to move two squares towards a rook. This is a very strong move as it allows the king to move away from the centre where it is generally more dangerous. It also allows the rook to have a more active role in the game. Therefore being able to retain the ability to perform this act can influence the game majorly. For the purpose of this project, the castling rights were considered for both kingside and queenside. King attack

balance is a very simple feature which only considers the number of pieces attacking the king. For this feature set a more complex attribute was used, king safety. King safety calculates the number of pawns in adjacent squares, which is known as pawn shield, and also calculates the number of pieces attacking adjacent squares to the king then returns the difference between the two. The last feature included in this feature set was game phase. This would output one of 3 values, opening, middle game or end game. This was calculated by giving values to each type of piece and summing up all the pieces on the board. Then the percentage of pieces left in the board was calculated. If it was more than 66% it returned 0 for opening, if it was between 33% and 66% it returned 1 for middle game and if it was less than 33% it returned 2 for end game.

4.2.3 Naive Bayes Classifier

There are many resources available to implement the Naive Bayes Classifier, the most commonly used is the one provided by the scikit-learn library. For this project, having complete control and understanding of the model was important, therefore the Naive Bayes Classifier was implemented from scratch. This allowed for more flexibility in the implementation and allowed for more experimentation with the model.

The main steps of the Naive Bayes Classifier are as follows:

1. Calculate the prior probabilities of each class.

2. Calculate the likelihood of each feature given the class.

3. Calculate the posterior probability for a class given a set of features using Bayes' theorem.

4. Predict the class of a new instance by choosing the class with the highest posterior probability for that feature.

The prior probabilities were calculated by counting the number of instances in each class then dividing it by the total number of instances. The numpy library was used to make this process more efficient. Then since for this project, continuous attributes were used, the Gaussian implementation of Naive Bayes was used, therefore after calculating the prior probabilities, the mean and standard deviation of each feature was calculated for each class, again aided by the numpy library. The likelihood was then calculated for each using the Gaussian formula as given in Equation 4.1.

$$P(X|C) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.1}$$

Then these likelihoods were used to calculate the posterior probabilities for each class given the set of features, which is done by using Bayes' theorem. Due to the assumption of conditional independence, the posterior probability can be simplified to the product of the prior probability and the likelihood of each feature given the class. Once the posterior probabilities were calculated, the class with the highest posterior probability was chosen as the predicted class. Two functions were implemented, `predict` and `predict_prob`. `predict_prob` returns the posterior probabilities for each class given a set of features which can be used to compare the confidence of the classifiers predictions. The `predict` function returns the class which the model classified the instance with, ie. the class with the highest posterior probability

Naive Bayes consists of the multiplication of multiple probabilities, which can lead to very small numbers causing underflow issues. To overcome this, the logarithm of the probabilities was used to predict the class. This then caused rise to another issue, the fact that $log(0)$. Due to the nature of the classifier, this could possibly occur. To prevent this from occurring, a small constant was added to the probabilities before taking the logarithm. This constant was set to 0.1. This was a small enough constant to not affect the results of the model but also large enough to prevent underflow issues.

The features that were extracted from the data was then used to train the model. The data was split into 80% for training and 20% for testing. This ratio is ideal as it provides enough data allow the model to learn and generalise well while enough to still test it's effectiveness. The features outlined in the previous section can all be in different scales which could give more importance to some features over others. Before feeding the data into the model, the data was standardised by using the StandardScaler from the scikit-learn library. This ensured that all features were on the same scale so the model would not be biased towards any feature. After training the models they were saved using the joblib library which allowed the use of the model without needing to retraining the model every time. Since there are 3 different groups of data and 4 feature sets, a total of 12 models were trained.

4.2.4 Model Evaluation

After training the model, it is important to know how well the model performs and how well it generalises to new data. Evaluating models also allows comparison of the findings with other findings in the literature. There are many ways to evaluate a classifier and for this project we will calculate a number of different metrics to gain a holistic view of the model's performance. The first metric calculated was the accuracy of the model. This is the most straightforward measure of the model's overall correctness, by providing the proportion of predictions that were correct 4.2.

$$\text{Accuracy} = \frac{\text{Correctly Classified Instances}}{\text{Total Instances}} \tag{4.2}$$

Where correctly classified instances is the sum of True Positives and True Negatives. The next two metrics calculated for the model were precision and recall. Precision is the proportion of true positive predictions to the total number of positive predictions made by the model. This is an important metric to consider because it provides insight into how many of the positive predictions made by the model were actually correct. Recall is the proportion of true positive predictions to the total number of actual positive instances in the dataset. This metric is important because it provides insight into how many of the actual positive instances were correctly predicted by the model. The equations for precision and recall are given in Equations 4.3 and 4.4 respectively. These two metrics are usually related as there is a trade off between the two, generally increasing one causes the other to decrease.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.3}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.4}$$

Usually it is more convenient to compare models using a single metric, which takes into account both precision and recall. F_β score is the weighted harmonic mean of precision and recall, providing a metric that balances the two. The β parameter allows control over the trade-off between precision and recall. A β value of 1 gives equal importance to precision and

Kappa Value	Agreement level
< 0	Poor agreement
0.01 - 0.20	Slight agreement
0.21-0.40	Fair agreement
0.41-0.60	Moderate agreement
0.61-0.80	Substantial agreement
0.81-1.00	Almost perfect agreement

Table 4.3: Interpretation of Kappa Statistic

recall which is what will be used for this project.

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{4.5}$$

The last metric used is the kappa statistic. This is a measure of how well the model performs compared to a random classifier. The benefit of this metric is that it takes into account the possibility of the model being correct by chance. The equation for the kappa statistic is given in Equation 4.6 .

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{4.6}$$

Where p_o is the observed accuracy of the model and p_e is the expected accuracy of the model. The expected accuracy is calculated by multiplying the proportion of instances in each class by the proportion of instances in each class. The kappa statistic is commonly understood by the categorisation in Table 4.3 [15].

4.2.5 MMNB Algorithm

The MMNB algorithm is a combination of the Naive Bayes Classifier and the traditional minimax algorithm. There were two implementations used for this project, one where the Naive Bayes completely replaced the evaluation function of the minimax algorithm and one where the Naive Bayes was used to improve the evaluation function by using it in conjunction with a traditional evaluation function.

The first implementation was built in the `minimax_NB_sub.py`. The benefit of the Naive Bayes classifier over other classifiers is that it can provide how confident the model is in its

predictions in the form of probabilities. In this version of MMNB, a standard minimax algorithm with alpha-beta pruning was used. Due to limitations in computational power and time, a depth of 3 was used. This allowed enough exploration of the game tree that it can be an informed decision but also to do this in a reasonable time period. When the maximum depth is reached or it met a terminal node (ie. the game has terminated with a win, lose or draw) then instead of calling a traditional evaluation function, an evaluation function implementing Naive Bayes was used.

In the revised evaluation function, the model and scaler were loaded from the joblib files. The board state is passed to the features function to extract the current features of the board. The features are then scaled using the loaded scaler. These features are then fed to the predict_prob function of the Naive Bayes Classifier. This function would then return the posterior probabilities for both classes, win and loss. These probabilities are then used to calculate the value of the node. The value of the node is calculated by taking the difference between $P(Winning|X)$ and $P(Losing|X)$. The value is positive when the classifier thinks white is at an advantage and negative when it thinks black is at an advantage. Terminal nodes also need to be considered, so if the board state is in a checkmate position, the naive bayes evaluation would be disregarded and a value of $\pm\infty$ would be returned dependant on the player who has won. The value of this evaluation function is then returned to the minimax algorithm where it continues to search the rest of the game tree.

The second implementation took a more traditional approach to the minimax algorithm. The Naive Bayes was not solely used but rather a combination of both was used to make a more informed evaluation score. This was implemented in the `minimax_NB_integrated.py` file. The minimax algorithm was implemented in the same way as the previous implementation, but when the maximum depth was reached or a terminal node was reached, a different version of the evaluation function will be used. In this function two factors were considered. The first was the Naive Bayes score, this was wen through the same process as the previous implementation. The second used was a more traditional evaluation which considered material balance and positional value. The material balance was calculated using the same values as used during the feature extraction for the Naive Bayes model 4.1. The positional values were also calculated similar to what was used for the feature extraction 4.2. These two values were summed up and used as the traditional score. The traditional score and Naive Bayes score will be at very different scales, which would cause the traditional score to dominate the Naive Bayes score. To prevent

this, the traditional score was normalised to be between 0 and 1. This was done by taking the maximum and minimum values of the traditional score and scaling it to be between 0 and 1. Due to the fact that the logarithm of the probabilities were used to calculate the Naive Bayes score, the Naive Bayes score was also scaled to be between 0 and 1. This was done by applying the softmax function to the Naive Bayes score, given by the equation 4.7.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{4.7}$$

The two scores were then combined by taking the weighted sum of the two scores. The default weights were set to 0.5 for both, however, during experiements it will be explored how the weights affect the performance of the minimax algorithm. The final score is then returned to the minimax algorithm where it continues to search the rest of the game tree.

4.3 Experiments

The main purpose of this project is to investigate the usefulness of applying Naive Bayes to minimax to add its evaluation. For this reason, the main evaluation is for the substitution and integration methods of MMNB. For this we could just rely on win rate against stockfish. However this may not give much insight to how well the algorithm is really doing. Many other metrics were used to assess the performance and effectiveness of each MMNB algorithm.

One such metric is `nodes_explored`. This is the total number of nodes explored in the game tree for a specific move to be made. For this, the minimax algorithm had to be slightly adjusted to keep track of and return the number of nodes explored. This will give a good indication on how well the alpha-beta pruning is working. Another measurement included is the time it takes for each move to be calculated. This is a good indicator of the efficiency of the algorithms.

The previously mentioned features don't consider how well the algorithm is actually playing. For this 2 other metrics were considered, piece balance and mobility. Piece balance is the number of pieces white has over black, generally this shows who is currently winning. Mobility is the total number of legal moves, again this shows how much control of the board the player has, which is generally advantageous. For each move, the game state is analysed by stockfish to give a score of the board, where a positive value means white is most likely going to win and a

negative value indicates black is more likely to win. These evaluations were also used to count blunders and good moves. Blunders are moves that cause the evaluation of stockfish to change negatively by 300 and good moves are moves that cause the evaluation of stockfish to change positively by 200.

To better test the performance of each algorithm, 2 opponents were used. The first being a random engine, this engine randomly picks legal moves with no incentive to win. The second opponent is stockfish, which was set at level 0. This level of stockfish isn't a grandmaster but understands how to win and can make decisions that favour it. For every game played, the games will be played against both opponents.

The impact of the dataset and feature sets used was also investigated. Games played with both MMNB algorithms also used the different models initially trained. To remove any chance of random chance and luck, a total of 30 games were played for each configuration. For integration, the weight of Naive Bayes and a traditional evaluation will can be adjusted. For the purpose of this project, it was investigated how the change of this weighting can affect its performance by using Naive Bayes weightings of 0.25, 0.5 and 0.75. Considering this the total number of games played will be:

Substitution: 3(datasets) x 4(features) x 2(opponents) x 30(games) = 720

Integration: 3(datasets) x 4(features) x 3(weightings) x 2(opponents) x 30(games) = 2160

Totalling a total of 2880 games to be played. This will give us a good overview of the performance of the algorithms and enough data to be able to have a strong understanding of how the different models affect the ability of the classifier to be able to classify.

Chapter 5

Legal, Social, Ethical and Professional Issues

Your report should include a chapter with a reasoned discussion about legal, social ethical and professional issues within the context of your project problem. You should also demonstrate that you are aware of the regulations governing your project area and the Code of Conduct & Code of Good Practice issued by the British Computer Society, and that you have applied their principles, where appropriate, as you carried out your project.

5.1 Legal Issues

As mentioned before, for this project the python-chess library was primarily used for the implementation of the chess engine. This library is licensed under the GNU General public License v3.0 (GPLv3) [16]. This license is a free software license that allows developers to freely use, study and modify the python-chess library for their projects [17]. The license also requires that any modifications made to the library must be distributed under GPLv3, meaning that the source code mush be available to the public which is fulfilled as the source code is publicly available on GitHub [18].

This research utlises the 'Chess' dataset available on Kaggle [14] uploaded by Mitchell J. The dataset is open to be used by the public under the Creative Commons CCO 1.0 Universal

license, meaning that this dataset can be freely copied, modified, distributed and used for any purpose without requiring permission from the creator. Despite not legally required, we would like to acknowledge the contribution the creator has had on this research project and others. This data does not contain any directly identifiable information from Lichess users. Player usernames were included in the dataset but these do not directly reveal real-world identities. The data does not include sensitive personal data like real names, email addresses or phone numbers.

5.2 Social Issues

The chess engine developed in this project is a tool that can be used to help players improve their chess skills, however it has been engineered for someone who has some technical ability. Understanding python and basic command-line usage is required to run the engine. Also the output of the engine is standard algebraic notation which most chess players are familiar with, but players can not gain an understanding of why the engine made a particular move. A GUI was implemented to help users visualise the board and the moves made by the engine. This GUI would be more beneficial paired with a more readable explanation of the engine's moves. In the future, the engine could be more accessible to a wider audience by implementing features like audio outputs for visually impaired users or support for other languages. Another feature that could be beneficial is an Open API that would allow developers to integrate the engine into their own applications, potentially leading to more innovative ways to use the engine and more research opportunities.

The advancements and increased accessibility of machine learning-based chess engines could have a major implications on the chess community. More powerful chess engines being very available could cause a reduction in demand for human chess coaches. These engines could provide personalised training, analyse moves and provide feedback to players, much better than a human coach may be able to do. This could lead to a decrease of people playing chess especially at the professional level. However this is very unlikely to replace human coaches but rather the increase in availability of chess engines could have a positive impact since it could allow those who may not have the resources to have a coach, lowering the barrier to entry for the game. It can be used as an educational tool for players, generating training exercises, analyse games and explain concepts.

5.3 Ethical Issues

An ethical advantage of using Naive Bayes over other machine learning techniques is its transparency and interpretability. Unlike models that are considered 'black-boxes' like Neural Networks, Naive Byes allows users to understand the reasoning behind the model's predictions. Users are more likely to trust the model id they can understand the engine's thinking process. A Naive Bayes chess engine wouldn't necessarily harm a person's life, it is the responsibility of developers to consider the ethical implications it could have. One main risk is potential misuse of the engine, primarily in online gaming or competitions. For this reason, we encourage users to use the engine to use this tool for learning and analysis and strongly discourage any form of cheating and encourage fair play.

5.4 Professional Issues

This project was inline with the principles as mention in the Code of Conduct & Code of Good Practice issued by the British Computer Society. I, Mohammad Ibrahim Khan, applied my knowledge and skills to the best of my ability and worked within my areas of competence and sought external guidance from my supervisor, Jeffery Raphael, where necessary. All data, results and conclusions presented in this report are accurate and truthful to the best of my knowledge. The intellectual property rights of others have been respected throughout, properly citing all external datasets, libraries and references. As discussed in previous sections, measures were taken to ensure privacy of individuals. Usernames were used as pseudonyms and no attempt was made to identify individuals.

A number of measures were taken to ensure the transparency and ease of use of the chess engine. A detailed explanation of how the Naive Bayes classifier evaluates chess positions was provided in this report including the features used and the training process. The limitations of the engine such as potential bias and inability to understand complex situations have been clearly communicated and potential ethical issues related to the engine were also discussed.

Chapter 6

Results

6.1 Naive Bayes Evaluation

Before evaluating the application of the Naive Bayes classifier in the chess engine, it is important to evaluate the classifier on its own. A total of 12 models were trained, including 3 different datasets and 4 different feature sets (labelled 0 to 3). Each model was trained on 300,000 games from the Lichess database. The results are summarised in Table 6.1.

Dataset	Model	F1 Score	Kappa Score	Accuracy	Recall	Precision
Master	0	0.5989	0.2112	0.6049	0.4821	0.6439
Beginner	0	0.6051	0.2210	0.6095	0.5041	0.6467
Random	0	0.6072	0.2237	0.6107	0.5156	0.6460
Master	1	0.6051	0.2139	0.6066	0.5447	0.6258
Beginner	1	0.6100	0.2229	0.6110	0.5600	0.6303
Random	1	0.6113	0.2249	0.6120	0.5694	0.6304
Master	2	0.6085	0.2182	0.6089	0.5763	0.6203
Beginner	2	0.6112	0.2242	0.6118	0.5738	0.6275
Random	2	0.6131	0.2269	0.6132	0.5909	0.6261
Master	3	0.6085	0.2173	0.6086	0.5947	0.6152
Beginner	3	0.6116	0.2232	0.6115	0.6073	0.6185
Random	3	0.6125	0.2248	0.6125	0.6163	0.6187

Table 6.1: Naive Bayes Evaluation

The average F1 value overall was 0.608 across the 12 different models. This shows that the classifier was able to learn some aspects of the game, however despite this, considering a random classifier would have an F1 score of 0.5, an F1 score of 0.608 is not very good. This is further supported by the Kappa score which averaged 0.221. This would come under the category of "fair agreement" according to Landis and Koch [15], demonstrating that the classifier was able to understand some indicators of winning or losing positions. However the low Kappa and F1 scores reinforce the known limitations of Naive Bayes, particularly its reliance in the assumption of conditional independence. The models were not able to learn the complex relationships between the features and the result of the game, which is crucial in chess. This indicates that Naive Bayes is not suitable for applications where the results are highly dependent on feature interactions. An average accuracy of 0.610 highlights the model's ability to learn some patterns in the data. Precision is important in the context of chess since it reflects how often the model's predicted winning outcomes are correct. A low precision would mean the model may not consider safer moves, being led to make riskier decisions. A precision of 0.620 was obtained indicating the model's ability to identify winning positions. Recall is another important measure to consider in the context of chess. It reflects the the proportion of actual winning positions that were correctly identified by the model. A low recall could cause the

engine to miss critical chances to press an advantage. The average recall was 0.561, indicating the model's poor ability to identify a good number of winning positions.

Another important aspect that is important to analyse is the influence of the feature selection on the performance of the model. Across the 12 models, the progression from feature set 0 and feature set 3 showed an increase in both F1 and recall. The average F1 score for feature set 0 was 0.604 increasing to 0.622 with feature set 3, reflecting a clear trend that more features yield better performance. This is consistent with the literature, like mentioned by Sahami et al. [9] where the addition of hand-crafted features led to significant improvement, and Rish on his analysis on Naive Bayes [19]. The average of recall from 0.500 to 0.618 also indicates the model's ability to identify winning positions improved with the addition of more features. This is an important result as it suggests that reliance on simplistic indicators like material balance and mobility alone is not sufficient to capture the complexity of chess. Another notable observation is the minimal performance difference between feature sets 2 and 3. Both feature sets obtained a F1 score of 0.612. This plateau indicates possible diminishing returns with the addition of more features. One possible explanation for this is that the added features, like king safety and castling rights, may be strongly correlated to existing features, leading to minimal information gain. Another explanation to this is that the model did not see enough examples of these features in play in the training data to learn their significance. These findings further reinforce that the performance of the Naive Bayes Classifier is not only dependent on the quantity of features but also on the quality of the features. The results also indicate that Naive Bayes may be limited in this context as the performance seems to reach a limit despite the increase in features and feature complexity, where more complex models may succeed. The small increase in feature set 2 and 3 could also imply that most of the useful information that the model uses for its predictions is from previous features like control of centre and pawn structure, this observation would not have been possible with a machine learning technique that is not interpretable like a neural network.

In addition to feature selection, the dataset used to train the model has a noticeable impact on the predictive power of the classifier. What was interesting from this data, is that across all feature sets, the models trained on the random dataset consistently outperformed the models trained on the beginner and master datasets. This trend was shown by the F1 score and accuracy where, the random dataset obtained the highest f1 and accuracy scores, with an average of 0.611 and 0.612 respectively. Whereas the master dataset obtained the lowest f1

and accuracy scores, with an average 0.605 and 0.607 respectively. This result is surprising as what would be expected is that the master dataset would be more informative of good moves and winning positions. There are a number of reasons this result could have occurred. These results could be due to the nature of games in each dataset. In master-level games, there tend to be more complex positions and strategies, including more subtle positional considerations and more long term sacrificing and planning, which are difficult to learn in a probabilistic model like Naive Bayes. These complex reasonings for each move violate the assumption of feature independence, resulting in poorer generalisation. This is in contrast to the random dataset which would include a more variety of playing styles and patterns, which could be based on more straightforward principles like material balance which Naive Bayes can detect with better accuracy.

6.2 MMNB Analysis

To evaluate the performance of Naive Bayes in a minimax algorithm, two versions were compared. The first being MMNB integration which combines the Naive Bayes evaluation with a traditional evaluation function, and the second being MMNB substitution which completely replaces the evaluation function in the minimax algorithm. Several performance indicators were used to analyse the 2800 games played including mobility, blunder frequency and stockfish evaluations. These indicators provide a comprehensive understanding of the engine's ability in making strategic decisions by both implementations.

The results between MMNB integration and MMNB substitution, reveal a much stronger performance from the MMNB integration. This is supported by the graph in Figure 6.1 as well as the raw values obtained.

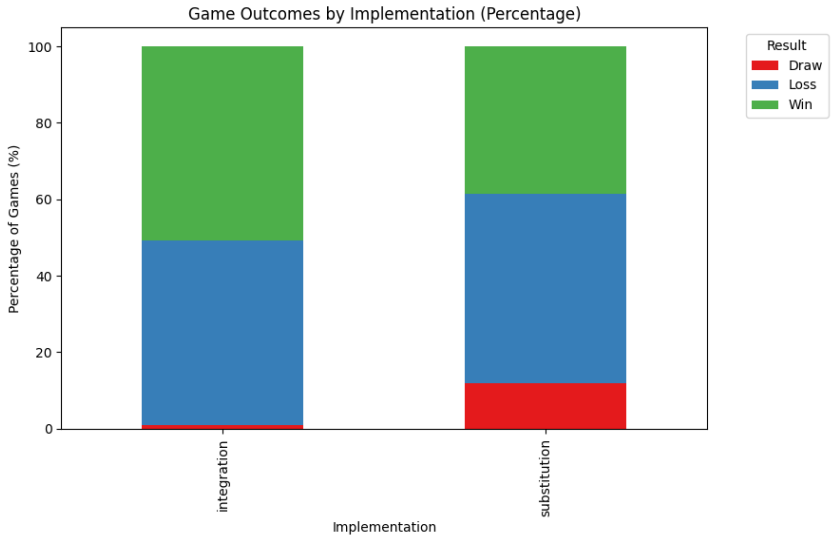


Figure 6.1: Game Outcomes of each Implementation

The integration implementation achieved a win rate of 50.7%, with loses at 48.3% and draws only at 1.0%. In contrast, the substitution implementation achieved a win rate of 38.6%, a similar loss rate of 49.4% and a much higher draw rate of 12.0%. This notable difference in draw rates suggests that MMNB substitution wasn't able to identify moves that would lead to wins even when in advantageous positions or the inability to identify crucial moves.

It is important to seek deeper insight into this pattern by comparing the results of the two implementations against both opponents, Stockfish and random engine. Against the random opponent, MMNB integration dominated, winning 99.4% of the games and never losing. MMNB substitution, however, did do well but achieved a much lower win rate of 77.3% and a much higher draw rate of 22.7%. This contrast of win rates, indicate that even against a low-skill opponent, with no strategy, the substitution engine was much worse at converting advantages into wins.

This is further supported by the games against Stockfish. MMNB integration was able to win 2.0% of the games and draw 1.4% of games whereas MMNB substitution failed to win even a single game, drawing only 1.2% of the time and losing 98.8% of the games. This highlights the limitation of the substitution implementation which is the evaluation function. The evaluation function which solely relies on the Naive Bayes probabilities, does not have the necessary understanding to win against a strong opponent. Despite being outperformed by Stockfish, the integration method showed that ability to exploit certain rare strategic opportunities, which

is likely only due to the hybrid approach. These findings reinforce the hypothesis that Naive Bayes is best used as a supporting tool in a chess engine. The simplicity of the Naive Bayes classifier can provide some useful insights but is insufficient to be used as a standalone evaluation function.

Figure 6.1 is a good indicator of the overall performance of both engines but does not provide an insight of the quality of moves and efficiency of the engines. One way to measure the quality of moves is to compare the stockfish evaluation after each move as well as the average value of the blunders made by each engine. The results are shown in Figure 6.2.

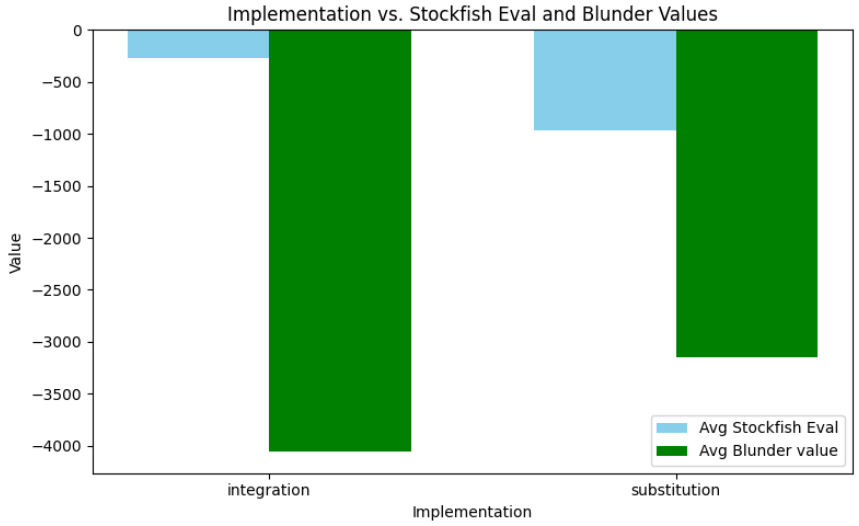


Figure 6.2: Stockfish Evaluation and Blunder Value of each Implementation

The Stockfish evaluation is a numerical representation of the the move quality, where a higher value indicates a better move. Figure 6.2 that the MMNB integration engine had a higher average stockfish evaluation than the MMNB substitution engine, which is consistent with the conclusions obtained based on the win rates. MMNB integration had an average stockfish evaluation of -340.02, whereas MMNB substitution had an average stockfish evaluation of -1048.25. However what is interesting is the average blunder value of each implementation. A blunder was defined as a move that caused a decrease in the Stockfish evalutaion by 300, where a more negative value indicates a more severe blunder based on Stockfish's opinion. MMNB integration had an average blunder value of -4693.34, whereas MMNB substitution had an average blunder value of -2652.59. This suggests that even though MMNB integration was able to win more games, it made more severe blunders. This could suggest that the integration method played much more aggressive and making more risky moves. This would explain the

higher win rate but also the much higher average blunder value. Whereas the substitution engine played much more conservatively and avoided making more riskier moves, which is why it had a lower win rate since it was unable to exploit certain opportunities.

As much as blunder value indicates that integration caused much more severe blunders, it is also important to consider the average number of blunders made by each implementation. Despite the average blunder value being lower for the substitution implementation, it made over 6 times more blunders per game than the integration implementation. MMNB integration had an average of 3.17 blunders per game whereas MMNB substitution had an average of 17.90 blunders per game. This shows that the substitution implementation was much more prone to blunders, despite having a much lower average blunder value. This further confirms the theory that solely basing the evaluation function on the Naive Bayes classifier is detrimental to the engine's performance whereas a an approach that combines traditional methods and Naive Bayes can be much more fruitful.

Two good indicators of the performance of the engines during different phases of the game, is mobility and piece balance. Piece balance meaning the difference in material between both players and mobility referring to the number of possible moves the player can make. Figure 6.3 and 6.4 show the average piece balance and mobility of each implementation over different phases of the game. The opening phase was defined by the first 25% of the game, midgame phase was defined by the next 50% of the game and endgame was defined by the last 25% of the game.

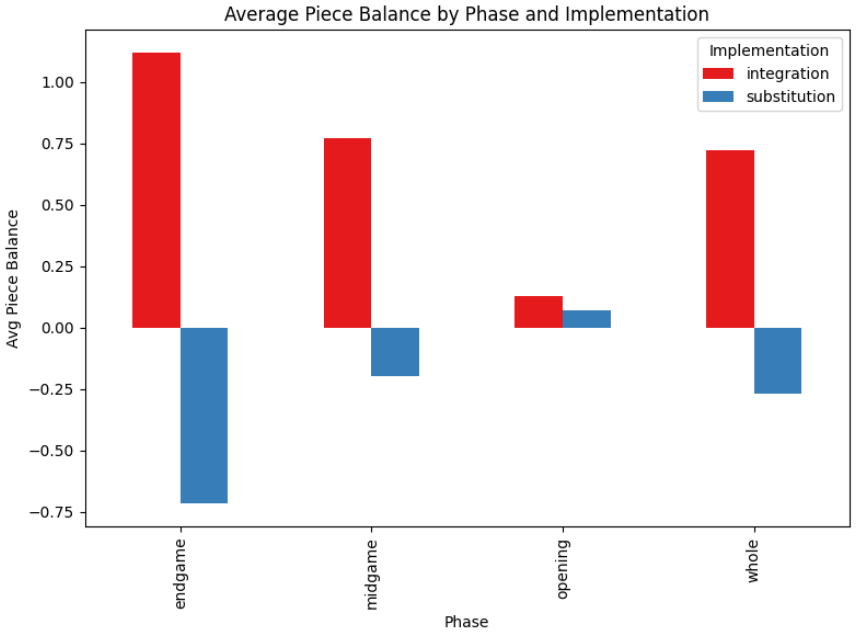


Figure 6.3: Average Piece Balance per Game of each Implementation

Figure 6.3 shows that the integration method always had a higher piece balance than the substitution method across all the different game phases. MMNB integration achieved an average piece balance of 0.723, indicating the engine's understanding of the importance of having material advantage over the opponent. On the other hand, MMNB substitution has an average piece balance of -0.267, indicating that the engine was often at a material disadvantage. The integration implementation consistently outperformed the substitution implementation in piece balance across all game phases. Both implementations, in the opening phase, had similar positive piece balances indicating their understanding of the importance of material advantage. However during midgame and end game, MMNB integration considerably outperformed the substitution implementation, achieving an average piece balance of 0.769 and 1.119 respectively, whereas the substitution implementation had an average piece balance of -0.198 and -0.716 respectively. This highlights the integration method's ability to preserve and accumulate material advantage, across different stages. These observations suggest that combining Naive Bayes with a classical evaluation function supports better piece management and leads to fewer unfavourable trades. Conversely, reliance purely on Naive Bayes leads to more frequent disadvantageous exchanges, decreasing overall material count over time which is critical in the midgame and endgame phases. The hybrid approach evidently takes advantage of core chess principles, while still benefiting from probabilistic insights, resulting in higher piece balance

and overall performance.

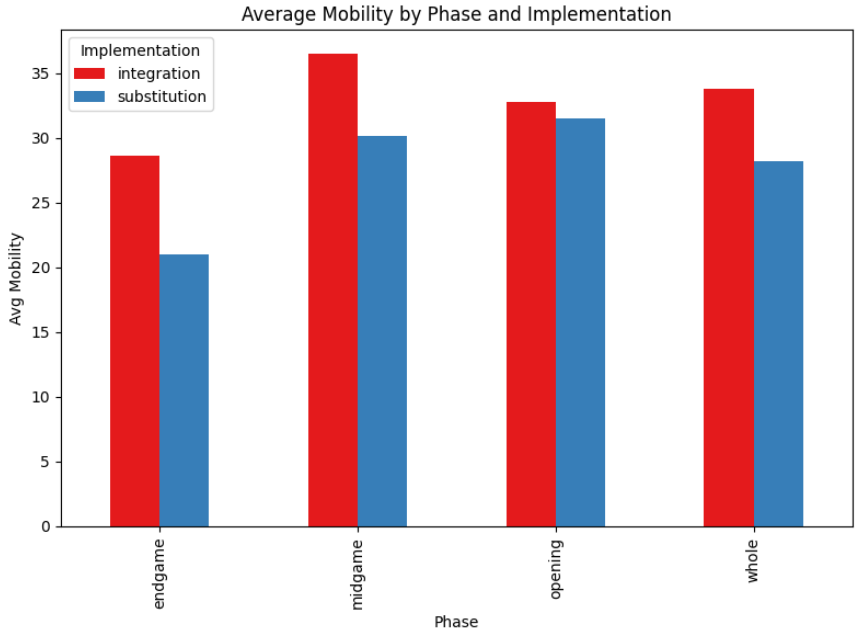


Figure 6.4: Average Mobility per Game of each Implementation

Mobility is measured by the average number of legal moves available at each turn. In chess theory, a higher mobility correlates with better board control, increasing tactical opportunities. Again, the integration method consistently surpassed the substitution method in mobility across all game phases. Overall, MMNB integration achieved an average mobility of 33.75 whereas MMNB substitution achieved an average mobility of 28.20, a gap of about 5.55 moves. In the opening, the average mobility is very similar inidicating similar strategies by both implementations. However, in midgame, where there is more complexity and opportunities since there are still a lot of pieces but are more developed. Integration has a higher mobility by about 6.34 moves (36.50 vs 30.16). This result was further amplified in the endgame where integration had an average mobility of 28.58 compared to 21.02 of the substitution implementation, a difference of 7.56 moves. Endgames require more precise and calculated moves minimising mistakes. This difference between the two indicates that MMNB substitution often causes pieces to move into more restricted or disadvantageous positions, while integration retains better piece coordination and mobility. There are a number of points that can be learnt from these results. Firstly, the integration method's superior mobility suggests it aims to avoid cramped locations and favour open positions, which is crucial throughout the game. MMNB substitution is strictly relying upon the assumption of feature independence. In the domain of chess where all features of the

game are interdependent, it can lead to suboptimal decisions. The integration method's ability to combine the strengths of both Naive Bayes and traditional evaluation functions allows it to better navigate the complexities of chess, leading to improved mobility. These findings corroborate what has been concluded from previous metric results, indicating that pure Naive Bayes is less effective in understanding the nuances of the board while a hybrid approach preserves strategic principles and enhances mobility.

So far what has been assessed between the two implementations is different metrics to assess the performance of the engine. Another factor that is important to consider is the the time taken to make each move as if it is to be used in real time, it is important that it is able to make decisions in a reasonable time period. Figure 6.5 shows the average time taken to make each move across the different phases of the game.

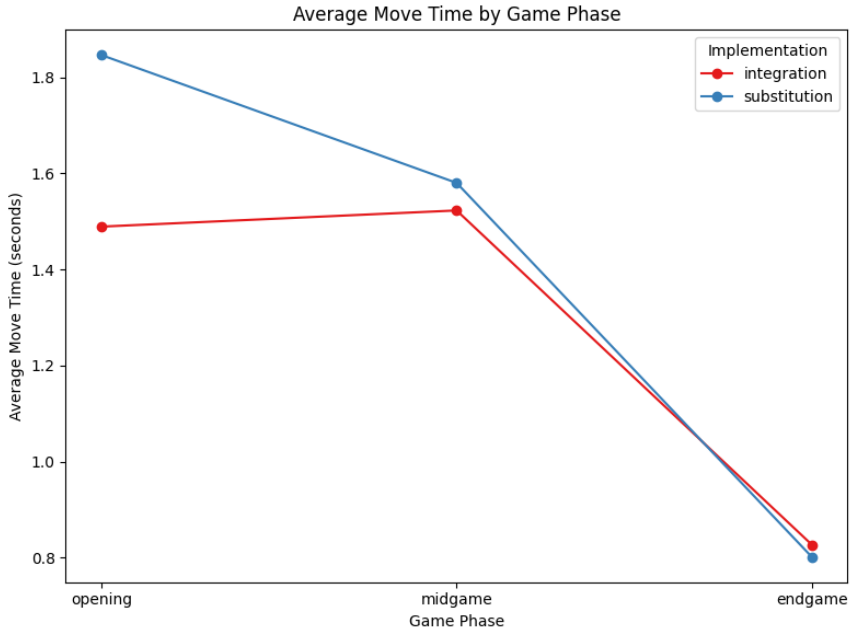


Figure 6.5: Average Time Taken to Make Each Move of each Implementation

The horizontal x-axis shows reflects the different phases of the game. The overall average time taken to make a move for MMNB integration was 1.334 seconds in comparison to 1.444 seconds for MMNB substitution. This indicates the integration method was able to make decisions faster than the substitution method. This is an important result as it hightlights that MMNB integration is not only more effective in winning games but also is more efficient in making decisions. In the opening phase, substitution has the longest move time at 1.846 seconds, which is 0.36 seconds longer than integration. This suggests that the pure Naive Bayes

evaluation requires more computational effort or more likely, struggles to prune effectively early on. This is likely due to the fact that the opening phase there are many more pieces so a lot more possible moves, requiring more time to evaluate moves. In midgame the overall time taken to make a move decreases in both methods to 1.522 seconds for integration and 1.580 seconds for substitution. During the midgame phase, the complexity of the game increases however the overall number of pieces on the board decreases, which would require less time to evaluate moves. In the endgame phase, both engines drop below 1 second, with integration taking 0.827 seconds and substitution taking 0.801 seconds. This is expected as during endgame there are much less pieces on the board, reducing the branching factor of the search tree. Substitution is slightly faster than integration in the endgame phase, but this is most likely due to what was discussed earlier and that substitution on average has less material on the board, resulting in less possible moves, requiring less time to evaluate. In the openinng and midgame phases, the substitution method seems to either take longer to evaluate moves or is less efficient in pruning the search tree. As the game progresses, there are less pieces and opportunities to make moves, narrowing the difference between the two implementations. Another important observation to note is the efficiency vs. effectiveness of the two implementations. Generally, an increase in one, causes a decrease in the other however the data shows otherwise. While substitution invests more time in evaluating moves, it is still not able to outperform integration, suggesting that longer computation time doesn't necessarily result in better moves.

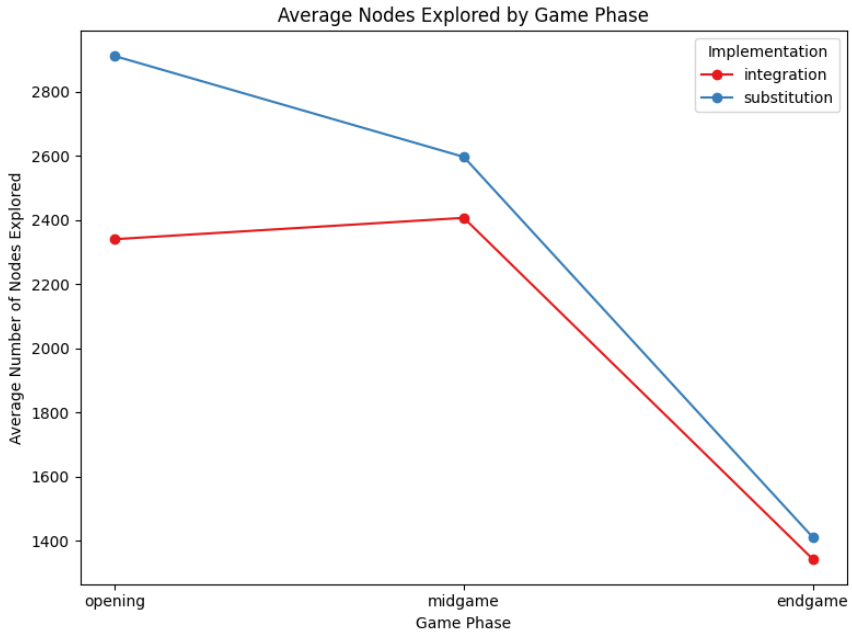


Figure 6.6: Average Nodes Explored per Game of each Implementation

Figure 6.6 shows the average number of nodes evaluated during the search of the game trees. The general trend is that the average number of nodes explored decreases as the game progresses. Similar to average time taken to make a move, this is expected as the number of pieces on the board decreases, reducing the branching factor of the search tree. The average number of nodes explored by MMNB integration was 2115 whereas the average number of nodes explored by MMNB substitution was 2366. In the opening phase, substitution explored 2910 nodes on average compared to 2339 for integration. This difference of 571 nodes indicates that the substitution method was less efficient in pruning the search tree, affirming the earlier observation that substitution takes longer to evaluate moves. The number of nodes explored closer to endgame is much closer between the two implementations. These results point towards the success of the hybrid approach in effectively pruning the search tree, leading to a more efficient evaluation process due ot the insight it gains from both the Naive Bayes classifier and the traditional evaluation function.

These two figures show that MMNB substitution contiuously spends more time and searches more nodes in early phases of the game but despite this still fails to yield better results than MMNB integration. This pattern reinforces that combining Naive Bates with standard heuristics genreally results a more efficient and reliable approach to the minimax search over exclusively relying on Naive Bayes, particularly during opening phases.

6.3 Feature Selection Analysis

The feature sets were designed to be progressively more complex, with the aim of evaluating the impact of feature selection on the performance of the Naive Bayes classifier. The results from the previous section show that the addition of more features generally leads to better performance. However it is also important to see their impact on the gameplay of the engine. Figure 6.7 shows the overall performance of each feature set in terms of wins, draws and losses.

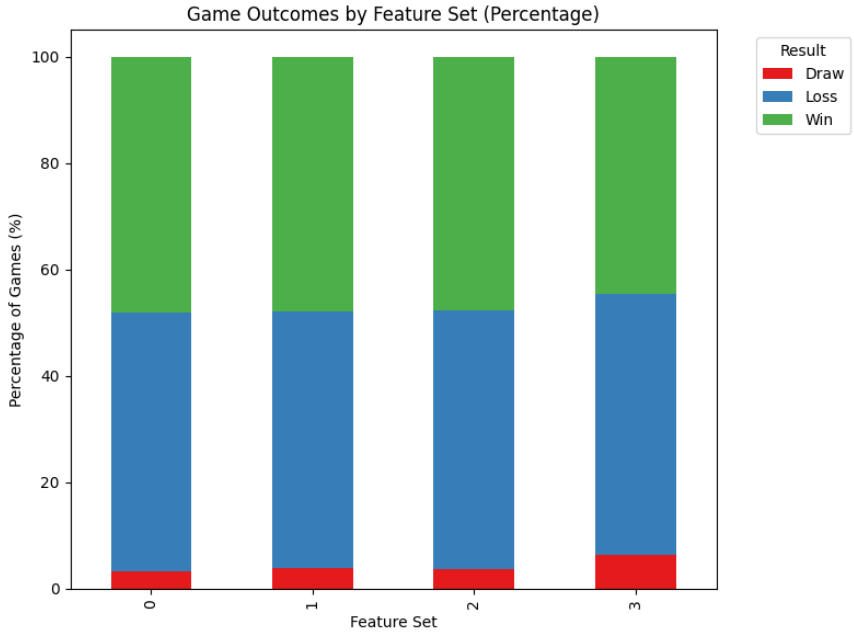


Figure 6.7: Game Outcomes of each Feature Set

Overall the different feature sets showed a similar trend in performance. Feature sets 0, 1 and 2 had win rates around 48%, loss rates around 48% to 49% and draw raters around 3% to 4%. However, feature set 3 had a noticeable jump in draw rates to 6.4%, nearly double the previous feature sets. It also had a noticeable decrease in win rates to 44.5%. The increase in draw rates could suggest that the engine was more conservative in its play, staying away from riskier moves. This would also clarify the decrease in win rate since the engine was not able convert games into wins due to its defensive play. These results indicate that the addition of more features does not have a big impact on the gameplay of the engines. The similar win and loss rates across feature sets 0, 1 and 2 indicate the engine's ability to apply basic chess principles but was unable to apply the increasing complexity of the features and learn from the increased information. The sharp change in rates for feature set 3, however, indicate that the the complexity of the features may have caused the engine to become more uncertain in its moves, causing it to draw more and fail to win as much. This could be due to feature overlapping features causing the classifier to assign incorrect probabilities [20] or could indicate a possible overfitting of the model. These results also could show the diminishing results of adding more features. The more simpler features might have captured most of the information and the added specialised features did not increase the information gained by the model. One last point that is notable between the first 3 feature sets is that the loss rates were

very similar, however there was a slight trend of decreasing win rates. Feature set 0 achieved a win rate of 48.1%, feature set 1 achieved a win rate of 47.9% and feature set 2 achieved a win rate of 47.6%. This suggests that the adding of features didn't majorly affect the engine's vulnerability to losing but did add some confusion to the model, causing it to convert potential wins into draws. Overall, this data shows that adding features slightly changes teh results, it does not necessarily improve the winning ability of the engine, as would be expected. This is proven by feature set 3, which included the features of all the other feature sets and more, still had the lowest win rate and highest draw and loss rate. This indicates that the underlying assumption of feature independence in Naive Bayes struggles to handle the interactions between a large number of chess features. Thus concluding more features doe not necessarily translate to better performance but rather depends on the quality of features.

Despite the similarity in win and loss rates, figures 6.8 and 6.9 show different metrics of each feature set.

Figure 6.8: Average Piece Balance of each Feature Set

Figure 6.8 shows the average piece balance of each feature set. Overall, all feature sets had a positive piece balance, indicating the engine's strong understanding in the importance of material balance. All of the feature sets included material balance as a feature. Feature 2 stands out in the data, achieving an average piece balance of 0.552, 0.345 pieces higher than the lowest. Indicating its ability to retain more material or gain more material across the game. This feature set included pawn structure as a feature which proves the importance of pawn structure in chess and the impact it can have on the overall material balance of the game.

Feature Set 3, surprisingly, again ranked the lowest with an average piece balance of 0.207. The additional complex features did not lead to improved material balance but rather made it significantly worse than the other feature sets.

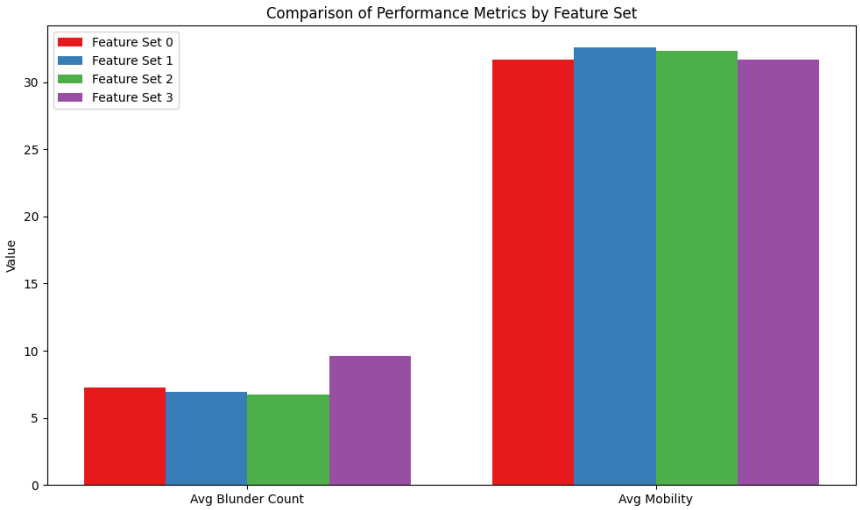


Figure 6.9: Average Blunder Count and Mobility of each Feature Set

Figure 6.9 shows the average number of blunders made by each feature set per game. The general trend indicates the increase in features used, decreases the overall number of blunders made. Feature set 0 obtained a blunder count of 7.258, and feature set 2 achieved the lowest at 6.728. This indicates the engine's increasingly ability to avoid making more blunders as more features were used, demonstrating the models increased learning. However, feature set 3 goes against this trend, obtaining the highest number of blunders per game at 9.615. Adding the most advanced features seem to unusually increase the number of severe mistakes made by the engine. This is provides further evidence that models trained using feature set suffer from overfitting or are affected by the dependency between features. The mobility of each feature set is roughly the same, with feature set 1 achieving the highest average mobility of 32.574 and feature set 0 achieving the lowest at 31.699. Higher mobility is a good indicator of better board control, resulting in more opportunities.

Considering both graphs, it can be concluded that the feature set with the best overall performance is feature set 2. It achieved the highest piece balance, lowest blunder count and close to best mobility. It appears to be able to balance between complexity of features and model assumptions. The pawn structures introduced in this feature set, isolated and doubled pawns, add meaningful power without overcomplicating the model which could cause it to

suffer. The most important point to understand from these results is the poor performance of feature set 3. Despite adding king safety, castling rights and game phase as features, it had the lowest piece balance, highest blunder count and close to bottom mobility. This suggests that the added features either caused the model to overfit or the fact that Naive Bayes fails to handle features that may contain more complex relationships. This demonstrates the importance of feature engineering, and that the increase in number of features does not necessarily lead to better performance but rather depends on the quality of features chosen.

Finally, it is also important to analyse the efficiency of each feature set and if there is a benefit in using more complex features. Figures 6.10 and 6.11 show the average time taken to make each move and the average number of nodes explored to decide each move.

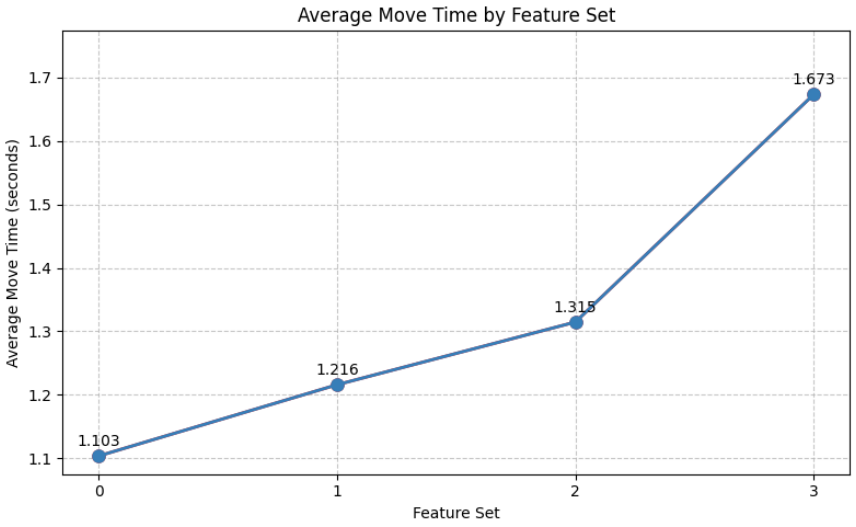


Figure 6.10: Average Time Taken to Make Each Move of each Feature Set

Figure 6.10 shows an overall trend of the increase in average time to make a move across the feature sets. It steadily increases from 1.103 seconds for feature set 0 to 1.315 seconds for feature set 2, which is as expected as more features are added, the model requires more time to calculate each feature, increasing the evaluation time. This is further supported by the sharp increase to 1.673 seconds for feature set 3. This is not only due to the increased number of features but also the drastic increase in the complexity of the features added. The addition of king safety, castling rights and game phase as features require much more computational effort to calculate. Even with a constant search depth, the evaluation becomes more expensive with increasing features.

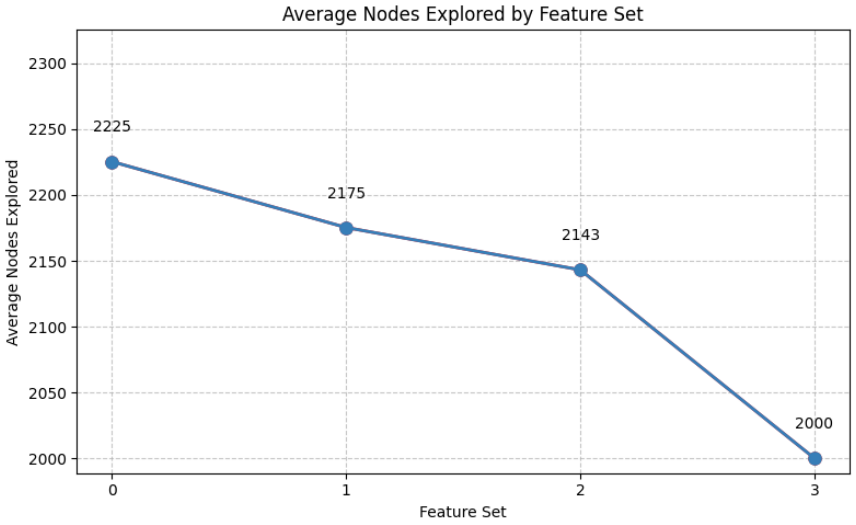


Figure 6.11: Average Nodes Explored per Game of each Feature Set

The general trend shown in figure 6.11 is that the average number of nodes explored decreases as the game progresses. There is a steady decrease between the first 3 feature sets from 2225 nodes for feature set 0 to 2143 nodes for feature set 2, with a more drastic decrease to 2000 nodes for feature set 3. This could suggest that the more complex features are able to prune the search tree more effectively. However, this goes against the earlier observation that the more complex features require more time to evaluate, since less nodes explored should lead to less time to decide a move. This suggests that the increase in time taken is not due to the number of nodes evaluated but rather the number of features used and the complexity of the features.

What can be concluded from these results is that increasing the complexity of the features does guarantee better outcomes. Despite fewer nodes being explored, the cost of evaluation increases the total move time significantly. In real-time chess engines, it is important to find a balance between a strong evaluation function and the efficiency of the evaluation. It also indicates the the additional overhead of utilising more features can be counterproductive.

Chapter 7

Conclusion and Future Work

The project's conclusions should list the key things that have been learnt as a consequence of engaging in your project work. For example, "The use of overloading in C++ provides a very elegant mechanism for transparent parallelisation of sequential programs", or "The overheads of linear-time n-body algorithms makes them computationally less efficient than $O(n \log n)$ algorithms for systems with less than 100000 particles". Avoid tedious personal reflections like "I learned a lot about C++ programming...", or "Simulating colliding galaxies can be real fun...". It is common to finish the report by listing ways in which the project can be taken further. This might, for example, be a plan for turning a piece of software or hardware into a marketable product, or a set of ideas for possibly turning your project into an MPhil or PhD.

???INCLUDE Training models on only a certain game phase like opening, mid or end

Also investigateing whcih features had the most impact, and if the complex features can be useful alone

???

References

[1] H. A. Davidson, *A Short History of Chess*. Crown.

[2] F.-H. Hsu, "IBM's Deep Blue Chess grandmaster chips," vol. 19, no. 2, pp. 70–81.

[3] C. E. Shannon, "XXII. Programming a computer for playing chess," vol. 41, no. 314, pp. 256–275.

[4] D. Klein, "Neural Networks for Chess."

[5] I. Kamlish, I. B. Chocron, and N. McCarthy, "SentiMATE: Learning to play Chess through Natural Language Processing."

[6] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence, Pearson, fourth edition, global edition ed.

[7] D. Lowd and P. Domingos, "Naive Bayes models for probability estimation,"

[8] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos, "An evaluation of Naive Bayesian anti-spam filtering."

[9] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz, "A Bayesian Approach to Filtering Junk E-Mail,"

[10] T. Ige and S. Adewale, "AI Powered Anti-Cyber Bullying System using Machine Learning Algorithm of Multinomial Naïve Bayes and Optimized Linear Support Vector Machine," vol. 13, no. 5.

[11] C. A. Ul Hassan, M. S. Khan, and M. A. Shah, "Comparison of Machine Learning Algorithms in Data classification," in *2018 24th International Conference on Automation and Computing (ICAC)*, pp. 1–6.

[12] E. Stephens, "The mechanical Turk: A short history of 'artificial artificial intelligence'," vol. 37, no. 1, pp. 65–87.

[13] S. DeCredico, "Using Machine Learning Algorithms to Predict Outcomes of Chess Games Using Player Data,"

[14] "Chess Game Dataset (Lichess)."

[15] J. R. Landis and G. G. Koch, "The Measurement of Observer Agreement for Categorical Data," vol. 33, no. 1, pp. 159–174.

[16] "Python-chess: A pure Python chess library — python-chess 0.14.1 documentation."

[17] "The GNU General Public License v3.0 - GNU Project - Free Software Foundation."

[18] N. Fiekas, "Niklasf/python-chess."

[19] I. Rish, "An empirical study of the naive Bayes classifier,"

[20] Z. Ahmed, B. Issac, and S. Das, "Ok-NB: An Enhanced OPTICS and k-Naive Bayes Classifier for Imbalance Classification With Overlapping," vol. 12, pp. 57458–57477.

Appendix A

Extra Information

A.1 Tables, proofs, graphs, test cases, ...

The appendices contain information that is peripheral to the main body of the report. Information typically included in the Appendix are things like tables, proofs, graphs, test cases or any other material that would break up the theme of the text if it appeared in the body of the report. It is necessary to include your source code listings in an appendix that is separate from the body of your written report (see the information on Program Listings below).

Appendix B

User Guide

B.1 Instructions

You must provide an adequate user guide for your software. The guide should provide easily understood instructions on how to use your software. A particularly useful approach is to treat the user guide as a walk-through of a typical session, or set of sessions, which collectively display all of the features of your package. Technical details of how the package works are rarely required. Keep the guide concise and simple. The extensive use of diagrams, illustrating the package in action, can often be particularly helpful. The user guide is sometimes included as a chapter in the main body of the report, but is often better included in an appendix to the main report.

Appendix C

Source Code

C.1 Instructions

Complete source code listings must be submitted as an appendix to the report. The project source codes are usually spread out over several files/units. You should try to help the reader to navigate through your source code by providing a "table of contents" (titles of these files/units and one line descriptions). The first page of the program listings folder must contain the following statement certifying the work as your own: "I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary". Your (typed) signature and the date should follow this statement.

All work on programs must stop once the code is submitted to KEATS. You are required to keep safely several copies of this version of the program and you must use one of these copies in the project examination. Your examiners may ask to see the last-modified dates of your program files, and may ask you to demonstrate that the program files you use in the project examination are identical to the program files you have uploaded to KEATS. Any attempt to demonstrate code that is not included in your submitted source listings is an attempt to cheat; any such attempt will be reported to the KCL Misconduct Committee.

You may find it easier to firstly generate a PDF of your source code using a text editor and then merge it to the end of your report. There are many free tools available that allow you to merge PDF files.