Naive Bayes Classifier in a Chess Engine

Final Project Report

Author: Mohammad Ibrahim Khan

Supervisor: Jeffery Raphael

Student ID: k22013981

March 27, 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

Contents

Chapter 1

Introduction

Since the creation of chess over 1500 years ago [1], most agree that the biggest event in the games history was when IBM's Deep Blue defeated 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 mentions there is 10^{120} possible positions in chess [3] which is more than the number of atoms in the observable universe. Therefore, it is infeasible to brute force the game and since 1997, there have been attempts to create the perfect chess engine. Solving chess could be a pathway to solve 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 aren't 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 is a popular algorithm which is fundamental in chess engines. This, in conjunction with Alpha-Beta pruning, allows the engine to search through a game tree in order to find the best move. Due to the exponential growth of the game tree, its not feasible to search the entire tree. This is where machine learning is used to predict the best move in order to reduce the search space. Many techniques have been used from Neural Networks [4] to Natural Language

Process [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 been seen to be 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 wants to see if this principle holds in this domain.

The standard minimax algorithm is limited by the evaluation function. Usually they only base themselves on piece values. However, chess has more intricacies that may not be seen at face value, including piece positions relative to each other as well as identifying weaknesses. This is where a Naive Bayes classifier could be utilised to try to learn these intricacies better than a standard evaluation function.

The purpose of this project is to explore the potential of using a Naive Bayes Classifier in a chess engine to improve the performance of a minimax based chess engine. Is it viable for a simpler machine learning algorithm like Naive Bayes to be used in a chess engine in order to reduce complexity of the engine but without sacrificing performance? This is the primary research question that this project will explore. It will also explore different ways of implementing the classifier, for example, could it completely replace the traditional evaluation function or used in order to support it.

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. Chess is an ideal candidate 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 is the game, therefore the game state is only determined by the each players moves.

2.2 Search Algorithms

The most popular way to design a chess engine is by using a search algorithm. 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:

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 a one such technique

Algorithm 1 Minimax Algorithm

```
function MINIMAX(Node, Depth, MaximizingPlayer)
    if Depth = 0 or Node = Leaf then
        return EVAL(Node)
    end if
    if MaximizingPlayer then
        $Value \leftarrow -\infty$
        for each in Node do
            $Value \leftarrow \max(value, \text{MINIMAX}(child, depth - 1, \textbf{false}))$
        end for
        return $Value$
    else
        $Value \leftarrow \infty$
        for each Child in Node do
            $Value \leftarrow \min(value, \text{MINIMAX}(child, depth - 1, \textbf{false}))$
        end for
        return $Value$
    end if
end function
```

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 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.

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 dependant 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

The most well-known chess engines are Stockfish and AlphaZero. AlphaZero, designed by DeepMind, uses a deep neural network in conjunction with reinforcement learning which allows it to teach itself how to play. Initially it has no understanding of the game other than the basic rules, then it plays against itself and uses the result of the game to update parameters in the neural network. Stockfish however uses a more traditional approach, utilising alpha-beta pruning with minimax with a evalutaion function that is based on numerous elements of the game. Recently it has also implemented a neural network to improve its evaluation function. However both these engines require a lot of computational power and also require a large dataset to generalise well, whereas Naive Bayes is a much simpler algorithm that requires less power and can generalise well even with a small dataset. This is the reason why this paper is concentrated on observing the impact of Naive Bayes on chess. This is because it is very simple and makes assumptions that are generally unrealistic, however it has been shown to be effective in domains like spam detection [7] despite these simplistic assumptions.

Naive Bayes, sometimes also known as Idiot Bayes or Simple Bayes, is a simple classification algorithm that is based upon Bayes' theorem (Equation 2.1) [8].

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

It is referred to as naive as it assumes each input variable $X_1, X_2, ..., X_n$ is conditionally independant given the class. Despite this assumption not being true in most cases, it still performs well in practise. This assumption allows probability distributions to be efficiently represented as the product of the individual probabilities of the the input variables (Equation 2.2) [8].

$$P(X_1, X_2, ..., X_n, C) = P(C) \cdot \prod_{i=1}^{n} P(X_i|C) \tag{2.2}$$

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 seen effective in text classification yet is not widely researched in chess. This literature reviews 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 divrese 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 [9] 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) [10] 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 results indicate the power Naive Bayes has despite its simplicity and assumption of feature independence. 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 was interesting from these results as well, was the fact that the model did significantly better when more features were used, such that the model was able to achieve 100% precision and 98.3% recall. Androutsopoulos et al. [9] 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. [10] , 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 [9], 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.

Naive Bayes has shown to be very effective in text-based domains including spam detection as discussed before as well as in anti-cyber bullying systems [11]. However it has also been shown to be not as effective in other contexts. Hassan, Khan and Shah (2018) [12] 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 [13]. 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 itselves 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 upon neural networks. This project aims to investigate if other techinques can yield the same result, specifically Naive Bayes.

3.3 Naive Bayes in Chess

The research on applying Naive Bayes in chess engines is very limited. It is a space where there is potential for exploration. One of the most relevant research on this topic is by DeCredico (2024) [14]. In this recent paper, DeCredico explore 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 combination 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 bu 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 seperately 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

Requirements and Specification

4.1 Introduction

TODO????

4.2 Functional Requirements

- The system should be able to play a game of chess
- The system should be able to evaluate the game state
- The system should use a Naive Bayes Classifier during evaluation
- The system should be able to generate moves
- The system should make sure generated moves are legal
- The system should generate moves based on evaluation

4.3 Non-Functional Requirements

- The system should be comptabile with an average device
- The classifier should be able to evaluate in real time

- The system should generate moves in a reasonable time
- The system should be testable against Stockfish
- The classifier should be scalable to utilise large datasets

Chapter 5

Methodology

5.1 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.

5.2 Random Chess Engine

5.3 Integration of Naive Bayes with Minimax

5.3.1 Data Preparation

The dataset used for this project was obtained from Kaggle [15]. 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.

5.3.2 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 5.1 were used [3].

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

Table 5.1: Values of Chess Pieces

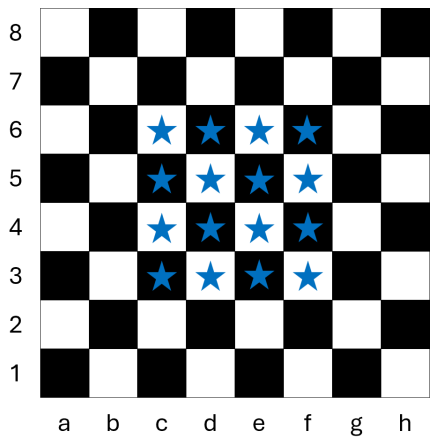
Piece mobility is the difference in number of legal moves between the two players. The more moves available to a player suggests that they have more control over the board which could give them a tactical advantage. King attack balance is the difference in number of pieces attacking the king. Most of 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 5.2.

	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

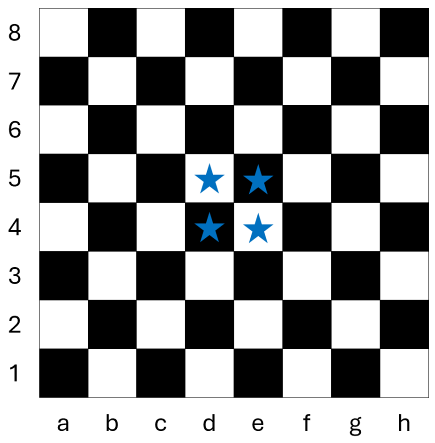
Table 5.2: Positional Value Table for Knight

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 5.1.



(a) Big centre of the Board



(b) Small centre of the Board

Figure 5.1: 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.

5.3.3 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 5.1.

$$P(X|C) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{5.1}$$

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.

5.3.4 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 5.2.

$$\text{Accuracy} = \frac{\text{Correctly Classified Instances}}{\text{Total Instances}} \tag{5.2}$$

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 5.3 and 5.4 respectively. These two metrics are usually related as there is a trade off between the two, generally increasing one causes the other to decrease.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{5.3}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{5.4}$$

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 5.3: Interpretation of Kappa Statistic

recall which is what will be used for this project.

$$F_\beta = (1+\beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{5.5}$$

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 5.6 .

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{5.6}$$

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 5.3 [16].

5.3.5 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 5.1. The positional values were also calculated similar to what was used for the feature extraction 5.2. 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 5.7.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{5.7}$$

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.

Chapter 6

Report Body

The central part of the report usually consists of three or four chapters detailing the technical work undertaken during the project. **The structure of these chapters is highly project dependent**. They can reflect the chronological development of the project, e.g. design, implementation, experimentation, optimisation, evaluation, etc (although this is not always the best approach). However you choose to structure this part of the report, you should make it clear how you arrived at your chosen approach in preference to other alternatives. In terms of the software that you produce, you should describe and justify the design of your programs at some high level, e.g. using OMT, Z, VDL, etc., and you should document any interesting problems with, or features of, your implementation. Integration and testing are also important to discuss in some cases. You may include fragments of your source code in the main body of the report to illustrate points; the full source code is included in an appendix to your written report.

6.1 Section Heading

6.1.1 Subsection Heading

Chapter 7

Design & Specification

7.1 Section Heading

Chapter 8

Implementation

8.1 Introduction

The implementation of the Naive Bayes classifier in the chess engine was done in Python. The implementation was done in two parts. The first part was to implement the Naive Bayes classifier and the second part was to implement the classifier in the chess engine. The implementation of the Naive Bayes classifier was done using the scikit-learn library. The implementation of the classifier in the chess engine was done using the python-chess library.

Chapter 9

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.

9.1 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) [17]. This license is a free software license that allows developers to freely use, study and modify the python-chess library for their projects [18]. 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 [19].

This research utlises the 'Chess' dataset available on Kaggle [15] 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.

9.2 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.

9.3 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.

9.4 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, gave 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.

To ensure transparency and ease of use of the chess engine, a number of measures were taken

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 as well as potential ethical issues related to the engine were also discussed.

Chapter 10

Results/Evaluation

10.1 Naive Bayes Evaluation

The first experiment was to evaluate the Naive Bayes classifier on its own. The classifier was trained on around 10,000 games from the Lichess database. Then features were extracted as mentioned in the methodology section, eventually resulting in the following number of instances:

Class	**Count**
Black wins	56826
White wins	57940

Table 10.1: Naive Bayes Training Data

This data was a good split of the two classes, removing the issue of class imbalance. Then this data was split into training and testing data. The classifier used the features as mention in the methodology section. After training, the classfier was tested on unseen data. The results are shown in the following table.

Metric	**Value**
Accuracy	0.6080
Precision	0.6254
Recall	0.5481
F1 Score	0.6066
Kappa Score	0.2165

Table 10.2: Naive Bayes Evaluation

The results show that the classifier doesn't perform very well. The accuracy is 0.0608 whcih is slightly better than randomly guessing which would have an accuracy of 0.5. This shows that the classifier is learning to some extent but not very well. The precision is 0.6254 and recall is 0.5481 whcihc suggests that the classifier is better at predicting the positive outcomes (white winning) than the negative outcomes (black winning). The F1 score shows that the calssifer is slightly effective but there is significant need to improve it, this is similarly shown by the low Kappa score.

Using this classifier in the chess engine, it will not be very effective. However, the classifier was used in the minimax algorithm to see how it would perform. For this experiment, the classifier will completely replace the evaluation function. It is tested by playing against a random engine.

The results are shown in the following table:

Games Played	10
Games Won	0
Games Lost	0
Games Drawn	10

Table 10.3: Naive Bayes Minimax Evaluation

The results show that when the classifier is used on its own, it is unable to win any games against a random engine. This is what was predicted as it would be hard for the classifier to learn the nuances of chess. However what is interesting from the games is that even though every game was drawn, on average it had 12 more pieces than random. This indicates that the classifier understands that having more pieces is advantageous so tries to protect its pieces and also capture opponent pieces. The issue is, however, that it doesn't understand how to cause checkmates or how to protect its king, which is vital to winning a game.

The engine that used Naive Bayes above was then also tested against Stockfish at level 0. The results are shown in the following table:

Games Played	10
Games Won	0
Games Lost	10
Games Drawn	0

Table 10.4: Naive Bayes Minimax Evaluation Against Stockfish

This was expected as if the engine couldn't win against a random engine, that has no strategy to its game play, it woudld be near to impossible to win against Stockfish.

The engine that used the naive bayes classifier above used a depth of 3 for the minimax algorithm. This is generally quite low for a chess engine. The same tests were run with the same Naive Bayes model but with a depth of 4 for the minimax algorithm.

The results are shown in the following table:

Games Played	10
Games Won	0
Games Lost	10
Games Drawn	0

Table 10.5: Naive Bayes Minimax Evaluation Depth 4

It seems from these results that increasing the depth of the minimax algorithm didn't improve the performance of the engine. This further suggests that the issue is with the evaluation function that solely relies upon the classfier. Another observation from this is that the time taken for each move is about double compared to when the depth was 3, which is expected as the search tree is much larger.

A minimax algorithm with alpha beta pruning using traditional evaluation function was also implemented, to be used as a benchamark as well. The evaluation only considered the material balance and the positinal value of pieces. The same tests wiere conducted with a depth of 3.

The results against the random engine are shown below:

Games Played	10
Games Won	10
Games Lost	0
Games Drawn	0

Table 10.6: Traditional Minimax Evaluation Depth 4

What is interesting from these results is that the traditional algorithm was able to win all games, whereas the Naive Bayes algorithm was unable to win any games. Shown below is the algorithm against Stockfish at level 0.

Games Played	10
Games Won	1
Games Lost	9
Games Drawn	0

Table 10.7: Traditional Minimax Evaluation Against Stockfish

Here the algorithm is at least able to win 10% of the time against Stockfish. However, when the depth of the algorithm is increased from 3 to 4, the algorithm is able to win 50% of the time. This shows the importance of the depth used and, the significant impact it can have on the performance of an engine. However what is also important to note is the time taken to make a move increased 10 fold.

10.2 Naive Bayes with Traditional Evaluation Function

Based on the results above, it is clear that the traditional evaluation function does much better than the one that uses the Naive Bayes classifier. However, it was shown that the Naive Bayes does understand some aspects that are important like material balance. Therefore, the next experiment wants to explore the idea of whether the Naive Bayes classfier can help support the traditional evaluation function.

TODO:FINISH

10.3 Influence of Feature Selection

The features used to train a model are crucial to the perfomance of the model. This is the only picture of the world that the model has. The better the features, the more realistic picture it has of the world. The features used previously when trained with 10,000 games seemed to only yield an F1 score of 0.6, which is not much better than random guessing. The next experiment investigates the impact of feature selection on the performace of the model. Therefore the same model was trained with the same number of games but with different features. All the models were trained with 100,000 games and the features used for each model are shown below:

- Model 0: Was restricted on only using material balance, positional value, mobility, king attack, as features
- Model 1: Same as Model 0 but also considred the control of the centre. This was 2 seperate features, one for number of pieces in the 2 by 2 square in the centre and the other for the 4 by 4 square in the centre.
- Model 2: Same as Model 1 but also considered the structure of pawns. This was determined by the number of isolated pawns and doubles pawns. This equated to 4 more features, 2 for each colour.
- Model 3: Same as Model 2 but included more complex features. One being the castling rights of each player as well as a way to determine the game phase, either beginning, middle or end game. Another feature this model considered was king safety and this was determined by the number of pawns around the king as well as number of attacks on sqaures adjacent to the king.

Table 10.8: Model Performance Metrics

Model	F1 Score	Kappa Score	Accuracy	Recall	Precision
Model 0	0.60398	0.22042	0.60914	0.49486	0.64928
Model 1	0.60943	0.22190	0.61047	0.55795	0.62995
Model 2	0.61158	0.22456	0.61198	0.57858	0.62618
Model 3	0.61263	0.22513	0.61264	0.61920	0.61678

These results subtly show that the more features considered, the better the model generally performs, proven by the increase in F1 score from 0.603 for Model 0 to 0.613 for model 3. This supports the idea that the more information given to the model, the more it can understand about world. However the increase in model accuracy is very small that if an F1 score of 0.7 is aimed for, more than 150 features would be required. This is not feasible as during real time play, the engine would take too long to extract these features in order to make a move. The problems of misclassifying could be down to two factors, the first being the features used are not complex enough and not extracting enough nuances in the game that the model requires. However increase complexity of features would result in an engine that is much slower and not feasible to be used for real time game play. The second factor could be that the model is not complex enough to understand the intricacies of chess and therefore its unable to learn the patterns that grandmasters make to win games.

Chapter 11

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.

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] J. J. Eberhardt, "Bayesian Spam Detection," vol. 2, no. 1.

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

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

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

[11] 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.

[12] 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.

[13] E. Stephens, “The mechanical Turk: A short history of ‘artificial artificial intelligence’,” vol. 37, no. 1, pp. 65–87.

[14] S. DeCredico, “Using Machine Learning Algorithms to Predict Outcomes of Chess Games Using Player Data,”

[15] “Chess Game Dataset (Lichess).”

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

[17] “Python-chess: A pure Python chess library — python-chess 0.14.1 documentation.”

[18] “The GNU General Public License v3.0 - GNU Project - Free Software Foundation.”

[19] N. Fiekas, “Niklasf/python-chess.”

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.