6CCS3PRJ Final Year Individual Project Report Title

Final Project Report

Author: Mohammad Ibrahim Khan

Supervisor: Jeffery Raphael

Student ID: k22013981

March 21, 2025

Abstract

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

March 21, 2025

Acknowledgements

It is usual to thank those individuals who have provided particularly useful assistance, technical or otherwise, during
your project. Your supervisor will obviously be pleased to be acknowledged as he or she will have invested quite a lot of time overseeing your progress.

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 fundemental in chess engines. This, in conjuction 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 searcht 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 to Natural Language Process.

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 see 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 Occams 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 I hope a Naive Bayes classifier can be utilised to try to learn these intricacies better than a standard evaluation function.

The purpose of this project is to explore the idea 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. 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 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

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
```

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 [4], 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 [5] despite the 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) [6].

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

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) [6].

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

Chapter 3

Literature Review

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' [7] Androutsopoulos et al. 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) [8] 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).

Features	**Spam Precision**	**Spam Recall**
Words only	97.1%	94.3%
Words + Phrases	97.6%	94.3%
Words + Phrases + Non-Textual	100.0%	98.3%

Table 3.1: Spam detection performance of Naive Bayes with different features

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. 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., 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 [7], 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 [9]. However it has also been shown to be not as effective in other contexts. Hassan, Khan and Shah (2018) 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 [10]. 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 [11]. 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 [11].

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.

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

As mentioned previously, the focus of this paper is to explore the uses of Naive Bayes in chess and how this could impact other domains in computer science. This chapter will mention the methodology that was used to implement the Naive Bayes classifier in the chess engine.

5.2 Random Chess Engine

For the purpose of this project, the python-chess library will be used as a base to create chess engine. The reason for this choice of library is because it has integrated features needed for this project. This includes support for FEN notation, methods to get legal moves, and also it was also chosen due to the optimisations it uses including representing the board as a bitboard.

The first step was to create a chess engine engine that randomly picks moves. This implementation was very simple due to python-chess' method *board.legalmoves()* which provides all the legal moves available to the current player. Using this list of moves, one is then chosen at random. The purpose of this random engine is be used as a benchmark for the main chess engine to be created.

5.3 MiniMax and Alpha-Beta pruning

Minimax is used by the majority of chess engines. The aim is to implement Naive Bayes to improve the minimax algorithm. So intuitively the first step is implement minimax. Alpha-Beta is much more powerful than basic minimax as it will always give the same output as minimax

and is much faster. Therefore, Alpha-Beta will be used directly. This is the pseudocode for the algorithm used. A similar method is use for alphaBetaMin.

Algorithm 2 Alpha-Beta Pruning Algorithm

```
function AlphaBetaMax(Board, Alpha, Beta, Depth)
    if Depth = 0 or Game Over then
        return Eval(Board), None
    end if
    BestValue ← −∞
    BestMove ← None
    for each Move in legal_moves do
        MakeMove
        Score ← AlphaBetaMin(Board, Alpha, Beta, Depth − 1)[0]
        if Score > BestValue then
            BestValue ← Score
            BestMove ← Move
            Alpha ← max(Alpha, Score)
        end if
        if Score >= Beta then
            break
        end if
    end for
    return BestValue, BestMove
end function
```

When using alpha-beta pruning, a depth of 3 was used. This was due to the limited computational power available. The depth of the search tree is a primary factor that determines the performance of the chess engine. The higher the depth, the more possibilities the engine can explore which means each move made would have been given more consideration resulting in a better move.

The other factor of the minimax algorithm that is important is the evaluation function. This is what determines the moves that are made. It gives a value to a game state to approximate which player is winning at that point in time as it is infeasible to search the entire game tree. So the evaluation function is used to estimate the winner at a certain point in the game. The evaluation function used for the purpose of this project will be very simple and will be mainly based on material balance. This is a good indicator of the current state of the game as generally the more pieces a player has the more options they have, therefore they are more likely to win.

$$\text{Material Balance} = \text{Number of White Pieces} - \text{Number of Black Pieces} \tag{5.1}$$

However, considering each piece of equal value is not representative of the true value of pieces during the game. For example 1 queen is worth more than 2 pawns. Therefore, this research will use the values in Table 5.1 to calculate the material balance, which are the commonly

accepted values for each piece [12].

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

Table 5.1: Values of Chess Pieces

This evaluation function is relatively strong and incentivises the engine to take pieces when possible and also to protect its own pieces. After testing this evaluation function, as expected, the engine was winning initially by taking pieces. However, it never won against the random engine. After analysing the games played, the problem was in the end game. The evaluation function did not incentivise the engine to check the opponent, which is the most important part of chess. Therefore, checkmates were considered terminal nodes by giving them a value of $\pm\infty$ dependant on the player that is winning. A value of 10 was also given to check as this is a very strong move.

5.4 Data Collection

The choice of dataset used for the Naive Bayes Classifier was an important task to ensure the classifier had enough data to be trained on but also have good data to be trained on. There were many datasets that were considered, however in the end, the dataset from kaggle.com was used which had 6.25 million chess games played on lichess.com in July 2016 . This provided a lot of flexibility as there is a lot of data to train on. There were other datasets like a set of 4 million games on chess.com only played with grandmaster players. However, the aim is to train a model to work for all level of players so the lichess database was chosen.

The lichess dataset was in csv format. These are the features that were part of the dataset:

- **Event:** The type of game.
- **White:** White's ID.
- **Black:** Black's ID.
- **Result:** The outcome of the game (`1--0` if White wins, `0--1` if Black wins and `1/2--1/2` if they draw).

- **UTCDate, UTCTime:** The date and time (UTC) when the game was played.
- **WhiteElo, BlackElo:** ELO of the players.
- **WhiteRatingDiff, BlackRatingDiff:** The change in rating points after the game.
- **ECO:** Opening in ECO (Encyclopaedia of Chess Openings) encoding,
- **Opening:** Opening name.
- **TimeControl:** The time allocated for each player, plus any increment in seconds.
- **Termination:** The reason the game concluded.
- **AN (Algebraic Notation):** The sequence of moves in algebraic notation.

A lot of the features would not be very useful to train the model. The ID's of the players would not be useful to train with so was discarded. The UTCDate, UTCTime also would not be useful as the time of day a game is played wouldn't affect the outcome of the game or to decide the best move to make. The Elo ratings of the players would be useful as players of similar levels may play in similar ways however this also wasn't used as this information won't always be available when using the engine. The way the game terminated is also information that wouldn't be beneficial as all that is important is the final game result. The most important features that was utilised was the result of the game and the sequence of moves in algebraic notation.

The data then needed to be preprocessed. The moves in algebraic notation needed to be converted in a format that can be used by the Naive Bayes Classifier. For each game, the python-chess library to simulate the game, so for each move the board is updated. However using every move would be too much data to train on, especially since consecutive moves are highly correlated. Therefore, for most of the game the features at every 6 moves was used however during end game, every other move was used. This is because during the end game, each move will most likely have a bigger impact on the result of the game. For the purpose of this project, end game is after 75% of the moves have been made. The last move is also included in the data as this is the move that determined the result of the game.

5.5 Feature Selection

Material balance is the first feature implemented. Arguably this is the most important feature to estimate the current state of a chess game. For the purpose of this project, the values in

Table 5.1 were used to calculate the material balance, using Equation 5.2. The material balance is positive if the values of White's pieces are grater than the values of Black's pieces and vice versa.

$$\text{Material Balance} = \text{Number of White Pieces} - \text{Number of Black Pieces} \tag{5.2}$$

Another feature used is the positional value of pieces. The location of specific pieces on the board can influence how effective it is within the game. An example of a piece position table is given in Table 5.2 to calculate the positional value of a knight.

	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 in the centre it can control more squares and has more options whereas when it is near the edge, the knight is much more restricted, especially the corners where they have only 2 possible moves. Another example are pawns which are also more valuable in the centre of the board but also in squares that are close to promotion. They are usually not effective in the starting ranks. The positional values of all the black pieces are then summed up and subtracted from the sum of the positional values of the white pieces.

Another feature considered for the Naive Bayes Classifier is piece mobility. Piece mobility is the possible moves a piece can make. Generally the more moves available to a player, the more control they can have over the board. This is calculated as the difference between the number of legal moves White has and the number of legal moves Black has.

King Safety is another feature that is critical to the game. The winning or losing of the game solely lies upon how well you cant protect the king. There are many ways to define king safety. In this project, we will use the amount of pieces that are attacking the king. This is an important factor to consider since, the more pieces attacking the king, the more likely the player is to lose.

Structure of pawns was also considered as a feature. The structure of pawns can determine how much control the player has, defending its pieces and preventing advancements from the

enemy. The two that were considered was isolated pawns and doubled pawns. Isolated pawns are pawns that do not have any friendly pawns on adjacent files. Usually this is a weak structure since they can't be defended by other pawns and also can be easily blocked by the opponent pieces however they can be considered strong in some cases. This is because they can have more control over the board but 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 considered is the castling rights of the player. 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 which is generally more dangerous. It also allows the rook to have a more active role in the game. Whether the play has castling rights both kingside and queenside are considered.

After collecting all the features, I standardised the data. This is because the features are on different scales so it's important to normalise the data to ensure the Naive Bayes Classifier is trained properly and doesn't give importance to features that are on a larger scale. I utilised the StandardScaler from the scikit-learn library. This is done by the following formula:

$$z = \frac{x - \mu}{\sigma} \tag{5.3}$$

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

This allows all the features to have a mean of 0 and a standard deviation of 1 but also keep the original distribution of the data.

5.6 Naive Bayes Classifier

The Naive Bayes Classifier is a simple classifier that is based on Bayes' Theorem. Bayes' theorem is a fundemental principle that describes how the new evidence can change the probability of an event. Bayes' Theorem is given by the following equation:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{5.4}$$

Where:

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

Posterior probabilty is the probability of event A happening after knowing new evidence B. Likelihood is the probability of evidence B happening given that event A is true. Prior probability is the probabilty of event we intiially know about A before any new evidence.

The first step for implementing Naive Bayes is to calculate the prior probabilities of each class. This is calculated by going through the whole dataset and working out the proportion of each class. This is done by the following formula:

$$P(C) = \frac{N_c}{N} \tag{5.5}$$

Where:

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

The next step is to calculate the likelihood of each feature given the class. Ther are two main type of Naive Bayes classifiers, Gaussian and Multinomial. Multinomial is used for discrete features, calcluating the likelihood based on the count of each feature.

$$P(X|C) = \frac{N_{X,C} + 1}{N_C + V} \tag{5.6}$$

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

However, most of the features used in this project are continuous so Guassian Naive Bayes is more ideal. Gaussian Naive Bayes calculates the likelihood, assuming the features are normally distributed. This is done by the following formula:

$$P(X|C) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{5.7}$$

Where:

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

Then we need to work out the Posterior probability for a class given a set of features $X = x_1, x_2, ..., x_n$ by using Bayes' theorem. Naive Bayes assumes that the features are conditionally independent given the class, which allows us to simplify the equation as the product of the likelihoods. This is done by the following formula:

$$P(c|X) = \frac{P(c) \prod_{i=1}^{n} P(x_i|c)}{P(X)} \tag{5.8}$$

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

Then this is used to predict the class of a new instance by choosing the class with the highest posterior probability for that feature. Since $P(X)$ is the same for each class, it can be removed when comparing the posterior probabilities.

$$P(c|X) \propto P(c) \prod_{i=1}^{n} P(x_i|c) \tag{5.9}$$

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

$$\hat{y} = \arg\max_{c} P(c) \prod_{i=1}^{n} P(x_i|c) \tag{5.10}$$

When doing calculations, since we are working with probabilities which are small numbers and multiplying them can result in very small numbers, causing underflow problems. To prevent this, we can use the log of the probabilities which also converts the multiplications to additions, given by this formula:

$$\log(P(c|X)) \propto \log(P(c)) + \sum_{i=1}^{n} \log(P(x_i|c)) \tag{5.11}$$

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

5.7 Model Training

The Naive Bayes Classifier was then train on the preprocessed data. Due to restrictions on available computing power, only a small subset of the data was used to train the model. The data was split into a training set and testing set, where 80% was used for training and 20% was used to test the model. This is usually the standard way to split data to train the model with the majority of the data but also test it on data it hasn't seen before to see how well it generalises.

5.8 Model Evaluation

After training the model, we are interested in knowing how well the model performs and generalises to new data. There are many ways to evalutate a model. One way to do this is to use a confusion matrix. A confusion matrix is a table that shows how well the model classfied instances. The confusion matrix is shown in Table 5.3.

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive	False Negative
Actual Negative	False Positive	True Negative

Table 5.3: Confusion Matrix

This is a good way to visualise what the model is doing and doesn't remove any information, compared to other evaluation metrics.

Using this confusion matrix, we can calculate the accuracy and precision of the model. Accuracy is defined as the proportion of instances that the model classified correctly. This is

given by the following formula:

$$\text{Accuracy} = \frac{\text{Correctly Classified Instances}}{\text{Total Instances}} \tag{5.12}$$

Where the correctly classified instances is the sum of the True Positives and True Negatives. Accuracy is an easy way to compare models but it generally doesn't handle imbalanced datasets very well. Another metric we can use is precision. Precision is the proportion positive predictions that were correct, given by the following formula:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{5.13}$$

Recall, also known as sensitivity, is the proportion of actual positive instances that were classified correctly, given by the following formula:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{5.14}$$

Usually it is easier to compare models with one metric that considers both precision and recall. F_β score is a metric that is the weighted harmonic mean of precision and recall, given by the following formula:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{5.15}$$

Usually β is set to 1, which is the harmonic mean of precision and recall, which is known as the F1 score. This is the metric that will be used to evaluate the model.

5.9 Model Implementation

The Naive Bayes classfier will then be implemented in the chess engine. There are many ways the classifier can be used to try improve the engine. One way is to completely replace the traditional evalutaion function with the classifier. This would be the simplest way to implement the classifier and will show how independant the classfier is and whether it would need support through other evalution methods. This can be done by using alpha-beta pruning and minimax and once it reaches a leaf node, the board features can be extracted from the board and used by the Naive Bayes Classifier. THe classifer's probability output can be used to determine the best move to make, using the following formula.

$$\text{Score}(X) = P(\text{Winning} \mid X) - P(\text{Losing} \mid X) \tag{5.16}$$

Worst case scenario for alpha-beta pruning is that no moves are pruned. This is usually due to the fact that the moves are not in a good order. To optimise alpha-beta pruning, the good moves should be search first. This can be done by implementing Naive Bayes. For each node, the classfier can output $P(X|Winning)$ and use this to order the moves from best to worst. This can help the alpha-beta algorithm to prune more nodes, which can speed up the search time and also allow the algorithm to search deeper.

One last way the classifier can be implemented is by using it with the support of traditional evaluation functions, which consider material balance and piece positions. The classifier can be used to give $P(X|Winning)$ and this can be used to calculate a weighted sum of both evaluations.

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 Section Heading

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

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

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

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

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

[9] 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.

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

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

[12] A. Gupta, C. Grattoni, and A. Gupta, "Determining Chess Piece Values Using Machine Learning," vol. 12, no. 1.

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.