School of Electronic Engineering
and Computer Science

Final Report

Programme of study:
Computer Science

Project Title:
Elo Prediction: Predictive Analytics for White Piece Chess Performance

Supervisor:
Dr. Søren Riis

Final Year
Undergraduate Project 2023/24

Student Name:
Ha Quang Minh An

Queen Mary University of London

Date: 25/04/2024

Abstract

The Elo rating system is an essential measure for assessing the skill levels of chess players. However, this system only considers the game's output and ignores the players' style. Most players often require numerous years and matches to achieve an Elo rating that reflects their skill level. Therefore, this project aims to find the best approach to train a model to predict a player's Elo rating from a single chess game. This project helps players, coaches, and enthusiasts of the game better understand strategic proficiency in making moves and decisions in chess.

In this paper, I will look at different ways to capture players' insight from a single chess game, introduce the previous works that have been done, and investigate the most effective way for feature selection with the utilisation of the state-of-the-art chess engine, Leela Chess Zero. The methodology section of this paper will examine various supervised learning methodologies and the application of recurrent neural networks (RNNs) for sequential prediction. Moreover, the model will be integrated into a web application where the user can interact with the model via the GUI. The users can play with the white pieces against the strongest chess engine, Stockfish 16, which was integrated within the web application with Elo set at 2400. The model can predict up to 43 moves with an error of 93 Elo. This is a reasonable figure due to the high complexity of chess.

Acknowledgement

I want to thank my supervisor, Dr. Søren Riis, for his continued support, motivation and guidance throughout the completion of the project.

Contents

Chapter 1: **Introduction**

1.1 Background

The Elo rating system created by Arpad Elo is an internationally recognised mathematical way of assessing a chess player's skill based on their game results. By using ratings to represent a player's skill level, the Elo system enables us to predict the outcome of matches and compare players' abilities against each other. These ratings are critical to creating rank lists, matchmaking, and tracking players' progress.

Currently, the ratings are determined using only the final result of a match - win, lose, or draw. Additionally, most players often require numerous years and matches to achieve an Elo rating that reflects their skill level. So, it might be more valuable to have a method to predict a player's Elo from a single chess game. There has been interest in developing new methodologies for determining a player's Elo rating from a single match, mainly reflective of advances in machine learning and big data, which allow the decomposition of chess strategies into smaller parts and understanding how machine learning can extract principal behaviours in chess games. It is now possible to analyse a single move and decompose the complexity of an Elo rating into underlying structures and features to assess player skills. Every move impacts the progression of chess strategies and represents a player's decision to achieve some part of the overall goal. Even if only a few moves were recorded from a long game, we still can gain insights into the players' skills.

1.2 Problem Statement

The primary challenge of this project is to construct a predictive model that utilises data from numerous historical chess games to estimate a player's Elo rating based on a limited sequence of moves. This attempt extends beyond traditional rating systems by seeking to capture the essence of a player's strategy and skill from a micro-perspective.

The dataset, comprising only a limited number of moves, offers unique analytical challenges. It demands a deep understanding of chess strategies and the capacity to identify important patterns from limited data. The model must effectively navigate the complex terrain of chess tactics and strategies, transforming a sequence of moves into a reliable estimate of a player's strength.

1.3 Aim

This project seeks to improve the application and accuracy of the Elo rating system, providing a dynamic tool for evaluating player abilities. By achieving this goal, the research will enhance our comprehension of chess expertise and establish an innovative framework for evaluating players, which can be applied in coaching and game analysis.

1.4 Objectives

Create a predictive model that can reliably predict the value of an Elo rating for a chess player based on the moves made by that player while playing with the white pieces in a single game.

Analyse the sequence of moves made by the player to gain insights into their strategic thinking and decision-making processes.

Evaluate the relationship between the player's number of moves and the accuracy of the Elo rating predictions, establishing the optimal conditions for predictive accuracy.

Enhance the understanding of how different levels of the opponent's Elo affect the predictive accuracy of the model.

Create a user-friendly web application that allows players to input their game data and receive instant predictions on their potential Elo ratings.

Chapter 2: **Literature Review**

2.1 State of the Art in Elo Rating Prediction

2.1.1 Innovation from Chess.com

Chess.com has also recently developed an interesting post-game Elo estimator that has been seen as an excellent predictor of actual post-game Elo ratings. One concern, discussed on the Chess.com site these last days by several players, is whether their Elo estimator is making allowances not simply for the quality of the moves played but also considering the opponent's Elo rating. From the observations of some players, this Elo estimator predicted significantly different Elo ratings in the same sequence of moves, depending on whether the opponent consists of a stronger or weaker bot. This would seem to imply that the Chess.com algorithm is adaptive. If one plays the same moves against a much higher-rated bot than a lower-rated one, the estimated Elo based on the same moves in a given game can be much higher than the estimated Elo resulting from the same moves in a similar match against a low-rated opponent. Potentially, some elements of comparative performance analysis might be integrated into its prediction logic.

2.1.2 Insights from Previous Work

A recent paper explored Elo prediction modelling with machine learning (Jevgenijus, Oisin and Anton, 2021). The model was explored using several techniques, such as k-nearest neighbours (KNN), Ridge Regression, Lasso Regression, ElasticNet Regression, and basic linear regression. Mean Squared Error (MSE) is used to assess the model's performance. The kNN model performs the best with the game2vec feature representation, edging out all other models with training MSE 9177 and test MSE 48247. The severely skewed MSE value of training versus the test indicates overfitting, a common problem in machine learning where the model has memorised the training data set very well but fails miserably trying to apply its knowledge to unseen data.

These results indicate the difficult task of predicting Elo ratings by machine learning methods and the crucial role of choosing suitable features in addition to a model's ability to learn from historical data and be used on new data points. The authors themselves comment that their models are unlikely to be suitable for practical use due to the performance limitations and that further study on using deep learning methodologies on a much larger dataset would be an open research question for future work (Jevgenijus,

Oisin and Anton, 2021). In that paper, the minimum value of the test MSE was 48247, which corresponds to a margin of error of 219.65 Elo points. A 200-point difference in Elo ratings is considered significant, as it usually implies a substantial difference in player ability and style of play. This means that existing predictive accuracy might not be sufficient to account for the subtle but significant differences in the skill of players.

2.1.3 Evaluation Methodologies in Chess Performance Analysis

Dr. Søren Riis critically examines the methodology used in the study "Computer Analysis of World Chess Champions" by Matej Guid and Ivan Bratko, which attempts to determine the greatest chess player of all time by comparing players' moves against those recommended by a computer program, specifically a modified version of Crafty. Dr. Søren argues that the approach, which measures how closely a player's moves align with those of a lower-rated chess engine, might not effectively reflect true player skill.

Firstly, Dr. Søren points out that using a relatively weak engine like Crafty (rated no more than 2700) to analyse the moves of world champions who often exceed this rating leads to questionable conclusions. The engine's limited ability means it might not fully comprehend the depth and subtleties of high-level play, particularly in complex positions favoured by players like Kasparov. For example, Crafty penalised well-known winning moves by Fischer and Kasparov due to their inability to understand their strategic depth, suggesting that the analysis heavily depends on the engine's capabilities and might underestimate players who employ highly sophisticated strategies.

Moreover, the review discusses the "horizon effect" where the engine fails to evaluate positions that extend beyond its calculation depth, further complicating its ability to judge the quality of moves made in longer sequences that unfold beyond its computational reach. This critique highlights significant flaws in using this method to rank player skill accurately, as it tends to favour those whose style aligns more closely with the engine's algorithm rather than those who might strategically plan beyond its analytical capabilities.

Dr. Søren's analysis provides a compelling argument that such computer-based assessments might misrepresent the true abilities of world-class chess players and suggests a cautious approach to using automated systems for ranking or rating players based on their gameplay. This perspective is crucial for understanding the limitations of current technologies in sports analytics and chess and underscores the need for more robust systems that can appreciate the complexities of top-level chess.

2.2 Elo Rating System

2.2.1 Origins and Evolution

The Elo rating system addresses the need for a standardised method to rate player skills in competitive environments, particularly chess. Originating as a solution to the limitations of previous systems like the Harkness system, the Elo rating algorithm has evolved from its initial chess application to broader uses in various sports and competitive activities. The system's simplicity and adaptability have driven its widespread adoption and evolution over the years.

2.2.2 Mechanics of the System

I will introduce the rating model with fundamental concepts

Elo Rating Calculation:

- **Basic Formula**: The Elo system calculates ratings based on the outcome of games about expected results:
  - R' = R + K x (S - E)

    Where:
    - R' = New Rating
    - R = Current Rating
    - K = Weight factor determining the maximum possible adjustment per game (K-factor)
    - S = Score achieved in the game (1 for win, 0.5 for draw, 0 for loss)
    - E = Expected score against the opponent
- **Expected Score Calculation**:
  - E = $\frac{1}{1+10^{(R_{opponent}-R)/400}}$

    Where: $R_{opponent}$ is the opponent's rating.
- **K-Factor Variability:** The K-factor determines how volatile a player's rating is: a player's rating changes more radically per game with a higher K-factor. For example, players with relatively few games played or those with low ratings tend to have higher K-factors than others.

The system calculates player ratings based on game outcomes relative to expected performance, adjusting ratings after each game. The algorithm's dynamic and probabilistic nature allows it to continually update player ratings, reflecting their current skill levels more accurately than static systems.

2.3 Forsyth-Edwards Notation (FEN)

Forsyth-Edwards Notation (FEN) uses a single line of text to encode information about which piece is on which square on a chessboard. This notation is fundamental to describing the status of the board in detail, and it is important in the analysis and archiving of games as well as for setting up particular positions on the board for further examination (Chess.com, 2020).

2.3.1 Historical Development

The notation had been developed by David Forsyth, a Scottish journalist, but subsequently revised by Steven Edwards in order to fit computerised game analysis and finding its digital-age usage in chess environments. This adaptability has made it in essence in any chess software in the world. (Chessprogramming.org, 2020).

2.3.2 Structure of FEN

The FEN combines the chess position in its exact form, using 6 major fields: pieces placement, active colour, castling availability, en passant target square, half move count for the 50 moves rule and the number of full moves. This detailed encapsulation allows to resume a game immediately after a pause, especially in educational purposes or masters analysis (Chess.com, 2020).

2.3.3 Role in Modern Chess and AI

FEN's utility is evident in the large amount of AI research and software development that relies on being able to quickly and accurately initialise complex game positions at runtime: in program suites such as Chess.com, which allows users to save, display and analyse games using FEN, or in machine learning applications, where the game positions have to be randomly drawn and studied en masse by AI to help it hone its own strategic algorithms (Chessprogramming.org, 2020).

2.3.4 Conclusion

Forsyth-Edwards Notation is still essential to the analysis of chess and the general application of AI to the problem of strategy in games. By standardising the format in which any game state can be presented, FEN bridges traditional chess with contemporary computational methods by providing a concise format to represent any game state, proving its enduring relevance and utility.

2.4 Machine Learning

2.4.1 Supervised Learning

Supervised learning models are foundational to predictive analytics and machine learning, used extensively in fields ranging from automated rating systems like Elo to strategic game analysis. This section delves into several key supervised learning techniques: Linear Regression, Decision Tree, Random Forest, and Gradient Boosting, each of which offers unique advantages depending on the application.

2.4.1.1 Linear Regression

Linear Regression is one of the simplest and most extensively used statistical techniques for predictive modelling. It assumes a linear relationship between the dependent and independent variables, which can be represented by a straight line fit through data points. This model is particularly effective for understanding relationships between variables and forecasting:

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n + \epsilon$$

Where Y is the dependent variable, $X_1, \ldots, X_n$ are independent variables, $\beta_0, \beta_1, \ldots, \beta_n$ are coefficients, and ϵ is the error term.

2.4.1.2 Decision Tree

Decision Trees are a non-linear form of supervised learning that can be used for classification and regression tasks. The model uses a tree-like graph of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It's particularly useful for its interpretability and the ease with which it handles categorical features.

2.4.1.3 Random Forest

Random Forests build on decision trees through ensemble learning, where multiple trees are generated and aggregated to improve the model's accuracy and robustness. This method addresses overfitting problems common to single decision trees and is effective across a broad range of applications. It is known for its high accuracy, importance ranking of variables, and robustness to outliers and noise.

2.4.1.4 Gradient Boosting

Gradient Boosting is a powerful ensemble technique that builds models sequentially, each new model correcting errors made by previously trained trees. The algorithm

combines weak predictive models to create a strong model in a stage-wise fashion. It is highly flexible and can optimize different loss functions. It also provides several hyperparameter tuning options that can make the model robust to overfitting.

2.4.2 Deep Learning

Deep learning, a subset of machine learning, utilises neural networks with multiple layers to analyse various data forms, including images, sound, and text. Its impact is particularly notable in the realm of chess, where it has been employed to predict outcomes, analyse positions, and evaluate player strengths, such as Elo ratings.

2.4.2.1 Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU)

LSTM and GRU are advanced types of recurrent neural networks designed to effectively handle sequence prediction problems by capturing long-term dependencies. This capability is crucial for complex decision-making tasks like chess, where understanding sequences of moves and their outcomes can significantly enhance predictive accuracy.

Hochreiter and Schmidhuber (1997) introduced LSTM networks, which are designed with unique 'gates' that regulate the flow of information. These gates ensure that LSTMs can retain important historical data over long sequences without suffering from the vanishing gradient problem, making them ideal for complex sequence prediction tasks such as those in chess analysis.

GRU simplifies the LSTM architecture by combining the input and forget gates into a single "update gate," and by using a single state to represent both the cell state and hidden state. This reduction in complexity allows GRUs to perform more efficiently on smaller datasets. GRUs have been found to outperform LSTMs on tasks with lower complexity sequences but perform comparably on higher complexity tasks (Cahuantzi et al., 2021)

LSTM and GRU can be instrumental in tasks such as evaluating player strengths and predicting game outcomes. The choice between using LSTM or GRU could depend on the complexity of the task at hand. For instance, LSTM might be preferable for analysing games involving high-level players due to the intricate strategies and longer-move sequences that characterize their games. On the other hand, GRU could be more suitable for quicker predictions in less complex scenarios.

2.4.2.2 Optimizer

Optimizers in machine learning are algorithms or methods used to change the attributes of the neural network, such as weights and learning rate, in order to reduce the losses

and improve the accuracy of model predictions. This is the equation for the Adaptive Movement Estimation (ADAM) optimizer (Ruder, 2016):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Where θ_t is weight, n is learning rate, v is uncentered variance and m is mean.

2.4.3 Error Calculations

Error metrics are crucial for evaluating the accuracy of predictive models in machine learning, including applications in rating systems like Elo and AI-driven game analysis. Three common metrics are Mean Squared Error (MSE), Mean Absolute Error (MAE), and the Coefficient of Determination (R^2). These metrics provide insights into models' prediction errors and are essential for optimizing algorithms.

2.4.3.1 Mean Squared Error (MSE)

Mean Squared Error (MSE) is a widely used metric for quantifying the accuracy of a model. It calculates the average of the squares of the errors—the average squared difference between the estimated values and the actual value. MSE is particularly useful in scenarios where large errors are potentially more harmful than smaller ones. It is mathematically represented as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Where Y_i is the actual value and $\hat{Y}_i$ is the predicted value from the model. A significant advantage of MSE is its differentiability, which allows it to be efficiently used in optimization algorithms (Ruder, 2016).

2.4.3.2 Mean Absolute Error (MAE)

Mean Absolute Error (MAE) measures the average magnitude of the errors in a set of predictions, without considering their direction. It's calculated as the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$$

MAE is particularly robust to outliers and provides a more intuitive measure of average error magnitude compared to MSE (Willmott and Matsuura, 2005).

2.4.3.3 Coefficient of Determination (R^2)

The Coefficient of Determination, denoted as R^2, is a key performance metric that provides insights into the amount of variance in the dependent variable that can be explained by the independent variables in the model. It is an indicator of goodness of fit and, thus, a measure of how well-unseen samples are likely to be predicted by the model through the proportion of total variation of outcomes explained by the model.

$$R^2 = 1 - \frac{Sum\ of\ Squares\ of\ Residuals}{Total\ Sum\ of\ Squares}$$

2.5 Leela Chess Zero (Lc0)

Leela Chess Zero (Lc0) represents a community-driven initiative to develop a chess engine based on the principles of deep learning and reinforcement learning, inspired by DeepMind's AlphaZero. Unlike AlphaZero, which was not released to the public, Lc0 is an open-source project, allowing enthusiasts, developers, and researchers worldwide to contribute and access its technology. By leveraging the neural network architecture and self-play methodology similar to AlphaZero, Lc0 has achieved significant milestones in the realm of computer chess, offering a new paradigm that combines human ingenuity with machine learning to advance the understanding and development of chess strategies (Dominik Klein, 2020).

The development of Lc0 has been marked by continuous refinement and innovation. Its neural network structure has evolved, integrating features like the Squeeze and Excitation blocks, distinct from AlphaZero's ResNet blocks, to enhance its learning efficiency. A unique aspect of Lc0 is its three-headed network architecture, which includes a pioneering "moves left" head, designed to predict the number of moves until the game concludes, thereby enriching its decision-making process. This feature exemplifies the engine's innovative approach to integrating deep learning in chess, demonstrating the potential of collaborative, open-source projects to push the boundaries of artificial intelligence and its applications in strategic games like chess (Dominik Klein, 2020).

Chapter 3: **Methodology**

3.1 Data Collection

3.1.1 Data source

The raw data for this project came from Lichess game database records from 2013 in the form of Portable Game Notation (PGN) files. However, for this project, the data used is a subset of this huge dataset, including only 7,000 games, totalling about 450,000 chess positions. All classical games are included, whereas bullet and blitz games were excluded. The reason why they were excluded is that due to time pressure, many moves are made that don't reflect the skill of the players, which might obscure the true Elo ratings of the players.

3.1.2 Feature Extraction

The raw PGN data was processed to extract and generate several features relevant to the analysis. See Appendix A for an example of 1 chess game in PGN. Each move of each game is stored in the data table as each row. These features are extracted from the PGN format:

- **Game_ID**: Unique identifier for a game.
- **FEN (Forsyth-Edwards Notation)**: A string that represents the current state of the chessboard.
- **Move_Count**: Total number of moves played in a single chess game.
- **TimeControl**: The time control setting of the game.
- **Elo**: The Elo rating of the player.
- **Move**: The move made in standard algebraic notation.

These features are extracted from the FEN string:

- **Piece_Count**: The total number of pieces on the board.
- **Material_Balance**: A metric indicating the material advantage or disadvantage in terms of pieces.
- **Number_Of_Legal_Moves**: The number of legal moves available.
- **Double_pawns**: The number of double pawns for both the players
- **Isolated_pawns**: The number of isolated pawns for both the players
- **Passed_pawns**: The number of passed pawns for both the players

These features are extracted using the state-of-the-art **Leela Chess Zero (Lc0)** engine:

- **Move_Quality**: Quality of the move is evaluated by Lc0 engines of varying sizes.
- **Position_Evaluation**: Evaluation of the position by Lc0 engines of varying sizes.

To make the **Position_Evaluation** more accurate, three different Lc0 network sizes were used, where each provides a different insight into the position of the chess game. A multiple-network approach results in a more robust **Position_Evaluation** and a better

overall evaluation of any given chess position. It can capture strategic insight and tactical assessments that are better reflected by assigning different networks different weights in consideration of players' unique strengths and playing styles.

Network Size	Purpose	Filters	Blocks	GPU Memory Usage	File Size
Large	GPU	768	15(mish activation)	2.4 GB	160-170 MB
Medium	GPU/CPU	512	15(mish activation)	1.8 GB	140-150 MB
Small	GPU/CPU	256	10(mish activation)	1.6 GB	30-40 MB

In addition, **Move_Quality** measures the changes in the position's evaluation due to a player's move, which is calculated by the difference between the position's evaluation after the move and the evaluation just before the move.

Here is the snipper of a code on how to use Lc0 to extract the features:

```
from lczero.backends import Weights, Backend, GameState

# Load LCZero
large_net_size = Backend(weights=Weights('/content/drive/MyDrive/Colab_Notebooks/model/Lc0_networks/768x15x24h-t82-2-swa-5230000.pb'), backend='cudnn')
medium_net_size = Backend(weights=Weights('/content/drive/MyDrive/Colab_Notebooks/model/Lc0_networks/t1-smolgen-512x15x8h-distilled-swa-3395000.pb'), backend='cudnn')
small_net_size = Backend(weights=Weights('/content/drive/MyDrive/Colab_Notebooks/model/Lc0_networks/t1-256x10-distilled-swa-2432500.pb'), backend='cudnn')


# Initialize a cache for storing evaluations of FEN strings
fen_evaluations_cache = {"large": {}, "medium": {}, "small": {}}

def LCZERO_Eval_with_cache(fen, name, network):
    # Check if the FEN has already been evaluated
    if fen in fen_evaluations_cache[name]:
        return fen_evaluations_cache[name][fen]

    # If not in cache, evaluate the position and store the result in the cache
    input_data = GameState(fen=fen).as_input(network)
    evaluation = network.evaluate(input_data)[0].q()
    fen_evaluations_cache[name][fen] = evaluation

    return evaluation

def move_quality_with_cache(fen_before, move):
    move_qualities = {}
    position_evaluation = {}

    for name, network in [("large", large_net_size), ("medium", medium_net_size), ("small", small_net_size)]:
        # Evaluate the position before the move, using the caching version of LCZERO_Eval
        pre_move_evaluation = LCZERO_Eval_with_cache(fen_before, name, network)
        position_evaluation[name] = pre_move_evaluation

        # Play move
        board = chess.Board(fen_before)
        board.push(move)
        fen_after = board.fen()

        # Evaluate the new position after the move
        post_move_evaluation = LCZERO_Eval_with_cache(fen_after, name, network)

        # Calculate the quality of the move
        move_quality =  - post_move_evaluation - pre_move_evaluation

        move_qualities[name] = move_quality

    return move_qualities, position_evaluation
```

Figure 1: Extract features using the Lc0 engine

The **TimeControl** feature was removed from the dataset because the data only included chess games in classical time control. Eliminating this feature sharpens the attention on

the most important features of the games, resulting in a more focused and efficient analysis process.

The final cleaned dataset now contains around 230,000 chess positions from 3,400 chess games, which will be used in the next step to predict Elo scores from the extracted features and game result.

3.1.3 Exploratory Data Analysis

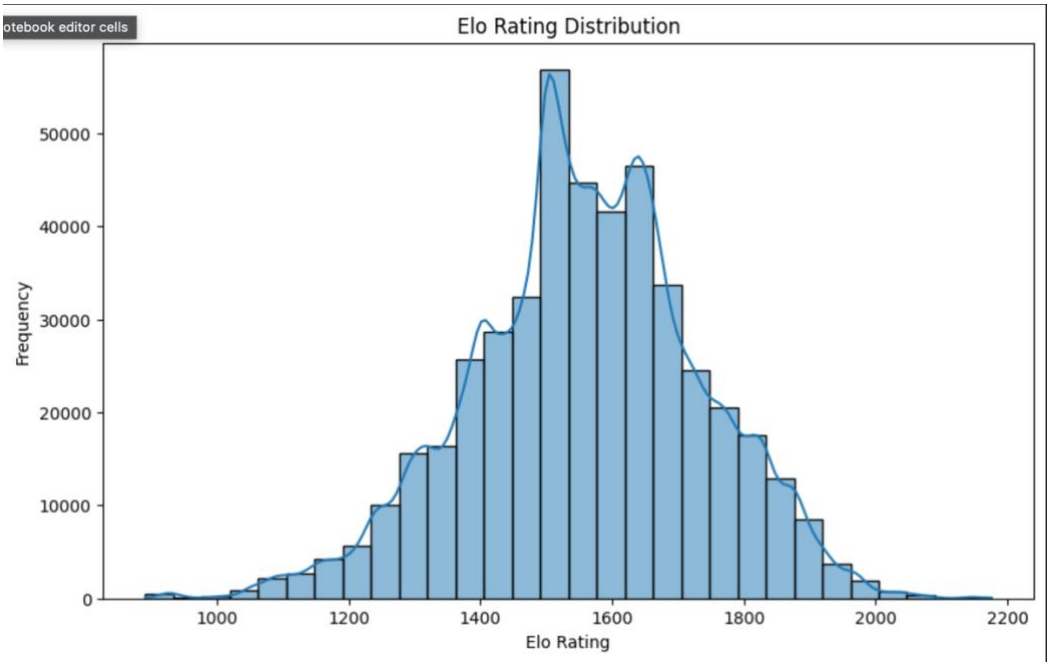


Figure 2: Elo Rating Distribution

The Elo rating distribution demonstrated in the histogram represents a wide range of chess-playing abilities, ranging from 891 to 2176. A significant concentration of players, as reflected by the histogram's peaks, exists between ratings of 1400 and 1700. This clustering indicates that the bulk of the dataset contains players who are neither beginners nor experts, resulting in a representative sample of intermediate-level performance. As a result, the prediction model created from this data is correctly tuned for players in this rating range. The model's utility may be most evident for individuals with Elo ratings between 1000 and 2000, where it can provide precise and informative assessments that reflect the majority of the dataset's playing strength.

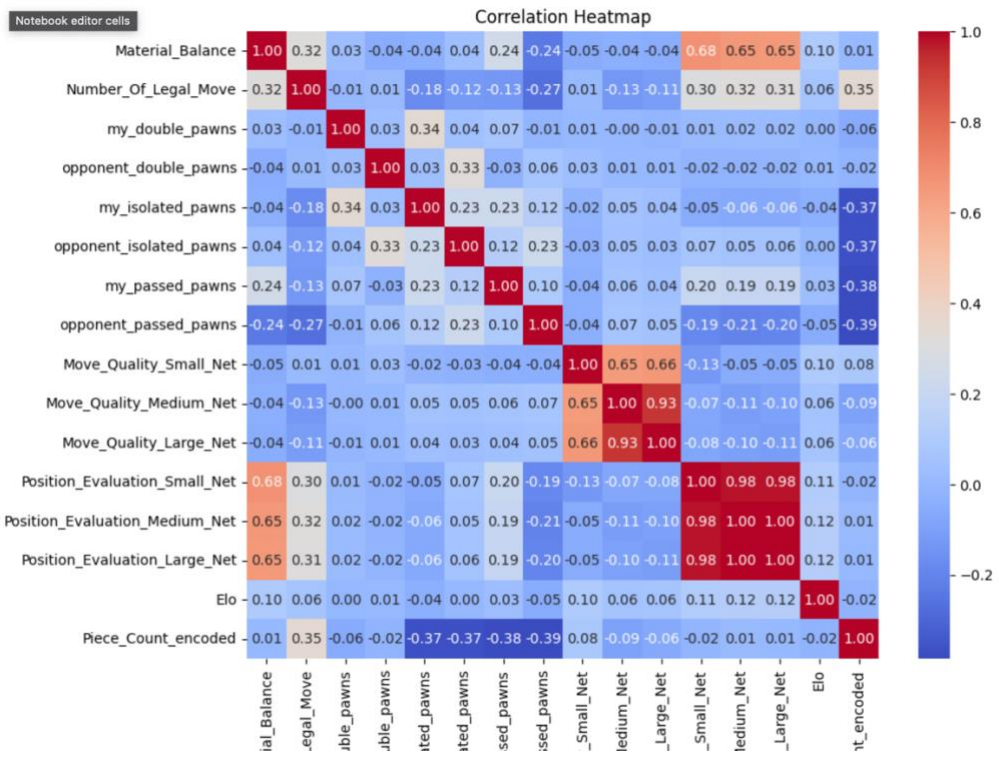


Figure 3: Correlation Heatmap of Numerical Features in data table

The correlation heatmap displayed provides a visual representation of the relationship between various features relevant to chess analysis. Notably, the **Material_Balance** feature shows a strong positive correlation with the **Position_Evaluation** features across Small, Medium, and Large networks, as indicated by the high correlation coefficients (ranging from 0.65 to 0.68). This suggests that material balance is a significant predictor of the position's evaluation, with higher material balance generally associated with more favourable positions.

The **Move_Quality** features, while measured across different network sizes, show moderate to strong correlations with each other (values between 0.65 and 1.00), which implies consistent move quality assessment across different Lc0 network sizes. Interestingly, there is a minimal correlation between **Move_Quality** and Elo ratings, indicating that the quality of individual moves as evaluated by Lc0 networks might not linearly correspond with the players' Elo ratings. In contrast, the **Piece_Count_encoded** feature seems to have a negligible relationship with most features except **Number_Of_Legal_Move**, where there's a moderate positive correlation, suggesting that the number of pieces on the board has some influence on the legal moves available.

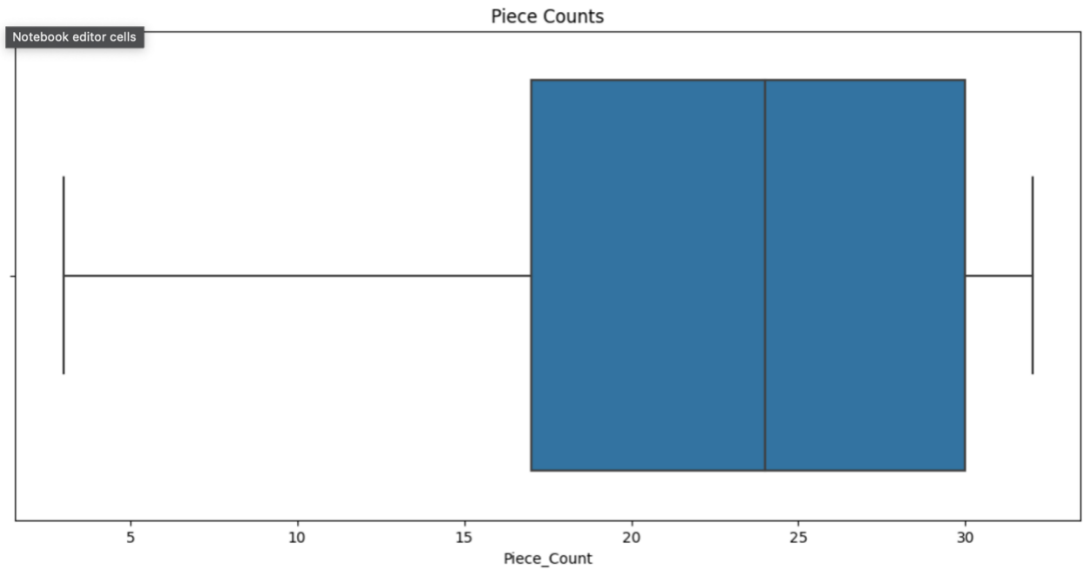


Figure 4: Distribution of piece count for all position in the data table

The **Piece_Count** histogram illustrates the distribution of total pieces on the board across various stages of chess games, with counts ranging from approximately 3 to 32 pieces. To ensure the accuracy of the model, the dataset is categorised into opening, mid-game, and end-game phases based on piece count. The first quartile (**Q1**) cut-off is set at 17 pieces, meaning any position with fewer pieces will be classified as an endgame. Mid-game is defined as positions with piece counts falling between the first quartile (**Q1** = 17) and the third quartile (**Q3** = 30). Positions with a piece count exceeding 30 are considered to be in the opening phase. This categorisation is critical for the model to make precise evaluations based on the game phase, which is a significant factor in the complexity and strategy of a chess position.

3.2 Training

The problem is considered a regression problem. First, I utilised four models from supervised learning to have an overall view of the dataset; the model will try to predict the player's Elo based on only one move. Second, I utilised the sequence model to predict the Elo of the player based on the sequence of moves, so it would be more accurate than predicting Elo from just one move.

3.2.1 Supervised Learning

3.2.1.1 Model Selection

A suite of regression models was applied to ascertain the predictive power of engineered features on Elo ratings. The selection was influenced by a desire to contrast the performance of both linear and tree-based models.

1. **Linear Regression**: We started with linear regression due to its simplicity, interpretability, and quick computation. It assumes a linear relationship between the input features and the target variable, making it a good baseline for comparing with more complex models.
2. **Decision Tree Regressor**: This model was selected for its ability to capture non-linear relationships through a series of binary decisions made on the feature values. It is easy to interpret and can model complex relationships within the data without needing any feature scaling.
3. **Random Forest Regressor**: An ensemble of Decision Trees, this model builds multiple trees and merges their outputs to improve prediction accuracy and control overfitting. Random Forests are known for their high performance on a wide range of problems, making them a valuable addition to our model suite.
4. **Gradient Boosting Regressor**: As another ensemble technique, Gradient Boosting builds trees sequentially, with each tree attempting to correct the errors of its predecessor. It often delivers superior predictive accuracy and is robust to overfitting, especially with appropriate hyperparameter tuning.

The performance of these models will be compared not just on their predictive accuracy but also on their ability to generalise to unseen data, evaluated through various performance metrics, such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared values. This comprehensive evaluation will help us select the most effective model for predicting chess players' Elo ratings.

3.2.1.2 Training

The dataset was processed by discarding non-essential features and defining the Elo rating as the target variable. A train-test split was performed, allocating 80% of the data for training and 20% for testing, with a random state set for reproducibility. The training phase involved fitting four regression models—Linear Regression, Decision Tree, Random Forest, and Gradient Boosting—to the training data. These models were chosen for their varying approaches to regression, which range from simple linear methods to complex ensemble techniques that can capture non-linear relationships in

the data. The ‘sklearn’ library in Python was utilised to implement these models, taking advantage of its comprehensive suite of built-in functions for machine learning.

Each model was then fitted to the training data. Following the training, these models underwent evaluation on the test set to assess their predictive accuracy. The specifics of their performance, including key metrics and detailed analysis, are presented in the results section of this report.

Here is the pseudocode illustrating this process:

```
For each model in [Linear Regression, Decision Tree, Random Forest, Gradient Boosting]:
    Fit model on training data
    Predict on training data
    Calculate training metrics: MSE, MAE, R²
    Predict on test data
    Calculate test metrics: MSE, MAE, R²
    Store results
```

3.2.1.3 Result

Table 1: Training a single move

Algorithm	Train MSE	Test MSE	Train MAE	Test MAE	Train R2	Test R2
Linear Regression	29813.0837	29793.7513	136.4114	136.3181	0.0310	0.0282
Decision Tree	1993.5256	53278.1893	10.0875	179.1077	0.9352	-0.7379
Random Forest	5561.28795	27871.1165	54.6307	131.1926	0.8192	0.0909
Gradient Boosting	28795.6063	28846.1262	134.0747	134.1185	0.0640	0.0591

These are graphs that compare the fit of the model by showing the relationship between the actual values and the predicted values:

Linear Regression:

Showed limited predictive power with a Mean Squared Error (MSE) of 29793.7513 on the test set and a corresponding Mean Absolute Error (MAE) of 136.3181. The R-squared values for the training and test sets were 0.0310 and 0.0282, respectively, indicating that only a small portion of the variance in the Elo rating was captured by the model.

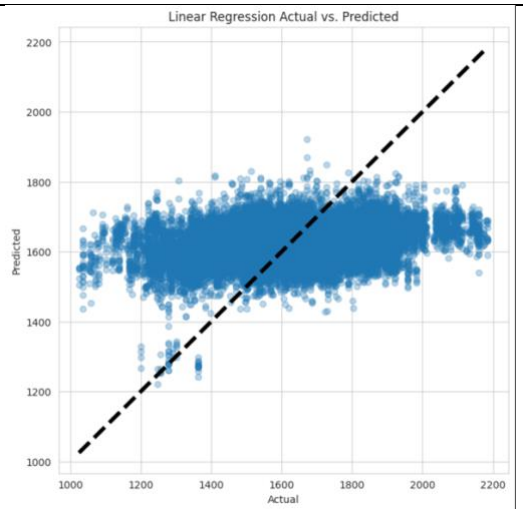


Figure 5: Linear Regression graph

Decision Tree Regressor:

Demonstrated a significant disparity between training and test performance, with an excellent Train MSE of 1993.5256 and Train R2 of 0.9352, but performed poorly on the test set with a Test MSE of 53278.1893 and a negative Test R2 of -0.7379. This suggests severe overfitting, as the model failed to generalize beyond the training data.

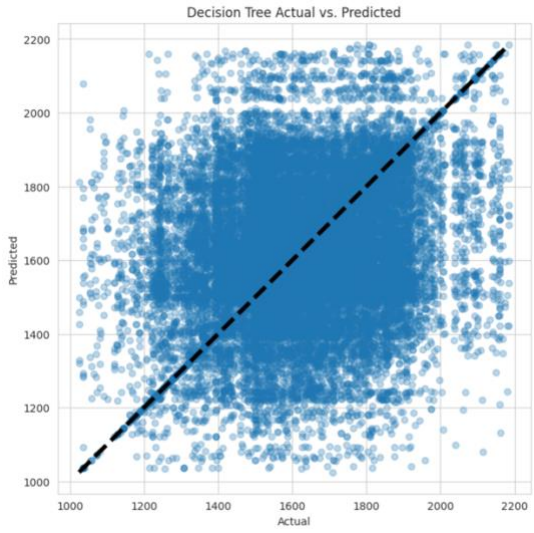


Figure 6: Decision Tree graph

Random Forest Regressor:

Offered an improvement over the single Decision Tree, reducing overfitting as reflected in the Train MSE of 5561.28795 and Test MSE of 27871.1165. However, with a Test R2 of 0.0909, the model still struggled to explain the variability in the test data effectively.

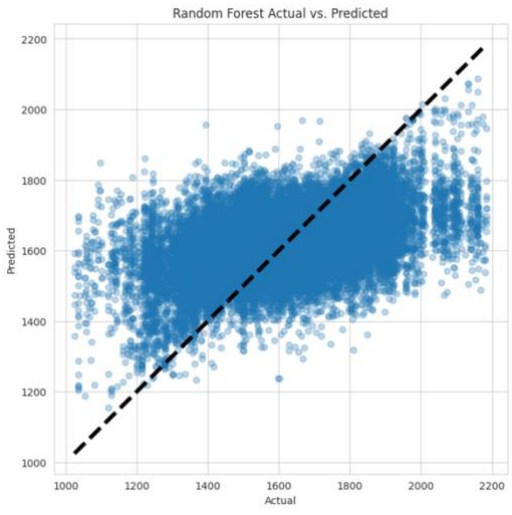
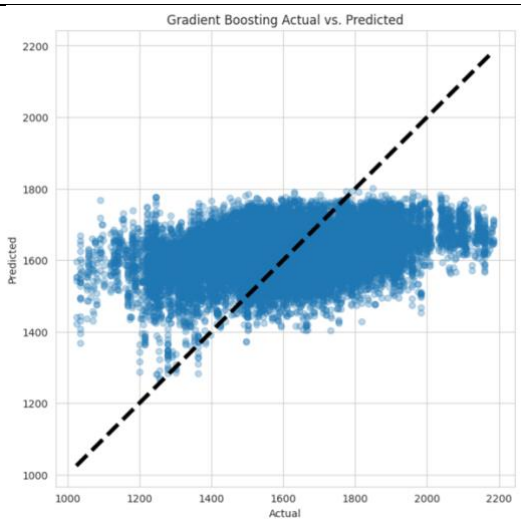


Figure 7: Random Forest graph

Gradient Boosting Regressor: Displayed consistent performance across the training and test sets, with MSEs of 28795.6063 and 28846.1262, respectively, and an MAE of roughly 134 for both. However, similar to linear regression, the R-squared values were relatively low, at 0.0640 for training and 0.0591 for testing, indicating modest predictive ability.	 *Figure 8: Gradient Boosting graph*

These results indicate that while tree-based models have the potential to capture complex patterns in the data, they require careful tuning to prevent overfitting. The ensemble methods, particularly Random Forest and Gradient Boosting, demonstrated a better balance between bias and variance but still showed room for improvement in model performance. The challenge moving forward is to refine these models or explore alternative approaches to improve their ability to predict Elo ratings accurately.

3.2.2 Recurrent Neural Network (RNN)

3.2.2.1 Model Selection

In our pursuit to analyse the data's temporal dependencies and sequential patterns, we selected Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models, renowned for their efficacy in sequence prediction tasks. Here's a detailed overview of why these models were chosen:

1. **Long short-term memory (LSTM)**

   LSTMs are an advanced type of Recurrent Neural Network (RNN) capable of learning order dependence in sequence prediction problems. This model was chosen for its ability to remember information for long periods, which is crucial in capturing the long-term dependencies within the data. Its unique architecture, featuring forget, input, and output gates, allows it to mitigate the vanishing gradient problem common in standard RNNs. This makes **LSTM** particularly suitable for our analysis, where understanding long-term sequential patterns is vital.

2. **Gated Recurrent Unit (GRU)**

GRU is a variation of LSTM designed to simplify the model architecture while maintaining similar performance. It combines the forget and input gates into a single update gate, reducing the complexity and computational overhead. The GRU model was selected to assess whether this simplified structure could provide a similar or improved performance compared to the more complex LSTM, especially in contexts where computational efficiency is a priority. GRU's ability to capture dependencies at different time scales makes it a compelling choice for our analysis.

The evaluation of these models will extend beyond mere accuracy metrics. We plan to analyse their ability to generalise on unseen data through rigorous cross-validation techniques. Performance metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and the coefficient of determination (R-squared) will be employed to assess and compare the predictive capabilities of LSTM and GRU models comprehensively. This thorough evaluation will guide us in selecting the optimal model for our specific sequence prediction task.

3.2.2.2 Training

To enhance the analysis's depth and precision, the dataset was carefully divided into two subsets, one representing all moves executed by white pieces and the other by black. This division is pivotal in chess analytics, as it acknowledges the inherent strategic differences between the two sides. White, having the advantage of the first move, often dictates the game's early pace and strategy, while black responds and adapts, forming its strategy in reaction. By segregating the data into white and black moves, the study aims to isolate and scrutinise the unique strategies, tendencies, and decision-making processes characteristic of players in each colour. This nuanced approach allows for a more detailed and granular analysis, providing insights into how the colour of the pieces influences player strategies and outcomes. By training separate models for white and black moves, the project aspires to uncover colour-specific patterns and strategies, offering a more comprehensive understanding of the gameplay dynamics in chess. This methodological partitioning is crucial for dissecting the complex interplay of strategy in chess, enabling a more targeted and refined analysis that can lead to more accurate and insightful predictive modelling.

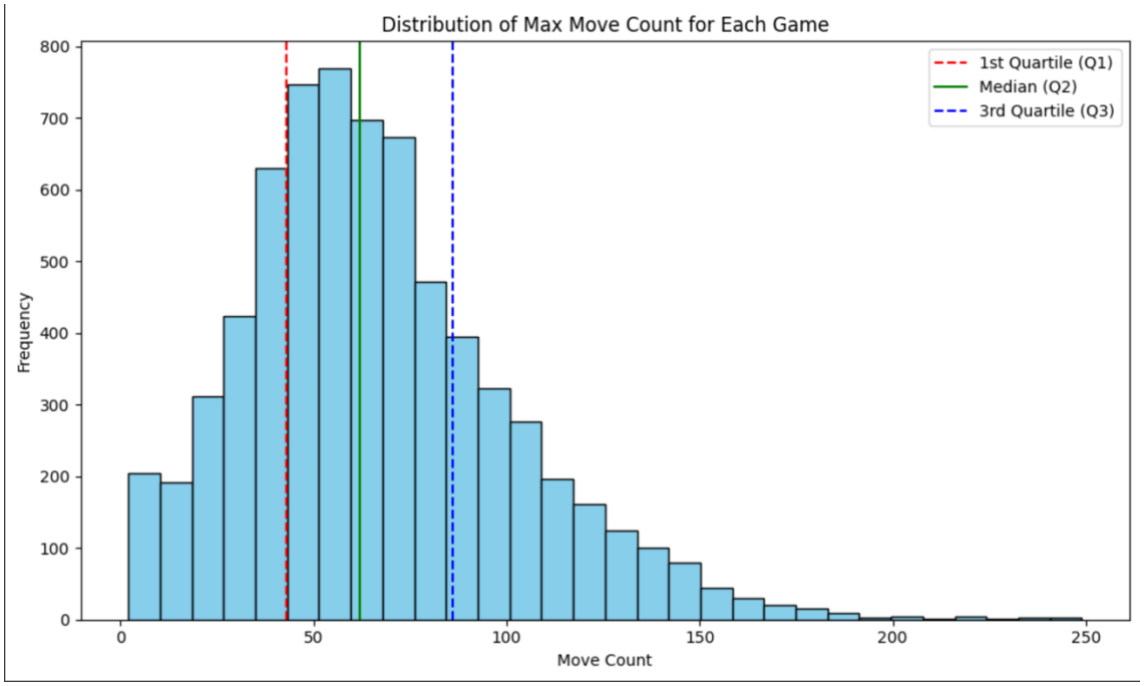


Figure 9: Distribution Count for Moves for all games

The dataset underwent a rigorous cleaning process. We grouped the data by **'Game_ID'** and calculated the maximum **'Move_Count'** for each game. This step was crucial for understanding the distribution of move counts across different games, aiding in the identification and removal of outliers. Games with a move count below Q1 (43 moves) or above Q3 (86 moves) were deemed outliers and excluded from the dataset. This was a strategic decision to focus our model on a more consistent and representative subset of data, improving the model's ability to learn meaningful patterns without being skewed by extreme cases.

Following the data cleansing is the sequence creation phase for training data. The aim was to transform the chronological move data from each game into a structured sequence format that an RNN could process. For each match, sequences of incremental lengths were generated. For instance, the first sequence contained only the first move, the second sequence contained the first two moves, and so on, until the entire game's moves were encapsulated in a sequence. This incremental approach allows the RNN to learn from progressively growing contexts, a method that aligns with how strategic decisions in chess build upon previous moves.

Here is the pseudocode illustrating this process:

```
Function create_sequences_detailed(dataframe):
    Initialize sequences as an empty list
    Initialize targets as an empty list
```

```
    For each game in the dataframe grouped by 'Game_ID':
        Set elo to the first 'Elo' value in the group
        Remove 'Game_ID' and 'Elo' columns from the group
        For i from 1 to the number of moves in the game:
            Create a sequence from the first move to the i-th move
            Append the sequence to sequences list
            Append elo to targets list
        For i from 2 to the number of moves in the game:
            Create a sequence from the second move to the i-th move
            Append the sequence to sequences list
            Append elo to targets list
    Return sequences, targets
```

However, RNNs require a uniform input shape. To address the varying lengths of these sequences, zero-padding were applied, ensuring each sequence was extended to the length of the longest sequence in the dataset. This padding process allows the RNN to process sequences of uniform length while distinguishing between meaningful move data and padded zeroes. The built-in function pad_sequences was utilised from TensorFlow to process the data.

Here is the pseudocode showing this process:

```
Function prepare_sequences(sequences, targets):
    Determine max_len as the maximum length of any sequence in sequences
    Initialize X_padded as an empty list
    For each sequence in sequences:
        Convert sequence to numpy array
        Pad the sequence to max_len with padding at the end
        Append the padded sequence to X_padded
    Convert targets to a numpy array and store in y
    Return X_padded, y
```

Here is the structure of the sequence after being processed:

MOVE(GAME, No. move)
N is no of moves by white in game 1
M is no of moves by white in game 2

sequence1 = [move(1,1),0, 0,……., 0]

```
sequence2 = [move(1,1), move(1,2), 0, 0, …., 0]
……
sequenceN = [move(1,1), move(1,2), …..., move(1,N)]

sequence N+1 = [move(2, 1), 0, 0, …., 0]
sequence N+2 = [move(2, 1), move(2, 2), 0, 0, …., 0]
…….
sequence N+M = [move(2, 1), move(2, 2), …..., move(2,M)]
```

Data Preparation and Splitting

Initially, the dataset is divided into training and validation sets using the train_test_split function from the sklearn.model_selection library. This function allocates 80% of the data to training and 20% to validation, controlled by a random_state for reproducibility. This step ensures that the model is trained on a diverse subset of the data and validated on a separate, unseen subset to evaluate its generalization capability.

Model Definition

A Sequential model from TensorFlow's Keras library is defined to facilitate the linear stacking of layers. Depending on the layer_type parameter, either an LSTM or a GRU layer is added as the first layer of the model. This layer is configured with a specified number of units, which represents the dimensionality of the output space and directly influences the model's ability to capture information from the data. The input shape of the layer is determined by the features of the training data (X_train).

Additional Layers and Compilation

Following the recurrent layer, a Dense layer is added to serve as the output layer of the model. This layer uses a linear activation function by default, making it suitable for regression tasks where the model predicts a continuous variable. The model is compiled with the Adam optimizer and mean_squared_error as the loss function, setting the stage for training.

Callbacks:

1. **EarlyStopping** monitors the validation loss and stops training if it does not improve for 10 consecutive epochs, preventing overfitting and saving computational resources.

2. **ModelCheckpoint** saves the model at the end of each epoch into a designated directory, ensuring that the best version of the model is preserved based on performance metrics.

Model Training:

The model is trained using the fit method, which processes the data in batches of a specified batch_size and iterates over the dataset for a given number of epochs. Validation data is used to evaluate the model at the end of each epoch, providing insights into its performance and allowing for early stopping if the model ceases to improve.

Model hyperparameters:

1. Epochs: 100
2. Batch size: 64
3. Units: 50
4. Test size: 0.2
5. Optimizer: Adam
6. Loss: MSE
7. Early stopping, patient: 10 epochs

Output:

The function returns the trained model and a history object containing the loss metrics, which can be used to analyse and visualize the model's training and validation performance over time.

```
# Function to create and train a model
def train_model(X, y, layer_type='LSTM', units=50, epochs=100, batch_size=64, val_split=0.2):
    # Split the data into training and validation sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=val_split, random_state=42)

    # Define the model
    model = Sequential()

    if layer_type == 'LSTM':
        model.add(LSTM(units, input_shape=(X_train.shape[1], X_train.shape[2])))
    elif layer_type == 'GRU':
        model.add(GRU(units, input_shape=(X_train.shape[1], X_train.shape[2])))

    # model.add(Dropout(0.2))
    model.add(Dense(1))  # Output layer for regression

    # Compile the model
    model.compile(optimizer='adam', loss='mean_squared_error')

    # # Define the early stopping callback
    early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1, mode='min')

    checkpoint = ModelCheckpoint('/content/drive/MyDrive/Colab_Notebooks/model/model_{epoch:02d}', save_best_only=True, save_freq='epoch', verbose=1)

    # Train the model
    history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
                        validation_data=(X_test, y_test), callbacks=[checkpoint, early_stopping])


    return model, history
```

Figure 10: Function used to train data using RNN (LSTM and GRU)

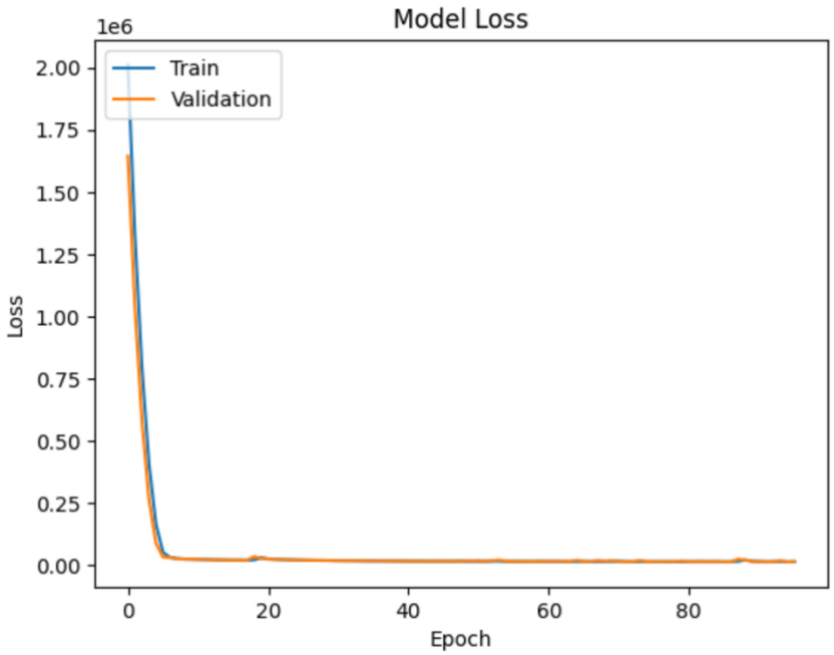
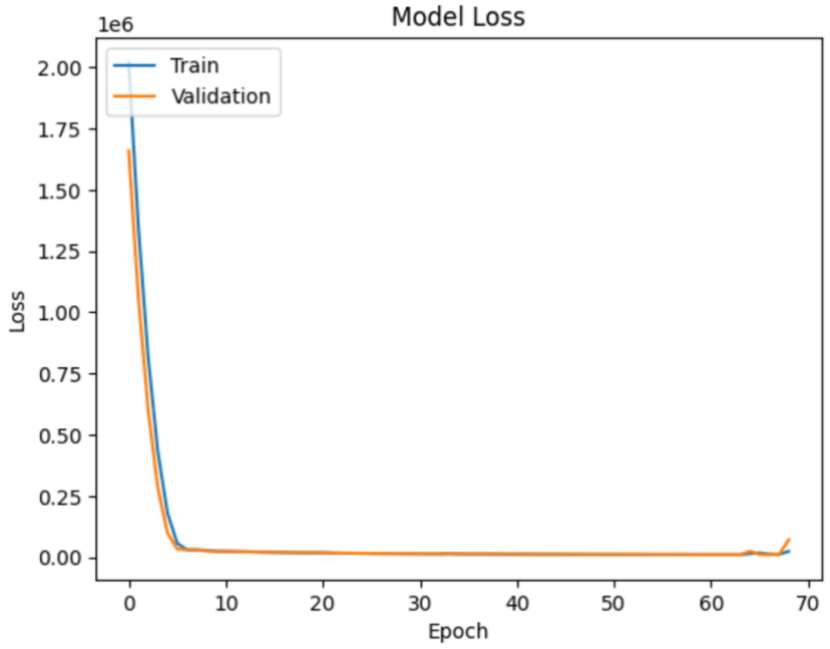
3.2.2.3 Result

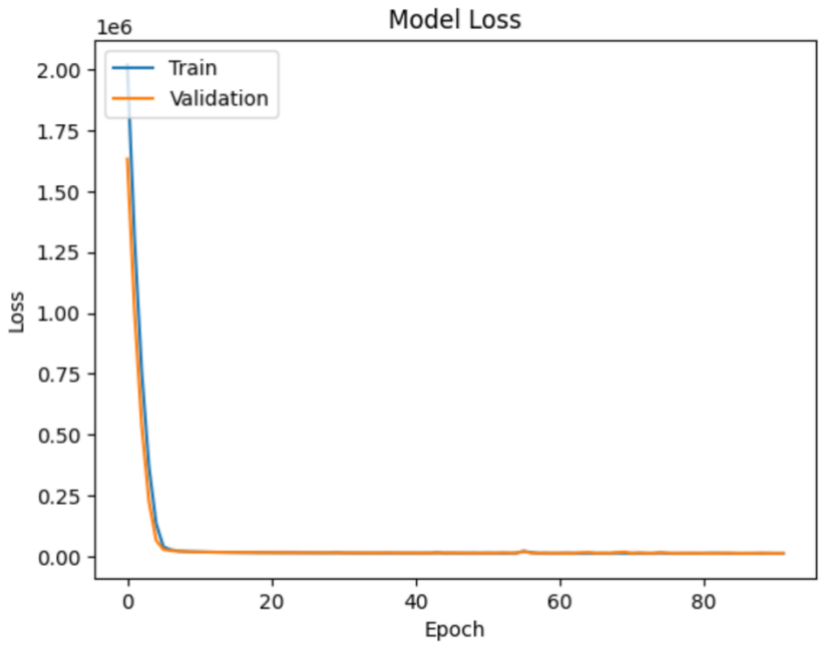
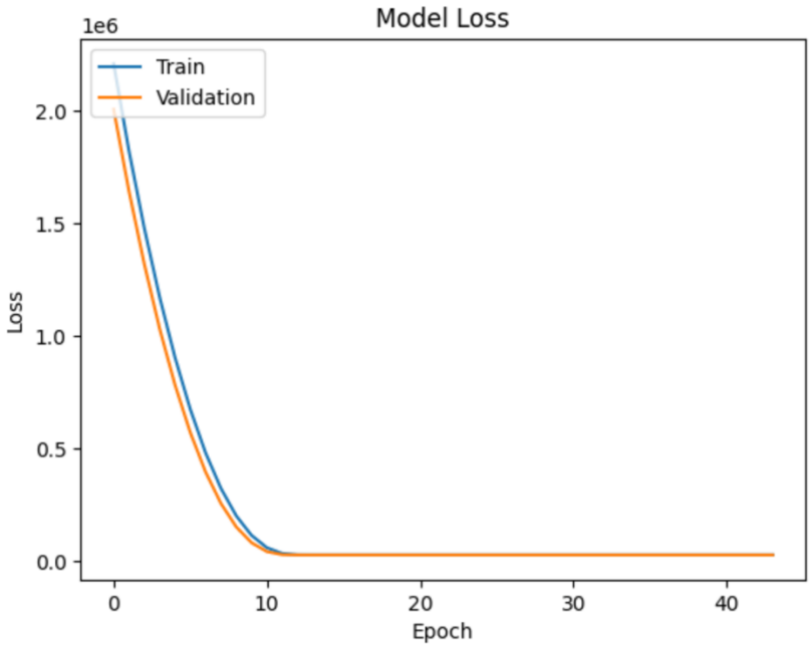
The results from the training of the GRU and LSTM model with 100 epochs, with a batch size of 64 and 50 units, exhibit a clear trend of improvement in both training and validation loss, demonstrating the model's ability to learn and generalise from the chess data effectively.

Table 2: Training sequence of moves

Algorithm	Both Pieces		White Piece	
	Train MSE	Test MSE	Train MAE	Test MAE
GRU	14030.8203	13417.4453	8560.6729	8618.7725
LSTM	11352.9766	11399.3115	29284.9336	29104.3223

The lowest recorded validation loss was 8,618.77246, trained using the GRU model, achieved at Epoch 98, marking the best state of the model during this training run. This suggests that the model was at its most effective at this point, demonstrating a robust ability to predict Elo ratings based on the input chess move sequences. The model achieved a prediction error of approximately 93 Elo points from the actual value. This level of error is considered acceptable given the complexity and inherent unpredictability of chess move sequences. Such challenges stem from the diverse strategic decisions players make, which are difficult to capture fully through model predictions.

	Graph for Loss function
Training with both pieces with GRU mode	 *Figure 11: Loss graph for Training both pieces with GRU model*
Training with both pieces with LSTM model	 *Figure 12: Loss graph for Training both pieces with LSTM model*

Training with white pieces with GRU model	 *Figure 13: Loss graph for Training white pieces with GRU model*
Training with white pieces with LSTM model	 *Figure 14: Loss graph for Training both pieces with LSTM model*

3.3 Web Application

3.3.1 Overview

The web application is a sophisticated platform designed to merge interactive chess play with advanced machine learning insights. At its core, the application offers users a unique experience: playing chess against the Stockfish engine.

Upon each move made by the player, the application leverages the previously trained machine learning models to predict and display the player's Elo rating in real-time, adjusting after every move to reflect the player's evolving skill level. This dynamic feature adds an educational and analytical dimension to the game, allowing players to see the immediate impact of their strategies on their predicted skill level.

In addition to the real-time Elo rating prediction, the application provides an in-depth analysis of each move's quality and the overall board position. This analysis is powered by the Lc0 engine, a renowned neural network-based chess engine, which calculates various weights to evaluate the position. Such detailed feedback gives players insights into the strengths and weaknesses of their moves, encouraging learning and improvement.

The user interface is designed for ease of use and intuitive interaction. To make a move, players simply click on the piece they wish to move and then click on the destination square. This straightforward mechanism ensures that players can focus on their strategies and the insights provided by the application, rather than navigating complex controls.

3.3.2 Design and Architecture

3.3.2.1 Frontend:

1. **User Interface Design:**

   Chessboard: The chessboard is the central element of the web application's user interface, designed to emulate the traditional appearance a physical chessboard. It's where the action happens, allowing players to visualise their game, plan their strategies, and execute their moves. The board's design is intuitive, ensuring that users, regardless of their chess proficiency level, can easily interact with the pieces and understand the game's progression.

Predictive Analytics Display: This display gives users real-time insights into their gameplay, offering a dynamic prediction of their Elo rating with each move they make. It's a tool designed not just for feedback but also for education, helping players gauge the effectiveness of their strategies and understand how their decisions impact their performance. The display is crafted to be user-friendly, presenting complex analytics in an easily digestible format, enabling players to make informed decisions and improve their chess skills.

Reset Position Button: The Reset Position Button is a crucial element of the user interface, offering a straightforward mechanism for users to start anew at any point in their game. This feature caters to the user's desire for flexibility and experimentation, allowing them to clear the board and reset their game without navigating through complex menus or interfaces. Positioned for easy access, this button ensures that users can quickly initiate a new game, encouraging continuous play and exploration of different strategies within the application. The simplicity and utility of the reset button contribute significantly to the seamless and engaging user experience that the application aims to provide.

2. **Interactivity and Responsiveness:**

   React Implementation: The front end is designed to be interactive and responsive, leveraging React component-based architecture. React efficient update and rendering system ensures that the user interface is dynamic, updating the chessboard and predictive analytics in real-time as the user interacts with the application.

   User Interaction: The application is designed with a straightforward interaction model. Users make moves by clicking on a piece and then clicking on the destination square. This interaction pattern is intuitive, minimising the learning curve for new users and ensuring that experienced players can focus on their strategies.

   Responsive Design: Only for web browsers in full-screen

The front end of the web application emphasizes user engagement and interaction by adopting a simple yet effective design. This allows players to immerse themselves in the chess experience while benefiting from the integrated machine-learning insights.

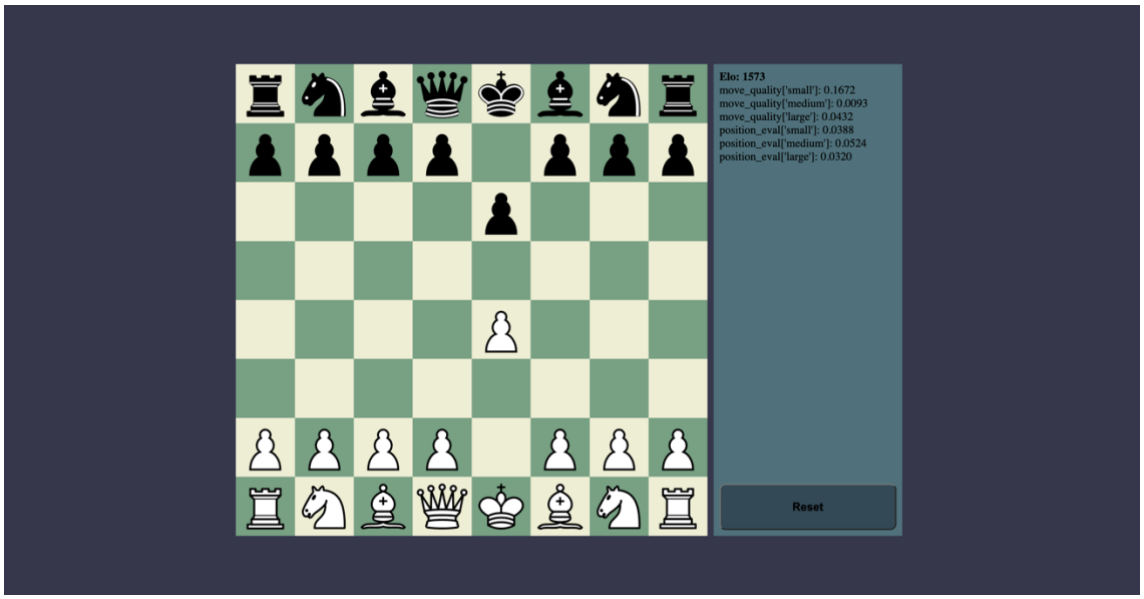


Figure 15: Photo of web application

3.3.2.2 Backend:

Integration with Machine Learning Models:

In the backend architecture, the integration with machine learning models is a critical component, facilitating the predictive and analytical features of the web application. The Predict class plays a central role in this integration, utilising TensorFlow to load and interact with the pre-trained machine learning model, specifically designed for predicting player Elo ratings and move quality in real time.

Chess Engine Integration:

The backend also includes the **'ChessEngine'** class, which interfaces with both Stockfish and LCZero engines. This class enables the application to not only predict outcomes but also to analyse move quality and provide real-time evaluations. The **'bestMove'** method utilises Stockfish (Elo: 2400) to suggest the best move based on the current board state and play against the user.

The **'move_quality_with_cache'** and **'LCZERO_Eval_with_cache'** methods interact with the LCZero engine to evaluate the board position before and after a move, calculating the move's quality based on these evaluations. These methods leverage caching to enhance performance, avoiding redundant computations for previously analysed positions.

Temporary Data Storage:

Without a database, the backend relies on in-memory data structures to store and manipulate data during active user sessions. The **'input_to_model'** list is a prime example, acting as a temporary store for the sequential game states required for model predictions. This data is held in memory just long enough to facilitate real-time analysis and prediction before being discarded once the session ends or the game is reset.

3.4 Implementation

3.4.1 Languages and Frameworks

Python and Flask:

Python was selected as the primary programming language for the backend development. This decision was driven by Python's readability, straightforward syntax, and extensive support for data manipulation and integration with machine learning models, which make it ideal for our predictive analytics tasks.

Flask is a micro web framework written in Python. It provides the tools needed to build a solid backend. It allows for the easy setup of web servers and is very adaptable to the needs of complex web applications. Flask is particularly favoured for projects that require a clean and efficient approach to handling web requests and managing data flow, which are crucial for our application's real-time features.

JavaScript and React:

JavaScript remains the cornerstone of modern web frontend development due to its universality in web browsers and its asynchronous capabilities, which are essential for updating the user interface without reloading the web page.

Chosen for the front end, React's component-based architecture makes it an excellent choice for developing highly interactive web applications. It helps efficiently update and render components that depend on user actions, which is vital for our application, as the state of the game board can change frequently. React's ability to handle state and props efficiently ensures that the user experience is fluid and responsive, which is critical in a dynamic environment like a chess game.

3.4.2 Chess Engine

The chess engine is a core component of our project, responsible for simulating an opponent and providing a benchmark for our predictive models. Stockfish has been

integrated, a highly powerful open-source engine, through the Python 'stockfish' library. This allows us to set Stockfish to play at a 2400 Elo rating, presenting a challenging opponent for users and generating high-quality gameplay data for subsequent analysis.

Here is the function to find the best move using Stockfish:

```
class ChessEngine:
    def __init__(self):
        self.large_net_size = Backend(weights=Weights('./Lc0_networks/768x15x24h-t82-2-swa-5230000.pb'), backend='eigen')
        self.medium_net_size = Backend(weights=Weights('./Lc0_networks/t1-smolgen-512x15x8h-distilled-swa-3395000.pb'), backend='eigen')
        self.small_net_size = Backend(weights=Weights('./Lc0_networks/t1-256x10-distilled-swa-2432500.pb'), backend='eigen')

        self.stockfish = Stockfish('/opt/homebrew/bin/stockfish')

        self.move_qualities = {}
        self.fen_evaluations_cache = {"large": {}, "medium": {}, "small": {}}


    def bestMove(self, fen, Elo):
        self.stockfish.set_fen_position(fen)
        self.stockfish.set_elo_rating(Elo)

        # Get best move
        bestMove = self.stockfish.get_best_move()

        return bestMove
```

Figure 16: Chess Engine class (The implementation of Stockfish and Lc0 to the web app)

In addition to Stockfish, Leela Zero, a state-of-the-art deep-learning chess engine, plays a pivotal role in our system. Leela Zero's neural network-based evaluation provides us with nuanced insights into the quality of each move and the overall position strength, which are indispensable for our predictive analytics. These features from Leela Zero form a critical dataset that enhances the precision of our Elo prediction model.

The integration of these two engines ensures that the application is not only an interactive platform for players to engage with but also a sophisticated analytical tool. With Stockfish as the opponent and Leela Zero as the analytics workhorse, this approach can produce better results than previous approaches.

Chapter 4: **Testing**

To demonstrate the validity of the developed ELO prediction model in practical use, I have tested the model with live players in the developed web application. The intention of the tests was to determine whether the prediction of the ELO rating scale in match games with real interactions among real players can be accurate to the developed model. The table below illustrates the test arrangement, the results and the key findings from the sessions.

Testing Setup and Results:

The tests asked participants to play against a chess engine utilised into the web app, and the model predicted a participant's Elo based purely on the results of their games. Here are the results for each of the participants:

Table 3: Real-world Testing (consider moves and results up to 43 moves)

Player	Actual Elo	Moves	Time	Stockfish Elo	Predicted Elo
Player 1	1837	26	12 mins	1600	1525
Player 1	1837	43(48)	28 mins	2400	1782
Player 2	823	23	23 mins	1600	1438
Player 2	823	14	11 mins	2400	1189
Player 3	1359	43(47)	17 mins	1600	1643
Player 3	1359	37	18 mins	2400	1444
Player 4	1273	43(65)	25 mins	1600	1419
Player 4	1273	32	10 mins	2400	1524
Player 5	1450	29	10 mins	1600	1638
Player 5	1450	22	13 mins	2400	1494
Player 6	1557	32	16 mins	1600	1475
Player 6	1557	68	25 mins	2400	1529

1. Gameplay Against 1600 Elo Engine:

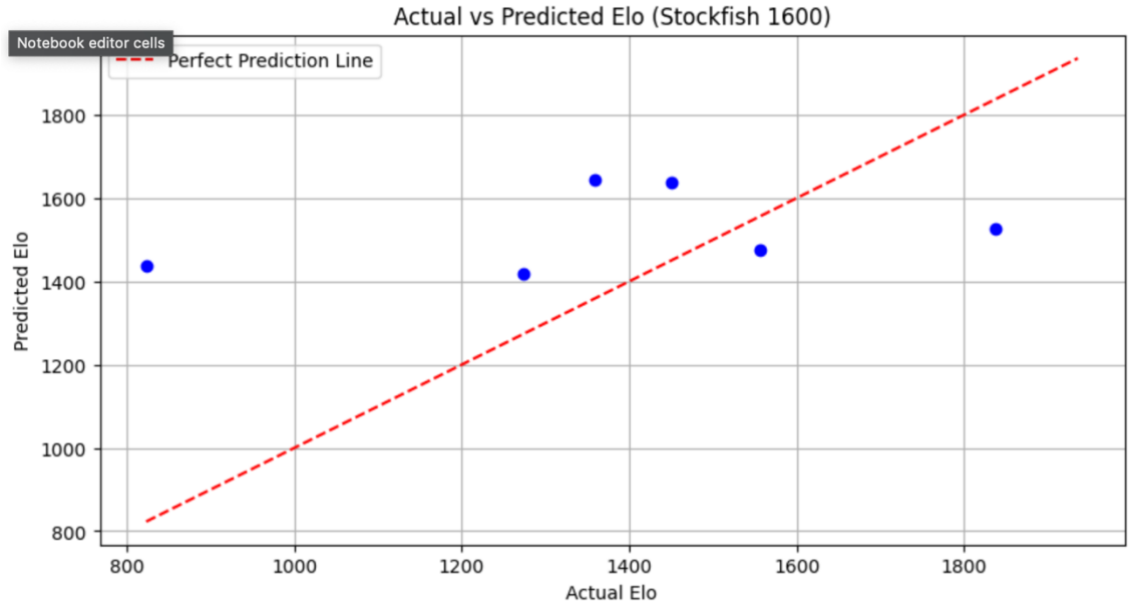


Figure 17: Stockfish 1600 as the opponent

In the first graph against the 1600 Elo engine, each graph bar shows that players tend to underperform compared to their actual Elo rating. Player 1's actual Elo rating was 1837, but the predicted performance is 1525, meaning players tend to underperform against the engine. Player 2's predicted Elo performance was 1438 where the actual Elo was 823, meaning players tend to overperform against the engine.

2. Gameplay Against 2400 Elo Engine:

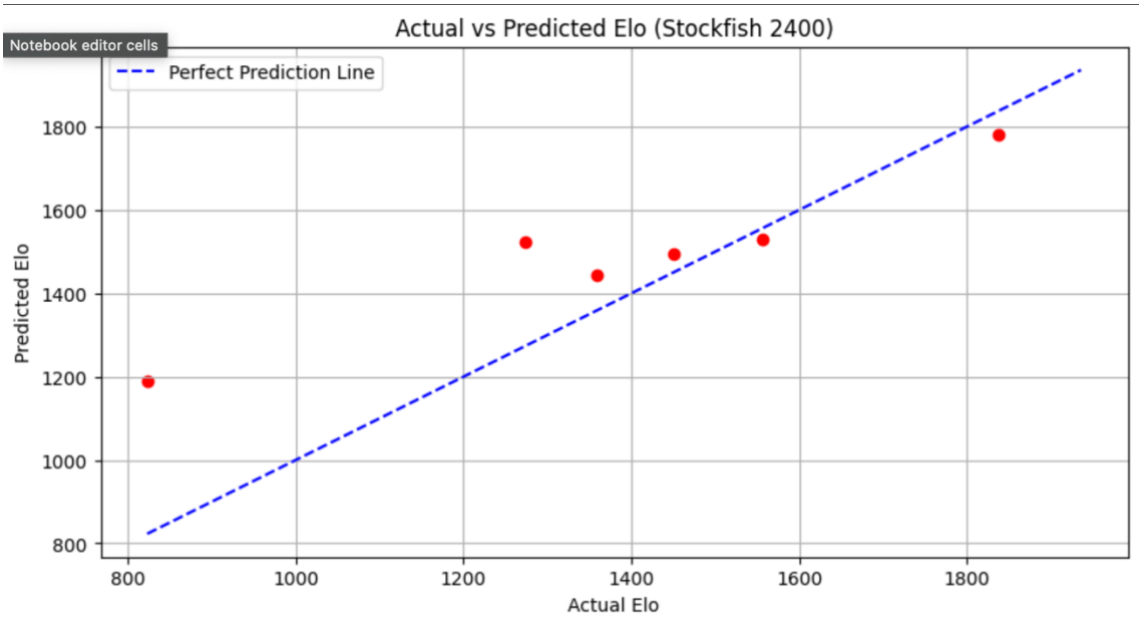


Figure 18: Stockfish 2400 as the opponent

In the second graph against the 2400 Elo engine, there is a significant increase in the accuracy of the model. Player 1 made a total of 48 moves. Since the model is set to predict the number of moves up to a maximum of 43, it might be inaccurate after those 43 moves. Up to the 43rd move, and over a duration of 28 minutes against a stronger 2400 Elo engine, the model initially predicted an Elo of 1782 which is quite near the actual Elo of the player (55 Elo difference). For every other player, the error of the Elo predicted ranges from 26 to 50, except for player 4. Moreover, Player 2 is an example of a player that have Elo out of range Elo interacting with the model. Therefore, it will predict with huge errors. Overall, the best-fit range for this model is the player has an Elo range from 1400 to 1900.

Model Sensitivity:

The model's performance is notably affected by the complexity of the chess positions encountered during play. Higher Elo engines tend to execute more consistent and challenging moves, complicating the game scenario for the player. This escalation in-game difficulty often prompts players to employ more sophisticated strategies, which in turn leads to more accurate Elo predictions by the model. These observations underscore the importance of refining the model's ability to factor in the opponent's quality more effectively in its predictions.

Conclusions:

These live application tests have been invaluable in demonstrating the Elo prediction model's real-world utility and areas for improvement. Enhancing the model's sensitivity to the quality of gameplay and refining its dynamic prediction capabilities are recommended to boost its accuracy and reliability. By continuing to test and iterate on the model with real-user data, further fine-tuning its performance can be done in the future to better serve the needs of the chess community.

Chapter 5: **Evaluation**

The evaluation of the "Elo Prediction" project focuses on accurately forecasting a chess player's Elo rating using a limited number of moves. In order to achieve this, a sequence of experiments was executed to apply various machine learning models and measurement metrics to enhance our forecasting procedure.

The initial trials started by employing regression models to forecast Elo ratings based on individual moves. Several methods, such as Decision Tree, Random Forest, Linear Regression, and Gradient Boosting, were implemented. Their performance was evaluated using metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2). The Random Forest model achieved the lowest Test Mean Squared Error (MSE) of 27871.1165, along with a Test Mean Absolute Error (MAE) of 131.1926 and a Test R-squared (R^2) value of 0.0909. Nevertheless, a notable gap was noticed between the metrics of the training and testing phases, suggesting that these models were overfitting.

In order to tackle this issue, Recurrent Neural Networks (RNNs) were utilised, notably Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, which provide superior capabilities for sequence prediction. Our objective was to improve the accuracy of our forecasts by assessing a player's Elo rating based on the sequence of moves rather than individual ones. The dataset included moves from both white and black pieces. The validation losses for the GRU and LSTM models were 13417.4453 and 11399.3115, respectively.

In order to refine the analysis of individual decision-making and playing styles, a subsequent methodology entailed training models purely with sequences of white-piece moves. The GRU model achieved a validation loss of 8618.77246, which is much lower than the LSTM model's loss of 29104.3223. This indicates that the GRU model had greater proficiency in capturing the user's playing style and characteristics. The reduction in validation loss also suggests a narrower margin of error in the anticipated Elo rating. This is especially significant considering the intricate nature of chess and the varied approach used in traditional Elo computation, which takes into account many games and time periods.

In addition to our machine learning efforts, a web application has been developed. This application functioned as both a platform for interactive participation and data collecting

and as a demonstration of the project's dedication to user experience and technology integration.

Compared to the previous work, their best MSE approach was 48247. In this project, I have successfully reduced the MSE to 8618.7725 using a different approach.

Overall, the "Elo Prediction" project has made significant progress in applying machine learning to chess analytics. This progress has been achieved through a series of methodological improvements and careful selection of models. The project has resulted in an effective model that can provide valuable insights into a player's skill level based on their gameplay.

Chapter 6: **Conclusion**

This project aimed to predict a chess player's Elo rating from a single game, with a specific focus on games where the player controls the white pieces. The model developed for this purpose analyses sequences of moves, shedding light on the strategic patterns and decision-making processes exhibited by the player. A key finding from our research is that the model's ability to accurately predict the Elo rating improves with an increase in the number of moves made by the player. However, due to the inherent complexity of chess, perfectly predicting a player’s Elo from a single game remains a challenging endeavour. Notably, the model demonstrates enhanced predictive accuracy in scenarios involving highly complex positions, as these provide more extensive move sequences and richer contextual information, enabling more precise Elo estimations.

6.1 Further work

Improvement of Model Sensitivity: Future work could focus on refining how the model accounts for different levels of opposition quality. This involves enhancing the model’s algorithms to better interpret and analyse the complexity and consistency of moves made by higher Elo opponents. By doing so, the model will be able to adjust its predictions more accurately based on the strategic demands of the game.

Dynamic Prediction Adjustment: Incorporating mechanisms for real-time adjustment of predictions as more moves are made could significantly increase the utility of the application in live game scenarios. Developing dynamic algorithms that update Elo predictions as new data becomes available during a game would provide players and coaches with immediate feedback on performance and strategy adjustment needs.

Expansion to Black Pieces: While this project focused on games where the player was using white pieces, applying the same modelling techniques to scenarios where the player is using black pieces could broaden the applicability of the model. This expansion requires adjusting the model to account for the strategic differences typically seen in games played with black pieces.

References

Dominik Klein, 2022. Neural Networks for Chess. [online] Available at: https://arxiv.org/abs/2209.01506 [25th March 2024].

Chess.com, 2020. 'FEN (Forsyth-Edwards Notation)', [online] Available at: https://www.chess.com/terms/fen-chess [Accessed 25th March 2024].

Chessprogramming.org, 2020. 'Forsyth-Edwards Notation', [online] Available at: https://www.chessprogramming.org/Forsyth-Edwards_Notation [Accessed 25th March 2024].

Ruder, S, 2016. An overview of gradient descent optimization algorithms. [online] Available at: https://arxiv.org/abs/1609.04747 [25th March 2024].

Willmott, C.J., & Matsuura, K, 2005. Advantages of the mean absolute error (MAE) over the root mean squared error (RMSE) in assessing average model performance. Climate Research, 30, 79-82, [online] Available at: https://www.int-res.com/abstracts/cr/v30/n1/p79-82/ [Accessed 25th March 2024].

Nagelkerke, N.J.D. (1991). A note on a general definition of the coefficient of determination. Biometrika, 78(3), 691-692, [online] Available at: https://www.cesarzamudio.com/uploads/1/7/9/1/17916581/nagelkerke_n.j.d._1991_-_a_note_on_a_general_definition_of_the_coefficient_of_determination.pdf [Accessed 25th March 2024].

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735-1780, [online] Available at: https://www.researchgate.net/publication/13853244_Long_Short-term_Memory [Accessed 25th March 2024].

Cahuantzi, R., Chen, X., & Güttel, S. (2021). A Comparison of LSTM and GRU Networks for Learning Symbolic Sequences, [online] Available at: https://ar5iv.labs.arxiv.org/html/2107.02248 [Accessed 25th March 2024].

Cistiakovas, J., Nolan, O., & Yamkovoy, A. (2020-2021). Machine Learning Project - Elo Prediction. CSU4406, The University of Dublin, Trinity College, Ireland, [online] Available at: https://github.com/OisinNolan/Elo_Predictor/blob/main/Elo_Predictor_Report.pdf [Accessed 25th March 2024].

Riis, Soren. (2006, June 6). Review of "Computer Analysis of World Chess Champions". ChessBase, [online] Available at: https://en.chessbase.com/post/computer-analysis-of-world-champions/6 [Accessed 25th March 2024].

Appendix A - Example

This is the PGN format of a game in chess:

```
[Event "Rated Classical game"]
[Site "https://lichess.org/j1dkb5dw"]
[White "BFG9k"]
[Black "mamalak"]
[Result "1-0"]
[UTCDate "2012.12.31"]
[UTCTime "23:01:03"]
[WhiteElo "1639"]
[BlackElo "1403"]
[WhiteRatingDiff "+5"]
[BlackRatingDiff "-8"]
[ECO "C00"]
[Opening "French Defense: Normal Variation"]
[TimeControl "600+8"]
[Termination "Normal"]
1. e4 e6 2. d4 b6 3. a3 Bb7 4. Nc3 Nh6 5. Bxh6 gxh6 6. Be2 Qg5 7. Bg4 h5 8. Nf3 Qg6 9. Nh4 Qg5 10. Bxh5 Qxh4 11. Qf3 Kd8 12. Qxf7 Nc6 13. Qe8#
```