Chess Game Outcome Prediction

Alex Anuszkiewicz
Department of Computer Science
Boston University
Boston, MA 02215
`aalex1@bu.edu`

https://github.com/Aanuszkiewicz/chess-prediction

Abstract

After thoroughly studying chess game data, conventions, and rules, we directly applied supervised machine learning methods to predict the ultimate outcome of chess games. A least-squares regression was utilized to show correlation between variables and identify important predictors. Then, we constructed a logistic regression model that could classify chess game outcomes with 84% accuracy given the ELO difference, time limit of the game, the opening played, and the final state of the chess board. Finally, we built a convolutional neural network (CNN) model that predicted game outcomes given solely a board state, with decreasing accuracy relative to the number steps away from a final board state. We observed an accuracy of 85.59% for endgame scenarios, which progressively declines to 45.29% for the start of a game. This project underscores the inherent complexity of chess, highlighting predictive challenges due to its combinatorial nature and lengthy strategic development. Overall, a decent CNN model has been developed that suggests areas for further predictive model enhancement and method evolution.

1 Introduction

1.1 The Problem

Let us consider two friends that sit down to play a game of chess. After white moves their first pawn, black also moves; the game begins. After an arbitrary amount of time, they observe the chess board and try to predict who is going to win. Essentially, this is the problem that we hope to solve. Given an arbitrary legal chess board[1], we want to predict whether a chess game will result in a win for white, a win for black, or a tie. Thus, this is a multinomial classification problem with three classes. Next, this brings up a question: which features are most important or useful in predicting game outcomes? The ELO difference[2] between the players, the opening played, the time limit of the game, the pieces captured, board control, the strategic layout of particular piece types, and the next legal moves are all useful in the scope of our problem. We will discuss the particular features engineered and utilized as well as the reason for using them in further sections. Due to the complexity of chess, there are countless features that could be analyzed, thus it is important to utilize the ones that tell the most about our data. That way, our models can capture its complexity to their maximal capabilities.

1.1.1 Why Chess Prediction?

Chess was invented around the 6th century, and it has around 605 million players worldwide, which is roughly 8% of the world's population[1]. Being such a widely played game, a highly accurate

[1]Legal chess board implies that it adheres to standard chess rules
[2]A rating system used to calculate the relative skill level of players in games

chess prediction model could benefit many players, especially new ones that are learning chess, in learning to recognize good and bad moves. For example, if a player was to make a very good move, they may see their win probability go up, which helps them recognize that as a positive move. Popular site Chess.com, for example, has a dynamic evaluation bar that performs complex calculations to determine who has the advantage, considering piece values and positions on the board [2]. By achieving more advanced prediction algorithms, learning chess becomes easier and more feedback-based. Now, viewing the larger picture, because chess presents a highly complex problem, we can actually model more applicable real-life problem solutions based on some methods that we obtain from solving complicated problems such as chess prediction. In that sense, it represents the evolution of computer science as a field, creating a intertwined web of growing complexity.

1.2 The Dataset

We utilized a Kaggle dataset of 6.2 million chess games played on Lichess during July 2016: `https://www.kaggle.com/datasets/arevel/chess-games`. This dataset has a large amount of data to work with (4.38GB), and there are already a decent number of useful features such as the game result, white ELO, black ELO, the opening in ECO encoding, the opening name, the time control, the reason for the game's end, and all movements made in algebraic notation. In section 3.2, we will discuss specific preprocessing methods applied to prepare the data.

1.3 Method and Results

Initially, we ran a few least-squares regressions to identify useful predictors. As expected, a least-squares model itself was not useful in actually making any predictions; however, it was useful in developing an understanding of data trends. We discovered a decent correlation between ELO difference and game outcome, and we found that white has a bias of roughly 5%. After, we built a logistic regression model that predicted outcomes based on ELO difference, time limit, and dummy variables representing opening played and final board state positions. This model had roughly an 84% accuracy, but it relies on many predictors and is not very generalizable to random chess board states. Finally, we constructed a CNN after multiple versions and parameter tuning, that had an accuracy of 85.59% for final board states and 45.29% for a clean board state (no moves made), with an average case accuracy of 59.33% for random board states. We will explore the specifics in section 5.3.

2 Related Work

Chess prediction is a highly explored problem due to the richness of the data that a single chess game provides. Now, we explore various popular methods and compare them with the methods we actually put into practice.

2.1 Prediction Based on Opening

Before considering the more complex data that board states offer, let us consider a more trivial prediction case, such as predicting the outcome of chess games based on the opening played. One option is utilizing decision tree learning [3]. Each combination of moves can represent a node, using both the opening and ELO difference as predictors. This yields a relatively high accuracy for openings that contain many moves [3]. Instead, we chose to train a multinomial logistic regression model using dummy variables for the opening. We trained off of not just the opening and ELO difference, but we also added the time limit of the game and the final board state at which a game outcome was determined also encoded in dummy variables. Even though this model had more predictors, it is possible that decision tree learning could have yielded higher accuracy. This suggests room for more experimentation with decision trees in the future to predict solely based off of opening or other more simpler variables. However, the ultimate goal of running this logistic regression was to create a baseline model for comparison and experimenting with the data, with the vision of a CNN in mind. Thus, it fulfilled its purpose well.

2.2 Prediction Based on Board State

Now, we will consider the problem of classifying chess game outcomes based on an arbitrary board state. A common approach for such a problem is utilizing convolutional neural networks, since they are able to identify spatial hierarchies of features with high accuracy. While critics argue that CNNs cannot adequately explain the complex tactics involved in chess, they offer very flexible solutions that can be fine tuned to the chess prediction problem [4]. Although beyond the scope of this project, if paired with other methods such as recurrent neural networks that are able to also capture the sequential aspect of chess, highly robust prediction models are able to be constructed. One of the greatest challenges in fitting a CNN to the chess prediction problem is deciding how to adequately tensorize the data to be able to capture the most valuable information about a board while also being mindful of space limitations. In *Predicting Moves in Chess using Convolutional Neural Networks*, Barak Oshri and Nishith Khandwala thoroughly discuss their approach for utilizing a convolutional neural networks to predict the next move given a legal board state [4]. They use seven CNNs, one for predicting which coordinate a piece needs to be moved out of, and the other six for representing advantageous positions for the different types of pieces. The data was represented as an 8x8x6 tensor, where the first two dimensions are the board and then there are 6 channels for each type of piece, where 1 represents white and -1 represents black. All CNNs were three layer convolutional neural networks of the form [conv-relu]-[affine]x2-softmax with 32 and 128 features, and they did not use any pooling layers or dropout layers with very minimal regularization to preserve as much data as possible. Meanwhile, I decided to represent my data as an 18x8x8 tensor which is less conventional for CNNs; this decision was made with the intended differentiation of the first 9 matrices from the last 9, since I trained my CNN on both random board states and the respective final board states of the game. Furthermore, my results actually did yield better results by using dropout layers with relatively light frequencies and I used higher regularization, however I did not use any pooling layers since data preservation was still very important. As for the matrices themselves, I took a similar approach to Oshri and Khandwala, where I represented white pieces with positive values and black ones with negative values. 6 out of the 9 matrices in both subgroups were piece specific, while the other three are not. Further details will be provided in 5.3.

3 Resources

3.1 Machine Specifications

All model training and data processing was performed on a 2020 MacBook Pro running macOS Montery Version 12.7.2. It has an Apple M1 processor that is an 8-core CPU with 4 efficiency cores and 4 performance cores, and it also has an 8-core GPU as well as a 16-core Neural engine that accelerates machine learning tasks. Additionally, it has 8GB of ram. In retrospect, it would have been better to train on a more powerful machine, as that would open many avenues for the representation of more complex data with more dimensions without as much negative influence on training time. Interestingly, on the processed dataset of approximately 600MB, the least-squares regression took multiple hours to run, logistic regression took only 10 minutes, and the convolutional neural network took an hour to train.

3.2 Data Preprocessing

After much experimentation with efficient data extraction and feature engineering, we implemented a configurable script to compile processed datasets. Starting with the full 6.2 million game dataset, the first step was to randomly sample a fraction of the dataset. We used at least half of the dataset for model training sets, and 1-2% for testing purposes or specific evaluations. Then, we only kept classical games to filter out non-standard game modes that are not representative of chess as a whole, i.e. bullet games. After, we dropped many columns that were unneeded such as the player names, game type, date and time of the game, and the ELO changes after the game. Then, we engineer all needed features. This includes the ELO difference (white ELO - black ELO), a win indicator (1 for a white win, 0.5 for a tie, and 0 for a black win), and a time limit feature. Finally, we engineer representations of the chessboard at specified times. This was the most difficult part of the data preprocessing, as there are many space and time constraints to consider. It is convoluted and inefficient to make a separate column for each piece, especially when we factor in that we need to store multiple chess board states. Instead, we utilize the Python-chess library, pushing moves from the algebraic notation of the game until we get to a desired state, then we save the board in Forsyth–Edwards Notation (FEN); this was highly efficient. We make two passes over each chess game. The first pass gets to the last move played, then saves the final board state and the total ply count p of the game. The second generates a random integer in the range $[1, p]$, then pushes the board to that state and saves it in FEN. Meanwhile, on the second pass we also save a configurable number of board states before the final state for model evaluation purposes. Finally, we save our processed dataset as a new .csv, a sample of which can be seen on the Git page.

3.3 Tools

All data processing and model training was done in Python 3.12.2. For efficient data manipulation via DataFrames, we utilized the Pandas library paired with NumPy. These were supported by the Python-Chess library, which was essential in efficiently parsing through games in algebraic notation and reading data that was used to vectorize board states. We wrote a simple python module on top of these libraries called Chess_Tools that tensorizes chess game data. For the least-squares regression we used Statsmodels, then for the logistic regression we utilized Sklearn. For the CNN model, we utilized TensorFlow paired with Sklearn for splitting and evaluation. Finally, for data visualization we used Matplotlib coupled with Seaborn. The model building and data visualization process was assisted by OpenAI ChatGPT, which helped with identifying parameters that could be tuned, suggesting potential strategies for vectorizing data, finding useful libraries, and creating visually appealing graphs.

4 Method

4.1 Linear Regression

Linear regression is a statistics procedure that is used to understand the association between between one or more independent variables (known as predictor variables) and some dependent variable. A linear regression may indicate a positive relationship between two variables (the dependent variable increases as the independent increases), no relationship (no change in the dependent variable upon changing the independent variable), or a negative relationship (the dependent variable decreases as the independent increases). To numerically determine this relationship we use correlation denoted by r, where $-1 \leq r \leq 1$. r only equals 1 or -1 when we have a perfect linear relationship where all points fall on the line perfectly. The formula for r is expressed as: $r = \frac{Cov(X,Y)}{S_X * S_Y}$ where the covariance between X and Y expresses the relationship between them but is then normalized and given a sign by dividing it by $S_X * S_Y$.

4.1.1 Least Squares Regression

A least squares regression fits a line which minimizes the sum of the squared distance between the actual values and the predicted values. The equation for a single independent variable is expressed as $y = \beta_0 + x_1\beta_1$, and, more generally, for n variables it is $y = \beta_0 + x_1\beta_1 + ... + x_n\beta_n$, where β_i is the slope associated with variable x_i and $\beta 0$ is the constant term. The goal is to minimize the minimize the sum of squares error (SSE), where $SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ [5]. Equivalently, the goal is to find a vector x^* such that $||Ax^* - b|| \leq ||Ax - b||$ for all $x \in \mathbb{R}$, where

$$A = \begin{bmatrix} 1 & a_{11} & a_{12} & ... & a_{1n} \\ & & \vdots & & \\ 1 & a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Solving for the gradient and setting it equal to 0, we get that $x^* = (A^TA)^{-1}A^Tb$. To perform a least squares regression on the chess dataset, we utilize the Statsmodels library and call on the OLS (ordinary least squares) method. This will utilize the aforementioned equation to solve for the optimal x. Then, it will summarize the statistics of the regression, giving us the R^2 value, adjusted R^2 (adjusted for the number of predictors), $F - statistic$ (compares the joint effect of all variables together), the constant, all values of vector x^*, and the p values for each variable (tells us whether or not a variable's difference is statistically significant).

4.2 Logistic Regression

While linear regression gives a continuous prediction based on independent variables, logistic regression predicts a discrete outcome. This is done through maximum likelihood estimation (MLE). The sigmoid function, $f(x) = \frac{1}{1+e^{-x}}$ takes a real number and maps it between 0 and 1, where $-\infty$ maps to 0 and ∞ maps to 1 [6]. With a binary classification problem, for example, we can say that an output that is greater than 0.5 classifies as yes to some problem while an output less than 0.5 classifies as no. Essentially, we must find the weights w_i and a bias term b to find a weighted sum of evidence z for a class where $z = (\sum_{i=1}^{n} w_i x_i) + b$. To actually create the probability, z is passed into the sigmoid function to map it to a value between 0 and 1. For a set of data points $\langle \vec{x}_k, y_k \rangle$ with $k \in [1, n]$, the likelihood can be expressed as

$$\prod_{k:y_k=1} f_{y_k} \prod_{k:y_k=0} (1 - f_{y_k})$$

which is a maximization problem that is equivalent to finding the $b, w_0, w_1, ..., w_n$ that maximize the log-likelihood [7]

$$l = \sum_{k:y_k=1} \ln(f_{y_k}) + \sum_{k:y_k=0} \ln(1 - f_{y_k})$$

Now, methods such as gradient ascent can be utilized to optimize l. In the case of multinomial logistic regression, we utilize the softmax function, which is a generalization of the sigmoid function, defined as [6]

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \quad 1 \leq i \leq K$$

By using this function and representing the weights, input, and bias as vectors, we can optimize and obtain a vector of output probabilities for each of the K classes. To perform a logistic regression with the chess dataset, Scikit-learn takes independent variables and a dependent variable, trains a logistic regression model on a testing set, then performs prediction on the testing set. By default, Scikit-learn utilizes limited-memory Broyden–Fletcher–Goldfarb–Shanno (lbfgs) to optimize the softmax function, however there are various algorithm options to choose from according to the particular application.

4.3 Convolutional Neural Network

Artificial Neural Networks (ANNs) are computational processing systems that hare inspired by the biological nervous system [8]. They are comprised of many nodes referred to as neurons, that entwine to collectively learn from the input in order to optimize the final output. CNNs are a subset of ANNs that focus on spatial recognition, making them very suitable for images. They are comprised of neurons in three dimensions, meaning the input volume will have a dimensionality of $x \times y \times z$, and the output will be $1 \times 1 \times n$, where n is the number of classes. Overall, CNNs are comprised of three layers: convolutional layers, pooling layers and fully-connected layers [8]. The convolutional layer plays a vital role in the operation of CNNs by utilizing learnable kernels that spread along the entirety of the depth of the input. This layer convolves each filter across the spatial dimensionality of the input to produce a 2D activation map. Essentially, the kernel moves along the input, computing a scalar product for each value in the kernel, then the network learns kernels that fire when they see a feature at a spatial position of the input; these are known as activations. The kernels are stacked to form the convolutional later. Convolutional layers are optimized through three parameters: depth, stride, and zero-padding. Depth defines the number of filters that go over the same region of the input, stride determines the depth around the spatial dimensionality of the input, or how much overlapping there is for the receptive field, and zero-padding is simply padding the border of the input. Pooling layers apply a kernel to reduce the dimensionality of the activation map, which can grow to be very large. They accomplish this through the use of the max function by passing over each region of the kernel and taking the max value. Finally, a fully connected layer simply contains neurons that are connected to the adjacent layers, where each neuron has a weight. The weighted sum plus bias is calculated, which is then passed through an activation function such as ReLu to introduce non-linear properties. Finally, we obtain an output. A common formula is to stack convolutional layers with ReLu activation with pooling layers between them, then eventually having a fully connected layer. This allows for the learning of increasingly complex features as we pass through the convolutional layers. TensorFlow allows for a comprehensive model definition through the specification of layer types in order, kernel sizes, strides, padding, activation functions, pooling layers, fully connected layers, other layers, regularization options, and more. Additionally, it allows for fast saving and loading of models, making it a perfect option for this project.

5 Results

5.1 Least-Squares Regression for Preliminary Analysis

Before running a least-squares regression, we made a parse over the dataset and determined the outcome distributions. Evidently, there is a bias toward white, which won about 50%, while black

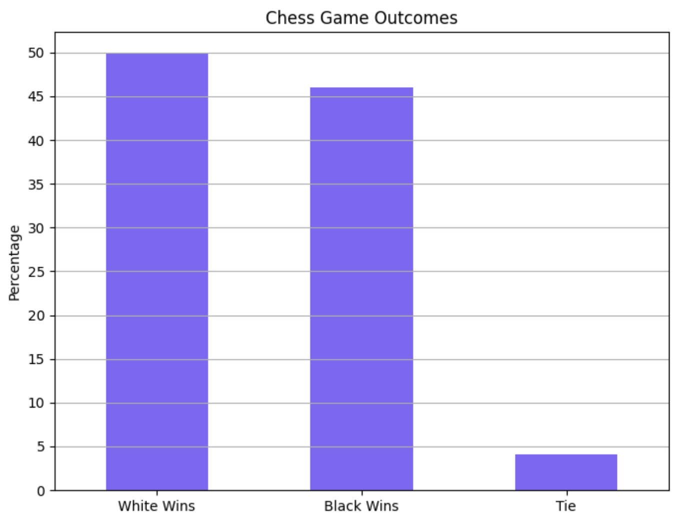


Figure 1: Game Outcomes

only won around 46%, and ties made up the remainder of outcomes. This observation would be fundamental to later model analysis, particularly in 5.3. This discrepancy comes from the advantage that white has in making the first move, being able to set the strategic trajectory of a game. Next, we decided to run a least squares regression on the data. To fit a categorical dependent outcome to a continuous prediction model, we constructed an indicator variable that specifies 1 as a win for white and 0 as a tie or a loss. If a prediction was made above 0.5 then we can classify the prediction as a win, and otherwise it can be a loss. However, a fundamental problem with this is that it groups ties in with losses, which is not accurate in the scope of chess. The second approach was to instead assign a value of 0.5 for ties, however the problem here is that it is inaccurate to make a range such as $[-\inf, \frac{1}{3}]$ a loss, $[\frac{1}{3}, \frac{2}{3}]$ a tie, and $[\frac{2}{3}, \inf]$ a win because it is probabilistically inaccurate to assign the cutoff values as such. In essence, our prediction from the least squares line means practically nothing in the scope of a multinomial classification problem. We ran a regression between white wins v. ELO difference and black wins v. ELO difference, giving us $r = 0.3448$, $r^2 = 0.1189$, $p = 0.0$. The p-value does indicate that there is statistical significance present, and the r^2 value indicates a weak relationship. Next, we calculated a least-squares regression with ELO difference, ECO[3] code dummies[4], and the time limit as the independent variables and white win for a dependent variable. The results are shown in Figure 3: Evidently, the r^2 value indicates a weak relationship. Once again, there are many problems with this approach, making it unfit for a multinomial classification problem, but it is useful for making initial observations about our data.

[3]Encyclopedia of Chess Openings; codes that specify particular openings, organized by strategy

[4]1 for the opening played, 0 for all other columns

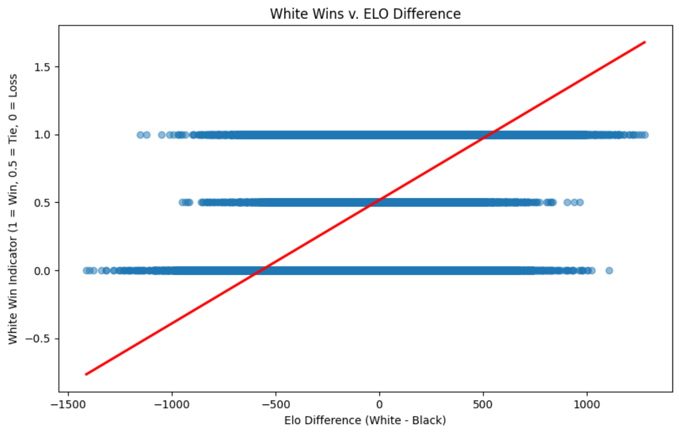
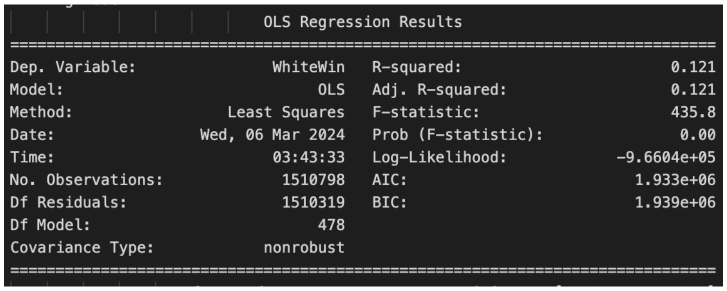


Figure 2: First Least Squares Regression

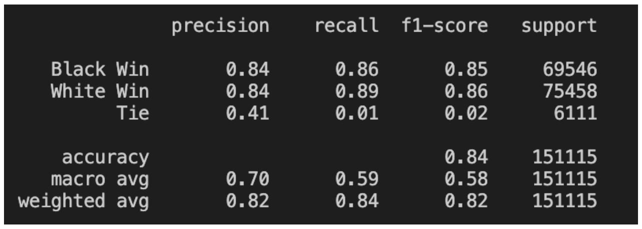


```
                            OLS Regression Results
==============================================================================
Dep. Variable:               WhiteWin   R-squared:                       0.121
Model:                            OLS   Adj. R-squared:                  0.121
Method:                 Least Squares   F-statistic:                     435.8
Date:                Wed, 06 Mar 2024   Prob (F-statistic):               0.00
Time:                        03:43:33   Log-Likelihood:            -9.6604e+05
No. Observations:             1510798   AIC:                         1.933e+06
Df Residuals:                 1510319   BIC:                         1.939e+06
Df Model:                         478
Covariance Type:            nonrobust
==============================================================================
```

Figure 3: Multiple Independent Variables Least Squares Regression

5.2 Logistic Regression: A Baseline Model

We ran a multinomial logistic regression using ELO difference, game time limit, ECO code dummies, and chess piece position dummies for the final board state. To create the position dummies, we have our first instance of board vectorization. To do this, we pass in the FEN of the final board obtained from methods described in 3.2, then we use Python-Chess to create a board and set its FEN. Now, we have a value mapping where pawns are 1 and -1, knights and bishops are 3 and -3, rooks are 5 and -5, queens are 9 and -9, and kings are 10 and -10 (positive for white, negative for black). Then, we simply flatten the board into a 1D array. We split the data with a standard 80/20 training testing split, then we run the logistic regression with a maximum iteration count of 10,000 to let it optimize as best as possible. Finally, we use it to predict on the testing set, and obtain the results in Figure 4. We



```
              precision    recall  f1-score   support

   Black Win       0.84      0.86      0.85     69546
   White Win       0.84      0.89      0.86     75458
         Tie       0.41      0.01      0.02      6111

    accuracy                           0.84    151115
   macro avg       0.70      0.59      0.58    151115
weighted avg       0.82      0.84      0.82    151115
```

Figure 4: Multinomial Logistic Regression Results

have a relatively high accuracy of 84%. The model has high precision and recall for cases where the class is a white win or black win, however, ties have a terrible precision of 41% and an even worse recall of 1%, indicating we only correctly identify 1% of ties. This brings down the whole model accuracy, and this issue arises from the underrepresentation of ties in the dataset, which only made up

4% of games. Furthermore, ties are more difficult to identify given the limitations of the predictors, considering the model has no context of how long a game has been going on for or the development of strategies over its course. This problem is reflected in the confusion matrix in Figure 5. Finally, it

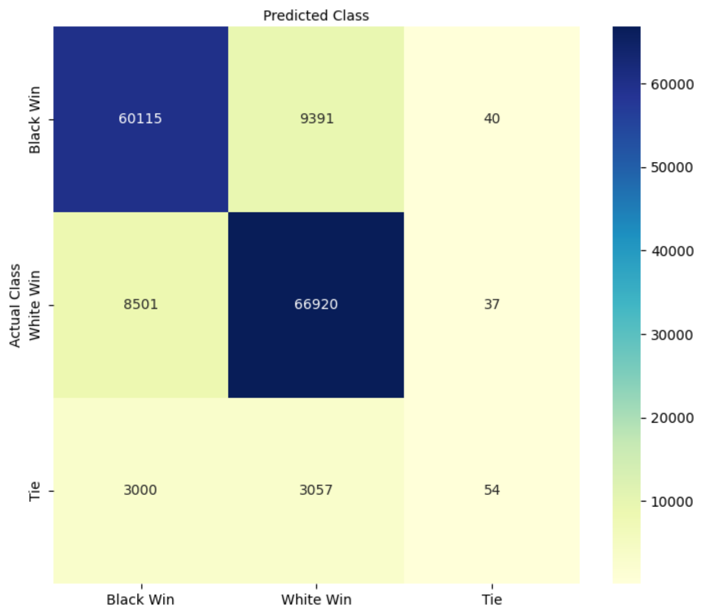


Figure 5: Multinomial Logistic Regression Confusion Matrix

is important to consider that our logistic regression is not very useful in the full problem scope. We may not know an opening until many moves in[5]. More concerningly, we will not have a final board state in the context of the problem where we want to predict from an arbitrary board state. Thus, we utilize a much more robust solution in 5.3.

5.3 Convolutional Neural Network: The Final Model

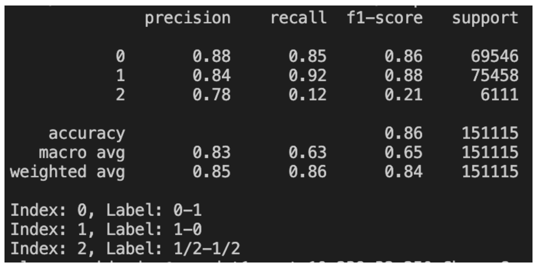
Through the use of a convolutional neural network, we present a solution to the chess prediction problem. First, we tensorized the data. Using the DataFrame for the random board state, we constructed 9 different 8×8 matrices. The first one was the same as the vectorized board in 5.2. The second and third represented all legal moves for the next move, where matrix 2 had 1s placed in the squares that could be moved from and matrix 3 had 1s placed in the squares that could be moved to. Then, the next 6 matrices represented the positions of all piece classes (pawns, knights, bishops, rooks, queens, kings), similar to the vectorization method in [4]. We had 1s placed wherever a piece of a given class was placed, and 0s in all other positions. Finally, this whole process was repeated a second time to create another 9 matrices for the final board state of a given game. All together, this created 18 matrices that were stacked to create an $18 \times 9 \times 9$ tensor for each game. We used an 80/20 training testing split for the data, and the model was built with the following layers:

1. Convolutional layer with 32 kernels, (3, 3) kernel size, same padding, ReLu for activation, and 0.01 L2 regularization
2. Batch normalization layer
3. Dropout layer with a 0.2 rate
4. Repeat 1-3 except with 64 kernels for the Conv2D layer
5. Flatten layer
6. Dense layer with 128 units, ReLu for activation, and 0.01 L2 regularization
7. Dropout layer with a 0.4 rate

[5]i.e. C89 Ruy Lopez: Marshall, Main line, Spassky Variation (18 moves) [9]

8. Dense layer with 3 units and softmax for activation

Better training speeds and accuracy resulted from the inclusion of dropout layers, batch normalization layers, and a small amount of L2 regularization. No pooling layers were used to preserve as much data as possible. After testing, we obtained the results shown in Figure 6. Evidently, the accuracy of



```
              precision    recall  f1-score   support

           0       0.88      0.85      0.86     69546
           1       0.84      0.92      0.88     75458
           2       0.78      0.12      0.21      6111

    accuracy                           0.86    151115
   macro avg       0.83      0.63      0.65    151115
weighted avg       0.85      0.86      0.84    151115

Index: 0, Label: 0-1
Index: 1, Label: 1-0
Index: 2, Label: 1/2-1/2
```

Figure 6: CNN Testing Results

86% is quite high, with good precision and recall on the white and black win cases. For the case of ties, we still suffer the same problem as in 5.2. However, we do see significant improvement, upping the precision to 0.78 and the recall to 0.12. Now, after running some more model evaluations, we see that the model accuracy decreases and loss increases as we take steps away from the final chessboard state of any given game as shown in Figure 7. Interestingly, our metrics oscillate quite significantly,

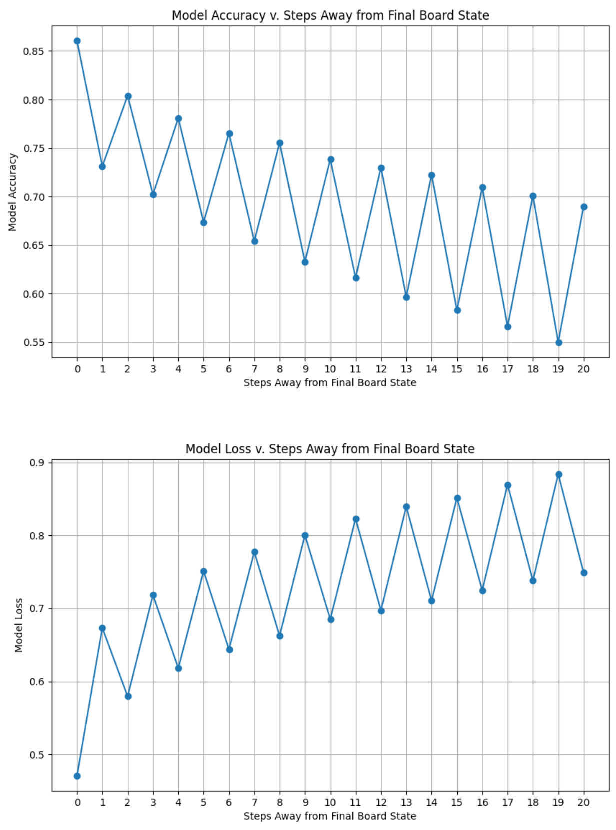


Figure 7: CNN Evaluation

dipping down on odd steps and going back up for even steps. This oscillation grows more drastic as we keep stepping away from the final state of a board, which is most likely due to the bias that white has in winning games. In the final state, it is more likely that white won, and this is reflected in the even steps away from it (more likely to be states right after white moved). Thus, we have much greater predictive power for even steps away from the final board. The bias is further reflected in Figure 8. Here, we can see that significantly more black wins were mistaken for white wins than

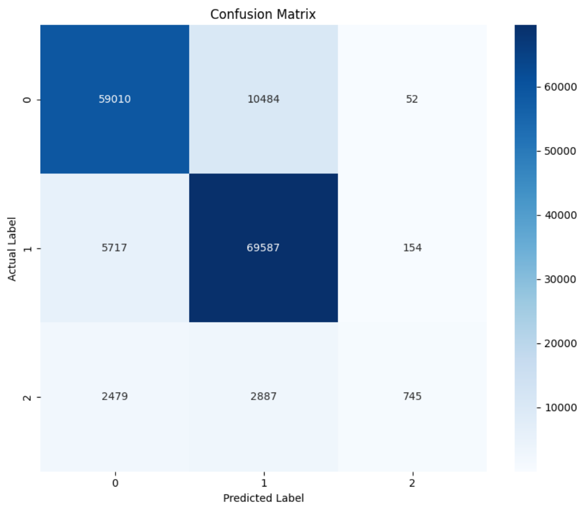


Figure 8: CNN Confusion Matrix

vice versa (10484 vs 5717), a much greater discrepancy than the logistic regression. This illustrates a limitation of our model which could be improved in the future. To fix this, it would be useful to increase the size of the dataset even more, and to also balance it in a way that generalizes well without harming realistic testing accuracy[6]. Performing a naive test, where we tested the model against clean board states with no moves made, we get an accuracy of 45.29%. Thus, as we keep stepping away from the final board state we converge to this accuracy. Finally, when we test the model on a set of completely random board states without the final states[7], we get a realistic test accuracy of 59.33%, which is relatively decent for three classes. By running a quick calculation, the expected ply of such a random board state would be about 32.

6 Conclusion

Overall, a convolutional neural network model has been constructed that can predict the outcome of chess games (white win, black win, or tie) with decent accuracy. The least-squares regression revealed insights that were useful in the evaluation of the CNN, and the logistic regression showed that more robust solutions are needed to accurately predict chess outcomes solely based on board state. The CNN still reflects bias in the model, with an oscillation that grows more extreme as we take steps away from final board states and into earlier game phases. In the future, it would be useful to consider the usage of recurrent neural networks paired with convolution neural networks to obtain more context about the strategic progression of games, which may result in much better results. However, such a project would involve many space limitations, necessitating careful and methodical application of rigorous theory. In short, we consider the project to be successful in the scope of the problem, but, more importantly, it provides a foundation that suggests areas for further model enhancement in future endeavors. Clearly, chess is a highly complex game, and predicting outcomes with high accuracy takes a great deal of thorough knowledge and application.

[6]We attempted to equally balance the white win, black win, and tie games, but it significantly harmed the realistic test accuracy due to overfitting which suggests alternative methods must be used for balancing

[7]This is done by tensorizing the same 9 matrices for random board states stacked twice

References

[1] Jannik Lindner. The most surprising chess statistics and trends in 2024. `https://gitnux.org/chess-statistics/`. Accessed: 2024-04-24.

[2] Chess.com. `https://chess.com/`. Accessed: 2024-04-24.

[3] Samia Khan, Minseo Kwak, Ethan Jones, and Kyle Keirstead. Predicting chess match results based on opening moves. `https://github.com/samiamkhan/4641Project`. Accessed: 2024-04-24.

[4] Keiron O'Shea and Ryan Nash. Predicting moves in chess using convolutional neural networks. `https://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pdf`. Accessed: 2024-04-24.

[5] Astrid Schneider, Gerhard Hommel, and Maria Blettner. Linear regression analysis. *Dtsch Arztebl Int*, 107(44):776–782, 2010.

[6] Daniel Jurafsky and James Martin. Logistic regression. `https://web.stanford.edu/~jurafsky/slp3/5.pdf`. Accessed: 2024-04-24.

[7] Charalampos Tsourakakis. Homework 2 cs365 spring 2024. `https://piazza.com/class_profile/get_resource/lrgnqll42av781/lux08l4572a211`. Accessed: 2024-04-24.

[8] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[9] Eco codes. `https://www.365chess.com/eco.php`. Accessed: 2024-04-24.