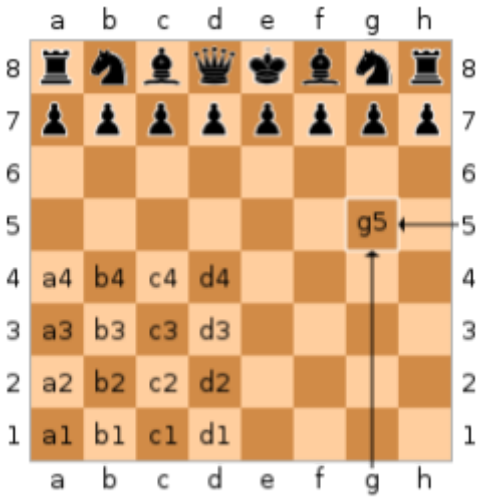
Predicting Win Rate of Chess Game States

Dylan Cianfrone

For this project, the goal was to be able to predict the chance that a player would win a game of Chess based on the game state. The data used was collected from Kaggle from user A.revel (https://www.kaggle.com/arevel/chess-games). It consists of a set of data from about 6 million games of chess that occurred on lichess.org. Each datapoint consists of the type of game played (there are several game modes available on Lichess), the usernames of each player, each player's ELO, or numerical ranking, the opening that was used in the game, and the set of all the moves that took place in the game.

First, states of the game had to be gathered from the moves that took place in the game. The moves are encoded in chess algebraic notation. A typical moveset in this notation looks like the following:

```
1. d4 d5 2. c4 c6 3. e3 a6 4. Nf3 e5 5. cxd5 e4 6. Ne5 cxd5 ...
```



A chess board, using files and ranks.

And so on. Each turn, white moves first, then black. The board is divided into columns, called files, which are referred to as the letters a-h. Rows, or ranks, are notated by numbers 1-8. Therefore, each square on the chessboard has a unique identifier created by its file and rank. The syntax for each move in algebraic notation is also notable. Pieces are notated by particular single capital letters. 'N' stands for knight, 'R' for rook, 'B' for bishop, 'Q' for queen, and 'K' for king. Pawns don't use a letter - any move not given a letter for the piece moving is assumed to be a pawn. The square in each move represents the square being moved to. For example, in move 4, the white player moves a knight to f3. Notably, the format does not take note of the square that the piece came from. Usually, only one piece could fulfill the move legally, so noting the square being left would take up space unnecessarily. The file or rank of departure is only noted when it is otherwise unclear that a piece would be moved. Captures are notated by the character 'x' being added, such as in move 5 when white's pawn on file c captures the piece on d5. There are additional syntax rules for events such as castling and promotion of pawns.

First, a set of game-states had to be extrapolated from the list of moves for each game. This requires a representation of a game-state to be decided upon. A 64-character string was

chosen such that string[0] would be a8, string[1] b8, string[8] a7, and so on. Empty squares were noted as underscore characters, and pieces were represented by the letters which represent them in algebraic notation (and 'P' for pawns). White pieces are notated by uppercase letters while blackcharacters are lowercase. This provides a simple way of printing the string for debugging. The string of moves was tokenized, and tokens that notated the move number were discarded. Then, moves were handled based on what type of move they were: pawn moving, non-pawn piece moving, promotion, castling, capture, etc. For each type of move, and each piece moving, a piece had to be searched for such that it could legally move to the space being referenced.

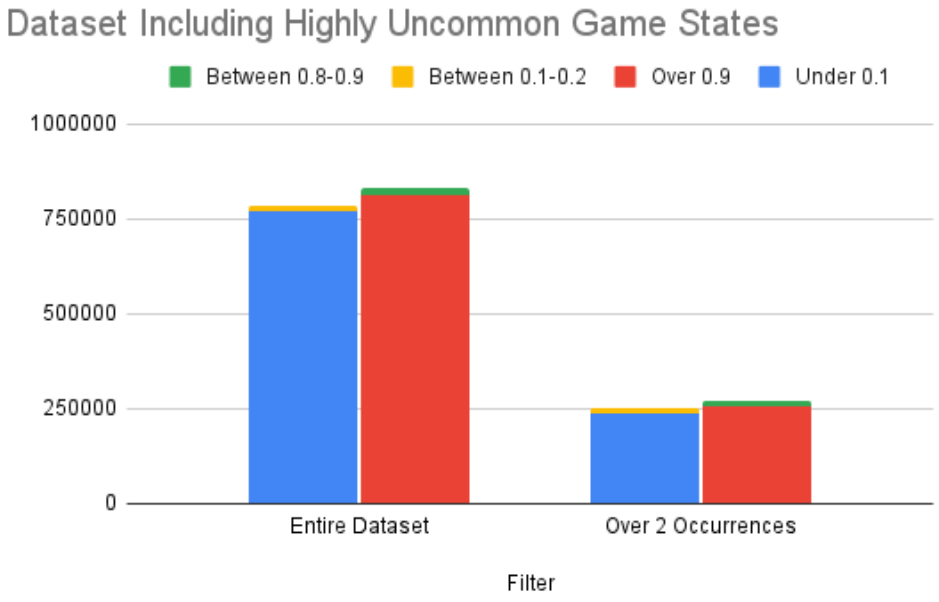
This algorithm was used on the original dataset, which was trimmed considerably. First, game types that were not classic chess were discarded as well as games which concluded because of time forfeit rather than checkmate. This was to keep all games standard. This still resulted in over a million games being considered. Each game had its moves replaced by a list of states which could be easily split into tokens each time it was used.

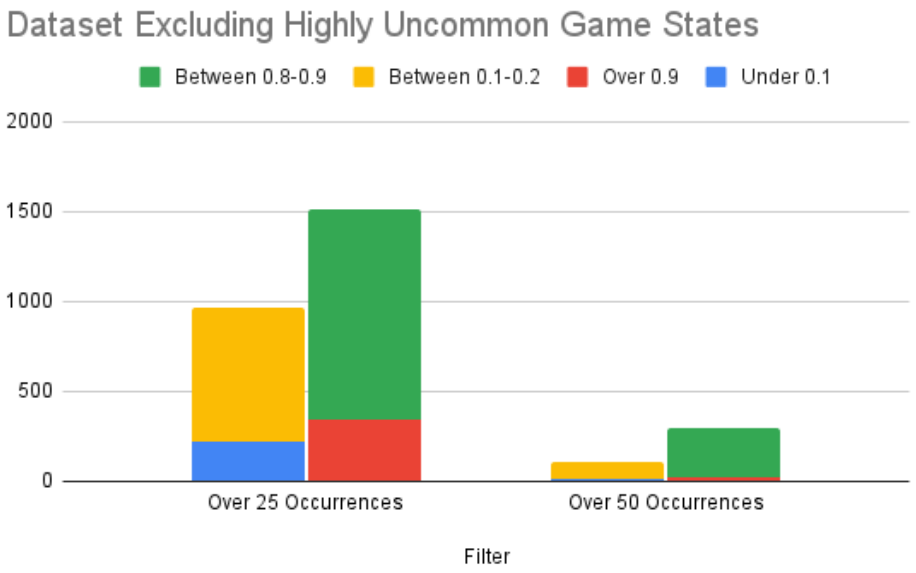
Storing the states of the games as strings was effective for debugging clarity as well as efficiently writing to file. However, the state of the games had to be further encoded in order to use the states for regression. First, states were combined by key in order to get an RDD (which was soon converted into a Dataframe) of the string game state and value a tuple of (number of times white won when this state occurred, total number of times the state occurred). Once put into a Dataframe, more features were captured from the board string.

One feature tested was a vector which represented the number of pieces of each type each player had on the board. Using this representation alone yielded mean absolute error of about 30% using Linear Regression. Then, an additional vector was appended representing the board. The vector had 64 entries, each 0 if the board was empty at the corresponding space and 1 if there was a piece there. This improved error slightly – from around 33% to around 30% – but the error was still much too high.

The problem came from states that appeared a very small number of times. This is because the win rate for these datapoints which have a very low sample size tends drastically towards 0 or 1, which massively skews both regression and evaluation. These states took up a massive percentage of the dataset, meaning that almost all points had win rates close to 0 or 1, when the true win rate of most of these states was much closer to the center.

The following graphs illustrate the number of highly uncommon game states appearing in the dataset filtered in several ways: The entire dataset, and the dataset filtered to exclude states appearing in less than 2, 25, or 50 games out of the over 1 million games considered.





Note the change in majority color from the less filtered to more filtered graph. In the samples with less filtering, most of the graph is taken up by win rates over 90% for each player (under 10% win rate for white is a 90% win rate for black). On the other hand, states which appear in more games have many more win rates closer to 50% compared to those highly skewed towards one player or the other.

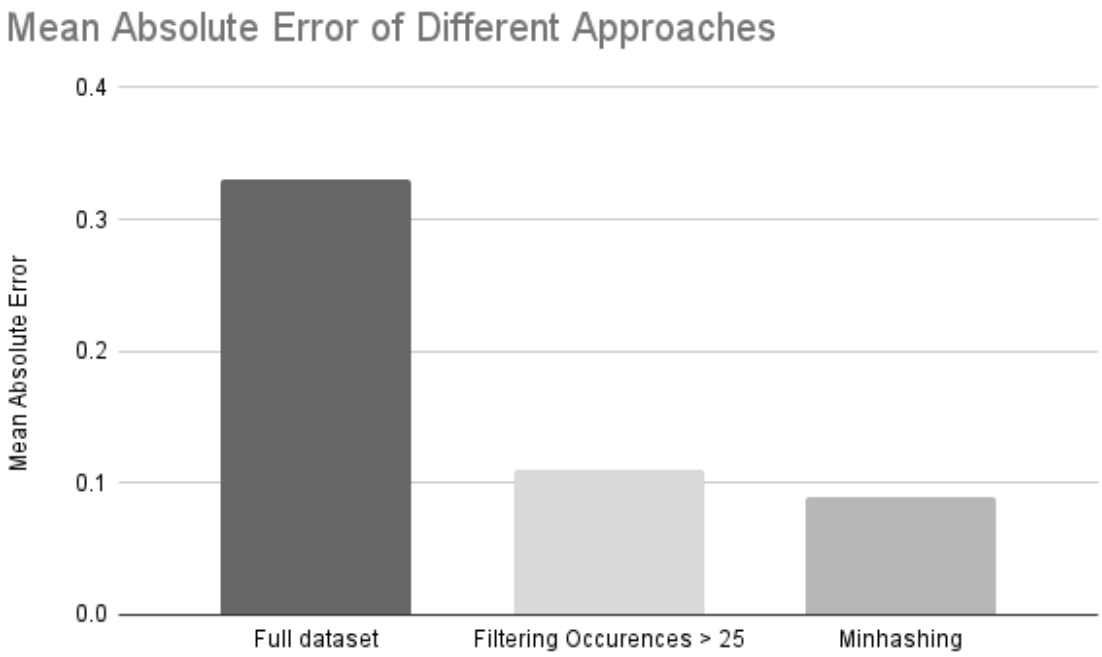
Filter	Percent Under 0.1	Percent Over 0.9
Entire Dataset	42.88396086	45.42350806
≥2 Occurrences	34.7604855	37.16830402
≥ 25 Occurrences	0.600455703	0.9328508243
≥ 50 Occurrences	0.06236033882	0.1351140675

This table shows the percent of the dataset with extreme win rates when the data is filtered in multiple ways. For each filtered dataset, the percentage of states recording a win rate of less than 0.1 or more than 0.9 were recorded. In the unfiltered dataset, almost all the states have extremely high or low win rates because these states are rarely seen in a game. However, when only states which appear greater than or equal to 25 games out of over 1 million are considered, less than 2% of the states have extreme win rates. Of course, it's impossible for a state which has only appeared once to have any win rate other than 0 or 1. In addition, a state with low sample size is more likely to have a skewed sample win rate than one which has appeared more times.

Thus, states under a particular threshold were excluded from the training and test sets for having inaccurate win rates. Indeed, error rates improved from over 20% to around 5%. However, this solution resulted in a massive waste of data. The entire dataset consisted of a total of 1791973 game states. Filtering the dataset to only consist of states appearing in at least 25 games left only 37305 behind, resulting in about 98% of the data being essentially discarded. The solution was to find a way to group these very uncommon states in with the more common ones, so that their win data was represented in the final regression model. To accomplish this, the Pyspark.ml minhashing functionality was used. First, the states were transformed into vectors as before. Rather than filtering the dataset before regression, however, the data was split into train and test sets and the training set was further processed. The decision to split the data into these 2 sets before minhashing was because in practice, test data would not be able to be factored into minhashing, only brought in after the training set had been minhashed.

The training set was split into common and uncommon portions. The minhashing model was fit to the common portion, which was the portion of the dataset occurring more than 25 times. Then, the common dataframe and the uncommon dataframe were joined by their approximate similarity. All pairs of a common and uncommon state which had similarity greater than 95% were joined together, so each row of the dataframe had 2 states and 2 sets of statistics for each. Then, the uncommon states were absorbed into the common state - the numbers of wins and occurrences in all the uncommon states similar to the common state were added up, and added to the common state's statistics. This meant that the data did not go to waste, but made the training data much more accurate to the hypothetical underlying probability that the white player would win from a specific game state. In essence, each element in the training set now represented the win rate of not just the particular board it referenced, but all the boards that were

similar to that one.



The mean absolute error of each different approach is represented here. Minhashing improved performance compared to simple filtering from about 0.11 to 0.09 MAE. This also results in a more descriptive model which represents the full scope of the data far better than discarding the vast majority of it. To further improve it, the vectorization process of converting from a chess board to a binary vector could be investigated, as this was not a major focus of my process.