Chess Positional Analysis using Statistical Methods

Dhiraj Khanal

Brown University

Chess, one of the initial challenges examined by artificial intelligence researchers, has evolved significantly over time. Contemporary chess engines such as Stockfish and Google's AlphaZero utilize primarily search-based or greedy algorithms to select optimal moves. This project presents a distinct approach by implementing machine learning algorithms to evaluate chess positions. Rather than choosing the most beneficial move, the project's goal is to create a function that estimates the likelihood of a player winning from a given chess position.

Though search-based chess engines provide a score for each position, it denotes the engine's belief about winning probability based on the Bayesian framework. The aim here is to classify the actual winning probability based on human gameplay.

The development of such a classifier function could potentially streamline the search process in chess engines and provide insights into the nature of the game.

Training examples were sourced from games played by grandmasters, obtained from the FIDE Database comprising over 100 million games played both offline and online. The games are in Portable Game Notation (PGN) format, encapsulating the entire game rather than a sequence of positions. The project's objective to classify positions necessitated the conversion of these PGN games into position sequences. This posed a technical challenge given the large memory footprint of position sequences compared to PGN-encoded games. However, specialized solvers like the stochastic subgradient method enabled us to avoid storing all positions in memory simultaneously.

Data filtering is a significant factor to consider as the skill levels and play styles among grandmasters vary considerably. For this project, the data was pre-filtered to exclude (1) Bullet and Blitz games due to their brief duration, (2) exceptionally short games with fewer than 10 moves, (3) games ending in a draw, and (4) games containing illegal moves or unreported results. Although this project utilises a binary classifier, the approach could be extended to include drawn games.

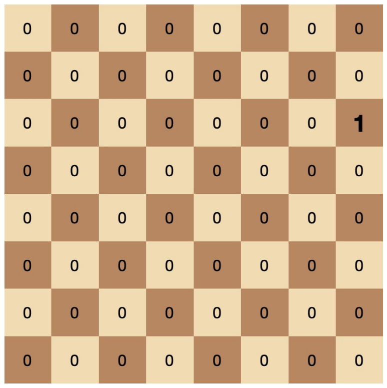
To formalize this concept within the framework of a machine learning (ML) model, we must consider our content $x_{(i)}$ as a legitimate chess position originating from a game played by humans, while our annotation $y_{(i)}$ represents the game's final outcome, whether it results in a victory or defeat for the player-to-move. Our objective is to predict the anticipated value of the game's outcome given the present board position. Notably, due to the human element in these games, the final outcome cannot be simplified to a mathematical function of the board state.

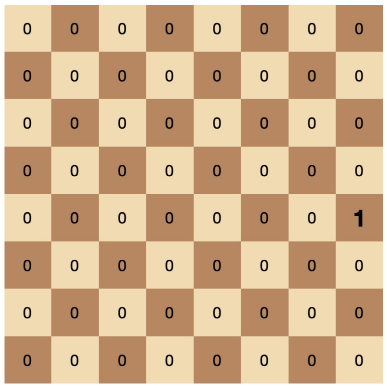
Moreover, the complexity of chess entails that the majority of positions the algorithm encounters will not inherently favor either color. Consequently, these positions are not only unsuitable for training the classifier but also elevate the error rate upon testing the classifier. Given these factors, it would be unfeasible to expect any classifier to achieve a near-zero error rate over this dataset.

The proposed methodology will require the representation of the board state as a multi-dimensional vector of features. Upon careful evaluation of various potential feature sets, I decided to represent the board state as a sparse vector. In this representation, each entry signifies the existence of a particular piece on a specific square.

Delving further into this representation, a chess position comprises 64 squares, each potentially housing a piece. There are six distinct pieces: the pawn, the knight, the bishop, the rook, the queen, and the king. Furthermore, each piece can be either white or black, generating a total of 12 unique pieces. We can thereby express a board state as a partial function in:

$$f(s) \in (1, 2, ..., 8 \times 1, 2, ..., 8) \rightarrow (\{P, N, B, R, Q, K\} \times \{White, Black\})$$


(a) Black King


(b) White King

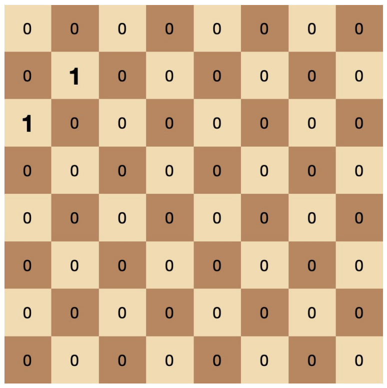

(c) Black Pawn

Figure 1: Sparse Matrix Representation of position given in the figure below

If this partial function describes our game state, it follows (from the definition of a partial function) that we can equivalently describe the game state as a subset of:

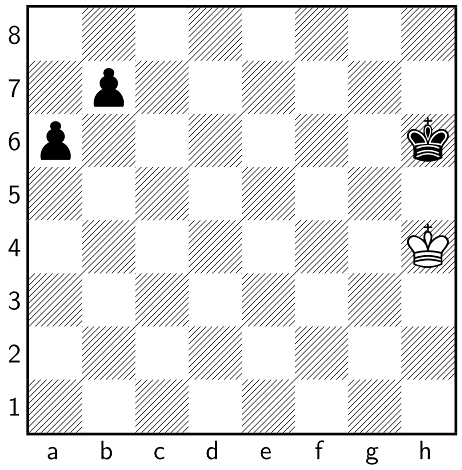
$$\{1, 2, ..., 8\} \times \{1, 2, ..., 8\} \times \{P, N, B, R, Q, K\} \times \{White, Black\}$$

Given that this set has a magnitude of $8 \times 8 \times 6 \times 2 = 768$, it can be inferred that we can represent any subset of it as a vector in $\mathbb{R}^{768}$. In this vector, the i-th entry is either 0 or 1, denoting the presence or absence of the i-th element of this set in the subset.

I proceeded to convert each position into a Forsyth–Edwards Notation (FEN) representation, a concise string notation of a board, using the chess library available in Python. For reference, the FEN for the starting position is provided.

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1)

Each chess position is transformed into a sparse matrix. Below is an illustration of a chess position and its corresponding sparse vector representations. All other pieces maintain a value of 0 for each ith position.



This representation, however, presents a few limitations. Specifically, the partial function does not incorporate certain non-position-based aspects of the game state: the turn order, the possibility of castling, and the potential for enpassant. I have made the assumption that the latter two elements are of negligible significance and can thus be safely excluded as features. The turn order, on the other hand, poses a more challenging obstacle. This challenge, however, can be overcome by leveraging the symmetry of the chessboard. If we reverse: (1) the board, (2) the colors of all the pieces, and (3) the result, our classifier should yield the same outcome. Hence, without loss of generality, we can restrict our analysis to positions where it is white's move. This representation proves quite effective for such positions. (In the following analysis, "white" refers to the player whose turn it is to move, while "black" refers to the other player.)

We postulate that, given any particular position, the probability of the human player to move winning the game can be modeled as a function of the position. This postulation appears quite reasonable given the structure of the chess game. If we further hypothesize this function to be a logistic function of the board representation with some parameter λ, then the maximum likelihood estimate of this parameter aligns with logistic regression. Consequently, I trained a logistic classifier using this representation.

I performed logistic regression over a dataset consisting of approximately 500,000 training samples and 500,000 test samples. I implemented my algorithm in Python using numpy and scipy to carry out the regression method. I managed the relatively substantial volume of data by employing sparse matrices whenever feasible, particularly for the training matrix X, where columns are sparse position vectors. The algorithm typically converged within about 100 iterations for the datasets tested. Here are the accuracy results I obtained:

```
training accuracy = 0.68, testing accuracy = 0.69
```

Run time: — 103.05 seconds —

The close proximity of these figures suggests that the model is not overfitting the data. Despite these error rates being less than ideal, the fact that they are significantly lower than 50% over a large dataset of positions implies that the logistic classifier has yielded a result that would be a valuable attribute for position value.

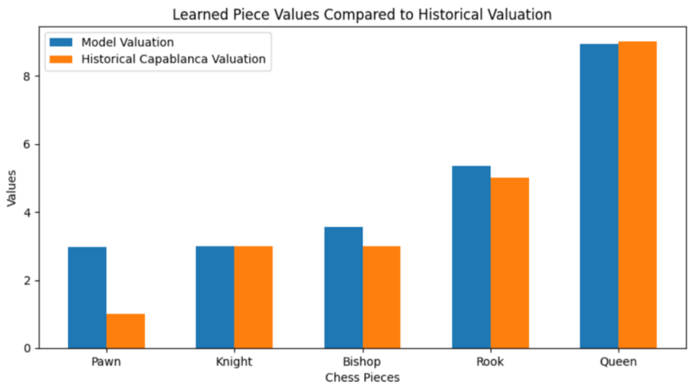
One of the benefits of our sparse representation is its ability to average the calculated weight parameters for a specific piece. This capability enables us to determine the perceived "value" of a given piece by the classifier. We tally the number of each piece on the board and examine how having a piece advantage influences victories or defeats. Based on the weights assigned to a specific piece in each position, we calculate the average

```
      Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                    win   No. Observations:               118000
Model:                            GLM   Df Residuals:                   117994
Model Family:                Binomial   Df Model:                            5
Link Function:                  logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                    nan
Date:                Tue, 10 Aug 2021   Deviance:                   9.6993e+06
Time:                        17:24:27   Pearson chi2:                 5.32e+20
No. Iterations:                   100
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept  -1.368e+15   1.99e+05  -6.87e+09      0.000   -1.37e+15   -1.37e+15
I(P - p)    1.233e+15   2.03e+05   6.09e+09      0.000    1.23e+15    1.23e+15
I(N - n)    1.652e+15   5.14e+05   3.21e+09      0.000    1.65e+15    1.65e+15
I(B - b)    2.077e+15   5.24e+05   3.97e+09      0.000    2.08e+15    2.08e+15
I(R - r)    2.812e+15   7.02e+05   4.01e+09      0.000    2.81e+15    2.81e+15
I(Q - q)    3.949e+15   1.38e+06   2.86e+09      0.000    3.95e+15    3.95e+15
==============================================================================
--- 31.053167819976807 seconds ---
```

Upon normalizing the knight to the value 3, we arrive at the following weights for each piece:

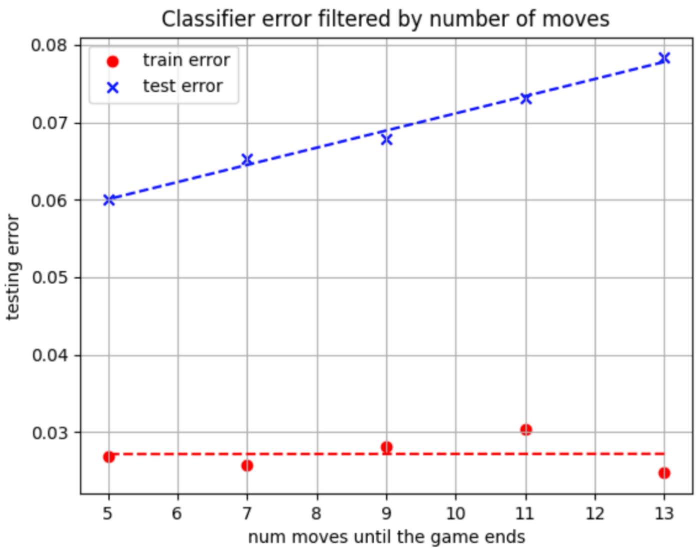
```
pawn = 2.23 ,knight = 3.0 ,bishop = 3.77
```

`rook = 5.10 ,queen = 8.17`



This coincides remarkably closely with the widely accepted Capablanca System of chess piece valuation, which values pawns at 1, knights at 3, bishops at 3, rooks at 5, and queens at 8 (figure 2). The independent derivation of a valuation akin to the Capablanca System by the linear regression algorithm suggests it is conducting meaningful classification.

The error rate for our classifier is still relatively high. While a substantial portion of this error can likely be attributed to player errors in the individual games, it's worthwhile to ascertain the extent of this contribution. If we hypothesize that player errors occur relatively uniformly over time, it follows that positions closer to the end of the game are less likely to be influenced by subsequent player blunders. Such blunders can lead to "mislabeling" of positions, skewing the expected outcome. This mislabeling increases the classifier's error rate. Therefore, if we filter the original dataset to only include positions from the final n moves of a game, we would expect the classifier error to decrease as n diminishes. The chart below depicts classifier error as a function of move filtering threshold.



Given that the testing error rate drops to approximately 6% within the last 5 moves of the game, it's apparent that around half of our general classifier error may be due to player errors causing "mislabeling" of the test data. Our classifier's ability to generate this curve signifies that it is performing adequately without overfitting to individual data subsections.

This curve could also be interpreted as a representation of the frequency of player errors over time, as a function of the proximity to the end of the game. An intriguing area of future research could involve examining how this curve varies for players of different strengths.

The classifier performs reasonably well for general case positions, successfully predicting the winner in two-thirds of positions. This is impressive, given the high frequency of drawn or unclear positions. Our analysis suggests that roughly 12% of the classifier error is due to human errors in the test set. The remainder likely results from unclear or volatile positions that are challenging for a position-based classifier to interpret.

Our algorithm has managed to learn a system of piece weights, which has been established knowledge for players over centuries, with high accuracy. Its results for pawn placement values seem intriguing, particularly in regards to variance by file, where the classifier results somewhat defy conventional wisdom on the subject.

Future endeavors could explore the discovery of novel features for this problem, though I haven't yet found any that yield satisfactory results. It could also be worthwhile to examine the performance of this function as a heuristic in a chess engine, using reinforcement learning feedback based on the engine's performance to adapt the heuristic accordingly.