Machine Learning Project - Elo Prediction

CSU4406, The University of Dublin, Trinity College, Ireland, 2020-2021

Jevgenijus Cistiakovas
cistiakj@tcd.ie
17325426

Oisín Nolan
oinolan@tcd.ie
17327530

Anton Yamkovoy
yamkovoa@tcd.ie
17331565

***Index Terms*—Machine learning, Chess**

I. Introduction

This project explores the possibility of predicting chess players' Elo ratings[1] given the moves of a chess game and other game information. This contrasts with how Elo ratings are normally calculated. A model capable of making such predictions could provide interesting insights regarding chess play-style, and useful applications such as cheater detection in online chess games. The input to our algorithm is a chess game stored in the Portable Game Notation[2] (PGN) format, and the output is two Elo ratings: one for white and one for black.

II. Dataset and Features

A. Data Source

FICS Games Database[3] was used as a source for chess game data in PGN format. It provides all the games played on FICS since November 1999. However, relatively small datasets were used for this project.

The datasets were cleaned by removing all games with less than 4 half-moves, bullet games and blitz games, leaving only standard games with ≥ 4 half-moves. The reasons are that ratings for different types of chess such as blitz and standard chess are not directly comparable, and 4 half-moves is the least number of moves in which it is possible to reach a checkmate. The final cleaned datasets used were a training dataset of approx. 50K games, and a smaller test dataset of approx. 30K games. Elo ratings typically follow a Gaussian distribution: see Fig. 1 for distribution of Elo in training data; Fig. 7 in Appendix for that of the test data.

The *python-chess*[4] library was used to parse the PGN files and interact with chess games through its core API. See Appendix *A* for an example of a chess game stored in PGN.

[1]https://en.wikipedia.org/wiki/Elo_rating_system
[2]https://en.wikipedia.org/wiki/Portable_Game_Notation
[3]https://www.ficsgames.org/download.html
[4]https://python-chess.readthedocs.io/

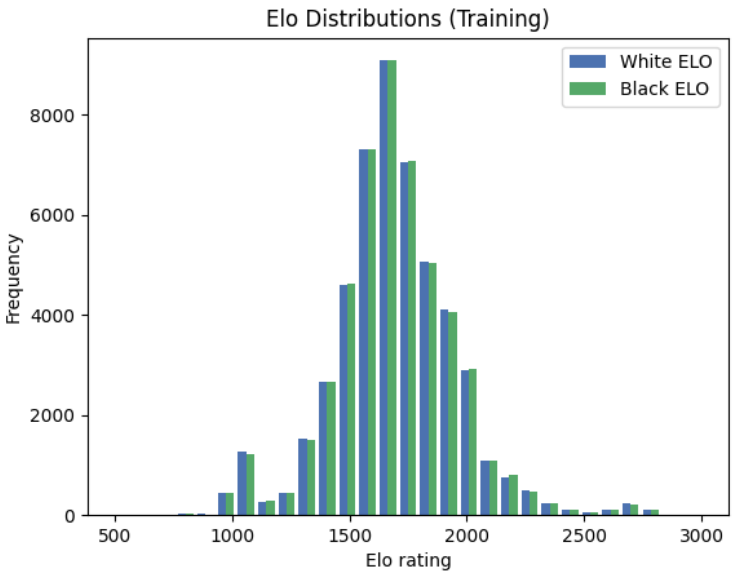

Fig. 1: Elo frequency.

B. Feature Extraction

A number of approaches to feature extraction were considered, exploring the many possible ways to encode a chess game and the various game information that might be relevant to Elo ratings. One feature we wanted to extract was the *advantage* of each player at a given board state. We had the option of using the chess engine *Stockfish*[5] to calculate advantage locally, however this calculation became increasingly time consuming to complete due to the time limit restrictions required by the engine for suitable evaluation, which is, at a minimum, 0.5 or 1 second to achieve performance for the engine moves. Given that we were using a large data-set, we decided to implement our own, less computationally expensive method, described in *short features*.

- **Short features** – This approach considers basic game information, such as *game result, number of moves, number of checks, game ending*, as well as statistics describing the strength of both players' positions throughout the game. Board position

[5]https://stockfishchess.org/

strength was estimated using the relative value of chess pieces and the strength of their position on the board[6]. Thus, a time series of player advantage is generated for both players in each game, for which statistics including *mean, variance, min/max element, skew, length of maximum increasing/decreasing subarray* are calculated. An example plot of player advantage time series is available in Appendix *B*.

- **Game2vec** – This algorithm encodes a given chess game as a single vector. It does so by encoding each board state as a matrix containing 0s on squares with no pieces, and numbers identifying piece type on squares containing pieces. Each board state matrix is then flattened to become a vector, and then each board state vector in the game is concatenated to form a single game vector.

  The intuition for this approach is that patterns in board position may correlate with Elo. For a simplified illustration of this idea, see Fig. 8 in Appendix, which shows the average *activity* of each board square over a number of games by players within a certain range of Elos. In this case, *activity* means proportion of moves for which a given square is occupied by a piece.
- **Text features** – In the PGN format, the moves of each game are represented as *movetext*, a string written in Standard Algebraic Notation[7] that fully describes the moves of that game. This approach used a one-hot encoding of the movetext for the chess game as a feature.

III. Methods

The problem considered is a regression problem. The only difference between this problem and a classical regression problem is that the output required is a vector of size two (Elo for both black and white player). The alternative to this would be to solve a classification problem for each white and black, dividing the possible Elo range into bands, which seems less intuitive. All the methods considered were implemented using the *scikit-learn* machine learning library for Python. The following subsections describe the learning algorithms used to approach this problem.

A. Linear Regression

A linear model is given by $h_\theta(\boldsymbol{x}) = \theta^T \boldsymbol{x}$, where $\boldsymbol{x}$ is a feature vector and θ is a vector of weights. Given a list of pairs $(\boldsymbol{x}, y)$ of input features and expected outputs, linear regression fits a linear model with weights θ to minimise a cost function. In particular, an ordinary least squares cost function is used, defined as $\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$, where m is the number of samples. For optimisation, gradient descent or coordinate descent algorithms can be used.

One problem with simple linear regression, especially when there is a large number of features, is that it is prone to over-fitting. This problem is addressed by introducing a penalty term into the cost function, where penalty is a function of θ. The cost function is then given by $J(\theta) = L(X, y, \theta) + \frac{P(\theta)}{C}$, where $L(X, y, \theta)$ is a loss function (e.g. ordinary least squares) and $P(\theta)$ is a penalty function, and n is the number of features. Three variations of linear regression with penalty were considered:

- **Ridge regression** – Penalty is given by $P(\theta) = \frac{1}{C}\theta^T\theta$. This results in a tendency of preferring solutions with small feature weights.
- **Lasso regression** – Penalty is given by $P(\theta) = \frac{1}{C}\sum_{j=1}^{n}|\theta_j|$. This results in a tendency of preferring solutions with fewer non-zero weights. That is effectively removing certain features.
- **ElasticNet** – ElasticNet regularization combines linearly the L_1 and L_2 penalties used correspondingly in the lasso and ridge methods the penalty is calculated using $(\alpha * L_1) + ((1 - \alpha) * L_2)$

Other issues with linear models include assumed linearity, which can, to an extent, be addressed by introduction of additional polynomial features, and the fact that the potential relationship between the output variables is not considered.

B. k-Nearest Neighbours Regression (kNN)

A *k*NN regression model calculates output predictions using the k training examples nearest to the input in the feature space. The k nearest training examples are identified using a distance metric $d(x^{(i)}, x)$ which calculates the distance between the input feature vector x and a given training example $x^{(i)}$. The neighbouring points may be weighted such that their contribution to the output is a function of their distance from the input. A Gaussian weighting function, for example, would be given by $w^{(i)} = e^{-\gamma d(x^{(i)},x)^2}$. The output of the model is given by a weighted mean of the k nearest neighbours' outputs: $\hat{y} = \frac{\sum_{i\in N_k} w^{(i)} y^{(i)}}{\sum_{i\in N_k} w^{(i)}}$.

The choice of k can mitigate over/under-fitting. A drawback evident in *k*NN models is that in order to find the k nearest neighbours for a given input, the algorithm must search over all training data. Thus, if there is a lot of training data, prediction can be slow.

C. Convolutional Neural Network (CNN)

CNNs are a type of deep neural network that normally consists of convolutional, pooling and fully-connected layers. Convolutional layers make it particularly suitable for processing images and other 3d tensors. Being a

[6]https://www.chessprogramming.org/Simplified_Evaluation_Function

[7]https://en.wikipedia.org/wiki/Algebraic_notation_(chess)

deep neural network, CNN can learn non-linear as well as linear relationships. CNNs are normally trained with stochastic gradient descent using backpropagation for computing the gradients.

IV. Experiments

Hyper-parameters for the models were chosen using 10-fold cross validation. As mentioned in *section II*, the data used consisted of two datasets. A training dataset of 50K games was used for the final training of the models. It was also used for hyper-parameters tuning, where it was split into training and validation sets. Another dataset of 10K games was used for final testing that allowed for unbiased performance evaluation.

Performance of the models was quantified using a mean squared error (MSE) metric, given by $\frac{1}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2$ (TODO: should we argue why?). In the problem considered, there are two output values, thus the final MSE is the average over MSEs for each output. For readability and presentation purposes, the models were also scored using R^2 score. It is given by $R^2 = 1 - \frac{\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2}{\sum_{i=1}^{m}(\theta^T x^{(i)} - \hat{y})^2}$ where $\hat{y} = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$, and presents the same information as MSE in such a way that it is easier to interpret.

For the baseline model, a simple model that always predicts the mean value was used. Given that the distribution of Elo scores is close to Normal with a relatively small standard deviation, this simple model gives a strong baseline.

The following sections discuss hyper-parameter tuning for this problem using the aforementioned models.

A. Linear regression

There are three hyper-parameters for basic linear regression. First, the maximum number of moves considered in a game. Second, the feature space can be expanded by an addition of simple polynomial features. They are either polynomials up to degree q, e.g. $x_1^2 x_2^1$. Or simple products of up to q features (i.e. max. power of 1), e.g. $x_1 x_3$.

It was determined that additional polynomial features decrease performance due to over-fitting. While the optimal number of moves is 30 as shown on Fig.2.

Note that the features after being extracted from the games, are first standardised and only then are used for training the model. Standardisation results in all the features having a mean of 0 and standard error of 1. This is needed as the features used have different scales.

B. Ridge regression

There are the same hyper-parameters to be chosen as in simple linear regression. Additionally, there is also a hyper-parameter C controlling the weight of the penalty. The higher the C, the lower the weight of the penalty and higher chance of over-fitting. Thus, a lower value of C was preferred.

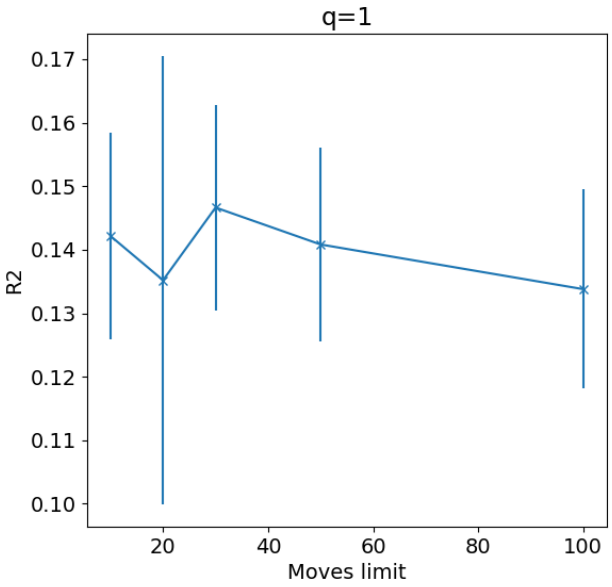

Fig. 2: Cross validation for #moves without polynomial features in Linear regression.

By cross-validation, it was determined that the optimal hyper-parameters are $\#moves = 50$, $C = 0.001$, and features expanded by polynomial features with max degree $q = 2$. Features are standardised.

Within Sci-kit learn, the standard solver for ridge regression models does not necessarily use gradient descent. The best performing algorithms for the used datasets were Cholesky decomposition that finds a closed form solution and an iterative conjugate gradient solver.

C. Lasso / ElasticNet

Lasso uses coordinate descent. Similarly to the previous models, we used the already set hyper-parameters relevant to the full dataset such as the number of moves as 50, we also used polynomial features to the maximum degree of 2, and a cross validated value for the C = 1. ElasticNet regression uses a combination error penalty of the L_1 and L_2 penalties from lasso and ridge regression. With suitable values for C in elastic net being dependant on the l1 ratio, and based on the similarity with a previous model.

By tuning the l1_ratio parameter in elasticnet regression, we can transform it into lasso / ridge, with 0.5 being a even mix of both, we found that there were multiple operational configurations for elastic-net bringing it closer to the lasso or ridge, we could achieve similar performance to them on the second dataset. With a balanced version performing worse than either separate model.

D. k-Nearest Neighbours

A *k*NN model was used on *game2vec* features, described in *section II B*. In essence, this model predicts players' Elo ratings by looking at games with similar positions to that of the input game. Because the input data feature vector exists in an integer-valued vector space (integers used to represent categorical data), we use Hamming distance as a distance metric. The Hamming distance between two equal-length vectors is defined as the number of entries at the same position that have different values. In *sklearn* this is implemented as the proportion of corresponding entries that are unequal: $d(x^{(i)}, x) = \frac{\#unequal(x^{(i)},x)}{\#(x)}$, where $\#unequal$ gives the number of corresponding entries that are unequal, and $\#(x)$ gives the number of entries in x (and $\#(x) = \#(x^{(i)})$). The number of neighbours considered, k, and the number of moves of a game encoded in the *game2vec* algorithm, $\#moves$, were considered important hyper-parameters for this model. $\#moves$ was considered a hyper-parameter because it determines the dimension of input to the model, which can affect the performance of *k*NN models. If the input vector exists in too large a vector space, the algorithm may not be able to identify any nearby examples. This is relative to the number of training examples given to the model; more training examples are likely to span more of the feature vector space. 10-fold cross-validation on 1K training points was used to choose values for these two hyper-parameters. Ideally all 50K training points would have been used, but due to the time cost associated with *k*NN this was not feasible. Fig. 3 shows a plot of the R^2 score achieved by a model with a variety of values for k and $\#moves$. After inspecting the plot and the associated matrix of R^2 scores, it was determined that optimal performance is given by $k = 12$, $\#moves = 35$.

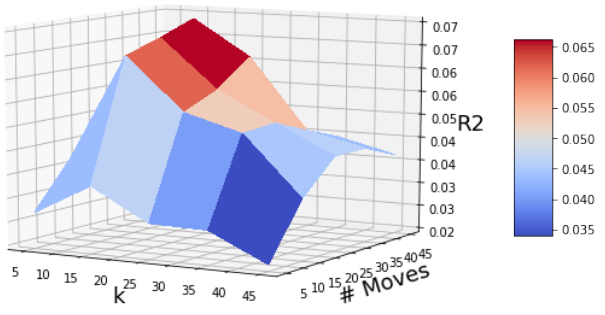

Fig. 3: Cross-validation plot showing R^2 score for a given pair of $(k, \#moves)$.

Additionally, a Gaussian weighting function was used to determine the level of contribution made by different neighbours. As mentioned in *section III*, this function is given by $w^{(i)} = e^{-\gamma d(x^{(i)},x)^2}$. Here, we consider γ a hyper-parameter, and so it was tuned once again using 10-fold cross-validation, on a model using the previously chosen values for k and $\#moves$. Fig. 4 shows and error-bar plot indicating for each γ used the resulting R^2 score. Using this and similar plots, the value $\gamma = 75$ was chosen.

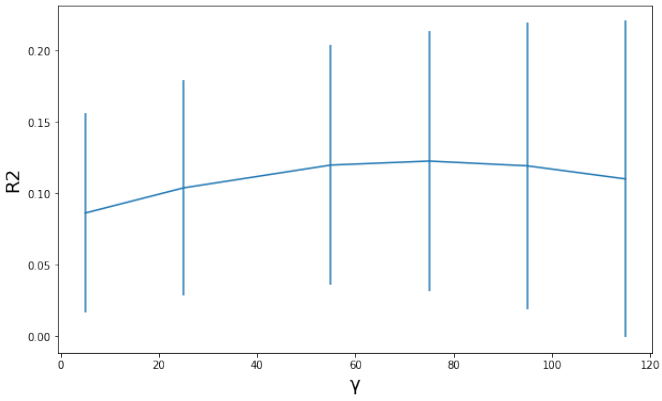

Fig. 4: Cross-validation error-bar plot showing R^2 score for a given γ.

Text features were also experimented with as input for a *k*NN model, although it did not show promising performance and thus is not discussed further here (implementation details available on *GitHub* repo).

E. CNN

The idea of using CNN was abandoned due to lack of computational resources and underwhelming performance during limited testing. However, we do believe that this is a potentially viable approach. One of the benefits of using CNN is that unlike naive linear models it can account for any possible relationship between the output variables.

V. Results

The results are given in Table I. It shows the values of MSE and R^2 score as measured for both 50K training data points and 10K unseen test data points. The models used hyper-parameters are given in the previous section. All the linear models were using *short features*.

Predictions made by the *k*NN model are visualised in Fig. 5 by a histogram comparing actual vs predicted Elo ratings.

It should be noted that out of all the models considered, kNN with *game2vec* took a significantly larger time for both training and prediction generation. Times cannot be quantified correctly as models were trained/tested in slightly different conditions. However, without considering time needed for feature extraction, the linear models took time in the order of seconds, while kNN took time in the order of tens of minutes. Difference in prediction time is on the order of $O(10^4)$.

TABLE I: Results

Algorithm	Train MSE	Train R2	Test MSE	Test R2
Baseline (mean)	69004	0.0	71997	-0.0037
Linear regression	58558	0.1514	63235	0.1184
Ridge regression	52236	0.2430	58213	0.1885
Lasso regression	53543	0.2240	58788	0.1803
ElasticNet regression	56668	0.1787	61644	0.1405
kNN with *game2vec*	9177	0.8670	48247	0.3274

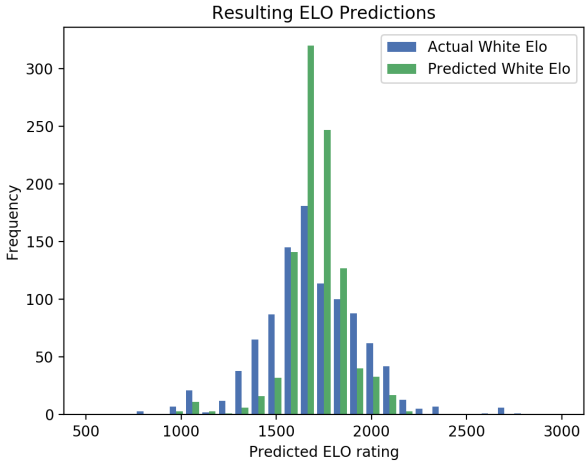

Fig. 5: Distrubtion of actual test Elos vs *k*NN predicted Elos.

VI. Discussion

Results presented in the previous section reveal that the problem being tackled is inherently complex as none of the approaches considered achieved a very small MSE or a very high R^2 score.

With respect to the baseline, all the models considered compare favourably. Both kNN model and all of the linear models perform better than the baseline on both training and test datasets. The R^2 scores give a good indication on how the models perform with respect to the baseline and to the ideal model.

Specifically, kNN model with *game2vec* features performs the best on the training dataset. It has more than 7 times lower MSE than the baseline, and more than 5 time lower MSE than the second best performing model (ridge).

With the test dataset, kNN is still the best performing model despite a significant drop in R^2 score. In fact, all the models have the same rank for both datasets. Thus, the metrics on the training data correctly predicted the expected performance.

Out of all the models, kNN had the largest relative drop in performance. Meanwhile linear models showed a more graceful degradation of performance.

Good performance of the kNN model indicates that the presence of certain board states can be indicative of a given Elo rating. The predictive accuracy would likely increase if the model as the model is given more training data, although the increase in time taken to train and predict with larger *k*NN models would likely make this approach infeasible.

Fig. 5 shows that the *k*NN model makes predictions closer to the mean than it should, i.e. the predictions have a smaller standard deviation than the actual Elo ratings. In future, it may be possible to improve the model with this in mind, by focusing more on those features that identify particularly high and low Elos.

VII. Summary

In this project, the problem of estimating Elo ratings of both players from a single chess game was tackled. Multiple approaches to extracting features from the chess game description in PGN format were considered. Similarly, multiple supervised machine learning algorithms were built and assessed.

In particular, a kNN model working with chess games encoded as series of board states was found to be the best performing. Different variants of linear models that work with shorter feature vector consisting of different statistics about the game were found to have worse performance. However, linear models exhibited a less steep performance degradation on a withheld dataset and had a better time complexity.

The key conclusion drawn from this project is that the problem tackled is inherently complex, and much of the complexity lies in the choice of features. The resulting algorithms are unlikely to be of much use for real world applications due to low performance. Future work includes exploring the viability of deep learning approaches on a much larger dataset.

VIII. Contribution

- **Jevgenijus Cistiakovas** – worked on the *short features*, ridge regression and simple linear regression. Did some experiments with CNN. Worked on code/report parts concerning these topics as well as contributed to the *Discussion* and *Summary* sections.
- **Oisín Nolan** – worked on code/report for *k*NN model. Implemented *game2vec* feature extraction function. Implemented & experimented with *text features*. Implemented extraction of game metadata

features for *short features*. Worked on *Introduction* and *Dataset and Features* sections of report.

- **Anton Yamkovoy** – worked on implementing board state evaluation using stockfish and through the simplified board evaluation function described previously for *short features*, tested experiments regarding *Lasso* and *ElasticNet* regression models.

IX. Code

Github link – https://github.com/OisinNolan/Elo_Predictor

X. Appendix

A. Example of PGN format

```
[Event "FICS rated standard game"]
[Site "FICS freechess.org"]
[White "GriffyJr"]
[Black "ssingh"]
[Result "1-0"]
[BlackElo "1627"]
[PlyCount "63"]
[WhiteElo "2119"]

1. e4 e5 2. Nf3 Nc6 3. Bc4 d6 4. c3 Bd7 5.
    Qb3 Qe7 6. Qxb7 Rc8 7. Bd5 Nb8 8. O-O
    Nf6 9. Bc4 Be6 10. Na3 Qd7 11. Re1
   Bxc4 12. Nxc4 Be7 13. Qxa7 O-O 14. d3
   Qg4 15. Ne3 Qe6 16. Ng5 Qd7 17. Nf5 h6
    18. Nxe7+ Qxe7 19. Nf3 Nbd7 20. Be3
   Ng4 21. a4 Nxe3 22. Qxe3 f5 23. a5 f4
   24. Qd2 g5 25. b4 g4 26. Qa2+ Kh7 27.
   Nd2 f3 28. gxf3 gxf3 29. Kh1 Qf6 30.
   Re3 Qg5 31. Rg1 Qf4 32. Rxf3 { Black
   resigns } 1-0
```

B. Advantage evaluation graph

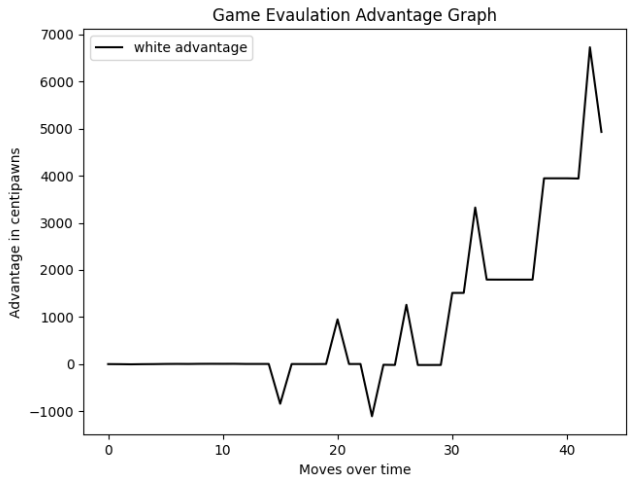

Fig. 6: Example of an advantage graph for a single game. From this type of graph statistics were calculated (set move limit per game)

C. Test data Elo distributions

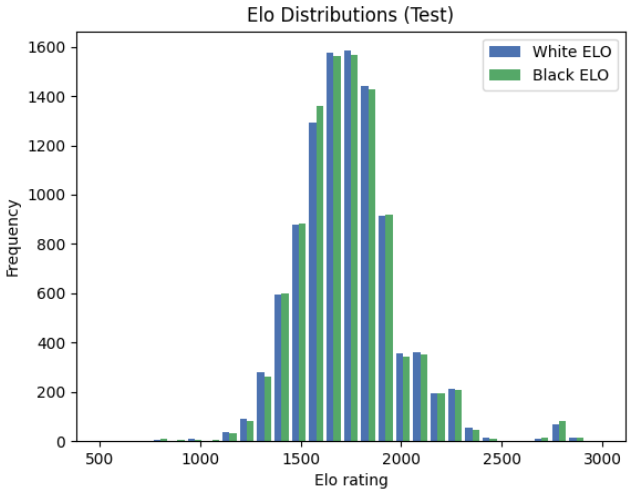

Fig. 7: Test data Elo distributions for white and black correspondingly

D. Board activity by Elo range

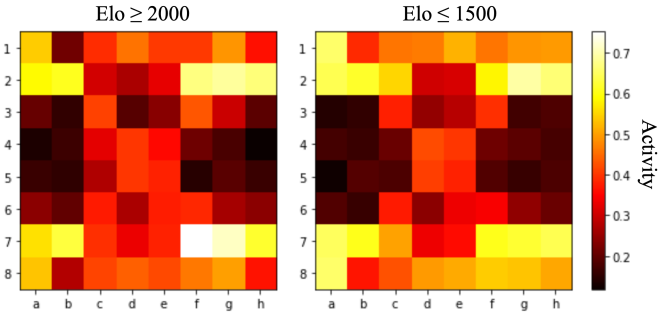

Fig. 8: Avg. board activity for Elo $\geq$ 2000 vs $\leq$ 1500