Háskólinn í Reykjavík
Datamining And Machine Learning
Michal Borsky

Chess Predictor

Group project

Group Members: *Joni Seraj, Tomáš Vozníčka*

1. Problem Description

Chess is a game full of patterns, that means it does offer a lot of options in the field of data mining. We decided to work with a dataset of chess games, that contains about 20000 played games recorded at the site Lichess.org, an online chess website. For each game, we also have a bunch of data such as player ratings etc.

We aimed to use the data we have to train a classifier to be able to “predict” the outcome of a chess game based on set of parameters we’ll derive from the dataset. Finding out the winner of a game is a very interesting topic to explore. It can prove really useful for statistics of championships, betting etc.

2. Data sets and tools used

Our software requirements are:

1. Python
2. Numpy Library
3. Panda Library
4. Sklearn Library
5. Chess Library for python [4]
6. Git for development

For the realization of our project, we used two datasets, one from the source of keggle and the other from the database of chess games of FICS[3]. Our main one is from Kaggle [2], it contains wide range of rated/unrated games played by players with chess rating mainly from 1200 to 1800. Data set contains following variable:,
Game id - unique identifier of the game at the site.
Rated - true or false value depending whether the game was rated or not.
Start time, **End time** - time records of start and end of the game.
Numbers of turns - Number of turns until the final state
Game status - the way the game ended.
Winner - value of who won the game (white/black) or draw.
Time Increment - standardized chess definition of game timing.
White player id, **Black player id** - unique identification of players on the site.
White player rating, **Black player rating** - value of player rating on the site.
Opening Eco - standardized id of opening sequence of game).
Opening Name, **Opening ply** - number of turns in opening sequence.
Moves - All moves of the game in SAN notation.

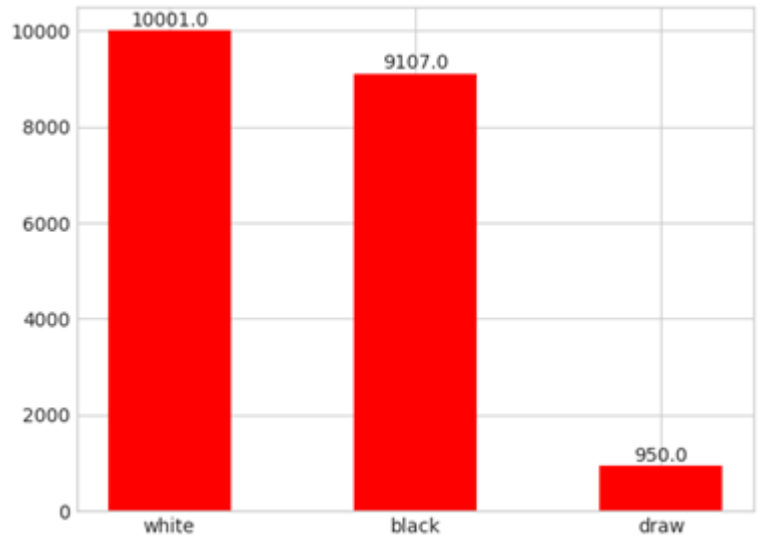
We are using this data set as our main data for training and testing model.

Our other data set is a collection of data from FICS Games Database [3]. This data set contains mostly the same variables as the first one but in a PGN format. This data set offers

wider spectrum of ratings, we are using this data set selectively, by picking games that are outside of rating range of our main data set and using them to test limits of our model.

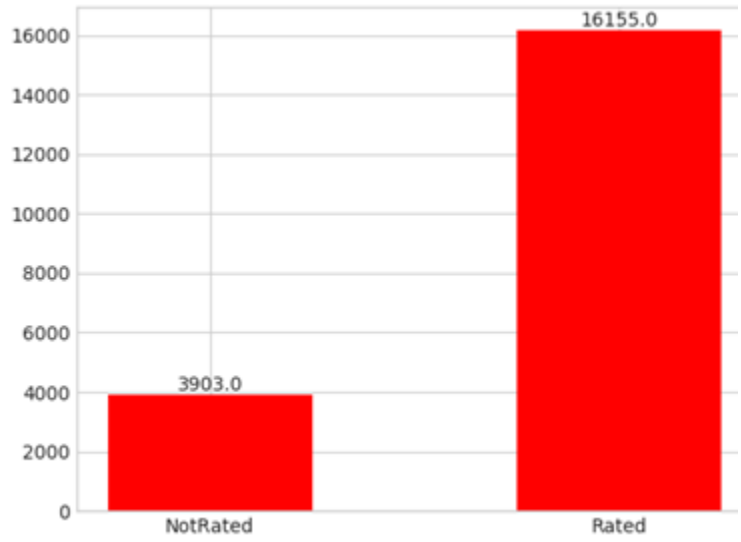
2.1 Data Analysis

To learn more about data we have we decided to do some analysis of several parameters in our data se. This was important for us as it show us some important properties of our data set.



Distribution of our target variable, the winner column.

Distribution of game ratings.



Distribution of rated games

We can see that in regards to the target variable, we have an almost equal distribution between two of it's values, the white and black variables. Meanwhile we have a very small portion of games that ended in draws.

In regard to the game rating distribution, we see most of our games are average rated. Most of them lie in the 1250-2000 range. We seem to have very few games below 1250 and above 2000.

Rating distribution is quite uneven. There are almost 4 times more rated games then non rated ones.

3. Preprocessing data

From our initial dataset, we took as inputs white_rating, black_rating, rated, moves and the opening_eco. Our target value is the column winner. Since winner can only have three possible values, we can see that we need a classifier.

We threw all the data into a pandas dataframe, which is essentially a table where each column represents the data of one input. Then we started transforming it.

The first thing we did, was transform our target value. We have three possible class labels, white wins, black wins or a draw. So we gave them all an index representative of 0, 1, 2 respectively.

After that, we decided we had to work on our input data. We decided we had to turn everything into correct numerical representations.

For the white_rating and black_rating, we combined them into a single variable, final_rating.

Final_rating = white_rating - black_rating

We can see that final_rating is a positive number if the white player is better rated than the black player and negative if vice-versa.

Then we had to turn our opening_eco, which was in string format into their index representatives. So in order to do that, we listed all the unique values in a python array, then indexed them accordingly. By replacing the strings in the pandas dataframe with the index representatives, we got a numerical representative for our data.

Our rated variable was a boolean string, True or False. We solved that easily by replacing the true values with a 1 and the false with a 0.

So all we had to do now was transform the moves variable into a good numerical evaluation. We were faced with the question, how do you evaluate a set of moves in chess? The website chessprogrammingwiki [5] gave us a lot of insight on what to features to evaluate and how to do them. After doing the research, we came up with a evaluation of the moves variable on the following features:

1. **Mobility**
2. **Material**
3. **Central Control**
4. **Pawn Structure**

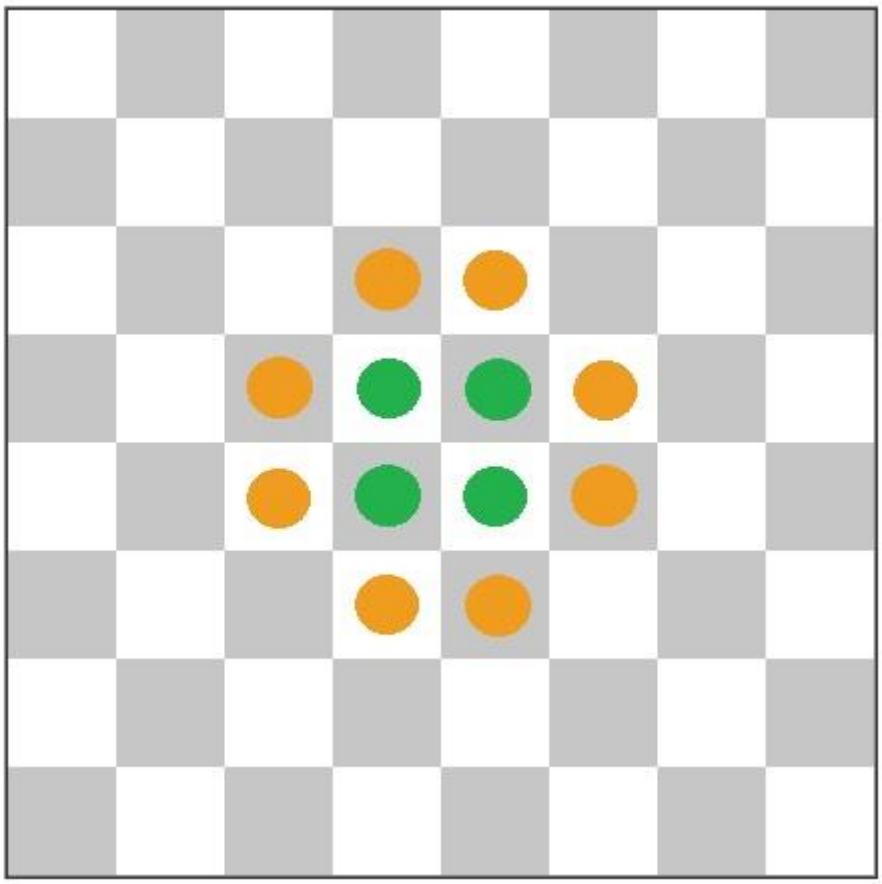
Mobility is an important evaluator for a chess game state. A piece of research conducted by Eliot Slater on mobility of chess games came to the conclusion that the more available move choices you have, the stronger your position is. That piece of research is *Statistics for the Chess Computer and the Factor of Mobility [6].*
In order to evaluate the mobility parameter, we decided to input the difference of legal moves the white player can make at the state of the board with the amount of legal moves the black player can make.

Mobility = WhiteLegalMoves - BlackLegalMoves

Material is another important factor in determining the strength of one's position. The more pieces a player has at hand, the better his position is. So we pulled an evaluator of the difference between the amount of pieces white has versus what black has.

Material = AmountWhitePieces - AmountBlackPieces

Central Control is a known chess strength for players. It basically means who has more control of the 4 central squares of the board. The reason it's so important is that those 4 central tiles allow pieces fast access of most of the board areas, and more movement and possibilities for attack and defense.



We can see on the picture above, all the central squares we considered, We named the 4 central squares C1, C2, C3, C4 and the orange squares S1 to S8. We calculated the central control by taking the difference of control in each of these squares. The control for a single player on a square was measured by attacking potential or occupying that square.

CentralControl = (CS1 – CS1') + (CS2 – CS2') + (CS3 - CS3') + (CS4 – CS4') + (S1 – S1') + (S2 – S2') + (S3 – S3') + (S4 – S4') + (S5 – S5') + (S6 – S6') + (S7 – S7') + (S8 – S8')

Pawn Structure is the metric that evaluates how well each player pawns are connected with one another. One way to evaluate this metric, is through the calculations of isolated, blocked and double pawns.
An isolated pawn is a pawn which has no other pawns in it's one square vicinity.
A doubled pawn is a pawn which shares it's file with another pawn.
A blocked pawn is a pawn which can not move forward because he is blocked by an enemy piece.

Taking the difference between the number of values for each isolated, blocked and doubled pawns of each side, gives us a good numerical representation of the pawn structure.

PawnStructure = (D-D') + (I-I') + (B-B')

With the above mentioned 4 metrics, we now remove the moves column from the dataframe and replace it with the 4 new columns.

The last thing for us to do, is scale the data in between -1 and 1 so our model can make better predictions.

In the end we are left with 7 inputs for our model:

1. Final Rating
2. Opening Eco
3. Rated
4. Mobility
5. Material
6. Pawn Structure
7. Central Control

We have just one target variable, the winner which is our classifier with three possible classes represented by numerical values, 0 ,1 and 2.

4. First model attempts

In the beggining, we did not have the evaluation parameters and we tried some models without the moves as an input. Instead we had two other inputs, *the number of turns* and *game_length* which we calculated by the difference of *start_time* and *finish_time.*
After settling selecting and preprocessing parameters we tried to run some basic classifiers such as Logistic regression, K Neighbors, SVM and Naive bayes algorithm to test the accuracy based on raw data(we did not have the evaluation parameters when we tried these models, which might have contributed to the low results we got). Not surprisingly, accuracy at this stage was just slightly better than random guess and we were getting an accuracy of around 60%. These results showed us that we needed to work on other ways to get more decisive data from our data set, also it prompted us to look at certain parameters that we were working with and see how strong and consistent they are.

While analysing the data, we found out that almost 10000 games in our dataset (almost 50%) had a *game_length* of **0**. So after analysing the data, we saw that the creation and finish times for our games were innaccurate. So we decided not to use the *game_length* as an input, which in retrospective makes sense since the *game_length* would be a metric you would know only after the game is finished. On this principle, we removed *nr_turns* as an input as well.

5. Final Model

We decided to take neural network as our final model. After some experimentation we settled on neural network that is non-recursive and has 5 layers in total, 7 neurons in input layer, 12/8/5 respectively in our hidden layers, and 3 in our output layer. Weights are initialized randomly. Our model uses a rectified linear unit function (3) as an activation function for neurons in hidden layer.

$$f(x) = \max(0, x)$$

Rectified linear unit function activator

For the output layer we are using softmax function (4).

$$\operatorname{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$$

Softmax Function

where Z_i represents the i th element of the input to softmax, which corresponds to class i, and K is the number of classes. The result is a vector containing the probabilities that sample x belongs to each class. The output is the class with the highest probability.

For our algorithm we used the *stochastic gradient descent* to manage the learning of neural network, with learning rate of 0.001.

The time complexity for our NN would be **O(n*m*hk*o*i)**, where ***n***-number of samples, ***m***-number of inputs, ***h***-number of neurons, ***k***- number of hidden layers, ***o***- number of output neurons and ***i*** number of iterations.

The use of this model has the following disadvantages:

1. Different random weight initialization leads to different accuracy results.
2. This model requires tuning of the parameters, such as how many hidden-layers to use, how many iterations etc.
3. This model is sensitive to input scaling, which we fixed by scaling the data.

6. Results

After splitting the data randomly by 50% for our training and testing sets, we came to a consistent 88% accuracy using our NN, with a training time of around 1 minute. That accuracy satisfied us, since we ran it over 10000 games of our testing set. We ran the program over 300 times and found out that our accuracy was consistently on a mean of 88% with a variance of 1.

The accuracy results were computed by taking the predictions our model came out with and differentiating them with the real ones. Then we found the percentage the model predicted correctly.

Our model and main data set has some limitation, with most of our games around 1500 rating we have decent predictor for winner, however we are getting lower accuracy when dealing with games that are played outside of our rating range. We believe that the drop in accuracy is partly dependent of our evaluation algorithms, as they might not be suited for such games.

This we proved by selecting games from the second dataset of ratings bigger than 2000 and found out an accuracy drop to 70-75%.

We do however come to the conclusion that a neural network is a good model to predict game chess winners and it tells us accurately the importance of features in solving the problem
.
Overall we are pleased with how the project turn out, although we know there are ways to improve it, which are listed at the future works section of this report.

The code we used for the program can be found on BitBucket, it's a public repository. To see the results yourselves, run the project.py.

7. Future Work

One of the things that could be improved in our project is our evaluation function. Instead of only evaluating the 4 metrics we did, you could extend it to:

1. King Safety
2. Tempo
3. Game-Phase Consideration
4. Connectivity
5. Trapped Pieces

Those are some of the possible extensions one could choose to improve the evaluation function.

Another possible thing to do in the future, is extend the predictability of the winning to not only who wins, but also how the player wins. In order to do this, one would need access to a player's match history and be able to predict how a player reacts to certain situations such as, does he resign when he makes mistakes, is he a hopeless optimist who plays the game hoping that his opponent will make mistakes etc.

8. References

[1] http://scikit-learn.org/stable/index.html
[2] https://www.kaggle.com/datasnaek/chess
[3] http://www.ficsgames.org/download.html
[4] https://pypi.python.org/pypi/python-chess
[5] https://chessprogramming.wikispaces.com/Evaluation
[6] http://www.eliotslater.org/index.php/chess/147-statistics-for-the-chess-computer-and-the-factor-of-mobility
[7] http://scikit-learn.org/stable/modules/neural_networks_supervised.html