Predicting chess games after one move

By Adam Tulling

2021-01-08

Introduction

This paper is the result of the final assignment in the professional certificate in Data Science course of Harvard University. The goal of the assignment was to choose a proper data set which allows it to be analyzed with the techniques taught in the Data science course. Especially focusing on multiple machine learning techniques. Due to my passion for chess I have chosen a chess games data set to be data set of choice for this assignment. The chess game data set is full of chess matches which were played on lichess (https://lichess.org/), a well known and free way to play chess with people you know or people you do not know (yet). The goal of this paper is to predict the result of a match after one move has been played. Because the player playing the white pieces always goes first, we only focus on the white player in regards to the moves played. The final result and effectiveness of the algorithm is determined by the accuracy of predictions on a validation set. The data set will be divided into a train set and a validation set. We will make a train set where the algorithm will be trained on and a test set where the algorithm will be evaluated on. The program used in this paper is Rstudio and was chosen because this was the main program taught in the Data analysis course. Before we start exploratory data analysis and train the machine learning algorithm we will first download the data mentioned.

Loading the data

The data we will use was found on kaggle.com. The file can be found using this URL " https://www.kaggle.com/datasnaek/chess". The amount of data consisted of 7.32 MB stored on one csv file. The data is stored on my github account (https://github.com/AJTulling).

```
# Downloading the data set
data <- read.csv("games.csv")
```

Not only the data set, but also some additional packages need to be installed before exploratory data analyses can take place. These packages are of importance to Rstudio. Rstudio incorporates lots of functions, for example base r functions, but there are lots of packages that can be added on to Rstudio.The packages below we will need for functions throughout this paper.

Loading the packages used for analysis

```
if(!require(tidyverse))
  install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret))
  install.packages("caret", repos = "http://cran.us.r-project.org")
if(!require(data.table))
  install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(dslabs))
  install.packages("dslabs", repos = "http://cran.us.r-project.org")
if(!require(lubridate))
  install.packages("lubridate", repos = "http://cran.us.r-project.org")
```

```
if(!require(foreign))
  install.packages("foreign", repos = "http://cran.us.r-project.org")
if(!require(nnet))
  install.packages("nnet", repos = "http://cran.us.r-project.org")
if(!require(ggplot2))
  install.packages("ggplot2", repos = "http://cran.us.r-project.org")
if(!require(reshape2))
  install.packages("reshape2", repos = "http://cran.us.r-project.org")
if(!require(ggpubr))
  install.packages("ggpubr", repos = "http://cran.us.r-project.org")
if(!require(scales))
  install.packages("scales", repos = "http://cran.us.r-project.org")
if(!require(tinytex))
  install.packages("tinytex", repos = "http://cran.us.r-project.org")
if(!require(randomForest))
  install.packages("randomForest", repos = "http://cran.us.r-project.org")
```

Methods

We use exploratory data analysis to figure out which variables of the data set are most useful in the prediction of the outcome of the chess match. So there are three possible outcomes, 'white', 'black' and 'draw'. After the initial data exploration multiple algorithms will be trained to predict the outcome of a chess match. In this paper we will use the following machine learning algorithms: multinomial regression, linear discriminant analysis, k-nearest neighbour and random forest. In the section 'Machine learning" we will further explain why we chose these machine learning techniques.

Exploratory data analysis

The data we downloaded has 20,058 number of observations and 16 variables. For further analysis we will explore all different variables and visualize which variables are useful for predictive analyses.

The data has 16 variables. Here below you can see which class every variable belongs to.

Variable	Class
id	factor
rated	factor
created_at	numeric
last_move_at	numeric
turns	integer
victory_status	factor
winner	factor
increment_code	factor
white_id	factor
white_rating	integer
black_id	factor
black_rating	integer
moves	factor
opening_eco	factor
opening_name	factor
opening_ply	integer

In the next section of the paper we will explain what all of the variables are and perform exploratory data analysis on them.

Winner

The "winner" variable is the one we want to predict. It is a factor with 3 possible values. "white", "black" and "draw". In the table below we depict the amount of times and proportion of times one of each occurs.

```
# Table depicting the amount of times and proportion of times an outcome occured
data %>% group_by(winner) %>% summarise(count = n()) %>%
  mutate(proportion = count/sum(count)) %>% ungroup() %>% knitr::kable()
```

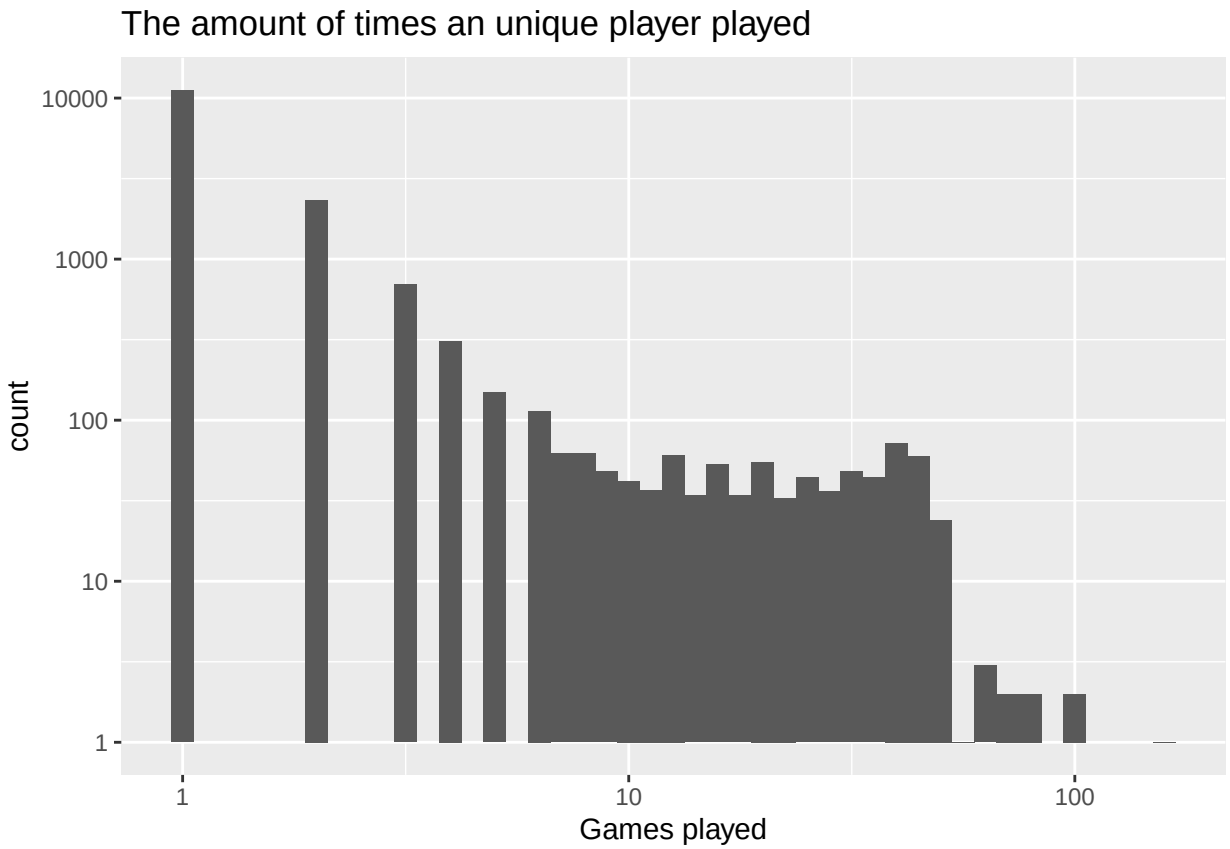
winner	count	proportion
black	9107	0.4540333
draw	950	0.0473626
white	10001	0.4986040

Like we might predict. "black" and "white" are the most frequently occurring values. In less than 5% of cases the match ended in a draw. As you can see, white more often wins than black does. In this paper we will examine the correlations between the different variables and the variable 'winner' that we want to predict. This gives a better view on which features are most effective for our prediction.

White id and black id

There are 15,635 unique players in the data set. Each player has at least played one chess game. Many players only played the black or the white pieces but not both. Some players have played only one match as black and some players have played multiple matches as black and white. The graph below visualizes the frequency of players playing one match, players playing two matches and so on.

```
# Creating 2 data frames containing all players irrespective of frequency occured
All_unique_white_players <- data.frame(all_players = data$white_id)
All_unique_black_players <- data.frame(all_players = data$black_id)

# Visualizing the amount of times an unique player played a chess match
rbind.data.frame(All_unique_white_players,All_unique_black_players) %>%
  group_by(all_players) %>% summarise(count = n()) %>% ggplot(aes(count)) +
  geom_histogram(binwidth = 0.05) +
  scale_x_log10() + scale_y_log10() + xlab("Games played") +
  ggtitle("The amount of times an unique player played")
```

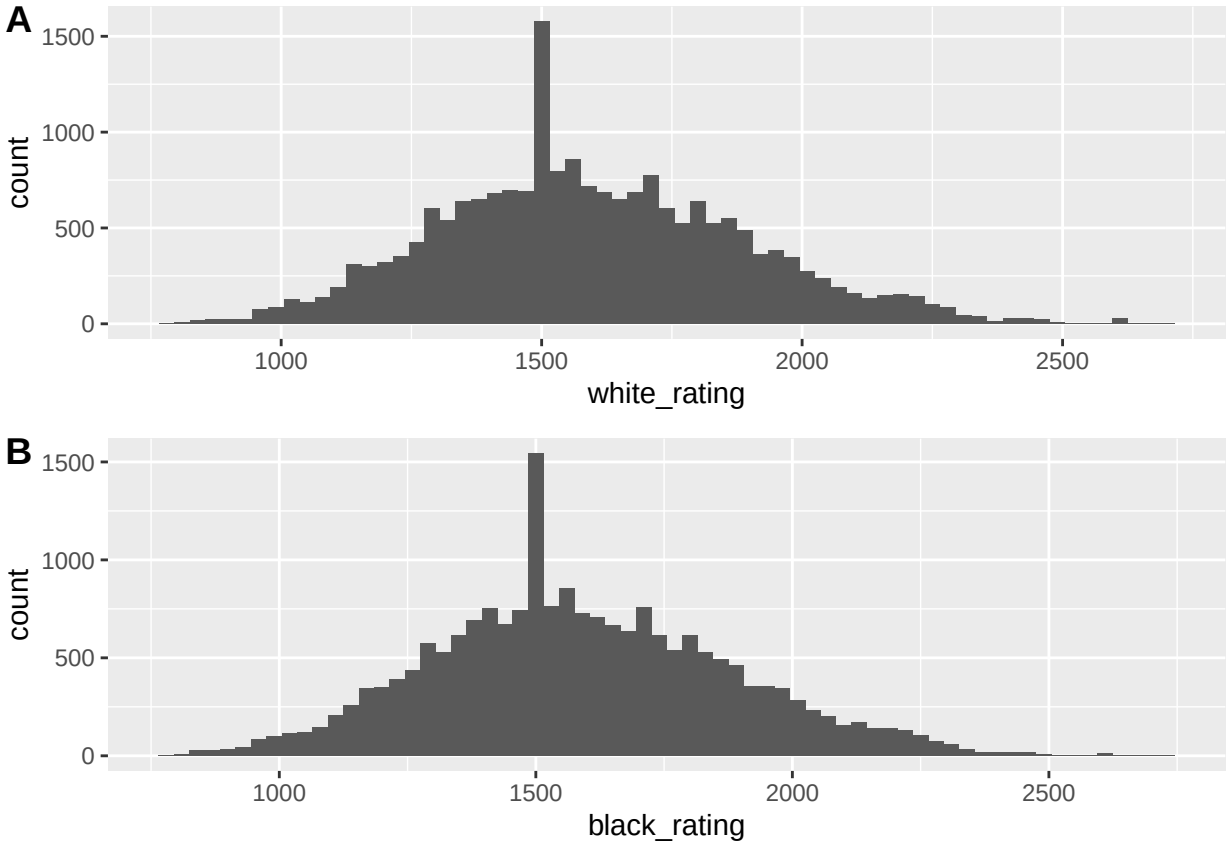


You can see that often a player only played one game.

White and black ratings

These are the most important features available for predicting the winner. The Elo rating is a mathematical method to rank different players. Theoretically this rating could be applied to all sports where one player competes with another individual. In chess this method is the main method of ranking one player. When making a new account in lichess you start with an Elo rating of 1500. When beating other people your rating increases, when losing the opposite happens. The Elo method also takes into account the ranking of your opponent . When losing to a much higher rated player your Elo rating decreases less than losing to a player with the same Elo rating as yours. Below we show how often an Elo rating occurs.

```
# Showing the frequency per Elo rating
graph1 <- data %>%  ggplot(aes(white_rating)) +
  geom_histogram(binwidth = 30)

# Showing the frequency per Elo rating
graph2 <- data %>%  ggplot(aes(black_rating)) +
  geom_histogram(binwidth = 30)

# Plotting both graphs
ggarrange(graph1, graph2,labels = c("A", "B"),ncol = 1, nrow = 2)
```



The ratings between black and white players look very similar. You have a 50/50 percent chance to play with the white or black pieces. Furthermore we see that the ratings are not normally distributed. The distribution is skewed. Moreover, there is a clear spike at the 1500 Elo rating. This is because the first Elo rating you receive is 1500. When playing enough matches this rating decreases or increases.

In the graph below we visualize all the matches and how this correlates with the different Elo ratings of black and white.

```
data %>% ggplot(aes(white_rating,black_rating, color = winner)) +
  geom_point()
```

Between all the noise there is an apparent correlation. A higher Elo rating then your opponent is significantly correlated with a higher chance of winning.

Rated matches and non-rated matches

Some players chose to play a match without the match affecting their Elo rating. These games are regarded as non-rated games. In the data set the variable "rated" depicts this. If the game is rated the data set gives a TRUE, if not it shows a FALSE. We will explore if there are significant differences between rated and non-rated games. As we look at the 'rated' column we see it is a column of the class factor with 4 levels, "TRUE", "True", "FALSE" and "False". Before we perform exploratory data analyses on this column we need to make sure that this column transforms into a logical class. Thereafter we will look at different variables for rated and non-rated games.

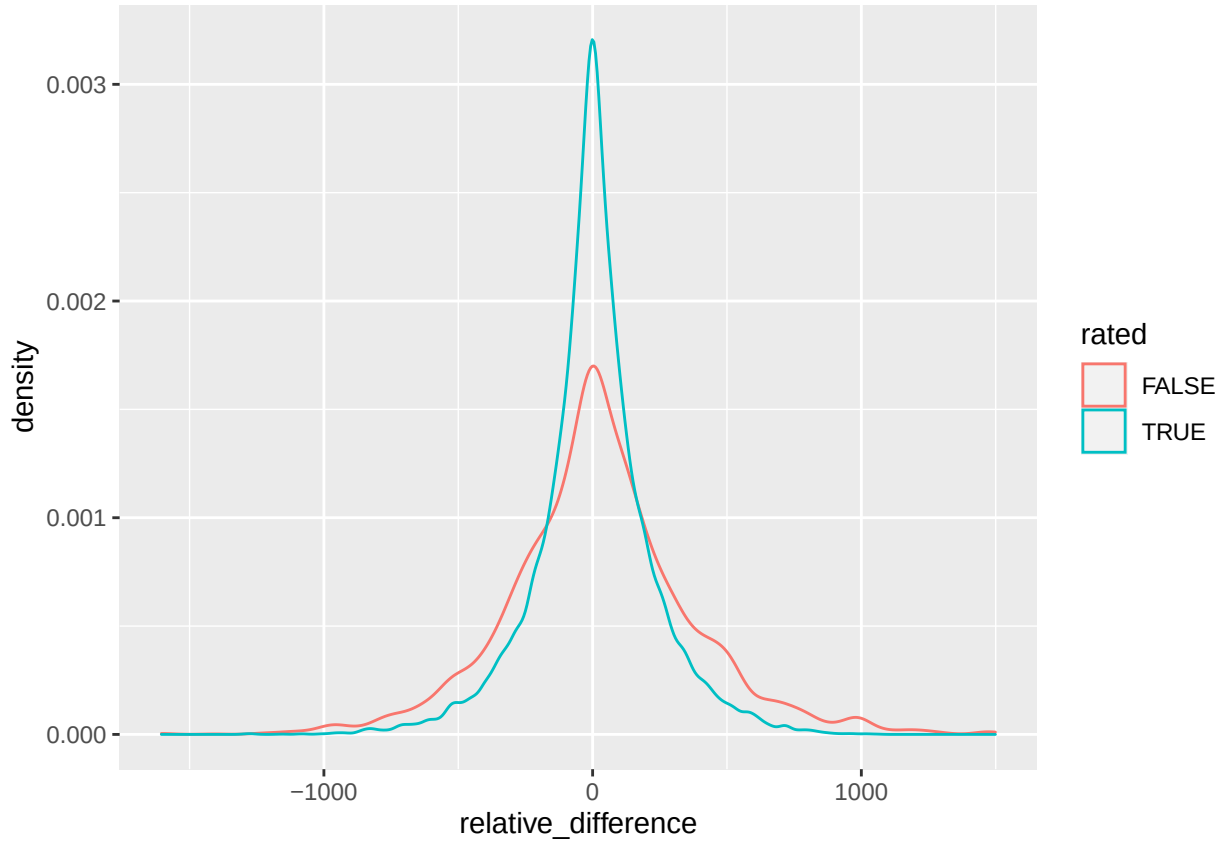
```
# Transforming the factor variable into a logical variable
rated_logical <- with(data, ifelse(rated=="TRUE"|rated=="True",TRUE,FALSE))
data_rated <- data %>% select(-rated) %>% mutate(rated = rated_logical)

# Comparing the winner variable and the relative difference of Elo ratings
# between the two participants
(rated_group <- data_rated %>% group_by(rated)%>%
    summarise(count = n(),
              white = mean(winner=="white"),
              black = mean(winner=="black"),
              draw = mean(winner=="draw"),
              relative_difference =
              mean(white_rating-black_rating))) %>% knitr::kable()
```

rated	count	white	black	draw	relative_difference
FALSE	3903	0.4993595	0.4414553	0.0591852	25.188573
TRUE	16155	0.4984215	0.4570721	0.0445063	3.598824

It seems there is little difference in the percentage of wins for the white or black players. There is however a significant difference in the amount of times a game ended in a draw and the average rating between black and white. Around 5.9% of times the game ended in a draw in the non-rated group, whereas this number was around 4.4% in the rated group. The relative difference is the difference of the rating of white to the rating of black. When white has a higher Elo rating than black this number is positive and vice versa. In the table you can see the relative difference between Elo ratings is positive, so white on average has a higher rating than black. In the non-rated group this difference is even greater. 25 point difference versus a 4 point difference in the rated group. Furthermore 80.5% of games were rated and 19.5% of games were non-rated. This variable, 'rated', could be useful in further analyses.

We will visualize how the relative difference between the two Elo ratings is associated with the "rated" variable.

```
# Graph visualizing the association with relative difference
# and whether or not the game is a rated game.
data_rated %>% mutate(relative_difference = white_rating-black_rating) %>%
  group_by(rated) %>% ggplot(aes(relative_difference, color = rated)) +
  geom_density()
```



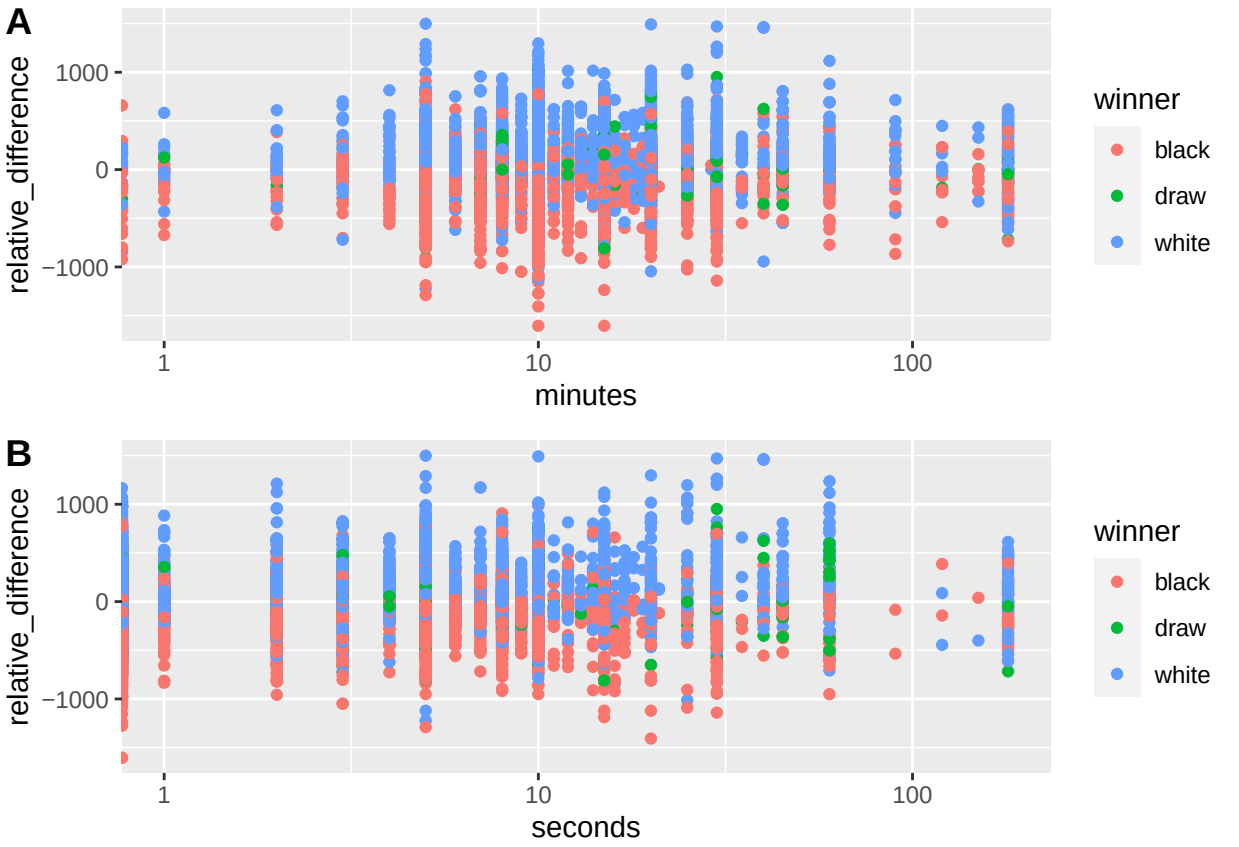
In the graph there is a apparent correlation between the relative difference of Elo ratings and whether or not the game is a rated game. If the relative difference is close to zero, meaning the Elo rating of white is close to the rating of black, the game more often is rated. If these ratings are further apart this is less the case.

Increment_code

There are theoretically 400 different possible levels for the variable 'increment_code'. Firstly we need to explain this code. Before starting a match you have the option to select a regular time frame in which the match is played for. The first number is the amount of minutes an individual player has and the second number is the amount of seconds a player gets after playing a move. This second number often is important late in the game. If playing moves very quickly you get rewarded by getting precious seconds. Between these two numbers there is a "+" character. The maximum of the first and second number is 180. There are 400 unique combinations of this variable in the data set.

Before we start we want to separate the two numbers into 2 columns. 1 column describing the minutes an individual player has to play a move and 1 column depicting the amount of seconds the player gets added up after making a move.

```
# Separating the increment code variable into a minutes and seconds column
data_sep <- separate(data,increment_code,into = c("minutes","seconds"),
                     sep = "\\+",convert = TRUE)

# Adding the most important confounder, the relative difference in player ratings
data_sep <- data_sep %>% mutate(relative_difference = white_rating-black_rating)

# Visualizing the association between the winner variable
# and the minutes variable and including the relative difference between Elo ratings
graph1 <- data_sep %>% ggplot(aes(minutes,relative_difference, color = winner)) +
  geom_point() + scale_x_log10()

# Visualizing the association between the winner variable and the seconds variable
# and including the relative difference between Elo ratings
graph2 <- data_sep %>% ggplot(aes(seconds,relative_difference, color = winner)) +
  geom_point() + scale_x_log10()

# Visualizing both graphs
(figure <- ggarrange(graph1, graph2,
                     labels = c("A", "B"),
                     ncol = 1, nrow = 2))
```



We see there is no apparent correlation between the time variables and the outcome of a match. It is not likely that these two variables will be useful for our final algorithm.

Moves

The variable moves shows all the moves that were played during the match. For us only the first move is of importance. White is always the first that makes a move. There are only 20 moves possible. All of these moves were played at least ones. In the table below we show the moves that most often cause the white player to win or to lose. Adding the amount of times the move occurred in the data set.

```
# Separating the moves variable into the first move and all other moves and
# removing the latter one
moves_data <- data %>% separate(moves,into = c("opening_white","the_other_moves"),
                                sep = " ") %>% select(-the_other_moves)

# Adding the relative_difference variable
moves_data <- moves_data %>% mutate(relative_difference = white_rating-black_rating)

# Generating a table with the first moves of white that results in the highest chance of
# winning for white, adding the frequency the move occurs
moves_data %>% group_by(opening_white)%>%
  summarise(count = n(),white_wins = mean(winner=="white"))%>%
  arrange(desc(white_wins)) %>% head() %>% knitr::kable()
```

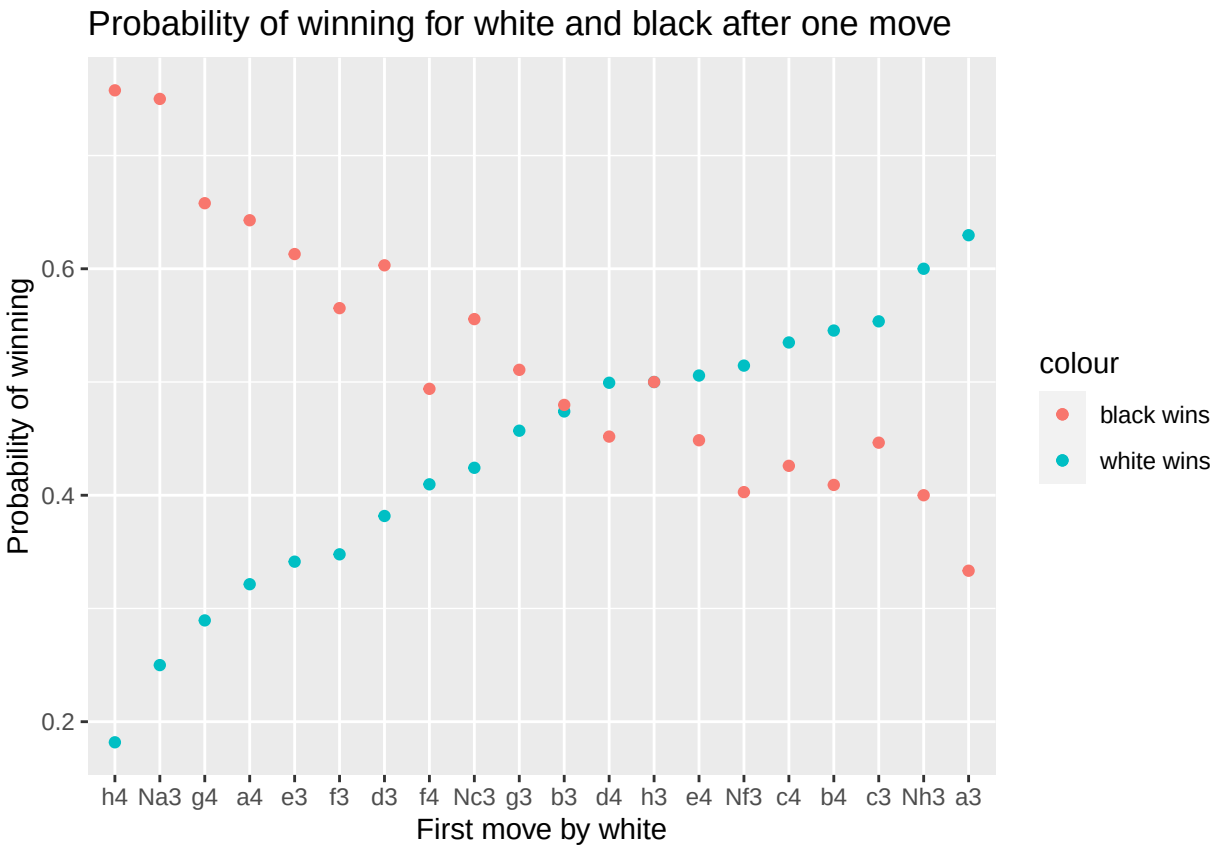
opening_white	count	white_wins
a3	27	0.6296296
Nh3	15	0.6000000
c3	56	0.5535714
b4	88	0.5454545
c4	716	0.5349162
Nf3	725	0.5144828

```
# Generating a table with the first moves of white that results in
# the lowest chance of winning for white, adding the frequency the move occurs
moves_data %>% group_by(opening_white)%>%
  summarise(count = n(),white_wins = mean(winner=="white"))%>%
  arrange(white_wins) %>% head() %>% knitr::kable()
```

opening_white	count	white_wins
h4	33	0.1818182
Na3	4	0.2500000
g4	38	0.2894737
a4	28	0.3214286
e3	416	0.3413462
f3	23	0.3478261

Below we will make a graph showing different kinds of first moves and the probability of white or black winning the match.

```
# The probabilities white or black wins for every different first move
table1 <- moves_data %>% group_by(opening_white)%>%
  summarise(count = n(), white_wins = mean(winner=="white"),
            black_wins = mean(winner=="black")) %>%  arrange(white_wins)

# Plotting these differences and arranging them from the lowest to
# the highest probability for white to win
table1 %>% ggplot(aes(x = reorder(opening_white,white_wins))) +
                        geom_point(aes(y = white_wins, color = "white wins")) +
                       geom_point(aes(y = black_wins, color = "black wins")) +
  ggtitle("Probability of winning for white and black after one move") +
  xlab("First move by white") + ylab("Probability of winning")
```



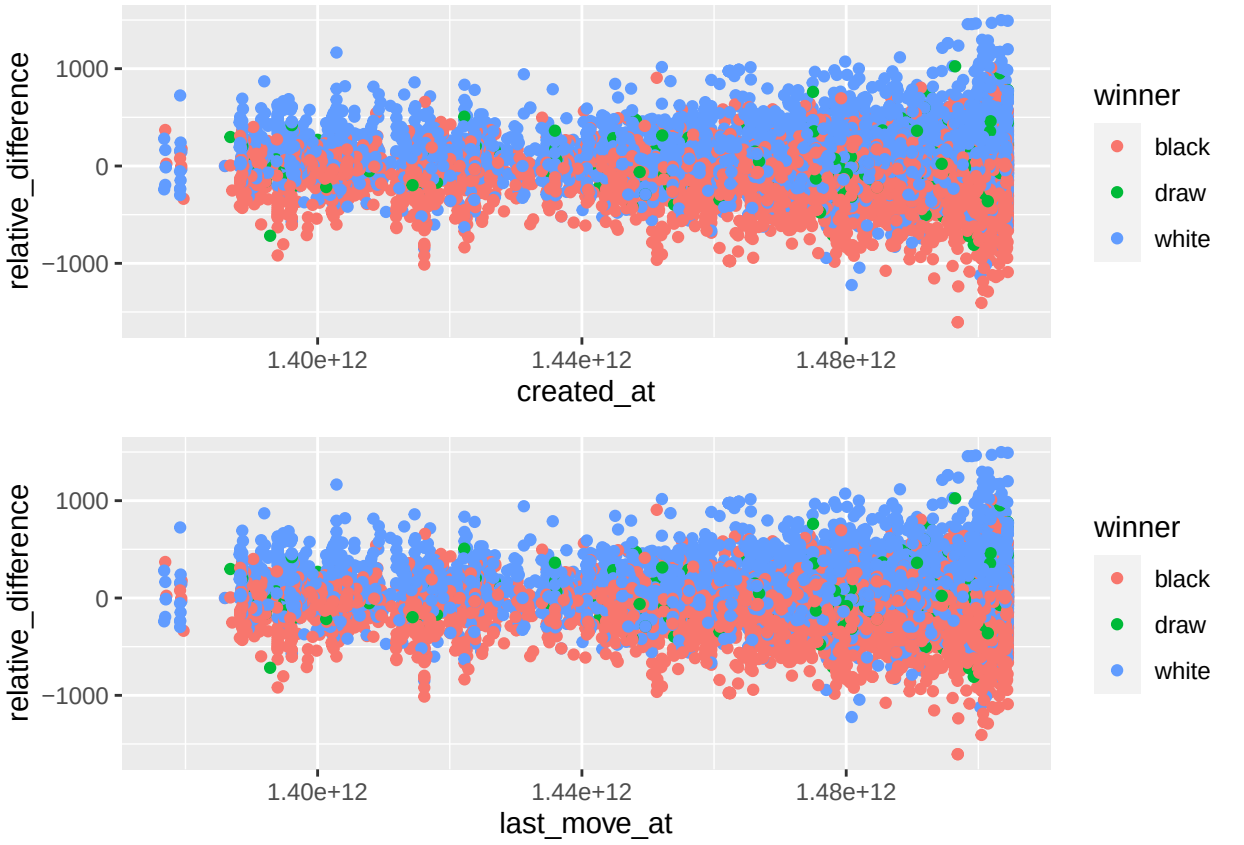
The graph shows there is a apparent correlation between the first move played and the outcome of the match. However a footnote needs to be placed. The graph does not show differences in Elo ratings. For example the move 'h4' could be a move primarily played by very bad players with very low Elo ratings. We know for a fact that 'h4' is a bad move and is often played by beginners. On the other extreme of the spectrum, 'a3' seems to be the best move for white. In the chess world in fact the moves 'a3' and 'h3' are regarded as one of the worst moves to play for white. Why do these two moves still have a relatively good success rate? One reason could be that players that are very good and have a high Elo rating want to demonstrate to the opponent that they can win a match even when playing one of the worst moves in the game. The Elo ratings of black and white are the most important con-founders in this graph. Still this variable could have a positive impact on the algorithm we will create in the next sections of this paper.

Last move at and Created at

In theory this variable would be a good way to calculate the duration of the match. 'created_at' tells us something about the moment the match was created and 'last_move_at' tells us on which moment the last move was played. When taking the absolute number of these two values we get the difference in time and so the duration of time the match was played for. Unfortunately the numbers are not exact. The numbers are written in a scientific manner. For example 1.35e14. It is impossible to extract this entire number. So there is no way to calculate the duration of the match. In the graph we show how the variables correlate with the outcome variable, 'winner'.

```
# Association between the created_at and winner variable, corrected for the relative difference
graph_created_at <- moves_data %>%
  ggplot(aes(created_at,relative_difference, color = winner)) + geom_point()

# Association between the last_moved_at and winner variable, corrected for the relative difference
graph_last_move_at <- moves_data %>%
```

```
  ggplot(aes(last_move_at,relative_difference, color = winner)) + geom_point()

 # Plotting both graphs
ggarrange(graph_created_at, graph_last_move_at, ncol = 1, nrow = 2)
```



We see that there is little to no difference between the winner of the match and the time variables. Furthermore when comparing these two graphs there seem to be little to no difference between the two graphs. There are only a couple of dots the move slightly. Only a couple of matches are of such a long duration it is notable on the graph.

Other variables

There are 6 other variables that are less useful for our analyses. One reason would be that it contradicts the goal of our paper. Our paper wants to predict the winner of the match after one move has been played. This would exclude variables as turns, victory_status, opening_eco, opening_name and opening_ply. The 'turns' variable is the number of turns the match is played for. Some matches consist of only 2 moves, others show a higher number of turns. The 'opening_eco', 'opening_name' and 'opening_ply' all stand for the same thing. An opening consists of a number of moves to start the game. There are simple openings like the Sicilian defense that consist of only two moves. The move e4 of white and the move c5 of black. But the amount of turns for an opening can greatly differ. These openings are categorized by a name, for example the Sicilian defense, a eco code, for example 'D01', or an integer as is seen in the 'opening_ply' variable. Furthermore we will not analyse the variable id. This is a variable describing the personal code of every individual match. We will not be working with this code, because it is a randomly generated character code without any extra information on the game itself.

Transforming the data

Before making specific algorithms we want to add and remove data that will not be incorporated in our further analyses. Inside the lines of code we explain which changes we made.

```
# Turning the 'rated' variable into 2 levels, 'TRUE' and 'FALSE'
rated_logical <- with(data, ifelse(rated=="TRUE"|rated=="True",TRUE,FALSE))
new_data <- data %>% select(-rated) %>% mutate(rated = rated_logical)

# Separating the increment code variable into 2 seperate columns
new_data <- separate(new_data,increment_code,into = c("minutes","seconds"),
                     sep = "\\+",convert = TRUE)

# Extracting the relative difference between two Elo scores
new_data <- new_data %>% mutate(relative_difference = white_rating-black_rating)

# Extracting the first move of white
new_data <- new_data %>% separate(moves,into = c("opening_white","other_moves"),
                                  sep = " ",extra = "merge") %>% select(-other_moves)

# Selecting the most important variables for further analyses
new_data <- new_data %>% select(rated, winner, minutes, seconds, white_rating,
                                black_rating, opening_white, relative_difference)
```

'new_data'is the name of the transformed data set our further analyses will rely on.

Constructing a train and test set

Before doing this section we need some information about the topic of data partitioning in a train and test set. Some people use the 80/20 method, others the 70/30 method, with the lower number always being the proportion of data for the test set. In this paper we will use the 80/20 proportions. 80 percent of the data will be used to train an algorithm and 20 percent of data will be used to evaluate this algorithm. A reason for this, is that our machine learning techniques need lots of data to create better predictive algorithms. That is why we will give it a lot of information, 80 percent of the data to be exactly.

```
# Setting a seed for reproducibility
set.seed(007, sample.kind = "Rounding")

# Partitioning the data into a train and test set by the fraction of 80/20
index <- createDataPartition(new_data$winner,times = 1, p= 0.2, list = FALSE)
test_set <- new_data[index,]
train_set <- new_data[-index,]
```

Creating a reference for our algorithms

Calculating the accuracy without any data incorporation

To know whether our prediction is of any good we need to create a reference. What if our prediction is completely random? The computer could chose between the outcome at random.The three levels the computer can choose from is "white", "black" and "draw". We will be simulating this situation. The prediction will be done comparing the test set.

```
# Generating the three outcomes
levels <- c("white","black","draw")

# Guessing the three outcomes
guessing <- sample(levels, size = nrow(test_set), replace = TRUE)
```

```r
# Calculating the accuracy using the test set
(accuracy_by_guessing <- mean(guessing == test_set$winner))
```

```
## [1] 0.3192126
```

As we might have guessed, the accuracy using only guessing is not very high.

Calculating by the most frequent winner

The most frequent winner is 'white', like we determined earlier in the exploratory data analysis section under the variable 'winner'. We will be simulating a situation where we predict 'white' every time.

```r
# Predicting white every time
white_every_time <- rep("white",times = nrow(test_set))

# Calculating the accuracy
(accuracy_white_wins_every_time <- mean(test_set$winner==white_every_time))
```

```
## [1] 0.4986295
```

Now our accuracy increases to almost 50%.

Calculating the accuracy using player ratings

There is another easy method to predict the winner. It involves deciding the winner based on the Elo ratings of both the white and the black player. The player with the highest Elo rating gets predicted. We will simulate this situation.

```r
# Predicting the player with the highest elo rating
highest_elo_rating <- ifelse(test_set$white_rating >= test_set$black_rating, "white" , "black")

# Calculating the accuracy
(accuracy_highest_elo_rating <- mean(highest_elo_rating == test_set$winner))
```

```
## [1] 0.6204834
```

The simple use of the Elo ratings causes a sharp increase in the accuracy of the predictions. A small footnote needs to be placed. Sometimes two players have the same Elo rating. Because white slightly more often wins we choose the white player over the black player. If running the code differently, giving the win to the black player when both ratings are equal results in a small decrease in the accuracy of the predictions.

Machine learning methods

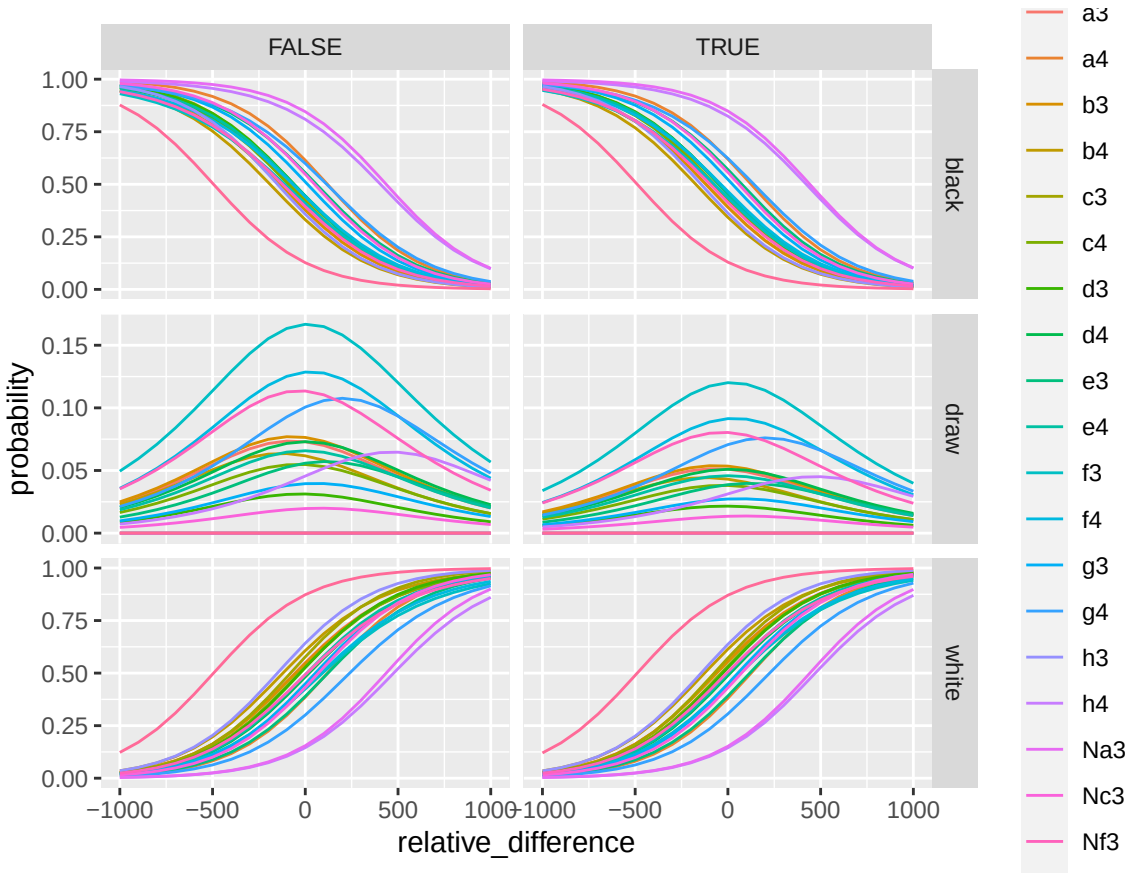
Calculating the accuracy by multinomial regression

One of the more easier methods of machine learning is using linear regression and logistic regression. Unfortunately these techniques often rely on a continuous or logical outcome. Because there are not 2 but 3 outcomes: 'white', 'black' and 'draw', we will use multinomial regression instead of these other techniques. The most important variables are the ratings of white and black and the relative difference between the two. As secondary features we will implement the opening move of white and whether or not the game was rated or not.

```r
# Generating an algorithm using multinomial regression
multinom_reg <- multinom(winner ~ relative_difference + opening_white + rated, data = train_set)
```

```
## # weights:  69 (44 variable)
## initial  value 17627.234172
## iter  10 value 13500.837188
## iter  20 value 12578.368914
## iter  30 value 12390.537498
## iter  40 value 12386.729549
## iter  50 value 12386.369241
## iter  60 value 12386.347130
## final  value 12386.346626
## converged
```

```r
# Creating a data frame for visualization purposes
all_variables <-
  data.frame(opening_white = rep(as.factor(unique(new_data$opening_white)), each = 42 )
             ,relative_difference = rep(seq(-1000,1000,100),40),
             rated = as.logical(rep(c("TRUE","FALSE"),420)))

# Predicting the probablities of white, black or draw being the outcome
# for the variables in the data frame 'all_variables'
all_scores_predict <- cbind(all_variables, predict(multinom_reg, newdata = all_variables,
                                                   type = "probs", se=TRUE))

# Visualizing the probability for a particular outcome
melting <- melt(all_scores_predict, id.vars = c("opening_white","relative_difference", "rated"),
                value.name = "probability")
ggplot(melting, aes(x = relative_difference, y = probability, color = opening_white)) +
  geom_line() + facet_grid(variable~rated, scales = "free")
```



In the graph above you can see the regression lines for different opening moves. It seems to be that the outcome 'draw' never has a higher probability than 'black' or 'white' does. Therefor this outcome will not be predicted by the multinomial regression algorithm. Furthermore, it seems to be that the rated games do not seem to be significantly different from the non-rated games. You see this when comparing the right three to the left three graphs. The plots above were made for visualization purposes. Below this text we make a prediction based on the algorithm we just made.

```
# Predicting using the multinomial regression
multinom_reg <- predict(multinom_reg,test_set,"class")

# Determining the accuracy
(accuracy_multinom <- mean(multinom_reg==test_set$winner))
```

```
## [1] 0.626464
```

The prediction algorithm using mulitnomial regression is only minimally better than simply choosing the outcome based on the highest Elo rating.

Linear discriminant analysis

Linear discriminant analysis is another machine learning algorithm which often works very well with all kinds of variables and often needs less computational time. This is why we chose to incorporate this technique into the paper. We tried to implement different kinds of variables and chose to use the different Elo ratings, the opening move, the 'rated' variable and the minutes and seconds variables. We made this selection because these seemed to be the most predictive for the outcome. This was established in the exploratory data analysis section.

```r
# Training with lda
train_lda <- train(winner~white_rating+black_rating+opening_white+rated+seconds+minutes,
                   data = train_set, method = "lda")

# Predicting the outcome with the lda algorithm
y_hat_lda <- predict(train_lda, test_set, method = "class")

# The accuracy of the algorithm
(accuracy_lda <- mean(y_hat_lda==test_set$winner))
```

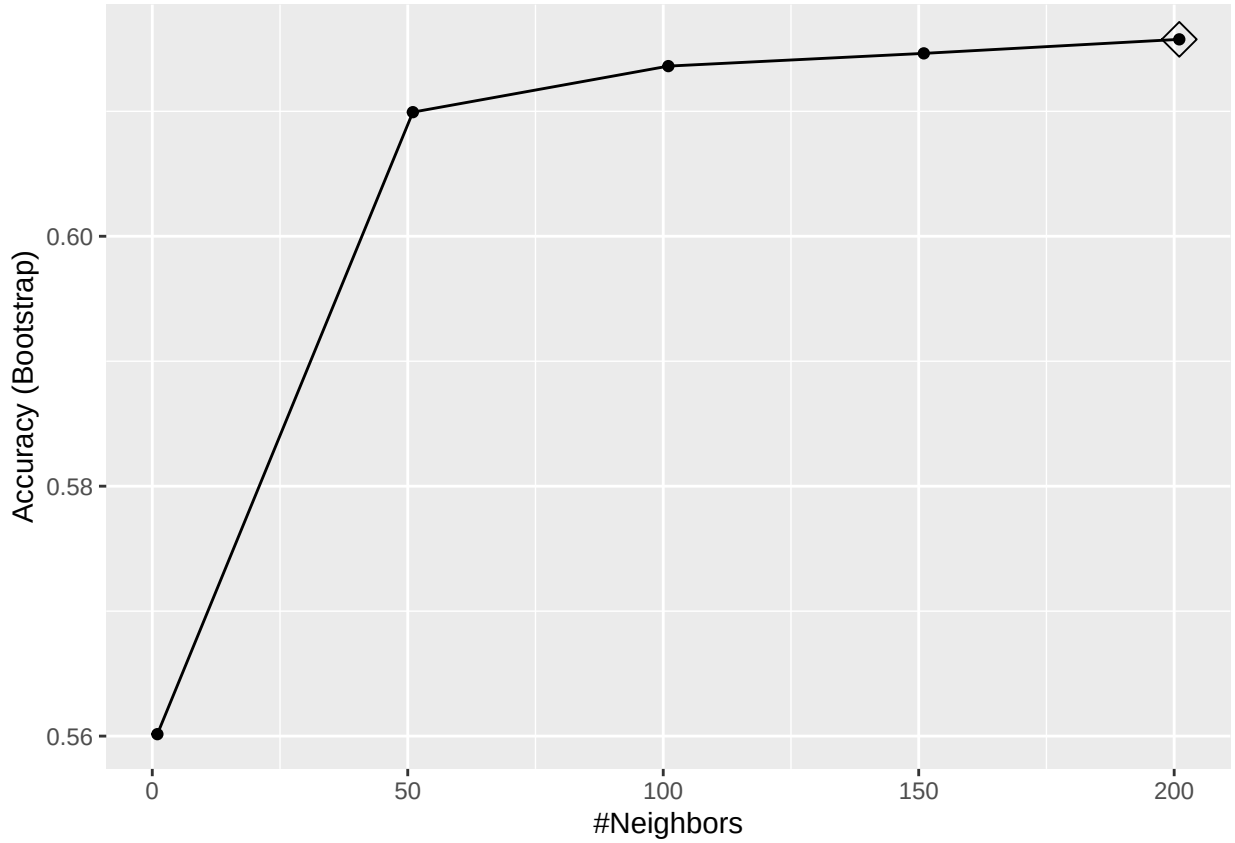
```
## [1] 0.6207326
```

Linear discriminant analysis shows almost the same accuracy rating as simply choosing the outcome based on the highest Elo rating.

K-Nearest neighbor

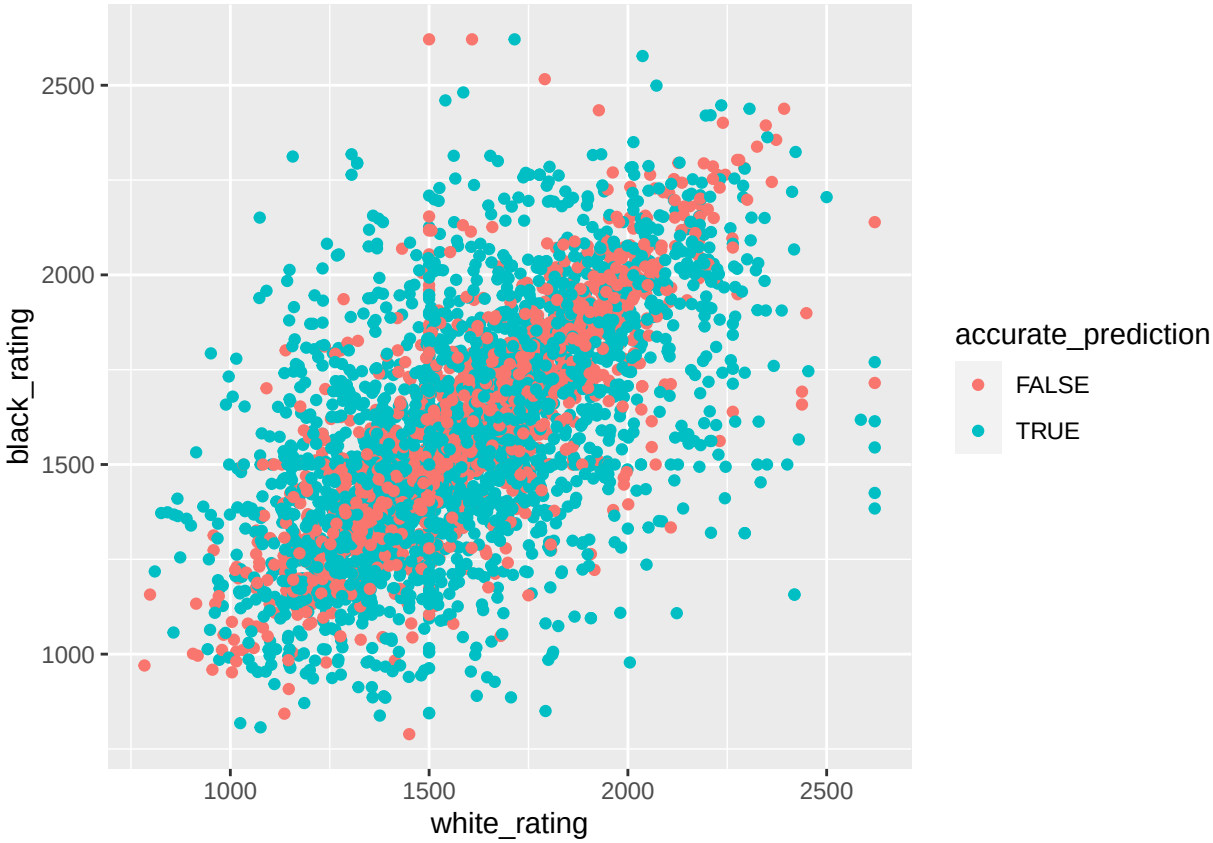
K-nearest neighbour is an algorithm which often shows great predictive power. It is one of the machine learning techniques explained during the course. Because adding more variables can dramatically increase the computational time we will only use variables we know have an effect on the outcome. We choose the different Elo ratings and the first move played by white.

```r
# Selecting the parameter k
ks <- data.frame(k = seq(1,201,50))

# Training with knn
train_knn <- train(winner~white_rating+black_rating, data = train_set,
                   method = "knn",tuneGrid = ks)

# Plotting the best accuracy for different k's
ggplot(train_knn, highlight = TRUE)
```



The graph shows the best value for the parameter k.

```
# Predicting the winner of the test set using the knn algorithm
y_hat_knn <- predict(train_knn, test_set, method = "class")

# Creating a data frame with our predictions
Prediction <- data.frame(y_hat_knn = y_hat_knn,
                         winner = test_set$winner,
                         black_rating = test_set$black_rating,
                         white_rating = test_set$white_rating,
                         accurate_prediction = test_set$winner==y_hat_knn)

# Visualizing the accuratly and wrongly predicted outcome
Prediction %>% ggplot(aes(white_rating, black_rating,color = accurate_prediction)) +
  geom_point()
```



The plot visualizes the wrongly and correctly predicted matches. In the piece of code below the accuracy will be determined.

```
# The accuracy of our knn algorithm
(accuracy_knn <- mean(y_hat_knn==test_set$winner))
```

```
## [1] 0.6154996
```

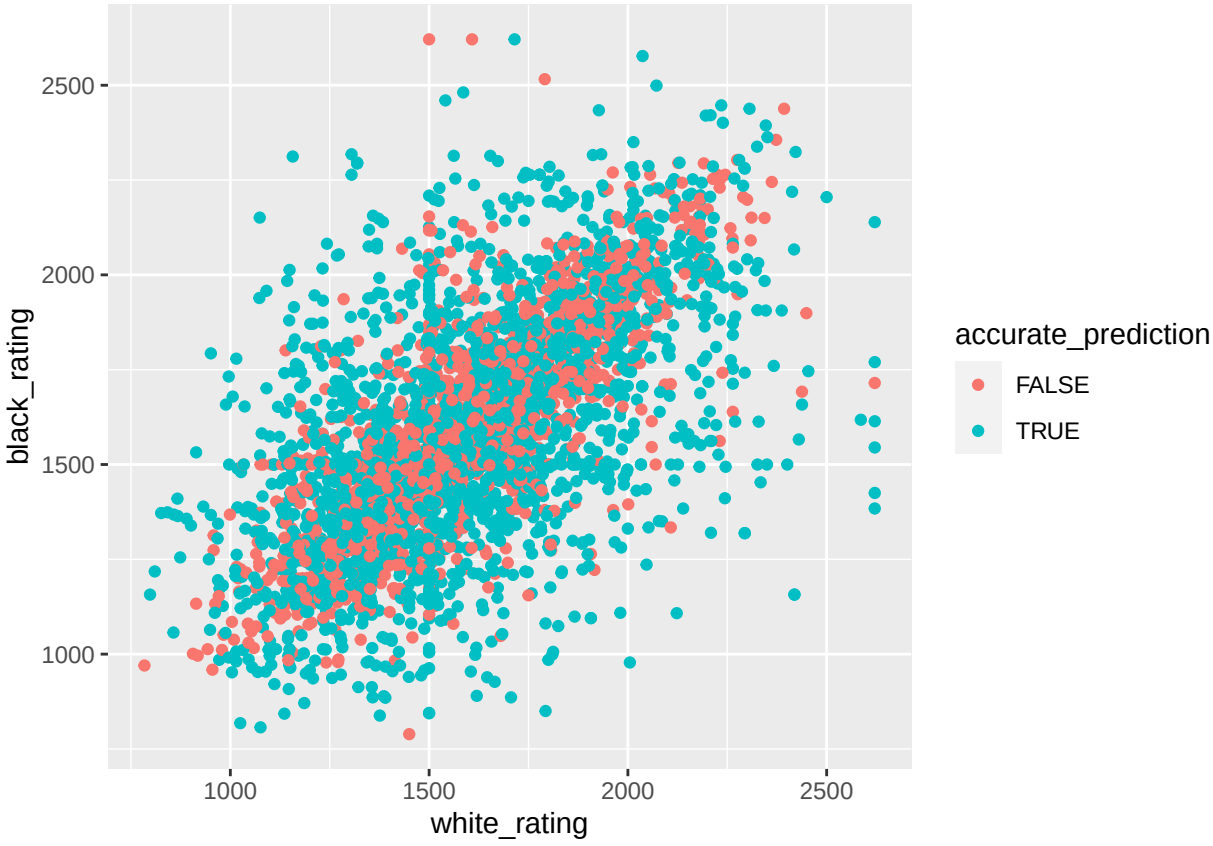
We see that the k-nearest neighbour algorithm we created does not perform better than our standard of choosing the player with the highest Elo rating.

Random forest

Random forest is an algorithm which often shows great predictive power. It is one of the machine learning techniques explained during the course. Adding more variables can dramatically increase the computational time, more so than in the k-nearest neighbour technique. We will only use variables we know that have an effect on the outcome. We will only use the Elo ratings of white and black players to train the algorithm. Adding another variable would be senseless, because this would take too much time to the train the algorithm. However, we will be tuning the algorithm with two tuning parameters, 'mtry' and 'ntree'.

```
# Creating a random forest which incorporates two parameters
twoParameterRF <- list(type = "Classification",
                       library = "randomForest",
                       loop = NULL)

# Creating a data frame containing the two parameters
```

```r
twoParameterRF$parameters <- data.frame(parameter = c("mtry", "ntree"),
                                        class = rep("numeric", 2),
                                        label = c("mtry", "ntree"))

twoParameterRF$grid <- function(x, y, len = NULL, search = "grid") {}

twoParameterRF$fit <- function(x, y, wts, param, lev, last, weights, classProbs) {
  randomForest(x, y,
               mtry = param$mtry,
               ntree=param$ntree)
}

# Predict label
twoParameterRF$predict <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
   predict(modelFit, newdata)

# Predict probability
twoParameterRF$prob <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
   predict(modelFit, newdata, type = "prob")

twoParameterRF$sort <- function(x) x[order(x[,1]),]
twoParameterRF$levels <- function(x) x$classes


# mtry between 1 and 2 since there are only 2 predictor variables
parameters_rf <- expand.grid(.mtry=c(1:2),.ntree=c(1, 100, 1000, 2000, 5000))

# Selecting the cross validation
control <- trainControl(method='repeatedcv', number=2, repeats=1)

# Training with random forest
train_rf <- train(winner~white_rating+black_rating, data = train_set, method = twoParameterRF,
                  tuneGrid = parameters_rf, trControl = control)

# Predicting the winner of the test set using out random forest algorithm
y_hat_rf <- predict(train_rf, test_set, method = "class")

# Creating a data frame with our prediction
Prediction <- data.frame(y_hat_rf = y_hat_rf, winner = test_set$winner,
                         black_rating = test_set$black_rating,
                         white_rating = test_set$white_rating,
                         accurate_prediction = test_set$winner==y_hat_rf)

# Visualizing the accurately and wrongly predicted outcome
Prediction %>% ggplot(aes(white_rating, black_rating,color = accurate_prediction)) + geom_point()
```



The plot visualizes the wrongly and correctly predicted games. Below this piece of code the accuracy will be predicted.

```
# The accuracy of our random forest algorithm
(accuracy_rf <- mean(y_hat_rf==test_set$winner))
```

```
## [1] 0.6137553
```

We see that the random forest algorithm we created, like our earlier k-nearest neighbour method does not perform better than our standard of choosing the player with the highest Elo rating.

Results

Results table

```
# Creating a table with all the results
data.frame(Method = c("Guessing","Predicting white every time",
                      "Predicting the highest Elo rating",
                       "Multinomial regression","Linear disciminant analysis",
                      "K-nearest neighbour","Random Forest" ),
   Accuracy = c(round(accuracy_by_guessing,digits = 4),
                round(accuracy_white_wins_every_time,4),
                round(accuracy_highest_elo_rating,4),
                round(accuracy_multinom,4),
                round(accuracy_lda,4),
                round(accuracy_knn,4),
```

```
                round(accuracy_rf,4))) %>%
  arrange(Accuracy)%>% knitr::kable()
```

Method	Accuracy
Guessing	0.3192
Predicting white every time	0.4986
Random Forest	0.6138
K-nearest neighbour	0.6155
Predicting the highest Elo rating	0.6205
Linear disciminant analysis	0.6207
Multinomial regression	0.6265

As we can see in the table above the results can be easily increased by predicting the player with the highest Elo rating as the winner of the match. The method of multinomial regression was the only one that minimally out performed this method. Other slightly more advanced algorithms, linear disciminant analysis, k-nearest neighbour and random forest, only worsened the accuracy.

Conclusion

This paper examined the possibilities of predicting chess matches when only the first move is known. Easier said, is it possible to predict chess matches and to what extent. After analyzing the data and using different kinds of algorithms based on several machine learning techniques the results were mediocre. Creating more advanced machine learning algorithms did not result in better predictions, sometimes even worsening the predictions. It seems that choosing the player with the highest Elo rating as winner results in one of the best predictive methods. Although multinomial regression was slightly more accurate in the predictions. This might provide enough proof that chess matches are not as easy to predict.

Discussion

There were some limitations to this study. A large footnote needs to be placed. In partitioning the data into a train and test set the computer uses randomness. It chooses which data gets into the train set and which data gets into to test set. Machine learning techniques also use randomness to choose for example different parameters. This is a lot of randomness which makes the results particular prone to fluctuations. For reproducibility we used the 'set.seed()' function. When changing the number in this function we can see the results in the table changing. Because a lot of these results are close to each other there is a possibility that the algorithm with the best accuracy can differ. Also an important limitation we encountered was the computational power. The computer we used decreased the amount of variables that could be incorporated into training the algorithm. For further research into this topic I would recommend using a bigger database. For example the one that is free to download from the lichess website. There are downloads with more than a million games, with more variables then the data set used in this paper. Furthermore I would advise to use matrix factorization as a good option for further research.