King and Rook vs lone King Endgames

Marcos A. Gonzalez

12/10/2020

Introduction

In this project I analyze the starting position of a very specific endgame in chess: White King + White Rook versus lone Black King. Depending on the initial position of the pieces the possible results of the game are as follows:

- White wins after zero moves because Black is already in checkmate
- White wins after one or more moves with optimal play
- Black draws (ties) after one or more moves with optimal play

The question is, can we predict the outcome of the game given the initial positions of the pieces as predictors?

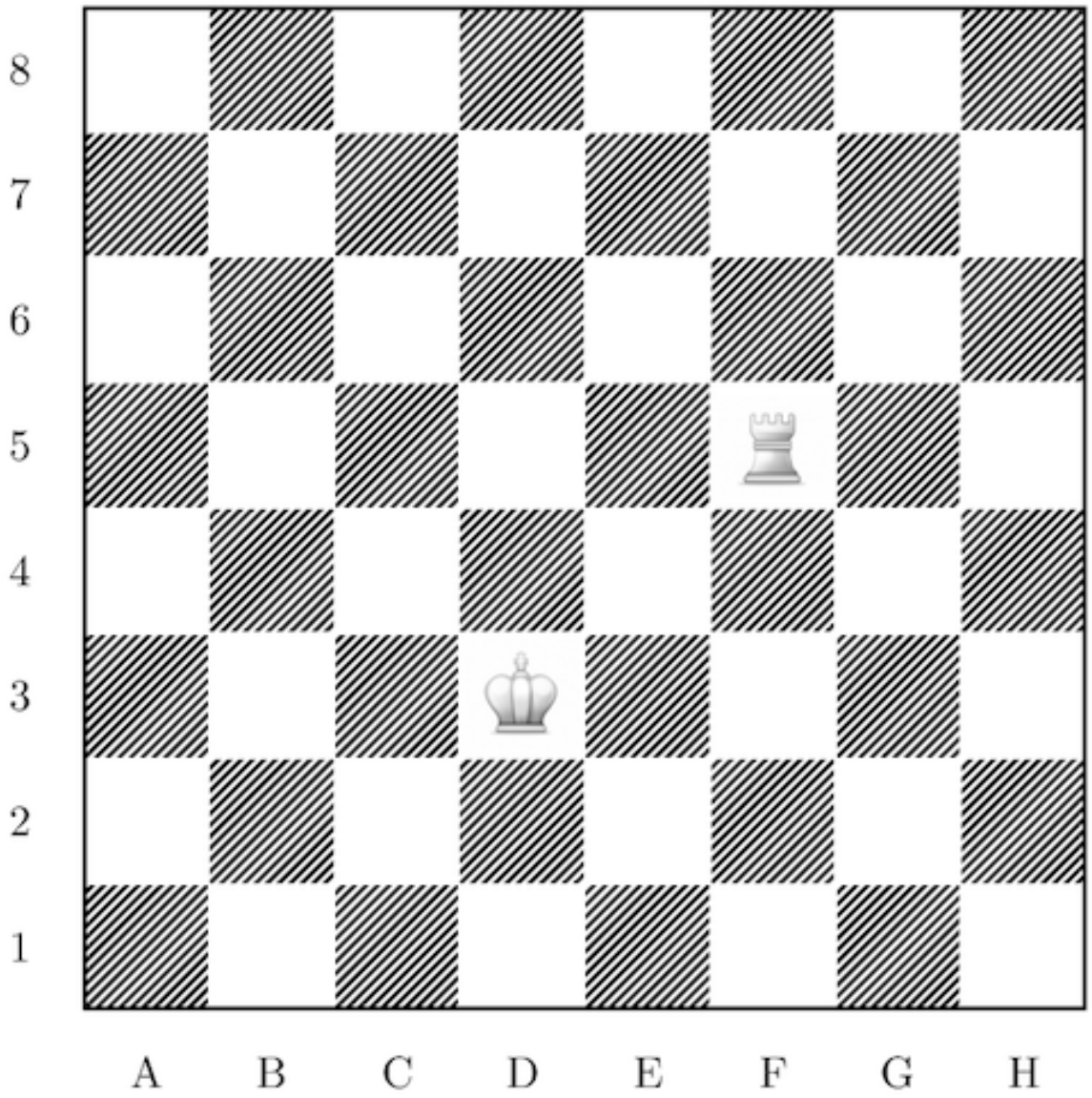
I explore the possibility to predict the outcome of the game using the initial position of the pieces as predictors. Note how I don't attempt to determine the series of moves needed to win or draw but only predict the outcome (either draw or the number of moves needed to win) based on the initial position of the pieces.

Analysis

The Dataset

This dataset is available at (https://archive.ics.uci.edu/ml/datasets/Chess+%28King-Rook+vs.+King%29). It includes the initial positions of each piece in this particular endgame using algebraic notation.

Algebraic notation: In chess the board is divided in 'files' (columns from a to h) and 'ranks' (rows from 1 to 8). The intersection of a file and a rank uniquely identifies each of the 64 squares. See the following figure (white king is on *d3* and white rook is on *f5*):



Partitioning the Data

The following block of code loads the dataset, assigns column names to the set, and converts the *result* column to a Factor. Also, we show its first few rows:

```
king_rook <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/chess/king-rook-vs-king
colnames(king_rook) <- c('w_king_file','w_king_rank','w_rook_file','w_rook_rank',
                         'b_king_file','b_king_rank','result')
king_rook$result <- as.factor(king_rook$result)
head(king_rook)
```

```
##   w_king_file w_king_rank w_rook_file w_rook_rank b_king_file b_king_rank
## 1           a           1           b           3           c           2
## 2           a           1           c           1           c           2
```

```
## 3          a           1          c           1          d           1
## 4          a           1          c           1          d           2
## 5          a           1          c           2          c           1
## 6          a           1          c           2          c           3
##   result
## 1   draw
## 2   draw
## 3   draw
## 4   draw
## 5   draw
## 6   draw
```

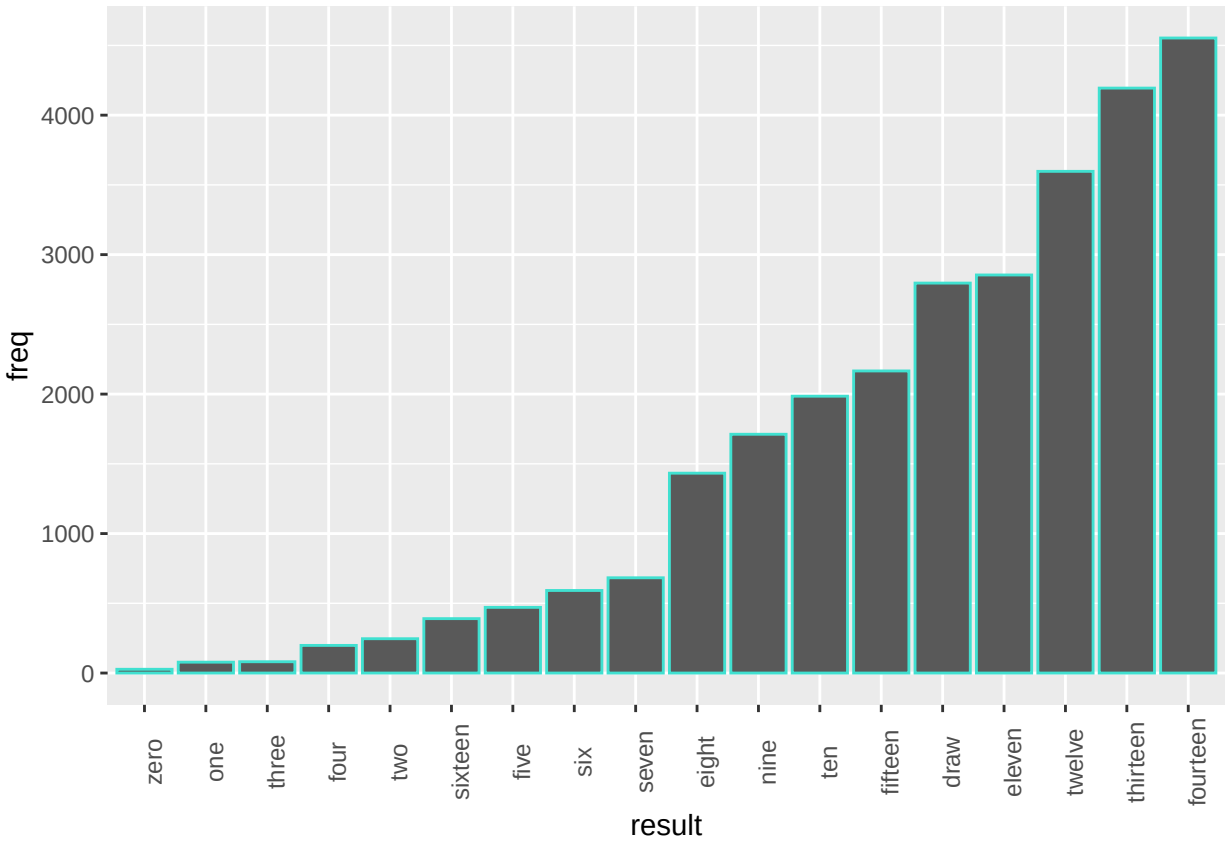
The first two columns in the dataset represent the square where the white king is standing (file and rank respectively), the following two columns represent white rook's position, and the following two are the black king's position. The last column is the outcome of the game from this position obtained using chess playing engines, table bases, and other methods.

All observations in the dataset assume it's black to move. For example, the first observation of the dataset represents the white king on 'a1', a white rook on 'b3', and the black king on 'c2'. The game's observed outcome from this position is 'draw'.

Other outcomes include the number of moves needed (0, 1,..,16) in order to win the game with optimal play. The distribution of outcomes is shown in the following plot:

```
king_rook %>%
  group_by(result) %>%
  summarize(freq=n()) %>%
  mutate(result = reorder(result, freq)) %>%
  ggplot(aes(result, freq))+
  geom_bar(stat = "identity", color="turquoise") +
  theme(axis.text.x = element_text(angle = 90))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```



Method

This is a classification problem, since we are trying to determine one of many outcomes based on the predictors. I'm going to use knn and random forests methods.

We first partition the data in training and test datasets. Training set is 80% of the total dataset. In turn, I partition the training set in 80% for actual training and 20% for cross validation.

```
test_index <- createDataPartition(king_rook$result, times=1, p=0.8, list=FALSE)
testing  <- king_rook[-test_index,]
whats_left <- king_rook[test_index,]
```

This has created the **testing** set with 20% of the total data. Next we assign 80% of the remaining data to the **training** set:

```
train_index <- createDataPartition(whats_left$result, times=1, p=0.8, list=FALSE)
training <- whats_left[train_index,]
```

Finally we assign the remaining to the **cross validation** data set:

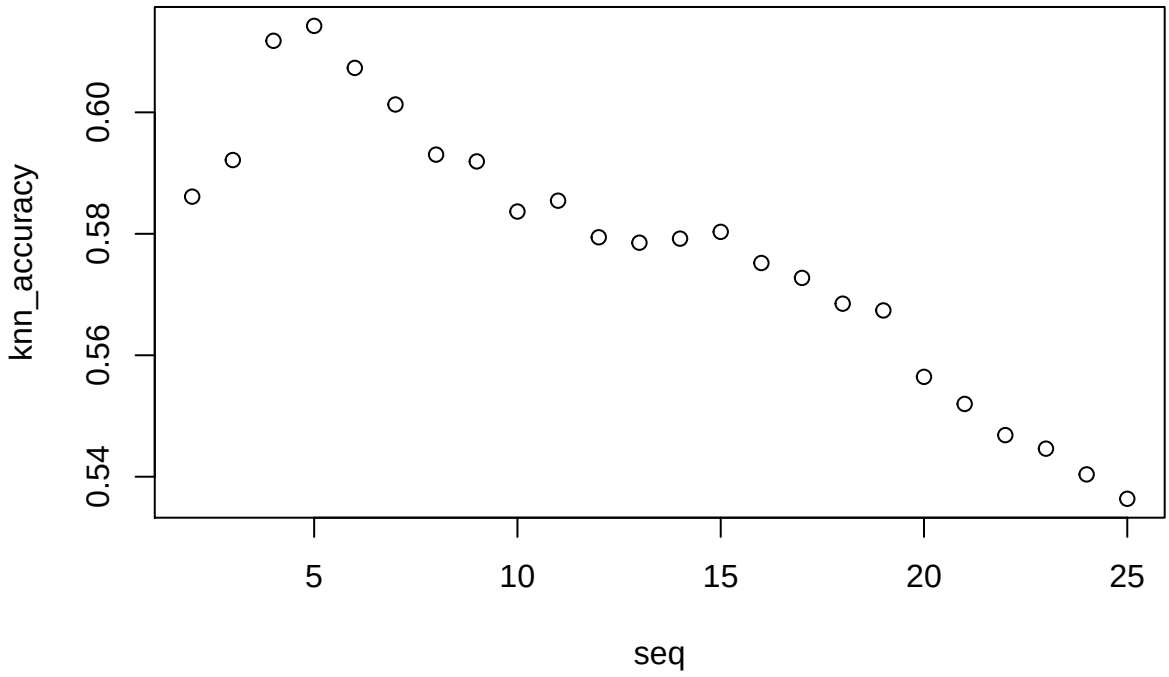
```
cross_val <- whats_left[-train_index,]
num_col <- ncol(king_rook)
```

Using KNN

KNN is susceptible to the tuning of the k parameter. Therefore, I apply several values of k to a function that trains a KNN model on the training data and obtains the value of k with the best accuracy. I'm using all 6 predictors in the definition of the training function:

```
seq <- 2:25
knn_k <- function(k) {
  knn_fit <- knn3(result ~
                    w_king_file + w_king_rank
                  + w_rook_file + w_rook_rank
                  + b_king_file + b_king_rank
                  ,data = training
                  ,k=k)
  knn_hat <- predict(knn_fit, cross_val, type="class")
  confusionMatrix(knn_hat, cross_val$result)$overall["Accuracy"]
}
```

```
knn_accuracy <- sapply(seq, knn_k)
plot(seq, knn_accuracy)
```




```
seq[which.max(knn_accuracy)]
```

```
## [1] 5
```

```
max(knn_accuracy)
```

```
## [1] 0.6142347
```

This method offers an accuracy of 0.6142347 even at the best value of k=5.

Using Random Forests

Random forest are very good at classifying so I decided to see how well this method would be able to predict the outcome. Also, this method allows for more tuning parameters to be set.

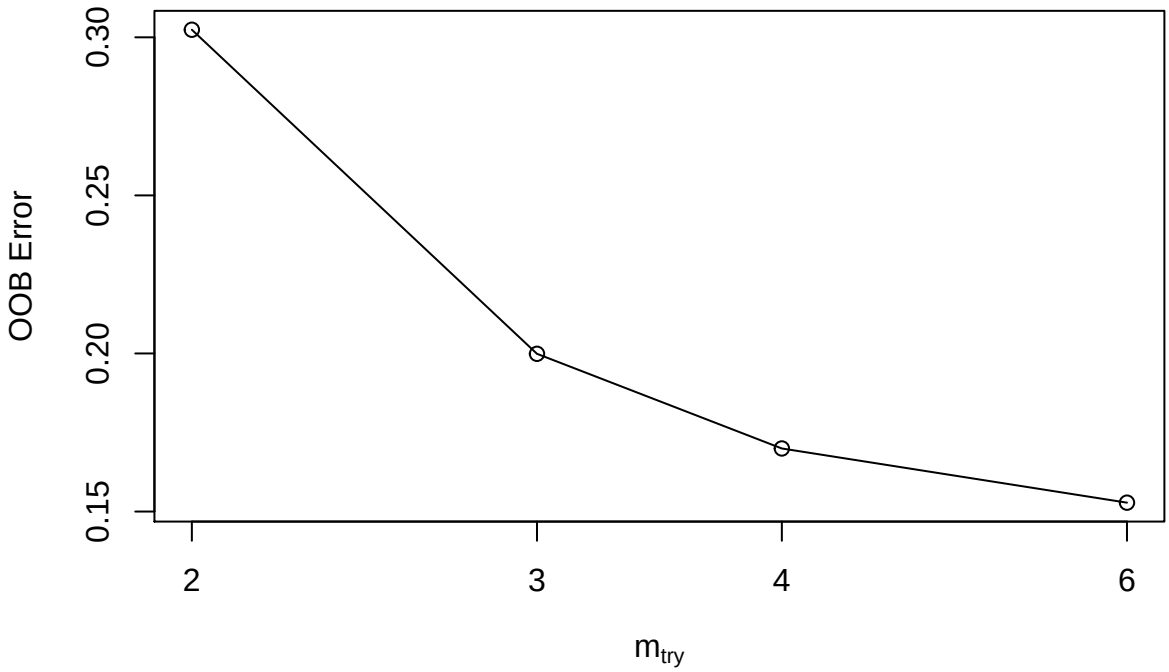
Tuning mtry

First, I train my *mtry* parameter to see what is a good value for later use in the final model's tuning grid:

```
trainMtry <- tuneRF(training[, -num_col]
                    , training$result
```

```
                    , stepFactor = 1.5
                    , improve = 0.05
                    , ntree = 250)
```

```
## mtry = 2  OOB error = 30.24%
## Searching left ...
## Searching right ...
## mtry = 3     OOB error = 19.99%
## 0.3389768 0.05
## mtry = 4     OOB error = 16.99%
## 0.1497773 0.05
## mtry = 6     OOB error = 15.28%
## 0.1008513 0.05
```




```
trainMtry
```

```
##       mtry  OOBError
## 2.OOB    2 0.3023929
## 3.OOB    3 0.1998887
## 4.OOB    4 0.1699499
## 6.OOB    6 0.1528102
```

```
best_mtry <- 6
```

This table shows the result of using the function *tuneRF* on the training data. The method will stop processing and show the table with values once it is unable to get an improvement of 0.05 from the previous step.

According to the documentation the lower the *OOBError* value the better the mtry. In our case this happens at $mtry = 6$. That's the best parameter for our model.

Tuning ntree

The other tuning parameter we can try is the number of trees, *ntree*. Here, we will try different values ranging from 1000 to 2000 in increments of 250. This step might take several minutes to run:

```
temp_mod <- list()
grid <- data.frame(mtry = c(best_mtry))
for (n in c(1000, 1250, 1500, 1750, 2000)) {
  set.seed(2021, sample.kind = "Rounding")
  train_rf <-  train(training[, -num_col], training$result, method = "rf",
                     metric = "Accuracy",
                     trControl = trainControl(method="cv", number = 6),
                     tuneGrid = grid,
                     ntree = n
                     )
  temp_mod[[toString(n)]] <- train_rf
}
model_results <- resamples(temp_mod)
summary(model_results)
```

```
##
## Call:
## summary.resamples(object = model_results)
##
## Models: 1000, 1250, 1500, 1750, 2000
## Number of resamples: 6
##
## Accuracy
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 1000 0.8214644 0.8253153 0.8274366 0.8273229 0.8295621 0.8327212    0
## 1250 0.8217987 0.8252319 0.8279372 0.8278796 0.8308982 0.8333890    0
## 1500 0.8228017 0.8243147 0.8269359 0.8273237 0.8305645 0.8320534    0
## 1750 0.8231361 0.8245648 0.8271028 0.8275462 0.8303973 0.8327212    0
## 2000 0.8239413 0.8252142 0.8277704 0.8276019 0.8294784 0.8317195    0
##
## Kappa
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 1000 0.8005621 0.8048112 0.8071823 0.8070725 0.8095028 0.8132109    0
## 1250 0.8009370 0.8047299 0.8077564 0.8076978 0.8109960 0.8139403    0
## 1500 0.8020487 0.8037192 0.8066498 0.8070797 0.8106129 0.8124536    0
## 1750 0.8024387 0.8039935 0.8068228 0.8073245 0.8104283 0.8131819    0
## 2000 0.8033180 0.8047537 0.8075715 0.8073934 0.8093915 0.8120870    0
```

```
best_ntree <- 1250
```

The loop above saves into a list several objects of class *train* -one for each possible value of ntree- and formats it for easy analysis and visualization using the *resamples* function.

We see that the best value of $Accuracy = 0.8333890$ happens at $ntree = 1250$. This is the value we are using in the final model.

Final Model

Now for the final model, I train using the obtained values of mtry and ntree:

```
grid <- data.frame(mtry = c(best_mtry))
set.seed(2021, sample.kind = "Rounding")
train_rf <-  train(training[, -num_col],
                   training$result,
                   method = "rf",
                   metric = "Accuracy",
```

```
                    trControl = trainControl(method = "cv", number = 6),
                    tuneGrid = grid,
                    ntrees = best_ntree
                    )
```

Now we are going to try this on the test set.

Result

Let's now use the *predict* function on the testing dataset using the final model:

```
sv <- predict(train_rf, testing, type = "raw")
mean(sv == testing$result)
```

```
## [1] 0.8495717
```

The PREDICT function achieves over 0.84 accuracy with this model. A surprisingly higher accuracy than KNN.

Conclusion

I didn't expect this result. Based only on the initial position of the pieces and without any type of analysis of the position itself, without using chess playing engines, nor feeding previous chess knowledge to the model, I expected the prediction to be no more than 70-75 percent accurate at best, given that the KNN method achieved poor results.

The bulk of the time creating this report was spent tweaking with the *trainControl* and *tuneGrid* objects, trying to adjust them to produce the best mtry and ntree values.

Does this result indicate that there are predictive methods of evaluating a chess position and determining what's the outcome without the need to analyze the position with engines? If the answer is yes, then this kind of predictive pre-analysis could improve a chess engine performance while saving precious time by selecting the appropriate path to winning/drawing without having to analyzing chess lines that don't secure the best outcome.

I think this approach deserves further analysis to decide one way or the other.

References

The following were used as reference to produce this report:

- Formatting R Markdown - https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf
- Markdown chunks of code - https://bookdown.org/yihui/rmarkdown/r-code.html
- Caret package - https://topepo.github.io/caret/index.html
- Cross validation - https://rafalab.github.io/dsbook/cross-validation.html
- Considerations for a neuronal newtwork to analyze chess positions - https://core.ac.uk/download/pdf/12756.pdf