

Universitat Politècnica de Catalunya
Barcelonatech
Facultat D'Informàtica de Barcelona

Predicting the Outcome of a Chess Game by Statistical and Machine Learning techniques

Héctor Apolo Rosales Pulido

Submitted in part fulfilment of the requirements for the degree of
Master in Innovation and Research in Informatics
October 2016

Abstract

This document will go through the process of Big Data analytics, which combines computer science, data warehousing and applied statistics. We plan to predict the result of Chess matches after a twenty full movements. To do this we are constrained to work with the complete database that was provided at the start of this project.

The Gorgo Base [12] consist of around three million matches, comes in an unknown database format, and once we were able to read it, we were confronted with it's size, this database is able to overwhelm any computer that tried to compute many operations at the same time, this was one major challenge to overcome. As with database this size, we had to spend significant of resources filtering out missing and faulty data.

To process this database we had to tokenize it, separate it into chunks we could actually compute, and then we started aggregating and filtering data. Aggregating data is an important part of any dataset creation, using all the database we were for example able to capture the average ELO of all the players we found. We also generated the score of every board later used to predict game results. At this step we generated our test and train files, we separated 70% to training and 30% for testing purposes.

One final challenge was to collect all the information of the board positions, this was challenging because we wanted to keep a record of the historical results for every game that was in our database, and to do this we had to compare and add results, and at the end we end up recording thirteen million board historical records. We did the same with the historical record of competitors, we stored their average ELO, and their results history, to create the competitors database.

The biggest problem in predicting chess matches is the enormous amount of legally possible board positions, it has been estimated at 10^{43} by Shanon [16] and others, but since we are not taking into account the endgame, because we want to predict the result at an early stage, we believe that we might be able to use the information on matches of this database.

Finally we gathered all the data from our three sources, the refined Gorgo Base, the Movement history, and the Competitors records, to generate a dataset we could work with. We applied an SVM with RBF kernel, and compared it to a random forest model. At the end we were

satisfied with our results, which showed us how powerful using big data is to solve problems.

Contents

Abstract	i
1 Introduction	1
1.1 Introduction	1
1.2 Project Structure	3
2 Project startup	6
2.1 Programming language discussion.	6
2.1.1 Python particularities.	7
2.2 The Gorgo Database.	8
2.2.1 Shane’s Chess Information Database.	8
2.2.2 The PGN format	9
2.2.3 Breaking down our database	11
2.2.4 JSON format	13
2.3 ELO	15

3	Data Cleansing.	16
3.1	Reading our dataset.	16
3.1.1	The Forsyth–Edwards Notation (FEN)	17
3.2	Filtering the Gorgo Base.	18
3.2.1	Chess relative piece values	19
3.3	Interpretation of data.	19
3.3.1	Python classes	20
3.3.2	Events.	22
3.3.3	Competitors	23
3.3.4	Board	24
3.3.5	Games	25
4	Grid Creation	27
4.1	Processing large quantities of data	27
4.2	The multiprocessing library	27
4.2.1	Grid Creation	28
4.3	Social Network Analysis.	32
4.3.1	Community detection.	32
4.3.2	Walktrap community detection.	32
4.4	Summary Statistics	35

5	Data Consolidation	38
5.1	Nodes and movements	38
5.1.1	Nodes.	39
5.1.2	EndNode Subclass	39
5.1.3	Movements	42
5.1.4	Creating the graphs.	43
5.2	Personal computer hardware limitations.	43
5.3	Board aggregation.	44
5.3.1	Our simple database.	44
5.3.2	Database design.	45
5.3.3	Querying the database.	47
5.4	Consolidating Competitors.	50
5.5	Final Dataset.	51
6	Analysis	53
6.1	R	53
6.1.1	Multiprocessing with R	54
6.1.2	Lapply	54
6.1.3	The parallel library	54
6.2	Derived data	55
6.3	Modeling the problem.	56
6.3.1	Kernels	56

6.3.2	RBF kernel	57
6.3.3	Support vector machines.	57
6.3.4	Random Forest	58
6.4	Building our model.	59
6.4.1	SVM with RBF kernel	59
6.4.2	Random forest model.	61
6.5	Results	63
6.6	Summary of Thesis Achievements	68
Appendices		70
A Grid Creation		71
A.1	Games statistics	71
A.2	Grid definition	75
A.3	Network Summary statistics.	76
B Database missing data.		79
C Results		83
C.1	Gaussian Kernel with three fold crossvalidation.	83
C.1.1	Without Draws.	83
C.1.2	With Draws.	87
Bibliography		90

List of Tables

1.1	Size of our datasets.	5
3.1	First filter of our data.	18
3.2	Value of pieces	19
3.3	Final json attributes.	26
4.1	Network short Summary	34
4.2	Summary statistics(short).	36
4.3	Result percentage	37
5.1	Data consolidation missing values (short)	50
6.1	Results of the SVM prediction for selected grid	64
6.2	Results of the Random forest prediction for selected grid	64
A.1	Basic Statistics	74
A.2	Grid Ranges	76
A.3	Network Summary	78
B.1	Percentage of Missing data	82

C.1 Result using SVM, without draws. 87

C.2 Result of draws using SVM. 90

List of Figures

1.1	Graphical flow of the project.	3
2.1	PGN format	10
2.2	Complicated PGN format.	12
2.3	PGN to JSON	14
3.1	Class diagram Competitors, Games, Events.	20
3.2	Player redundancy	23
3.3	Packages Diagram.	25
4.1	Plot of the grid showing mostly division with White ELO.	29
4.2	The grid with Black and White Subdivisions.	31
4.3	Community walktrap grid zero, second try.	33
4.4	Community walktrap grid 99.	35
4.5	Box plot of our figures.	37
5.1	Node class.	39
5.2	Movement class.	42

6.1 Random forest ROC comparison. 62

6.2 Drawless Results of the SVM model. 65

6.3 SVM model results with draw. 65

6.4 Drawless Results of the random forest model. 66

6.5 Random forest model results with draw. 67

List of Code Listings

2.1	Recursive function to split the 2GB file.	13
3.1	Python class example	21
4.1	Simple pool.map example	28
5.1	CreateTable.sql	46
5.2	Queries to our db.	47
5.3	ID selection.	48
5.4	Multiple ID selection.	49
5.5	Text data sample.	51
6.1	parLapply example.	55
6.2	R function to tune ksvm parameters.	60
6.3	R function to random forest parameters.	61

Chapter 1

Introduction

1.1 Introduction

Entertainment is a feature of human behaviour since the earliest of times, countless games and other forms of entertainment have been devised over the centuries, one of the earliest games recorded in relatively recent human history that it is still played worldwide is Chess, or Checkers.

Chess is one of the most easily recognizable board games, created around the seventh[3] century in what is now India, it started by the name of Chaturanga, it then evolved into it's modern version in the XVI century, and it was first introduced by the Moors in Europe. It was also introduced through the rest of the silk road from what is now Iran, going as far western Europe to the Chinese empire.

As in any other competitive task, humans have been looking for an advantage in competitive chess, for this any tool available to humans were used to try to obtain knowledge, mechanics and mathematics were used to "beat" chess from the earliest of times. Starting with the invention of *The Turk*, which pretended to be a machine operated chess player, it won against notable personalities such as Napoleon Bonaparte, and Benjamin Franklin, it attracted the attention of the people when it played and toured the world without anyone chess playing machine. However

the fact that a machine was able to beat humans attracted and captivated the collective human mind at the time.

One event at the turn of the century once again gained the attention of our society, it was the same kind of event, a machine, built by humans, could by itself overcome human competition, in a very complex game known to be dominated by skilled players and not by chance, it was the matches of Garry Kasparov vs IBM's Deep Blue. It was broadcasted and constantly updated by the media onto millions of homes, it put Chess at the spotlight, but most of all we saw how this specialized machine was able to beat the pinnacle of the human mind and skill, the great Chess Grand Master of the era.

This trend has continued, at the time Deep Blue was a room sized machine, but then personal computers started beating Chess Grand Masters, and now mobile phones are able to overwhelm human competition. Nowadays the best "*players*", are a combination of both human/computer teams, in a variant of chess called Advanced chess, where a human player chooses a program to help him make decisions which he ultimately makes, centaur chess as its called beats both, human and computer chess players individually.

According to IBM's Shanon [16], there are around 10^{43} possible number of board positions, which is a lower bound estimate, thus making it almost impossible to solve the problem using brute force, and also it explain the complexity of chess itself, we think it might be possible that the hardware of modern computers has improved enough to provide a somewhat accurate prediction, using advanced machine learning techniques, and database technology, using our personal computers.

The goal of this project is to use a big database, that was provided to us, to predict the result of chess matches after twenty full movements(after black player has moved), then use all the characteristics and properties of the data we manage to capture in our data process. For this we will not design an chess engine, or use more than what we have in our database.

The current document is inspired by the work that Vence [19] started doing, we are not calculating distances of pieces, as he proposed, but we are limiting ourselves to process the enormous

database we have in hand and work with the complete database to prove that we don't need to be an expert on the field, or the actual game in this case, to predict the outcome of a game if we have enough data on chess. We however use the database that was found by him which is what kickstarted the interest on solving this problem. The next subsection will explain how we worked through this problem and how we obtained our results.

1.2 Project Structure

We start by talking about how we structured the project, this document reflects the organization of the code, we think, it would make easier for anyone with an interest in the project to read, both the code and this document.

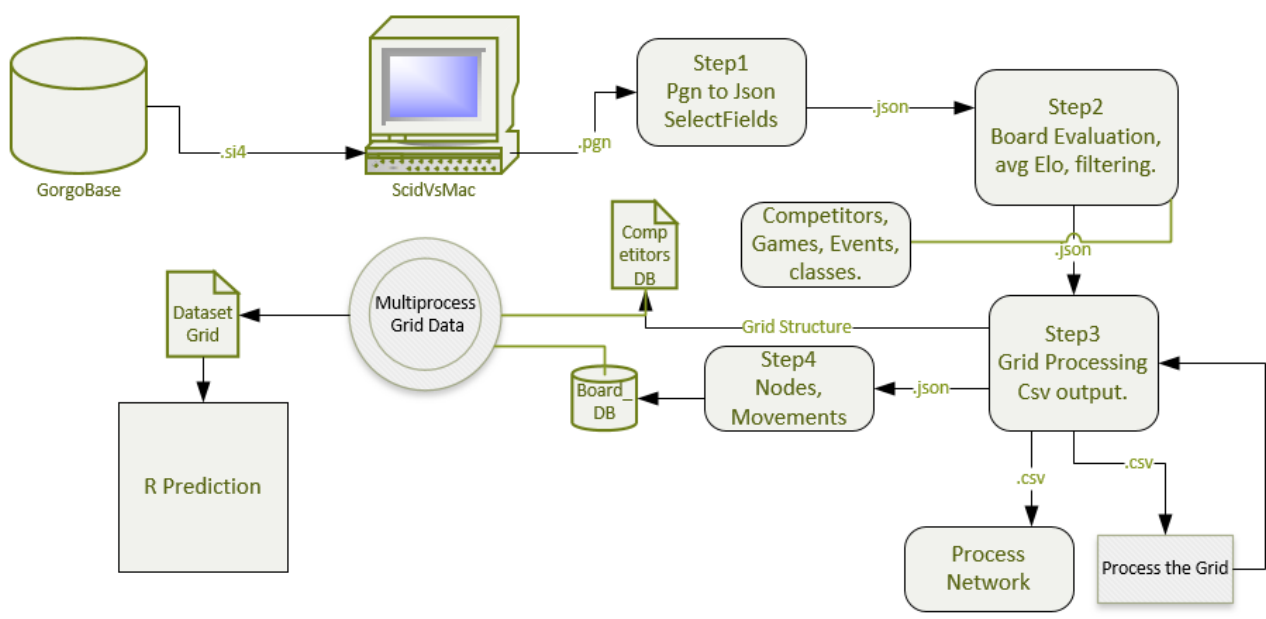


Figure 1.1: Graphical flow of the project.

As figure 1.1 shows we start with the Gorgo Base[12], then after we have transformed it into a more Python friendly format we start by modifying the data. The majority of time we invested in this project is devoted to process and transform the data from a database format, with raw data, to a more friendly format that has more concise database. To do this, in every step of the way we performed different kinds of routines to purge data we deemed useless, we also aggregated data that we considered would help us better predict outcomes. These are the

Chapters/steps into which we divided this document:

- **Step 1:** Presented in Chapter 2, we start by describing some of the new formats and the steps we performed to transform the chess database into something we could make use of. Once we were able to explore our data we started by filtering out data we deemed irrelevant. However since the dataset was so big, it was a challenge only to read the dataset.
- **Step 2:** In Chapter 3 we started to compile information about the data, and to explore the overall dataset, although we only get a small look into the data because of its size it's enough for it to make us realize of some problems we had, to produce and refine the data a little bit more.
- **Step 3:** Chapter 4 explains this step we make a preliminary analysis of the data, which was made by producing some csv files about interesting information we wanted, this is also where we determined how we were going to process the data in a grid like fashion, because it was impossible to load with our personal computer at once.
- **Step 4:** Chapter 4 deals with trying to make use of our data, we started by creating a couple of tree structures, the first one captures every sequence of boards for every read game, each with nodes going back and forwards, we also keep track of movement number in every game, this in another tree. Even though we couldn't use this as we originally wanted, we were able to create a database using the data generated using these trees, after we had generated our databases using all the data, we then created a consolidated dataset for every grid element.
- **Analysis and prediction:** In the final Chapter 6, we build a couple of models to predict the result of matches after 20 movements using the test dataset, we created two models to predict matches that include draws, and another that excludes them. To do the modeling we are using a kernel SVM function, for the second model we implemented a random forest to compare the two methods. We then present our results, and have a discussion on them.

- **Conclusions:** We present our conclusions and possible future work directions if the reader is interested in continuing the project.

We will also show how we went up handling the data, and for every step we made we had some problems handling the size of the database. Figure 1.1 shows the historic size for the database, for each of these we changed and tweaked things to making it easier on us to process ¹.

	Description	Format	Size
1	Gorgobase Original	si4	500 MB
2	Gorgobase PGN	pgn	2000 MB
3	Thousand files	json	14,600 GB
4	Refined Json	json	18,350 GB
5	Graphs in runtime	Python	>50 GB
6	Test	json	4.53 GB
7	Train	json	10.58 GB
8	Alt Test	json	2.41 GB
9	Alt Train	json	5.63 GB
10	Board.db	sqliteDB	2.48 GB
11	Final Test	csv	64 MB
12	Final Train	csv	152 MB

Table 1.1: Size of our datasets.

¹For the alt versions we deleted half moves to reduce the file size.

Chapter 2

Project startup

2.1 Programming language discussion.

One of the first big, important decisions to make early in the project was to decide which would be the programming language that we would use to develop our project. Out of all the programming languages, we had to start with open source languages, that had big capacity to process big chunks of data, and also had a libraries to work with statistical analysis. Python, R and C/C++, made the most sense, each of them had big advantages over the others by design.

- Obviously R is a statistician's war elephant, it has a wide array of libraries and a very big community which updates and uploads new packages with functions to use and test.
- C/C++ is the most used language for efficient low level programming, which has been around for at least thirty years, because of this is well documented, also since is open source it has several proprietary versions, which would adds to the complexity in developing C/C++, a very passionate community that is always very open to help and implement new libraries, such as those for statistical analysis, and machine learning.

C++ is a proper object oriented language, and unlike scripting languages like Python 2.x, is type safe, which means that the developer must declare the type of the variables it will use. This in our opinion is a good design option because other developers will be

able to see what type a variable is supposed to be, and this information is also used by the IDE(development environment) to help the developer avoid errors by pointing out if we are inputting a different type of variable to a function etc.

- Python in our opinion is between the two, it was conceived as a programming language, but it is not a compiled language, which affects it's performance in computationally intensive tasks, this is also an advantage in some cases, by being a high level language it's easily portable to other systems by avoiding compilation.

Another advantage we see in it, is that it forces the users to write the code in a legible way, by forcing indentation to properly run the code, which is a very welcome advantage when working with other developer's code. It also includes very good libraries, such as SciKitLearn, numpy, matplotlib and others.

Even though we were familiar with the three, we had not used C++ in a few years, and Python and R were our everyday tools, so we decided to do most of the processing of the data in Python 2.x, because it's simple enough for us to use and it's also more organized and less OS dependent than C/C++. And for the final analysis step we will use R, to create or models and most if not all of our plots.

2.1.1 Python particularities.

Python's GIL

The global interpreter lockdown [13], is a known limitation when multithreading with Python 2.x that was put into the system by design(in 1995 there were not commercially available multicore systems), it does not allow the use of all the processors at the same time, it locks the access to the interpreter by serializing it usage, the *GIL* does not affect I/O tasks, such as communications with a web service. This problem mainly affects the *Thread*, and *threading*, class and interface in python. The *GIL* does not affect other Python compilers such as Jython

and IronPython. Basically the GIL makes threading only a simulation, except when we connect to external services.

There is another package that works like the Thread class, which has an interface similar to that of Thread, thus making it easier to use, it's called **Multiprocessing**, as the name states, it creates a new process, which is not exactly the same as a new thread, it does get over the GIL. We must remember that processes do not share the same memory space, so everytime we generate a new process much more overhead is generated by copying variables and starting/terminating the new process.

Finally Multiprocesses do not have to synchronize which make them more straightforward in their implementation, and debugging.

This was important to us because of the large number of files, that we had, and at first we didn't have the environment set up for this to work with multiprocesses, so at first processing the files took longer than it would if we had used multiprocesses from the beginning.

2.2 The Gorgo Database.

After we downloaded the database from the gorgobase [12] website, the downloaded file had to be decompressed and transformed into a format we could read with Python, from its original state in the Scid database format, we knew it was some sort of database because the files had a binary format, meaning that when we opened it with a text processor, we only saw gibberish. So we had to find a way to process this file that was in this unidentified format.

2.2.1 Shane's Chess Information Database.

Scid (Shane's Chess Information Database) is an open source format, that was created by the Chess community, as a way to store chess matches to store, for later analysis. It's also used to view and maintain huge databases of Chess Games.

Scid has multiple features useful to serious or hobbyist alike, it's database format is very popular and widely used by applications.

The gorgobase [12] is a Scid's database, therefore it comes with the following files:

- **The Index file (.si4)** it contains a description for the database and a small fixed size entry for each game. Every game includes: the *result*, *player*, *event*, *site*, *name ID*.
- **Name File (.sn4)** Contains all *player*, *event*, *site* and *round* data. This is usually the smallest file of the three.
- **The game file (.sg4)** This is where the moves are actually saved, it also contains variations and comments for each game. Scid actually only saves each move instead of the whole board, this helps with storage, as we shall see in chapter 3.3.3. Most moves take only a single byte, this is done by storing the piece to move in 4 bits ($2^4 = 16$ pieces), and the move direction in another 4 bits.

To be able to read this database we used a computer program that is also able to transform the data into another format. *Scid vs* , is an open source engine to play, record, analyze and replay chess games, which is available in most platforms and it's being maintained by what it seems to be a large community.

We were able to load the Gorgo Base to the program, and then after looking around we found a way to transform it into a pgn format, which was another format we were not familiar with, however this time we had a human readable file, and we saw a file with all the games that were on the Gorgo Base. Next we will explore this database format, and see if we could find something useful to process it.

2.2.2 The PGN format

The PGN *Portable Game Notation* is a human readable format for storing chess games, it stores both the moves and the properties of the game, it has wide support by most chess software and libraries.

As we saw in table 1.1, the size of the compressed database to this uncompressed database, greatly increased the space it used in memory, we went from "only" 500 MB, to over 2.4GB, which fortunately was not a problem to us since we had recently increased and formatted our personal computer.

Our gorgo base now had around three million examples as figure 2.1 shows. We now had to figure what each field meant and how to work with this format.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6
8. c3 O-O 9. h3 Nb8 10. d4 Nbd7 11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7
14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5 Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6
20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6 23. Ne5 Rae8 24. Bxf7+ Rxf7
25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5 hxg5 29. b3 Ke6 30. a3
Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5 35. Ra7 g6 36. Ra6+
Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6 Nf2 42. g4 Bd3
43. Re6 1/2-1/2
```

Figure 2.1: PGN format example[20]

The pgn file is composed of several tag pairs, which are enclosed between brackets, the data is provided in seven fields that may appear in the following order:

- **Event:** The name of the tournament or event.
- **Site:** the location of the event.
- **Date:** the starting date.
- **Round:** the playing round ordinal of the game event.
- **White:** white pieces player in last name, first name format.

- **Black:** black pieces player, same format as before.
- **Result:** The result of the game, 1-0 White, 0-1 Black, 1/2,1/2 Draw, or * other, e.g. ongoing game.
- **FEN:** Other important tag is the **FEN** tag, which is included to record partial games, that start in a different position, or other chess variants, like chess 960.

As we can see the pgn stores each movement of the data that as we can see in it's simplest way is storing the movement of each white and black players, and the movement number is indicated for each movement. But there are far more complicated examples of this types of pgn data.

Also it's worth noting that the data was not uniform throughout the 2.8 million games, for some games there were some data missing, and some had more fields, which had to be filtered later.

After encountering that there were some matches which didn't have the same format, as figure 2.2 shows, we realized that we would need some sort of a library to process matches such as this, and since this was actually a format that the chess community had created, we thought we might find a library that would help us process it.

2.2.3 Breaking down our database

When trying to process data files as large as this, one must take into account that any file stored in our HDD, it will increase its size if we load it into main memory, therefore we will need much more capacity to be able to load it, and even though we had sixteen gigabytes of main memory space, we couldn't make it work, we first started trying with a scrip from Rasmus [6] that we thought could help us, but couldn't make it work because of the size of our pgn file.

Then we came up with the idea that if we divided the large database we had into smaller files we could be able to process each one of them independently without problems. To split the file

```

[Event "2nd HM Corus"]
[Site "?"]
[Date "2008.??.??"]
[Round "?"]
[White "Matous, M."]
[Black "[+0034.20e6g4"]
[Result "1-0"]
[EventDate "2008.??.??"]
[SetUp "1"]
[PlyCount "17"]
[FEN "3b4/8/4KnP1/7P/6k1/2N5/8/8 w - - 0 1"]

1. h6
  ( 1. g7 $2 Kxh5 2. Kf7 Ng4 )
1. ... Kg5 $1 2. h7
  ( 2. g7 $2 Kxh6 3. Kf7 Kh7 4. Nd5
    ( 4. Ne4 Ng8 )
  4. ... Ng8 )
2. ... Nxh7 3. g7 $1
  ( 3. gxh7 $2 Bf6 4. Ne4+ Kg6 5. Nxf6 Kg7 )
3. ... Nf6 4. Kf7 $1
  ( 4. Ne4+ $2 Kg6 $1 )
4. ... Ng8 $1
  ( 4. ... Kf4 5. Nd5+ )
  ( 4. ... Kf5 5. Nd5 Ng4 6. Ne3+ $1 )
  ( 4. ... Kh5 5. Ne4 $1 Ng4 6. Ng3+ Kh4 7. Nf5+ )
  ( 4. ... Kh4 5. Ne4 Ng4 6. Kg6 Ne5+ 7. Kh7 )
5. Ne4+ $1
  ( 5. Kxg8 $2 Kg6 $1 6. Ne4
    ( 6. Nd5 Bg5 $1 7. Kh8 Kh6 $1 8. g8=Q
      ( 8. Kg8 Kg6 9. Kf8 Bh6 )
    8. ... Bf6+ $1 9. Nxf6 )
  6. ... Bh4 $1
    ( 6. ... Bc7 $2 7. Kf8 $1 )
    ( 6. ... Bb6 $2 7. Kf8 $1 )
  7. Kh8 Kh6 8. Kg8 Kg6 9. Kf8 Be7+ $1 10. Kg8 Bh4
    ( 10. ... Ba3 {(or} 11. Nf6 Bb2 12. Kf8 Ba3+ )
  )
5. ... Kf5 6. Kxg8 Kg6 7. Kh8 $1 Kh6 $1 8. Nf2 $1 Bf6
  ( 8. ... Bb6 {(main} 9. Ng4+ )
9. Ng4+ 1-0

```

Figure 2.2: Complicated PGN format.

we would need to be careful not to cut down games in the pgn, so we had to figure out a way to do this more cleanly.

To achieve this we divided the file by obtaining the length of the file divided by a thousand, but before we even cut the file, we had to find first where the PGN started. If we remember from figure 2.1 the starting point of the PGN is a blank line followed with a line starting with the opening bracket, the "[" character, having this in mind we created a recursive function, shown in code snippet 2.1, that would start at a point where our initial division gave us, and then move ahead until it found this pattern

```
1 //split_pgn.py
def goToPGN_Start(data , i):
3     if (len(data[i])<1):
        if not (data[i-1].startswith("[")):
5         return i
    return goToPGN_Start(data , i-1)
```

Code Listing 2.1: Recursive function to split the 2GB file.

After we had all the starting points for each game, we then wrote the range of lines corresponding to each of the corresponding files, with this we ended up with a thousand two megabyte files, instead of a 2 gigabyte file. Now we were able to process our database in a more comfortable way without breaking down our computer.

2.2.4 JSON format

Now that we had a way to process the files independently, we decided we needed to convert our PGN files to JSON, a more known and straightforward "database" file format.

JSON(JavaScript Object Notation) is a lightweight format that is used for interchanging and storing data. It's format is simple to understand and it has wide support in several programming languages, applications, libraries etc.

We found a script useful to change our pgn file to a json format on github [6], that we modified to serve our purposes but was a very good starting point. However we also added something

that we thought would be useful, we converted the movements that are shown in the pgn file into complete boards, this was not a fast computation because for each of the three million games we have on average 40 movements each, as shown in our analysis in appendix A.1, so to try to increase a little bit the speed we tried using the threading class in Python, which unfortunately was useless because of the GIL we saw in section 2.1.1.

The board conversion was done with the help of the PythonChess [10] library, it reads the pgn movements and converts it using the function `board.fen()`, this will show the entire board, movement by movement, this will be useful later on the project, and also added a bit of a problem.

```
{
  "Event": "Open",
  "Site": "Liverpool ENG",
  "Date": "2007.09.03",
  "Round": "1",
  "White": "Minnican, A",
  "Black": "Frith, Robert",
  "Result": "1-0",
  "WhiteElo": "2110",
  "ECO": "A00",
  "fen": ["rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1",
    "rnbqkbnr/pppppppp/8/8/8/2N5/PPPPPPPP/R1BQKBNR b KQkq - 1 1",
    .
    .
    .
    "3r1k2/p5pp/1p1np3/2pb4/2P3P1/PP1R3P/5B2/4R1K1 b - - 0 27"]
}
```

Figure 2.3: PGN to JSON(shortened).

As we can see in figure 2.3, our first json file is almost like the pgn, it looks familiar enough, however notice that instead of having movements represent the state of the board, we had the complete board for every half movement. This increased massively the size of our dataset, as seen in table 1.1, it went to up to 14 gigabytes, from a little more than 2 gigabytes.

2.3 ELO

So far we have seen talk about the ELO, statistic, some of our readers might have heard of it from Baseball games, or NFL games, or even as a replacement for the controversial FIFA ranking of football. However ELO, was originally proposed and is named after Arpad ELO[7].

Historically the highest ever rated players are in the range of 2700 and 2800, and the average ELO of competitors in a national tournament is 1500 with 250 of standard deviation.

Next we present how to calculate the ELO of a player in equation 2.1,

$$R_{new} = R_{old} + \frac{K}{2} \left(W - L + \frac{\sum_i D_i}{2C} \right) \quad (2.1)$$

W is the number of wins, L the number of losses, D is the opponent rating minus the player's rating, C Is equal to 200, and K is 32.

The ELO captures the strength of a player, it compares the previous performance with the latest versus other players, and it respond rapidly to unexpected losses or victories. The ELO is well known as a predictor of the outcome of a match.

At the end of this Chapter we are now able to read the files and see what's going on, however as we can see in the *FEN* field, we still have some work to do in refining the data, we would like to know what data we have and for this we're going to have to continue to process and refine it, as we will see in the next Chapter.

Chapter 3

Data Cleansing.

We define data cleansing as the process during which the data is transformed from its raw form into a more processed standardized form, in this chapter we will see how we manage data cleansing problems such as inconsistent data, incomplete data, missing data and erroneous data. Data cleansing is always an important part of data warehousing and data analysis, it would help us to have better predictions if we pay close attention to our data, and make sure it doesn't have any inconsistencies that might introduce undesired noise to our prediction models.

Since most of the data in this database was introduced by a human process, it has several inconsistencies, as expected, we alleviate this problem in this chapter and we also start producing some interesting information about our database in hand. To solve this we had to process the data, this implies we had to treat it, this is when we start to create some code infrastructure that we will later use to the data aggregation phase, but this will also be useful to perform the data cleansing.

3.1 Reading our dataset.

Once we had the data in an easier to read format, it was time to start manipulating the data, but before doing this, we needed to know what the `chess.python` library [10] added along with the board information

3.1.1 The Forsyth–Edwards Notation (FEN)

The FEN notation is a well known standard for describing the specific board position of a chess game, as well as other information regarding the description of the game in hand.

Taking the following fen notation from the Figure 2.3,

```
rnbgkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

We see that the record has six fields separated by a space[1] The fields are:

1. The board, the pieces are named in English(P = pawn, N = knight, B = bishop, R = rook, Q = queen, K = king). White pieces are represented using upper case characters, black pieces are shown using lowercase characters. Empty spaces are annotated using the number of empty squares. The board is shown from whites perspective, and ”/” separates the rows.
2. Active color, w means it’s whites move, b means it’s blacks turn to move.
3. Castling availability, K/k means player can castle on the king’s side, Q/q player can castle on the queen’s side.
4. En passant availability for a pawn.
5. Halfmove clock, the number of halfmoves for each player to determine if the game is a draw, depending of the rules of the particular game/tournaments, games can end in a draw if the clock reaches fifty.
6. Number of full moves, incremented after Black’s move.

Rethinking of the FEN.

Since we thought we would be able to perform some operations on the individual boards, we thought it might be better to replace the numbers shown in the FEN, with a string that indicates

that they were representing the same characteristic in the board, this certainly increased the size of our dataset as table 1.1 shows, we thought that we were adding something of value.

We replaced each of the numbers in the FEN data with points ".", the thought was that this will be better to interpret if we used it for pattern searching. To do this we used a combination of regular expressions and the `string` library in python, to read the numbers and replace them with the number of characters that were indicated by the number. This is found in the `board.py` file. We also got rid of the extra information on the right, however if we were to produce a better evaluation of pieces, as section 3.2.1 refers, one that reflected possible movements by player, we would need this information.

Our new FEN pattern became:

```
rnbqkbnr/pppppppp/...../...../...../...../PPPPPPPP/RNBQKBNR
```

However after doing a few test we discovered that we wouldn't be able to perform any operation on these strings, other than using them as ID's, however this didn't impact negatively in our project, so we left it like it was.

3.2 Filtering the Gorgo Base.

Since we had a somewhat inconsistent database, we started by selecting a few fields we thought might be useful in the future. As we saw in figure 2.1 and figure 2.2, our games had fields that appeared only in a few matches, therefore to reduce our database size we filtered some of the data, and at the same time we standardized our database to contain only the following fields:

White	Result
Black	Name
White Elo	Round
Black Elo	FEN

Table 3.1: First filter of our data.

We were accepting missing values for all of these fields except for the *result* field. With this

we aimed at making the dataset a little bit more consistent, for example a game that had no conclusive result would be useless to us. This would be a recurring theme in the development process, we continued to purge games that contained faulty information, or improve on the data as we later will show.

3.2.1 Chess relative piece values

At this step we thought that having an evaluation function to obtain a metric of the board, that would show us who, if anyone had any advantage. Using piece values that are widely known, these values ignore the position in the board of the pieces and the availability of possible movements per player, which is what more precise evaluation functions do, as shown by[9]. We will discuss the usage of this table in section 3.3.4.







	Name	Value
	Pawn	1
	Bishop	3
	Knight	3
	Rook	5
	Queen	9
	King	200

Table 3.2: Value of pieces

3.3 Interpretation of data.

The next step we wanted to do was to standardize the data, because the dataset was so enormous, it contained all kinds of games, with some of errors and missing data, thus we had to process it, and to process it we had to manage the data that we were reading.

Figure 3.1 shows the final class structure we built to process the data we considered would be relevant in the prediction stage, table 3.1 shows those fields, after making this first process, trough the data we were going to eliminate mainly those fields that didn't have a result, or those in which the starting FEN was not the the standard one, and later we added a more

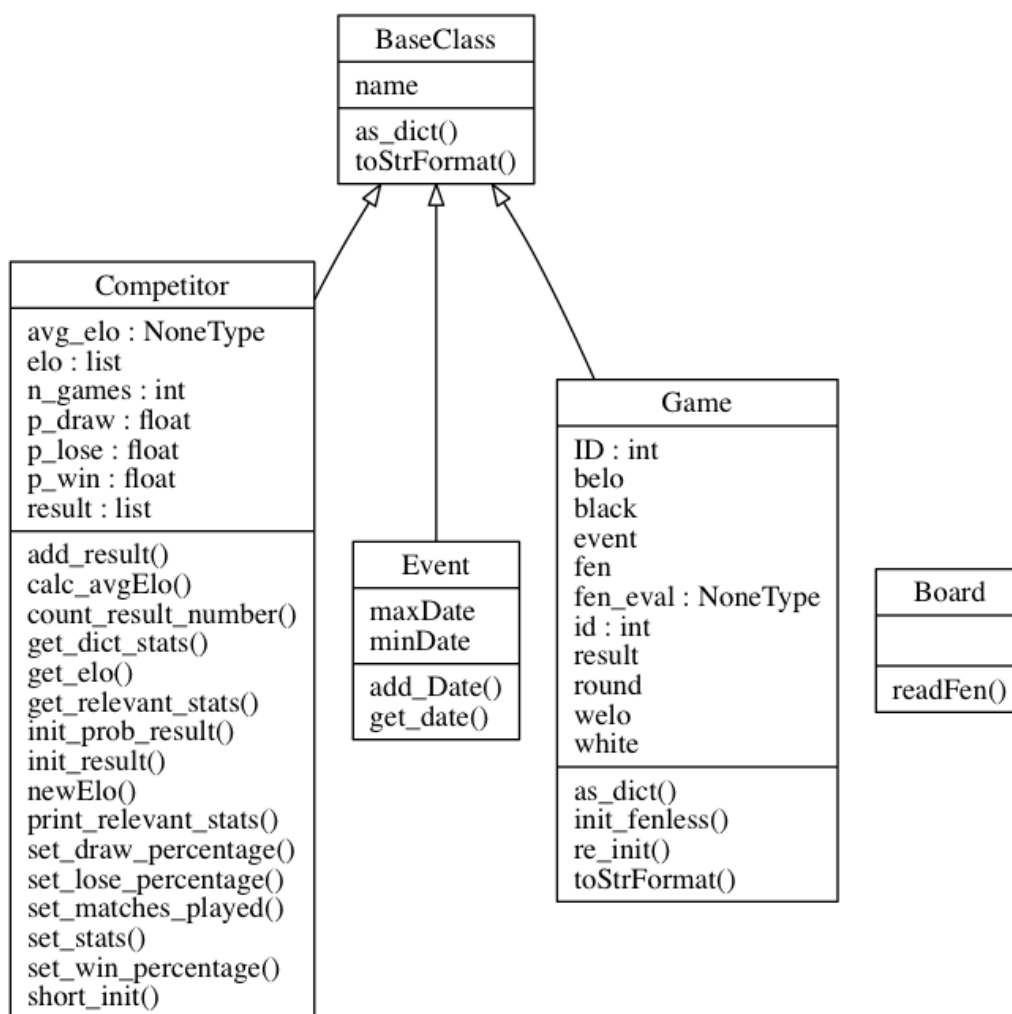


Figure 3.1: Class diagram Competitors, Games, Events.

complex filter that we explain in section 3.3.1. This would reduce our dataset from 2.8 million games, to around 2.1 million games, although we lost a high percentage of games, because of it's sheer size we think it won't make a difference.

3.3.1 Python classes

In Python as in other object oriented languages, there is a global base class, which all objects must inherit, in Python this class is the **object** class, and as in other languages such as Java, it has class methods that are predefined, such as `__name__`, notice that all of these special methods, start and end with double underscore, there are also other methods that are predefined by the object class to be customized by the developer.

Next we present our base class that we used to inherit to the rest of our classes.

```
//base.py
2 class BaseClass(object):
  # name = "" # static class property
4
  def __init__(self, name):
6      # constructor
      self.name = name
8
  def __str__(self):
10     # returns string conversion when needed
      return self.name
12
  def __repr__(self):
14     # useful for debugging purposes
      return "name:%s" % (self.name)
```

Code Listing 3.1: Python class example

Here we see the two types of class properties that are defined in the `object` class which we are inherit(as all other classes in Python do). First we define our constructor, which we use to assign our object property, `name`, notice the commented line right below the class definition, that is a global class property, and it's a static property.

The difference between the two is that the class property is seen and can be modified by all instances of the class, and the object property is unique for every class instance. This small difference caused us some problems when implementing this simple class structure, because compared to other languages, like Java, the static class methods are very clearly defined, but here in Python it completely depends on where in the code is the variable declared.

The next two definitions were defined for debugging purposes, we sometimes needed to print objects and since these two definitions are known by compilers and the Python language designers, we can take advantage of their use.

- `__str__` helped us when printing an object we could simply write `print object` and it would print whatever definition of the object in this method.
- `__repr__` this was also very useful because in the IDE's debugger this property is used to print the object definition, which we replaced by our own method which sometimes was

the number of games this object had participated in, or the win lose draw percentage.

As the *object* Python class, we used the *BaseClass* to start all the classes, we defined the most basic properties that all of our classes would be using and defined a few of properties that later we accessed via the *super* clause.

Our classes.

As pointed out in section 3.3.1, we created this small package of classes for the purpose of collecting and processing the data from the Gorgo base.

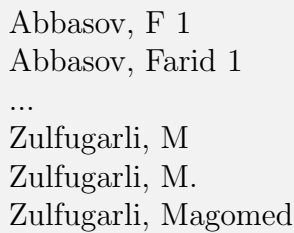
For each of these classes we created a simple way to initialize the data, and then we added a new constructor for the class because it was not very clear how to do this because in Python is impossible to overload a function. To solve this we used the decorator `@classmethod`, which creates a method that is bounded to the class, but not to any particular instance, with this we can create several constructors that return an instance of a class. This will be useful when obtaining and manipulating data after we preprocessed it.

3.3.2 Events.

The events are the tournaments, that contain several matches, and rounds, they also defined a date. This was a particular headache because the data being not very refined, the date came in different formats, that we had to process. The Python community developed[2] a very comprehensive parser that could handle most of our dates formats, as they were presented, in DD/YY/AA, written form or many other forms. With the help of the *dataparser* library we were able to move over this problem right away.

3.3.3 Competitors

Usually when you find a database over the Internet, no data cleaning or preprocessing has been performed, we usually find that it has not been cleaned, some redundant data, slightly different names for the same thing etc.



```
Abbasov, F 1
Abbasov, Farid 1
...
Zulfugarli, M
Zulfugarli, M.
Zulfugarli, Magomed
```

Figure 3.2: A player is shown in different entries.

Figure 3.2 shows, as expected some problems, where with dataset of this size we notice data redundancy in respect to the player's names. We see that there's a high probability that there are several players that were captured with slightly different names, thus they could be counted more than once by the program whereas they were all the same player. Because of this we created a method that tries to collapse several players that are named with a similar name into one, this could be important later on the process of analyzing players. Although didn't merge all the competitors with different names, we did merge all redundant names in the database we solved some of these problems.

Apart from the name, using this class, we created a method in which we created a dictionary of players which stored a name with a list of ELO's we found related with this name, using this information we created an average ELO, of each player we found. Using this in the next iteration of our dataset we added a couple of new fields that we named, **white_avg_elo** and **black_avg_elo**, which we later use to filter games, all of our games have an average ELO regardless if for a match they have ELO or not.

Later in the development of Chapter 5, we added a little bit more information on this dataset, that we used to capture the record of the player, in wins, losses and draws.

3.3.5 Games

Essentially the database are records of games played over time, it made sense to us to use the same format to iterate through the database over the same concept, so we organized the other classes around the *Games* class as figure 3.3 shows.

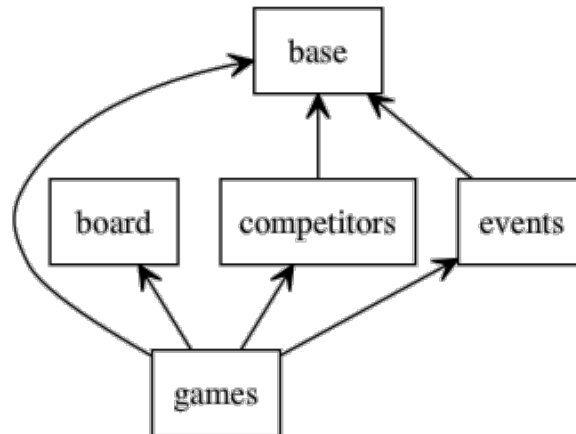


Figure 3.3: Packages Diagram.

Before computing the whole games, we first computed all the competitors over all of our files, then with the ELO list that each competitor had we calculated it's average and saved it as a new property to track in our the In the Games class we do not perform difficult computations, we replace the format of the FEN win/lose/draw, with a simpler one, 1 for white, 0 for draw, and -1 for black. we only store the other instances related to each game, like the event, and the players competing in a particular game. To process the games we load the competitors that were previously processed, and processed each game, load the event from the data, and then call the Board class to obtain our formatted FEN, and the "fen_eval" from the board and add them both to different list. Then we return a dictionary of the game and the relevant fields, to be stored in a new JSON file.

At this point we would still need some new modifications to our data but with this we were happy to start working and knowing about our database.

Continuing At the end of this chapter, we have generated a new json file which only included our desired files, as we talked about in section 3.2, we had some fields we were interested in,

then during this phase we aggregated some data and we ended up with the following adding the following fields and creating a new json database with the fields presented in table 3.3

White	Result
Black	Name
White Elo	Round
Black Elo	FEN
White Avg Elo	fen eval
Black Avg Elo	

Table 3.3: Final json attributes.

In the next chapter we will start working out some data to try to create a more refined version of our dataset.

Chapter 4

Grid Creation

4.1 Processing large quantities of data

During the development of this project we encountered with the computer processing for long hours, which for us was a very different experience with programming because once we set a program running, usually in a few seconds we would get the result, however this was not the case for this project and sometimes we encounter problems that we did not expect, the program would throw at us some errors, that we would found out hours later only to start over again.

To try to alleviate this problem from the beginning we tried to use the `threading` library, but realized that they were not effective because of the GIL 2.1.1. This was a bit of a frustration for us, but then we learned that the `multithreading` library was not affected by the GIL, so we investigated how to use it.

4.2 The multiprocessing library

The `multiprocessing` library, introduced in Python 2.6, backs spawning a process in our local processor or a distributed system, the biggest advantage for us is that it uses an API similar to that of the `threading`, that we knew beforehand, so this was something we were happy to

```
1 def f(x):  
    return x*x  
3  
4 if __name__ == '__main__':  
5     pool = Pool(processes=4)           # start 4 worker processes  
6  
7     # print "[0, 1, 4,..., 81]"  
    print pool.map(f, range(10))
```

Code Listing 4.1: Simple pool.map example

use in our project. Another advantage that we had while using the multiprocessing is that we will be processing the files independently of each other, so having the data treated in different processes makes this faster, and since we are not interchanging data from one process to another simplified its implementation, because we did not have to worry about locks or synchronization errors etcetera.

Even though there are different ways to implement the multiprocessing [14] library we implemented it using the `Pool` object, which is implemented in the same way as the `saapply/lapply` in R, as the following example shows:

Notice we have a definition of function `f` and we pass the function to the `map` method with a any list, for our function to process. *Pool.map* returns a list with all the results in a synchronized way, meaning it respects the same order of the input list. There are other, faster methods to do this asynchronously, but we didn't use them. We used this function for the rest of the project as a way to process independently each of the files we generated, which helped us reduce processing and waiting time to theoretically a quarter of the time.

4.2.1 Grid Creation

Since we couldn't read the whole dataset and perform operations with it before our computer completely stopped working we had to devise a way to operate the data and still obtain relevant

results.

Our data had been separated in a thousand files, as we explained in Chapter 3, that contained many types of games and tournaments, from amateurish level to the big great masters.

To better handle this, we decided to order the data into a hundred files, that would then be separated into test, train files. To do this separation we used a dataset that we generated using all the files, with the names and average ELO, of all competitors. We decided to use a hundred separators instead of a thousand because our computer was able to organize and process several files at the same time, and we calculated that a hundred files would not overwhelm our computers.

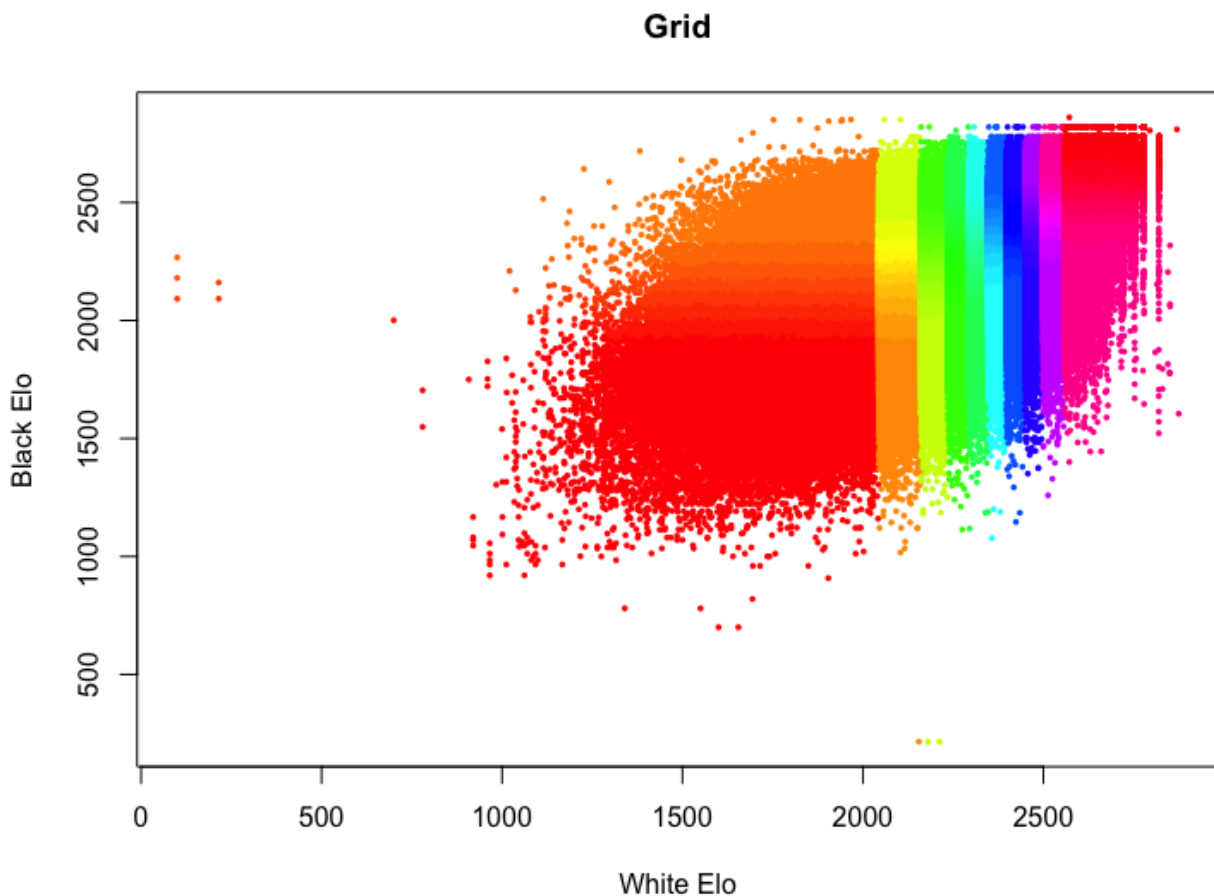


Figure 4.1: Plot of the grid showing mostly division with White ELO.

This helped us distribute the data in an equal way, in ten different ranges, with more or less equivalent number of games, for white, and for each of these ranges we created ten black ranges.

The white ranges contained around twenty thousand different games, and for each of the black ranges within each white range, it had around two thousand different games. We present the exact ranges in the appendix A.2.

To select the ranges, we condensed all the data into a csv file, that we then loaded to **R** with the `data.table` library to process large files. And then using a combination of `ggplot2`'s `cut_number` to cut the whites in 10 roughly equivalent sets, we then proceeded to cut the rest of the data.

Once we had the ranges we processed the data in python. We selected a path where using a UNIX command we created a hundred folders, one for each of the grid. Then for every file we had, we would check every game's white and black average ELO, and then assign them we would write in each of these folders write a file corresponding to the file that was processed. We did it this way to avoid having synchronization errors, by creating an output file for every processed file we would avoid writing twice by two different processes in the same file.

```
for num in {0..99};  
do mkdir $num;  
done
```

But comparing every game with 10 ranges for white and then compare the black to other 10 different black ELOS to find the correct position for all 2 million games, would be extremely demanding computing wise.

To avoid this problem we used an old programming technique, interchanging computing demand for memory demand. We created a function that returned the column of the `white` it corresponded and, with thus number we then had each of this dictionaries pointed to the corresponding black dictionaries, with this we obtained the proper grid number without making many comparisons.

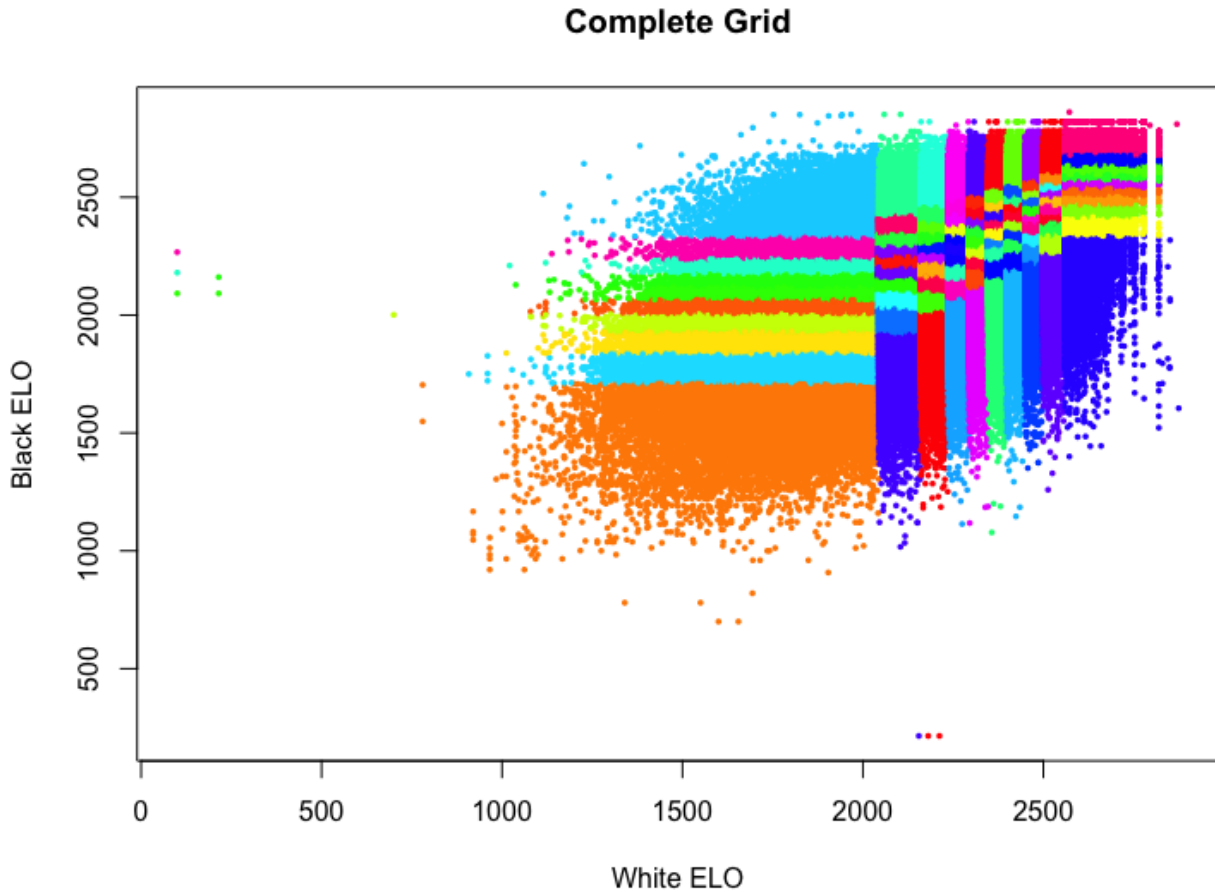


Figure 4.2: The grid with Black and White Subdivisions.

Train/Test dataset creation.

After we had 100 folders that represented the grid number, each containing files to be merged into one, to merge them we used a UNIX command, once we had done this we proceeded to separate the data into two different datasets. To do this we used an uniform random generator function to select whether a game would go to the test or train dataset, we decided to have 70% of the games stay in the training dataset and the rest went to the test dataset.

At the end we had one folder for testing and another for training.

4.3 Social Network Analysis.

The topology of social networks have been the subject of profound study in recent history. It has been identified that networks such as these play an important role in many systems, and examples range from computer networks, the propagation of information in social networks etc.

4.3.1 Community detection.

Social networks have been found to have community structures if the vertices of the network can be grouped into sets of vertices, this means that a community structure set of vertices will be densely interconnected internally, and sparser connection between vertices of different groups will be made.

4.3.2 Walktrap community detection.

This measure tries to find densely connected subgraphs, also called communities in a graph via random walks. The idea is that short random walks tend to stay in the same community. This is reflected figure 4.4 we can see that the Grand Masters play each other regularly, and thus they know each others style of play.

To do social network analysis we used igraph [8] to read our data and then obtain some basic statistics of our graphs at hand. Which is very simple to use even though its documentation is hard to digest, and it's a beta API.

Our main goal of doing social network analysis, was to obtain information about the players, what's the difference between the grid 0 where the ELO of both white and black are low, or the grid 99 where the ELO of both white and black player is the highest.

We started by performing a summary of the graphs for the rest of the grid, to see if we could find something interesting in any of the rest of the grid.

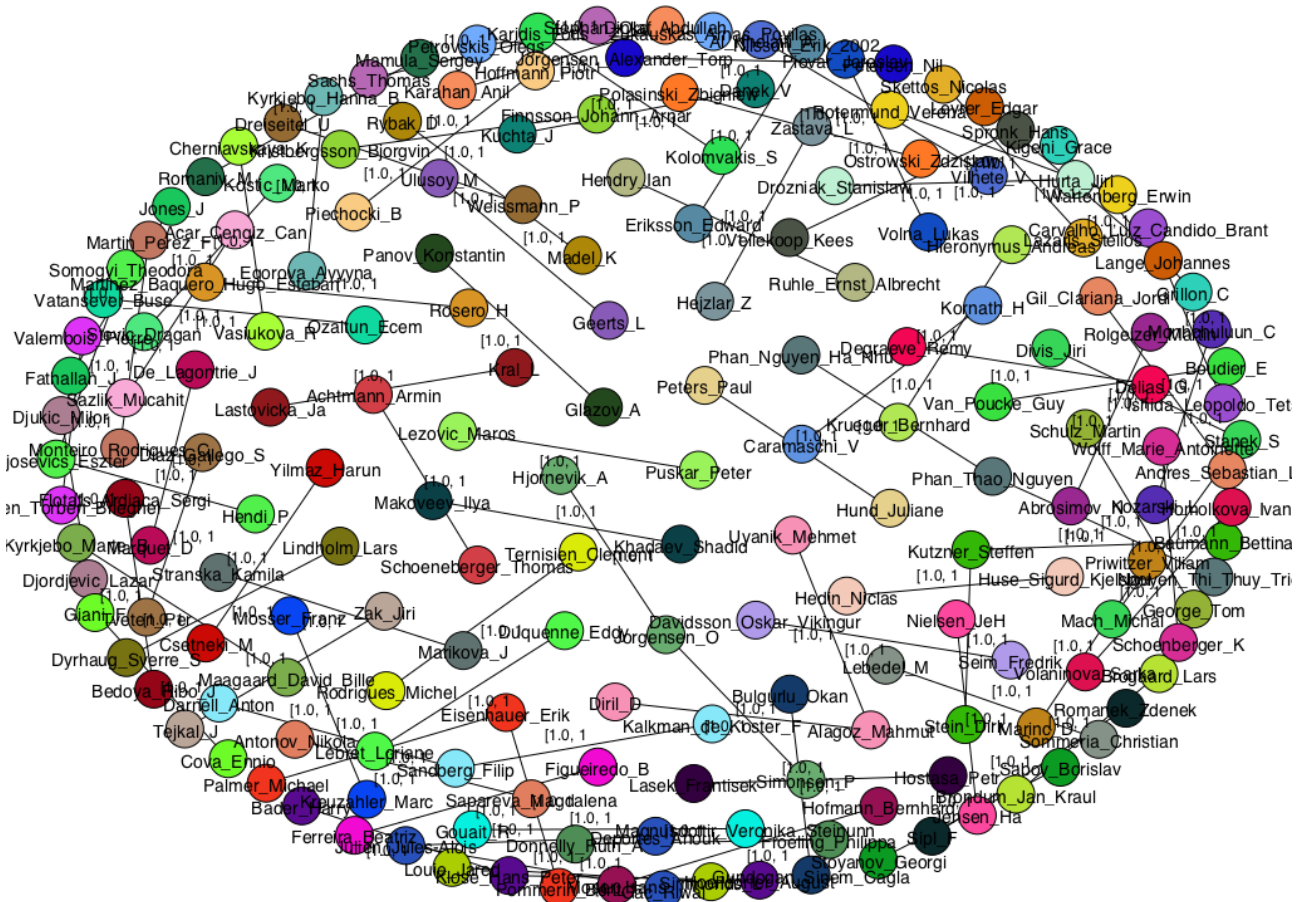


Figure 4.3: Community walktrap grid zero, second try.

We defined the following terms:

- **N**: number of vertices(also called nodes or points).
- **E**: number of edges(also called arcs or lines).
- k : mean degree, defined as: $k = E/N$.
- δ : Network density of edges. Defined as: $\delta = \frac{2E}{N(N+1)}$.

Each of these basic statistics will help us determine which dataset in the elements would respond more favourably to an analysis made based on the players.

We present here a small version of our table of properties we generated, you can see it whole in appendix 4.1, we notice that for example the number of **N**(vertices) and **E**(edges) are two to

one, which is close to a completely random graph, we believe this is due to it having the lowest ranked players, in the events registered by the database, which are usually not constantly in tournaments that are much more players like that.

#	N	E	k	δ
0	12060	18274	1.51	0.00025
1	13371	17884	1.33	0.00020
2	13504	17308	1.28	0.00018
86	853	15763	18.47	0.04327
87	765	14866	19.43	0.05073
88	985	15558	15.79	0.03203
89	971	14174	14.59	0.03003
90	9834	15711	1.59	0.00032
91	3952	15607	3.94	0.00199
92	1881	15121	8.03	0.00854
93	1216	15198	12.49	0.02053
94	867	14995	17.29	0.03985
95	663	14740	22.23	0.06696
96	463	14987	32.36	0.13952
97	464	14640	31.55	0.13570
98	448	14484	32.33	0.14401
99	450	14263	31.69	0.14055

Table 4.1: Summary of network properties for selected grid elements.

If we look at the last elements of the grid we can see that both the k and δ , have increased significantly, and this is the case specially the case where the highest values for white and black are represented. This makes us think that having information of the player might help us better to predict any result in this part of the grid.

We then continued to reduce it as we suspected that the data of this particular dataset, as figure 4.3 shows, it has a very low mean degree, meaning than most of the vertices had only one edge connected to it, thus rendering any analysis of competitors very hard to subtract any conclusion or heuristic out of graphs such as these.

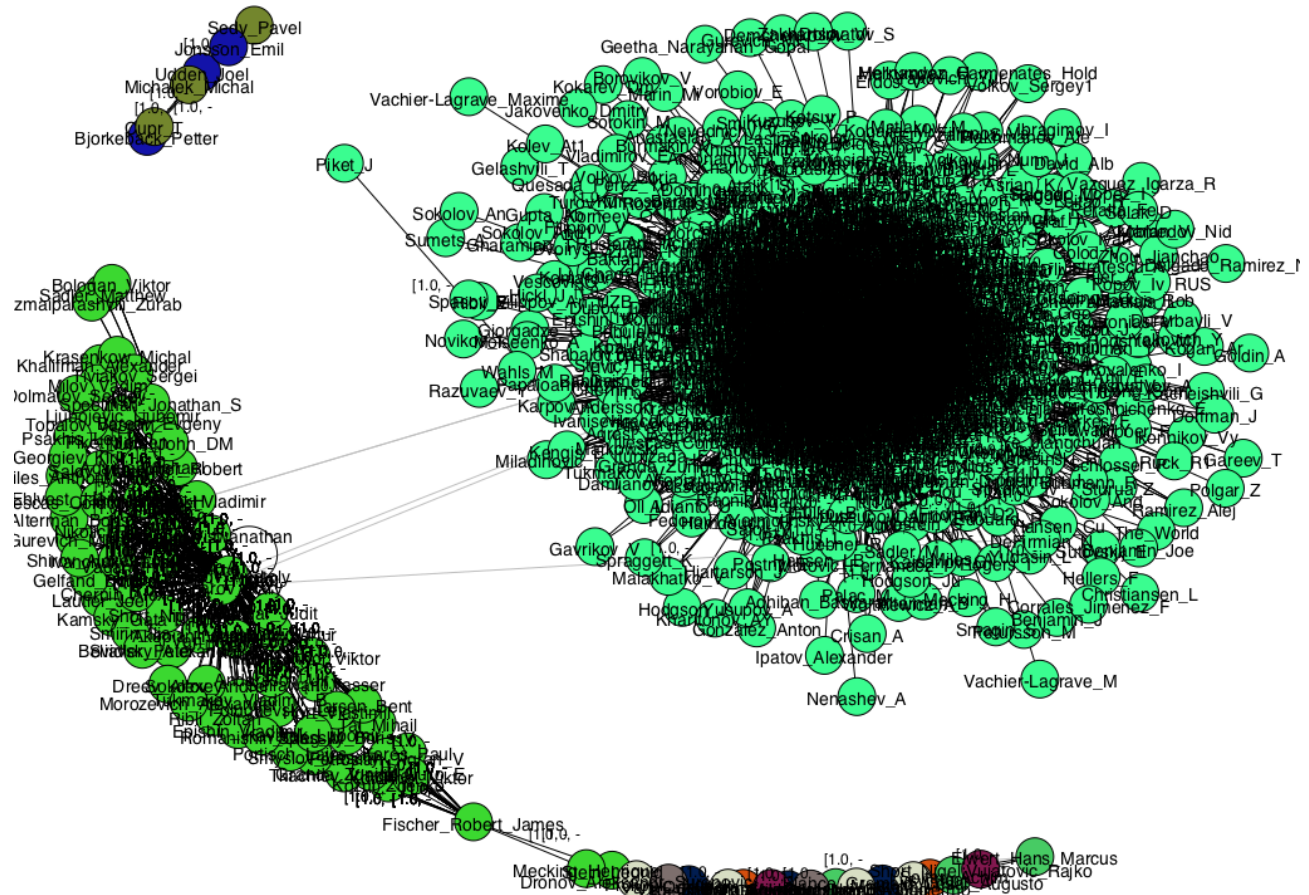


Figure 4.4: Community walktrap grid 99.

Here we can see that there is more community, and so we checked what was the percentage of winning for some of the participants, which for the most part are grand masters, at least in this level and close to these levels. We think that we might be able to use a sort of history between competitors in these levels to successfully predict a result.

4.4 Summary Statistics

Using the data and the classes from the last two chapters we were able to generate some basic statistics, as shown in appendix A.1, we created a few summary statistics for our grid (explained in Chapter 4).

Table 4.2, shows some basic statistics of the game in which we figured

#	% W	% B	% Draw	μ	median
0	0.60	0.21	0.18	39.64	38
1	0.46	0.30	0.23	40.28	38
13	0.33	0.33	0.33	40.93	40
14	0.27	0.39	0.33	41.13	40
15	0.22	0.45	0.32	41.54	40
32	0.48	0.20	0.31	41.48	40
33	0.40	0.24	0.35	41.32	40
34	0.33	0.29	0.37	41.16	40
76	0.28	0.19	0.52	38.73	38
77	0.22	0.22	0.54	38.41	38
78	0.18	0.27	0.54	38.64	38
79	0.14	0.35	0.50	40.78	40
95	0.36	0.12	0.50	41.78	40
96	0.31	0.13	0.54	41.16	40
97	0.29	0.15	0.54	41.81	41
98	0.26	0.18	0.54	42.38	41
99	0.23	0.23	0.53	43.23	41

Table 4.2: Summary statistics(short).

We also generated a couple of plots to compare how the difference in ELO, would change in the result. Figure 4.5, shows the box plot of all matches according to the absolute value of the difference in average ELO of players, to the left we see those matches where the ELO difference was lower to two hundred, and to the right those whose difference was bigger than four hundred.

If we look from the left we would see that our, the biggest the negative difference is (meaning black player has highest ELO.), we get the highest likelihood of black players winning, and the same is true for the white player. Looking at the second box plot in figure 4.5, we see that the same holds true, and we find that a higher proportion of games goes undecided, after many moves, this is probably because of the fifty movement rule we explained in Chapter 1.

Table 4.3 shows what we were suspecting, a lot of games that are decided either too early or too late, taking forty movements as the expected number of games of a match, go on to draw.

Next chapter we will continue to look into the dataset and add a few new data we can use to get more knowledge of our dataset.

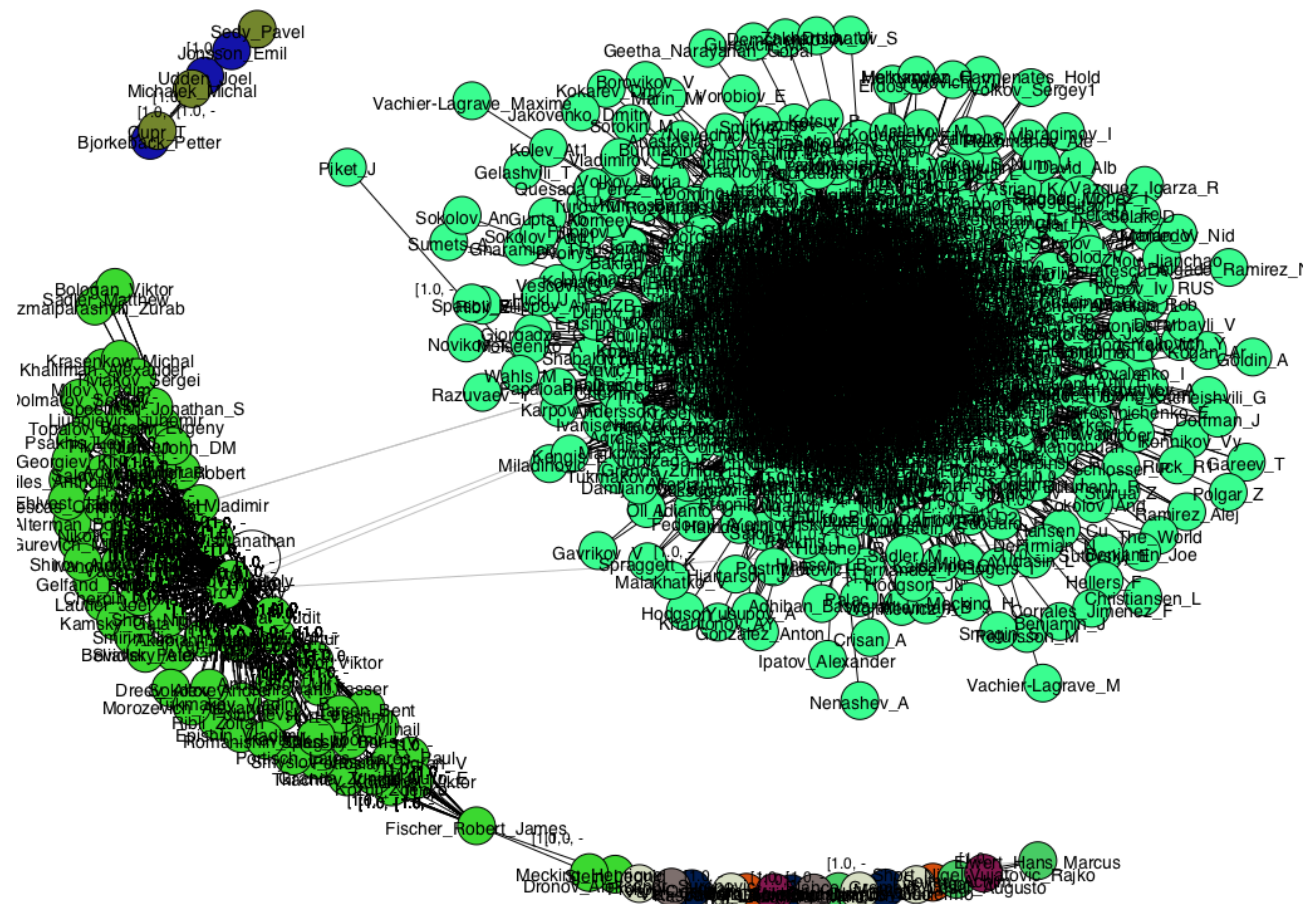


Figure 4.5: Box plot of our figures.

	n	black	draw	white
<20	199748	6.51	80.67	12.82
30	370939	23.85	36.86	39.29
40	528266	32.77	23.30	43.94
60	707847	32.56	28.89	38.55
>60	235781	30.05	36.95	33.01

Table 4.3: Result percentage

Chapter 5

Data Consolidation

5.1 Nodes and movements

Now that we had somehow refined the dataset to a level we desired, we continued then to process a little bit all the data into something we thought would get the most of the sheer size of the dataset. To this end we started to think how best to gather all the data into one structure where we could manipulate and access easily compared to what we had in the beginning of the project. We spent a good amount of time during this phase of this project because we tested the code so that we didn't have any bugs, while counting or doing operations with the node, and movement structure, more details in the following subsections.

5.1.1 Nodes.

We started by creating a **node** class that would help us to save and record all the movements that we register as we moved through the database. We created it as a *tree* data structure, in which we would save it's previous and proceeding nodes by it's id, which in our case it was the complete board. In this case and for explaining purposes is better to remind that node is equivalent to board position, since we were creating a new node for a new board position, and if not we would add the relevant variables to the already existent node.

5.1.2 EndNode Subclass

We had two separate types of nodes, which was the *root node* and the *end node*, which as indicated by the names contained the initial node with the start of all standard chess games, and the end node was saved with the name *END*, to which all nodes pointed to. As figure 5.1 shows we created a subclass of Node, to handle the *root node* and *end node*

As mentioned in section 3.3.1 we overloaded the `__str__` and the `__ref__` methods to debug our methods and the collection of data, we used them mostly to show statistics that we were interested in knowing and debugging the collection of data.

A great advantage of this is that all the methods and sub methods were present in the subclass, so we only implemented a few methods that we considered special for this objects, however it wasn't that useful because of the type safety in

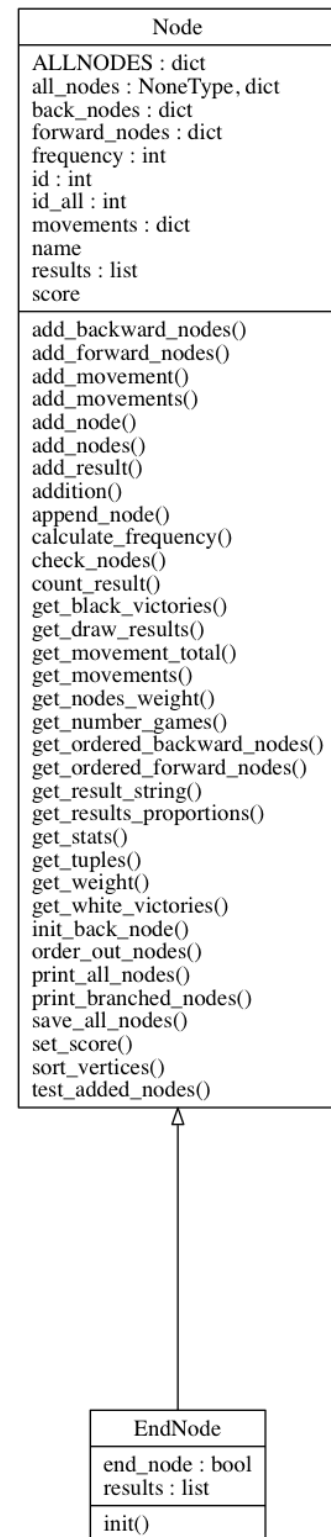


Figure 5.1: Node class.

Python is not enforced, however we took advantage of this structural change.

Class Variables.

Also called static variables, we created two class variables that would be treated as pointers to which all the objects currently collected would point to, we had to do this in order to both, have a method of readily access and add new nodes coming, and also to avoid having the garbage collector delete all the data that was generated. This method was

ALLNODES: We defined this variable because sometimes we had a hard times not losing data when changing of scope, so this would be available at all times when we needed it to be. We created reflection of this data because we were trying to serialize the objects we created using `cpickle`.

ID_STATIC: We thought of using an ID to replace our usual identifier, because it was creating a lot of memory problems due to it's size, since each of the grid members had around 200,000 unique board(or FEN) positions, so we thought we might be able to use this with a conversion table. However we couldn't use it because of the sheer size of the dataset, we also tried to hash the FEN string to no avail.

Object Variables.

Also called object properties, we created a few variables that helped us follow up with the relevant data that we were interested in.

Back and forward nodes: These two variables are a basic part of a tree dataset, these are dictionaries that contain a list each, at first we had the number of times that such a node we

had encounter, and also the object of the back/forward node, however we had to change it to be a simple dictionary that, for any FEN(board) we had connected, the dictionary returned the number of times it had been encountered. These two variables are handled separately.

Results: We created a list that saved the result of the matches that passed through any node, this would help us determine the percentage of white victories, draws or black victories.

Movements: This is also a list that contains the number of movement this a particular node appeared.

Other variables.

- **name:** Name of the node, which is equivalent to the FEN of the board.
- **Id:** Unique identifier of this node, the number is determined by order of appearance.
- **score:** The evaluation of the board, from `fen_eval` in our json database.

5.1.3 Movements

Movement
ALL_MOVEMENTS : dict all_movements : dict last_movement n_black : int n_draw : int n_white : int next_movement nodes : dict number_str : str total_movements : int
add_node() add_result() add_results() freq_freq() get_number_nodes() get_ordered_backward_nodes() get_results_string() init_last() init_node() init_all_possible_movements() save_all_movements() set_last_movement() set_next_movement() set_total_movements()

Figure 5.2: Movement class.

We created the movements because we thought this approach might also be useful to us, the movements will represent what the data had, at the beginning we tried processing all the data, meaning we would capture the half-movements(when a player moved), but then we decided to erase these half movements to save processing time and storage space. So at the end we end up representing the complete moves.

We based the architecture of the class in that of the nodes, because it would be easier to us to maintain the two somewhat related classes.

Class variables.

ALL_Movements: Like in the node class we created this variable to avoid having trouble with the scope of the variables.

Object variables.

number_str: We used the number as the identifier of the movement, we had relatively low number of movements, because as pointed out by [16], and as shown in our appendix A.1, on average the duration of a game is around 40 movements, but there were some other matches that were substantially longer, with the longest being around 160 movements.

nodes: This was only a dictionary that saved the FEN as the key, and the number of times that we had encountered this board position in the same movement as value.

Rest of variables: The rest of the variables we used them to count the results for every node, that passed through this node, this variables would later be used for our prediction function.

Unused implementations.

For these two classes we dedicated a lot of effort into trying to make them work correctly, at first we were working with the original thousand files. Since we would be working with these files separately, what we did was to process the files, and then save a pickle of the data.

However we after testing with the processed data, we noticed that it was noticeably slower loading pickles from python, compared with processing the files using, the nodes and movements classes.

With this came, several ways to merge our libraries, that ended up not being used because our computer would crash everytime we tried to load more than a few files at the same time.

5.1.4 Creating the graphs.

To collect and create the graphs, we used the other file, in the folder, the class files don't implement anything, and everything is done with both, `process_data.py`, which contains the methods that select the files to be processed, remember that we're using multiprocessing to reduce computing time. The file that coordinates and computes the graph formation is the `stats.py`, it also contains other methods.

5.2 Personal computer hardware limitations.

We developed this project in OSX, we had just recently reinstalled everything, because we changed our main drive, from mechanical technology(HDD), for a newer electronic technology(SSD), this was because I didn't want to buy a new laptop, and I figured I would be better off just buying a new hard disk, and more memory, we also expanded RAM to sixteen gigabytes.

And this was very important, as we have seen in data base management, and data warehousing when designing a query one must be very careful as to maintain the principal queries, and their answers in memory, because every time we try to access gigabytes of data that is in magnetic memory, even the time the needle it takes to read from the disk counts, this is relevant because if we remember how operating systems work, once we have ran out of temporal memory, the OS starts using the main memory, to cache some processes that are in the pipeline, and even though we had a flash like memory connected to this cache, it was impossible to process all the data we had.

Thus we decided to try to create a database to handle all of our board data, it was much simpler than we thought and this was a really good decision in the long run.

5.3 Board aggregation.

5.3.1 Our simple database.

The database we used was Sqlite [17], is a very simplified version of a database, in which you don't need to start a server(or at least codify it), and that it's implementation in Python makes it as easy as opening a file, but we still need to query the database.

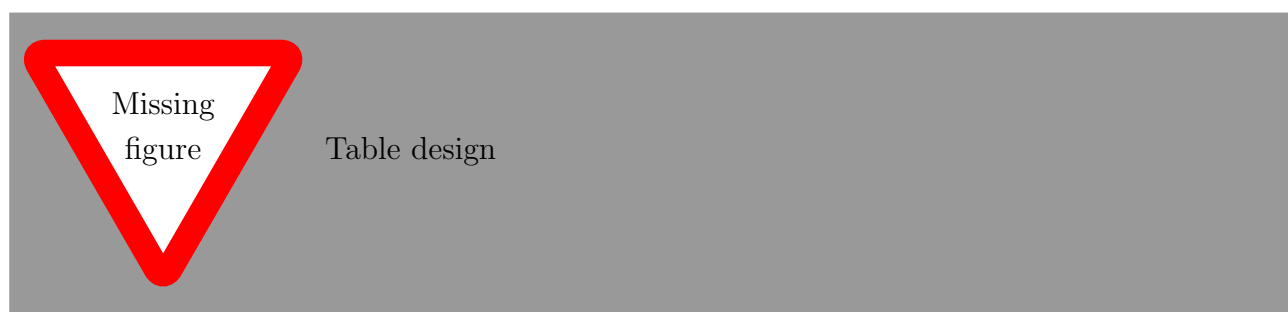
Sqlite is widely used in smartphone applications, it's based on SQL, and it is also well supported by the documentation. Something very important, it has most of the well known properties and functions of SQL, but not all, since it's a compact version of SQL, however as we will see we were able to take advantage of what was available.

The library in Python for Sqlite, is `Sqlite3`, and it's simple to use. To use Sqlite, we relied on what we had built before, in this step we used particularly the nodes graph, in which we recorded the number of results of any board position we had encounter, however the problem we had is that we couldn't sum them all into one big access, and since we had to "merge" all the data from 100 files, we had to somehow do something to handle collisions, where a board was present in more than one file, to do this, we let the database handle it by using triggers.

5.3.2 Database design.

For our database, we didn't use anything very complex, we know that databases can be exported from a class structure into a table structure, but this would make our project even more complex, so we decided to only have a database that contained one table, which we used to store all the data relevant to the boards. And we only inserted the training data into our database, which will try to use, to gain knowledge for testing our functions.

We created the following table



As we see we create a table with 4 variables, and a problem we had at first is that our **PRIMARY KEY** was a text variable (we used the FEN from subsection 3.1.1 as ID), we knew that if we used a text instead of a string the indexing would be slower, and since inserting the data to the database was so slow, we thought that at the end we would not be able to use the database, because querying would take too long.

Thus we tried to convert our data to an int, because this would also allow us to use a faster index, because SQLITE treats integers primary keys as clustered indexes[18], and they are not implemented in SQLITE, then we would gain potentially more speed while querying. To convert the text to integers we used the hashing functions in python's `hashlib`, which allowed to convert, and compress our string while avoiding collisions. However this didn't affect much the performance of the database, the fact that we had our database file in our HDD made a much bigger impact, than the ID not being an integer.

Sql Triggers.

To insert new data to the database, we had to have a way to check if the ID we were inserting was be new, and if it was not new we should add the fields of the existing register to the new one. To do this efficiently was in our best interests, because if we didn't it would potentially be exponentially more time consuming.

To do this, we added an sql trigger in our database, which Sqlite accepts, even if it is a simplified sql engine. An sql trigger is a database object that is *attached* to a table, and for practical purposes is similar to a stored procedure.

A trigger is only fired when a clause of *INSERT*, *UPDATE* or *DELETE* occurs, then we have to specify what to do when the trigger is activated. We then created a trigger, shown in figure 5.1, that everytime it found that a query was trying to insert an existing ID, we would instead update the table by adding the new values for white, draw, black.

```
create table BOARD(ID text PRIMARY KEY, white INT, black INT, DRAW INT);
CREATE TRIGGER t2 BEFORE INSERT ON BOARD WHEN NEW.ID IN (SELECT ID FROM BOARD) BEGIN

    UPDATE BOARD SET black=COALESCE(NEW.black, 0)+
        COALESCE((SELECT black FROM BOARD WHERE ID=NEW.ID), 0) WHERE ID=NEW.ID;
    UPDATE BOARD SET white=COALESCE(NEW.white, 0)+
        COALESCE((SELECT white FROM BOARD WHERE ID=NEW.ID), 0) WHERE ID=NEW.ID;
    UPDATE BOARD SET draw=COALESCE(NEW.draw, 0)+
        COALESCE((SELECT draw FROM BOARD WHERE ID=NEW.ID), 0) WHERE ID=NEW.ID;

    SELECT RAISE(IGNORE); -- Ignore INSERT
END;
```

Code Listing 5.1: CreateTable.sql

This small script enabled us to finally merge all the data about the board positions we had, and to do this, our little script took about 50 hours of processing for the sheer amount of data, and maybe because we used a text as **PRIMARY KEY**, as stated in section 5.3.2, an integer would have make this process faster.

With this we were able to accurately and simply consolidate the board steps, into a single database, at the end our database occupies, around 2.48 gigabytes of memory space. However

if we would have done this without triggers, we probably would have needed to query the database to ask for every movement in every game, we would query the database if it existed, if it did we would create an update query, and if it didn't we would just insert the data. This change would have made this task several orders of magnitude more time consuming than it was.

At the end we had recorded the whole dataset of two million games, see figure 5.2, and we had more than 13 million board positions registered, with this we were able to query for positions and see if we had any probabilities, we suspect this will probably be more useful in the higher end of the grid, not in the lower end, because experienced players tend to use some known moves, particularly in the early stages of the game.

```
1 sqlite> select * from boards where ID="END" ;  
END|596285|551520|457222  
3 sqlite> select count(*) from boards ;  
13890970
```

Code Listing 5.2: Queries to our db.

After a few tests, we moved the database, from our local HDD, to our local SSD, which greatly reduced querying time. We were generating around 35000 queries to our database, to process a file in the grid, and using the SSD, by the time it had finished the HDD had only processed around 1000, we then realized that if we had processed the database in our local SSD instead of our HDD, we would have saved quite a bit of time.

5.3.3 Querying the database.

The purpose of having the database is that we would use the data we had about certain board positions, to try to predict the result at the 20th movement, remembering that most games lasted only 40 movements [16] and this was also the case for our dataset as well, as we discussed in section 4.2.

At first we started testing the database, we were querying what was the 20th movement in our dataset to see if we had some board position on our dataset, but for most of the cases (96%),

we didn't have anything, so we decided to try to expand this concept to the last 5 movements, and then sum anything we had found, but still we ended up with around 60% of missing data.

Since we were interested in knowing if having a historical record of our board would help us predict results in the testing dataset, we devised a way to make use of our database. We started by saving the historical record of the last movement of each game, in a file that we would then use to build our models, however, if we remember from Shanon [16], there are more than 10^{43} , possible board positions. This would make it impossible to find any matches, but since we're not using the last movements, we think we had something of a chance to actually find some matches.

We made our very simple query.

```
SELECT white, draw, black FROM BOARDS WHERE ID="rnbqkbnr/pppppp.pp/...p../...
.../.../.../PPPPPPP/RNBQK"
```

Code Listing 5.3: ID selection.

But since we didn't find many matches we had to come up with another strategy.

Final query

Since our original plan, was to use the last movements of the match we were analysing, and their probability, we needed to come up with a solution to show the tendency in the game, we would like to know the historical results of the last movements in the current game compared to our database. If any movement existed we added to the other latest moves, and with this we would construct a set of chances that other games have taken in the past.

We think this would be specially useful the grand master level, as we saw in 4.1 here we are likely to find the same kind of players, and thus the same kind of moves and openings, such as the Sicilian defense, and the Catalan opening, we think that because of this it is likely that we find some of this movesets, or their variations in our dataset.

To try to expand on our move probability we decided to make the query a little bit more complex, we would look further into the game to try to come up with, a board position that exists in our dataset (and hopefully one that has many entries), we made the query look for the last 7 moves, to try to find data in our small database, and to make sure we don't add the first move in our probabilities we deleted it, otherwise it would have the 2 million games we recorded, and would nullify any data we find in the rest of the moves.

In sql we used the IN statement which allowed us to query for several IDs at the same time:

```
1 SELECT white, draw, black FROM BOARDS WHERE ID IN (id1, id2, id3 ...)
```

Code Listing 5.4: Multiple ID selection.

Doing this would affect the performance of the script, it would make it considerably slower to process, compared to the first version of our query, because we would be making many more request to our database per game. However thankfully we had the database, in our SSD disk, this simple fact probably saved us hours of waiting for the machine to handle all the queries.

Now that we were done with this problem we would continue to the next phase of our data consolidation. Even though we couldn't use the Movements class 5.2 as we had hoped for at first, we still were able to take advantage of the infrastructure we build to make it work using the database approach, which in the end was successful.

Since we were concerned about how likely is that any of the seven last movements of a game was in our database, we created a table B.1 that showed missing data, meaning that we didn't find any information about any of the last movements, or about the white and black player. An extract of this table is in table 5.1, notice that this table is showing us somewhat the same trend that we saw in the Network Analysis chapter, it seems that the last games, even though they are closer Elo-wise, we might be able to better predict them because of our other important historical data.

number	board	white	black
0	0.918662	0.015444	0.00669241
1	0.887497	0.0100237	0.00619889
27	0.675664	0.000146692	0.00440076
28	0.654855	0.000150784	0.00361882
29	0.693585	0	0.00898979
97	0.471272	0	0.000484183
98	0.48156	0.000491723	0
99	0.455161	0.000165153	0

Table 5.1: Data consolidation missing values (short)

5.4 Consolidating Competitors.

As we saw in section 4.3, we realized that we might be able to use the competitors results history, to try to predict some of the matches, specially in the later grid elements.

We used the Competitors class, to create a sort of database, that contained info relevant to us about the competitors. We saved their results for all the data in the training dataset, that we then used to obtain the number of games, won, drawn and lost for every competitor we saw in our database.

To process the competitors we chose to store the data of the competitors in a json file, we called this file, the Competitors database. To load the data from this database and make use of it, we re-used the Competitors class, we talked about in section 3.3.3 adding a constructor and a few methods we were able to use the class to serve as a query connection.

The `init_prob.results` constructor to initialize the database, shown in 3.1, it initialized the Competitor's name, number of wins, number of draws, and number of losses, at first we were using percentages, but we changed the implementation to have only counters, as it would be easy to have percentages if needed later in the prediction phase.

Then we used a `COMPETITORS` global variable, which was a dictionary that paired the names of competitors with the Competitor object of the corresponding player, we also added a method `get_tuple.results`, to get a tuple of with the `white`, `draw`, `black`, historical results for the competitor.

Loading the dataset was not a big problem because we only had 96788 competitors, with the players with the most games registered, were almost three thousand games, with the majority of the competitors had very few registered games.

5.5 Final Dataset.

As seen in the previous chapter, we consolidated a final dataset, using everything we had processed thus far. We then used all the data to predict the matches after twenty complete moves(when both players have moved), to process this, we cut short our database to eliminate both half moves, and also we reduced the number of movements to twenty when possible. For matches that had shorter games, we left the whole game there.

```

1   2   3   4   5           6 7   8           9 10  11  12 13 14 15 16 17 18
-- -- -- -- --
0 -10           1907 Pustel J 0 1917 Franke R 0 0 0 2 4 3 3 13 1
0 0 2080 1972 1917 Gravett A 0 2029 Kruif K 6350 4577 1585 14 27 34 18 26 27
0 0 1979 1875 1893 Muler Wo 1 1930 Kell Al 0 0 0 1 1 2 2 1 5
-- -- -- -- --

```

Code Listing 5.5: Text data sample.

The header of the data is as follows in the same order:

1. **fen_eval**: The evaluation of the board, as we saw in section 3.2.1.
2. **sum_eval**: The sum of all the evaluations in the game so far.
3. **white_elo**: The registered ELO for the White player.
4. **black_elo**: The registered ELO for the Black player.
5. **black_avg_elo**: Average ELO of the Black player.
6. **black**: Black player's name.
7. **result**: Result of the game.

8. **white_avg_elo**: Average ELO of the White player.
9. **white**: White player's name.
10. **fen_white**: As explained in section 5.3.3, the number of White player's victory result for the last 7 boards of this game.
11. **fen_draw**: Number of draws in the last 7 boards of this game.
12. **fen_black**: Number of Black player's victories, for the last 7 recorded board positions.
13. **white_win**: Number of victories of white player as explained in section 5.4.
14. **white_draw**: Number of draws for the White player.
15. **white_lose**: Number of loses for the White player.
16. **black_win**: Number of victories of Black player.
17. **black_draw**: Number of draws of Black player.
18. **black_lose**: Number of loses of Black player

Now that we have this dataset that reflectes all the properties we found to be useful, and that we described in the previous sections, we think that this would be enough to predict with a somewhat low error rate the result of a chess match after a few movements. In the next chapter we will process this data, to do this we switched to R, as we have been doing through the project to generate some plots and make other analysis. Because now the data was not that overwhelming we could do the analysis in R, as we had hoped to do.

Chapter 6

Analysis

During this chapter we will analyze the dataset we generated, in the last chapter and we explained in section 5.5, we will discuss the results the theory behind the methods to obtain these results.

We should note that two models are built, one in which we predict if the result of a match was white, draw or black. The other model we built to determine who of white or black won. Then we will compare the two results and discuss if having one less option would be relevant to the game, part of the logic behind this is that, within a tournament a draw gives both players $1/2$ a score, as we noted in section 2.2.2.

6.1 R

Since it's creation R [15] has been built to be statistics friendly, and since we already have experience with the analysis, and prediction of R, as well as practical programming skills to do other operations. We decided to continue the project using R, which is a much lean testing environment for statistical analysis than Python is, and it's specially good when we are not handling enormous datasets. Of course there are packages in Python such as SciKitLearn, Anaconda, numpy, and jupyter that works kinda like R/Rstudio, but we would have to learn to use it's functions and overall gain experience in using these libraries.

Since our effort to produce a lean and somewhat small dataset was successful, and we had reduced the dataset size significantly 1.1, we will be doing all this analysis in R, as well as some variable definitions from the data we had created in chapter 5

6.1.1 Multiprocessing with R

As mentioned in section 4.2 multiprocessing is very useful to multicore systems, and since we had more than a hundred files to process, keep in mind that the processing will be done independently, we could easily and efficiently take advantage of the multicore processor.

6.1.2 Lapply

Lapply is a very important function in R, it's a very efficient way to execute functions in R. It is used to apply a function over a List or Vector, it returns a list with the result in the same order of the original input object, and it also returns it in the same container.

To use *lapply* we need to look at it's formula first: `lapply(X, FUN, ...)`

As we can see `x` represents the input object and `FUN`, which is the function to be executed.

Most of the time faster to execute a function over a matrix in R than to use what a computer scientist would find more obvious, using a *for* statement, so this is a key thing to learn in R. We encourage the reader to use R's `vignette("lapply")` to read more on this subject.

6.1.3 The parallel library

We have said already that the multiprocessing package in python, specially its `pool.map()` function is very similar to the `apply` function in R. After looking into continue to take advantage of our multicore processor we looked into how to use the parallelization in R.

Thus we found the parallel library, which is part of the R core package [15], to do parallelization we used the `parLapply` function.

It's usage is very simple, and it's not very different to the previously explained *lapply* function, however it has it's particularities.

To use it we need a to create a cluster that will later be used to do the task, as in python's multiprocessing library, this will create a pool of workers which will actually be doing the job. In this case we used the maximum number of cores minus one because, on the first test we were unable to comfortably use our computer, so we reduced the number of cores minus one, which in our case would be 3.

Next we would start our cluster, with the desired number of workers, we added the fork option because it would allow us to use our main environment in the multicore environment, without this we would not be able to access methods that we declared, and were executing inside our function, this **Fork** option is only available to UNIX systems, however there's an option for windows.

```
1 # Get the number of cores
  ncores <- detectCores() - 1
3 # start cluster
  cluster <- makeCluster(ncores, type="FORK")
5 parLapply(cluster, X, FUN...)
```

Code Listing 6.1: parLapply example.

Then we would use the function as shown in code snippet 6.1, with the added requirement of the cluster we would be using.

With this we were able to accelerate the processing of all our elements in our grid which were somewhat computive intensive, but looking back the methods would execute much more faster than most of the things we did before.

6.2 Derived data

Before moving into what we used to model our problem we combined our data to create a few features, with the idea that this combination would reflect more the relationships of the

variables.

- **Delta Avg ELO:** The difference of white and black average ELO.
- **Percentage results:** We added a few new columns that represent the percentage of results for, *fen*, *white* and *black*.
- **Delta percentage:** For these variables in the case of the model which has no draws, we also added the difference of win/lose draw.

WE also created the following variables for the model in which we are predicting only wins or losses.

- **Delta White:** The difference between wins and losses for white player.
- **Delta Black:** Same as before, but for black.
- **Delta FEN:** Same, but for the board results.

With this we are hoping to capture the relationship these variables have between each other, and thus make them better predictors for the game.

6.3 Modeling the problem.

6.3.1 Kernels

Kernel methods are an effective alternative to feature extraction. A kernel function is a function returning the inner product between the mapped data points in a higher dimensional space.

The kernel function is given by the relation:

$$k(x, x') = \phi(x)^T \phi(x') \quad (6.1)$$

The kernel function is typically symmetric($k(x, x') = k(x', x)$) and non-negative($k(x, x') \geq 0$).

This means that only functions that can be transformed in such a way that the inner product is $x^T x'$ can be replaced by "kernelized". The learning then takes place in the feature space, provided the learning algorithm can be entirely rewritten so that the data points only appear inside dot products with other data points. The fact that Kernel methods do not depend on the dimensionality of the feature space, and that they can use any kernel function makes them a good candidate for different classification tasks.

6.3.2 RBF kernel

One of the most well known functions is the RBF kernel, also known as the Gaussian kernel, the radial basis function kernel is defined as:

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) \quad (6.2)$$

This is an example of a radial basis function because each basis functions depends only on the radial distance from a center, which in this case is the $\|x - x'\|$. We see in definition 6.2 the σ^2 is known as the bandwidth.

6.3.3 Support vector machines.

To apply the rbf kernel we need a sparse kernel machine, for this we will use a support vector machine. The SVM's are supervised methods that use the a learning algorithm that can be used for both classification and regression problems, the SVM will build a model that will assign new data into one category or another.

Consider the following function:

$$J(w, \sigma) = \sum_i 1^N L(y_i, \hat{y}_i) + \sigma \|w\|^2, \text{ Where: } \hat{y}_i = w^T x_i + w_0 \quad (6.3)$$

If L quadratic loss, then this function is equivalent to ridge regression, and we know that this function can be kernelized, and thus it has a solution in the form of $\hat{w} = (X^T X + \sigma I)^{-1} X^T y$.

However, if we replace the quadratic/ log-loss with some other loss function, to be explained below, we can ensure that the solution is sparse, so that predictions only depend on a subset of the training data, known as support vectors. This combination of the kernel trick plus a modified loss function is known as a support vector machine or SVM. This technique was originally designed for binary classification but can be extended to regression and classification.

6.3.4 Random Forest

Random forests are an ensemble classification method. The method produces a classification tree at each iteration. This Classification tree is built from a random subset of the data, and at each node in the tree a random subset of predictor variables are selected. Multiple trees are constructed in this fashion until at test time the classification of this individual trees are combined to form a final prediction. By constructing many trees and averaging many estimates we are able to reduce variance. This is why were interested in the Random Forest, because this property helps us reduce bias. We train M different trees on different subsets of the data, chosen randomly with replacement, and then compute the ensemble

$$f(x) = \sum_{m=1}^M \frac{1}{M} f_m(x) \quad (6.4)$$

where f_m is the m th tree. This technique is called bagging[4], which stands for bootstrap aggregating. However, if we simply re-run the same learning algorithm on different subsets of data, it can result in highly correlated predictors, which limits the amount of variance reduction that is possible. Which is why at each node in the tree a random subset of predictor variables are selected. This is what ultimately differentiates bagging from Random Forests.

We then vary the tree parameter of the random forest to try to find the best parameter for our given K , as we did before for SVN, then comparing the OOB(out of bag) error we choose the

best ntree value.

$$ntrees = (110, 200, 3981)$$

6.4 Building our model.

As discussed before, our challenge is to try to predict the results of the games after 20 movements, after going through a lot steps to finally reduce our dataset to a more easily computable level, while ridding ourselves from data that we didn't need, we think we have done a good job to get the most important data out of the 2 gigabyte original file.

Now we are going to try to produce a couple of results, one in which we have draws, and one that doesn't, this is because we are sure to increase precision and at the same time a draw affects less the overall match result than other result.

To accelerate the execution of the dataset, which remember that as we saw in Section 4.2.1 for each of the grid elements, we have around 14000 training games, and 7000 test games for each of the grid elements(see A.2). Because of this we used the *parallel* library in R, to accelerate a little bit the calculations because we had many operations to do.

6.4.1 SVM with RBF kernel

Our first model consists of a SVM, with a RBF kernel, to tune this we needed to pay attention to two parameter, one for each of these functions, the C parameter which is the complexity parameter, which represents the trade off between the training error and margin width. We also have to calibrate the bandwidth parameter (σ) of the RBF kernel function.

To tune σ we are using kernlab's [11] sigest (automatic sigma estimation) in the `ksvm()` function. In the case of the C parameter we are going to generate a small list of option to calculate a model with a different C each time. We also are using 3-fold cross validation because, as we stated before we have too much data to use the standard ten-fold cross validation.

```

1 get_best_model <- function(trainL,c){
  err = 0
3   model = list()
  modelList = list()
5   for(i in 1:length(c)){
     print(i) # print the number of
7     model <- ksvm(y=trainL$y,x=trainL$x,C=c[i],cross=3)
     modelList <- list(modelList, list(model))
9     err[i]<-cross(model)
  }
11  n = which.min(err)
  model = unlist(modelList)[n]
13  return(model[[1]])
}

```

Code Listing 6.2: R function to tune ksvm parameters.

As we saw in section 6.3.1, kernels are memory based methods, that use their own data to better fit our model, and this is why we need to tune the parameters to obtain a better prediction.

In code snippet6.2 we are creating a series of models each with a different C, each model created we add it to a list, so that we don't have to create the model from zero. Then we select the model with the lowest training error, and use this model then to predict our results.

This is an example of a model function 6.2 would return.

Support Vector Machine object of class "ksvm"

SV type: C-svc (classification)

parameter : cost C = 0.01

Gaussian Radial Basis kernel function.

Hyperparameter : sigma = 0.419494047330239

Number of Support Vectors : 174

Objective Function Value : -1.3194

Training error : 0.159036

```

get_best_rf_model <- function(trainL, trees){
2   tab = table(trainL$y)
    num = round(tab[which.min(tab)]*.3)
4   re = length(table(trainL$y))
    samp = rep(num, re) # we will have the same number of
6   names(samp) = c("-1", "1")
    modelList = list()
8   model = NULL
    err3=0
10  for(i in 1:length(trees)){
        print(i)
12    rf <- randomForest(y=trainL$y, x=trainL$x, ntree=trees[i], proximity=
        FALSE, strata=trainL$y, sampsize = samp)
        modelList <- c(modelList, list(rf))
14    err3[i]<- rf$err.rate[trees[i],1]
    }
16  n = which.min(err3)
    model = modelList[n]
18  return(model[[1]])
}

```

Code Listing 6.3: R function to random forest parameters.

Cross validation error : 0.158986

We see that the parameters we just talked about are present in this summary. This model returned a C of 0.01, and the σ parameter was set to 0.41.

6.4.2 Random forest model.

For our random forest models, we were using a tuning variable that we later used to compare ODB error, and then select the best model we found to then predict the results of the test dataset.

If we take a closer look into what our code snippet 6.3, we see that we are selecting the model with the lowest out of bag(OOB) error, and then returning this method. Also we are using the *sampsize* parameter, to potentially reduce the number of operations, by reducing the data sample.

Random forests have the property of being able to use down sampling without losing data, since this is a tree ensemble method. A large number of bootstrap data is taken from the

training sample, and a separate unpruned tree is created for each dataset, and at each step of the creation of each split of the tree, which selects randomly a subset of predictors to promote diversity. When predicting a prediction is made by every tree which is then added up to produce a prediction for the sample.

Before going ahead of this we tested that this was the case with one of our grid elements, we made the prediction of win/loss for a dataset with the whole data and one with the sampled dataset. Then we compared the ROC of both models, as figure 6.1.

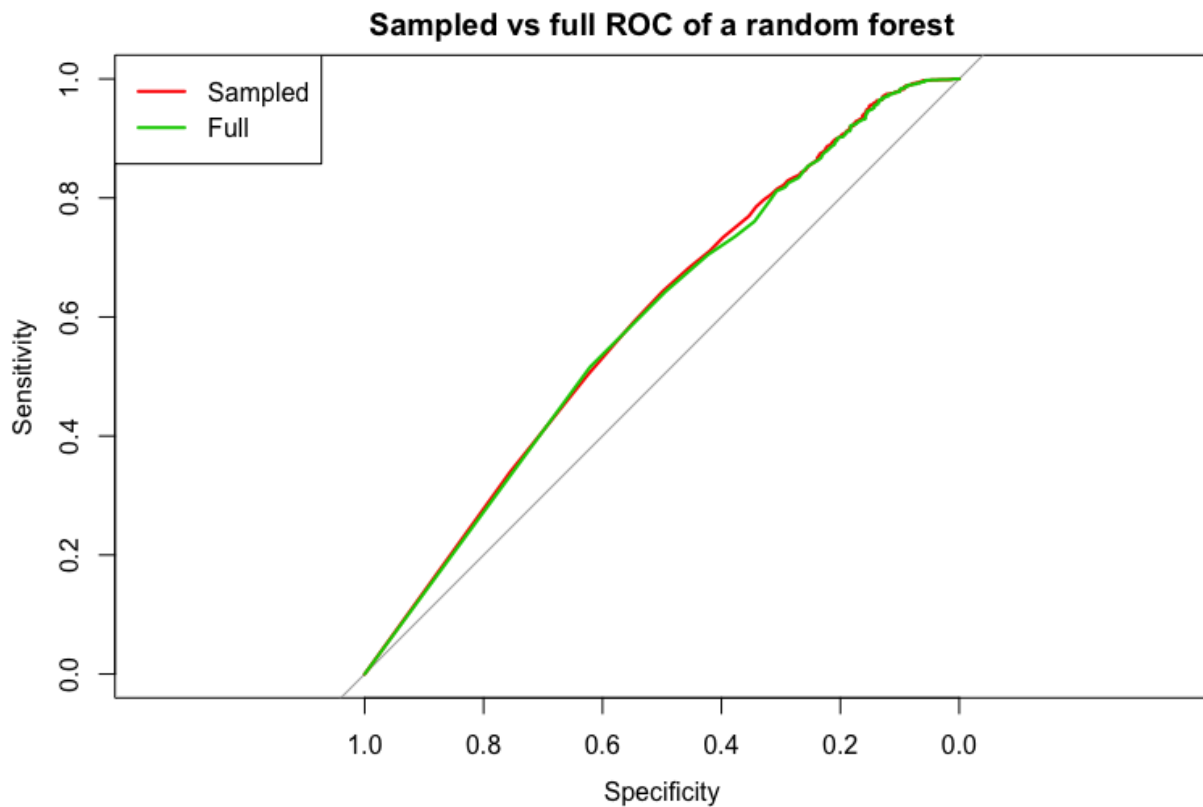


Figure 6.1: Random forest ROC comparison.

For both of these files we got an area under the curve of 0.5945 for the sampled model, and 0.5912 for the model which used all the data. We then were sure to continue using the reduced dataset, and we used for both types of models our code shown in code snippet 6.3.

6.5 Results

With the effort we put on gathering our dataset, we were able to train a reasonably good model, at least with SVMs, which, in turn helped us produce reasonably good results, even if as a result of having to devote ourselves more into building the dataset than creating our model. Our results are shown complete in appendix C, for both our models including draws C.2, and ignoring draws C.1.

We would expect the results with the highest difference in ELO to actually show the better prediction capabilities, since the best error are either on the extremes of the grid, if we remember Figure 4.2, our grid positions were determined by the average ELO. This means that at the beginning and at the end of the white range, are the highest differences in ELO, whether it's in favor of the white player for the first black player range, or in favor of the black in the last of the black ranges for a given white range. In graphical terms, the lower part of the graph will favor white, and the higher will favor the black player.

Overall our results of the SVM are better than the random forest, and even the results of the random forest, and even that is not terribly bad. We think that this difference in performance is by design in their algorithms, we can see this clearly if we look at the first 3 columns of both table 6.1 and table 6.2, we see that the SVM was selecting completely one of the variables or the other. And as discussed in section 6.4.2, this is due to the fact that the Random forest in each tree iteration it has a probability to select one of the predictors, to improve it's diversity, and this we think is the reason why it didn't go all the time for one answer or the other.

On the other hand our predictions for the model that includes draw was pretty good, we think that this is because there is one more possible answer, and thus if we are already distributing the predictions into a new factor to predict, we might going to be better, this is mostly due to the fact that in the dataset, most of the chance to win are very skewed one way or the other, depending on the grid position due to the fact that there are times where there is a big difference in ELO averages.

	error	Black	White	error	Black	Draw	White
9	0.13	0.00	1.00	0.31	0.08	0.84	0.93
29	0.16	0.00	1.00	0.38	0.15	0.69	0.89
30	0.09	0.00	1.00	0.30	0.10	0.81	0.87
31	0.11	1.00	0.00	0.29	0.92	0.84	0.09
57	0.43	0.15	0.79	0.50	0.85	0.10	0.87
72	0.11	1.00	0.00	0.34	0.89	0.83	0.10
74	0.19	1.00	0.00	0.47	0.89	0.42	0.42
84	0.18	1.00	0.00	0.46	0.90	0.45	0.37
avg	0.263	.	.	0.43	.	.	.

Table 6.1: Results of the SVM prediction for selected grid

	error	Black	White	error	Black	Draw	White
25	0.44	0.04	0.89	0.56	0.47	0.65	0.55
17	0.25	0.02	0.93	0.49	0.34	0.52	0.89
48	0.29	0.02	0.89	0.54	0.66	0.36	0.70
49	0.21	0.02	0.89	0.51	0.53	0.35	0.84
75	0.62	0.09	0.80	0.48	0.86	0.43	0.40
85	0.64	0.13	0.79	0.50	0.86	0.33	0.56
88	0.51	0.12	0.79	0.43	0.82	0.15	0.80
99	0.49	0.09	0.76	0.46	0.69	0.28	0.65
avg	0.491	.	.	0.48	.	.	.

Table 6.2: Results of the Random forest prediction for selected grid

If what we think about, that grid position affect the result, if we plot our data in sequence of 10, each of them representing a grid range for the white player and the X axis, represents the grid of the black range for each white range, we will see more or less an arc at least in the middle of the ranges, from grid number 30 to 60, or maybe a wider range.

As we can see, this is indeed the case for most of our models. In the case of the draw less SVM model, we get very good results, however this is due to the fact that, after eliminating the draws from our possible results we have around 80% of results going to one side or the other.

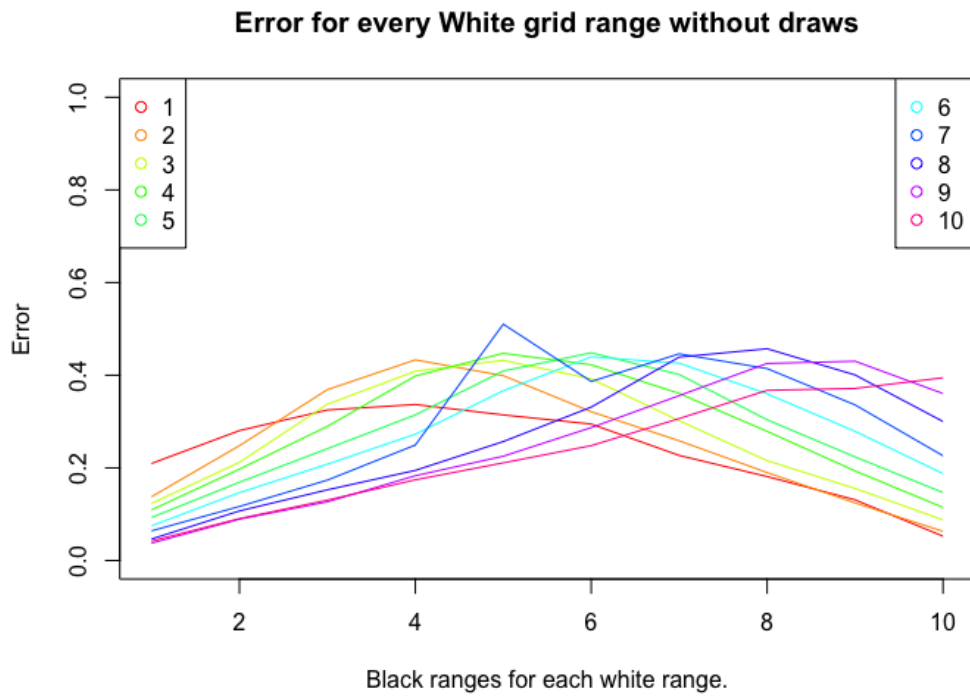


Figure 6.2: Drawless Results of the SVM model.

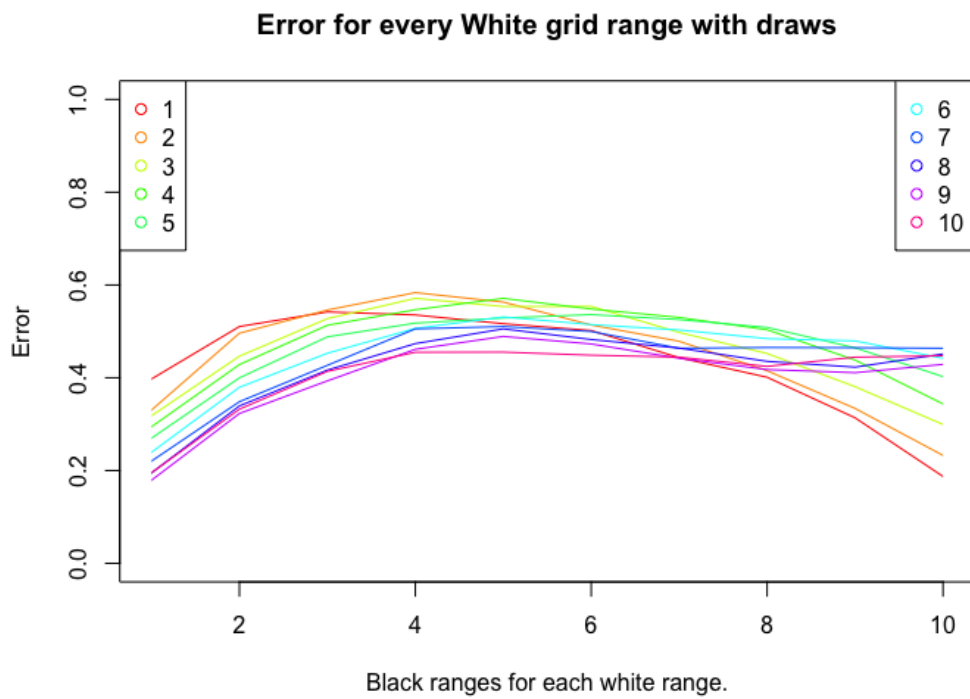


Figure 6.3: SVM model results with draw.

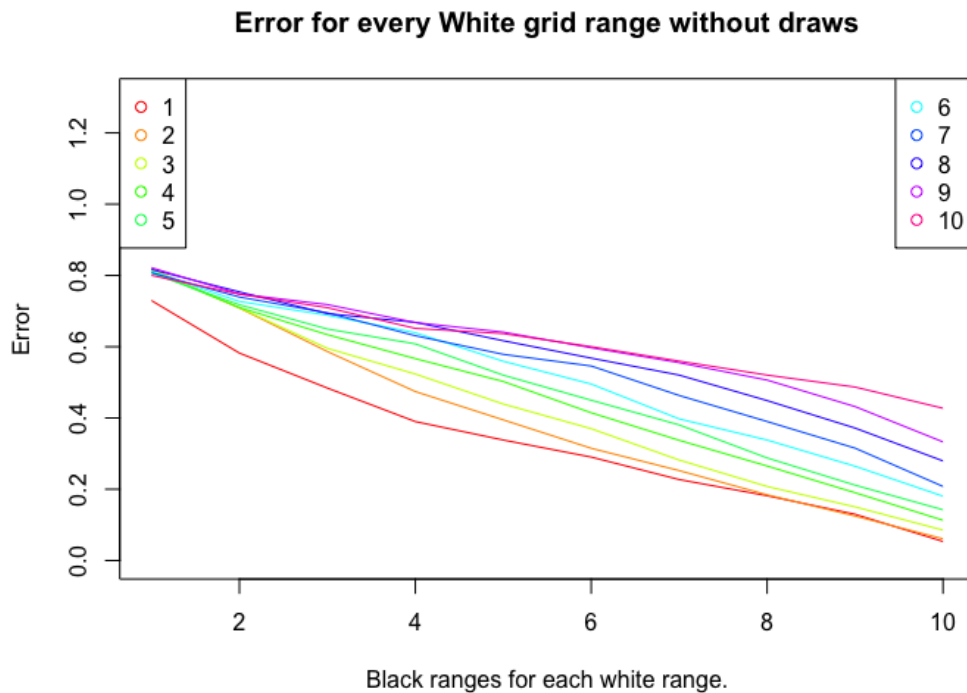


Figure 6.4: Drawless Results of the random forest model.

For the model of the random forest with draw less result, we tried quite a bit of different things, and we couldn't improve the results, we increased the number of trees, changed the variables, etc but we got similar results. More interesting is the reason of why is the plot having a slope, for most if not all of the variables, we double checked if we had done something bad while plotting and we found nothing. This we think is related to the fact that as we approach the upper part of a white range, the black average ELO gets better, and then, the pairings are not that skewed in one player's favor.

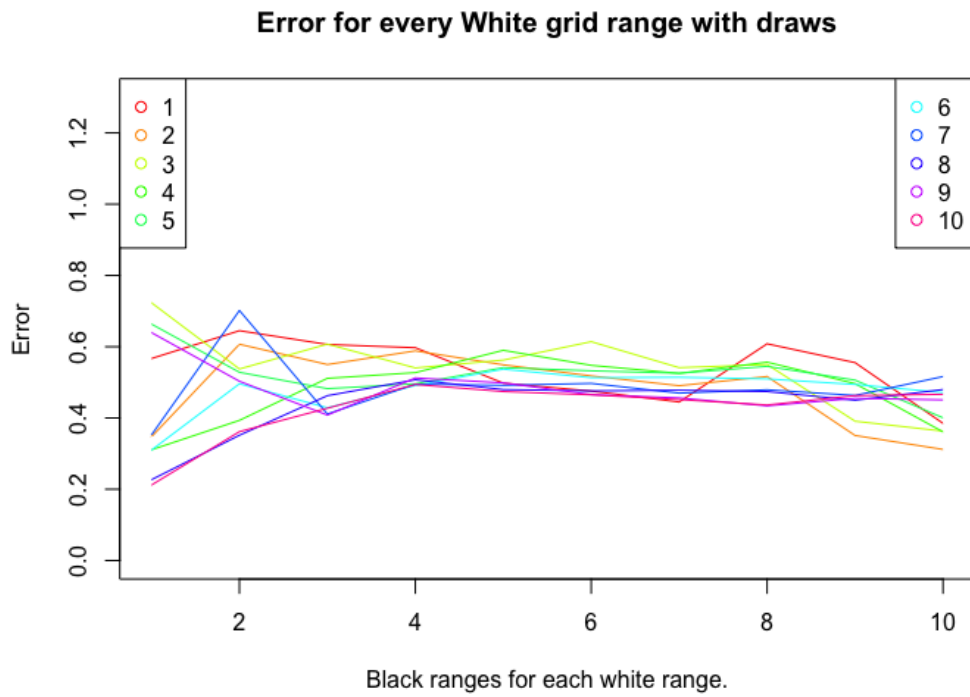


Figure 6.5: Random forest model results with draw.

6.6 Summary of Thesis Achievements

To be able to extract knowledge from a big data file, we needed to cover many steps, starting from a foreign database that we downloaded from the Internet, we were able to decode it, and start manipulating it. Then we started to create the process that would be able to perform operations in the database. As in the data warehousing process, we then started filtering undesired data, aggregating data to create more knowledge and refined data that was inconsistent from the database. We then divided the database into a grid that we would later operate, and each of these grid elements got divided into a train/test suite. We then were confronted into how best to manage a file so big, but we needed to be able to store all of the movements of the board as well as some historical record on how was the result every a match after it went through such a board position. We then decided that we needed to have a traditional database, that even though it was terrible complex, it served well its purpose and allowed us to generate a complete database with all the movements in the training set of our grid, all 1.5 million games that contained more than 13 million different board positions(meaning the pieces position on the board).

We also created a second database with the historical record of the competitors that were in our database, this was much less of a problem because there were no more than 100 thousand players.

Having three sources to condense in our data was what we think helped us at the time of predicting the outcome of matches, at the end we used one of the most well known and used methods in machine learning, that we know would generalize quite well, specially the kernel SVM, in our case.

This is specially interesting if we contrast this with what Vence did in our previous work where a special function to perform then logistic and linear regression, in our case we mostly focused on processing the big database and then consolidating our final dataset to perform our predictions over the test dataset. The SVM specially performed better than using the method that Vence proposed, and we were able to predict the games we kept from our initial purge.

As future work we think it might be of interest to continue testing which variables we could select and create from our dataset, we might have missed something that might be of interest. We think that if there was some implementation that differentiated grid elements zero and ninety nine, using different models for such disparaging dataset would probably make a noticeable improvement. Also we think that the results could be improved using ensemble methods or a neural network, which we didn't implement for time constraints.

At the end of the project we were satisfied with our results, because we went through all the process of big data mining, and we used a generalizing function to predict the results of chess matches after 20 movements.

Appendices

Appendix A

Grid Creation

A.1 Games statistics

#	% W	% B	% Draw	μ	median
0	0.60	0.21	0.18	39.64	38
1	0.46	0.30	0.23	40.28	38
2	0.36	0.36	0.26	40.48	39
3	0.28	0.43	0.27	40.82	39
4	0.25	0.47	0.27	40.68	39
5	0.20	0.52	0.26	40.75	39
6	0.16	0.57	0.25	40.88	39
7	0.14	0.62	0.23	40.60	39
8	0.10	0.70	0.19	40.81	39
9	0.04	0.83	0.11	39.16	37
10	0.69	0.11	0.19	39.62	38
11	0.54	0.18	0.27	40.97	39
12	0.42	0.25	0.31	40.82	39
13	0.33	0.33	0.33	40.93	40
14	0.27	0.39	0.33	41.13	40

15	0.22	0.45	0.32	41.54	40
16	0.18	0.51	0.29	41.75	40
17	0.14	0.58	0.27	42.16	40
18	0.09	0.68	0.22	42.04	40
19	0.05	0.80	0.14	40.50	38
20	0.70	0.10	0.19	39.61	37
21	0.57	0.15	0.26	40.70	39
22	0.46	0.21	0.31	41.11	40
23	0.37	0.27	0.34	40.92	40
24	0.30	0.34	0.35	41.31	40
25	0.25	0.38	0.35	41.70	40
26	0.19	0.46	0.34	41.90	40
27	0.15	0.52	0.31	42.08	41
28	0.11	0.61	0.27	42.25	40
29	0.06	0.73	0.20	41.72	40
30	0.73	0.08	0.17	39.77	38
31	0.59	0.14	0.26	41.35	40
32	0.48	0.20	0.31	41.48	40
33	0.40	0.24	0.35	41.32	40
34	0.33	0.29	0.37	41.16	40
35	0.26	0.33	0.39	40.86	40
36	0.21	0.39	0.38	41.31	40
37	0.18	0.46	0.35	41.69	40
38	0.13	0.54	0.31	42.35	41
39	0.08	0.66	0.25	42.45	41
40	0.76	0.07	0.16	40.12	38
41	0.61	0.12	0.26	41.47	40
42	0.51	0.16	0.32	41.43	40
43	0.42	0.20	0.37	41.08	40

44	0.35	0.23	0.40	40.52	40
45	0.28	0.28	0.42	40.50	40
46	0.23	0.33	0.43	40.50	40
47	0.18	0.40	0.41	40.88	40
48	0.13	0.48	0.37	41.48	40
49	0.09	0.59	0.31	42.60	41
50	0.78	0.05	0.15	39.87	38
51	0.63	0.10	0.25	41.69	40
52	0.54	0.14	0.31	41.62	40
53	0.45	0.17	0.37	40.98	40
54	0.36	0.20	0.42	40.48	40
55	0.30	0.24	0.44	39.74	39
56	0.24	0.28	0.46	39.73	39
57	0.19	0.34	0.46	39.86	39
58	0.15	0.40	0.43	40.94	40
59	0.10	0.52	0.36	42.40	41
60	0.80	0.05	0.14	39.88	38
61	0.66	0.09	0.24	41.50	40
62	0.57	0.12	0.30	41.95	40
63	0.47	0.14	0.37	41.24	40
64	0.38	0.17	0.44	40.51	39
65	0.31	0.20	0.48	39.50	39
66	0.25	0.23	0.50	38.59	38
67	0.21	0.29	0.49	39.39	39
68	0.16	0.33	0.50	39.79	39
69	0.12	0.43	0.44	41.65	40
70	0.83	0.04	0.12	39.64	37
71	0.69	0.07	0.23	41.85	40
72	0.58	0.10	0.30	41.86	40

73	0.50	0.12	0.37	41.64	40
74	0.41	0.14	0.43	40.79	40
75	0.33	0.16	0.49	39.73	39
76	0.28	0.19	0.52	38.73	38
77	0.22	0.22	0.54	38.41	38
78	0.18	0.27	0.54	38.64	38
79	0.14	0.35	0.50	40.78	40
80	0.84	0.03	0.11	39.05	37
81	0.70	0.06	0.22	42.21	40
82	0.60	0.09	0.30	42.19	40
83	0.52	0.10	0.36	41.69	40
84	0.43	0.12	0.43	41.21	40
85	0.36	0.13	0.49	40.13	39
86	0.31	0.16	0.52	39.44	39
87	0.25	0.17	0.56	38.66	38
88	0.20	0.21	0.57	38.58	37
89	0.17	0.29	0.53	41.02	40
90	0.83	0.03	0.12	39.87	38
91	0.69	0.06	0.24	43.05	41
92	0.57	0.08	0.33	43.07	41
93	0.49	0.09	0.40	42.67	41
94	0.42	0.11	0.45	41.94	41
95	0.36	0.12	0.50	41.78	40
96	0.31	0.13	0.54	41.16	40
97	0.29	0.15	0.54	41.81	41
98	0.26	0.18	0.54	42.38	41
99	0.23	0.23	0.53	43.23	41

Table A.1: Grid Basic Statistics

A.2 Grid definition

		White		Black				White		Black	
#	Games	Min	Max	Min	Max	#	Games	Min	Max	Min	Max
1	20590	780	2043	700	1709	51	20671	2348	2397	1078	2153
2	20678	908	2043	1710	1836	52	20804	2348	2397	2154	2229
3	20634	1012	2043	1837	1930	53	20710	2348	2397	2230	2279
4	20365	700	2043	1931	2005	54	20315	2348	2397	2280	2323
5	20657	1079	2043	2006	2065	55	20670	2348	2397	2324	2364
6	20703	100	2043	2066	2120	56	20582	2348	2397	2365	2402
7	20499	215	2043	2121	2173	57	20983	2348	2397	2403	2442
8	20450	100	2043	2174	2237	58	20397	2348	2397	2443	2481
9	20432	100	2043	2238	2329	59	20443	2348	2397	2482	2529
10	20495	1114	2043	2330	2851	60	20577	2348	2397	2530	2820
11	20427	2044	2160	215	1922	61	20911	2398	2448	1146	2189
12	20585	2044	2160	1923	2026	62	20854	2398	2448	2190	2262
13	20306	2044	2160	2027	2097	63	20898	2398	2448	2263	2314
14	20284	2044	2160	2098	2156	64	21277	2398	2448	2315	2359
15	20489	2044	2160	2157	2205	65	20460	2398	2448	2360	2396
16	20412	2044	2160	2206	2249	66	20474	2398	2448	2397	2433
17	20486	2044	2160	2250	2292	67	21220	2398	2448	2434	2468
18	20568	2044	2160	2293	2347	68	20835	2398	2448	2469	2504
19	20076	2044	2160	2348	2421	69	20682	2398	2448	2505	2548
20	20313	2044	2160	2422	2851	70	20511	2398	2448	2549	2820
21	20428	2161	2235	215	2013	71	20351	2449	2498	1351	2221
22	20505	2161	2235	2014	2096	72	20255	2449	2498	2222	2295
23	20388	2161	2235	2097	2162	73	20237	2449	2498	2296	2348
24	20422	2161	2235	2163	2216	74	20283	2449	2498	2349	2389
25	20755	2161	2235	2217	2260	75	20174	2449	2498	2390	2426

26	20245	2161	2235	2261	2301	76	20383	2449	2498	2427	2459
27	20497	2161	2235	2302	2346	77	19817	2449	2498	2460	2491
28	20332	2161	2235	2347	2394	78	19981	2449	2498	2492	2525
29	20260	2161	2235	2395	2458	79	20123	2449	2498	2526	2566
30	20383	2161	2235	2459	2820	80	20137	2449	2498	2567	2820
31	20896	2236	2293	1114	2070	81	20472	2499	2558	1259	2259
32	20627	2236	2293	2071	2151	82	20131	2499	2558	2260	2339
33	21054	2236	2293	2152	2212	83	20408	2499	2558	2340	2388
34	20549	2236	2293	2213	2256	84	20474	2499	2558	2389	2428
35	20806	2236	2293	2257	2296	85	20549	2499	2558	2429	2462
36	20929	2236	2293	2297	2338	86	19729	2499	2558	2463	2492
37	20585	2236	2293	2339	2378	87	20423	2499	2558	2493	2523
38	20865	2236	2293	2379	2425	88	20094	2499	2558	2524	2555
39	20696	2236	2293	2426	2482	89	21108	2499	2558	2556	2595
40	20541	2236	2293	2483	2820	90	19315	2499	2558	2596	2820
41	20149	2294	2347	1118	2114	91	20346	2559	2875	1400	2333
42	20018	2294	2347	2115	2193	92	20430	2559	2820	2334	2416
43	20217	2294	2347	2194	2246	93	20124	2559	2820	2417	2466
44	20041	2294	2347	2247	2288	94	20170	2559	2820	2467	2505
45	20171	2294	2347	2289	2330	95	20294	2559	2820	2506	2537
46	19504	2294	2347	2331	2369	96	20265	2559	2820	2538	2566
47	20283	2294	2347	2370	2410	97	20694	2559	2820	2567	2595
48	19812	2294	2347	2411	2453	98	19868	2559	2820	2596	2634
49	20120	2294	2347	2454	2506	99	20228	2559	2820	2635	2677
50	19829	2294	2347	2507	2820	100	20088	2559	2870	2678	2861

Table A.2: Grid Ranges

A.3 Network Summary statistics.

#	N	E	k	δ	#	N	E	k	δ
0	12060	18274	1.51	0.00025	50	13128	16351	1.24	0.00018
1	13371	17884	1.33	0.00020	51	9889	16617	1.68	0.00033
2	13504	17308	1.28	0.00018	52	8186	16358	1.99	0.00048
3	13395	16755	1.25	0.00018	53	6773	15817	2.33	0.00068
4	14252	17300	1.21	0.00017	54	4688	16062	3.42	0.00146
5	15658	17093	1.09	0.00013	55	3342	15925	4.76	0.00285
6	14941	16557	1.10	0.00014	56	4354	16087	3.69	0.00169
7	15498	16857	1.08	0.00014	57	3774	15388	4.07	0.00216
8	15230	16825	1.10	0.00014	58	3518	15334	4.35	0.00247
9	14387	16103	1.11	0.00015	59	3308	15303	4.62	0.00279
10	15623	16817	1.07	0.00013	60	12559	16427	1.30	0.00020
11	14432	16982	1.17	0.00016	61	8705	16315	1.87	0.00043
12	11817	18035	1.52	0.00025	62	6596	15868	2.40	0.00072
13	10857	18070	1.66	0.00030	63	5084	16369	3.21	0.00126
14	13928	17818	1.27	0.00018	64	3994	15587	3.90	0.00195
15	13655	17282	1.26	0.00018	65	2000	15948	7.97	0.00797
16	12832	16871	1.31	0.00020	66	2426	15985	6.58	0.00542
17	12210	16296	1.33	0.00021	67	2512	15581	6.20	0.00493
18	11322	15446	1.36	0.00024	68	2387	15556	6.51	0.00545
19	10646	15335	1.44	0.00027	69	2292	15119	6.59	0.00575
20	15456	16843	1.08	0.00014	70	11754	16071	1.36	0.00023
21	13876	17243	1.24	0.00017	71	7248	15632	2.15	0.00059
22	13001	17675	1.35	0.00020	72	4855	15266	3.14	0.00129
23	9416	18342	1.94	0.00041	73	3499	15111	4.31	0.00246
24	10807	18179	1.68	0.00031	74	2713	15261	5.62	0.00414
25	11073	16961	1.53	0.00027	75	1782	15246	8.55	0.00959
26	10397	16682	1.60	0.00030	76	1123	14742	13.12	0.02335

27	9522	15950	1.67	0.00035	77	1516	14933	9.85	0.01298
28	8730	15570	1.78	0.00040	78	1499	14616	9.75	0.01300
29	8268	15435	1.86	0.00045	79	1469	14840	10.10	0.01374
30	15224	17553	1.15	0.00015	80	11021	15934	1.44	0.00026
31	12853	17328	1.34	0.00020	81	6007	15457	2.57	0.00085
32	12123	18115	1.49	0.00024	82	3540	15421	4.35	0.00246
33	8887	17921	2.01	0.00045	83	2504	15679	6.26	0.00499
34	6974	17707	2.53	0.00072	84	1743	15675	8.99	0.01031
35	9049	17169	1.89	0.00041	85	1392	14840	10.66	0.01530
36	7955	16168	2.03	0.00051	86	853	15763	18.47	0.04327
37	7767	16246	2.09	0.00053	87	765	14866	19.43	0.05073
38	6847	15753	2.30	0.00067	88	985	15558	15.79	0.03203
39	6376	15491	2.42	0.00076	89	971	14174	14.59	0.03003
40	13783	16203	1.17	0.00017	90	9834	15711	1.59	0.00032
41	10771	15969	1.48	0.00027	91	3952	15607	3.94	0.00199
42	10076	16915	1.67	0.00033	92	1881	15121	8.03	0.00854
43	8489	16426	1.93	0.00045	93	1216	15198	12.49	0.02053
44	5102	16382	3.21	0.00125	94	867	14995	17.29	0.03985
45	5560	15213	2.73	0.00098	95	663	14740	22.23	0.06696
46	6199	15657	2.52	0.00081	96	463	14987	32.36	0.13952
47	5426	15159	2.79	0.00102	97	464	14640	31.55	0.13570
48	4930	14725	2.98	0.00121	98	448	14484	32.33	0.14401
49	4575	14762	3.22	0.00141	99	450	14263	31.69	0.14055

Table A.3: Competitors Network Summary

Appendix B

Database missing data.

number	board	white	black
0	0.918662	0.015444	0.00669241
1	0.887497	0.0100237	0.00619889
10	0.842572	0.0151053	0.00180155
11	0.792979	0.00568891	0.00249757
12	0.748188	0.00452899	0.00258799
13	0.725759	0.00266383	0.00332978
14	0.717105	0.0025	0.00355263
15	0.717791	0.00241319	0.00388792
16	0.712677	0.00152905	0.00569919
17	0.72495	0.000717772	0.00803905
18	0.727585	0.000596748	0.00865284
19	0.73379	0.000152207	0.0199391
2	0.853523	0.00623306	0.00758808
20	0.800746	0.0181014	0.00234904
21	0.738092	0.00728647	0.00161921
22	0.694474	0.00252391	0.00172689
23	0.692198	0.00426246	0.0041333

24	0.674242	0.00205444	0.00295326
25	0.673472	0.000811249	0.00338021
26	0.668966	0.00070373	0.00253343
27	0.675664	0.000146692	0.00440076
28	0.654855	0.000150784	0.00361882
29	0.693585	0	0.00898979
3	0.846438	0.00410228	0.00984548
30	0.774516	0.0164109	0.000266845
31	0.719775	0.00439138	0.000960615
32	0.685393	0.00361617	0.00116234
33	0.664129	0.00157501	0.00105001
34	0.654531	0.00107081	0.00107081
35	0.637922	0.000274198	0.000685495
36	0.631772	0.000146413	0.000585652
37	0.641432	0.000141503	0.00183954
38	0.633643	0	0.0032434
39	0.669155	0.000152323	0.00274181
4	0.826246	0.00285287	0.00910202
40	0.745584	0.0154938	0.00115841
41	0.678735	0.00429307	0.00100172
42	0.643023	0.00204611	0.000136407
43	0.630131	0.00115224	0.00043209
44	0.603817	0.00115674	0.000433777
45	0.581795	0.000312793	0.00125117
46	0.602568	0	0.000447895
47	0.587378	0	0.00144161
48	0.602963	0	0.000604778
49	0.62198	0.000159974	0.00111982
5	0.81221	0.0035432	0.00953938

50	0.732115	0.0152996	0
51	0.679061	0.0041001	0.000282765
52	0.643219	0.00140687	0.000281373
53	0.613466	0.00187428	0.000144175
54	0.589584	0.000443853	0.000147951
55	0.569123	0	0
56	0.555685	0	0.000145582
57	0.554172	0	0.000311333
58	0.569964	0	0.000155304
59	0.614332	0	0.000310897
6	0.802249	0.0018508	0.0126708
60	0.70912	0.0187293	0.000144071
61	0.657988	0.00254705	0.000566011
62	0.618719	0.00158387	0
63	0.589498	0.000870322	0
64	0.56301	0.000302206	0
65	0.540066	0.000148943	0.000148943
66	0.52127	0	0
67	0.51133	0	0.000298151
68	0.533015	0.000149388	0
69	0.574816	0	0.000152999
7	0.79193	0.00150974	0.0166072
70	0.704489	0.0220751	0
71	0.653771	0.00240855	0
72	0.599006	0.000903478	0
73	0.569157	0.00030003	0
74	0.554627	0.000303905	0.000151953
75	0.523168	0	0
76	0.50552	0	0

77	0.486478	0.000153657	0
78	0.483994	0	0
79	0.508275	0	0
8	0.800962	0.000707314	0.0236243
80	0.688541	0.0211926	0.000146156
81	0.622109	0.00225293	0
82	0.592997	0.000748167	0
83	0.569238	0.000147471	0
84	0.542996	0	0
85	0.505403	0	0
86	0.493476	0	0
87	0.467796	0	0
88	0.457068	0	0
89	0.490222	0	0
9	0.804435	0.000144009	0.0550115
90	0.666372	0.0167696	0
91	0.608153	0.000451264	0.000300842
92	0.572789	0.000151172	0.000453515
93	0.536026	0	0
94	0.504221	0	0
95	0.492144	0	0
96	0.487782	0.000155642	0
97	0.471272	0	0.000484183
98	0.48156	0.000491723	0
99	0.455161	0.000165153	0

Table B.1: Percentage of Missing data for our final dataset.

Appendix C

Results

C.1 Gaussian Kernel with three fold crossvalidation.

C.1.1 Without Draws.

	SVM			Random Forest		
	Total	Black	White	Total	Black	White
0	0.21	0.65	0.06	0.73	0.02	0.97
1	0.28	0.49	0.14	0.58	0.01	0.96
2	0.32	0.35	0.30	0.48	0.02	0.96
3	0.34	0.20	0.55	0.39	0.01	0.97
4	0.31	0.09	0.73	0.34	0.02	0.95
5	0.29	0.04	0.89	0.29	0.01	0.96
6	0.23	0.00	0.99	0.23	0.01	0.97
7	0.18	0.00	1.00	0.18	0.01	0.96
8	0.13	0.00	1.00	0.13	0.00	0.97
9	0.05	0.00	1.00	0.05	0.00	0.96
10	0.14	1.00	0.00	0.82	0.05	0.94
11	0.25	1.00	0.00	0.71	0.06	0.92

12	0.37	0.83	0.10	0.59	0.05	0.90
13	0.43	0.39	0.48	0.47	0.04	0.90
14	0.40	0.14	0.75	0.39	0.02	0.91
15	0.32	0.03	0.93	0.31	0.02	0.92
16	0.26	0.00	1.00	0.25	0.02	0.93
17	0.19	0.00	1.00	0.18	0.01	0.92
18	0.12	0.00	1.00	0.12	0.01	0.95
19	0.06	0.00	1.00	0.06	0.00	0.94
10	0.12	1.00	0.00	0.82	0.07	0.92
21	0.21	1.00	0.00	0.71	0.07	0.88
22	0.34	0.97	0.02	0.60	0.07	0.86
23	0.41	0.70	0.20	0.52	0.06	0.86
24	0.43	0.22	0.67	0.44	0.04	0.89
25	0.39	0.09	0.86	0.37	0.03	0.90
26	0.30	0.00	1.00	0.28	0.02	0.89
27	0.22	0.00	1.00	0.21	0.01	0.91
28	0.16	0.00	1.00	0.15	0.01	0.92
29	0.09	0.00	1.00	0.09	0.00	0.93
20	0.11	1.00	0.00	0.81	0.06	0.90
31	0.20	1.00	0.00	0.71	0.07	0.87
32	0.29	0.99	0.01	0.63	0.08	0.86
33	0.40	0.78	0.17	0.57	0.07	0.86
34	0.45	0.60	0.32	0.50	0.06	0.88
35	0.42	0.12	0.80	0.41	0.04	0.87
36	0.36	0.02	0.96	0.34	0.04	0.87
37	0.28	0.00	1.00	0.27	0.02	0.91
38	0.19	0.00	1.00	0.19	0.01	0.93
39	0.11	0.00	1.00	0.11	0.01	0.93
30	0.09	1.00	0.00	0.81	0.08	0.88

41	0.17	1.00	0.00	0.72	0.10	0.84
42	0.24	1.00	0.00	0.65	0.09	0.83
43	0.31	1.00	0.00	0.61	0.09	0.85
44	0.41	0.85	0.11	0.52	0.06	0.83
45	0.45	0.45	0.45	0.45	0.06	0.85
46	0.40	0.04	0.92	0.38	0.04	0.88
47	0.30	0.00	1.00	0.29	0.02	0.89
48	0.22	0.00	1.00	0.21	0.02	0.89
49	0.15	0.00	1.00	0.14	0.01	0.92
40	0.07	1.00	0.00	0.81	0.09	0.87
51	0.15	1.00	0.00	0.73	0.08	0.84
52	0.21	1.00	0.00	0.69	0.10	0.84
53	0.27	1.00	0.00	0.64	0.10	0.84
54	0.37	1.00	0.00	0.56	0.11	0.82
55	0.44	0.74	0.20	0.50	0.08	0.83
56	0.43	0.15	0.79	0.40	0.04	0.86
57	0.36	0.02	0.97	0.34	0.03	0.88
58	0.28	0.00	1.00	0.26	0.02	0.90
59	0.19	0.00	1.00	0.18	0.01	0.91
50	0.06	1.00	0.00	0.81	0.12	0.85
61	0.12	1.00	0.00	0.74	0.08	0.83
62	0.17	1.00	0.00	0.70	0.11	0.82
63	0.25	1.00	0.00	0.63	0.09	0.81
64	0.51	0.33	0.59	0.58	0.09	0.80
65	0.39	0.90	0.07	0.55	0.10	0.82
66	0.45	0.61	0.29	0.46	0.06	0.84
67	0.42	0.00	1.00	0.39	0.05	0.87
68	0.34	0.00	1.00	0.32	0.03	0.88
69	0.23	0.00	1.00	0.21	0.01	0.88

60	0.05	1.00	0.00	0.82	0.08	0.85
71	0.11	1.00	0.00	0.75	0.10	0.83
72	0.15	1.00	0.00	0.69	0.12	0.80
73	0.19	1.00	0.00	0.67	0.14	0.80
74	0.26	1.00	0.00	0.62	0.09	0.80
75	0.33	1.00	0.00	0.57	0.10	0.80
76	0.44	0.71	0.24	0.52	0.11	0.83
77	0.46	0.13	0.78	0.45	0.08	0.81
78	0.40	0.01	0.98	0.37	0.05	0.85
79	0.30	0.00	1.00	0.28	0.03	0.87
70	0.04	1.00	0.00	0.82	0.14	0.85
81	0.09	1.00	0.00	0.75	0.11	0.81
82	0.13	1.00	0.00	0.72	0.12	0.81
83	0.18	1.00	0.00	0.67	0.13	0.79
84	0.23	1.00	0.00	0.64	0.13	0.79
85	0.29	1.00	0.00	0.60	0.14	0.78
86	0.36	0.95	0.04	0.56	0.13	0.78
87	0.42	1.00	0.00	0.51	0.12	0.79
88	0.43	0.34	0.53	0.43	0.07	0.82
89	0.36	0.00	1.00	0.33	0.04	0.86
80	0.04	1.00	0.00	0.80	0.12	0.83
91	0.09	1.00	0.00	0.75	0.13	0.81
92	0.13	1.00	0.00	0.71	0.12	0.80
93	0.17	1.00	0.00	0.65	0.12	0.76
94	0.21	1.00	0.00	0.64	0.13	0.77
95	0.25	1.00	0.00	0.60	0.12	0.76
96	0.31	1.00	0.00	0.56	0.11	0.76
97	0.37	1.00	0.00	0.52	0.12	0.75
98	0.37	0.63	0.19	0.49	0.09	0.76

99	0.39	0.41	0.38	0.43	0.08	0.78
mean	0.26			0.491		

Table C.1: Result using SVM, without draws.

C.1.2 With Draws.

Error	SVM				Random Forest			
	Total	Black	Draw	White	Total	Black	Draw	White
0	0.40	0.82	0.84	0.12	0.57	0.68	0.49	0.55
1	0.51	0.56	0.63	0.41	0.64	0.10	0.88	0.89
2	0.54	0.55	0.55	0.53	0.61	0.48	0.87	0.53
3	0.54	0.32	0.59	0.81	0.60	0.67	0.86	0.23
4	0.52	0.31	0.56	0.86	0.50	0.15	0.80	0.82
5	0.50	0.27	0.61	0.91	0.48	0.13	0.77	0.93
6	0.44	0.20	0.67	0.92	0.44	0.19	0.73	0.88
7	0.40	0.17	0.70	0.92	0.61	0.57	0.81	0.43
8	0.31	0.08	0.84	0.93	0.56	0.53	0.57	0.68
9	0.19	0.05	0.90	0.93	0.38	0.31	0.86	0.70
10	0.33	0.95	0.83	0.10	0.35	0.92	0.81	0.13
11	0.50	0.92	0.59	0.31	0.61	0.42	0.74	0.60
12	0.55	0.83	0.40	0.49	0.55	0.58	0.79	0.36
13	0.58	0.74	0.26	0.75	0.59	0.27	0.70	0.79
14	0.56	0.61	0.28	0.84	0.55	0.60	0.51	0.53
15	0.51	0.39	0.45	0.87	0.52	0.30	0.62	0.83
16	0.48	0.28	0.57	0.89	0.49	0.34	0.52	0.89
17	0.42	0.17	0.71	0.89	0.52	0.42	0.66	0.63
18	0.33	0.10	0.79	0.92	0.35	0.13	0.79	0.90
19	0.23	0.07	0.86	0.90	0.31	0.20	0.71	0.86
20	0.32	0.97	0.80	0.10	0.72	0.60	0.40	0.82

21	0.45	0.91	0.62	0.25	0.54	0.58	0.78	0.42
22	0.53	0.88	0.42	0.42	0.61	0.33	0.68	0.70
23	0.57	0.80	0.29	0.66	0.54	0.80	0.62	0.28
24	0.55	0.62	0.27	0.81	0.56	0.47	0.65	0.55
25	0.55	0.64	0.24	0.85	0.61	0.81	0.68	0.23
26	0.50	0.42	0.39	0.87	0.54	0.53	0.46	0.71
27	0.45	0.28	0.55	0.87	0.55	0.61	0.32	0.81
28	0.38	0.15	0.69	0.89	0.39	0.17	0.69	0.87
29	0.30	0.10	0.81	0.87	0.36	0.24	0.63	0.85
30	0.29	0.92	0.84	0.09	0.31	0.89	0.81	0.12
31	0.43	0.90	0.65	0.22	0.39	0.87	0.78	0.11
32	0.51	0.87	0.44	0.42	0.51	0.83	0.44	0.43
33	0.55	0.89	0.29	0.57	0.53	0.87	0.66	0.20
34	0.57	0.82	0.18	0.79	0.59	0.24	0.61	0.86
35	0.55	0.81	0.13	0.84	0.55	0.75	0.40	0.51
36	0.53	0.63	0.23	0.85	0.53	0.55	0.31	0.85
37	0.50	0.45	0.39	0.87	0.56	0.66	0.31	0.79
38	0.44	0.23	0.61	0.89	0.50	0.47	0.40	0.84
39	0.34	0.12	0.78	0.87	0.36	0.16	0.74	0.85
40	0.27	0.94	0.84	0.07	0.66	0.48	0.72	0.67
41	0.40	0.89	0.70	0.18	0.53	0.69	0.56	0.48
42	0.49	0.86	0.46	0.39	0.48	0.79	0.58	0.32
43	0.52	0.87	0.25	0.57	0.50	0.77	0.57	0.31
44	0.53	0.85	0.16	0.75	0.54	0.75	0.19	0.82
45	0.54	0.84	0.12	0.84	0.53	0.55	0.33	0.82
46	0.53	0.76	0.14	0.87	0.53	0.38	0.46	0.84
47	0.51	0.60	0.25	0.89	0.54	0.66	0.36	0.70
48	0.46	0.41	0.39	0.86	0.51	0.53	0.35	0.84
49	0.40	0.18	0.67	0.89	0.40	0.16	0.70	0.86

50	0.24	0.92	0.87	0.06	0.31	0.84	0.84	0.16
51	0.38	0.90	0.71	0.16	0.50	0.76	0.55	0.43
52	0.45	0.89	0.54	0.28	0.43	0.88	0.73	0.14
53	0.51	0.88	0.30	0.54	0.49	0.85	0.65	0.23
54	0.53	0.89	0.12	0.78	0.54	0.63	0.50	0.53
55	0.51	0.86	0.11	0.85	0.51	0.78	0.31	0.61
56	0.50	0.85	0.10	0.87	0.51	0.62	0.29	0.82
57	0.48	0.76	0.13	0.86	0.51	0.47	0.45	0.72
58	0.48	0.56	0.25	0.87	0.49	0.24	0.62	0.81
59	0.44	0.25	0.57	0.88	0.47	0.35	0.55	0.78
60	0.22	0.90	0.88	0.06	0.35	0.89	0.63	0.27
61	0.35	0.90	0.76	0.13	0.70	0.35	0.62	0.78
62	0.43	0.91	0.59	0.24	0.41	0.89	0.71	0.14
63	0.51	0.88	0.30	0.55	0.49	0.82	0.53	0.36
64	0.51	0.90	0.15	0.75	0.49	0.88	0.40	0.42
65	0.50	0.88	0.11	0.84	0.50	0.79	0.24	0.69
66	0.46	0.86	0.09	0.85	0.47	0.83	0.18	0.72
67	0.47	0.82	0.10	0.86	0.48	0.56	0.32	0.77
68	0.46	0.84	0.09	0.84	0.46	0.49	0.33	0.81
69	0.46	0.57	0.25	0.84	0.52	0.78	0.17	0.80
70	0.19	0.93	0.89	0.05	0.23	0.89	0.85	0.10
71	0.34	0.89	0.83	0.10	0.35	0.87	0.80	0.13
72	0.42	0.90	0.61	0.23	0.46	0.83	0.53	0.36
73	0.47	0.89	0.42	0.42	0.51	0.85	0.43	0.48
74	0.51	0.87	0.17	0.72	0.48	0.86	0.43	0.40
75	0.48	0.88	0.10	0.82	0.48	0.83	0.37	0.46
76	0.46	0.88	0.09	0.84	0.48	0.76	0.27	0.66
77	0.44	0.87	0.08	0.84	0.47	0.73	0.29	0.66
78	0.42	0.84	0.09	0.85	0.45	0.69	0.25	0.72

79	0.45	0.66	0.19	0.85	0.48	0.37	0.46	0.81
80	0.18	0.92	0.86	0.06	0.64	0.86	0.29	0.68
81	0.32	0.89	0.81	0.11	0.50	0.86	0.37	0.51
82	0.39	0.90	0.72	0.16	0.41	0.89	0.66	0.21
83	0.46	0.90	0.45	0.37	0.51	0.89	0.33	0.56
84	0.49	0.88	0.21	0.65	0.50	0.86	0.33	0.56
85	0.47	0.87	0.12	0.77	0.47	0.84	0.42	0.37
86	0.44	0.87	0.10	0.82	0.46	0.75	0.34	0.51
87	0.42	0.88	0.09	0.83	0.43	0.82	0.15	0.80
88	0.41	0.86	0.09	0.83	0.46	0.54	0.32	0.73
89	0.43	0.81	0.09	0.86	0.45	0.72	0.20	0.81
90	0.20	0.92	0.87	0.06	0.21	0.87	0.84	0.09
91	0.33	0.89	0.81	0.10	0.36	0.87	0.72	0.18
92	0.41	0.88	0.67	0.19	0.43	0.87	0.66	0.22
93	0.45	0.89	0.41	0.40	0.49	0.87	0.28	0.60
94	0.46	0.88	0.22	0.61	0.47	0.85	0.34	0.52
95	0.45	0.86	0.14	0.71	0.47	0.82	0.31	0.55
96	0.44	0.86	0.10	0.81	0.45	0.84	0.18	0.72
97	0.42	0.84	0.11	0.79	0.44	0.80	0.17	0.74
98	0.44	0.84	0.11	0.80	0.46	0.69	0.28	0.65
99	0.45	0.85	0.10	0.85	0.47	0.71	0.22	0.79
mean	0.43				0.48			

Table C.2: Result of draws using SVM.

Bibliography

- [1] Laws of chess - the notation. Accessed: 2016-03-10.
- [2] Adam LeVasseur. dateparser python parser for human readable dates DateParser 0.4.0 documentation.
- [3] H. E. Bird. *Chess History and Reminiscences*. Project Gutenberg.
- [4] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [5] B. Blasius and R. Tönjes. Zipf’s law in the popularity distribution of chess openings. *Phys. Rev. Lett.*, 103:218701, Nov 2009.
- [6] R. Bth. Million base to json., 2015. Accessed: 2016-02-21.
- [7] N. Charness. Expertise in chess: The balance between knowledge and search. *Toward a general theory of expertise: Prospects and limits*, pages 39–63, 1991.
- [8] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.
- [9] S. Droste and J. Fürnkranz. Learning the piece values for three chess variants. *ICGA Journal*, 31(4):209–233, 2008.
- [10] N. F. et al. python-chess: a pure python chess library. 2012. Accessed: 2016-02-22.
- [11] A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis. kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004.

- [12] PossibleOathMeal. Gorgonian's chess site, downloads., 2015. Accessed: 2016-02-20.
- [13] Python.org. GlobalInterpreterLock - python wiki.
- [14] Python.org. multiprocessing process-based threading interface python 2.7.12 documentation.
- [15] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.
- [16] C. E. Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer, 1988.
- [17] SqliteCommunity. About SQLite.
- [18] SqliteCommunity. SQLite CVSTrac.
- [19] E. Vence. Chess games study & prediction through the use of web-scraping and regression analysis. 2015.
- [20] Wikipedia. Portable game notation., 2016. Accessed: 2016-02-30.