Improving Duckworth-Lewis: Statistical Methods for Revising Score Targets in Limited-Overs Cricket

Matthew Knowles

Department of Mathematics,
University of York,
United Kingdom

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Mathematics

Abstract

A Neural Network model for predicting scores in the sport of Cricket is presented. The aim is to use our model to improve upon the results of the method currently employed by the International Cricket Council (ICC). We explore this current method, known as the DLS method [Stern, 2016] in some detail before going on to develop our own model. This is followed by some exploratory data analysis, which is imperative in making certain assumptions that aid in the building of the Neural Network. For the Neural Network, we take an in-depth look at the mathematical underpinning of building and training such a model. This is accompanied by code snippets demonstrating how these methods are implemented in a high-level programming language. We then perform an analysis of the results of our model before applying it to the data from the ICC 2019 Cricket World Cup, in order to see how it faired against the DLS method. We find that our method does not predict scores perfectly, but does produce slightly more attainable score targets than DLS- as illustrated by the World Cup simulation study.

Acknowledgements

I would like to thank Dr. Jessica Hargreaves for their invaluable supervision, encouragement and advice throughout this project. In addition, many thanks to Michael Najdan at Kent County Cricket Club for their insight into how data is used in the professional game of cricket, and to Harrison Allen at Yorkshire County Cricket Club for demonstrating how limited-overs data was put into practice in preperation for games. I couldn't write a dissertation about cricket without mentioning the University of York Men's Cricket Club- it has been wonderful to share the playing field with you all over the past 4 years. Finally, I would like to dedicate this dissertation to my friends from Lindley G; thank you for being the most wonderful and supportive people I've ever met. You'll never know how much you mean to me.

Statment of Originallity

The enclosed content is my own work. Any work taken from another source has been appropriately acknowledged and referenced.

Notes

All code written in support of this project can be found on GitHub at:

- *https://github.com/mattknowles314/mmathThesis.*
- *https://github.com/mattknowles314/CricNet.*

Contents

List of Figures

Chapter 1

Introduction

I don't like discussing cricket off the field.

MS Dhoni

This chapter provides the background for the area of research we explore in this dissertation. Data has become widely employed in both professional and amateur sport in the past few years, and in this chapter we will look at how data is used in cricket. We go on to give a brief description of the game of cricket itself, which the sports-inclined reader may still find useful as it helps with understanding where the metrics used in the remainder of the project come from. That disscusion leads into explaining the need for the DLS system used in the modern game, and the problems thereof. The aims of the project are outlined and we take a look at current literature on the topic of data in cricket.

1.1 Data in Cricket

The use of data in sport has become a massive part of how teams prepare for games and competitions in recent years. In cricket particularly, one way data has changed the game is the use of metrics such as strike-rate for selecting bowlers in the starting XI. In previous eras, more qualitative methods such how "green" the pitch looks would have taken precedence. That's not to say such things don't still have a part to play, but the introduction of quantitative methods has become much more commonplace in the modern game. Similarly, a coach may look at how wicket-taking in the "powerplay"[1] vs in the "middle overs" will change the outcome of the game. Data visualisation is key to putting this information across to playing staff in order to highlight key areas for improvement. It is for this reason, we have taken an approach of using extensive data-visualisation to convey how the maths being done translates onto the cricket field.

1.2 Limited-Overs Cricket and the need for Duckworth-Lewis

Cricket is a game played by two teams of eleven players. A coin toss decideds which team will "bat" first, and which will "field" first. Unlike other sports, the scores in cricket are counted in "runs". Runs are achieved in many ways, such as the two batters physically running to change sides of the pitch they are on, or by hitting the ball over a pre-defined boundary. The game of cricket itself has two main forms. In this project we will only look at "limited-overs cricket". Cricket is played in "overs", each one consisting of 6 individual bowls delivered consecutively by the same bowler from one end of the pitch. Traditionally, there is no upper limit on the amount of overs a team may bowl, they keep going until the batting team "declares", meaning

[1] In ODI cricket, this refers to the first 10 overs of the game, where certain fielding restrictions are in place.

they believe they have a tactical advantage if they bowl now instead of later, or preferably for the bowling team, are all bowled out. Batters can be given "out" in quite a few different ways, but once a batter is out- that's it for them. Once ten of the eleven players on the batting team have been given out, the innings comes to end and the team bowling gets their turn to bat. If they hit more runs before losing all their ten wickets or before the innings comes to an end, then they win. Cricket is an immensely complex game with countless intricacies and neuances, but at it's core, applying mathematics to it is not an absurd proposition.

The traditional format of cricket, called "first-class cricket" lasts up to 4 days. This obviously has disadvantages from a spectator's point of view. In 1963, the cricketing calender in the UK had a different format of the game ammended to it alongside the usual first-class fixtures. The "Gillette Cup" introduced a form of cricket wherein each team only has 1 innings, lasting 50 overs isntead of the prior method of no limit. The idea was that this competition, coming to a conclusion within the space of a day, would increase spectator numbers and by extension, ticket sales. Rather than a league as is the structure with first class fixtures, this competition would be tournament based knock-out competition. This tournament ran for the last time in 2009, although a predecessor still populates the cricketing calender today. It is now known as the "Royal London One-Day Cup", however due to the introduction of other tournaments and so-called "franchise cricket" competitions, such as the Indian Premier League or Pakistan Super League, the tournament has lost some relevancy in England and Wales.

Given the tournament-based nature of this new competition, we have the natural need for a definitive result. This is something that is not always guarenteed in first class cricket, where draws are common.[2] As such, in order to allow for a result to be determined when a game is cut short, the idea of target-revision was introduced. This meant that a team could be set a roughly equivalent run target off of a reducded number of overs that would still classify them as the winners. This is where Duckworth and Lewis came in, proposing their D/L method [Duckworth and Lewis, 1998]. As will be discussed in Chapter 3, the D/L method has been slighlty improved upon since it was published in the late 1990s, but the overarching philosophy of the method has stayed the same. One extension was that of Stern in 2004, [Stern, 2016], now known as the DLS method. This allowed for better score resetting in Twenty-20 (T20) cricket- an even shorter form of the game introduced to last only over the course of an afternoon instead of a day.

1.3 Problems with DLS

In [Phanse and Deorah, 2011], the authors found that DLS has a bias towards not only the team batting first, but whoever won the coin toss at the start of the game. The winner of the toss chooses whether their side will bat or bowl in the first innings. They go onto to propose a simple extension to DLS for reducing these biases, but we won't cover that in this project.

DLS can also be prone to setting targets that are too far-fetched. We will see an example of this in Chapter 7. When such a ludicros target is set, it can kill the game well before the end, since the batting team know that they cannot obtain the target, and are fighting an already lost battle. Not only is this bad for the batting team, but it ruins the game for spectators, some of whom may have paid considerable amounts of money to spend the day watching the game.

[2]Note that "draw" and "tie" are not interchangable terminology in cricket. A "tie" refers to achieving the exact same score, whereas a draw is where neither teams wins nor loses

1.4 Project Aims

The main aim of this project is to look at methods for *predicting* cricket scores before resetting them. It is important to make the distinction between this and *projecting* the scores, which is commonplace in the game at the minute. Projecting scores assumes a constant run rate, which it will be seen in later chapters is not really the case. However, we can make use of the patterns in run rates to try and extract a predictable score. At the lowest level, we are going to train a neural network with inputs being run rates and an output being score. We can then pass the run rates of a game that gets cut short to this network, and it will try to predict a score based on the patterns in the runrates of that game.

The runrate is defined by the following formula,

$$\text{Run Rate} = \frac{\text{Total Runs Scored}}{\text{Total Overs Bowled}}.$$

This rate evolves as the game flows, and can skyrocket or plummet as one team or the other begins to outplay their opponent. To our knowledge, a run rate based neural network does not exist in the literature, and so our model is somewhat novel.

1.5 Review of Current Literature

We begin the look at literature on this topic with the paper published by F. Duckworth himself in 1998 [Duckworth and Lewis, 1998]. In this paper, Duckworth proposes the "D/L" method. However, there is an issue with the sentence that appears after defining the function $Z(u, w)$; "Comercial confidentiality prevents the disclosure of the mathematical definitions of these functions", which means we can't access the actual parameters that they use for defining the model. This won't affect the development of our model, but it is unfortunate as it means getting a clear picture of DLS is slightly harder. The claim is that these functions have been defined via experimentation, which gives rise to the first issue: the first T20 game of cricket was not played until 2003. So clearly, D/L was not designed with T20 in mind, and so the functions derived from experementation and research will not be accurate for T20 cricket.

In 2004, a year after the first T20 matches were played, Duckworth and Lewis published another paper [Duckworth and Lewis, 2004], in which they report that the table used for calculating the method can be employed by anyone at all levels of the game- from grass-roots cricket up to the highest standard of the professional game. The table is slightly updated as well due to the increased amount of data from which to calculate parameters.

Another noteworthy paper is that of [Saqlain et al., 2019], looking at predicting scores from the 2019 Cricket World Cup, an ODI tournament held across England. They used data after the previous world cup, held in Australia and New Zealand in 2015; all the way to 10^{th} November 2018. The interesting thing about this paper, is they do not use data from individual matches, but instead on 10 meta-statics, including "Number of series victories", "total wickets", "centuries" and "all out", among others. They apply the TOPSIS method to this specific problem, using Algorithm 1.

They used the results of this process to predict how the 2019 World Cup would look, rather than games for individual scores. However, they were largely unsuccessful. Not only did the authors leave Sri Lanka out of their calculations entirely, they only succesffuly predicted India would finish top of the points table, and Afghanistan at the bottom. It's worth mentioning

Algorithm 1 Modified TOPSIS Algorithm

Require: $m, n \geq 0$

1: Calculate normalised matrix $X_{ij} = \frac{x_{ij}}{\sqrt{\sum_{i=1}^{m} x_{ij}^2}}$

2: Calculate weighted normalised matrix $V_{ij} = X_{ij} W_j$

3: **for** $i \in \{1, \ldots, m\}$ **do**

4: Calculate Euclidean distance from the best outcome $S_i^+ = \sqrt{\sum_{j=1}^{n} (V_{ij} - V_j^+)^2}$

5: Calculate Euclidean distance from the worst outcome $S_i^- = \sqrt{\sum_{j=1}^{n} (V_{ij} - V_j^-)^2}$

6: **end for**

7: Calculate performance score $C_i^* = \frac{S_i^-}{S_i^+ + S_i^-}$

that this may not be an issue with the mathematical model iteself, but the imbalance of data. Teams like Afghanistan play far less games than a team like England or Inida. This introduces a bias favouring teams who have played more games not only from a statistical point of view, but from a cricketing one too since the teams that play less will have less match practice going into a big tournament like the World Cup.

However, the paper [Kumar and Roy, 2018] takes an approach along the same lines of the aim of this project. It is not necessary to discuss the individual aspects of their paper here, as it will be discussed when we dive into the methods themselves. However it is worth noting the results of this paper now to see how ours differ. They found their limited dataset to be a problem in classifying. This is something we have been aware of, but there isn't much that can be done given the nature of how mmany cricket games are played year on year.

Machine learning is certainly not new to cricket, [Kampakis and Thomas, 2015] managed to predict the winners in English T20 cricket in "almost two thirds" of their test cases. While this is a fantastic result, in this dissertation we have not concerned ourselves with directly predicting winners, but actual scores. This is again something that is hard to find in cricket specific literature, so we have taken inspiration from industries such as finance, as well as other sports in building our model.

Chapter 2

Data

> One person's data is another person's noise
>
> K.C. Cole

The purpose of this small chapter is to give an overview of the data that is used for this project.

2.1 Data Origin

The primary source of data for carrying out this project was downloaded from "cricsheet"[1] and stored locally on a private server. In total there are 2167 individual matches of data. Each in a JSON format, covering matches ranging from the 3^{rd} of January 2004 to the 20^{th} of July 2021.

There is actually an R package called `cricketdata` that allows for the direct import of data from ESPNCricinfo[2] and Cricsheet. However this package was published in February 2022, several months after the bulk of work on this project started. In hindsight, using this package would have been a lot easier and saved a considerable amount of time. With that in mind, future work on the `CricNet` package (See Appendix C) will incrorperate the `cricketdata` package in order to streamline the analysis pipeline as a whole.

2.2 Attributes

Each JSON file contains a considerable amount of metadata surrounding the match in question, along with ball-by-ball data for the entire match. We have access to attributes such as the date, where the match was played, the entire teamsheet for both teams, who the officials were, who won- and by what margin, who won the toss; and many others.

We also have the ball-by-ball data. So for every ball bowled, it gives who were the striking and non-striking batsmen, how many runs were scored and how. It also details if a wicket was taken that ball, and how.

2.3 Pre-Processing

In order to clean data and make it usable, Python scripts were written to take the original JSON files and turn them into CSV files on which analysis could be performed in R. These scripts made heavy use of the base-Python libraries `JSON` and `CSV`.

[1] https://cricsheet.org/

[2] https://www.epsncricinfo.com/

2.4 Problems

When it comes to machine learning, the more data the better is a general rule. Now this can sometimes lead to sub-problems, such as overfitting. But on the whole it is much better to have as much data as possible. We will be training a neural network on 1435 data points. Stictly speaking, this isn't a lot of data, but there isn't much that can be done about this due to the cricketing calender only having a certain number of limited-overs matches each year.

Chapter 3

DL, DLS, and Beyond

> In an ideal world, you knock the runs off and win the game.
>
> Moeen Ali

We now look at the mathematics behind both the DL, and DLS methods. Whereas DL was the original method, the DLS method was a modified and modernised version developed by Prof. Steven Stern. Although not much is taken from this method in the development of our model, it is important to look at how DLS works in order to see what our model does differently and visa-versa. DLS uses much more traditional statistics than the deep learning we employ. At their core, both models are trying to accomplish the same task, just in two very different ways.

3.1 Origins: Duckworth and Lewis

We begin by looking at the original paper [Duckworth and Lewis, 1998]. The first thing to establish is how many runs are scored, on average, in a given number of overs. This is given by the equation:

$$Z(u) = Z_0[1 - exp(-bu)] \tag{3.1}$$

Where u is the number of overs, b is an exponential decay constant, and Z_0 is the average total score in first class cricket, but with one-day rules imposed.

Because we don't have access to actual values for Z_0 or b, a plot to see what Equation 3.1 looks like was created by using 3 sample values for Z_0. For b, it was a process of trial and error to find a value that resulted in the graph having a similar shape to original figures in the DL paper. Some intuition was required here based on our own experience in cricket to find a reasonable starting value. In future, it may be worth taking the time to mine through all available data and obtain a (possibly more accurate) estimate in a similar way to their method.

However, we have not yet looked at what happens when wickets are lost. To introduce this metric, Equation 3.1 is revised to incorperate the scenario that w wickets have been lost, and that there are u overs remaining. The revised equation is given as follows:

$$Z(u, w) = Z_0(w)[1 - exp(-b(w)u)] \tag{3.2}$$

Where now, we have $Z_0(w)$ giving the average total score from the last $10 - w$ wickets in first class cricket. We now also have $b(w)$ as the exponential decay constant, which changes depending on wickets lost.

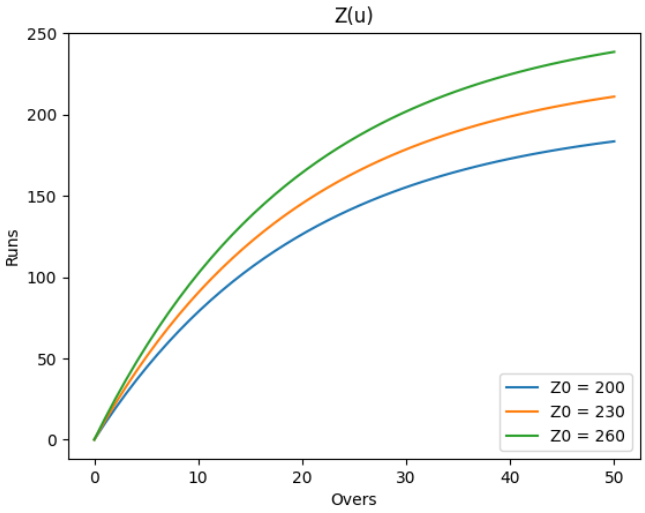


Figure 3.1: Graph showing how the rate at which runs are scored decays as a game progresses.

With this in mind, we can look at the specific case of Equation 3.2 with $u = N$ and $w = 0$, namely, the conditions at the start of an N-over innings. We have:

$$Z(N, 0) = Z_0[1 - exp(-bN)]. \tag{3.3}$$

Which we then incorperate into the ratio

$$P(u, w) = \frac{Z(u, w)}{Z(N, 0)}. \tag{3.4}$$

This ratio gives, keeping in mind there are u overs still to be bowled, with w wickets lost, the average proportion of the runs that still need to be scored in the innings. It is this ratio that is where the revised scores come from. A huge table is constructed from these ratios, and can be used to reset the target based on the state of a game. Let us now look at how that works practically.

Example 3.1.1. *Assume there is a break in the second innings (due to rain or similar), which results in the second team missing some overs. Let u_1, u_2 be the number of overs played before the break and available after it respectively. We impose the condition that $u_2 < u_1$. Further assume that at the time of the break, the second team had lost w wickets. The aim is to adjust the required score to account for the $u_1 - u_2$ overs they have lost. The winning "resources" available are given by the following equation.*

$$R_2 = [1 - P(u_1, w) + P(u_2, w)].$$

The values for $P(u_1, w)$ and $P(u_2, w)$ can be looked up from the resource table. Therefore, if the first team batting scored S runs, then the new tartget is given by

$$T = \lceil SR_2 \rceil.$$

3.2 Improvements by Stern

We now look at the DLS method, introduced in [Stern, 2016] to extend the original DL method. The motivation for DLS was that very high scoring cricket matches presented a pattern of straightening towards average run rate which is not uniform across the innings considered

(around 500). To account for this, Stern introduces an additional damping factor to the equation for DL. The updated equation is now given by:

$$Z_{dls}(u, w, \lambda) = Z_0 F_w \lambda^{n_w+1} \left\{1 - e^{-ubg(u,\lambda)/F_w \lambda_w^n}\right\}. \tag{3.5}$$

This equation assumes ther are u overs remaining, with w wickets lost by the batting team in an M-over match with a final run total of S. Here, we have:

$$g(u, \lambda) = \left(\frac{u}{50}\right)^{-(1+\alpha_\lambda+\beta_\lambda u)} \tag{3.6}$$

In the above, we define

$$a_\lambda = -\frac{1}{1 + c_1(\lambda - 1)e^{-c_2(\lambda-1)}}$$
$$\beta_\lambda = -c_3(\lambda - 1)e^{-c_4(\lambda-1)}.$$

The constants $c_1, \ldots, c_4$ are based on what Stern describes as a "detailed examination" of high scoring cricket matches.

3.3 Extension: Bayesian Modelling

This section will look at the work of [Bhattacharya et al., 2018]. Bayesian modelling was used to try and improve the score predictions of the DL method. The authors used the same dataset as we are using for this dissertation, although over a slightly smaller range of games. Only games between 2005-2017 are used, whereas our dataset goes up to 2021. Note that to keep consistent with the fact we are talking about a different model now, we swich from using $Z(u, w)$, to using $R(u, w)$, as to keep consistency with the different papers being looked at. They start by introducing the following nonlinear regression model:

$$\bar{R}(u, w) \sim N(m(u, w; \theta), \frac{\sigma^2}{n_{uw}}). \tag{3.7}$$

Where $\bar{R}(u, w)$ is the sample average of runs scored by a team from the total number of matches in the data set. We have $m(u, w; \theta)$ as the corresponding modeled population average of runs scored by a team when a considerable amount of games are taken into account. θ denotes a vector of parameters that will be specified later on. Since $R(u, w)$ is not calculated in all games, the average score is taken over all matches where scores are present, this is given by n_{uw}. The sample mean, $\bar{R}(u, w)$ is then calculated over this quantity. This is actually the point which motivates using Bayesian statistics to extend the DL method. The authors report that approximately 26.8% of values for $R(u, w)$ were missing in the dataset, so by using a Bayesian inference model, we can use the posterior predictive distribution to account for missing values. Which in turn should increase the predictive accuracy of this model.

We return now to look at the $m(u, w; \theta)$ parameter that appears in Equation 3.7. This mean function, based on the plots produced in the original DL paper, (see Figure 3.1 for an example), the authors adopted an exponential decay. We can see why this choice makes sense from Figure 3.1. The resources considered by this method, namely runs and wickets, do decrease exponentially as a game progresses. $m(u, w; \theta)$ depends on the parameters $a_w = Z_0(w)$ and $b_w = b(w)$, each one depending on the number of wickets fallen at that given time, with u overs remaining. We therefore have:

$$m(u, w; \theta) = a_w(1 - exp(-b_w u)) \tag{3.8}$$

$$\theta = \{(a_w, b_w); w \in \mathcal{W} = \{0, 1, \ldots, 9\}\} \tag{3.9}$$

Before proceeding with defining the prior specifications on the parameters for this model, we need the following distribution:

Definition 3.3.1. *$X \sim Ga(a, b)$ has a **Gamma Distribution** with mean $\frac{a}{b}$ and variance $\frac{a}{b^2}$ if its probability density function is*

$$\frac{b^a}{\Gamma(a)} x^{a-1} e^{-bx}.$$

We can now define the prior specifications on the parameters. First, we fix A_0 and B_0 large enough such that $R(50, 0) < A_0$. We initialise the model with $a_0 \sim U(0, A_0)$ and $b_0 \sim U(0, B_0)$. Then given any pair (a_0, b_0) and for $w = 0, 1, \ldots, 8$, we generate:

$$a_{w+1}|\sigma^2, a_w, b_w \sim U(0, a_w) \tag{3.10}$$

$$b_{w+1}|\sigma^2, a_{w+1}, b_w, a_w \sim U\left(0, \frac{a_w b_w}{a_{w+1}}\right) \tag{3.11}$$

$$\frac{1}{\sigma^2} \sim Ga(a, b). \tag{3.12}$$

Above, $U(a, b)$ represents a uniform distribution over the open interval (a, b). Now that the prior distribution is obtained, the likelehood function is then given by:

$$L(\theta, \sigma^2) = \left(\frac{1}{\sigma/\sqrt{n_{uw}}}\right)^{500} exp\left\{-\frac{1}{2(\sigma^2/n_{uw})} \sum_{u=1}^{50} \sum_{w=0}^{9} (R(u, w) - m(u, a_w, b_w))^2\right\} \tag{3.13}$$

The authors chose their parameters $A_0,\ B_0,\ a$ and b such that the prior is not sensitive to the posterior inference. The values chosen were therefore $A_0 = 200$ $B_0 = 100$ and $a = b = 0.1$.

One thing briefly mentioned in chapter 3 of their paper is the average score at the fall of each wicket. This isn't discussed further, but it's interesting to see how it behaves. So we take a brief detour to look at it. This can be seen in figure 3.2.

Initially, one may think that this is normally distributed. It certainly holds a bell curve-like shape. However, a Kolmogorov-Smirnov test [Massey Jr, 1951] was performed to see if these values were normally distributed, and it turns out they aren't. This test was performed using the R function *ks.test()*, which returned a p-value of $2.2\text{x}10^{-16}$ for the two-sided alternative hypothesis parameter.

The shape of this curve makes sense from a cricket standpoint because usually batters 3 and 4 are actually the best in the side, so the fact they put on more runs is not a suprise. But what this does do is reinforce the idea behind using an exponential decay for modelling resources. This is because cumulatively, the "tail end" (last 4) of batters will not put on as many runs as the higher/middle order.

Most of the work from this paper went into creating a better resource table for calculating revised scores. Which isn't particularly relevant to this project, but what is is the section on score prediction. The authors split every match at the 35^{th} over to try and predict the final score. They found the bayesian model gives a better prediction of final match scores. Figure 3 of their paper outlines these results.

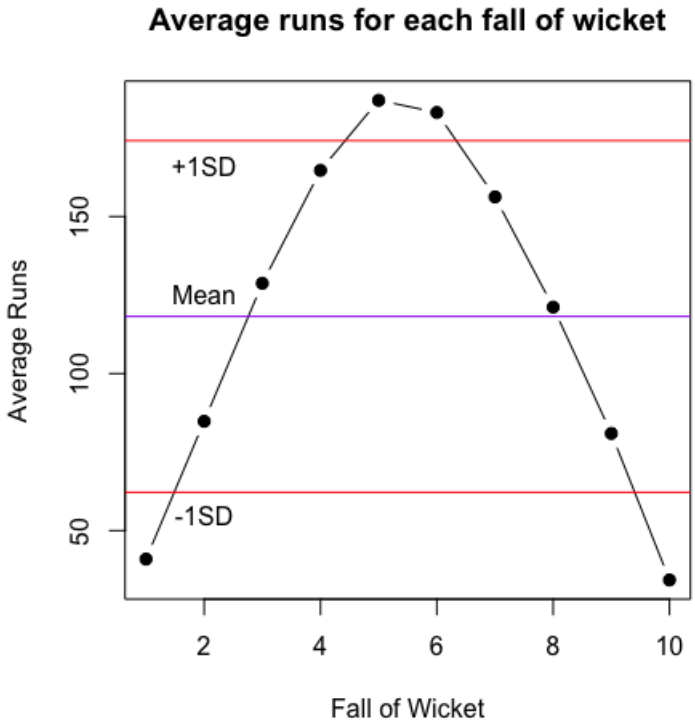


Figure 3.2: Plot of average runs scored at the fall of each wicket. Averages taken over 1437 innings.

Chapter 4

Exploratory Data Analysis

> Cricket often leaves you scratching your head
>
> James Anderson

The purpose of this chapter is to explore the obtained dataset in more detail, which is necessary for carrying out the work in later chapters. In general, and no less in the world of statistics, making assumptions is dangerous, and so in order to make the assumptions we do in later chapters, we must have the evidence to back it up. This chapter not only provides that evidence, but allows us to become more familiar with our dataset, and see how modern data fits in with the previous work done in this field.
We begin by looking at the probability densities in the Fall of Wicket variables. F.o.W is key in the DLS method, so we do this to get an idea of what lies underneath the surface. We are able to explain the way F.o.W distributions are shaped based on the way cricket games unfold. This consistency allows us to make an assumption about Run Rates later in the chapter too.
As with any dataset, there are outliers, and the number of such will have influence on the error function that we use in later models. For that reason, we give brief discussion to this, before going on to look at whether or not scores in games of cricket are normally distributed around some mean.
Run Rates are the discussed in more detail, looking at whether or not the runrates in certain periods of the game also follow a normal distribution. Finding this out is imperative for using Monte-Carlo simmulation later on in the paper.

4.1 Fall of Wicket Densities

In this chapter, we are looking only at the first innings of the games, and only those games in which the full 50 overs were played. The reason for this is the models we will build are going to try and predict a score as if a full innings has been played. We begin our exploration of the data with a look at how the density of the runs scored per fall of wicket changes. This has been done for each individual team in the dataset, and in Figures 4.2-4.6, we can see how this evolves.

In Figure 4.2, we see the density is heavily skewed to the left. This makes sense, as the bowling team will presumably be starting their innings by using their best bowlers, who will be hunting to get wickets early on. In Figure 4.4, we see a much more normally distributed density function. But in actual fact, we see this interesting second, smaller peak appearing lower down in the score. Does this make sense? It's certainly not suprising. What these two peaks exemplify is the fact games can go heavily in favour of the bowling team, which can be seen in the first small peak, wherein they have taken a lot of wickets in quick succession, meaning the

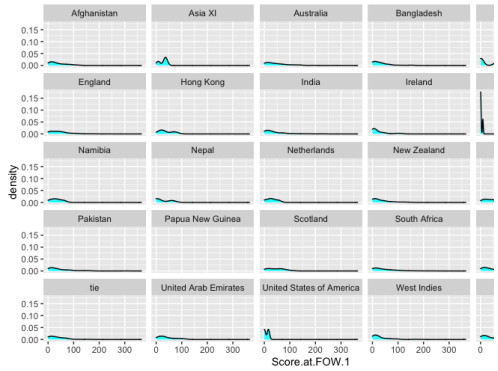


Figure 4.1: Density of all teams for first wicket falling

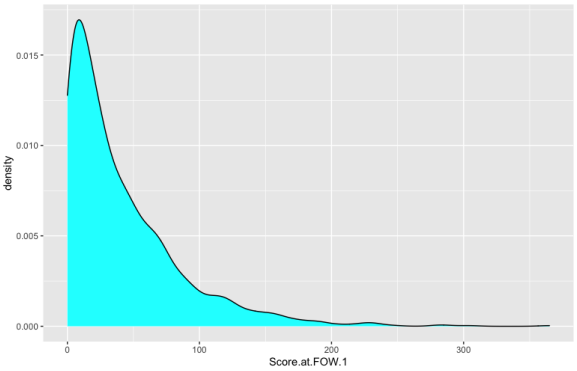


Figure 4.2: Overall density plot for FOW 1

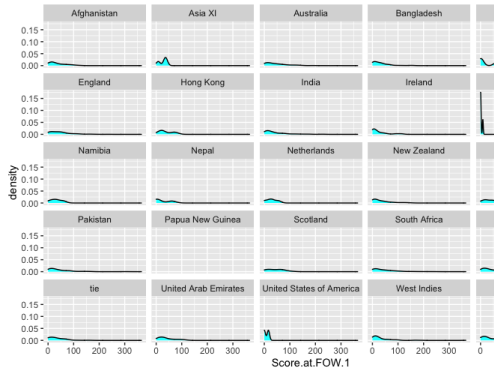


Figure 4.3: Density of all teams for fith wicket falling

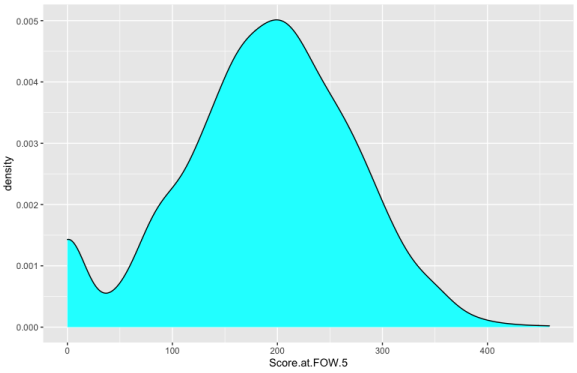


Figure 4.4: Overall density plot for FOW 5

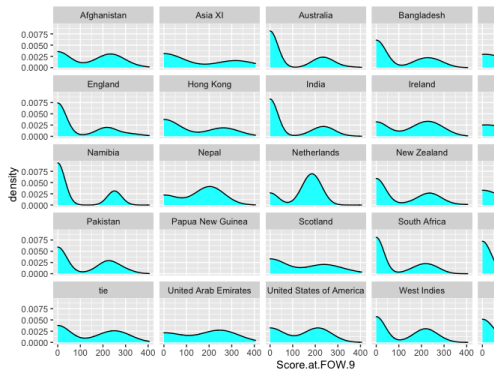


Figure 4.5: Density of all teams for ninth wicket falling

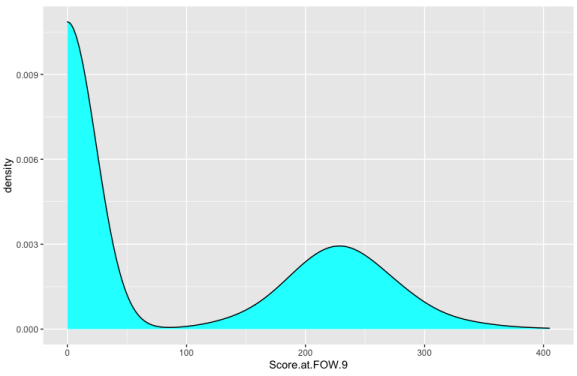


Figure 4.6: Overall density plot for FOW 9

lower order batters are coming in to bat earlier than usual. Secondly, it shows when the batting team is having a good day, because we have this much larger peak around the 200 runs mark.

Finally, in Figure 4.6, we can see that the earlier bowling advantage peak is much higer, because the lower order batters are traditionally less skilled at batting, and so the bowling team have a distinct advantage in taking wickets against these players. But we also see the second, batting-favoured peak is no much lower. This corresponds to the scenario in which the earlier batters have laid a good foundation of the game, and the lower-order batters have not had to contribute much to the score.

4.2 Outliers in Runs Data

In the next chapter, it will be necessary to choose a loss function for improving the neural network that we create. To aid in determining which function to use, we need to look at the spread of runs scored in a full innings of data. This can be seen in Figure 4.7.

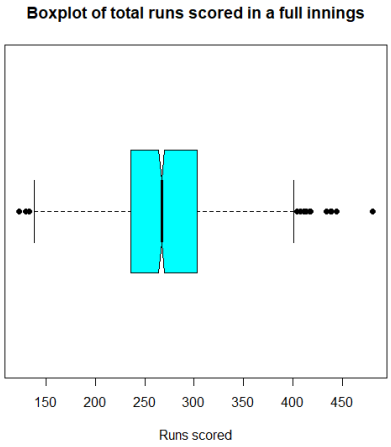


Figure 4.7: Boxplot showing the spread of runs scored

There are 363 data points greater than the third quartile, while 671 are below the first quartile. So 25.3% of our data lies above the third quartile, and 46.7% below the first. For that reason, we make the decision to use the Mean-Squared-Error (MSE) loss function, which is commonplace in regression problems.

4.3 On the Normality of Run Totals

It will be useful in later parts of this dissertation to know whether or not scores are normally distributed. To test whether or not they are, we use a Q-Q plot to check. This is a graphical way for checking normality, by comparing the quantiles of a dataset (in this case, scores) to quantiles drawn from a theoretical distribution. If the resulting points follow a straight line, it is likely that the data came from the distribution. In this case, we use the R function *qqnorm()* to test if the runs from the 1436 games of a full 50 overs follow a normal distribution. See Appendix A for a more detailed discussion of this method.

We can see from Figure 4.8 that the plot follows along the straight line and so we can conclude that the scores are infact normally distributed. Further, we can calculate the parameters of this distribution using the R functions *mean()* and *var()*. Doing so gives that the distribution of scores in 50 overs, S_{50}, can be given as $S_{50} \sim \mathcal{N}(270.56, 51.34^2)$.

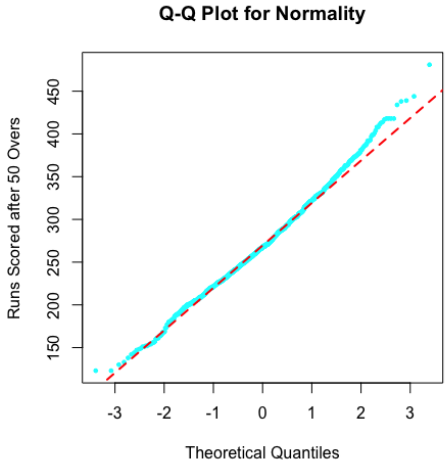


Figure 4.8: Q-Q plot for runs scored after 50 overs.

To further test that this is indeed the case, we can create a sample plot based on this distribution, which can be seen in Figure 4.9

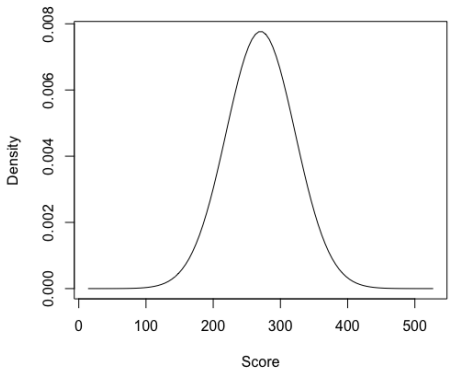


Figure 4.9: Sample plot created from the derived distribution of S_{50}

With this in mind, we can now look at a density plot for the actual data. This is given in Figure 4.10.

It is unsuprising that runs are normally distributed, but to be able to draw a mean and variance from this will be very helpful in future aspects of the project.

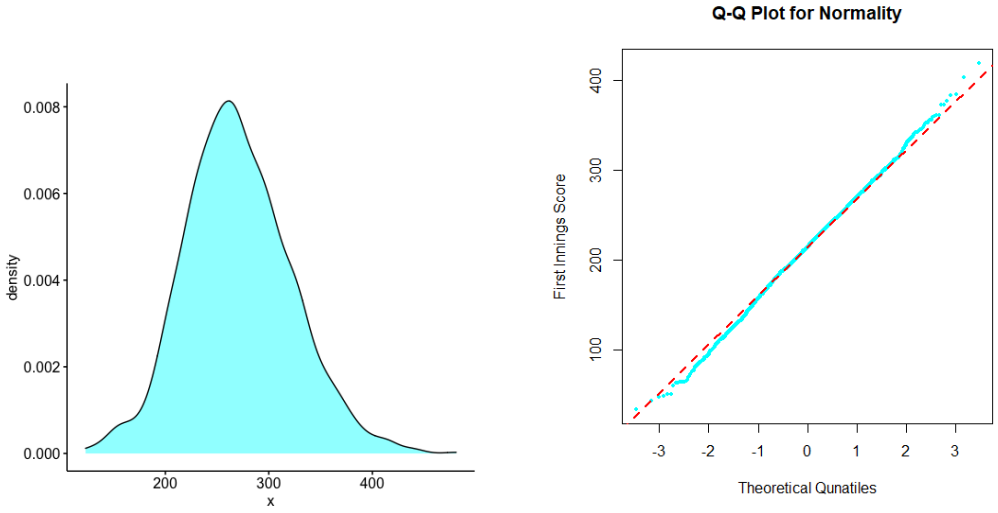
We have seen that first innings scores in a full 50 overs are normally distributed. We can check, using the same methods if runs in a first innings that doesn't necessarily go for the full allowance of overs is normally distributed.

We find that $S_{FI} \sim \mathcal{N}(213.49, 56.91^2)$.

4.4 Run Rates

4.4.1 General Exploration

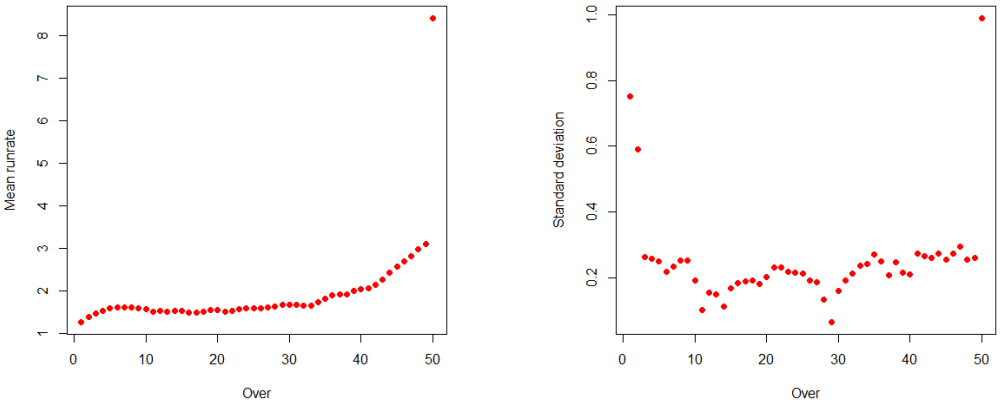
The work that follows in this section is essential for allowing the Neural Network constructed in Chapter 5 to predict the scores of games. The aim of this section is to see if we can draw the run rates at specific overs from a statistical distribution. This will in turn enable us to fill the gaps in the missing overs of a game. In turn, with the gaps filled, we can pass the simmulated



(a) Density for Runs Scored (b) Q-Q Plot for First Innings

Figure 4.10: Error Density and QQ for 50-over Data

run rates to the neural network, and allow for a score to be predicted.



(a) Average run rate per over (b) Run rate Standard deviation per over

Figure 4.11: Error distribution with Q-Q plot

We can see from Figure 4.11, that average Run Rate seems to stay consistent in the middle overs, before rapidly increasing as the risk associated with losing a wicket falls off due to the end of the innings coming closer. If we break the game into five ten-over segments, as these are generally different periods of the game from a tactical perspective[1], we can model the segments. A game of ODI-cricket can roughly be broken up to three segments, the first 10 overs, known as the "powerplay", where teams are slightly more conservative and look to build a foundation on which the rest of the game is built. The middle overs are 11-35. The idea of this game is to continue building on the foundation, and save wickets. The last 15 overs are where more risks are taken, as the value of a wicket decreases. The rule of thumb for the batting team is to double your score from the first 35 in the last 15, which is why we see the massive increase in

[1]This is to do with different bowling options being used to be more effective as the pitch conditions change

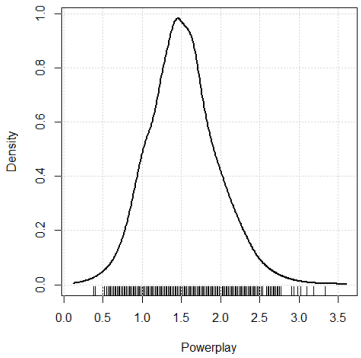


Figure 4.12: Powerplay Run Rate density

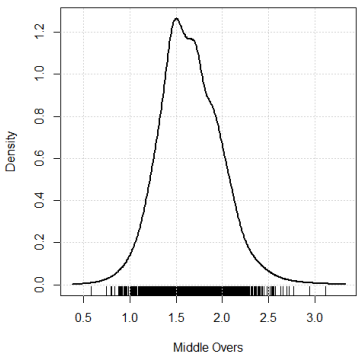


Figure 4.13: Middle Overs Run Rate density

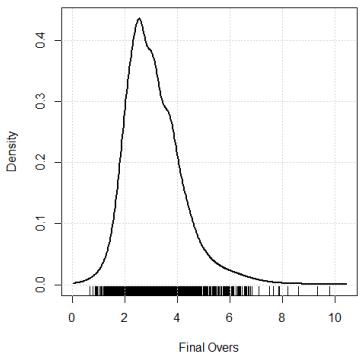


Figure 4.14: Final Overs density

run rate in these last overs.

Using this information, we can create density plots of the average Run Rate in these overs, as in figures 4.13-4.15.

The run rates are in different stages of the game roughly normally distributed, which can be seen in Figure **??**. It is reasonable that due to the Central Limit Theorem, with more observations, these distribiutions would be smoothed out and appear more normally distributed than it does at present. This is imporant, because it allows us to investigate using a monte-carlo method to fill in gaps later on in the project.

4.4.2 Feature Selection

Feature selection is an important aspect in data science. Some experiments can have an overwhelming number of variables, which can lead to computational inefficiency. In this subsection, we will explore selecting the most important overs in the run rate data. Given the lack of data already available, this may seem unnecessary, however it is an interesting consideration for future work. When we build the network in the next chapter, the model has to be passed a formula, telling it how to treat the variables. By default, since we have 50 overs worth of data (V1 $\rightarrow$ V50) and a variable containing the final score, V51, we pass the R formula $V51\ V1 + \ldots + V50$. The purpose of this section is to see if it is worth reducing this somewhat combersome formula. LASSO, **L**east **A**bsolute **S**hrinkage and **S**election **O**perator, is a regression method for performing both normalisation, and feature selection. The term LASSO was introduced by [Tibshirani, 1996]. Before showing how we implement this in R, and what it means for our project, it is useful to discuss the mathematical foundation of the method, as in section 2.1 of Tibshirani's paper.

For $i = 1, \ldots, N$, we have a dataset $(\mathbf{X}^i, y_i)$. For the training dataset we use, $N = 1436$. Specifically, we define $\mathbf{X}^i = (x_{i1}, \ldots, x_{ip})$ to be the vector of predictor variables (for us, $p = 50$). Further, y_i is the response variable. The assumption that these variables are independent is immediately satistfied by the nature of our study, since the runs scored in one over does not depend at all on the runs scored in the prior over. In addition, our data is normalised when it is passed the neural network, and so the normalisation assumption is also satistfied by default.

Definition 4.4.1. *We begin by letting* $\beta = (\hat{\beta}_1, \ldots, \hat{\beta}_p)^T$*, then the* ***LASSO estimate*** $(\hat{\alpha}, \hat{\beta})$*,*

for a tuning parameter $t \geq 0$ *is given by*

$$(\hat{\alpha}, \hat{\beta}) = argmin \left\{ \sum_{i=1}^{N} \left(y_i - \alpha - \sum_j \beta_j x_{ij} \right)^2 \right\}$$

Suject to $\sum_j |\beta_j| \leq t$.

We have that $\forall t$, $\hat{a} = \bar{y}$. However since the data is normalised, by defintion $\bar{y} = 0$ and so we can ignore the parameter α altogether. Although not discussed directly here, the authors do go on to propose algorithms for computing solutions to this problem in chapter 6 of their paper. These algorithms are employed directly by the R package `glmnet`, which we can now use.

We use the function *cv.glmnet()*. This can be seen in the following code snippet.

```
lassoModel <- cv.glmnet(X,y,standardize=T,type.measure="mae")
plot(lassoModel)
```

Figure 4.15: Fitting a LASSO model

The plot produced by this code can be seen in Figure 4.16.

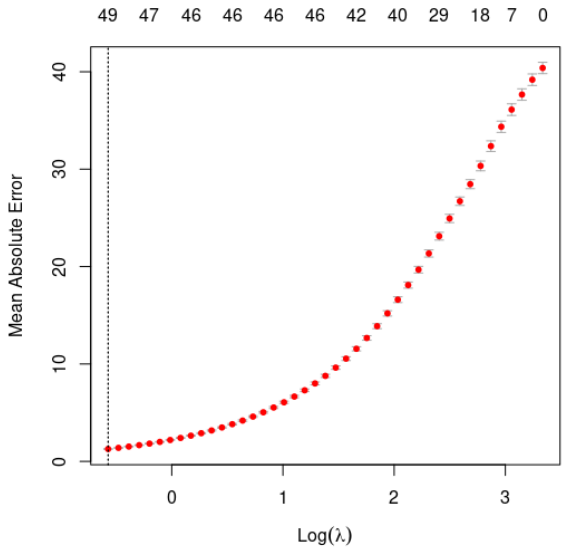


Figure 4.16: LASSO Model Plot

The red dots here show the mean absolute error when the number of variables used in the model. We can see from this plot that using all of the overs results in the lowest MAE. Seeing which variables contribute the most can be done using the code in Figure 4.17.

```
df_coef <- round(as.matrix(coef(lassoModel, s=lassoModel$lambda.min)), 2)
order(df_coef[df_coef[, 1] != 0, ])
```

Figure 4.17: Contributing Variables

This code orders the variables depending on how "important" they are. This ordering is done by the minimum lambda value. As it turns out, the 5 most important overs are 31, 46,

35, 38 and 48. Unsuprisingly, the least important is over 1. This is more than understandable from a cricketing standpoint, since a lot can change after the first over, whereas the later overs can make or break an innings.

Chapter 5

Pattern Recognition with Neural Networks: Background and Implementation

> Nowadays, with technology coming into cricket, people start to analyse, and if you only have one or two tricks, people will start to line you up
>
> Jasprit Bumrah

We begin this chapter by looking at the mathematical background of neural networks. More specifically, the construction of them using Linear Algebra, and then algorithms for training them. By "training", we mean updating the parameters associated with the network in order to improve their predictive accuracy. After this mathematical discussion, we look at using the `neuralnet` package in R to implement a network, although the resulting network is dicussed in the next chapter. The chapter finishes with a discussion of time complexity of the algorithms used.

5.1 A Brief Introduction to Neural Networks

Neural networks (NNs) have been the subject to a lot of discussion and exploration in recent years. They are a machine learning method that is being applied to many problems in all sorts of fields, such as finance [Naeini et al., 2010] and medicine [Ganesan et al., 2010]. The network will be trained on run rate data, so for each game we have calculated the evolution of the run rate, and then we have the overall score for that game in the final column of the matrix. An example of a network can be seen in Figure 5.1. The first layer is the input layer, and the last layer gives the predictions. The middle layers are hidden and where the work of the network is done.

We see there are p input nodes at the bottom, M nodes in the hidden layer, and K output layers. The lines between the layers are given a "weight" and a "bias". These properties will be discussed in more detail shortly. The number of hidden layers to be used will be the subject of experimentation. It's important to note how we will measure the success of the model. The main dataset, consisting of 1438 games of 50 overs is split into two subsets, one containing 70% of the games, and one containing 30%. We use the larger of the two to "train" the model, and the smaller one as a test set. Not doing this could lead to a phenomenom known as "overfitting"; where the network learns how to predict patterns in the dataset it was trained on really well,

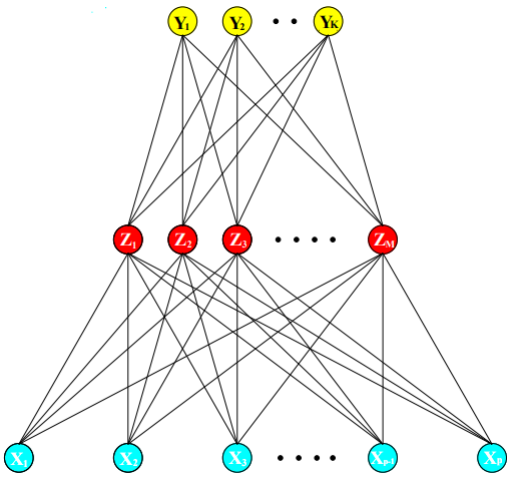


Figure 5.1: Example of a single hidden layer neural network, as in [Friedman, 2017]. Inputs are inputted through the bottom layer, the final activity value in the top layers are taken as the prediction.

but then isn't so good at predicting unseen values. We use corrolation as the main metric of success in this model, which will be discussed later on.

As mentioned, NNs have been used extensivelly in finance- more specifically, in predicting future stock price behaviour. Naturally if one can predict how the value of a stock will change over some time period, then one can protect themself from a bad investment, or profit heavily from a good one. We will use similar methods here for building our neural network. In [Naeini et al., 2010], the authors test two different Neural Netowrks, and find that using a "Multi-Layer Feed Forward Nerual Network" is the better choice for predicting how stock values will change. With these motivations in mind, we can begin to construct the networks.

5.2 Builiding The Network

Our input layer will have 50 nodes, one for the run rate at the end of each over. We will begin with 5 hidden layers, although this is subject to change. The output layer will of course only have one node, the value of which will predict the score of the game.

We begin by looking at a single node. The proper name for each node is "perceptron". Each perceptron takes in the values of a vector, and an extra "+1" intercept term. The perceptron then outputs a value h given by Equation 5.1:

$$h_{W,b}(x) = f(\mathbf{W}^T x) = f(\sum_{i=1}^{K} W_i x_i + b). \tag{5.1}$$

Here, $K \in \mathbb{N}$ is the number of elements in the vector, and $f : \mathbb{R} \to \mathbb{R}$ is called the activation function. There is a bit of choice in which activation function to use. The two common choices are:

$$f(z) = \frac{1}{1 + exp(-z)} \tag{5.2}$$

$$f(z) = tanh(z) \tag{5.3}$$

The comparrison of these two functions can be seen in Figure 5.2.

We will be using Equation 5.2 as our activation function. There is some leniancy over which activation function is used. The paper [Sibi et al., 2013] found that actually the more important

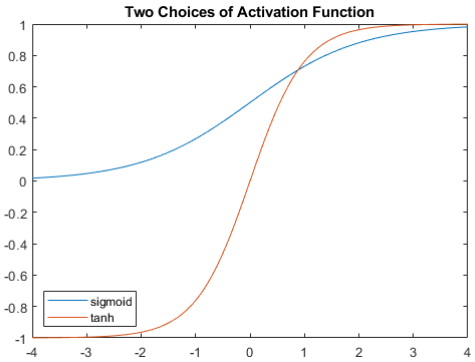


Figure 5.2: Graph showing the shape of different activation functions.

aspect of the network is the choice of training algorithm, along with other parameters such as learning rate α. For now, lets look at the network we're going to use.

We begin with getting from the input layer, denoted L_0. Each node in L_0 takes a value $a^{(0)_i}$ for $i \in \{1, 2, \ldots, 50\}$. We use $\mathbf{a}^{(0)}$ to denote the vector containing these values. For every node in L_0, we have 25 connections coming away, one going to each of the nodes in L_1. 25 was an arbitraty choice for the size of L_1, and is subject to change based on results from initial testing. So that means we have $50 \times 25 = 1250$ weights for just the first layer alone. Some linear algebra is to be done here to get values for the vector $\mathbf{a}^{(1)}$. We use w_{ij} to denote the weight from node j in one layer to node i in the prior layer. We can then construct a matrix containing the weights between L_0 and L_1, which we denote $W^{(1)}$. This matrix is given by 5.4.

$$W^{(1)} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & a_{1,50} \\ w_{2,1} & w_{2,2} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{25,1} & a_{25,2} & \cdots & a_{25,50} \end{bmatrix} \tag{5.4}$$

Using 5.4, and combining with $a^{(0)}$, we can get the following:

$$A^{(1)} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & a_{1,50} \\ w_{2,1} & w_{2,2} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{25,1} & a_{25,2} & \cdots & a_{25,50} \end{bmatrix} \times \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_{50}^{(0)} \end{bmatrix} + \begin{bmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_{25}^{(0)} \end{bmatrix} \tag{5.5}$$

$$= W^{(1)}\mathbf{a}^0 + \mathbf{b}^1 \tag{5.6}$$

Where $\mathbf{b}^1$ is the vector containing the biases for each node. At this point, we are incredibly close to having values for $\mathbf{a}^{(1)}$. The last thing to do is apply the activation function. The above has resulted in a 25×1 vector, we use the notation $f(A^{(1)})$ to denote applying the activation function to each element in the vector $A^{(1)}$. Putting this all together, we have the following equation:

$$\mathbf{a}^{(1)} = f(W^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^1) \tag{5.7}$$

Which gives us values for all the nodes in the first hidden layer. This process is then repeated for each layer, creating a diffferent weights and bias matrix going between each layer in the network. These matrices are naturally of different sizes, but the principle is exactly the same. When we start the training process for this network, we will only have random variables for the weights and biases. To gain better values that can predict accurate results in the future,

we need to train the network by feeding it feature vectors and their associated outputs.

To allow the netowrk to be trained, we first must define a "cost function". The way we do this is to input a vector to the input layer, let the network produce a result, say y', which initially is likely to be quite wrong, and square the difference between this and the true value y for that particluar training vector. Put more precisely, let $\mathbf{X}$ be a feature vector containing 50 elements, and y the corresponding output. Then define the cost function, $C(X, y) := (y' - y)^2$. The lower the value of $C(X, y)$, the better the network has done at predicting. The average value of $C(X, y)$ for each X and y in our training set, is then a good measure of the network's performance.

The cost function is at the heart of how these networks "learn". All we're doing is minimising a cost function, to give matrices of weights and biases that produce the best output. The algorithm for minimising this cost function is called "gradient descent" [Cauchy et al., 1847]. Suppose we take every single weight and bias of our network, and turn it into a giant column vector.

Example 5.2.1. *In this first example, we initialise four random weights matrices, four bias vectors, and we run through the process outlined above. We know going into this that the cost is going to be high. Initialising the weights as random values $w_{ij} \in [-1000, 1000]$, we get an initial cost function for our full dataset of a huge 3098.4.*

5.2.1 Training the Network: Gradient-Descent and Backpropogation

We discussed in section 4.2 the need to use a MSE loss function based on our data. Let us now define this function, as in [Huber, 1992].

Definition 5.2.2. *Let N be the number of datapoints, y_i' be the prediceted value from the i^{th} datapoint, and y_i be the true value. Then for the input vector X, and a parameter θ, define the* ***Mean Square Error*** *by*

$$E(X, \theta) = \frac{1}{2N} \sum_{i=1}^{N} (y_i' - y_i)^2 \tag{5.8}$$

The method of *Gradient Descent* necessitates calculating the gradient of $E(X, \theta)$ with respect to weights and biases in each layer. The idea is to find a local minimum, as this will give a set of weights and biases for which the error is lowest, and such that the network is giving well approximated results. Define a *learning rate* α, and incorperate the set of weights and biases into the parameter θ. Then we update the weights and biases each iteration t via the relationship 5.9

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(X, \theta)}{\partial \theta}. \tag{5.9}$$

With this in mind, we can now start to go through the Backpropogation process. Firstly, we need to calculate the partial differential of E with respect to a given weight w_{ij}^k. This calculation is given as follows:

$$\frac{\partial E(X, \theta)}{\partial w_{ij}^k} = \frac{1}{2N} \sum_{d=1}^{N} \frac{\partial (y_d' - y_d)^2}{\partial w_{ij}^k} \tag{5.10}$$

$$= \frac{1}{N} \sum_{d=1}^{N} \frac{\partial E_d(X, \theta)}{\partial w_{ij}^k}. \tag{5.11}$$

In the above, we have $E_d = \frac{1}{2}(y'_d - y_d)^2$. We must emply the chain rule to calculate the partial derivavtive of E_d with respect to an individual weight. Let a_j^k be the activation value of node j in layer k before it is put through the activation function. Then we have

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k}. \tag{5.12}$$

In 5.12, the first derivative on the righthand side, is known as the *error*, often denoted as δ_j^k. The second derivative on that side is the *output* of node i in layer $k-1$, denoted o_i^{k-1}. So we can simplify 5.12 to be written as

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}. \tag{5.13}$$

The aim of backpropogation is to minimise the error δ_1^m, where m denotes the final layer. We can express E_d in terms of a_1^m as follows:

$$E_d = \frac{1}{2}(f(a_1^m) - y_d)^2. \tag{5.14}$$

Where f is the activation function. We have:

$$\delta_1^m = (y' - y)f'(a_1^m). \tag{5.15}$$

So in the final layer, we have:

$$\frac{\partial E_d}{\partial w_{i1}^m} = \delta_1^m o_i^{m-1} \tag{5.16}$$

$$= (y'_d - y_d)f'(a_1^m)o_i^{m-1}. \tag{5.17}$$

The above is all well and good for the final layer, but we now must consider the hidden layers. We will consider the general case here, and then plug in the relevant numbers for our model when it comes to the implementation later on. For a layer k such that $1 \leq k < m$, the error term δ_j^k is given by:

$$\delta_j^k = \frac{\partial E_d}{\partial a_j^k} \tag{5.18}$$

$$= \sum_{l=1}^{r^{k+1}} \frac{\partial E_d}{\partial a_l^{k+1}}\frac{\partial a_l^{k+1}}{\partial a_j^k}. \tag{5.19}$$

In the above, r^{k+1} is the number of nodes in the next layer, and $l = 1, 2, \ldots, r^{k+1}$. Since we can write $\delta_l^{k+1} = \frac{\partial E_d}{\partial a_l^{k+1}}$, the above can be written as

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1}\frac{\partial a_l^{k+1}}{\partial a_j^k}. \tag{5.20}$$

Since the activation of a layer in one node is the sum of weights and avtications in the prior layer, we write:

$$a_l^{k+1} = \sum_{j=1}^{r^k} w_{jl}^{k+1} f(a_j^k). \tag{5.21}$$

Therefore,

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} f'(a_j^k). \tag{5.22}$$

Plugging the above in 5.20, we obtain the following:

$$\delta_j^k = f'(a_j^k) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}. \tag{5.23}$$

The equation 5.23 is known as the *Backpropogation formula*. The final step is to calculate, $\frac{\partial E_d}{\partial w_{ij}^k}$, which is given by:

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} \tag{5.24}$$

$$= f'(a_j^k) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}. \tag{5.25}$$

5.2.2 RPROP+

Backpropogation is a powerful and widely used algorithm, In implementing the neural network in this project, we used a variation of it, known as the "RPROP+" algorithm [Riedmiller and Braun, 1993], short for "**R**esiliant back**PROP**ogation". The + is notation for showing we include backtracking in the algorithm. In Algorithm 2, we give the algorithm in detail.

Algorithm 2 RPROP+

for all eights and biases **do**
  if $\frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) > 0$ **then**
    $\Delta_{ij} = \min(\Delta_{ij}(t-1) \times \eta^+, \Delta_{\max})$
    $\Delta w_{ij}(t) = -\text{sign}\left(\frac{\partial E}{\partial w_{ij}}(t) \times \Delta_{ij}(t)\right)$
    $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$
  else if $\frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) < 0$ **then**
    $\Delta_{ij} = \min(\Delta_{ij}(t-1) \times \eta^-, \Delta_{\min})$
    $w_{ij}(t+1) = w_{ij}(t) - \Delta w_{ij}(t-1)$
    $\frac{\partial E}{\partial w_{ij}}(t) = 0$
  else if $\frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) = 0$ **then**
    $\Delta w_{ij}(t) = -\text{sign}\left(\frac{\partial E}{\partial w_{ij}}(t) \times \Delta_{ij}(t)\right)$
    $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$
  end if
end for

What makes RPROP "resiliant" is the intrduction of an invividual update value Δ_{ij} for each weight w_{ij}. It is this quantity alone that determines how much the weight will be updated by. The learning rule is given by

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ \times \Delta_{ij}^{(t-1)} & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ \eta^- \times \Delta_{ij}^{(t-1)} & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ \Delta_{ij}^{(t-1)} & \text{otherwise} \end{cases} \tag{5.26}$$

As mentioned previously, the + refers to having backtracking in the algorithm. It is helpful to formally define this.

Definition 5.2.3. *Suppose the partial derivative* $\frac{\partial E}{\partial w_{ij}}$ *changes sign. This means the prior step was too large, meaning the minimum is missed. For that reason, we* ***backtrack*** *to the previous weight update. I.e,*

$$\Delta w_{ij}^{k} = -\Delta w_{ij}^{(k-1)} \; if \; \frac{\partial E^{(k-1)}}{\partial w_{ij}} \times \frac{\partial E^{k}}{\partial w_{ij}} < 0$$

5.3 Implementing the Neural Netowk

It was decided to implement the neural network using the R programming language, and more specifically, the `neuralnet` package by [Günther and Fritsch, 2010]. This package allows us to specify all the parameters we wish for, and automatically performs the Backpropogation algorithm to train a network. The whole aspect of this process is the encapsulated in line 4 of Figure 5.3. It was decided to use a prebuilt package for this rather than building it ourself since the code for this package has already been optimised, and so will be more efficient than an implementation we come up with. Since we would have to take time out to optimsie the code and this would detract from the purpose of this project.

Recall the logistic function, $\sigma : \mathbb{R} \to (0, 1)$, defined by:

$$\sigma(x) = \frac{1}{1 + exp(-x)}.$$

For extremely positive or negative values of x, σ will return -1 or 1, so we must normalise the data before training the neural network. Once the data were normalised, several versions of the network were ran with varying parameters. The final parameters decided were a learning rate, $\alpha = 0.02$, and a hidden network of layers consitsting of 25 neurons, two layers of 10 neurons, and a final hidden layer of 3 neurons. In total, 1000 iterations were ran, each randomly initialising neuron values by drawing from the $N(0, 1)$ distribution. The code for this can be seen in Figure 5.3.

```
1 #Put the network into a function for ease of access
2 runNet <- function(hiddenLayer, reps,alpha){
3   message("STARTING TRAINING")
4   scoreNet <- neuralnet(formula, data=trainNorm, act.fct = "logistic", hidden =
      hiddenLayer, linear.output=T,rep = reps,
5                         stepmax = 1e+12, learningrate=alpha, lifesign = "minimal"
    ,algorithm="rprop+",err.fct = "sse")
6   message("TRAINING FINISHED")
7   return(scoreNet)
```

Figure 5.3: R code to implement the neural network

A plot of this network can be seen in Figure B.1. Implementing the network this way is fairly efficient, and does not cause any issues with time-complexity. Infact, Backpropogation was found to have a median time complexity of $\mathcal{O}(N^4)$ in [Lister and Stone, 1995]. Running 1000 training iterations took 10 hours on an *AMD-Ryzen 5 1600 Six-Core Processer @ 3.20Ghz* with 16GB of available RAM.

More details on the results of the network will be available in the next chapter, although to outline how we obtained the data there, it is first a case of selecting the training repitition with the lowest error on the training data. This is done using the *which.min()* function in R,

which looks through all the data in the vector we give it, and returns the index of the minimum value. We then use the function *neuralnet::compute()* to use the neural network for predicting the (normalised) values in the training data. These predicted values are stored in a dataframe next to the original values. We then perform a simple "Root Mean Squared Error" calculation by $\sqrt{(x_{original} - x_{predicted})^2}$, and use this as the third column in the dataframe. This completes the implementation of the neural network. A plot of the final network can be seen in Appendix B.

Chapter 6

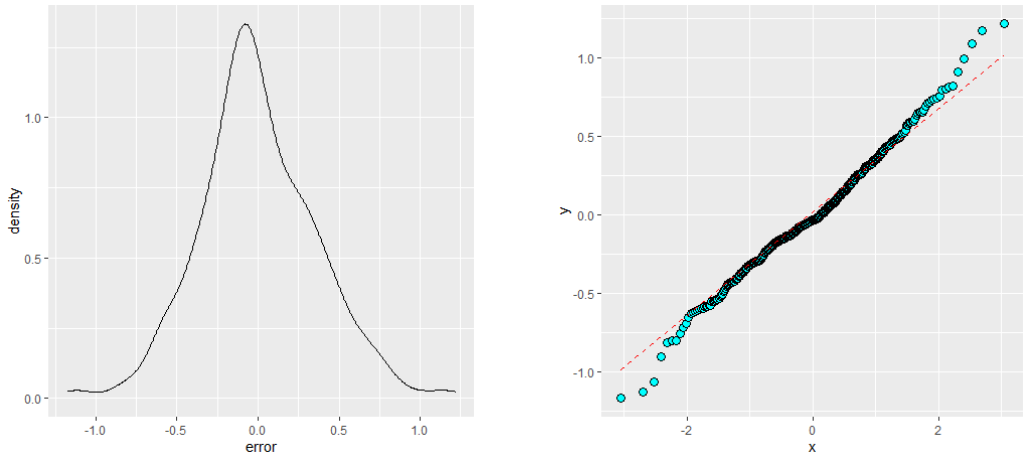
Model Analysis

> The more we play cricket, the more players will learn from it
>
> Inzamam-ul-Haq

This chapter is dedicated to analysing the results of the neural network model implemented in the prior chapter. We first look at how the model did on predicting the test dataset that is in exactly the same form as the training data. We then give a brief mathematical interlude on Monte-Carlo simmulation before using this to create a simmulated dataset to test the model on. Finally, we give the network actual DLS game data to see how it does. In each of the cases, we perform similar analysis to compare the performance of the model. The main metric is the corrolation between actual results and those predicted by the network.

6.1 Fully Trained Neural Network

The first thing to look for is the normality of the errors in the predicted values. If the errors are normally distrubted (and ideally around 0), then it means that our predictions are sufficiently accurate. We produce a density plot of the errors using the `ggplot2` package in R.



(a) Distribution of Errors

(b) Q-Q Plot of Errors

Figure 6.1: Error distribution with Q-Q plot

As we can see in this figure, there is a general bell curve, but not quite perfect as we have a bump between -0.5 and -0.75, as well as between 0.125 and 0.5. This non-normality is reflected in the Q-Q plot of the error. Measuring accuracy of the results is harder for these value

prediction networks is harder than in classification problems. This is because we can't construct a prediction matrix. We don't expect the network to predict values down to the exact run, this would require a lot more data than is available, and a large amount of experimentation. One metric we can therefore use to see how accurate our model is, is to calculate the corrolation between the actual results and the predicted results. Using the inbuilt *cor()* function, we obtain a corrolation value of 0.9382. Given how close this is to 1, which would be perfect corrolation, it is fair to say that this method has done well to predict scores.

The issue that one may point out here is that this network has been trained on a full-over dataset, and then been tested on a full-over dataset. But the purpose of this investigation has been to look at the scenario in which a full game hasn't been completed. So the method is currently ineffective at doing the task it set out to solve. For this reason, we must come up with a way to 'fill in' the missing overs. The idea for this is to use Monte-Carlo simmulation. We discussed in 4.4 how depending on the stage of the game, the runrates can be drawn from one of three normal distributions. So for the overs that are missed, we can simply fill in the gaps by drawing a value from the distribution that the missing over falls into.

6.2 Interlude: Monte-Carlo Simmulation

Before simmulating cricket games to test our network on, we first find it necessary to delve into the mathematics of the methods used to for doing the simmulating. This is where "Monte-Carlo" simmulating comes in.
Let H be some random variable. At this stage, the distribution of H is irrelevant, but we note that $\mu = E(H)$. Formally, we have the following definition.

Definition 6.2.1. *Let $n \in \mathbb{N}$ and let $\{H_i \mid i = 1, \ldots, n\}$ be i.i.d copies of H. The **Monte-Carlo Estimate** of μ, is given as $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} H_i$.*

Recall the well-known *Strong Law of Large Numbers.*

Theorem 6.2.2. *Let $n \in \mathbb{N}$, and let H_1,H_2,... be an i.i.d sample from a distribution with expectation μ and standard deviation σ, with μ, $\sigma < \infty$. Then*

$$P\left(\lim_{n\to\infty} \bar{H}_n = \mu\right) = 1$$

It follows from Theorem 6.2.2 that

$$\lim_{n\to\infty} \hat{\mu} = \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} H_i \tag{6.1}$$

$$= E(H) \tag{6.2}$$

$$= \mu. \tag{6.3}$$

6.3 Match Simulation

With the theoretical framework for Monte-Carlo simmulation established, we can now look to build an algorithm for simmulating cricket matches. Based on our own work in Chapter 4, we begin by defining three random variables, $R_{\text{powerplay}} \sim N(\mu_{\text{powerplay}}, \sigma_{\text{powerplay}})$, $R_{\text{middle}} \sim N(\mu_{\text{middle}}, \sigma_{\text{middle}})$ and $R_{\text{final}} \sim N(\mu_{\text{final}}, \sigma_{\text{final}})$.

The numerical values for these are given in the Table 6.1.

The code for doing this simmulation was not hard to write, and after a few small performance enhancements ran almost instantaneously. The code can be seen in figure **??**.

Numerical Values	μ	σ
$R_{powerplay}$	1.5298	0.4486
R_{middle}	1.6541	0.3355
R_{final}	3.1493	1.1349

Table 6.1: Numerical values of the parameters used for Monte-Carlo simmulation

```
1
2  datafile = open("../mcRR.csv", "w+").readlines()
3
4  #Define mean and standard deviation parameters
5  means = [1.5298,1.6541,3.1493]
6  sds = [0.4486,0.3355,1.1349]
7
8  #Define other parameters
9  n = 20
10 overs = 50
11 state = 1
12
13 runrates = []
14
15 #Function for gaining the monte-carlo estimate
16
17 def MCE(n,state):
18     t = 0 if (state<=10) else 1 if (state>10 and state<=35) else 2
19     return(np.mean([random.normalvariate(means[t],sds[t]) for x in range(1,n)])
       )
20
21 #Perform Monte-Carlo Simmulation
```

Figure 6.2: Implementing a Monte-Carlo Simmulation fo Cricket Matches

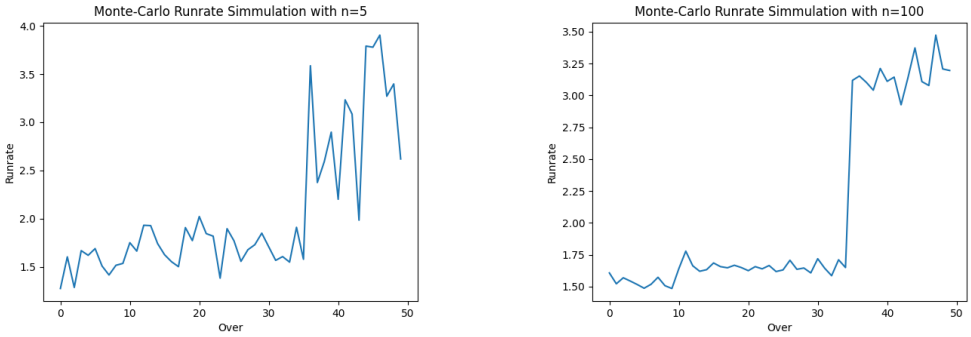
It is best to think of n as a sort of fine-tuning parameter. If we have n too large, we would simply be simmulating the average cricket game. But if we have n too low and we are open to having an outlier. As for the other parameters, we have *overs* being the number of overs in the game, and *state* determines the over to start simmulating from. So if we want to simmulate the whole game, set $state = 1$ and $overs = 50$. The array *runrates* just holds the data.

We then have the function *MCE(n,state)*. The first line of this is a "Ternary Operator" to determine a parameter t. The job of t is to be an index which obtains the correct parameters from the arrays in lines 2 and 3. This is then fed into the next line, uses the *random.normalvariate()* function to populate an empty list with random values drawn fro the appropriate distribution of R. This is then averaged and returned by the function. Completing the main part of the Monte-Carlo method. Finally, a while-loop runs through the overs needed and adds the estimation values to the *runrates* array.

To give an idea of how the value of n affects the resulting simmulation, we ran two simulations, using a small value $n = 5$, and a larger one using $n = 100$.

If we compare these figues with (a) in Figure 6.3, we see that with large n, we have fallen victim to the "Central-Limit Theorem", and it looks as if the three sections of the game are unrelated. In the end, $n = 20$ was the chosen value as it provided an appropriate middleground.

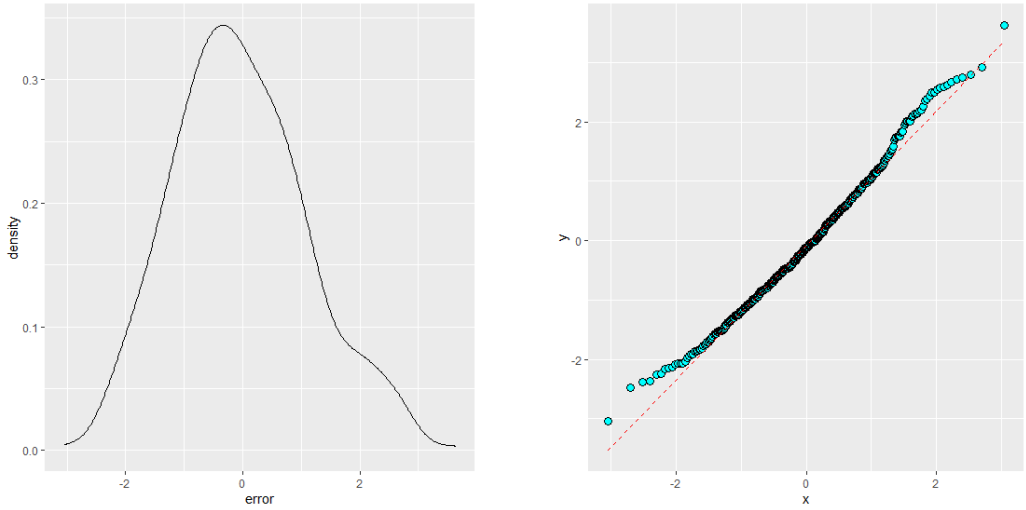
Testing if Monte-Carlo simmulation works was done in R by first cutting off each game in the original test set and random points, filling the rest in using the Monte-Carlo algorithm as previously described, and then feeding this into the neural network we built in the prior



(a) $n = 5$ (b) $n = 100$

Figure 6.3: 50-Over simmulation of $n = 5$ and $n = 100$

chapters.



(a) Error in Predictions from Monte-Carlo Simmulation

(b) Q-Q Plot of Monte-Carlo Prediction Errors

Figure 6.4: Monte-Carlo Simmulation Error Density

Up until an error of around 2, we see this was more normally distributed than the original predictions. It is worth noting that this could be a side-effect of the fact that during the monte-carlo simmulation, all the games are being drawn directly from a normal distribution. To see definatively how the model compares to the neural network with full data, we take the corrolation of the actual results with those of the predicted values. We find that $\rho_{Network} = 0.9382$ and $\rho_{Monte-Carlo} = -0.0636$. So we see that when filling in the gaps with monte-carlo methods, the performance drops significantly. To further see this difference in performance, we calculate the difference of the monte-carlo method predictions and the original predictions, and take a boxplot.

We can see in Figure 6.5, that the difference is mostly centered around 0, which is promising, but the fact the corrolations are close to 0 means this is not a reliable method for predicting cricket scores. This is clearly an issue, because when it comes to resetting scores, the game obviously wont have a full 50 overs played, and so we need to fill in the gaps. We can then decrease the target score in proportion with the number of overs lost.

To put these results in more context, the data must be unscaled. The data was originally

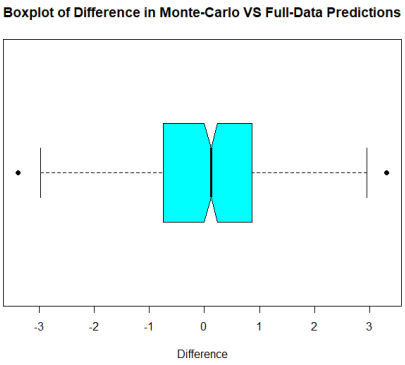


Figure 6.5: Boxplot of prediction differences

scaled using the base-R function *scale()*. This applies the function

$$x_{\text{scaled}} = \frac{x_{\text{original}} - \mu}{\sigma}.$$

The parameters in the scaling can be re-obtained in R using the *attr()* function. So implementing a function that performs unscaling on a new dataset is not too difficult, and can be achieved in afew lines of code. Note that the value of 51 has been hardcoded here as this is the column in which the runs scored are stored, but to make this more general, that value could be assigned dynamically.

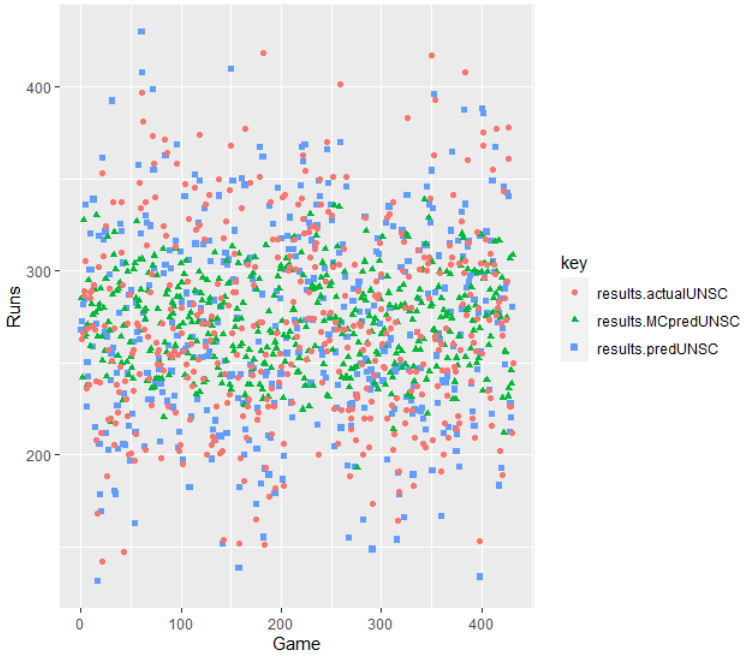
```
unscale <- function(target, original){
  scale_val = attr(original,"scaled:scale")[51]
  cent = attr(original, "scaled:center")[51]
  UNSCData <- c()
  for(i in 1:length(target)){
    UNSCData[i] <- target[i] * scale_val + cent
  }
  return(UNSCData)
}
```

Figure 6.6: R function for unscaling data

The function allows us to then see the actual predicted runs from both the complete neural network, and the neural network with gaps filled in via the Monte-Carlo method. Once this is done, for each game in the test set we plot the actual score, the score predicted by the full neural network and the score as predicted via the monte-carlo method.

This plot is naturally very busy, and it wont be used for detailed analysis of the accuracy, but what it does do well is give an overarching picture of the predictions. We can see that while the spread of red (actual scores) and blue (full network predictions) aren't too dissimilar, the spread of green points (monte-carlo predictions) is small and centered mainly around average game scores. This explains the poor performance of the model, it doesn't fair well with games that deviate far from the average. It is natural to ask how we can fix this. The one way that could lead to higher performance is to shrink the game into smaller chunks. Rather than looking at the three meta-stages of the game as we did originally, what about drawing from distributions of every 5 or 10 overs. While this may be more beneficial, the primary problem with this is it would be very computationally expensive, and as a result could lead to issues with the accessability if this model is deployed by means of a website or mobile application.

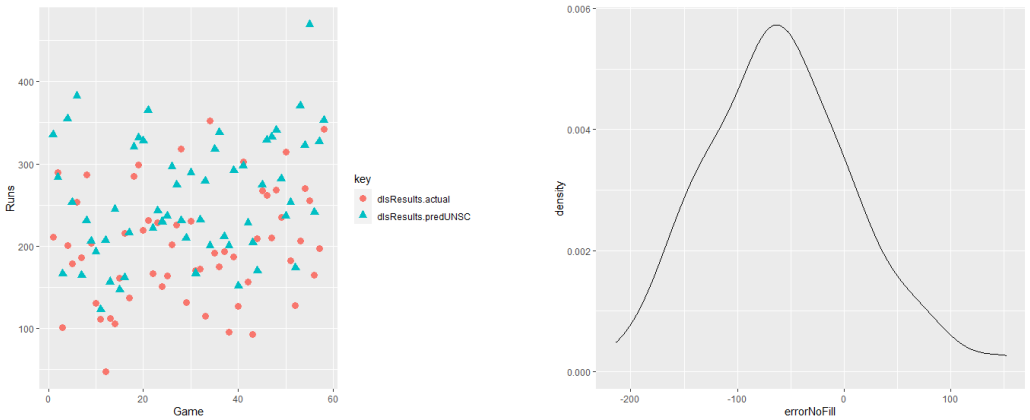
Figure 6.7: Plot of predictions for all methods



6.4 Unseen DLS Game Data

Since using Monte-Carlo methods didn't work too well, the next step is to look at seeing if something simpler will work. To do this, we take games that were decided through DLS, and feed them into the network to see how well it does. Rather than filling in the blanks of overs not played, we just set the runrate to 0 in these overs, and feed the resulting games into the neural network. We took 51 games which were decided by DLS, and looked at what the target set by DLS was.

The results for this were actually better than for the monte carlo gap filling method. We achieved a corrolation in the results of 0.4983. Which is naturally lower than we would like, but a large improvement on the prior method. We can see in Figure 6.8 how the spread of points isn't in a narrow band as it was with monte carlo predictions. We can also see the errors that came along with the predictions.



(a) Network predictions for DLS games (b) DLS Prediction Errors

Figure 6.8: Monte-Carlo Simmulation Error Density

These errors are follow a rough bell-curve shape, however we must remember that there aren't many datapoints here, so it may well be that with more datapoints, we would see more of a normal distribution. The downside however is that this curve is not centered around an error of 0, but around -58.22. This means that on average, the model sets teams a target of a much higher score.

Chapter 7

Simulating the ICC 2019 Cricket World Cup

In this chapter, we use the model we have built and apply it to games decided by DLS in the 2019 cricket world cup. The objective is to see if our model gives a similar outcome to the tournament as DLS did, or to see how the tournament would have differed using our model. WE give a sceheme for how score targets will be reset, and use this to simmulate the games in question.

7.1 DLS in the 2019 World Cup

Only three games were decided by DLS in the 2019 Worlc Cup, two of which involved Afghanistan. The other was a game between India and Pakistan. The ball-by-ball data for each of these games was collected and tidied in the previous ways. There are three different ways in which we can apply the neural network depending on how much of the first innings gets played.

- If the full first innings isn't played, predict a score using the network, and give the target as the proportional score to the number of overs available.
- If the full first innings is played, and some of the second innings is played, apply the network to the equivalent overs in the first innings, and use that to set the score.

7.2 Applying the Neural Network Model

7.2.1 Sri Lanka vs Afghanistan

Batting first, Sri-Lanka ended their rain-affected innings on 201 after 36.5[1] overs. Under DLS, Afghanistan were set a target of 187 from 41 overs. However, Afghanistan were bowled out for 152 from 32.4 overs, coming up 34 runs short of the required target from just 79.7% of their alloted innings. Using the data from the first Innings, our model predicted that Sri Lanka would have gone on to score 338 runs had they have had a full innings. Since Sri Lanka only played 61.3% of their innings, under this method, Afghanistan would have been set a target of $0.613 \times 338 = 207$ runs. However, that is if Afghanistan were given a full 50 overs, which they didn't have in this game due to light. For that reason, we need to again reduce this by the ration of available overs, 41 in this case. Therefore we reduce this by $\frac{41}{50} = 0.94$, giving a reduced target score of 195 runs from 41 overs. This is considerably higher than the score set by DLS, and given how Afghanistan batted, quite far out of reach. This means our method has had no affect on the standings of the World Cup from this game.

[1] In cricketing notation, this means 36 overs and 5 balls, not 36 overs and 3 balls, which would be the case if .5 meant half an over.

7.2.2 Afghanistan vs South Africa

This time batting first, Afghanistan were bowled out for 125 from 34.1 overs. Midway through the first innings, the game was reduced to 48 overs per side. After the first innings, South Africa were set a target of 127 from their 48 overs. Interestingly, our model predicted that Afghanistan would reach a score of 125 had they have had a full 50 overs. Strictly speaking, in this scenario that's a perfect prediction as they had no batters left so couldn't go any higher than the 125 they actually did achieve! Since South Africa had 48 overs available to them, that's an increase of 1.4, so the score target set under our scheme is $1.4 \times 125 = 175$ runs from 48 overs. Using our model on the South Africa innings, a predicted score of 151 is obtained. From 48 overs instead of 50 this gives a predicted score of $0.96 \times 151 = 145$. This naturally falls up short of the 175 target, so actually Afghanistan win this game, and would give them their only win of the tournament. They stay bottom of the points table, South Africa however would have dropped two points and fallen to eight place, allowing Bangladesh to move up into position seven.

7.2.3 India vs Pakistan

After a stellar innings from Indian batsman Rohit Sharma, hitting 140 off 113 deliveries, India finished their 50 overs with a score of 336- losing only 5 wickets in the process. Pakistan stepped up to the plate, reaching 166 from 35 overs, chasing 337. But then the rain came down over Manchester and the score was revised by DLS to them needing 136 more runs from just 5 overs. This requires a mammouth effort of 27.20 runs per over. Needless to say, Pakistan fell short, going on to hit a total of 212 from their 40 overs. Our model predicted that after the 35^{th} over, India would go on to score 284 in their 50. Reducing this by 0.2 to give a predicted score after 40 overs, we get a target score of 227. Paksitan reached 212 off their 40, so fall short and still lose, meaning nothing changes in the table. Given that Pakistan were on 166, this new target of 61 from 5 overs is far more realistic, and would have meant for a far more enjoyable game for the spectators, as it would have at least given Pakistan a fighting chance, and not essentially killed the game by setting such a high target from a short number of overs.

7.3 Summary of Model Affect on the Tournament

There is only one change to the final table, and that change has no impact on the overall impact of the tournament, since India still won against Pakistan. Afghanistan and South Africa both finished in the bottom half of the table, and even with our method, South Africa would have only dropped one place, while Afghanistan's position wouldn't change.

While our model gave similar outcomes to DLS, the score targets set were arguably more realistic in some cases, especially the India-Pakistan game, and certainly from an entertainment perspective, this has great value in maintaining the competaive aspect of the game. Naturally, using proportions as a way to decrease or increase something that is inherently random isn't ideal, but this leaves room for future work on the topic.

It is worth mentioning that on average, this model had a tendancy to predict a higher score by an average of 58.22 runs. This could perhaps be taken into account by reducing the target set by this or a similar ammount.

Chapter 8

Conclusions and Future Work

> I'm completely cricketed out. If I never have to write another word about cricket again, I'll be a happy man.
>
> Joseph O'Neill

8.1 Conclusions

We presented a Neural Network model for predicting cricket scores based on patterns in the run rate of a game. Neural Networks have been used in cricket for things such as generating commentary [Kumar et al., 2019] or classifting shots based on footage [Foysal et al., 2018], but we couldn't find a paper using them in this way before. The Neural Network model we built for predicting cricket scores did fairly well when given a full innings of data, a 94% corrolation with actual values is certainly not something to be glossed over. However, the aim of this project was for predicting results when the full data is not available due to an innings being cut short. In this domain, the network did considerably worse- correlating poorly with test data when monte-carlo methods were used to fill in the missing games, and only achieving only 50% corrolation when the gaps were left as 0s. Despite this, applying the network to an actual tournament gave similar results to the standard method used for achieving the same task, and had marginal affect on the tournament's outcome. One key thing that is worth mentioning is the fact our method gave more attainable scores for the batting team to achieve. This is more favourable than some of the score targets set by DLS due to keeping the competetive spirit of a match alive even when the game has been interrupted.

8.2 Future Work

Rather than decrasing/increasing scores via a proportion, it would be benificial to incorperate the fall of wicket densities looked at earlier in the paper in some way. This is one area our model doesn't incorperate at all, let alone well. Yet, it is a key aspect of cricket, and so in the future it could be highly effective if taken into account.

Using team specific data rather than an average of all games could be interesting thing to look into, but the computational cost of this would be high, and the accessability of the method would be poor as a result, defeating the point of making this a system easily employable across cricket.

As mentioned before, the lack of data available is a shame, but this is something that can't

be helped due to the nature of the cricketing calender. It would be interesting to revisit this topic in a couple of years when more games have been played to see how results differ. One other avenue would be to look at combining the international games that have been included with domestic 50 over matches from around the world. At present, this would have increased the number of games by 436.

In appendix C, we discuss how the code used for this project is turned into a publicly available R package. In future, this will allow for easily experimenting with different neural network parameters such as the activation function and hidden layer arrangements. It goes without saying that this lends itself well to future work at building a much better neural network for predicting cricket scores. Naturally, a lot of this is trial and error and so it would require a considerable amount of computing time to find an acceptable one.

In Section 4.4.2, we saw how the order of "most important" variables was different, and so we could use the package to change the formula used by the network to take advantage of the more important overs only. Again this raises several more variables, since we would need to experiment with different numbers of important overs and such.

Appendix A

Q-Q Plots and the Normal Distribution

Q-Q plots have been used extensively throughout this project, due to their utility in testing if a sample is normally distributed. For that reason, delving into the maths behind them a bit more helps with interpreting their results.

At its core, a Q-Q plot shows the quantiles of two distributions against one another. This can either be drawn from two exact datasets, or, as is common in our case, one dataset against samples from a particular probability distribution. This is the case this appendix will focus on. The Q in Q-Q plot refers to "Quantile". Quantile functions rely on the distribution function of a particular distribution.

Definition A.0.1. *Suppose X is a random variable. The **Cumulative Distribution Function** (CDF) of X is $F : \mathbb{R} \to [0, 1]$ given by*

$$F(x) = P(X \leq x)$$

In the case of the normal distribution, we have the following lemma.

Lemma A.0.2. *Let X be normally distributed with mean μ and variance σ^2. Then the cumulative distribution function of X is*

$$F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right) = \frac{1}{\sigma\sqrt{2\pi}} \int_{\infty}^{x} exp\left(-\frac{(t-\mu)^2}{2\sigma^2}\right) dt$$

In almost every application throught this project, we have been looking to see if data follows a standard normal distribution. In which case, the CDF is

$$F(x) = \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{\infty}^{x} exp\left(-\frac{t^2}{2}\right).$$

The following lemma will allow us to use a useful result in building Q-Q plots.

Lemma A.0.3. *$\Phi(x)$ is monotonically strictly increasing*

Proof. We have the standard result $\frac{d}{dx}\left(exp(-x)\right) = -exp(-x)$. Now since $exp(x)$ is by definition strictly increasing, $exp(-x)$ is strictly decreasing. It therefore follows that $-exp(-x)$ is strictly increasing. From the fact $\frac{1}{\sqrt{2\pi}}$, and the prior argument, it follows that $\Phi(x)$ is strictly increasing. The fact that this is the case on the entire domain of $exp(x)$, means that $\Phi(x)$ is also monotonic. □

The reason we need to show $\Phi(x)$ fits this in particular property is that if for some distribution with CDF F, F is continuous and strictly monotonically increasing, then the *Quantile Function* Q is given by $Q = F^{-1}$. Infact, the standard normal distribution's quantile function, $\Phi^{-1}(p)$ $p \in (0, 1)$, is called the *Probit* function. The Probit function is defined in terms of the relative error function. Formally, this is given by the following:

Definition A.0.4. *The **Relative Error Function**, $erf(x)$, gives the probability that the random variable $X \sim N(0, \frac{1}{2})$ takes a value between $-x$ and x inclusive.*

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x exp(-t^2)dt.$$

With that, we can formally define the quantile function of the standard normal distribution.

Definition A.0.5. *The **Probit** function, $probit(x)$ is given by:*

$$probit(x) = \sqrt{2}erf^{-1}(2p - 1).$$

It is this probit function that forms the x-axis in our Q-Q plots. The y-axis contains the sample quantiles. Consider the case where we sampled the same (theoretical) distribution twice, putting one on the y-axis and one on the x-axis. Clearly, the plot would form a straight line equivalent to $f(x) = x$ on the 2D cartesian grid. This is the rationale behind the utility of the plot, since if the line is as close to $f(x) = x$ as possible, then we can say with a fairly high confidence that the sample distibution follows the theoretical one we are testing against. If, however, the line formed is curved or differs in another way, then it is safe to say the sample distribution doesn't follow that distribution.

Appendix B

Neural Network Plot

The Neural Network produced in Chapter 5 is quite large, but the `neuralnet` package does allow us to view it nonetheless.

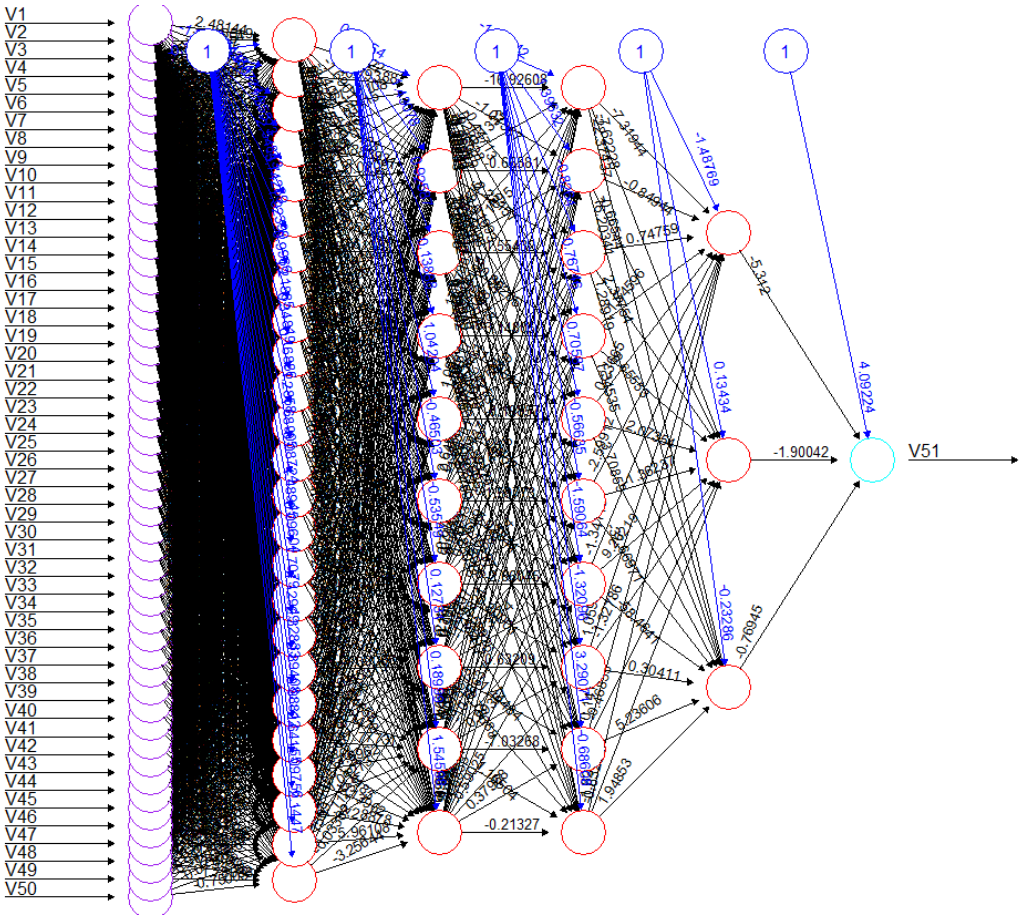


Figure B.1: Final trained network.

It is hard to see the exact weights and biases themselves, but these values aren't individually that important, the plot is more for demonstrative purposes.

Appendix C

The CricNet Package

This chapter contains no new mathematics or results. We discuss turning the code written here into an R package that can be used for easily expanding on the work of this project. The package comes complete with the dataset that we trained our network one, meaning the interested reader could replicate our results or even improve upon them.

C.1 Introduction

R as a language is built on packages, the **C**omprehensive **R** **A**rchive **N**etwork, CRAN, stores thousands of packages that can be used for performing tasks in R. This saves people re-inventing the wheel when performing statistical analyses. In this project, we have used CRAN to obtain access to the several packages that were imperitive in undertaking the task of building a neural network. Packages are installed from CRAN using the function *install.packages("package")* and loaded with *library("package")*. However, due to the relatively small scale of this project, we won't be storing the package on CRAN (at this point in time), but rather on Github. This is preferable due to how easy it is to access code on Github, and because it also allows other people to directly contribute to the expansion of the package should they wish.

C.2 Building an R Package

Building an R package is quite a simple concept- we are just putting all our data and functions into one place that can allow for easy modification and the replicating of results. Actually building a package requires using other packages, specifically the `devtools` package. Running the function *usethis::create_package()* from the `usethis` package creates a directory with the necessary files for an R package. This file structire initially can be seen in the directory tree below.

Figure C.1: Default File Structure for an R Package

```
CricNet
├── CricNet.Rproj
├── DESCRIPTION
├── NAMESPACE
├── R
└── data
```

The R directory is unsurprisingly where the R files containing all the functions go. Naturally not every piece of code we wrote will go into the package, as some of it was simply for

demonstrative purposes. All the code for actually running the network (which is more or less a wrapper on the `neuralnet` package), along with the code for analysing those results goes into the package. Each function gets its own file, for ease of access, documentation and debugging.

The files NAMESPACE and DESCRIPTION are metadata files used by the `roxygen2` package for mainting things such as version number, liscencing and name(s) of the author(s). NAMESPACE is also used to store any dependencies on packages that the package relies on. For example, our *scoreNet.R* file, as mentioned, is a wrapper on the function *neuralnet::nerualnet()*, so it clearly needs to import the required package.

C.3 CricNet Structure

The documentation for what each function does can be found within the package documentation, or using *help(function)* in the R console. The purpose of this section is to give an overview of the package and how to use to replicate results from this project.

The CSV containg the run rates that the network was initially trained on exits in the *data* directory as "rrmat.rda". Saving this data is done using the function *usethis::use_data()* function which puts everything in the right format and direcotry automatically. There are four R functions that come with the package, "scoreNet.R", "genResults.R", "unscale.R" and "netAnalysis.R". Note that they should be used in this order. No output comes from the first two, but by running them, several R objects are created. One for the network itself (an object of type *nn*), and one for storing the results. The analysis script takes this results dataframe and displays a corrolation score and a density plot of the errors. In the future, more analysis features will be added to this function. Release version 0.1 of the package is the one that corresponds directly to the work of this project, with no expansions.

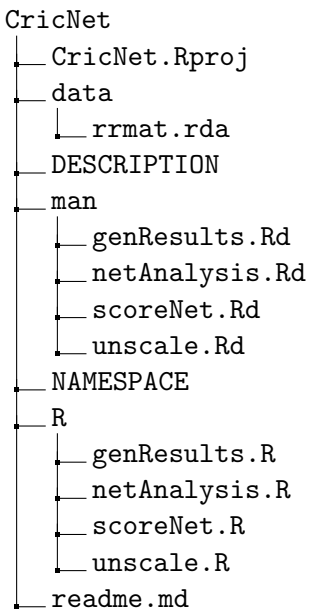
Adding documentation to each function is done using the `roxygen2` package. It is a case of simply writing docstrings above the functions in their respective files, and using *usethis::document()* to build the file. In the docstring, the "export" tag is what allows the functions to be used directly by anyone who installs the package. These pieces of documentation create new ".Rd" files in the "/man" directory. Storing them here allows them to be viewed alone by running the command *help("function")* in an R console.

The overall structure of the `CricNet` package can be seen in the directory tree in Figure C.2.

C.4 Using The Package

The package can be accessed at "https://github.com/mattknowles314/CricNet". It can be installed directly in R using the `devtools` function *install_github(mattknowles314/CricNet)*. Any bugs or issues can be reported via the "Issues" tab on the Github page.

Figure C.2: File structure for the CricNet package



Bibliography

[Bhattacharya et al., 2018] Bhattacharya, I., Ghosal, R., and Ghosh, S. (2018). A statistical exploration of duckworth-lewis method using bayesian inference. *arXiv preprint arXiv:1810.00908*.

[Cauchy et al., 1847] Cauchy, A. et al. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538.

[Duckworth and Lewis, 2004] Duckworth, F. and Lewis, A. (2004). A successful operational research intervention in one-day cricket. *Journal of the Operational Research Society*, 55(7):749–759.

[Duckworth and Lewis, 1998] Duckworth, F. C. and Lewis, A. J. (1998). A fair method for resetting the target in interrupted one-day cricket matches. *Journal of the Operational Research Society*, 49(3):220–227.

[Foysal et al., 2018] Foysal, M., Ahmed, F., Islam, M. S., Karim, A., and Neehal, N. (2018). Shot-net: a convolutional neural network for classifying different cricket shots. In *International Conference on Recent Trends in Image Processing and Pattern Recognition*, pages 111–120. Springer.

[Friedman, 2017] Friedman, J. H. (2017). *The elements of statistical learning: Data mining, inference, and prediction*. springer open.

[Ganesan et al., 2010] Ganesan, N., Venkatesh, K., Rama, M., and Palani, A. M. (2010). Application of neural networks in diagnosing cancer disease using demographic data. *International Journal of Computer Applications*, 1(26):76–85.

[Günther and Fritsch, 2010] Günther, F. and Fritsch, S. (2010). Neuralnet: training of neural networks. *R J.*, 2(1):30.

[Huber, 1992] Huber, P. J. (1992). Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer.

[Kampakis and Thomas, 2015] Kampakis, S. and Thomas, W. (2015). Using machine learning to predict the outcome of english county twenty over cricket matches. *arXiv preprint arXiv:1511.05837*.

[Kumar et al., 2019] Kumar, R., Santhadevi, D., and Barnabas, J. (2019). Outcome classification in cricket using deep learning. In *2019 IEEE international conference on cloud computing in emerging markets (CCEM)*, pages 55–58. IEEE.

[Kumar and Roy, 2018] Kumar, S. and Roy, S. (2018). Score prediction and player classification model in the game of cricket using machine learning. *International Journal of Scientific & Engineering Research IJSER*, 9(8).

[Lister and Stone, 1995] Lister, R. and Stone, J. V. (1995). An empirical study of the time complexity of various error functions with conjugate gradient backpropagation. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 1, pages 237–241. IEEE.

[Massey Jr, 1951] Massey Jr, F. J. (1951). The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78.

[Naeini et al., 2010] Naeini, M. P., Taremian, H., and Hashemi, H. B. (2010). Stock market value prediction using neural networks. In *2010 international conference on computer information systems and industrial management applications (CISIM)*, pages 132–136. IEEE.

[Phanse and Deorah, 2011] Phanse, V. and Deorah, S. (2011). Evaluation and extension to the duckworth lewis method: A dual application of data mining techniques. In *2011 IEEE 11th International Conference on Data Mining Workshops*, pages 763–770. IEEE.

[Riedmiller and Braun, 1993] Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE international conference on neural networks*, pages 586–591. IEEE.

[Saqlain et al., 2019] Saqlain, M., Jafar, N., Hamid, R., and Shahzad, A. (2019). Prediction of cricket world cup 2019 by topsis technique of mcdm-a mathematical analysis. *International Journal of Scientific & Engineering Research*, 10(2):789–792.

[Sibi et al., 2013] Sibi, P., Jones, S. A., and Siddarth, P. (2013). Analysis of different activation functions using back propagation neural networks. *Journal of theoretical and applied information technology*, 47(3):1264–1268.

[Stern, 2016] Stern, S. E. (2016). The duckworth-lewis-stern method: extending the duckworth-lewis methodology to deal with modern scoring rates. *Journal of the Operational Research Society*, 67(12):1469–1480.

[Tibshirani, 1996] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.