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 detailed discussion on the application of Neural Networks in cricket is presented. These novel results follow on from a look into the current methods used by the International Cricket Council for solving the problem of reducing score targets. We present the mathematical background of building and training neural networks, accompanied by code snippets of implenentation of the methods. A detailed analysis of the models results is given, through three different datasets. We find that in trying to predict scores, using monte-carlo simmulation to simmulate a match performs significantly worse than not filling in the gaps.

Acknowledgements

The author would like to thank Dr. Jessica Hargreaves for their invaluable supervision, encouragement and advice throughout this project. Secondly, my eternal gratitude and admiration for my friends in York, who have made my time here so wonderful from the very begining, and for all the support you have given me every step of the way. My thanks to Michael Najdan at Kent County Cricket Club for his insight into how data is used in the professional game of cricket. In addition, to Harrison Allen at Yorkshire County Cricket Club for showing me how limited-overs data was put into practice in preperation for games. Finally, to the UYMCC, who have enriched my university experience beyond belief.

Notes

All code written in support of this project can be found on GitHub at:
https://github.com/mattknowles314/mastersThesis.

Contents

List of Figures

Chapter 1

Introduction

I don't like discussing cricket off the field.

MS Dhoni

In this chapter, we outline the background behind this project. Looking into why the topic is one of relevancy, and why it is worth looking at. This comes after an introduction to cricket itself, the sports-inclined reader may still find this of use, as it talks about the specific aspects of the game that will be most important in the remainder of the project. The issues raised by the current methods which we are trying to improve upon are explained. This leads to a review of some of the important literature on the topic to give more context behind the work done in this project.

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 thing is the use of metrics such as strike-rate for selecting bowlers for the starting XI. In limited overs cricket, asking questions about how scoring boundaries will affect the score at different grounds. Or how wicket-taking in the "powerplay"[1] vs in the "middle overs" will change the outcome of the game. Data visualisation is key in putting across this information to playing staff to highlight key areas for imprtovement. For this reason, this project has taken an approach of using extensive data-visualisation to tell the story of the work being done.

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

Cricket is a game played by two teams, each with 11 players. The objective for the batting team is to score as many "runs" as possible without losing 10 of their batsmen[2], who can be "out" in a variety of ways. Cricket is played in "overs", each over lasting 6 deliveries. Traditionally, the game lasted for 4 days and there was no limit to the number of overs the bowling team could bowl.

In 1963, the cricketing calender in the UK had for the first time a different format of the game ammended to it. The "Gillette Cup" introduced a form of cricket wherein each team has 1 innings, lasting 50 overs. 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.

However, 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. [3]. 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.

[1] In T20 cricket, this refers to the first 6 overs of the game, where certain fielding restrictions are in place

[2] There have to be two batters on the pitch, so if 10 wickets are lost, it leaves one batter stranded.

[3] Note that "draw" and "tie" are not interchangable terminology in cricketing terms.

1.3 Problems raised with DLS

In [**?**], 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 for reducing these biases, but we won't cover that in this project.

1.4 Project Aims

The main aim of this project is to look at methods for *predicting* cricket scores. It is important to make the distinction between this and *projecting* the scores, which is what goes on in the game at the minute. Projecting scores assumes a constant runrate, 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.

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 [**?**]. In this paper, Duckworth proposes the "D/L" method based on 5 principles. Hwever, the 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". The claim is that these functions have been defined via experimentation. This however 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 [**?**], in which they report that the table used for calculating the method can be employed by

Another paper that is worth noting is by Saqlain et. al. [**?**], 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 the following algorithm.

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: Calculate Euclidean distance from the best outcome $S_i^+ = \sqrt{\sum_{j=1}^{n} (V_{ij} - V_j^+)^2}$ for $i \in \{1, 2, ..., m\}$
4: Calculate Euclidean distance from the worst outcome $S_i^- = \sqrt{\sum_{j=1}^{n} (V_{ij} - V_j^-)^2}$ for $i \in \{1, 2, ..., m\}$
5: Calculate performance score $C_i^* = \frac{S_i^-}{S_i^+ + S_i^-}$

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.

However, a paper by Kumar and Roy, [**?**] 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.

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.

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, the Python programming language was used 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 packages `JSON` and `CSV`.

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.

[1] https://cricsheet.org/

Chapter 3

The DLS Method in Detail

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

We now look at the mathematics behind the D/L, and DLS methods. D/L being the original method, and DLS the method that Stern helped to revise. In the original paper, the authors state "Commercial confidentiality prevents the disclosure of the mathematical definitions of these functions." [**?**]. Which is naturally a slight problem for this, but what we can do is instead use sample values based on data we have to look at how these functions behave, so all is certainly not lost.

3.1 Origins: Duckworth and Lewis

We begin by looking at the original paper. 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 the exponential decay constant, and Z_0 is the average total score in first class cricket, but with one-day rules imposed.

Now because we don't have access to actual values for Z_0 or b, a plot to see what equation **??** 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 D/L paper.

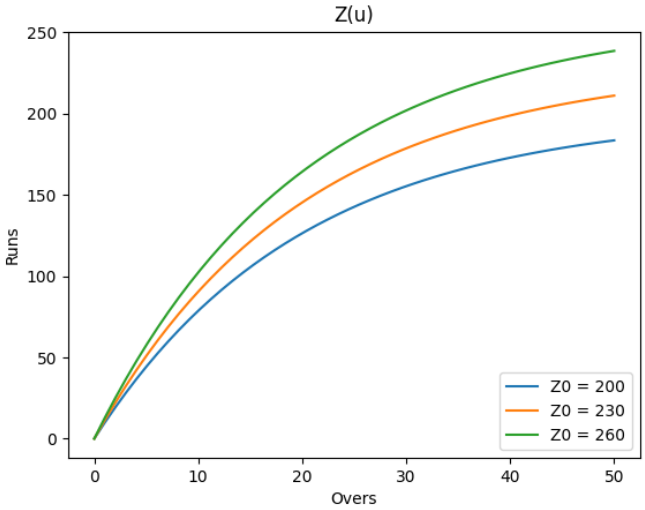


Figure 3.1: Graph showing how the rate at which runs are scored decays as a game progresses.

However, we have not yet looked at what happens when wickets are lost. To introduce this metric, equation **??** 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 now changes depending on wickets lost.

With this in mind, we now look at the specific case of equation **??** 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

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

The 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. Let us now look a bit more 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 $u_2 < u_1$. At the time of the break, the second team had lost w wickets. The aim is to adjst the required score to account for the $u_1 - u_2$ overs they have lost. The winning "resources" available are given by*

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

Which means, 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 by Stern in [**?**] to extend the original D/L 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 D/L. 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 u overs remaining, w wickets down in an M-over match with a final 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 $a_\lambda = -1/\{1+c_1(\lambda-1)e^{-c_2(\lambda-1)}\}$ and $\beta_\lambda = -c_3(\lambda-1)e^{-c_4(\lambda-1)}$. 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 paper [**?**], which used Bayesian modelling to try and improve the score predictions of the D/L method. The authors used the same dataset as we are using for the 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 the calculated over this quantity. This is actually the point which motivates using Bayesian statistics to extend the D/L 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 **??**. This mean function, based on the plots produced in the original D/L paper, (see **??** for an example), the authors adopted an exponential decay. We can see why this choice makes sense from figure **??**. 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 inderence. The values chosen were therefore $A_0 = 200$ $B_0 = 100$ and $a = b = 0.1$.

One thing briefly mentioned in chapter 3 of this 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 **??**.

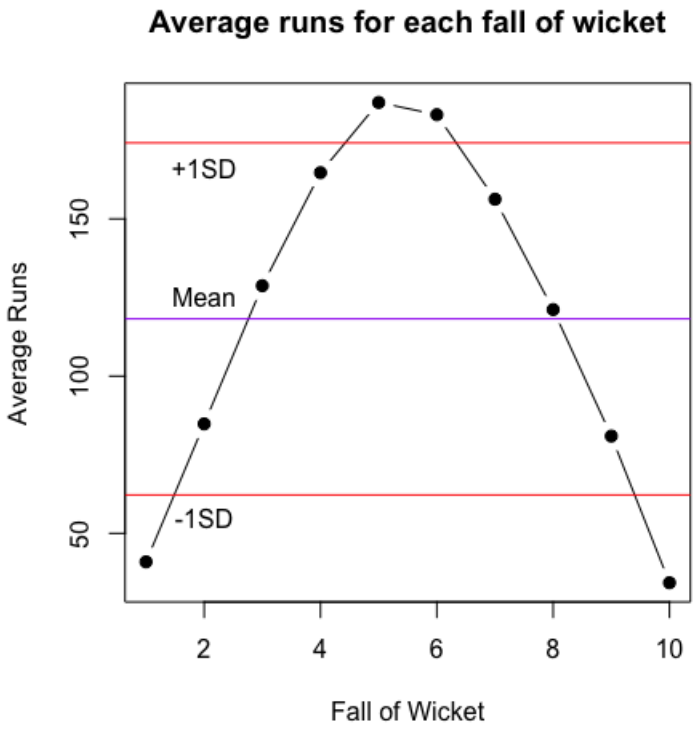


Figure 3.2: Plot of average runs scored at the fall of each wicket. Averages taken over 1437 innings.

Initially, one may think that this is normally distributed. It certainly holds a bell curve-like shape. However, a Kolmogorov-Smirnov test [**?**] 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.

Does the shape of this curve make sense? From a cricket standpoint it does, 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.

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. This 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 onthe way cricket games unfold. This consistency allows us to make an assumption about Runrates 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.
Runrates 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 **??**-**??**, we can see how this evolves.

In **??**, 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 **??**, 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 later order batters are coming in 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 **??**, 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

Figure 4.1: Density of all teams for first wicket falling

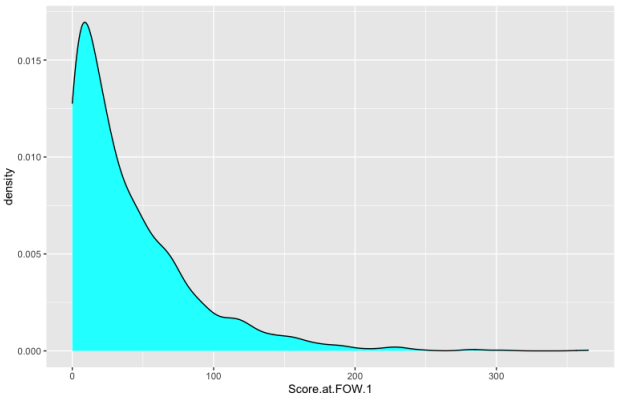


Figure 4.2: Overall density plot for FOW 1

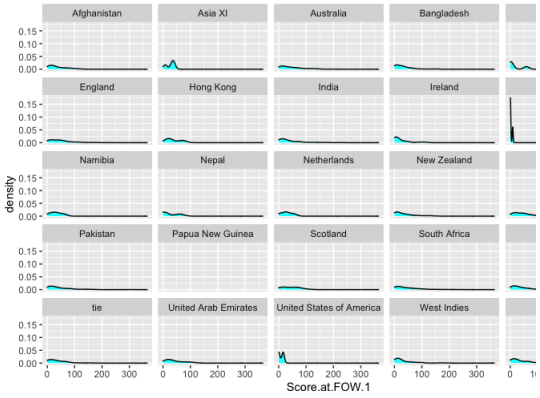


Figure 4.3: Density of all teams for fith wicket falling

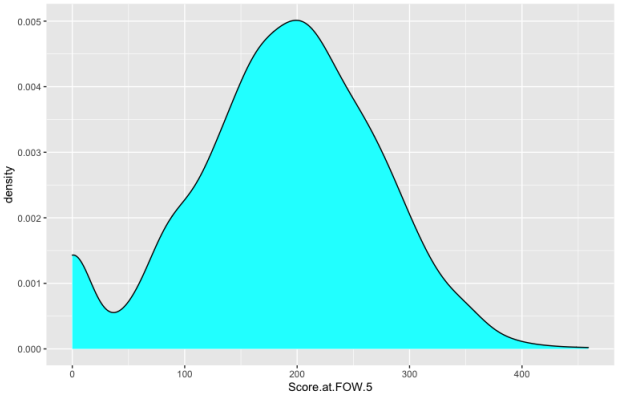


Figure 4.4: Overall density plot for FOW 5

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 this, we need to look at the spread of runs scored in a full innings of data. This can be seen in **??**.

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.

We can see from the above figure 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

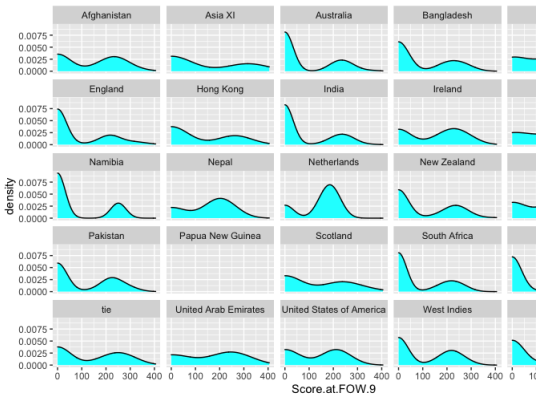


Figure 4.5: Density of all teams for ninth wicket falling

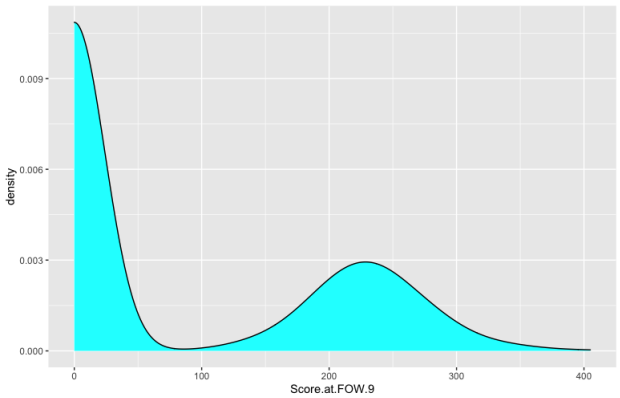


Figure 4.6: Overall density plot for FOW 9

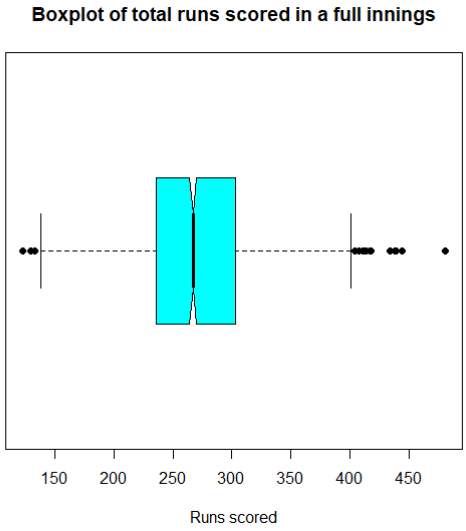


Figure 4.7: Boxplot showing the spread of runs scored

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 $S_{50} \sim \mathcal{N}(270.56, 51.34^2)$.

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 **??**

With this in mind, we can now look at a density plot for the actual data. This is giving in figure **??**.

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 Runrates

The work that follows in this section is essential to 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 runrates 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 runrates to the neural network, and allow for a score to be predicted.

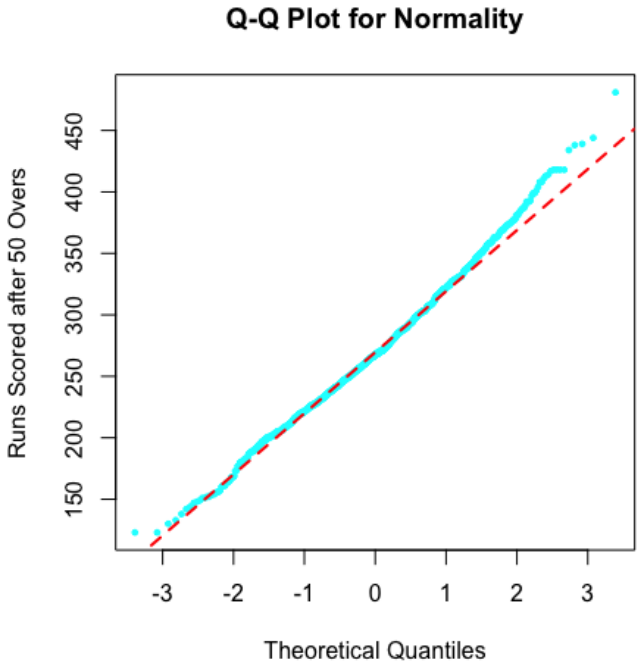


Figure 4.8: Q-Q plot for runs scored after 50 overs.

We can see from this figure, that average runrate 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 mentality is to double your score from the first 35 in the last 15, that is why we see the massive increase in runrate in these last overs.

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

The runrates are roughly normally distributed. 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 use our

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

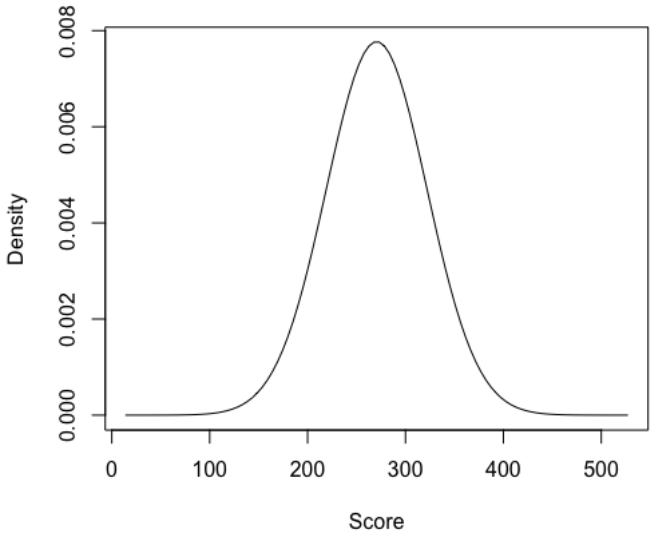
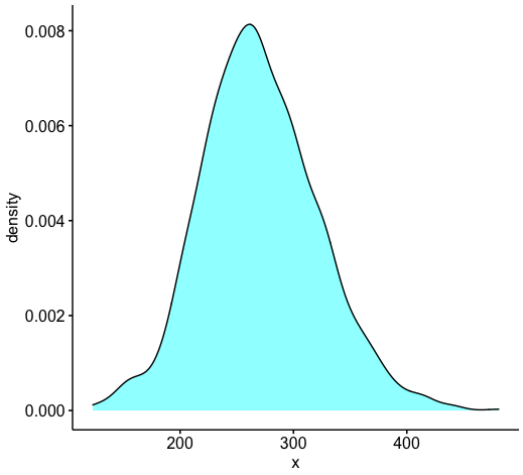
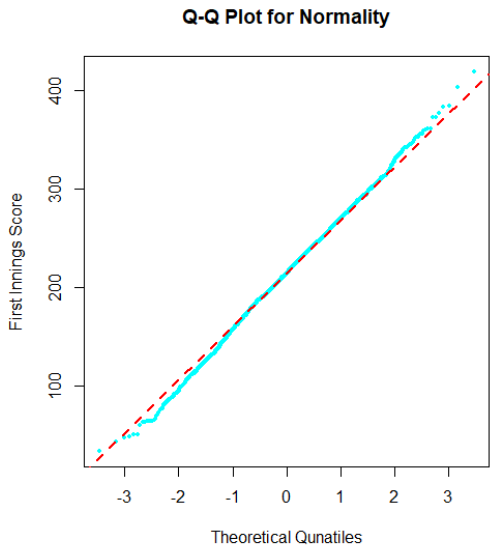


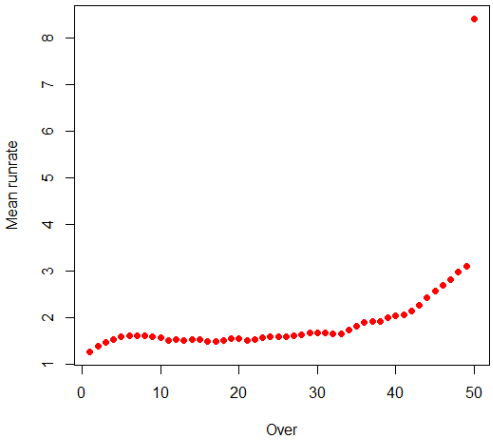
Figure 4.9: Sample plot created from the derived distribution of S_{50}



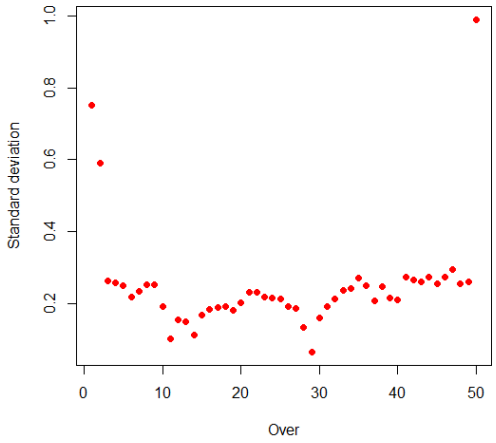
[Density for Runs Scored]



[Q-Q Plot for First Innings]



[Average runnrate per over]



[Runrate Standard deviation per over]

Figure 4.10: Error distribution with Q-Q plot

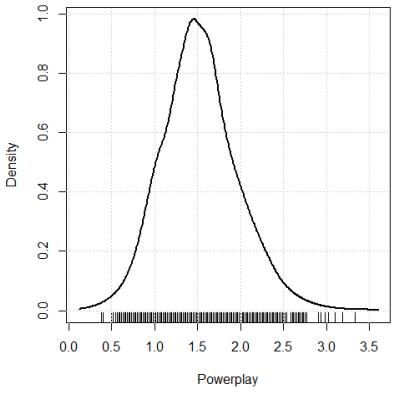


Figure 4.11: Powerplay runrate density

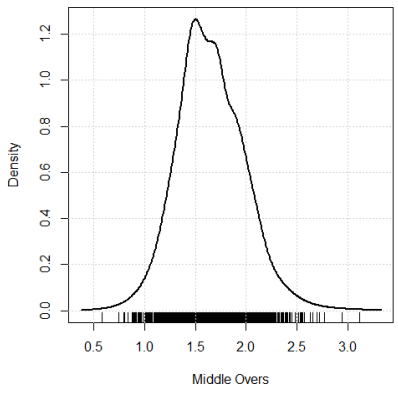


Figure 4.12: Middle Overs runrate density

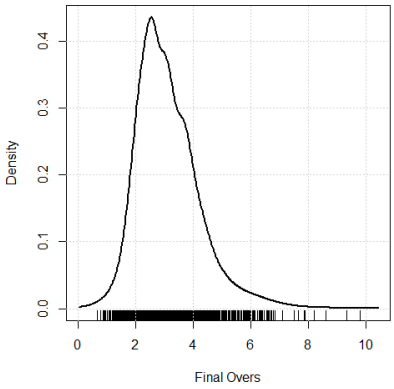


Figure 4.13: Final Overs density

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 hype in recent years. They are a machine learning method that is being applied to many problems in all sorts of fields, such as finance [**?**] and medicine [**?**]. The network will be trained on runrate data, so for each game we have calculated the evolution of the runrate, 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 **??**. 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.

Let's look at what a NN actually looks like. The below is an example of a network with just one hidden layer.

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. In order to find try this out, we will have to randomly split the matrix into a training matrix and a testing matrix. We can then evaluate the network by trying to predict values from the testing set and seeing how well it does. This will then lead to us refining the number of hidden layers as appropriate.

One area of the world where neural networks are being applied to value prediction is in the stock market. 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 [**?**], the authors twest two different Neural Netowrks,

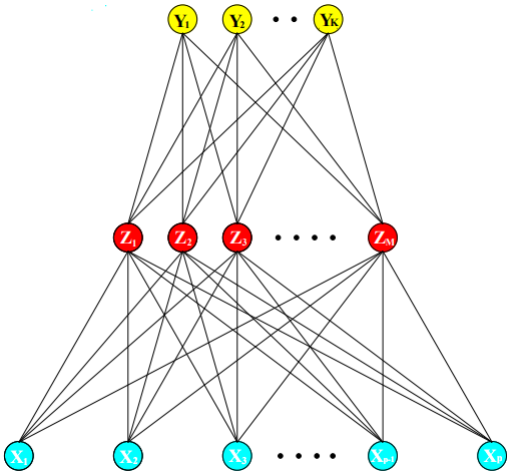


Figure 5.1: Example of a single hidden layer neural network, as in [**?**]

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 runrate 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 valye h given by **??**:

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

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 **??**.

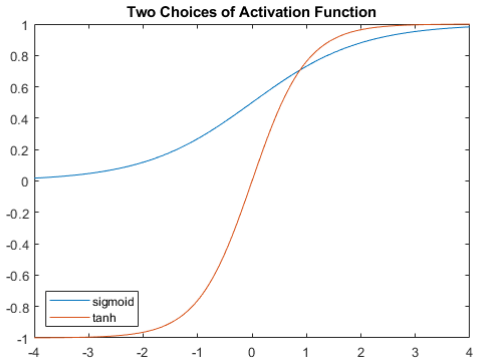


Figure 5.2: Graph showing the shape of different activation functions.

We will be using (5.2) as our activation but will return to the activation function shortly. For now, lets look at the network we're going to use. The initial neural network that we're building. The code for this activation can be seen in **??**, note we have included the derivative of this function too, which will be important later.

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 **??**.

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

Using **??**, 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 all the the remaining layers in the network.

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 will be hooribly 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" [**?**]. 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 **??** the need to use a MSE loss function based on our data. Let us now define this function, as in [**?**].

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 **??**

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

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

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

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

5.3 Implementing the Neural Netowk

It was decided to implement the neural network using the R programming language, and more specifically, the *neuralnet* package [**?**]. This package allows us to specify all the parameters we wish for, and automatically performs the Backpropogation algorithm to train a network. The whole process is therefore summarised in one line of R code:

Recall the logistic function,

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

For large values of x, this function treats them all the same, we must normalise the data before training the neural network. Several versions of the network were ran with varying parameters. The final parameters decided were a learning rate, α of 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 below.

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

Figure 5.3: R code to implement the neural network

A plot of this network can be seen in Figure **??**. 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 [**?**]. 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()* in R. We then use the function *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. The final net can be seen here, although due to the nature it is hard to see the specific weights and biases- although this is less important than the results that are discussed in the next chapter.

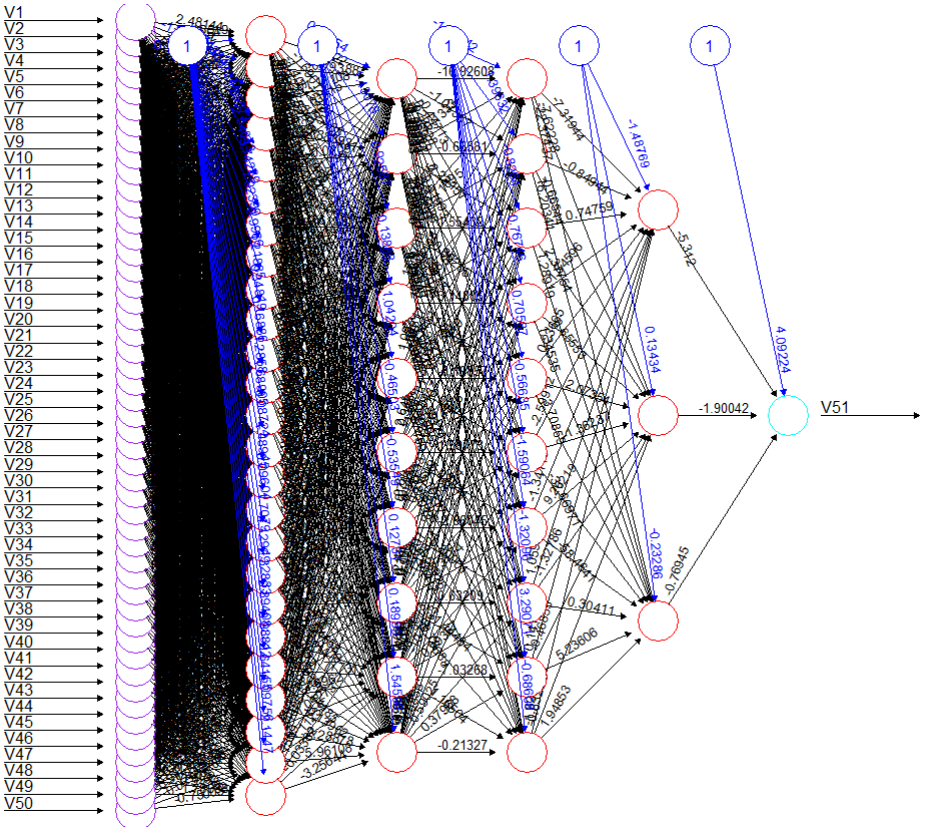


Figure 5.4: Final trained network.

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

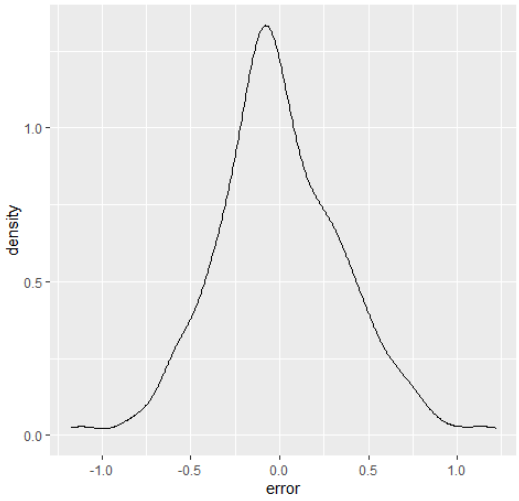
The first thing to look for is the normality of the errors in the predicted values. If the errors are normally distrubted (and ideally around 0), then it means that our predictions are sufficiently accurate. We produce a density plot of the errors using the *ggplot2* library in R.

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.

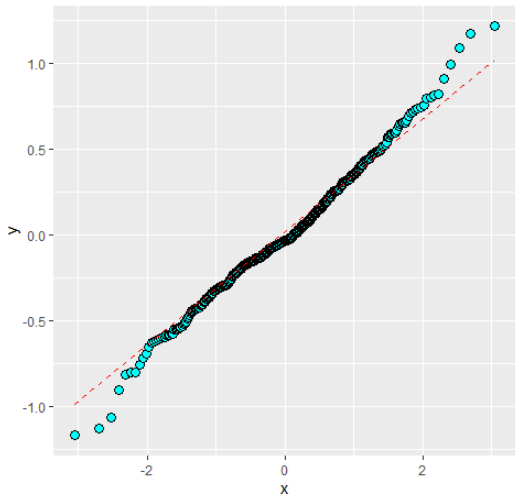
The issue that one may point out here is that this network has been trained on a full-over dataset, and then been tested on a full-over dataset. But the purpose of this investigation has been to look at the scenario in which a full game has been completed. So the method is currently ineffective at doing 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 **??** 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



[Distribution of Errors]



[Q-Q Plot of Errors]

Figure 6.1: Error distribution with Q-Q plot

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 $\mu,\ \sigma < \infty$. Then*

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

It follows from Theorem **??** 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 following table.

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

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 **??**.

```
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
```

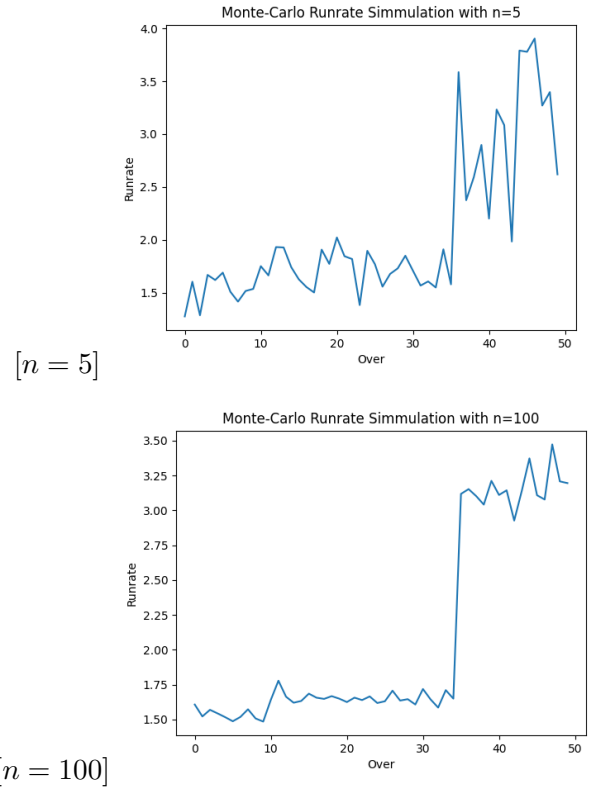
Figure 6.2: Implementing a Monte-Carlo Simmulation fo Cricket Matches

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, while 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 **??**, 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,



$[n = 5]$

$[n = 100]$

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

$n = 20$ 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 chapters.

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.

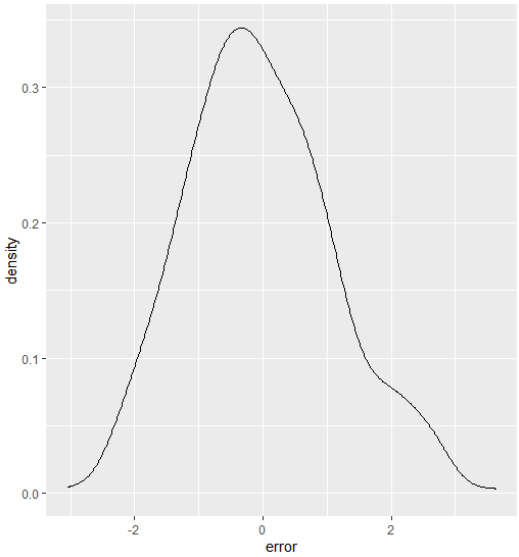
We can see in Figure **??**, that the difference is mostly centered around 0, which is promising, but the fact the corrolations are close to 0 means this is not a reliable method for predicting cricket scores. This is clearly an issue, because when it comes to resetting scores, the game obviously wont 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 scaled using the base-R function *scale()*. This applies the function

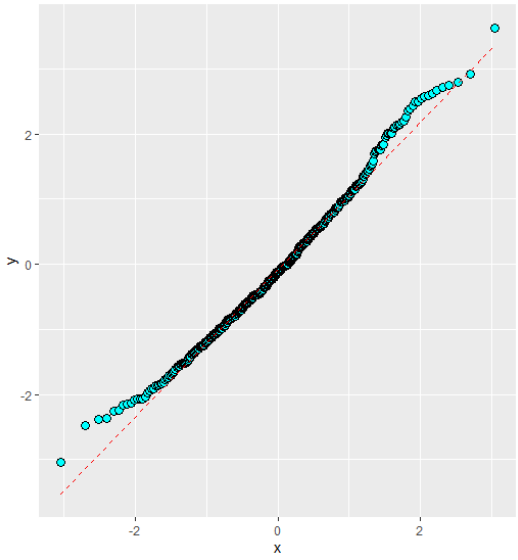
$$x_{\text{scaled}} = \frac{x_{\text{original}} - \mu}{\sigma}.$$

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.

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



[Error in Predictions from Monte-Carlo Simmulation]



[Q-Q Plot of Monte-Carlo Prediction Errors]

Figure 6.4: Monte-Carlo Simmulation Error Density

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.

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

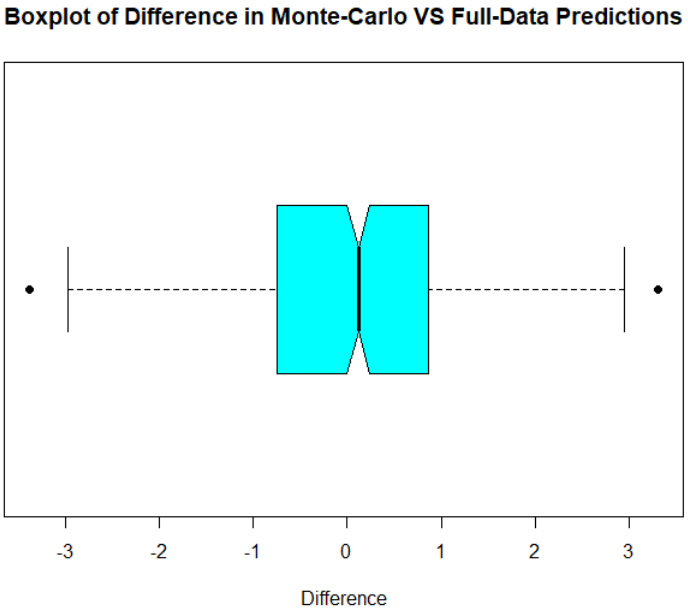


Figure 6.5: Boxplot of prediction differences

```r
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

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 **??** how the spread of points isn't in a narrow band as it was with monte carlo predictions.

We can also, as is standard practice, look at the errors of the predictions made by this model.

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.

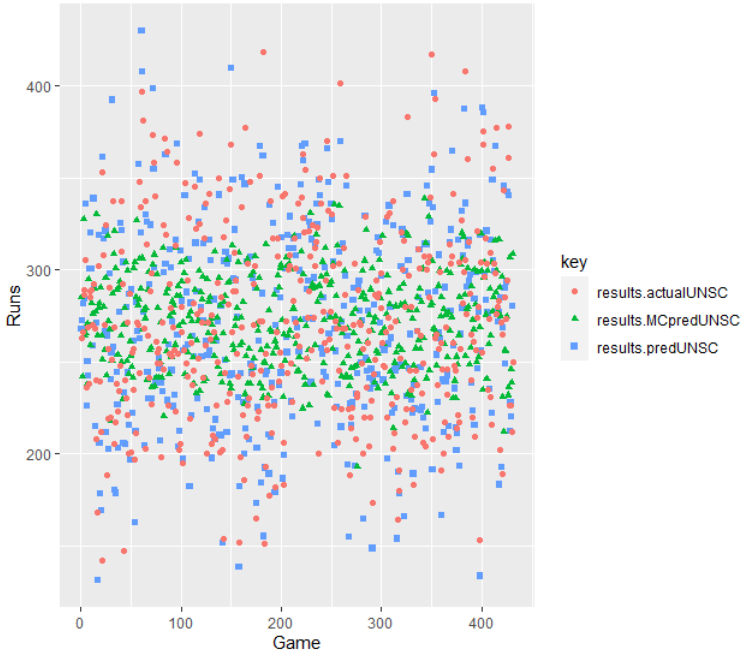
6.5 England VS South Africa, 1992 World Cup

In this section, we take the models that were produced in the prior chapter and apply them to the controversial game at the 1992 Australian Cricket World Cup. In the second semi-final, played between England and South Africa at the Sydney Cricket Ground. England had not completed their 50 overs by 18:10pm, and so the number of overs was reduced to 45 overs. In South Africa's innings, rain stopped play 5 balls into over 43. At this point in the game, South Africa were 231/6 with 13 balls left, chasing 253. The game was reduced to 43 overs, and using the *most productive overs method*, South Africa were set a target of 252 off 43 overs, leading to the impossible requirement of 21 off 1 ball.[1]

Note the afforementioned *Most Productive Overs Method* is given by the equation:

[1] At the time, the electronic scoreboard at the ground, and the TV coverage incorrectly displayed 22 to win off 1 ball.

Figure 6.7: Plot of predictions for all methods



$$\text{Target in X overs} = \text{Runs scored by Team 1 in their highest-scoring X overs} + 1. \tag{6.4}$$

A Duckworth-Lewis calculation was done retrospectively, and would have first set South Africa a target of 273 fron 45 overs, then 257 from the 43 overs.

Figure 6.8: Network predictions for DLS games

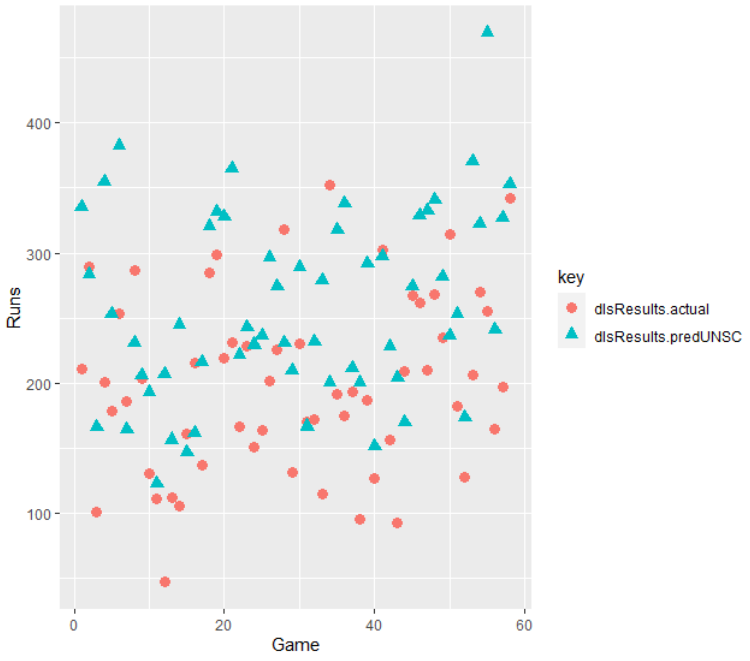
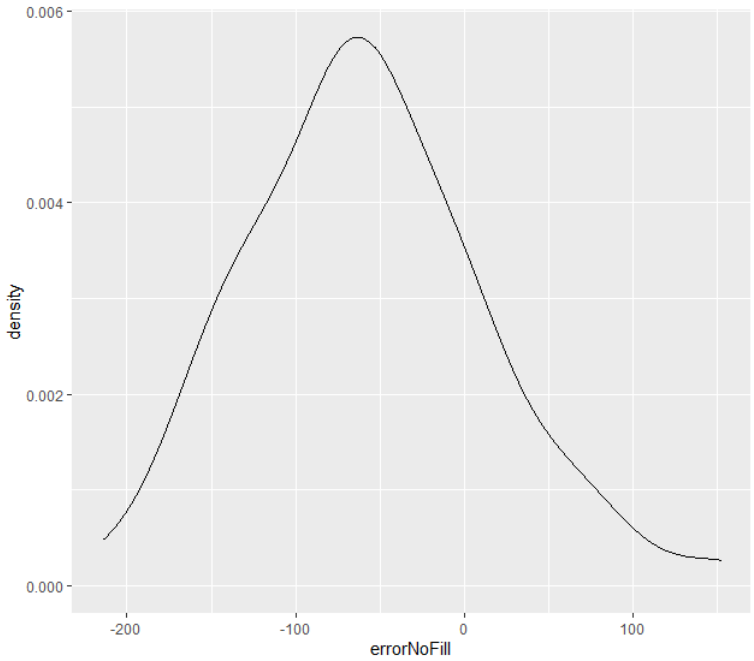


Figure 6.9: DLS Prediction Errors



Chapter 7

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

7.1 Conclusions

7.2 Discussion and Future Work

Following on from what was mentioned in Section **??**, rather than simply decreasing scores in proportion with the number of overs loft, it would be worth investigating decreasing the score depending on the fall of wicket distribution, as we looked at earlier in the paper. This could be done using Bayesian Techniques.

It's worth looking at expanding on the monte-carlo method; more specifically, breaking the game up into smaller stages to see how the accuracy of the model changes.

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.