Investigating the Predictive Performance of a Simulation Based T20 Cricket Model

University of Exeter

Alex Michaels

December 2021

Contents

1 | Introduction

In this thesis, I present the methodology behind a statistical model I created, which attempts to forecast the outcomes of and within[1] Indian Premier League (IPL) Twenty20 (T20) cricket matches. I then test the performance of the model against two distinct benchmarks.

Knowledge of the game of cricket, particularly the T20 format, is assumed throughout. If the reader is unfamiliar with how the game works, a short primer can be found in appendix A.

The model is trained on ball-by-ball data from the IPL's inception up to, and including the 2020 season. We test the model on predictions it makes about the 2021 season, comparing these forecasts to those from the two benchmarks. The first benchmark is historical odds from the sports betting market. I anticipate the sports markets to be a tough test for the model to overcome, especially given the decently liquid markets[2] for games in prized tournaments such as the IPL. The second benchmark we use for evaluating the model's performance is another, much simpler simulator. This has the benefit of being able to test the model's performance in areas beyond simply match outcome. With this ball-by-ball modelling approach, almost every event in the match is simulated. The results of these simulations can therefore be compared against what happened in the real world.

1.1 Motivation

The motivation behind this project comes out of sheer curiosity to see whether this type of modelling could be effective in predicting the outcome of games. When considering the problem of modelling sports more generally, my first instinct is to naively think something like, 'how can I most authentically replicate this game, from my computer?' I decided on the sport of cricket, given that it was clear to me that a simulator of this kind mimics the game of cricket especially well. For

[1] I define the outcome of a match as simply, the team that wins the match. In contrast, an outcome within a match includes answers to questions such as: which player scored the most runs? Who took the most wickets? Which team will score the most runs in the first six overs of their innings?

[2] More liquidity tends to translate to high accuracy/efficiency of the market.

example, one key factor in determining the outcome of a given delivery is the specific match-up between the batter and bowler. My knowledge of the game tells me that these match-ups can have a great deal of impact on the respective probabilities for each possible outcome of that ball. A ball-by-ball simulator accounts for these possible match-ups in a way that's superior to any other approach I could think of. Before each delivery, the model will check to see who is the bowler and who is the batter, amongst other variables related to the state of the game at the time. Once these factors are known, it can sample from a distribution of probabilities, determined by these inputs. A more traditional modelling approach might take a more top-down stance. It may take team line-ups as an input and output some expected number of runs for each team across each innings as a whole. Where this approach falls short, in my opinion, is that it fails to attribute significant weight to the individual match-ups, at the heart of the game. Moreover, in the format of T20, the outcome of each individual ball, has significantly more impact on the result of the match, compared with longer formats of the game. It therefore seems appropriate to try to model the game at the most granular level we can - that of the ball.

1.2 Aims of the Project

The ultimate aim of the project is to build a model that gives accurate predictions in all areas. A ludicrously lofty ambition, I am aware. If we are able to manage that however, the possibilities for how a tool like this could be used are endless. It could guide and assist coaches in selecting optimal line-ups for their team. This includes solving problems like finding the batting order to maximise win-probability for a given match. Or answering questions such as, "how does a team's win-probability change if they sacrifice a good middle-order batter for an excellent extra bowler in their line-up?"

We could also utilise this tool in betting into the liquid IPL prediction markets when the next tournament comes around in the spring of 2022. With a model that can in theory return us probabilities for any outcome we choose, there would be no shortage of opportunities for us to bet at bookmakers and provide liquidity on the exchanges.

I know that achieving anything even remotely close to what I have just described above will be a tremendous result. I must emphasise that the point of this experiment is not to build something of the description above, but to test whether something like this *can* be built.

1.3 Overview of the Model

The model that I propose can be broken down into several parts. The first is essentially a multi-class classification problem. Given there are only a finite number of possible outcomes that can happen from each ball, a probability can be assigned to each of these by the model, given a set of input parameters. A typical distribution based on frequentist probabilities across the dataset can be found in fig. 1.1. Incidentally, this is the exact distribution used in the benchmark simulator.

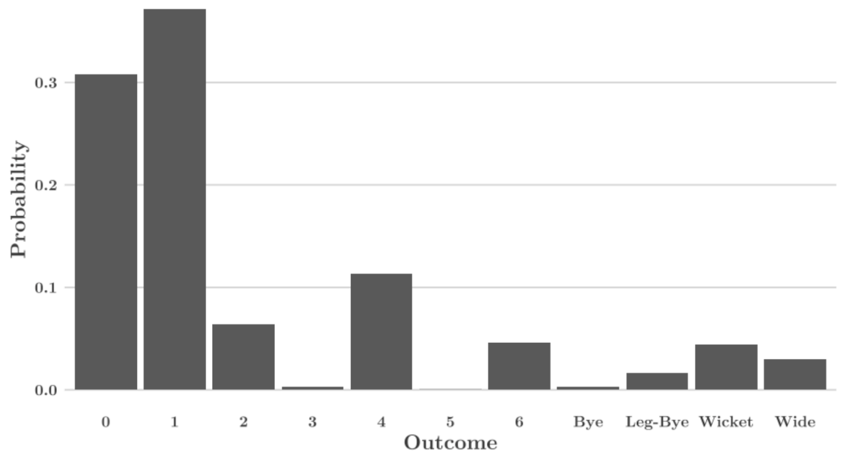


Figure 1.1: Showing the probability distribution of the 11 possible outcomes from a delivery.

Based on which outcome gets sampled here, subsequent questions are asked, and the responses are sampled in a similarly probabilistic manner to above. For example, it is possible and not uncommon for a no-ball to be bowled by the bowler, in addition to the batsman scoring runs. This is why 'no-ball' is not one of the 11 initial outcomes. So if the outcome is one from zero to six runs, we sample to check for a no-ball. Run-outs are also possible in addition to runs and byes (assuming no boundary is scored) so this is checked and sampled at the end of a 'ball' simulation. As we'll discover in chapter 2, this practice of going beyond the initial outcome sampling is a step further than what has been done before in previous experiments of this kind.

2 | Literature Review

2.1 Prior Attempts at Solving Similar Problems

Throughout my research of this topic, I have come across many interesting ideas from other academics and hobbyists attempting to solve similar problems as I am here. One example comes from Swartz, Gill and Muthukumarana [1] in their paper, *Modelling and simulation for one-day cricket*. Another came from hobbyist, Andrew Kuo, aka. dr00bot [2] in the form of a blog post: *Predicting T20 Cricket Matches With a Ball Simulation Model*. In this section, I describe the methods and findings of both papers before comparing their choice of methods and suggesting ways that I can add to and improve upon their research.

2.1.1 Swartz et al.

Introduction

In the paper, they describe their process for building a simulator for one-day international[1] (ODI) cricket matches. They approach this problem in a similar way to me – realising that there are a finite set of outcomes that are possible for each ball. Each of these outcomes is then assigned a probability based on historical ODI data.

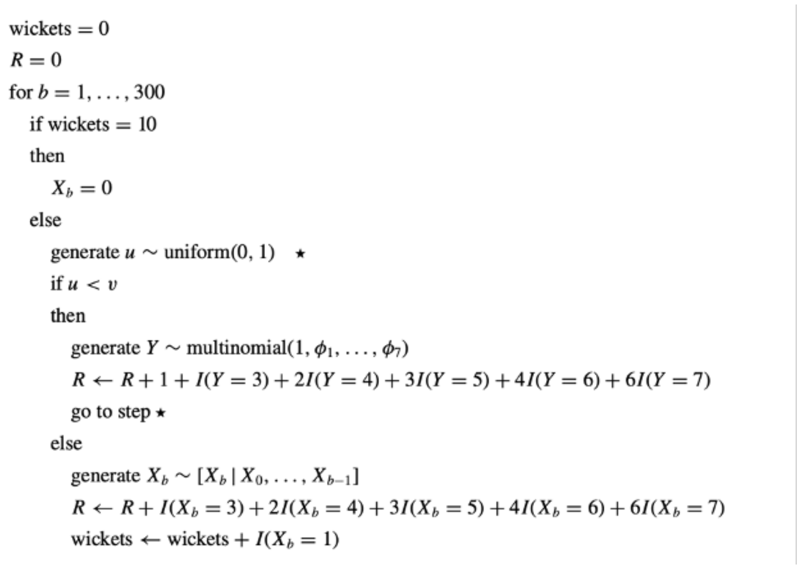
A significant goal of theirs was to answer game-specific and strategy related questions. For example, 'what would be the expected outcome of England changing the order of their third and sixth batters?' They concluded that a simulator would be necessary to best answer these types of questions.

[1] 50 overs per innings

Methods

Their simulation proceeds as follows: A sample is drawn from a uniform (0, 1) distribution and a no-ball/wide is assigned if the sample is less than a defined threshold. In this case, a single run is added to the batting team, and the ball is not counted as part of the limit of 300 for an ODI. Additional runs may be scored in addition to a no-ball/wide, and these are sampled from a multinomial distribution, with the same outcomes as the main one described below.

Assuming no no-ball or wide, the simulator in this experiment samples from seven possible outcomes. (Wicket, 0, 1, 2, 3, 4, 6.) Here, 'wicket' includes run-outs, and leg-byes and byes are included within the results of 0-6. Figure 2.1 is taken from their paper and describes the logic nicely.



```
wickets = 0
$R = 0$
for $b = 1, \ldots, 300$
  if wickets = 10
  then
    $X_b = 0$
  else
    generate $u \sim$ uniform(0, 1)   ⋆
    if $u < v$
    then
      generate $Y \sim$ multinomial$(1, \phi_1, \ldots, \phi_7)$
      $R \leftarrow R + 1 + I(Y = 3) + 2I(Y = 4) + 3I(Y = 5) + 4I(Y = 6) + 6I(Y = 7)$
      go to step ⋆
    else
      generate $X_b \sim [X_b \mid X_0, \ldots, X_{b-1}]$
      $R \leftarrow R + I(X_b = 3) + 2I(X_b = 4) + 3I(X_b = 5) + 4I(X_b = 6) + 6I(X_b = 7)$
      wickets ← wickets + $I(X_b = 1)$
```

Figure 2.1: The logic of the Swartz simulator

Swartz et al. model each ball as a conditional distribution where the outcome of a given ball is conditional on the outcomes of the balls that have come before it. Figure 2.2 below shows this. X_b refers to the outcome of a given ball, where $b = 1 \ldots 300$.

One key feature of the modelling approach applied here is the significant distinction made between first and second innings. For the first innings, only the following factors are considered as inputs to

$$[X_b \mid X_0, \ldots, X_{b-1}]$$

Figure 2.2: The conditional distribution which models each ball

the model: the batter, bowler, number of wickets lost, and the number of balls bowled. A Bayesian latent variable model is developed to get estimates for the probabilities of the possible outcomes of X_b. This outputs probabilities based only on the batter and bowler. The output is then updated by a neat addition which is termed 'batsman aggressiveness'. This modifies the output of the model based on the state of the game at the time (wickets remaining, balls remaining) where the batter's willingness to take more risks will change. The model is fitted in winBUGS and 851 unknown parameters are estimated – the vast majority of these corresponding to each of the batters and bowlers in the dataset.

In the second innings, the conditional distributions also depend on the first innings score, as well as the batting team's current score in the match. The first innings model is modified to account for these extra variables. This is done by considering the batting team's balls and wickets remaining as combined 'resources' which are required to be 'spent' in order to score runs. This idea is taken from Duckworth and Lewis. [3] [4] A formula is devised which modifies the probability distribution calculated using the same model as in the first innings. As an example: take a scenario where the team batting second has many runs left to score to win the match, proportional to relatively few resources with which to score them. This would be a scenario where we would expect the batter to play more aggressively. This has the effect of increasing his probability of dismissal, and his probability of an optimal score of six runs. Consequently, the probability of a more typical outcome like a score of zero, or 1 run, decreases.

Conclusions

Recall that the goal of Swartz et al. in this experiment was to build a simulator to be able to answer complex game and strategy specific questions. This, in my opinion, proves to be a difficult task to assess comprehensively and with quantitative rigour. Here, we outline the steps they took to assess their model.

The first test is to see how the predictions change as the model is presented with differing scenarios. A fixed batter, A Cook in this case, is pit against two bowlers of differing abilities, G McGrath and N Hossain. As expected, Cook's expected runs per over are smaller against McGrath than Hossain. (McGrath is widely perceived to be a superior bowler than Hossain.)

Next, these batter/bowler match-ups remained the same (Cook vs McGrath) and the game-state inputs were changed. It was discovered that on ball 1 of the innings, with all 10 wickets remaining, Cook's expected runs per over are low at 3.7, and his probability of dismissal is also small (0.024). As we fast-forward to ball 271 (out of 300) with just 2 wickets lost – a situation that calls for significantly greater risk and aggression to be deployed by the batter – Cook's expected runs per over do in fact increase substantially to 6.1 while his probability of dismissal also increases to 0.039.

We can therefore say that the model is performing well in accounting for game-state here.

More pertinent to my investigation is their test where actual runs scored are compared to simulated runs scored across first innings. This should give us a much better representation of how this simulation-based model performs in a predictive context. For this test, 23 matches between Sri Lanka and India were selected where Sri Lanka batted first. These included 15 matches from the original training data and 8 external to the training data. For each match, the real-life batter and bowler orders were recorded for the first innings, and the simulator was run, for 1000 times per match. The simulated runs were compared against the real-life outcomes. Results proved favourable towards the model, and these are shown in fig. 2.3 below.

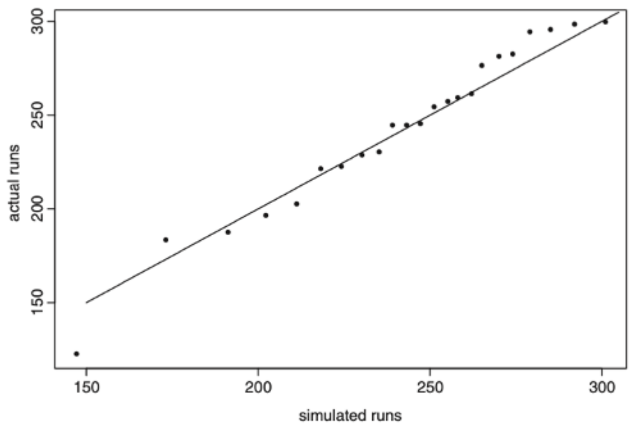


Figure 2.3: QQ-plot showing simulated first innings runs vs actual first innings where Sri Lanka bat against India. Taken from Swartz et al.

The final test is to check the effectiveness of the second innings adjustment for batter aggressiveness. By considering simulated matches where Australia bat second, comparisons can be made when the simulator is set to output probabilities prior to the second innings adjustment versus including the adjustment proposed in the model description. In the simulation including the adjustment, Australia was found to have used up their full 50 overs on 8.5% of occasions. This is in comparison to a result of 13% when the second innings adjustment is not in operation. Comparing these benchmarks to real-life observations, we see that when Australia batted second they used up their full allocation of deliveries on 7% of occasions. Suggesting that the model is working as intended when making these adjustments.

2.1.2 dr00bot

Introduction

The second cricket simulation idea I have taken inspiration from is a blog post by Andrew Kuo, also known as 'dr00bot'. It was originally published on the Towards Data Science site, but can be found with no sign-in required on his website, linked here. [2]

The goal was defined by Kuo to 'experiment with a probabilistic, bottom-up approach to modelling'. Rather like me, it seems he was curious to see how these methods would perform at prediction.

The dataset he used was comprehensive, comprising ball-by-ball information of 3651 T20 matches across 7 leagues between 2003, and 2020.

Methods

As in the Swartz paper looked at earlier, Kuo identified the crux of the project to be the generation of a probability distribution of all possible outcomes from each ball. This is termed the ball prediction model. The methods used in generating the distribution are significantly different to Swartz et al. Instead of treating the batter and bowler as factors within the model (where each batter and each bowler in the dataset has an individual parameter associated with them) Kuo inputs the bowler and batter's respective historical statistics. This is a nice solution as it massively reduces the number of parameters required to be estimated – resulting in a model that is far less likely to suffer from overfitting. [5] In addition, some variables related to 'match state are also given as inputs to the model. These are innings, over, number of runs scored, number of wickets taken and first innings score (if we're in the second innings). Each of these inputs was given to a trained feed-forward neural network model. The network comprised two dense layers each of 50 nodes, using the ReLU activation function on each node. It is a shame that more details about the intricacies of the model were not shared in the article.

The simulation engine is devised in a similar fashion to Swartz et al. above. One key difference is that wide balls are dealt with in the main simulator. Another is that there is peculiarly no mention of byes and leg byes and how they are treated. I assume that these are baked into the results corresponding to runs. This of course leads to the issue that runs for a given batsman might be ever so slightly overestimated in this model.[2]

The simulation proceeds as follows:

A coin-toss commences the match. This decides which team bats first. The ball prediction model

[2] Byes and leg-byes are recorded as extras and not attributed to the on-strike batter for the ball on which they occur.

is then run with the current match state and relevant bowler and batter stats as inputs to the model. A random sample from the distribution output is then taken and the match state and team states are updated as determined by the sampled outcome. This process repeats until the innings is complete.

Conclusions

To test the model, Kuo took 1126 matches from his dataset and simulated them 500 times each, noting that the number of simulations was limited to 500 for reasons of computational efficiency. He compared the results of his simulations against the outcomes in the corresponding real-life games.

The first test was a crude visual comparison between simulated matches and real-life matches to check to see how well his simulations compared with real-life games from the point-of-view of realism.

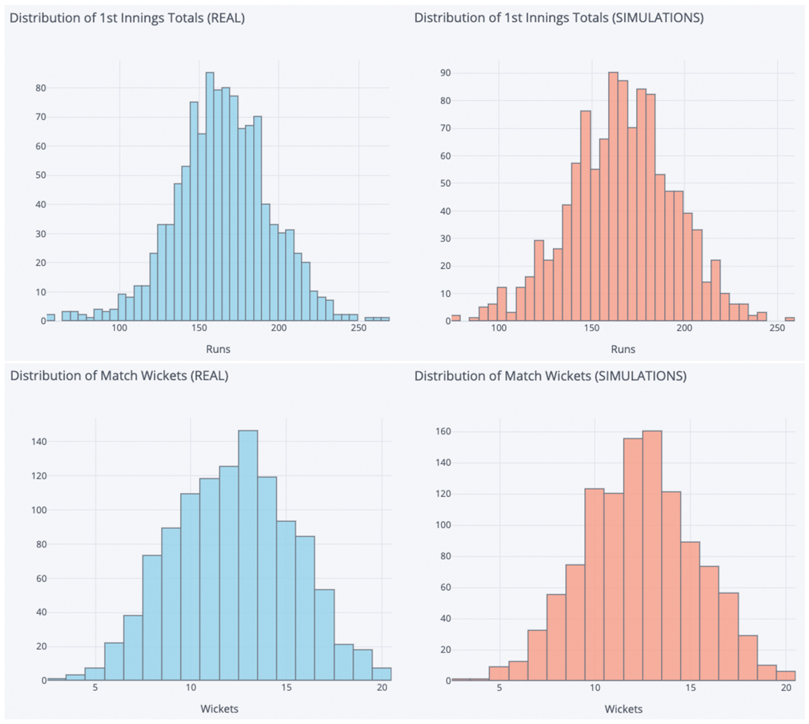


Figure 2.4: Results from Kuo's simulations show a very reasonable likeness between his simulations and outcomes in real matches.

Looking at the graphs in fig. 2.4 the conclusion was made that the simulations were a good representation of the real-life games.

A more quantitative test followed this where match outcome accuracy of the simulation model was tested. This yielded an accuracy of 55.6%, corresponding to a log-loss figure of 0.687. When this is compared to historical betting odds from Bet365, Kuo's model outperforms them. The bookmaker predicted the winner with an accuracy of 54.2% and a higher log-loss score of 0.695. While encouraging, the author notes that a model with no information (i.e. guessing the winner with probability 0.5) corresponded to a log-loss of 0.693. Meaning that the Bet365 odds are less accurate than a naïve model with no information! The conclusion drawn from this is that predicting T20 match results is really rather difficult. A hypothesis is posed that accuracy could be improved if simulations were possible beyond the limit of 500.

The second part of the results looks at the model's performance for outcomes within the game. The first test is first-innings runs. The results for which can be found in fig. 2.5.

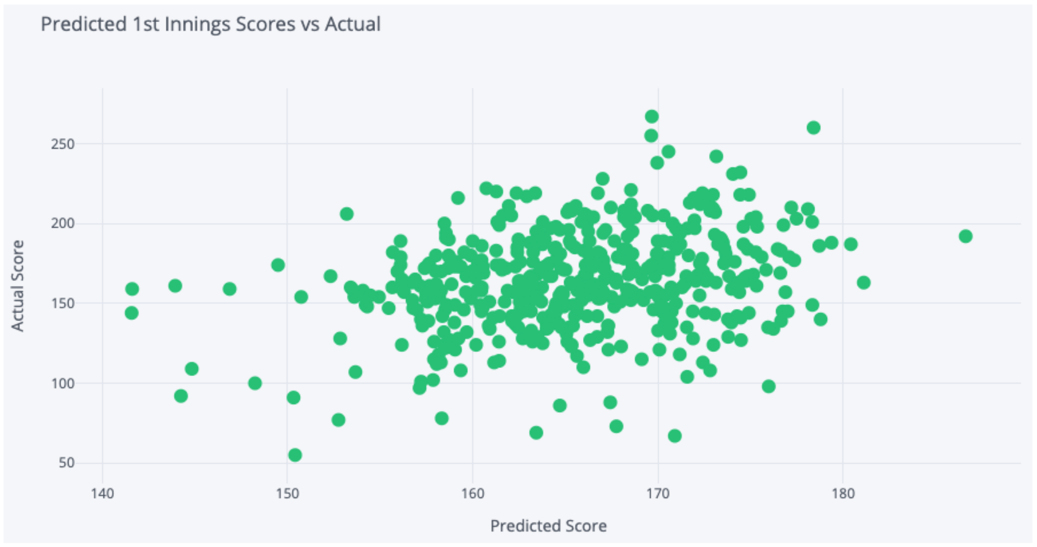


Figure 2.5: Predicted first innings scores against actual first innings scores. A general trend is detectable, but it's noisy.

The correlation between actual runs and predicted runs comes to 0.305 with a p-value of essentially 0. The model clearly captures the underlying factors that go into a successful prediction however the high standard deviation (28.6 runs) suggests that several other key variables are missing from the model.

The final two tests were concerned with trying to predict the top run-scorer and top wicket-taker in a given match. Kuo determines the model's prediction for these by selecting the player who was the highest run-scorer/wicket-taker in the most simulated innings. The results were that the highest run-scorer in a given innings was predicted with around a 25% success rate. The highest run-scorer appeared in the model's top 4 selections on 80% of occasions. For the highest wicket-taker, the raw success rate was 35%. On 75% of occasions, the highest wicket-taker in the innings appeared in the model's top 3 selections.

2.1.3 Comparing the Two Approaches

Both papers identify the crux of this problem to be generating a probability distribution of a number of categorical outcomes. The Swartz paper uses Bayesian methods to estimate the parameters of the model, while the Kuo paper uses a neural network. Here, I outline the arguments for each of these methods and consider ways of improving upon them.

The Bayesian techniques applied by Swartz et al., are inherently stochastic, and add an extra layer of randomness and chance to the system. This is exactly what we want in a stochastic simulator. Before the model is run, each parameter input is sampled from a posterior distribution. Only then are these sampled parameters fed into the model to output the final distribution of probabilities. It's noted in the paper how this is initial sampling is analogous to accounting for the form of the players involved in a given delivery.

The downside to this technique is the number of parameters used as inputs to the model. In this case, each of the more than 800 bowlers and batters in the dataset was assigned a parameter. This could result in the model over-fitting the data. Evidence against this suggestion was conspicuously lacking in the paper. A way in which this could be mitigated, while staying true to the underlying Bayesian principles could be to use a clustering analysis technique, to categorise the batters and bowlers into clusters, based on the similarities of their historical statistics. Some experimentation would likely be required to find the optimal number of clusters. Another advantage to this would be that the parameter values would be trained on a much wider pool of data. Rather than the model being trained on historical information for a specific bowler (who may have bowled very few deliveries in the dataset) it can consider deliveries from many more bowlers of a similar profile.

Kuo's neural network takes far fewer parameters as inputs compared with the Swartz model. Here, batters and bowlers are not treated as factors. They are represented solely by their historical statistics. This is ideal, as the model can still use information from the entire dataset when making predictions for future outcomes. It is also the case that in a deterministic set-up like this, players with similar past statistics will be expected to perform similarly into the future which lines up with our intuition.

We saw in the results section in fig. 2.5 that Kuo's model succeeded in capturing the underlying features that go into predicting the score of a T20 innings, but the relationship between the predicted score and the actual score was riddled with noise. It may be that a more complex model might give predictions with greater precision and a smaller standard error. One area where significant detail can be added is in the simulation. Kuo includes 8 of the most probable outcomes but I think more scenarios could have been added. No mention is made to no-balls and the resulting free-hit, nor is any attempt made at dealing explicitly with byes and leg-byes. Improving in this area could result in increased predictive performance. Of course, it may also be the case that more variables influence these scores than Kuo or I have to model with. Weather, pitch conditions, ball conditions[3] and many other factors, are all variables that I don't have access to in my data that could conceivably have a significant impact on projected runs scored in an innings.

Having considered both modelling approaches carefully, I am more inclined to use Kuo's machine learning approach for my model. One reason for this is that Bayesian computation is often extremely expensive computationally. Another is that I see inputting batters and bowlers as factors to be a significant disadvantage for reasons outlined above. For my purposes, a more classic machine learning model makes the most sense. In the next section of the literature review, I look at machine learning methods more closely and decide on a suitable one to use for my model.

2.2 Machine Learning Methods

Having decided on using a machine learning method for the main model, the next part of the literature review attempts to show how we arrived at a choice for the type of model that we'll use for this experiment.

We know that we need a machine learning tool that can handle multinomial classification. In addition to this, we require the output of the model to be a vector of probabilities, each representative of a certain outcome, which sums to 1, given each outcome is independent of all others. I choose, therefore, to investigate two methods that meet these requirements. The first is multinomial logistic regression (MLR), and the second is an artificial neural network (ANN).

2.2.1 Multinomial Logistic Regression

The short paper by Dr Jon Starkweather and Dr Amanda Kay Moske from the University of North Texas on MLR [6] was extremely useful to me in considering this technique's use in the model. It is essentially a primer on MLR, detailing the benefits and drawbacks of the method.

Based on Starkweather and Moske's assessment and my requirements, it seems that MLR would be an adequate tool for use in this project. However, one area of concern I have relates to a line in the article to do with training data. "Sample size guidelines for multinomial logistic regression

[3]By which I mean the state of the cricket ball itself, which is known to degrade over the course of a match.

indicate a minimum of 10 cases per independent variable." I was hoping to improve upon the Kuo model by increasing the number of output categories, which would result in some of these categories being considered very rare events, in the context of the data. I expect the model to have at least 20 inputs, which would mean I would need more than 200 instances of the rarest outcome (5 runs) in my data, which I do not have.

Another reservation I have about this method is in the construction of the model. MLR models are extensions of the generalized linear model (GLM) framework which are underpinned by the linear predictor function. Whilst this is a purposefully flexible framework, it may be the case that the inputs to my model (comprising many different variables on different scales) are unable to comfortably fit into this structure. A more flexible idea that offers similar functionality to MLR is the artificial neural network model, introduced in the following section.

2.2.2 Artificial Neural Networks

Before this project, I had no experience working with neural networks. The resources I talk about here in this section are therefore quite rudimentary, but I include them as they gave me the understanding that I now have of these systems, and enabled me to carefully evaluate their suitability for this project.

3blue1brown

The first reference is the YouTube series [7] from the channel, *3blue1brown*, created by Grant Sanderson. The series is a phenomenally well explained, comprehensive and ground-up introduction to feed-forward neural networks, with no prior experience assumed.

The series is split into four episodes. The first explains the structure of the network while walking us through the classic hand-written digits example to explain what's going on inside the network. Individual neurons are explained before the layers of the network are introduced. Then we discover how the many parameters (weights and biases) work to lead the network to output a certain prediction. Beautiful and easy-to-follow graphics accompany everything throughout the video.

Subsequent chapters get into the maths of the whole process a little more. Chapter 2 is dedicated to gradient descent - how neural networks learn. It introduces cost functions and explains how the gradient descent algorithm changes the parameters of the network to minimise this function.

Ghatak

The second reference here is a book by Abhijit Ghatak called *Deep Learning with R.* [8] All of the code that we use in building and running the simulator is written in the R programming language. This book was useful as it gave me the skills to implement the principles that I learned in the Sanderson videos in the development of my own models, that can be integrated seamlessly into the simulator.

Chapter 1 of the book, an introduction to machine learning, goes through many of the basic principles of fitting machine learning models to data. The bias-variance trade-off is talked about as well as over and underfitting. It also goes through and introduces many of the hyperparameters we can decide to tweak or add to, governing the training process of the build. Chapter 3 introduced the R package Keras used for building and training neural networks, which is what we use in the main model. Chapter 6 revisits the Keras package and explains the endless tuning options available to us, how to use them and why we might want to use them. It was a vital reference book for me during the model building phase of the project.

3 | Methods

3.1 The Data

3.1.1 Description

The dataset that we use in the experiment comes exclusively from Indian Premier League cricket matches. The data comprises two .csv files. The first contains information about each match, while the second shows detailed information on a ball-by-ball basis. It is this latter dataset, that we use almost entirely in the analysis. It documents every ball of every IPL game from the opening season of the competition (in 2008) through to the culmination of the 2020 season. The data is originally sourced from the cricsheet [9] website, a data collection project compiled by Stephen Rushe. I was fortunate to come across the data through Kaggle [10] in an upload by Prateek Bhardwaj. The Kaggle data was already pre-processed from its original form to an easy-to-work-with, .csv file type, meaning I could work with it straight out-of-the-box.

Each row of the ball-by-ball data details 18 variables. We're told where the ball was bowled in the context of the game (inning, over, ball number in the over) as well as information of the batsman, non-striker, and bowler. Next are variables relating to the outcome of the delivery: batsman runs, extra runs and an 'is wicket' logical variable. Finally, we have a series of descriptions of those outcomes, like dismissal kind (caught, bowled, lbw, etc.), player dismissed, type of extra runs (byes, wides, leg-byes, etc.).

In all, we have 193467 rows of data in the ball-by-ball set, from 816 IPL matches between 2008-2020.[1]

[1] I should note here that while analysing the data I noticed several minor errors which I corrected manually. These were all in relation to no-balls being declared byes in the data. They were easy to pick up on when my code picked up an extra delivery than it expected in these overs.

3.1.2 Data Wrangling

Whilst what we have is comprehensive, there was a significant amount of processing that we need to apply to the data, to obtain many other variables significant in impacting the outcome of a given delivery.

Outcome

The first variable to obtain is what I term the 'outcome' variable. This will act as the response variable in the main model. This is a categorical variable that describes the principal[2] outcome of a given delivery. The variable was designed in conjunction with the simulator, with each of the eleven categories representing a mutually exclusive outcome of a ball in a match. The categories of this variable are: runs off the bat from 0 to 6, wide, bye, leg-bye and wicket. The assignment of wicket is not given to balls where a run-out occurred. This is because it is possible, and indeed common, for at least 1 run to be scored during a run-out. Notice also that no-ball is not one of the eleven categories since runs are often scored during a no-ball. However, wides, byes and leg-byes are included in this variable because we know with certainty that 0 runs off the bat were scored.

Game State

Lacking in the data were any variables related to the current game state. This is therefore something that we need to infer from the data and add extra columns for. We add current runs scored, wickets remaining in the innings, and a first-innings score variable to the data. We also add balls remaining in the innings and runs required to win (if in the second innings). These final two variables were easy enough to obtain for most games but became really quite complicated for those games which had been shortened due to weather. Often these games had stoppages during the second innings, resulting in the umpires declaring fewer overs to be played. This then adjusted the second innings batting team's target score per the Duckworth-Lewis-Stern method, and of course, reduced the number of balls remaining in the innings too. In matches where this was the case, I resorted to scouring the espncricinfo commentary pages [11] which proved to be a godsend. Another game-state variable that we add is a logical for 'is powerplay'. The powerplay is the first 6 overs of each innings[3], where more restrictive fielding restrictions limit the number of fielders in the outfield (outside the thirty-yard circle) to just two (down from five in the remaining overs). This is designed to encourage attacking play and favours the batting team.

[2]I say 'principal' here since the simulator we build, described in section 3.2, allows for further events to take place during a delivery.

[3]Assuming a full twenty over innings. Powerplay overs are reduced in shortened games. Where this is the case, this is reflected in the data.

Players

Possibly the most important variables in determining the probability distribution of outcomes are the bowler and the batter. We take inspiration from the Kuo model here in gathering the historical statistics for the batter and bowler on a given ball and adding columns for these in the main dataset. The statistics that we use here are derived from the outcome variable. We get the number of 0s, 1s, 2s, 4s and 6s[4] that each batter has scored in their IPL career and divide by the number of legal balls faced (total balls minus wides and no-balls), to get a number that corresponds to 1s per ball, 2s per ball etc. We also get their wickets, byes, and leg-byes per ball.

The same method is applied to each bowler. We get the 0s, 1s, 2s, 4s and 6s conceded per ball bowled throughout the given bowler's IPL career. This is in addition to wickets taken per ball and wides, no-balls and byes conceded per ball.

Venue

The final two variables added are a logical variable to say whether the current batting team has home-field advantage and a categorical variable for the venue. Cricket is unlike many other sports in that the playing field has no strict dimensions. These often change substantially from venue to venue. Consequently, some smaller venues tend to see many more runs scored, and this variable attempts to account for this.

3.2 The Simulator

3.2.1 Simulating a Delivery

This section describes the way the simulator runs through a ball, from start to finish. It explains how we work our way through the range of possible outcomes and how this solution enables every conceivable outcome of a delivery in T20 cricket to be possible in its simulations.[5] See fig. 3.1 on page 21 for a detailed diagram of this whole process.

We begin each ball by sampling from the distribution of eleven possible outcomes. Section 3.3 explains in detail how this distribution is generated. The second step we go to, is dependent on the

[4]Notice there is no record of 3s or 5s. These are so rare, that I didn't deem them significant. It's also the case that a score of 3 or 5 is likely the result of a fielding error, and so little influenced by the batter.

[5]There are only two instances where this is not the case. The first is the exceedingly rare occasion of 5 penalty runs being awarded to either team. This tends to occur after instances of more blatant cheating such as ball-tampering, or repeated time-wasting after receiving a warning. The second scenario which the simulator fails to account for is the similarly rare event where a batter is 'retired hurt'.

initial outcome sampled. More often than not, this will be a score of 0-6 runs, in which case we check for the no-ball. If this check returns as false (it will >99% of the time) the simulator checks to see whether four or more runs were scored. If they were, we assume that the ball ran all the way to the boundary making a run-out impossible. If less than four runs were scored, the possibility of a run-out is checked for.

If the no-ball check is returned as true, the 'free-hit' variable is triggered to activate for the subsequent delivery.[6] The simulator also ignores the count of the ball in the over, and one extra run is added to the batting team's score. We then check to see whether we arrived at this point via an initial outcome of 0 or wide. We check for this because we know that outcomes of 1-6 are runs off the bat, and therefore no byes can be scored. If the outcome is a 0 or a wide, then we sample to check for the number of byes (if any). As before, if the number of byes is less than four, the run-out is also sampled. If the initial outcome is byes or leg-byes, we jump immediately to this point of the simulator that checks for the number of byes.[7]

Finally, if a wicket is sampled initially, we make a further simulation to see whether the type of dismissal was 'stumped'. This is because it is possible for a wide to be bowled and the batter to be given out stumped. So, if the stumping check returns true then we also check to see if it was a wide ball.

Once we reach an endpoint, a number of variables relating to the events of the delivery are appended to. These variables are batter's runs, extra runs, byes, 'is wicket' and type of dismissal and player dismissed (if there is a wicket). We also track the innings, over and ball number in the over, as well as variables for 'is powerplay', 'is free-hit', balls remaining, wickets remaining, current runs scored, the batter, the batter's order in the batting lineup, and finally the bowler and if they're a spin bowler.

After adding to these variables we then need to update them for the next ball. The game-state variables we just apended to concern the state of the game prior to this ball being bowled. Now we update them for the subsequent delivery. The variables changed here are things like the balls remaining counter, wickets remaining counter, current runs. At this point, we also switch the batter and non-striker variables if an odd number of runs were scored.

These updated variables then get fed back into the main model and we go again.

[6]A note on free-hit balls: these by definition cannot result in a wicket (apart from run-outs). When this variable is activated, we remove the ability for a wicket to be sampled and instead sample from the remaining 10 possible outcomes using their respective probabilities. The fact that these no longer sum to 1 is no issue since the function we use normalizes them by dividing each probability by the sum of all 10.

[7]In these cases (where the sampled outcome is 'bye' or 'leg-bye') we know with certainty that a byes score of 0 is impossible. Therefore, the exact same process as above is applied to the score of 0 this time.

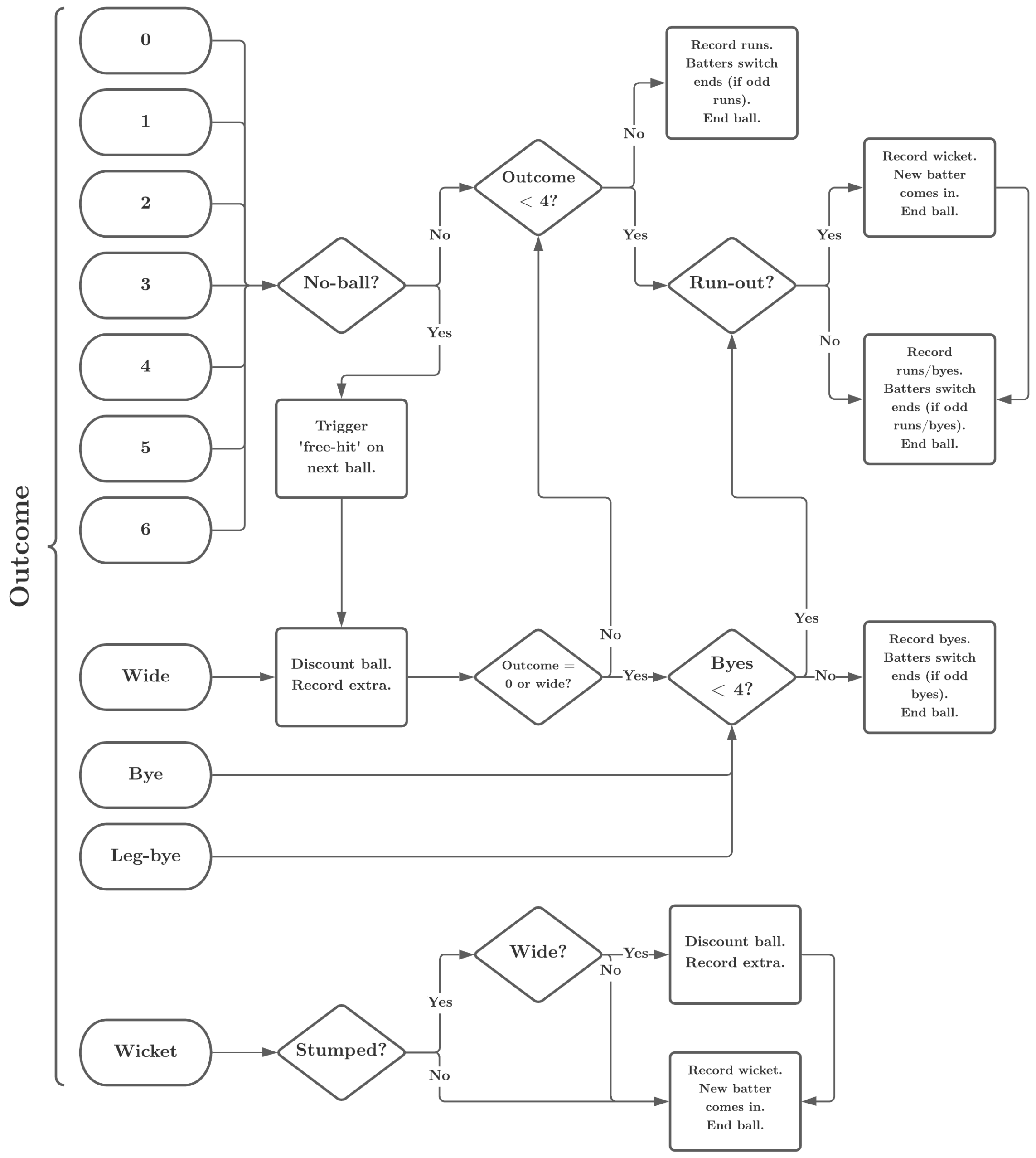

Outcome
0
1
2
3
4
5
6
Wide
Bye
Leg-bye
Wicket
No-ball?
No
Yes
Outcome < 4?
No
Yes
Record runs. Batters switch ends (if odd runs). End ball.
Run-out?
Yes
No
Record wicket. New batter comes in. End ball.
Record runs/byes. Batters switch ends (if odd runs/byes). End ball.
Trigger 'free-hit' on next ball.
Discount ball. Record extra.
Outcome = 0 or wide?
No
Yes
Byes < 4?
Yes
No
Record byes. Batters switch ends (if odd byes). End ball.
Stumped?
Yes
No
Wide?
Yes
No
Discount ball. Record extra.
Record wicket. New batter comes in. End ball.

Figure 3.1: Flowchart showing the decision-making process of the simulator during the course of a ball.

3.2.2 Simulating a Match

Now we have a framework for how to simulate a ball, we can extend this to running through the whole match.

Pre-Match Setup

This process begins by collecting some general information about the game which is needed for the simulator. This information includes the venue of the match, the team names of each side, and the lineups of each team.

Team lineups are presented in the order of the batting lineup. Handy for us, since this is one less variable to estimate. However, the team lineups don't give any suggestion into the likely bowlers or order of their deployment within an innings. So this is something that we need to think about here. First I consider how many overs each player might bowl. This is inferred manually, and almost entirely based on each player's average number of overs bowled, per appearance. The number of overs each player bowls is also fixed across all simulations of the given match. This is an okay solution, but it is a shame that I wasn't able to come up with a general-purpose algorithm that could perhaps be used for simulating this also. Next time! Once we have decided the number of overs each bowler is to bowl, a function is used to get the order of bowlers for each over in the innings. This is a stochastic process and does change for each new match simulation. The order is generated as follows:

1. Get lineup of the team along with the number of overs to bowl corresponding to each player.
2. Get the list of 20 bowler names. Each name in the list represents an over to be bowled by a player in the team lineup.
3. Shuffle the list of 20.
4. Check to see if this is an eligible bowling order.
5. If not, go back to step 5.

We define a bowling order to be eligible by the following two checks. Firstly it must alternate after each over. A single player bowling two consecutive overs is forbidden in the laws of the game. Secondly, we check that of the first four and final four overs, a minimum of six of these are bowled by the best bowlers.[8] This adds an extra layer of realism to the bowling orders since the important opening and closing overs tend to be the domain of the most potent bowlers.

[8]Bowlers bowing their maximum allocation of four overs.

Toss

Now we decide which team bats first in the match. In doing this we naively assume that both teams' preference for batting first vs second is equal. So a simple 50:50 virtual coin toss is all that's required here.

Initialising Variables

To prepare for the start of the match we need to initialise some variables which tell the model the state of the game before each ball is simulated. Recall that these variables are altered at the end of each ball, but we need to set them up here prior to the first ball of each innings. Variables include a logical 'is first innings', first innings score, batter, non-striker, next batter to enter, bowler, next bowler to bowl, current runs, wickets remaining, balls remaining, over number, ball number in the over, and the 'is free hit' logical variable.

Innings

To simulate an innings we simulate an over twenty times, which itself is a repetition of the delivery simulation six times. Between deliveries, we check to see if the wickets remaining reach zero, or if the target score is reached. This signals the end of the innings/match. At the end of each over, the batters switch ends, counter variables are updated and the bowler changes.

Output

The results of a simulated match are collected in three tables. The first shows the events of each ball in every simulated match, where each row represents a ball. The second table aggregates all of these simulations and gives the results of each simulated match, where each row represents a match. The final table aggregates results from each of the matches and returns a probability of various events occurring.

Find all of the code for these processes (and the rest of the project) in the Github repository, described in appendix B

3.3 Model Building

3.3.1 Main Model

The main model, used in the simulation to generate the initial outcome of a delivery, is a multi-layer perceptron. The model takes in 67 one-hot encoded variables to the input layer, which are passed through a dense hidden layer of 25 neurons, to an output layer of 11 neurons. The output layer applies the softmax activation function to each of the 11 possible outcomes, resulting in a probability estimation of each possible outcome where all 11 of these sum to 1. These are the probabilities that the simulator subsequently samples from.

Input Variables

When considering the game-state variables, I wanted to give the model a little more information than the raw numbers themselves when deducing what the relationship might be between some of these. I know from my experience watching a lot of cricket the effect that different game-states might have on the outcome of certain deliveries. For example, when the team batting first find themselves in the fortunate position of having many wickets remaining with only a couple of overs, say, remaining in their innings, they can take great risk, increasing the probability of both a boundary being scored as well a wicket falling.

We therefore include a variable derived from balls remaining and wickets remaining which I term 'aggression'. This variable is simply wickets remaining divided by balls remaining and maxes out at 1. A second variable that I create from modifying the usual statistics is the required run rate. This is applicable only in the second innings and is the number of runs required to win divided again by the number of balls remaining in the innings. In games where the team batting first wins very comfortably, this number can get quickly get ridiculous. We therefore again attach a maximum value to this one of 6, since this is the highest it can possibly get before the game becomes all but theoretically out of reach for the chasing side.

When considering the batter and bowler statistics and how these should be input, I had two main concerns to address. For players with very few appearances in the data, their statistics for bowling/batting were often quite strange looking, as we might expect. We also need to think about how when it comes to testing the simulator on new games, with players that we have no data for, how we could include some other variable that could imply their ability and make an educated guess as to their possible performance.

Considering the first problem: for players with limited data, I simply removed them from the tables containing batter and bowler statistics. The restriction that I applied here was that each batter needed to have faced more than 20 deliveries, and each bowler needed to have bowled more than 3 overs in their IPL career. These choices were admittedly arbitrary, but this seemed to do the job of removing the noisy data. The secondary benefit to doing this was that it exposed the model to the possibility of encountering new players for whom we may have no prior data. In these instances, we should try to use some other variable that is available to us as a proxy for their possible skill and likely performance. For batters, I used the position of the batter in the batting line-up.[9] For bowlers, I used the number of overs that the bowler is due to bowl in the innings.[10]

So, the final list of 28 variables input to the model is as follows:

- Venue
- Is first innings? (logical)
- Is powerplay? (logical)
- Home advantage[11] - batting team (logical)
- Is free-hit? (logical)
- Aggression
- Required run rate (set to 0 if ‘is first innings’ is true.
- Batter position in the lineup.
- Batter missing from data? (logical)
- Batter’s 0s, 1s, 2s, 4s, 6s, byes, leg-byes, wickets per ball.[12]
- Bowler’s number of overs in the innings.
- Is spin bowler? (logical)
- Bowler missing from data? (logical)
- Bowler’s 0s, 1s, 2s, 4s, 6s, byes, wides conceded and wickets taken per ball.[13]

These variables are fed to the model in a one-hot encoded format. This is because these types of models must have entirely numeric data fed to them. The way to solve this when working with

[9] Batters higher up the order are generally superior to those further down.

[10] In the training data I consider the average number of overs that the particular bowler bowls per innings, rounded to the nearest whole number. This is because the second innings is often concluded early meaning that not every bowler bowls their full allocation. In addition, games are occasionally curtailed due to weather, so using the actual number of overs bowled in these instances would also not be correct. I thought that average overs per appearance was therefore the best compromise, especially given that this metric was the one that I paid the closest attention to in deciding the number of overs that each bowler is to bowl for each match in the 2021 season.

[11] Due to the circumstances of the 2021 IPL season being played during the pandemic, very few venues were used. This resulted in no team ever playing in their ‘home’ ground. As such this variable remains set to 0 when running the simulations on the 2021 games.

[12] Set to 0 if batter data is missing.

[13] Also set to 0 if the bowler data is missing.

categorical variables is to create a column for each possible category (venue, say) and use a binary 1 in the column of the venue of the particular match and 0s in all the remaining venue columns. True/false logical variables are treated in much the same way, though integers, such as batter order, are not.

Hyperparameters and Tuning the Model

Now we have all the predictor variables, formatted correctly and ready to input to the model for training. In this training phase, we use a random 90% of the total data. The remaining 10% is used for validation at the end. The validation process is for us to make sure that the model is behaving as it ought to be before we come to use it for predictions on the 2021 season. It is during this training process where the model 'learns' the parameters (weights and biases) of the model resulting in optimal performance. The training process itself though is governed by more parameters and settings, set by us. These tweak the way in which the model goes about its learning process and can have a sizeable impact on overall model performance. I chose the hyperparameters of this model, mostly through an iterative trial and error process. These decisions are outlined below.

We choose the number of hidden layers to be 1 layer of 25 nodes, with the ReLU (rectified linear unit) activation function applied to each node. I wanted to keep the model fairly simple, given my lack of experience with these methods. I initially had the number of nodes in the hidden layer set to 40 - this being the average of the number of input and output nodes which seemed a reasonable starting point. Changing this down quite a long way to 16 resulted in slightly worse model performance (in terms of the categorical cross-entropy loss function). I subsequently tried a number somewhere in the middle and happened on a bit of a sweet spot at 25 nodes in the hidden layer. I tried increasing the number of layers to 2 layers of 25 nodes, though this had little effect. In preference of simplicity, I retained my initial choice of one hidden layer.

The loss function that we use when evaluating the model and any changes to the hyperparameters, is the categorical cross-entropy function. This is often used in models of multi-class classification (such as this one) and it essentially measures the accuracy of the output distribution. [12] Other additions we make to the model are adding batch normalization and dropout regularization to the hidden layer of the network. The dropout rate (which we set here to 0.2) is where nodes in the hidden layer are randomly powered off (with probability 0.2) during the training of the model. This has the effect of reducing the network's dependence on certain nodes, forcing the network into learning a more balanced representation of the data and thus reducing over-fitting. [8] Batch normalization is another modification to the hidden layer. This normalizes the inputs to the hidden layer which makes the network faster and more stable. [13]

The model ‘learns’ in a similar kind of trial and error way to us in the way we tweak the hyperparameters. It first looks at some rows of the data (a batch), and makes predictions for the distribution of possible outcomes. Then it tweaks those parameters used to make the prior prediction once it sees the actual outcomes that resulted from those prior sets of inputs. The loss function computes a kind of error score, quantifying the distance between the model’s output and the real-life outcome. This tells the model how good or bad its predictions were, at which point it then it goes back and alters the weights and biases of the network in such as way as to reduce the output of this loss function. It then repeats this process until the loss function reaches a local minimum. An optimizer algorithm is responsible for this process of re-tuning the weights and biases between each batch of data that the model ‘sees’. There are many choices available to us here - we use the Adam (adaptive moment estimation) optimizer. This function takes in another tunable parameter - the learning rate. This is a measure of how severe the changes to the model parameters are by the optimization algorithm. We set the learning rate initially to 0.0001, down from the algorithm’s default value of 0.001 since the learning curve was bouncing around too much using the default learning rate value. We then ask the learning rate to reduce when the learning curve reaches a plateau, enabling finer and finer adjustments to be made to the model parameters.

The final hyperparameters we set are the maximum number of epochs that the model trains for. It is the maximum since another setting we use, tells the model to stop training when the loss function reaches its local minimum. We use 300 - a recommended setting from Ghatak’s hugely helpful *Deep Learning with R* [8]. This is the number of times that the model sees the entire training dataset. The batch size I set to 50 - the number of rows of the training data it looks at before altering the parameters in accordance with the Adam gradient descent function. Lastly, we use a validation split of 0.1. As the model trains, it tests its current parameters on an unseen portion of the dataset (the validation set), giving us a validation loss score, as well as a training loss score.

Testing the Model

The final part of the process of developing the main model is testing it on the remaining 10% of the data, that we set aside before training. First, we take a look at the number of each outcome in the testing data and compare this to the model’s expected number of occurrences for each outcome across that same data. Given that the model gives us probabilities for each outcome, we infer its expected number for each of these by simply summing up each probability for all rows of the data.

Looking at table 3.1, we can see that the model performs well for predicting frequencies across all categories. We judge this on the p column. This p-value measures the probability of seeing at least the observed absolute deviation from the model’s expected numbers of each outcome, under the assumption of the Poisson distribution (where λ is the model’s expected number of observations).

Given all the p-values are above the 0.05 significance level, we can say that none of the observed outcomes differs significantly from the model's expectations, which is a good sign!

Outcome	Number Predicted	Number in Test Data	p
0	5932.47	5995	0.413
1	7189.05	7212	0.781
2	1270.05	1203	0.060
3	61.64	67	0.449
4	2181.01	2181	0.989
5	6.00	4	0.570
6	895.26	870	0.401
Bye	51.88	41	0.142
Leg-bye	317.83	324	0.702
Wicket	864.83	861	0.914
Wide	575.97	588	0.598

Table 3.1: Performance of the main model on unseen test data.

Most important to us in this experiment, however, is accurately predicting the total number of runs and wickets across a large sample. We can examine this in table 3.2, where the predicted total runs is calculated by multiplying the expected number of 0s, 1s, 2s...6s from the previous table by 0, 1, 2...6. This ultimately results in a p-value of 0.098. Notably smaller than the mean p-value across the runs categories in table 3.1 (0.524), it seems we're seeing the sizable impact of scores of 6 runs already. Whilst the p-value for total runs is small, it remains the case that the observed total number of runs, doesn't differ significantly from expectations. This remains the case for predicting wickets, where the model performs encouragingly well.

Outcome	Total Predicted	Total in Test Data	p
Runs	24039.67	23783	0.098
Wickets	864.83	861	0.914

Table 3.2: The main model's performance on the two most important categories: total runs and wickets.

3.3.2 No-Ball Model

Recall from section 3.2 that if the outcome sampled in the main model is from 0-6 runs, the no-ball is checked for. To sample for this we build a new model that returns the probability for a no-ball, given some inputs. In building this model, I originally attempted to use similar techniques as in the main model for predicting no-ball probability. However, the predictions made by this approach were shockingly bad. (Possibly due to a combination of the rarity of the event, and therefore a lack of no-balls amongst the training data. More likely due to my own poor tuning of the model.) Then I tried a logistic regression with several inputs that I deemed suitable (outcome, bowler, game-state, venue, bowler-type, etc.) which ultimately showed that only a single variable was significant in predicting the probability of a no-ball. This variable was the 'is spin' logical. As such, our model here is refreshingly simple. If the bowler is a spin bowler, we assign a probability of the no-ball to be 0.00162. Consequently, if they're a pace bowler, we set the no-ball probability to be 0.00599. These numbers are derived from the rows of data that were eligible to have been called a no-ball (i.e. not an already assigned bye, leg-bye, wide, or wicket). We split them, as before, into a training and test set using the same 90:10 ratio. Probabilities for each category are calculated as the frequentist probability of a no-ball occurring, given each bowler-type across the training data. Results of this approach on the testing set are given in the table below.

No-Balls Predicted	No-Balls in Test Data	p
77.20	70	0.451

Table 3.3: Performance of the no-ball model on unseen test data.

3.3.3 Byes Model

When the simulator reaches the point in a ball simulation where the number of byes is required, it inputs to the byes model, which returns a probability distribution of possible outcomes from 0-5 byes. This is the second and final multi-class classification model after the outcome model and is also one of the neural network variety. It uses a new variable, the number of byes, as its response. This is a number from 0-5 and represents only the number of byes scored on a given ball. So if there was a no-ball or wide, the 1 penalty run for this is not included in this count.

The inputs to this model total 7. We have logical variables for first innings, powerplay, free-hit, bowler type, the numerical game-state variables of aggression and required run rate and the category of the initial outcome. I found that a more simple tuning of the hyperparameters to this model was sufficient in obtaining a good performing model. As in the main model, we prefer simplicity here. So any additions made to the model resulting in a negligible change in performance were dropped. There is no batch normalization, nor dropout regularization applied here. The number of nodes in the hidden layer is reduced to 10 which works well. We retain the Adam optimization algorithm from the main model as well as the ReLU activation function on nodes of the hidden layer.

The model is developed using a specific subset of the ball-by-ball data where byes are possible. These are balls where the outcome is either ‘wide’, ‘bye’, ‘leg-bye’, or the balls where the outcome is 0 *and* the ‘is no-ball’ variable is true. This data totals just shy of 10000 rows. As before, we randomly split this up into a training and test set according to a 90:10 ratio. Below we have information on the performance of the trained byes model on the test data. Again, all p-values are greater than 0.05, suggesting that none of the observed counts differs significantly from the model's expectations.

Number of Byes	Number Predicted	Number in Test Data	p
0	559.72	541	0.442
1	331.93	343	0.522
2	22.56	27	0.298
3	2.48	5	0.081
4	58.82	60	0.810
5	0.50	0	0.792

Table 3.4: Byes model's performance on predicting the number of byes

3.3.4 Run-Out Model

The next question from the simulator that needs dealing with is deciding whether a run-out occurred on a ball for which this is a possible event. As in the previous models, we develop this model only on data for which a run-out is a possible outcome. This model is a logistic regression. It is reduced down such that only variables significant in predicting the probability of a run-out are present. This results in us using two separate models to estimate this since the required run rate (a variable only applicable in the second innings of the match) is significant. The second and final significant variable is aggression. Figure 3.2 shows the relationship between run-out probability and the aggression variable nicely. This kind of relationship is expected and exactly what I hoped for. I know that many more run-outs tend to occur at the end of each innings since the batters running have far less to lose at this stage of the game. It's encouraging that this model is able to capture this.

As before, we compare actual numbers of run-outs observed in the test data, against our model predictions. These p-values look good.

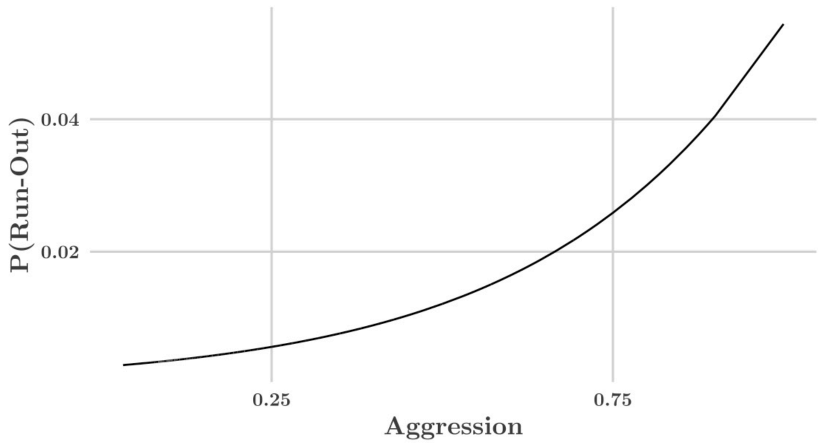


Figure 3.2: Relationship between the run-out model's predicted probability of a run-out, and the state of the innings at the time. Recall aggression is calculated as wickets remaining / balls remaining. It has a maximum value of 1.

Innings	Number Predicted	Number in Test Data	p
First	50.59	55	0.482
Second	36.83	39	0.644

Table 3.5: Run-out model's performance on predicting the number of run-outs in unseen test data.

3.3.5 Stumped and Wide Models

The final two areas to consider are the rare circumstances where a wicket is taken off a wide ball. This can only occur (for non-run-out wickets) if the batsman has been given out stumped. We can check for this by first evaluating the probability that a specific wicket is a stumping. If it is, we then check to see whether the delivery was a wide ball.

We first try using a logistic regression model for this. As was the case when training the no-ball model, we find that only the 'is spin' logical variable is significant in predicting the probability of a stumping occurring, given the ball is a wicket. So, as before, we end up with a probability of a stumping given the bowler is a spinner (0.0986) and the probability of a stumping, given the bowler is a seamer (0.00141).

The situation for wides on stumping balls is less clear-cut between the two types of bowlers, to the extent that no significant difference is present between the two. We use the aggregate probability figure of 0.0884 to estimate this.

These probabilities have been generated across the entire dataset (so no splitting for training/testing here). Given the lack of instances, I wanted to use as much data as I could in training these two models. It is also the case that had we split the data prior to training, we would be testing these models on a tiny test set, which wouldn't be useful.

4 | Results

4.1 Running the Simulator

The simulator was run on each of the 60 games of the 2021 IPL season. We were able to simulate each game 1000 times, resulting in more than 14 million simulated deliveries to analyse. The data that we have is presented in three tables. As detailed in section 3.2.2, we have one table of simulated ball-by-ball data, a second showing the results and other information pertaining to each simulated match (60,000 matches) and a final table that collates these results into probabilities and expected values of various events and outcomes in each of the 60 matches.

4.2 Testing the Simulator's Predictions

4.2.1 Picking Winners

Overall Accuracy

An obvious place to start seems to be in evaluating how good the model is at picking the winner for each match. A primitive and straightforward way of doing this is to simply take the team with the greatest win probability (estimated by the model's 1000 simulations of each match) as its 'pick' to win the game, and then sum up the number that it predicts correctly.

After doing this we find that it correctly predicts the winner on 30 occasions, exactly half of the total number of matches. A somewhat underwhelming result, though this test is far from conclusive. We can use the sports betting market as a benchmark and compare the two. The prices that we use here are the closing lines (prices immediately before the start of the match) from the international sportsbook Pinnacle[1], widely perceived to be one of the sharpest bookmakers in the world. [14] Repeating the same test above, using the favourite according to the Pinnacle odds as the selection, yields 32 correct guesses for an accuracy score of 53.3%. Not significantly higher than what our model managed, which gives us some hope. I did expect the Pinnacle closing lines to be more accurate than this.

Win Totals

A more rigorous test would be one that takes into account the model's certainty in one selection over another. The above test treats a selection of 51% certainty exactly the same as another with 75% conviction, which clearly isn't ideal. The following test accounts for this by summing up the projected win probabilities for each team across the regular, and postseason. This sum represents the expected number of wins for each team, which we can then compare to their actual number of wins. Again, we compare the model's projected win totals against the same benchmarks as before. These being the blind 50:50 guesswork of the null model, and the Pinnacle closing line probabilities.[2]

Team	Simulation Model	Null Model	Pinnacle	Actual
Mumbai Indians	8.33	7.00	8.17	7
Chennai Super Kings	9.43	8.00	8.06	11
Kolkata Knight Riders	9.01	8.50	8.63	9
Punjab Kings	5.18	7.00	6.26	6
Royal Challengers Bangalore	5.79	7.50	7.76	9
Delhi Capitals	8.03	8.00	8.42	10
Rajasthan Royals	6.19	7.00	6.29	5
Sunrisers Hyderabad	8.03	7.00	6.42	3
Mean Absolute Error	**1.891**	**1.750**	**1.534**	

Table 4.1: Comparing model expected win totals with projections derived from Pinnacle closing odds, the null model and the actual number of wins for each team across the 2021 regular and postseason.

As might be expected against the sharpest lines in the world, our model is handily outperformed by the Pinnacle odds. It also comes up short against the null model which is disappointing.

[1]Prices sourced from Odds Portal [15]

[2]With the bookmaker's overround removed.

4.2.2 First Innings Score

The second area we look at is gauging the model's skill (or lack of it) at predicting the first innings score of a match. To test this, we take the mean first innings score across all simulations of a given match as the model's estimate for expected first-innings runs. Then we compare these predictions to the observed first innings scores and calculate the mean absolute error. We compare this against the mean absolute error for predictions generated from a much simpler simulation engine, which acts as our benchmark.[3]

The logic of the simple simulator works in exactly the same way as the main one we devise in section 3.2. The key difference here, are the models used to generate probabilities for each event occurring. Where in the main simulator these probabilities change on each ball as the state of the game changes, in the simple version we use the exact same probabilities across all balls. These fixed distributions are derived from simple frequentist probabilities across the entire ball-by-ball training dataset.

Model	Mean Absolute Error
Main	26.53
Simple	24.75

Table 4.2: MAE of main and simple simulators predicting the number of first-innings runs.

4.2.3 Highest Individual Score

We can again use the simple simulator to benchmark our estimate for the highest individual score in the match. As before we take the mean highest individual score from the simulators as our expected values, and compare these to observed values using the mean absolute error metric.

Model	Mean Absolute Error
Main	18.10
Simple	17.07

Table 4.3: MAE of main and simple simulators predicting the highest individual score of the match.

[3]It's unfortunate that I wasn't able to find any historical betting lines for first-innings runs, nor for any other markets.

4.2.4 1st Over Runs

Now we test the model against the benchmark simulator as they attempt to predict first over runs.

Model	Mean Absolute Error
Main	3.216
Simple	3.594

Table 4.4: MAE of main and simple simulators predicting the total runs of the first over of the match.

4.2.5 Total 6s

The final test is to see how our model performs at predicting the total number of 6s scored in each match.

Model	Mean Absolute Error
Main	4.790
Simple	4.183

Table 4.5: MAE of main and simple simulators predicting the total number of 6s in the match.

4.3 Discussion and Limitations

These results are disappointing for our model. In all but one of the tests that we put it through, we find that a much simpler solution would have fit the observed data with a closer fit. Here, we attempt to figure out why that is and discuss limitations and potential improvements that could be made to the model to improve its predictive performance.

4.3.1 Ideas for Improvement

Update the Model as the Season Progresses

The model is trained only on data up to and including the 2020 season. As the 2021 season progresses, the model's predictions stay the same, regardless of how good or bad specific teams and players were performing in real-life at the time. In contrast, betting odds are updated constantly and will often move significantly from their opening lines to their closing lines, as new information about the game is ingested by the market.

This leads us onto a similar topic, concerning the historical data used to train the model. We treat each row of the training data the same as any other, regardless of when it was recorded. For example, a given player's performances from ten seasons ago are judged with equal weight as their statistics from last season. Clearly, more recent data carries more significance and this should be accounted for in the training process. I would very much welcome the opportunity to dig into this kind of research in the future. Perhaps some kind of decay function could be developed to diminish the impact of old data, whilst heightening the effect of more recent events?

Altering Aggression/Required Run-Rate Inputs to the Main Model

Something that occurred to me since training and testing the model is that I may have been better off not modifying the input variables relating to balls and wickets remaining in the innings. Given we're feeding them into a model which is repeatedly 'learning' the patterns of the input data, it may have deduced a relationship between these variables that is more nuanced and complex than the simple relationships that we forced upon it. It will be interesting to test this.

Changing the Validation Process

Going through the results of the model's simulations it's clear that it tends to over-predict runs. In the 'first innings score' test in section 4.2.2, the mean error (model prediction − observed first innings score) was 10.88. So on average, the model over-predicts first innings runs by almost 11 runs. In addition to this, carrying out the same test on predicted total 6s in each match yields an average overestimation of 2.12 6s per game. Looking back now at the validation checks on page 28, it seems that the over-predicting of total runs and the small p-value associated with that check likely was a cause for concern after all. An overestimation of the number of 6s shown in table 3.1 should have been paid closer attention to, as well as the overestimation in the number of 2s. Had I instead multiplied the number of predicted 2s, 3s, 4s, 5s and 6s, by their corresponding number of runs, I would have discovered the overestimation in both 2s and 6s to be significant (yielding p-values of 0.00756 and 0.0386 respectively), forcing me into making some changes to the model to address this.

4.3.2 More General Limitations

Computationally Intensive

Running these simulations is an intensive process and it does take a long time. Had we achieved good predictive results with this model we might find ourselves in a tricky spot. This is because lineups for cricket matches are often not released until half an hour prior to the match getting underway. In its current state, the simulator takes a lot longer than this to run through 1000 games, however, this can be optimised. At the moment it churns through each match one at a time, using only a single thread of CPU. However, the code can be changed to enable parallel computation across all available CPU threads, for a much quicker overall run-time.

Opaque Model

Neural networks are inherently opaque. It is very difficult to deduce what exactly the effect is of altering a specific input up or down some amount. This means that making improvements to the model could be tricky. For example, we just discussed the need to make changes to the model to lower the number of 2s and 6s it predicts, but we have no indication of a reasonable way of doing that! So we must resort to lots of trial and error, which is very time consuming and computationally expensive.

4.4 Summary

We have achieved what we set out to when starting this project. I wanted to build a simulation-based T20 cricket model and then test its predictive performance. This is what I've done. This results section shows that it didn't perform well, which is a shame. I suggest several ways to improve upon this initial prototype to have it make more accurate predictions, and I'm confident that implementing these can improve prediction accuracy.

A | How Cricket Works

I recommend watching this short two minute video which explains the game succinctly. Linked here: [16]

Transcript from the video

Cricket is played on an oval shaped field. In the middle of the field is a rectangle of hard turf known as the pitch. At each end of the pitch are wickets, made up of three wooden stumps to support two bails. The game involves two teams with 11 players making up each team. The aim of the game is to score more runs than the opposition. The umpires judge fair and unfair play. A match begins with one team going out on the field and the other comes into bat. The fielding team has a bowler, who uses different techniques to bowl the ball to the batter. The batter tries to hit the ball and score runs by hitting the ball and running between the wickets; making it to the other end of the pitch before the fielders can hit the wickets with the ball. Each time the batter runs a full length of the pitch, he scores 1 run for his team. Hitting the ball to the boundary, along the ground is 4 runs. Hitting the ball over the boundary without touching the ground is 6 runs - the highest scoring shot. In the meantime, the fielding team tries to get the batter out as quickly as possible in a number of ways. Knocking the bails off the stumps with the ball when bowling (bowled out); catch a batters shot, before it touches the ground (caught out); hitting the batter's leg if their leg was deemed by the umpire to be blocking the ball's path to the stumps (leg before wicket); or hitting the wickets before the batters can run to the other side of the pitch (run out). Each set of six legal balls bowled, constitutes an over. Our focus is on a shortened form of cricket, Twenty20. In this form of the game, each innings lasts for a maximum of 20 overs (120 balls). Once all the batters are out, or if they run out of overs, the innings comes to a close and the fielding team comes into bat, with the goal of chasing down the first innings score, within the allotted 20 overs.

B | Code

I have uploaded all the code to a repository on GitHub. You can find it by following this link: https://github.com/asmichaels/Cricket-Project---Masters

Bibliography

[1] T. B. Swartz, P. S. Gill, and S. Muthukumarana, “Modelling and simulation for one-day cricket,” *Canadian Journal of Statistics*, vol. 37, no. 2, pp. 143–160, Jun. 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cjs.10017

[2] A. Kuo, “Predicting T20 Cricket Matches With a Ball Simulation Model,” Jan. 2021. [Online]. Available: https://dr00bot.com/blog/t20-cricket-simulation-engine.html

[3] F. C. Duckworth and A. J. Lewis, “A fair method for resetting the target in interrupted one-day cricket matches,” *Journal of the Operational Research Society*, vol. 49, no. 3, pp. 220–227, Mar. 1998, publisher: Taylor & Francis _eprint: https://doi.org/10.1057/palgrave.jors.2600524. [Online]. Available: https://doi.org/10.1057/palgrave.jors.2600524

[4] ——, “A successful operational research intervention in one-day cricket,” *Journal of the Operational Research Society*, vol. 55, no. 7, pp. 749–759, Jul. 2004, publisher: Taylor & Francis _eprint: https://doi.org/10.1057/palgrave.jors.2601717. [Online]. Available: https://doi.org/10.1057/palgrave.jors.2601717

[5] B. S. Everitt and A. Skrondal, *The Cambridge Dictionary of Statistics*. Cambridge University Press, Aug. 2010, google-Books-ID: C98wSQAACAAJ.

[6] J. Starkweather and A. K. Moske, “Multinomial Logistic Regression,” 2011. [Online]. Available: https://it.unt.edu/sites/default/files/mlr_jds_aug2011.pdf

[7] 3Blue1Brown, “But what is a neural network? | Chapter 1, Deep learning,” Oct. 2017. [Online]. Available: https://www.youtube.com/watch?v=aircAruvnKk

[8] A. Ghatak, *Deep Learning with R*. Springer, Apr. 2019, google-Books-ID: QlmSDwAAQBAJ.

[9] S. Rushe, “Cricsheet.” [Online]. Available: https://cricsheet.org/

[10] P. Bhardwaj, “IPL Complete Dataset (2008-2020).” [Online]. Available: https://kaggle.com/patrickb1912/ipl-complete-dataset-20082020

[11] "Live cricket scores, match schedules, latest cricket news, cricket videos." [Online]. Available: https://www.espncricinfo.com

[12] "Categorical crossentropy loss function." [Online]. Available: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy

[13] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on Machine Learning*. PMLR, Jun. 2015, pp. 448–456, iSSN: 1938-7228. [Online]. Available: https://proceedings.mlr.press/v37/ioffe15.html

[14] M. Norheim, "Closing line: The most important metric in sports trading," May 2017. [Online]. Available: https://tradematesports.medium.com/closing-line-the-most-important-metric-in-sports-trading-58e56cdb4458

[15] "IPL Results & Historical Odds, Cricket India Archive." [Online]. Available: https://www.oddsportal.com/cricket/india/ipl/results/

[16] Global News, "Cricket rules explained in 2 minutes," Jul. 2019. [Online]. Available: https://www.youtube.com/watch?v=wHEIT32ZEVs