Applicability of Recurrent Neural Networks to Player Data Analysis in Freemium Video Games

Jonathan Tan  
*The University of Western Ontario*

Supervisor  
Katchabaw, Michael J.  
*The University of Western Ontario*

Graduate Program in Computer Science  
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science  
© Jonathan Tan 2019

Follow this and additional works at: [https://ir.lib.uwo.ca/etd](https://ir.lib.uwo.ca/etd)

Part of the [Artificial Intelligence and Robotics Commons](https://ir.lib.uwo.ca/etd/6180)

**Recommended Citation**
[https://ir.lib.uwo.ca/etd/6180](https://ir.lib.uwo.ca/etd/6180)

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlsadmin@uwo.ca.
Abstract

We demonstrate the applicability and practicality of recurrent neural networks (RNNs), a machine learning methodology suited for sequential data, on player data from the mobile video game *My Singing Monsters*. Since this data comes in as a stream of events, RNNs are a natural solution for analyzing this data with minimal preprocessing. We apply RNNs to monitor and forecast game metrics, predict player conversion, estimate lifetime player value, and cluster player behaviours. In each case, we discuss why the results are interesting, how the trained models can be applied in a business setting, and how the preliminary work can serve as a foundation for future research. Finally, as data on video game players is typically proprietary and confidential and results of research often go unpublished, this thesis serves to contribute to the literature on game user research.

**Keywords:** Game user research, recurrent neural networks, machine learning, business intelligence, *My Singing Monsters*
Acknowledgments

I would like to thank my supervisor Dr. Mike Katchabaw for his support and guidance. Thank you for encouraging me to pursue a thesis, for his assistance in helping me develop my research topic, and for the opportunity to work with Big Blue Bubble to complete my work. You have my most profound gratitude.

I would also like to thank Big Blue Bubble for having me for eight months and welcoming me as part of their family. I learned a great deal beyond my thesis topic during my time there. Thanks to Claudette and Bev for handling all the paperwork so that I could focus on my thesis. Thank you to the entire Business Intelligence team for helping me learn about what goes into collecting, analyzing, and using player data and their assistance in helping me organize, process, and understand the data that I used. Special thanks to Chris and Joel for some interesting conversations, related to work or otherwise.

Finally, I would like to thank my family and friends for their support throughout my life. I would not be where I am today without them. Thank you.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
<td>ii</td>
</tr>
<tr>
<td><strong>Acknowledgments</strong></td>
<td>ii</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>vi</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>viii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Big Blue Bubble</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 Description of <em>My Singing Monsters</em></td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Monetization in <em>My Singing Monsters</em></td>
<td>4</td>
</tr>
<tr>
<td>1.2 Related Work</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Neural Networks and Deep Learning</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Game User Research</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Contribution</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Roadmap</td>
<td>9</td>
</tr>
<tr>
<td><strong>2 Relevant Theory</strong></td>
<td>10</td>
</tr>
<tr>
<td>2.1 Overview of Machine Learning</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Linear Regression</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Logistic Regression</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Neural Networks</td>
<td>13</td>
</tr>
<tr>
<td>2.4.1 Gradient Descent</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2 Validation and Testing</td>
<td>16</td>
</tr>
<tr>
<td>2.5 Recurrent Neural Networks</td>
<td>16</td>
</tr>
<tr>
<td>2.5.1 Variants of RNNs</td>
<td>18</td>
</tr>
<tr>
<td>2.5.2 Bidirectional RNNs</td>
<td>20</td>
</tr>
<tr>
<td>2.5.3 Attention</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Clustering</td>
<td>21</td>
</tr>
<tr>
<td>2.6.1 k-means</td>
<td>22</td>
</tr>
<tr>
<td>2.6.2 DBSCAN</td>
<td>23</td>
</tr>
<tr>
<td>2.6.2.1 HDBSCAN*</td>
<td>25</td>
</tr>
<tr>
<td>**3 Forecasting Hourly Metrics in <em>My Singing Monsters</em></td>
<td>27</td>
</tr>
<tr>
<td>3.1 Motivation</td>
<td>27</td>
</tr>
</tbody>
</table>
4 Predicting Player Conversion in My Singing Monsters 45
4.1 Motivation .................................................. 45
4.2 Data Preparation ........................................... 46
  4.2.1 Session and Player Feature Vectors .................. 47
  4.2.2 Session Start Encoding .............................. 47
  4.2.3 Player and Session Sampling ......................... 48
  4.2.4 Cost-Based Player Weighing ......................... 50
  4.2.5 Feature Transformation ............................... 51
  4.2.6 Zero-Padding ........................................ 51
4.3 Model Architecture ......................................... 51
4.4 Model Training ............................................ 53
4.5 Model Evaluation ........................................... 54
  4.5.1 Precision and Recall ................................ 54
  4.5.2 Class Probabilities Over Time ....................... 57
4.6 Clustering .................................................. 59
  4.6.1 Applying HDBSCAN* ................................ 59
  4.6.2 Cluster Hierarchy .................................... 60
  4.6.3 Cluster Analysis ...................................... 60
  4.6.4 Soft Clustering ....................................... 63
  4.6.5 Applying $k$-means .................................. 63
4.7 Discussion .................................................. 64

5 Predicting Lifetime Customer Value in My Singing Monsters 66
5.1 Motivation .................................................. 66
5.2 Data Preparation ........................................... 66
5.3 Training Multiple Models ................................... 67
  5.3.1 Evaluation on Different Value Cohorts ............. 70
5.4 Adding a Whale Classifier to the Model .................. 71
5.5 Training the Production Model ............................ 74
5.6 Comparison to Current Production Model ................ 74
  5.6.1 Evaluation on World Data Set ....................... 75
  5.6.2 Evaluation on US Data Set ........................... 75
5.7 Evaluation on Individual Players ......................... 77
5.8 Publication as a Shiny App ................................ 79
5.9 Discussion .................................................. 79
## List of Figures

2.1 A graphical representation of a neural network. .................................... 15
2.2 An unrolled RNN. ................................................................. 18
2.3 $k$-means on various data sets and using various values of $k$. ................. 23
2.4 DBSCAN on various data sets and using various values of $\varepsilon$. ............ 24
2.5 HDBSCAN* on various data sets. ............................................ 26

3.1 A snapshot of the hourly metrics of interest. ........................................ 28
3.2 Missing metrics (highlighted) before and after. ................................... 30
3.3 Diagram of our hourly metrics model. ........................................... 33
3.4 Training and validation losses for the hourly metrics model. .................. 34
3.5 Forecasted hourly metrics from training set. ...................................... 38
3.6 Forecasted hourly metrics from test set. ......................................... 39
3.7 Attention maps for examples in Figure 3.5. ...................................... 40
3.8 Attention maps for examples in Figure 3.6. ...................................... 41
3.9 First example of forecasts on new data. .......................................... 42
3.10 Second example of forecasts on new data. ....................................... 43
3.11 Snapshot of the Shiny application for the hourly metrics model. ............. 44

4.1 Positional encoding of day of the week. .......................................... 49
4.2 Diagram of our player conversion classifier. ...................................... 53
4.3 Training and validation losses for the conversion prediction model. ........ 54
4.4 Confusion matrix for the combined training and validation set. ............... 55
4.5 Confusion matrix for the test set. ............................................... 56
4.6 Various predictions and their respective attention maps. ....................... 58
4.7 Visualizations of the clustering produced by HDBSCAN*. ..................... 60
4.8 HDBSCAN* cluster hierarchy. ................................................... 61
4.9 Visualizations of the soft clustering produced by HDBSCAN*. ............... 63
4.10 Visualizations of the clustering produced by $k$-means for $k = 4$. ........ 64

5.1 Diagram of our D120 LTV prediction model. .................................... 68
5.2 Training and validation losses for the LTV prediction models. ............... 69
5.3 Confusion matrix for the whale classifier trained on training set. .............. 72
5.4 Confusion matrix for the whale classifier trained on a larger data set. ....... 74
5.5 Input features, predictions, and attention weights for example players. .... 77
5.6 Snapshot of the Shiny application for the D120 LTV prediction model. ....... 79

6.1 Training loss for the player autoencoder. ......................................... 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2 Archetypal analysis clustering of the 30-day player encodings.</td>
<td>84</td>
</tr>
<tr>
<td>6.3 Relative sizes of the archetypal analysis clusters.</td>
<td>84</td>
</tr>
<tr>
<td>6.4 30-day player encodings overlaid with their D120 LTVs.</td>
<td>85</td>
</tr>
<tr>
<td>6.5 30-day player encodings overlaid with their acquisition source and their platform.</td>
<td>86</td>
</tr>
<tr>
<td>6.6 30-day player encodings overlaid with the player’s retention status.</td>
<td>86</td>
</tr>
<tr>
<td>6.7 Training loss for the retained player autoencoder.</td>
<td>87</td>
</tr>
<tr>
<td>6.8 Archetypal analysis clustering of the 60-day retained player encodings with the relative sizes.</td>
<td>88</td>
</tr>
<tr>
<td>6.9 60-day retained player encodings overlaid with their D120 LTVs.</td>
<td>88</td>
</tr>
<tr>
<td>6.10 Daily minimum and maximum levels for each of the six cluster archetypes.</td>
<td>89</td>
</tr>
<tr>
<td>A.1 Example of the main view in My Singing Monsters.</td>
<td>103</td>
</tr>
<tr>
<td>A.2 Island select screen.</td>
<td>104</td>
</tr>
<tr>
<td>A.3 Market screen.</td>
<td>104</td>
</tr>
<tr>
<td>A.4 Breeding screen.</td>
<td>105</td>
</tr>
<tr>
<td>A.5 Speedup screen.</td>
<td>105</td>
</tr>
<tr>
<td>A.6 Tribal Island.</td>
<td>106</td>
</tr>
<tr>
<td>A.7 Composer Island.</td>
<td>106</td>
</tr>
<tr>
<td>B.1 Cluster 1 from HDBSCAN*.</td>
<td>108</td>
</tr>
<tr>
<td>B.2 Cluster 2 from HDBSCAN*.</td>
<td>109</td>
</tr>
<tr>
<td>B.3 Cluster 3 from HDBSCAN*.</td>
<td>110</td>
</tr>
<tr>
<td>B.4 Cluster 4 from HDBSCAN*.</td>
<td>111</td>
</tr>
<tr>
<td>B.5 Cluster 1 from k-means.</td>
<td>112</td>
</tr>
<tr>
<td>B.6 Cluster 2 from k-means.</td>
<td>113</td>
</tr>
<tr>
<td>B.7 Cluster 3 from k-means.</td>
<td>114</td>
</tr>
<tr>
<td>B.8 Cluster 4 from k-means.</td>
<td>115</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Optimized hyperparameters for hourly metrics model.</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Top ten model architectures based on validation loss for forecasting hourly metrics.</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Optimized hyperparameters for player conversion model.</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Top ten model architectures based on validation loss for predicting player conversion.</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>Precision, recall, and $F_1$ scores for the combined training and validation set.</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>Precision, recall, and $F_1$ scores for the test set.</td>
<td>56</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation of LTV model on the test set.</td>
<td>70</td>
</tr>
<tr>
<td>5.2</td>
<td>Evaluation of LTV model, but with a manual check to set predictions to at least the D14 LTV.</td>
<td>70</td>
</tr>
<tr>
<td>5.3</td>
<td>Evaluation of LTV model on whales in test set.</td>
<td>71</td>
</tr>
<tr>
<td>5.4</td>
<td>Evaluation of LTV model on non-whales in test set.</td>
<td>71</td>
</tr>
<tr>
<td>5.5</td>
<td>Evaluation of whale and non-whale models with classes known.</td>
<td>73</td>
</tr>
<tr>
<td>5.6</td>
<td>Evaluation of whale and non-whale models using the trained classifier.</td>
<td>73</td>
</tr>
<tr>
<td>5.7</td>
<td>Evaluation of LTV model trained on more data on the test set.</td>
<td>75</td>
</tr>
<tr>
<td>5.8</td>
<td>Evaluation of the LTV model on the World data set.</td>
<td>76</td>
</tr>
<tr>
<td>5.9</td>
<td>Evaluation of the LTV model on the US data set.</td>
<td>76</td>
</tr>
<tr>
<td>6.1</td>
<td>Analysis of six different retained player clusters based of their archetypes.</td>
<td>90</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The video game industry is large and ever-growing. The NPD Group report that year-to-date spending on console, portable, and PC games reached $4.8 billion in October 2018, the highest since 2011 and is up 16% since the previous year [1]. SuperData report that “the audience for gaming video content grew 10% to reach 850 [million] unique viewers in 2018” [2], thanks to streaming platforms like Twitch and video-hosting services like YouTube. They also report that interest in new technologies like virtual reality (VR) and augmented reality (AR) has increased, with the revenue from these immersive technologies increasing by $2.1 billion between 2017 and 2018. Platforms such as Steam make it possible for anyone with a unique and marketable idea for a game to publish titles, as well as providing a centralized distribution service for large publishers to put their games. The cost of computers has also become more affordable for the average person, meaning that almost anyone can access these games. Different games will appeal to different people; dedicated, hardcore players might be attracted to competitive multiplayer games, while more casual players might prefer a single-player game with a lighter or comedic tone [3]. Another important feature to these players is the cost of a game. Hardcore players might be willing to pay $80 CDN to buy a game from a publisher they trust and for a game they are almost certain they will like. On the other hand, casual players might not want to pay nearly that much for a game about which they do not know much and which they may only play for a few hours.

Even hardcore players might find the high cost of a game a difficult barrier of entry to cross [4]. A paradigm that has become more popular in recent times is the “freemium” model for games [5]. In this model, the game is free for anyone to play. Instead, publishers will make money from the game by selling various in-game features, such as currency to buy in-game items or temporary power-ups to alter the game experience. The cost and effect of these features vary among games, but are generally inexpensive and can be purchased more than once. However, because the game is free to play, the barrier-to-entry is much less than a paid game and can attract many more players. In turn, of the large number of players that play the game, a small percentage will spend money on those small, in-game features. These players might repeatedly spend on the game and, despite representing only a small percentage of the overall player base, will still generate more revenue than players of a paid game who only spend money upfront for the game.

Freemium games exist on many different platforms but are most prominent in the mobile game space [5]. The mobile game space is also an attractive option for casual players who
might only play games during work breaks, after dinner, and the like. Thus, freemium mobile games are generators of a large amount of data on these players. The large amount of players will likely also mean a large variety in how the game is played, even if the game itself is fairly straightforward. With a large amount of data and only a small fraction of it pertaining to players who actually spend money (and thus keeping the game company in business), it is then the job of data analysts to sift through the data and learn how to make the game more profitable. In fact, because free-to-play games are usually constantly being updated, there are more opportunities to make changes because of things analysts learn from the data than there is for paid games that are not updated post-release.

The games industry is not the only one that generates a lot of data. The fields of finance and medicine also produce a lot of data, as do many Internet of Things (IoT) applications. In order to process these large data sets, data mining and machine learning techniques have been developed. Consequently, as the resultant model can sift through more data than a person, it may be able to provide a deeper analysis than traditional analysis methods. Recognizing this potential, some steps have been made towards applying data mining and machine learning algorithms to game analytics \[6, 7\] in recent years.

However, because of the confidential nature of these data sets, what is typically published in game user research literature can be quite limited. Usually, only select researchers can publish on a particular game and the methods presented may be specific to that game or type of analysis. Additionally, specific model architectures (i.e. design parameters of the model) may be kept private. In this paper, we collaborate with local mobile game publisher Big Blue Bubble and apply machine learning techniques to data about players from their mobile video game *My Singing Monsters*. In particular, we demonstrate how a type of machine learning technique, recurrent neural networks, can be applied to a variety of analyses dealing with player data.

### 1.1 Big Blue Bubble

Big Blue Bubble (BBB) is an award-winning mobile video game studio located in London, Ontario, Canada. It was founded in 2004 and has developed and released multiple mobile and non-mobile titles. Additionally, it has worked with several third-party developers to publish several games. Notable titles include *Burn the Rope* which “would see over a million downloads on the iPhone alone, and would go on to become the #1 game in 58 countries” \[8\] and *My Singing Monsters* which has had over 67 million downloads on iOS and Android combined and, at the time of writing, enjoys over 300000 daily active players.

Big Blue Bubble is best-known for the *My Singing Monsters* mobile game franchise. The first game was released in 2012 for iOS and in 2013 for Android and is still the most popular Big Blue Bubble game. The game also has a sequel, *My Singing Monsters: Dawn of Fire*, released in 2015. The franchise also has a few other games, notably *My Singing Monsters: Composer* and *Jammer Splash*, but we will focus primarily on the original *My Singing Monsters* (MSM) game. Big Blue Bubble also sells related apparel, plushies, phone cases, and accessories on their Amazon store. In 2017, the company announced that an animated series, *Those Singing Monsters*, was in development in collaboration with Wind Sun Sky Entertainment and Skybound \[9\].
1.1.1 Description of My Singing Monsters

The My Singing Monsters game is a world-building game where the player breeds and feeds various monsters. For screenshots of the game, please refer to Appendix A. The monsters live on an island and the player can move the monster’s position on the island. The players starts with one island unlocked (Plant Island) and various other islands are unlocked as the player progresses through the game. Depending on the island, only a subset of the monsters in the game can be placed and bred. Each monster is a combination of various elements and the player breeds two monsters in an attempt to get a new monster with the union of its parent’s elements. For example, on Plant Island, a Mammott, a monster with the cold element, can be bred together with a Potbelly, which has the plant element, to produce a Furcorn which has both the cold and plant element. At the time of writing, monsters can have up to four unique elements and there are five total unique (natural) elements in the game. Breeding two monsters can take anywhere from a few seconds to over a day and produces a monster egg. The egg must then be incubated for a similar amount of time before the monster hatches and can be placed on an island. Except for certain cases, monsters cannot be moved between islands.

The game is unique and gets its name from the fact that the monsters sing a unique song when placed on an island. Each monster has a different song for each different island. Furthermore, when the player places multiple unique monsters on their island, the different songs combine together naturally to form a larger song. Thus, one implicit objective of the game is for the player to breed all the possible monsters available for an island in order to hear the complete song. There are currently 12 unique songs available for the player to discover, with more currently in development.

A secondary objective of the game involves coins, the primary currency of the game. Over time, the monsters the players place on an island will earn coins which the player can then collect. The player uses coins to buy single-element monsters, decorations for their islands, useful structures, or treats for the monsters. In particular, the Bakery structure can bake treats at the cost of time and coins. The treats are then used to level up the player’s monsters to increase the rate at which the monster earns coins.

Diamonds are the premium currency of the game, though they can be earned in-game via daily events, the mine structure, or by certain monsters. Diamonds can be used to buy certain monsters or to instantly complete a breeding, incubation, or baking process (called a speedup event). The game also has a few specific currencies. Keys are used to purchase a class of monsters called Dipsters, while relics are used to purchase Werdos. Shards are primarily produced on and are used to interact with Ethereal Island. Finally, starpower can be used to purchase monsters and decorations for the in-game StarShop. All currencies except relics and starpower can be directly purchased with real money via in-app purchases, though diamonds can be traded into relics. Alternatively, there is some mechanism or another in the game to earn these currencies without in-app purchases.

Two unique islands in particular are Tribal Island and Composer Island. On both these islands, the players does not breed monsters. To access Tribal Island, the player must either create or join a tribe. Then, the player can move one of their monsters (it is more of a copy as the monster stays on its original island) to the tribe’s island. The island has a level calculated from the sum of the levels of the monster on that island. The player can only level up their

\[\text{For example, the (current) song on Plant Island can be heard here: https://youtu.be/bP0D53mFxFA}\]
monster on the island, but can do so using not only treats but shards and coins as well. On Composer Island, the player must purchase the monsters using coins. The player can then modify the song that the monster sings using the in-game interface, allowing the player to create custom, complex melodies from the various timbres provided by the monsters. Other players can visit other players’ Tribal and Composer Islands, thus these islands represent the main social elements of the game.

1.1.2 Monetization in My Singing Monsters

Apart from merchandise sales and in-app purchases of gold, diamonds, and other in-game currencies, Big Blue Bubble monetizes the game using in-game advertisements. As the player plays the game, short advertisements, typically 30 seconds long, will interrupt their play. The player must watch through the entire ad before they can continue playing the game. As with many free-to-play games, this model earns the company revenue based on the number of ads viewed by the player. A viewing of an ad is called an impression. Once a player makes an in-game purchase (using real-world currency), these ads are no longer shown. However, any player has the option to speed up the time it takes to breed, incubate, or bake by 15 minutes by watching an ad. Thus, it can still generate impressions from players who have already made a purchase. Arguably, this is a win-win for both the player and the company as the player opts in to viewing these ads. The ads which interrupt play are riskier because there is a chance that they frustrate the player and cause them to quit the game. Optimizing around these ads is thus an important and ongoing endeavour.

Another method of monetizing the game is through offers. The player can access an offer wall in the game for coins, treats, and diamonds. An offer wall is basically a list of offers that the player can complete. Offers can include reaching a certain point in another game, creating an account for a certain website, or completing some other activity outside of the game. In return, the player earns a proportional amount of that currency for their time. As offers are completed, the company earns revenue from the appropriate ad network. Since offers are opt-in, the player should not feel too frustrated with them. Of course, there is the risk that the player is directed to a malicious site or game, or that the player leaves our game for another game because of an offer.

Since Big Blue Bubble is a business and money is what keeps the game and the company alive, it is of high importance to understand where the money comes from. To that end, it is equivalently important to understand the players. Without an understanding of player behaviour in the game, any action we take runs the risk of alienating or angering our player base. It is the job of the Business Intelligence (BI) team at Big Blue Bubble to perform such analyses on the players and to monitor and report on the balance between the players and the monetization of the game.

1.2 Related Work

The field of data analytics and business intelligence is not new [10] and there has been a large body of published work on this topic. However, this is not true when talking specifically about data analytics on games. One of the reasons is that the field is relatively new and only recently
have game companies begun to put systems in place to do any sort of analysis. This requires the large-scale collection of game metrics which consists of “quantitative measures of attributes of objects”. As a result, analysts may not be capable of making use of the data collected simply due to the sheer quantity of data at their disposal.

1.2.1 Neural Networks and Deep Learning

Historically, there have been many statistical techniques to analyze data. Techniques can range from simply taking a mean from a sample, to fitting a line of best fit, to performing statistical tests such as the t-test or $\chi^2$-test, to more complex statistical models. However, a trend that has risen in popularity in recent years is the topic of machine learning. Machine learning involves having a computer system learn a model by feeding it a large amount of data. Hopefully, the model is able to learn a model of the data which can be applied to new, unseen data. Such techniques might not have worked in the past because of a lack of data, but with the magnitude of data that is collected nowadays, machine learning is quickly becoming a strong force in the field of data analysis.

The study of machine learning is vast, but one area in particular that has received notable attention by both industry and academia is neural networks. We discuss the topic of neural networks in greater detail in Section 2.4 but large neural networks are composed of a multitude of layers. They can be described as being deep and so the term deep learning has quickly become a synonym for training large neural networks.

The origins of deep learning can be traced as far back as 1958 with the development of the Perceptron by Rosenblatt. Of course, back then, models were not very “deep”. However, as research continued, significant advances to the field were made. Lecun et al. showed that convolutional neural networks, a special kind of neural network, could reliably identify handwritten digits. This was further expanded by Krizhevsky et al. in 2012 with the development of ImageNet which uses 8 layers to identify images as belonging from one of 1000 classes. The area of computer vision has become one of the largest topics in deep learning, with neural networks being used in facial recognition systems, medical diagnoses, image manipulation, and more.

The topic of this thesis, however, deals more with sequential data, not image data. Sequential data involves a data set where each data point is a list of values and the ordering of these values is important. A field that has effectively used deep learning in conjunction with sequential data is natural language processing (NLP). Neural networks have been used to create word representations (encode words as vectors), to perform machine translation, to generate captions for images, and much more. The main tools used here are recurrent neural networks (RNNs) which are discussed in Section 2.5. Of course, RNNs are not limited to being applied on NLP tasks and are also used when sequential data is available. For our purposes, the sequential data available to us will include the sequence of game events a player logs as they play My Singing Monsters.

Various industries have recognized the value to be gained by using deep learning. To name a few examples, Castanedo et al. and Wangperawong et al. use deep learning to predict customer churn in the telecommunication industry. Churn is loosely defined as the customer leaving or quitting the service. Wu et al., Quadrana et al., and Li et al. use deep learning in recommendation systems on e-commerce sites. Du et al. use a RNN
to identify anomalies in system logs.

However, one large concern with machine learning, and neural networks in particular, is that they are “black boxes” in that it is hard to easily understand how the networks arrive to their outputs. For that reason, decision trees are a popular machine learning technique in that they allow for easy interpretation. However, if a very wide or deep decision tree is necessary to attain a certain level of performance, the model would still not be easily interpretable. Thus, we see that even simple-to-understand machine learning algorithms can be difficult to interpret. Neural networks are particularly difficult to interpret since they are parameterized by thousands, if not millions, of parameters. Additionally, they have a multitude of hyperparameters, parameters of the how the model is built and trained, that appear to also have an effect on the model’s performance. For this reason, it is still not clear how to interpret how exactly these models work.

However, we can try to extract information from this model that tells us how they arrive to their predictions. Lipton [40] refers to this as post-hoc interpretability and can be achieved through techniques such as asking the model to output a text explanation or visualizing certain parameters to “see” where the model is “looking”. The latter is particularly useful in scenarios where the input is an image, though it can also be used when the input is a sequence. We will apply this visualization technique with some of our models to get an idea of where the models appear to focus. However, these visualizations, as Lipton notes, can be misleading and do not necessarily prove that the model will work as we might expect on unseen data. Nevertheless, it is still a useful technique to pull more information out of a neural network model than just the prediction and try to understand where weakness in the model may be found.

1.2.2 Game User Research

Analysis requires data and there is much data that can collected with games. For example, one can define events in a game, such as a button press, associate it with a timestamp, and store that data. We can also define more complex events, such as a particular action being completed, and store those as well. Different actions will have different parameters and will return different results and part of the trouble with collecting game data is the sheer variety of data types that has to be organized. One example of a proprietary system that attempts to organize and use data in a useful manner is Microsoft’s “Tracking Real-Time User Experience” (TRUE) [11].

While monitoring the financial stability of a game is important, what is arguably more important is understanding the behaviour of its player base. This is self-evident as the money the game makes comes largely from its players. If players are leaving the game, or are disliking the game, the game will not be financially successful as these players will not spend their money on the game. Particularly for free-to-play (freemium) models, where the user can play all or most of the game for free and is given the option to pay for in-game perks, companies would then operate at a loss. What is worse is that these players can then leave negative reviews and, through word of mouth, other players will simply not even try the game, or even other games that the company releases. As such, the goal of game user research is to investigate and understand how players interact with the game and how we can improve the game so that the player experience is a better one.

Game data is proprietary and, as a result, it is extremely difficult to find published studies of game user research that provide the data that was used. However, there has been recent
acknowledgment in trying to get more papers published which describe how analyses were done so that the community as a whole can learn from past successes and failures [41]. One concrete example is presented in Drachen, Canossa, et al. [42] where they take game metrics (such as the total playing time and number of ways the player dies) for the game Tomb Raider: Underworld and use self-organizing maps (SOMs) to cluster, or group, players into one of four emergent categories. Mahlmann et al. [43] extend the analysis by taking the same metrics for the early levels of the game and apply various techniques to predict which is the last level the player will complete and how long it takes for them to complete the game. Player clustering for Tomb Raider: Underworld is further explored by Sifa, Drachen, et al. [44] who cluster using simplex volume maximization (SiVM) and then look at clusters independently for each level and see how they effectively “evolve” in the game. Similarly, in Drachen, Sifa, et al. [45], clustering on game metrics is performed on several games and the authors discuss the importance of interpretability in the generated clusters.

Weber et al. [46] look at player retention in Madden NFL 11 and predict how many games are played based on a feature vector derived from taking various metrics on how the player plays. From this, the authors then see which features, and thus which behaviours, most strongly correlate with the number of games played. Bauckhage et al. [47] look at the distribution of total play time across five different games and find that the Weibull distribution is a good fit. They conclude that “an average player’s interest [...] evolves according to a non-homogeneous Poisson process whose intensity function is given by a power law.” Sifa, Bauckhage, et al. [48] extend this work on a larger scale, fitting Weibulls to over 3000 PC games. They also cluster on the fitted distributions and produce four prototypical profiles of total playtime. The largest of these clusters, containing mostly action and indie (independently developed) games, has most of the players quitting after 2-3 hours into the game.

Player churn, which is the act of a player quitting a game, is a highly-studied topic in game user research. Of course, there are always factors that can contribute to a player leaving a game which are simply out of the developer’s control, but any insights to if there is a particular problem in how the game is being perceived or played can help minimize player churn. Particularly in games following a freemium model, retaining any players who are likely to make in-game purchases is crucial to keep the game profitable. Runge et al. [49] use neural networks and hidden Markov models (HMM) to predict churn in two casual social games. Additionally, they also show the positive effects after performing A/B tests based on the results of their churn model. Rothenbuehler et al. [50] also use HMMs to predict churn using the daily app usage numbers to model player motivation. They find that, when using a HMM with 10 states, users in the lower activity states are more likely to churn. They propose targeting such users with some sort of incentive to try to get these players to a higher activity state. Kim et al. [51] use a variety of models including random forests and recurrent neural networks to predict churn. Here, they define churn differently in that they are using an observational period of set length to predict if a player will play in the contiguous churn prediction period. For the three traditional algorithms they use, they perform feature engineering to develop features from the raw play log data. However, for the convolutional and recurrent neural networks, they try to preprocess the play log data as minimally as possible. They find that all algorithms perform more or less the same. Perianez et al. [52] use conditional inference survival ensembles [53] to predict when the player will churn rather than if the player will churn.
1.3 Contribution

A lot of the work that has been published that deals with using machine learning for game user research consists of tackling a particular problem (e.g. churn prediction, player clustering) on a particular game. As the data sets used in these studies are confidential, it is likely that only researchers working closely with the game company are able to complete studies pertaining to a particular game and, of those studies, only a small number are actually permitted to be published. Games like Tomb Raider: Underworld which have a notable number of papers published on it are few and far between. This thesis aims to contribute to the field by conducting four different analyses on a single game, the mobile game My Singing Monsters by Big Blue Bubble, and adding these analyses to the body of work related to game user research.

Additionally, we also aim to unify this study by applying a single methodology to all four analyses. Since player game data in naturally event-driven and sequential, recurrent neural networks will be the choice of algorithm. RNNs, to the best of our knowledge, have not been widely applied in the field of game user research. RNNs also afford us the ability to work with raw event data. Commonly, when performing analyses on player data, game metrics are engineered and calculated from the raw numbers. For example, rather than looking at each login as an event, the analyst might use an averaged number of logins per day. One issue with this is that the metrics we decide to use may not represent the actual player data very well. Since event data is temporal, we may also lose this temporal information if we do not engineer the metrics well. RNNs allow us to use the raw, sequential event data as input and will hopefully pick up on any patterns in the data.

Another affordance with RNNs is that they form an internal, latent representation of the input sequence data. That is, we can design our model such that it is forced to squeeze the input into a limited set of values. This might be considered a form of automatic feature engineering. Of course, we might not be able to interpret what these features are, but we can use them as representations of the input data. We apply this to cluster on player data in Chapter 4 and Chapter 6 and allows us to visualize our player base. This is an extremely useful technique to help understand and identify patterns and correlations.

In summary, we aim to demonstrate the usefulness and effectiveness of RNNs when it comes to game user analysis. To do this, we apply RNNs to four different analyses of player data. We show that many different types of analysis commonly done in game user research—forecasting, churn prediction, player value prediction, and player classification and clustering—can be done with the help of RNNs. In addition, we do not need to spend much deciding of how to combine the data in the preprocessing stage and can instead work with the raw data with minimal preprocessing.

It should be noted that the goal of this thesis is not to show that RNNs are the best-performing model in each of the four use cases. Since benchmarks are generally not available in this field of research, it would be difficult to compare various techniques. However, chasing benchmarks may be a futile goal in and of itself as the top-performing model may simply be a well-tuned model on that particular benchmark data set. Rather, our goal is to show that RNNs are a practical option for analysts to consider when performing game user research of almost any type and rather painless to get going. We discuss how the results presented here can be usable in a business setting in the Discussion sections at the end of Chapters 3-6.
1.4 Roadmap

In this thesis, we explore four applications of RNNs. To this end, we build up the necessary theory in Chapter 2, starting with simple linear regression and working up to neural networks and, finally, recurrent neural networks. In Section 2.6, we also discuss various clustering algorithms.

Our first application of RNNs is introduced in Chapter 3 where we forecast hourly features. The model we build here takes as input historical hourly data for six different time series features at any given date and predicts the values of these features one day ahead. The model also outputs an interval around these predictions in which most of the actual, observed values should lie and which can be used to identify anomalies. Thus, the model can be used to monitor these hourly features as well as to detect any significant deviations that may be a cause for concern.

In Chapter 4, we predict player conversion. When a player makes an in-game purchase using real-world currency, they are said to be “converted”. Converted players make up a large fraction of the revenue for the game so it would be useful to identify such players, particularly early on. For this, we collect unique event counts, as well as some other player information, for the first ten game sessions to make a classification as to whether the player will convert or not. Since we are using recurrent neural networks, we can also use less than the first ten days and observe how the model’s predictions change over time. Finally, because neural networks must form an internal representation of the input data, we take that internal representation to cluster the players. We apply two clustering algorithms and observe that we get clusters corresponding to the target classes.

Next, in Chapter 5, we predict lifetime player value. Here, we use a minimal set of early daily features to predict lifetime value, that is, the amount of revenue the player will earn the company, 120 days in the future. We try different strategies for training models. First, we train three different models trained on different lengths of data, but predicting on a fixed length. We note that high-value players are the most difficult to predict. To this end, we train a classifier to separate these players apart and then train two different classifiers for both groups. We compare this model with the current model in operation at Big Blue Bubble and find comparable or even competitive results.

For our last analysis in Chapter 6, we attempt to build an encoding of players using an autoencoder. These encodings are vector representations of players that contain important information to describe that player generally. This can be thought of as automatic feature engineering. We take these encodings and cluster on them and look for correlation in lifetime value, acquisition source, platform, and retention status. We then further investigate retained players and look at the archetypal behaviours in each of the clusters we find.

Finally, we summarize the analyses and discuss them holistically in Chapter 7. We also describe how this work can be expanded in the future and state our closing remarks.
Chapter 2

Relevant Theory

2.1 Overview of Machine Learning

In this chapter, we will develop the necessary theory for understanding the work with neural networks that will be explored in the next chapter. To begin, we discuss the general motivation behind machine learning (ML). Simply put, the goal of machine learning is to have a machine, or a system of machines, accept an input and to produce a desired output. In the case of *supervised machine learning*, the expected output is known beforehand. The system is given multiple input/output pairs and iteratively “learns” to produce the correct output. In the case of *unsupervised machine learning*, the expected output is not known beforehand and the correctness of the output must be evaluated manually.

In general, a machine learning model is comprised of three main components: the input, the machine learning algorithm, and the output. The input is typically represented as a tensor \( x \) and the output a tensor \( y \). In order to describe a single example on which the model should learn, it is useful to think of the individual components of \( x \) as features of that particular example. For example, if the input of the model is a single player of *My Singing Monsters*, \( x \) might be a \( d \)-dimensional vector where each component, or feature, is a game metric for that player like their current level, the average number of unique events per session, or the number of in-app purchases they have made in total. We will refer to such an input as a *feature vector*. In another case, \( x \) might be an image and might be best represented as a tensor in \( \mathbb{R}^{m \times n \times c} \) where \( m \) and \( n \) would be the height and the width of the image and \( c \) would be the number of colour channels. For images using a RGB colour space, each component of \( x \) would be the intensity of a particular pixel in a particular channel.

The output of a machine learning model, \( \hat{y} \), is also a tensor. In some cases, it may simply be a scalar, such as in the case of linear regression where we fit a line to the input data points and our output for a new data point is its projection onto that line, or in the case of binary classification where the model can only output either 0 or 1, each corresponding to a truth value associated with a particular input. In other cases, the model can be expected to output a feature vector, an image tensor, or something else. In supervised machine learning, as the model iterates through multiple example input/output pairs, it learns to improve the correctness of its predicted output \( \hat{y} \) to the actual (or true) output \( y \) by using a *loss function* \( L(y, \hat{y}) \). The function is also called the *objective function*, or *cost function* of the model. In the unsupervised
case, \( y \) is not known and therefore the model cannot effectively self-evaluate and must instead must use a criterion based only on \( x \).

The ML algorithm is then tasked with optimizing the loss function \( \mathcal{L} \). The optimization of \( \mathcal{L} \) is typically its minimization via a method known as gradient descent. Effectively, with each example, the algorithm will tune itself in the direction of the descending gradient of \( \mathcal{L} \) and will hopefully arrive to some local minimum. Ideally, this would also be the global minimum of \( \mathcal{L} \), though this is rarely the case. The tuning, or learning, is done by changing the parameters \( \mathbf{w} \) of the algorithm. The difference between various ML algorithms is the number of parameters as well as how they are applied on each input example. There also exists various methods of performing the optimization of \( \mathcal{L} \). Generally speaking, a ML algorithm can be viewed as a function \( f \) which takes an input \( \mathbf{x} \) and learned parameters \( \mathbf{\tilde{w}} \) and produces an output \( \hat{y} \) that minimizes \( \mathcal{L}(y, \hat{y}) \).

\[
f(\mathbf{x}, \mathbf{\tilde{w}}) = \hat{y} \quad \text{s.t. } \mathcal{L}(y, \hat{y}) \text{ is minimal} \quad (2.1)
\]

### 2.2 Linear Regression

We now explore more concrete examples of ML algorithms. To begin, let us consider a simple case where the input \( \mathbf{x} \) is a \( d \)-dimensional feature vector and the output is a single scalar \( \hat{y} \). The algorithm to generate \( \hat{y} \) is to simply multiply each feature \( x_i \) with a learned weight \( \mathbf{\tilde{w}}_i \) and sum over all features. We can conveniently express this as an inner product between the input vector and the learned weight vector:

\[
\hat{y} = \sum_{i=1}^{d} \mathbf{\tilde{w}_i} x_i = \mathbf{\tilde{w}} \cdot \mathbf{x} = \mathbf{x}^T \mathbf{\tilde{w}} \quad (2.2)
\]

\( \mathbf{\tilde{w}} \) is tuned over the course of multiple input/output examples to minimize a loss function \( \mathcal{L} \). This process is also referred to as the training phase of the model and the examples are thus referred to as training examples. Because we can have millions of training examples, we combine the process of training on multiple examples into a single matrix operation:

\[
\hat{y} = \mathbf{X} \mathbf{\tilde{w}} \quad (2.3)
\]

where \( \hat{y} \) is a vector where \( \hat{y}^{(i)} \) denotes the \( i \)-th component of \( \hat{y} \) and represents the model’s output for the \( i \)-th training example, and \( \mathbf{X} \) is matrix where the \( i \)-th row represents the feature vector for the \( i \)-th training example. Thus, element \( x_{ij} \) represents the \( j \)-th feature of the \( i \)-th training example. For consistency, we will instead opt to denote this as \( x^{(i)} \). Depending on the method of optimization we use for optimizing the loss function \( \mathcal{L} \), \( \mathbf{X} \) (and thus \( \hat{y} \)) might contain all training examples, or a proper subset of the training examples. We will denote the number of training examples used as \( M \). Combining the training of multiple examples is efficient because matrix operations on a batch of training examples is much faster than iterating one-by-one over training examples on modern-day CPUs and, in particular, GPUs.

Finally, we note that we can geometrically interpret this as finding the line of best fit, as parameterized by \( \mathbf{\tilde{w}} \), on the \( M \) training examples. However, our line is constrained as it must pass through the origin. In order to avoid this, we can also learn an intercept (or bias) vector \( \mathbf{\tilde{b}} \) and include it in our formulation:

\[
\hat{y} = \mathbf{X} \mathbf{\tilde{w}} + \mathbf{\tilde{b}} \quad (2.4)
\]
However, equations where this bias term is written out explicitly become cumbersome and, as such, we will assume that the bias is always implicitly included for all ML algorithms discussed in this thesis.

Note that this formulation is linear regression and that we can produce a closed-form solution for \( \hat{w} \)

\[
\hat{w} = (X^T X)^{-1} X^T y
\]

that minimizes the mean squared error (MSE) loss function:

\[
L(y, \hat{y}) = \frac{1}{M} \sum_{i=1}^{M} (y^{(i)} - \hat{y}^{(i)})^2
\]

In this case, we need not use gradient descent to optimize the loss function as we have its closed-form solution. In practice, however, when \( X \) is very large, computing both \( X^T X \) and its inverse (or employing some other technique such as QR decomposition) is computationally infeasible compared to the more general method of gradient descent. In fact, most other ML algorithms do not have a closed-form solution and must use gradient descent (see Subsection 2.4.1).

### 2.3 Logistic Regression

The basic linear regression formulated we have just presented is limited in that one must fit a hyperplane (a line in the two-dimensional case) to the data points rather than something more general. In particular, it is not applicable in the case of binary classification when the output is either a 0 or 1. To tackle this problem, we wrap our linear regression equation (Equation 2.3) in a non-linear function \( \phi \):

\[
\hat{y} = \phi(X \hat{w})
\]

The function \( \phi \) is also called an activation or transfer function. As a concrete example, we will use the model of logistic regression which uses the logistic, or sigmoid, function \( \sigma \) as \( \phi \) where:

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]

The range of the logistic function is \((0, 1)\) so we can then use the output for binary classification by thresholding:

\[
\hat{y} = \begin{cases} 
1 & \text{if } \sigma(X \hat{w}) > 0.5 \\
0 & \text{else}
\end{cases}
\]

Note that we do not necessarily have to use 0.5 as our threshold. Alternatively, we can skip thresholding and simply interpret a predicted output value as the probability of that training example being labeled 1:

\[
\hat{y}^{(i)} = P(y^{(i)} = 1 \mid X, \hat{w})
\]

and, consequently,

\[
1 - \hat{y}^{(i)} = P(y^{(i)} = 0 \mid X, \hat{w})
\]
For logistic regression, a MSE loss function does not really make sense as its use with linear regression assumes that the output value follows a normal distribution. In the logistic regression setting, the output value instead follows a Bernoulli distribution. We wish to maximize the probability of predicting the correct binary value. In other words, we aim to maximize

\[
\prod_{i=1}^{M} P(\hat{y}^{(i)} = y^{(i)}) = \prod_{i=1}^{M} P(y^{(i)}) = \prod_{i=1}^{M} \hat{y}^{(i)y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}
\] (2.11a)

Equivalently,

\[
\log \prod_{i=1}^{M} P(y^{(i)}) = \log \prod_{i=1}^{M} \hat{y}^{(i)y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}
\] (2.11b)

\[
\sum_{i=1}^{M} \log(P(y^{(i)})) = \sum_{i=1}^{M} \log(\hat{y}^{(i)y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}})
\] (2.11c)

\[
= \sum_{i=1}^{M} (\log(\hat{y}^{(i)y^{(i)}}) + \log((1 - \hat{y}^{(i)})^{1-y^{(i)}}))
\] (2.11d)

\[
= \sum_{i=1}^{M} (y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))
\] (2.11e)

Maximizing Equation 2.11c is the same as minimizing its negation. As such, we arrive to our loss function

\[
\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^{M} (y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))
\] (2.12)

which is referred to as the binary cross-entropy loss. Note that when the true output (or label as it is referred to in the classification setting) \(y^{(i)}\) is 0, the term in the summation reduces to \(-\log(1 - \hat{y}^{(i)})\) and when it is 1, it reduces to \(-\log(\hat{y}^{(i)})\). As such, this loss function corresponds to the negative log-likelihood of our predicted values. If our model is able to correctly classify each training example, the total value of the loss function would be 0. Of course, it is unlikely that the model would ever obtain a loss value of exactly 0, nor would that be desirable as it may mean that the model will not be able to generalize well to new data (i.e., it has overfitted to the training examples). However, the more uncertain the model is, or the more certain the model is about an incorrect prediction, the more the model is penalized by the cross-entropy loss.

### 2.4 Neural Networks

With logistic regression, non-linearity comes from only one place and that is the logistic function. But, in many cases, one source of non-linearity may not be enough to fit the training examples, or to generalize to new examples, well enough. One way to have more non-linearly
is to have many of these logistic units, each with their own separately-learned weights, and sum them all up to get the final prediction. That is,

\[ \hat{y} = \sigma(X\tilde{w}_1) + \sigma(X\tilde{w}_2) + \ldots + \sigma(X\tilde{w}_L) \quad (2.13) \]

where \( L \) is the number of non-linear units we wish to use and \( \tilde{w}_i \) are learned weights for unit \( i \). Of course, such a model would only be able to output in the range \((0, L)\) since each individual logistic unit can only output in the range \((0, 1)\). To fix this, we can also put weights, denoted by \( \tilde{c} \), on each of the logistic units so that we can output in \( \mathbb{R} \):

\[ \hat{y} = \tilde{c}_1\sigma(X\tilde{w}_1) + \tilde{c}_2\sigma(X\tilde{w}_2) + \ldots + \tilde{c}_L\sigma(X\tilde{w}_L) \quad (2.14) \]

Note that we do not have to use the logistic function for each of the individual non-linear units; we can also use functions like tanh, sin, and ReLU [54]. This model can now be used for regression, but we can extend Equation 2.14 to the classification setting by simply wrapping it in a logistic function, as we did for logistic regression.

At this point, we can describe the learned parameters of our model with a weights matrix \( \tilde{W} \) where we just stack the individual weight vectors and \( \tilde{c} \) which are the weights on each of the non-linear, or activation, units. This makes it easier to talk about the model as well as being more efficient to train on GPUs (which we will simply note in passing in this thesis). At this point, it is simpler to think of our model as a directed graph that describes the function. As a graph, our model might look like Figure 2.1.

Here, the individual activation functions, as well as the features in the input vector and the output value, are represented as nodes. Each of the directed edges in the graph represents a multiplication of the value in the source node with a weight. The product is then used as input to the target node. A node sums all of its inputs and applies an activation function before sending the calculated value into all outgoing edges. In this setup, we can think of our model as having layers. There is the first input layer, and the final output layer. The “meat” of the model is found in the middle layer, referred to as the hidden layer. This layer has \( N \) nodes, or hidden units, each adding some non-linearity to the model. Of course, we need not limit the model by having the hidden layer immediately followed by the output layer. We can include as many hidden layers in our model as we think are necessary, each with their own number of nodes, and possibly their own activation functions. Each node in a hidden layer behaves in the same way: it sums its inputs, applies an activation function, and sends its output to all other nodes in the next hidden layer (or the output layer). We now redefine \( L \) as the number of hidden layers in our model and \( L^{(i)} \) as the number of nodes in layer \( i \). As we add more and more layers, the model can be thought of as getting deeper and deeper. Additionally, the number of parameters that are required to be learned can get very large, with the largest models having millions of parameters. These models, which are essentially many logistic regression units strung together, are called neural networks. They are named as such because they are an attempt to mimic the structure of neurons within our brains. Of course, these are but a rough approximation of the biological processes which occur in the brain. Indeed, we know very little about exactly how the brain “learns”. Finally, a layer in a neural network in which all nodes are connected (have outgoing edges) to all other nodes in the next layer is called a fully-connected layer.

Note that the graphical representation of a neural network shows how one training example would be processed, or propagated, through the model. In practice, sending examples one at a
2.4. Neural Networks

Figure 2.1: A graphical representation of a neural network. This neural network accepts a 4-dimensional input vector and outputs a single scalar. It also has one hidden layer with 5 hidden units.

time through the network would be too slow and in most software packages, neural networks are internally represented as a set of weight tensors which can be applied to many training examples at once, speeding up the training process. The method with which these networks are trained is called backpropagation \cite{55}. The general idea of backpropagation is that the value of the loss function at the output layer is propagated backwards through the network. At each hidden layer, a local gradient descent is made before continuing the loss propagation backwards through the network. With this algorithm, neural networks can be efficiently trained.

2.4.1 Gradient Descent

The method by which many machine learning models are trained is called gradient descent. Imagine that the current values of the trainable parameters of the model is $w$ and we wish to update it to $w'$ so that our the value of our loss function $L$ decreases (in the case we wish to maximize the loss function, we would employ the very similar algorithm of gradient ascent). In order to do this, we might take all the training examples to calculate the gradient of the loss function. We thus update our weights to be

$$w' = w - \eta \cdot \nabla L$$  \hspace{1cm} (2.15)

where $\eta$ is the amount to move in the direction of descending gradient, commonly called the learning rate or step size. When we use all the training examples to compute the gradient, we called this batch gradient descent, with one iteration through all the training examples is referred to as an epoch. However, computing this gradient can be extremely expensive if we have a lot of training examples. On the other end of the spectrum, we can update the weights based on a gradient calculated using only a single training example and repeat this for all training examples. This technique is called stochastic gradient descent. However, this can be an expensive process as well. The most popular method is mini-batch gradient descent where the gradient is computed on a subset of $M$ training examples. $M$ is thus called the batch size.
Modifications to the basic gradient descent equation (Equation 2.15) can be made in an attempt to allow the gradient to converge to a minima quicker. Popular alternatives of optimizers use an adaptive learning rate which alters $\eta$ over the course of the training. Such optimizers include AdaGrad [56], AdaDelta [57], RMSProp [58], and Adam and AdaMax [59]. However, there have been recent discussion on situations where these algorithms may not perform as well as standard gradient descent [60, 61]. Overall, it is likely that it is the optimizer coupled with appropriate hyperparameter tuning that will result in a well-converged model.

### 2.4.2 Validation and Testing

When computing the gradient on the entire training set, one can consider it as a “true” update to the gradient. However, this can cause problems when applying the trained model on unseen data points. This is a problem known as overfitting and can occur when the model is too focused on minimizing the loss on the training examples and does not attempt to generalize to unseen examples. One way to monitor overfitting is to divide data points into a training and test set. The model is trained only on the training set and is evaluated on the test set. In this way, we can observe how well the model generalizes to unseen data. While we will expect that the value of the loss function will be higher on the test set, if the difference is very significant, or if the loss function is increasing on the test set, then the model appears to be overfitting.

In addition to this, we may want to test how well different model architectures with different hyperparameters perform. However, we are unable to compare by using the test set as that would lead to a biased evaluation of the model (i.e. a particular model might be better on the test set than others, but may not on unseen data). To ameliorate this, we also divide the data into a validation set, on which the models are not trained, but which is used when comparing the different models. Typical splits of the data include a 90-5-5 or a 80-10-10 training/validation/test split. However, the exact split which is appropriate can depend on the data set. In general, a model will want to train on as much data as possible.

When a data set is particularly small, it may not be an option to even take 5% of the data as a validation set. If the model does not have enough data on which to train, it will not be able to learn anything about the data, let alone generalize to unseen data. In such cases, adopting a $k$-fold cross-validation scheme is more appropriate. First, the data are divided into $k$ subsets, or folds, of equal sizes. Then, one fold is assigned to be the validation set while the remaining $k-1$ folds act as the training set. The model is trained and the loss on the validation fold (validation loss) is calculated. This is repeated $k$ times until each fold has acted as the validation set exactly once, at which points the validation losses are averaged to get a final validation loss for the entire data set. Other cross-validation schemes include leave-one-out cross-validation where only a single data point acts as the validation set.

### 2.5 Recurrent Neural Networks

Up to now, we have assumed that the input $x$ is a $d$-dimensional feature vector. This is fine if each example in the data set can be represented as a single row in a relational database. However, this is not always the case. Take, for example, a time series as an input. A time series is measured at a particular frequency over a set range of time. We could, of course, use each of
these measurements as a single feature in a feature vector. However, the concern here would be that we are throwing away the temporal information in the series. When setting up a neural network, we never specified that the order of the features in the feature vector mattered. Now we have an issue where we do want to keep information about the order.

Another example where this is the case is if we used sentences as input. In a sentence, the order of the words, or even the characters, matters. Consider the sentence: “She picked up the orange basketball.” Certainly we cannot equate it with “up the picked orange She the basketball.”, nor could we say that the latter is grammatically correct. In addition, to understand the value, or meaning, of a particular word, we must understand its context. Does “basketball” refer to the sport or the ball? To whom does “she” refer? Is “orange” a noun or an adjective and, if it is the latter, what it is describing? While sentences may not necessarily have a temporal component, the order of the words still matters. Additionally, the lengths of the various sentences can differ wildly.

In order to tackle these issues with sequential data, recurrent neural networks (RNNs) were developed. The idea is simple: apply the same hidden layer to each time step of the input (or in the case of sentence input, to each word), along with some information about the previous applications of the hidden layer. In the case of a single time series, the first measured value $x^{[1]}$ goes through the hidden layer and an output $y^{[1]}$ is made. In addition, a hidden state vector $h^{[1]}$ for that time step is calculated as a function of the parameters of the network. The function takes the input of the current time step as well as the hidden state of the previous time step. The hidden state is initialized to some values (typically zeros) for the first hidden state. For general time step $t$:

$$h^{[t]} = \phi_h(\tilde{W}_x[h^{[t-1]}, x^{[t]}])$$

$$y^{[t]} = \phi_y(\tilde{W}_y h^{[t]})$$  \hspace{1cm} (2.16a)

where $[\ldots]$ represents the concatenation of two vectors, $\tilde{W}_x$ and $\tilde{W}_y$ are learned weight matrices, and $\phi_h$ and $\phi_y$ are some activation functions and are generally chosen to be the hyperbolic tangent function tanh. Again, biases are implicit in these equations. As a result of this, information from previous time steps can be propagated to inform the output at the current time step. The final output for the final time step is typically used as the prediction for the entire time series. We note that this model does not take into consideration future time steps when predicting for any given time step, which we will resolve in Subsection 2.5.2. If we were to draw a general RNN model unrolled, it would look like Figure 2.2 where the hidden $\phi$ layer is shared across time steps. While we denote the input and output of the model at each time step as a scalar, the model can easily be extended to use vectors or tensors. RNNs are trained in a similar fashion to standard neural networks via an algorithm called backpropagation through time (BPTT) [62, 63].

The architecture presented here works fine when a single output tensor is desired, but can be easily extended to output a sequence by collecting all the intermediate output values $\{y^{[1]}, y^{[2]}, \ldots, y^{[T]}\}$ where $T$ is the number of time steps in the input. In such a case, it is referred to as a many-to-many problem. However, there are also cases where the output sequence is of a different length than our input sequence, or where the output sequence is not of a fixed length and instead varies depending on the input. For example, a text summarization model might take in an entire document as input and its output should be approximately one to two
Figure 2.2: An unrolled RNN. In a many-to-one problem, only the last output, $y^{[T]}$, is kept as the final output.

paragraphs depending on how well the information in the document can be summarized. One solution, in the context of neural machine translation and explored in [27, 64, 65], is to use two RNNs. The goal of neural machine translation is to develop a neural network model that can translate an input sentence in one language to another language. The first RNN in the model is a many-to-one encoder which encodes the input sentence one word at a time and generates a single $d$-dimensional vector. In other words, it creates a vector representation of the entire sentence. The second RNN, called the decoder, then takes this vector as its initial input and generates as output the first predicted word in the translated sentence. The successive inputs to this RNN are then the previously predicted words. The decoder continues until a special end-of-sentence (“EoS”) token is generated. In this way, the output translation does not have to be the same length as the input sentence, which is appropriate in most cases. The “EoS” token must also be present in the training examples for this to work correctly so that the model can “learn” when a sentence should be terminated.

2.5.1 Variants of RNNs

One issue, identified by Hochreiter and Schmidhuber [66], of RNNs is that they are prone to exploding gradients where error gradients (i.e. weight changes) are accumulated during BPTT and cause excessive updates to the model’s weights and lead to instability, and vanishing gradients where error gradients disappear, due in part to activation functions like tanh whose derivatives lie in a very small range, and cause the model train very slowly or even get stuck.

The solution developed by Hochreiter and Schmidhuber [66] and extended to by Gers et al. [67] and Graves and Schmidhuber [68], called the long short-term memory (LSTM) unit, modifies the simple tanh function used for a RNN’s hidden state with a more complicated configuration that uses three gates—a forget gate $f^{[t]}$, an input gate $i^{[t]}$, and an output gate $o^{[t]}$—as well as a hidden cell state $s^{[t]}$, and can be trained with standard BPTT. The equations for the
2.5. Recurrent Neural Networks

LSTM unit are:

\[
\begin{align*}
    f^{[t]} &= \sigma(\tilde{W}_f[h^{[t-1]}, x^{[t]}]) \\
    i^{[t]} &= \sigma(\tilde{W}_i[h^{[t-1]}, x^{[t]}]) \\
    o^{[t]} &= \sigma(\tilde{W}_o[h^{[t-1]}, x^{[t]}]) \\
    s^{[t]} &= f^{[t]} \odot s^{[t-1]} + i^{[t]} \odot \tanh(\tilde{W}_s[h^{[t-1]}, x^{[t]}]) \\
    h^{[t]} &= o^{[t]} \odot \tanh(s^{[t]})
\end{align*}
\]

where \(\tilde{W}_f, \tilde{W}_i, \tilde{W}_o, \) and \(\tilde{W}_s\) are all learned matrices and \(\odot\) denotes the Hadamard product, or element-wise multiplication.

A single LSTM unit contains a cell state local to itself that can trap error gradients, preventing the exploding and vanishing gradients problems by not allowing them to propagate through time. This trapping mechanism is implemented by the three gates. The role of the forget gate is to learn how much of the previous cell state should be kept in the current cell state. The role of the input gate is to learn the amount of contribution the input at the current time step has to the cell state, and the output gate learns how much of the cell state should be output as the hidden state (and consequently the prediction output \(y^{[t]}\) as that equation remains unchanged from Equation 2.16b). Note that the cell state for the current cell \(s^{[t]}\) is a function of the cell state of the previous cell \(s^{[t-1]}\). Because of this, long-term dependencies can be learned if the forget gate lets most of the previous cell state through and the input gate restricts most of the current time step from interfering.

Cho et al. [27] introduce a simplified version of the LSTM unit called the gated recurrent unit (GRU). In GRUs, the forget and the input gates are combined into a single update gate \(z^{[t]}\) which determines how much of the previous hidden state is kept. The GRU also does not use an additional hidden cell state. In addition, the LSTM’s output gate is replaced with a reset gate \(r^{[t]}\). The lower the value of the reset gate, the more the input at the current time step \(x^{[t]}\) contributes to the current hidden state. The formulae for the GRU are:

\[
\begin{align*}
    r^{[t]} &= \sigma(\tilde{W}_r[h^{[t-1]}, x^{[t]}]) \\
    z^{[t]} &= \sigma(\tilde{W}_z[h^{[t-1]}, x^{[t]}]) \\
    h^{[t]} &= z^{[t]} \odot h^{[t-1]} + (1 - z^{[t]}) \odot \tanh(\tilde{W}_h[r^{[t]} \odot h^{[t-1]}, x^{[t]}])
\end{align*}
\]

where \(\tilde{W}_r, \tilde{W}_z, \) and \(\tilde{W}_h\) are learned weight matrices.

Because the GRU contains less parameters to learn than the LSTM unit, it is typically faster to train a model that uses GRUs. Consequently, however, the GRU has less representational power than the LSTM. Chung et al. [69], Józefowicz et al. [70], and Greff et al. [71] test various RNN variants and come to the same conclusion that LSTMs and GRUs perform similarly and are, generally, the best recurrent units to use. However, the search for better RNN models is still ongoing. Zoph and Le [72] use reinforcement learning to automatically learn a new recurrent unit which they call a NASCell and reports better performance on language modeling tasks. Seo et al. [73] propose a query-reduction network (QRN) which performs better on question-answer (QA) problems. The authors also demonstrate how visualization of the model’s weights can be interpretable. Ostmeyer and Cowell [74] introduce a recurrent model that uses a recurrent-weighted average (RWA) and report better performance than LSTM on a
variety of tasks. However, for this thesis, we will not explore these architectures nor will we attempt to develop a novel recurrent unit.

2.5.2 Bidirectional RNNs

Introduced by Schuster and Paliwal [75], bidirectional RNNs (BRNNs) are a solution to the restriction in RNNs that, at a given time step, the recurrent unit can only learn from past time steps, but not future ones. Essentially, in a BRNN, each recurrent unit has two hidden states: one representing the past cells $\vec{h}[t]$, and one representing the future cells $\overrightarrow{h}[t]$. Thus, architectures like LSTM and GRU easily extend to BRNNs. Training BRNNs is similar to training other RNNs, but requires a slight modification to the BPTT algorithm [75]. For clarity, when referring to the hidden state of a BRNN, we mean the concatenation of both $\vec{h}[t]$ and $\overrightarrow{h}[t]$:

$$ h[t] = [\vec{h}[t], \overrightarrow{h}[t]] \quad (2.19) $$

2.5.3 Attention

While the hidden states of BRNNs can contain contextual information of time steps in both the past and the future, because of how RNNs are designed, the hidden states will be inherently biased to “remember” more about time steps closer to it. Additionally, while encoder-decoder models can work for many-to-many problems, the requirements for a fixed-length encoding of the input can be problematic. Particularly, for long sequence lengths [28], or if the dimensionality of the encoding is not sufficient to adequately contain information about the entire sequence, it will be hard for the features of the encoding vector to accurately represent all sequences. This, in turn, sets up the decoder for disaster. One resolution to these issues, which has arguably been one of the biggest breakthroughs in RNNs, is the idea of attention [29]. A model that uses attention can bypass the restriction of a fixed-length encoding vector and instead introduces a dynamic-length representation of the input sequences while also being able to understand the global context of each element in the sequence.

We will now describe the model as introduced by Bahdanau et al. [29]. To encode the input sequence, we use a BRNN, which allows for each element in the sequence to have some information about the surrounding elements. At this point, however, we do not condense the input into a fixed-length vector. Instead, we introduce a new decoder, the attention mechanism. For each element in the output sequence $\hat{y}[t]$ for $t \in [1, T_y]$ where $T_y$ is the number of elements in the output sequence, the attention mechanism takes all the hidden states of the BRNN, $h_x^s$ for $s \in [1, T_x]$ where $T_x$ is the number of elements in the input sequence, and computes a context vector $c^t$. The context vector is then fed into a RNN, which we will refer to as the post-attention RNN, and which is of similar architecture to a GRU but accepts an additional input $\hat{y}^{[t-1]}$:

$$ r^{[t]} = \sigma(\hat{W}_r[c^t, h^{[t-1]}, \hat{y}^{[t-1]}]) \quad (2.20a) $$
$$ z^{[t]} = \sigma(\hat{W}_z[c^t, h^{[t-1]}, \hat{y}^{[t-1]}]) \quad (2.20b) $$
$$ h_y^{[t]} = (1 - z^{[t]}) \odot h^{[t-1]} + z^{[t]} \odot \tanh(\hat{W}_h[c^t, r^{[t] \odot h^{[t-1]}}, \hat{y}^{[t-1]}]) \quad (2.20c) $$
where $h^{[t]}_t$ is the hidden state of the post-attention RNN at time $t$ and again $\tilde{W}_r$, $\tilde{W}_c$, and $\tilde{W}_h$ are learned weight matrices.

The context vector $c^{[t]}$ is computed by taking a weighted sum of each of the hidden states in the BRNN:

$$
c^{[t]} = \sum_{s=1}^{T_x} \alpha^{[t]}_s h^{[s]}_s
$$

(2.21a)

where

$$
\alpha^{[t]}_s = \frac{\exp(e^{[t]}_s)}{\sum_{i=1}^{T_x} \exp(e^{[t]}_i)}
$$

(2.21b)

$$
e^{[t]}_s = \tilde{v}^T \tanh(\tilde{W}_e [h^{[s]}_s, h^{[t-1]}_y])
$$

(2.21c)

and $\tilde{v}$ and $\tilde{W}_e$ are learned weight matrices.

In Equation 2.21b, we are essentially putting $h^{[s]}_s$ and $h^{[t-1]}_y$ through a short neural network without a hidden layer. Additionally, we bring attention to Equation 2.21b where we are computing what is called the softmax of $e^{[t]}_s \in \mathbb{R}^{T_x}$. This results in the entries of $\alpha^{[t]}_s \in \mathbb{R}^{T_x}$ summing to one. As such, each entry $\alpha^{[t]}_s$ can be thought of as the relative importance of the input features at a particular time step $s$ when predicting the output sequence at time $t$. This allows the model to learn on which time steps to focus when it makes with its prediction at a global context (over the entire input sequence) rather than making its decision on the hidden states of the BRNN which only have local context (context of nearby time steps). This also makes it natural to visualize $\alpha^{[t]}_s$ as attention weights so that we can make an hypothesis as to why the model found particular time steps important.

Other implementations of attention exist. Luong et al. \[76\] introduce variants of calculating $e^{[t]}_s$ as well as introducing the idea of local attention, which only focuses on a subset of the input time steps rather than all of them. In a similar vein, Xu et al.\[33\] discuss the idea of soft and hard attention, where the latter focuses on specific times steps, rather than using a softmax over a subset of them. They also discuss the idea of doubly stochastic attention which uses an additional parameter $\beta$ which appears to help for their task. Lin et al. \[77\] introduce the idea of self-attention to construct a matrix representation of the input sequence by relating the various elements of the sequence to one another. Vaswani et al. \[30\] take it a step further by only using attention in their model, introducing scaled dot-product attention and multi-head attention.

### 2.6 Clustering

We now consider an instance of unsupervised machine learning called clustering where we wish to group, or cluster, examples in our data set. For example, suppose we have a data set of animal images and we want to cluster based on the biological family of the animal in the image. For example, we want a cluster of cat images, a cluster of dog images, and so on. However, we do not know beforehand how many families there are in the data set. If the data set only has cat images, should there be one large cluster? Or does it make sense to cluster by species, or subspecies? Depending on our application, it may also make sense to cluster images by colour,
or by size, or by some other property. Now imagine we have a data set of players. How should we cluster the players? Does it make sense to cluster by play style? If so, how many different styles are there? What about clustering by the player’s age or by the month the player started playing? What if we want to cluster by some complex combination of these features?

In general, there may not be one right answer to the clustering problem. As a result, it is difficult to evaluate the output of a clustering algorithm. We cannot apply a conventional loss function since there are no target outputs on which to evaluate. However, we still need to determine a function to optimize so that the algorithm can improve its clustering over time. In some cases, we may also have to manually tell the algorithm how many clusters to look for, a value we will call $k$. When the algorithm finds these $k$ clusters, it is up to us to manually evaluate them and see whether the clusterings can be of use. Of course, we can still apply clustering even when we do know the actual cluster labels of the data points. In these cases, there are various metrics we can use to evaluate the clustering. However, we will not explore these here.

### 2.6.1 $k$-means

The $k$-means clustering algorithm, also referred to as Lloyd’s algorithm [78], is a simple and popular clustering algorithm. The algorithm discovers $k$ centroids and data points are assigned to the centroid to which it is closest using some distance function, such as the Euclidean distance. As a result, this algorithm effectively minimizes the squared distances of points within each cluster (i.e. the variance of each cluster). This results in rather spherical clusters.

The algorithm begins by randomly initializing the $k$ centroids. These do not need to be actual points in the data set but must not be all initialized to the same value. The algorithm then alternates between an assignment step and an update step. In the assignment step, all data points are assigned to the centroid that is nearest to them. This creates the initial clusters. Then, in the update step, the centroids are then moved so that they are in the middle of all the points assigned to them. Effectively, a new centroid for each cluster is computed and the old centroid for the cluster is replaced. The algorithm continues alternating between the assignment step and the update step until no new cluster assignments are made (i.e. the centroid assignments for each data point in the data set following an assignment step does not change) in the update step. Practically, we can also set an upper bound as to how many iterations the algorithm performs.

The algorithm has its advantages in that it is very simple to implement and understand and its complexity is generally linear in the number of data points in the data set (so long as we upper bound the number of iterations). However, the algorithms has several key disadvantages. The first is that we must supply the value of $k$. Generally, this is an unknown value and could require multiple runs of the algorithm varying $k$ before getting a satisfactory clustering (see Figure 2.3a). The second is that the algorithm only works well for spherical clusters. If the “true” clusterings are of an oblong shape, the algorithm will fail to find it (see Figure 2.3b). Finally, $k$-means will completely fail to identify nested clusters (see Figure 2.3c).

In Figure 2.3, we see $k$-means applied to various data sets. The clusters of the data sets are identified using unique colours. In Figure 2.3a, there are five blobs in the data sets which $k$-means successfully identifies. Note however, we had prior knowledge that there are five clusters in the data. In practice, other values of $k$ might seem reasonable enough and that is fine since we do not have a hard evaluation metric for the clustering. In our example, if $k = 2$
appears to satisfy the needs of the application, it is fine. However, sometimes $k$-means fails to capture structural features of the true clusters such as in Figure 2.3b and Figure 2.3c. For Figure 2.3b, one can get the true clusters with $k = 7$ by manually grouping four of the found clusters together and grouping the other three together. Of course, in practice, it is unlikely we would have such intuition to do so since we are often working with high-dimensional data sets and projected visualizations of them may not always clearly show the true clustering.

### 2.6.2 DBSCAN

$k$-means clustering defines clusters using centroids which act as a center of gravity for all points in its neighbourhood. However, we clearly see two clusters in Figure 2.3b that do not fit this definition of a cluster. DBSCAN [79] uses a density-based definition of clusters. Intuitively, clusters are regions with a high density of points separated by areas of low density. In particular, Ester et al. [79] define the notion of *density-reachable* and *density-connected* in order to formally define a cluster.

The DBSCAN algorithm is parameterized by two parameters: $\text{minPts}$ and $\varepsilon$. Consider a point $p$ and the neighbourhood $N_p$ of points around $p$ within a radius $\varepsilon$. A point $q$ is *directly
density-reachable from $p$ if $q \in N_p$ and $|N_p| \geq \text{minPts}$. $q$ is then density-reachable from $p$ if there exists a chain of directly density-reachable points going from $p$ to $q$. Note that this relation is not symmetric. Finally, $p$ and $q$ are density-connected if there exists another point $o$ from which both $p$ and $q$ are density-reachable. This relation is symmetric.

We can now define a cluster $C$. Informally, $C$ contains a core wherein for all pairs of points $p,q$ in that core, $p$ and $q$ are density-connected (remember that this is a symmetric relation). Additionally, $C$ may contain border points which are defined as the set of points $b$ where for all $p \in C$, $b$ is density-reachable from $p$ but $p$ is not density-reachable from $b$. This definition also allows us to introduce the notion of noise points. Recall that in $k$-means, we had to classify all points as being a member of a particular cluster. However, it may be possible, and is likely, that some points are simply noise. In DBSCAN, noise points are those which are not a member of any cluster $C$.

![Figure 2.4: DBSCAN on various data sets and using various values of $\epsilon$.](image)

In general, we have to test various values of $\text{minPts}$ and $\epsilon$. However, another advantage of DBSCAN over $k$-means (other than the redefinition of a cluster) is that we do not need to know $k$ beforehand. As for picking the hyperparameters of DBSCAN, Ester et al. [79] offer a heuristic. In Figure 2.4, we show the effect of varying $\epsilon$ while keeping $\text{minPts}$ constant at 5 on the same three data sets as presented in Figure 2.3. Any points labeled as noise are shown...
in grey. We see that at appropriate values of $\epsilon$, the algorithm correctly identifies the clusters in all three data sets, minus a couple points mislabeled as noise in Figure 2.4b. However, we note that its improper use can have a dramatic effect. Note that the default value of $\epsilon$ in the popular scikit-learn library [80] is 0.5 and that would have lead to incorrect clustering in all cases.

2.6.2.1 HDBSCAN*

HDBSCAN* is an extension to DBSCAN and was introduced by Campello et al. [81]. Essentially, it adds a hierarchical component to the algorithm by introducing minimum spanning trees. The big advantage of HDBSCAN* is that the $\epsilon$ hyperparameter is eliminated and leaves only the $minPts$ parameter. However, both algorithms are fairly insensitive to this parameter and HDBSCAN* has effectively no parameters to tune (other than perhaps the choice for the distance metric).

We provide an informal description of the algorithm here and refer the interested reader to Campello et al. [81] for more. HDBSCAN* works by starting with all points in a single cluster and determines the required $\epsilon$ value for this. It then slowly shrinks the $\epsilon$ parameter until the cluster fragments into smaller clusters. This is done efficiently using a minimum spanning tree. It continues this process until a termination condition, such as the size of the clusters, is satisfied. This results in a tree-like structure and we might visualize this process as a dendrogram (see Section 4.6 for a concrete example of this). The algorithm then extracts “prominent” clusters by defining the notion of the stability of a cluster and working bottom-up from the cluster hierarchy tree. This can also be done in a semi-supervised manner to impose conditions on the clustering.

We show the application of the HDBSCAN* algorithm in Figure 2.5 using the implementation provided by McInnes et al. [82]. The algorithm is able to correctly identify the clusters (with some noise) in all cases without any hyperparameter tuning.
Chapter 2. Relevant Theory

(a) HDBSCAN* on a blobs data set.

(b) HDBSCAN* on a moons data set.

(c) HDBSCAN* on a circles data set.

Figure 2.5: HDBSCAN* on various data sets.
Chapter 3

Forecasting Hourly Metrics in *My Singing Monsters*

3.1 Motivation

In this section, we discuss the application of recurrent neural networks for forecasting hourly features, or metrics, in *My Singing Monsters* (MSM). For MSM, Big Blue Bubble aggregates various hourly game metrics across the entire player database. These data can be grouped by fields such as by platform or by country, but for this section we aggregate across all fields and simply group by hour. There are six hourly metrics which we will observe: the number of new players (*new_users*), the number of game sessions (*sessions*), the number of purchases made (*purchases*), the number of offers claimed (*offers*), the total revenue from purchases (*iap_dollar*), and the total revenue from offers (*offers_dollar*). For our training examples, we use data from October 1, 2016 to September 30, 2018, inclusive. We selected this particular date range because data from before October 1, 2016 were often incomplete or missing. A snapshot of the metrics is shown in Figure 3.1.

The goal is that, given any date, we use the hourly metrics of the previous four days (or 96 hours) to forecast the next 24 hours of metrics. In other words, our input is a $96 \times 6$ matrix of hourly metrics and our output is a $24 \times 6$ matrix of hourly metrics. The numbers here were arbitrarily chosen, but appear to work reasonably well. The model should automatically learn any periodicities that may be present in the data, such as those on a hourly, daily, or even weekly basis, as well as any trends that occur over time. It should not be expected to be able to predict all outliers in the data, but should be able to make a good, general forecast.

We can use such a model to plan ahead in the event we wish to schedule some event or activity when a particular feature is forecasted to be at its lowest or highest within the next 24 hours. Additionally, we can compare the forecasted peaks/troughs with the past days’ values. For example, if the projection is that the number of new players in the next 24 hours is lower than what it was the day before, it may be worthwhile to investigate as to its cause. It may be that it is simply a seasonal trend, or that the number of new players has been decreasing over time, or that some combination of the hourly metrics have led the model to believe that the number of new players will be lower. However, it may be something more immediately serious such as a bug in the game disconnecting players or an error with the app store causing the game
to not show up. In any case, it might be critical to take action until the forecasted metrics have some upward trajectory.

In another use case, we can use the model to identify whether a point is significantly different than its forecasted value. If so, that may be a reason to investigate further. For example, if the number of purchases has a temporary spike that is significantly above the expected (forecasted) value, it may be that some fraudulent purchases have been made. It could also be due to an error in the software or simply a subset of players simultaneously deciding to make in-app purchases independently. Again, manual investigation would be required to confirm or reject the significance as being important, but the model would allow for a more focused review.

Finally, if we add an attention mechanism (Subsection 2.5.3), we can then visualize the weights placed on each of the input time steps (see Equation 2.21b) in order to see on which hours the model focused on in its forecast, which helps clarify how the model came to its
output. Note that this does not satisfactorily answer the question of why the model puts more attention on certain time steps (though the obvious and technically correct answer is that it is the choice which minimizes the model’s loss function). Any interpretations should be kept as such and should not be taken as ground truths about the data set. Different models, or even the same models trained on different initialization of its weights, can lead to different results and may focus attention on different hours.

## 3.2 Data Preparation

The data used in this application are stored in an Amazon Redshift database. As such, a simple query enabled its transfer to a regular comma-separated values (CSV) file. From there, the data are loaded into a Python environment and stored in a pandas DataFrame. Given the date range we specify, we expect there to be 17520 rows in our DataFrame. However, there are only 17512 rows which means that data for eight hours are missing. The missing hours come from the morning hours of a particular date. We also note that the values of the metrics for the hours that are present for this day are abnormally low, with many being 0. It is likely that the missing data are due to a game server outage or maintenance. In order to rectify this, we take the average of feature values of the previous and the next days and use that in place of metrics values for the day with missing values. This is a rather simple approach but will likely be sufficient as the surrounding days are quite similar. Finally, we acknowledge that the true values of the data here were likely indeed 0, but we are fabricating data in order to continue the general trend of the data. Considering that this happened only once in the span of two years, we accept that one day’s worth of data are technically incorrect in order to prevent our model from over-fitting, if ever so slightly, to such an anomalous event.

Next, the data are plotted to get an idea of its shape. As expected, the data are quite periodic, with a peak occurring in the middle of the day for all metrics. It should be noted here that the times in the database are stored in UTC. The means for all the metrics also appear to exhibit a slight downward trend over time and the variance in the metrics also appear to decrease over time. This is more visible in a feature like offers, but less so in offers_dollar. While not shown in Figure 3.1, the scale of the various metrics ranges quite wildly, so our next step is to normalize the data.

Scaling all metrics to be within a similar range is a common approach in machine learning. The idea is to avoid the issue of exploding/vanishing gradients by ensuring the weights need not compensate for vast differences in the scales of the input and output values. Additionally, neural networks are generally faster to converge when metrics are normalized. For our application, we will use the Yeo-Johnson transformation as implemented in scikit-learn. The transformation is a power transformation that tries to normalize the data. The exact transformation, $\psi$, is given by

$$
\psi(\lambda, x) = \begin{cases} 
((x + 1)\lambda - 1)/\lambda & (x \geq 0, \lambda \neq 0) \\
\log(x + 1) & (x \geq 0, \lambda = 0) \\
-((x + 1)^{2-\lambda} - 1)/(2 - \lambda) & (x < 0, \lambda \neq 2) \\
-\log(-x + 1) & (x < 0, \lambda = 2)
\end{cases}
$$

(3.1)

https://aws.amazon.com/redshift/
where $\lambda$ is a parameter of the transformation. scikit-learn attempts to find the best values of $\lambda$ for each feature using Brent’s method \[\text{[85]}\]. The values of $\lambda$ for each feature (in the order they were introduced) are 0.2426, 0.3348, 0.2378, 0.0774, 0.2052, 0.2232. Note that because of the non-negative nature of our data, the transformation reduces to the first case in Equation 3.1. Once all data points are transformed, the mean of each feature is subtracted and the resulting value is divided by the variance in each feature.

It should be noted that in order to interpret the quality of the forecasts, it would be sensible to scale up forecasted values back to their regular range. This can be easily achieved by applying the inverse transformation to the model’s output. As such, it is a good idea to save the parameters of the normalizing transformation (that is, the lambdas, means, and variances for each feature) along with the learned parameters of the trained ML model.

Next, we have to divide up our long sequence of data points into training examples. The easiest way to divide up the data points is to break up the sequences into chunks of length 120. The first 96 data points in a chunk represent the last four days worth of points and is used to forecast the last 24 data points. With this process, we have a total of 17401 training examples.

Finally, as is typical in machine learning, we divide our training examples into explicit training, validation and test sets. First, we randomly take 10% of the original training data and assign them to be the test set. Then, we randomly take 10% of the remaining data as the
validation set and call the leftover data points the training set. Thus, the sizes of the training, validation, and test sets are 14094, 1566, and 1741 respectively. Note that the examples in the data set are overlapping and this may raise a concern that data in the test set is effectively leaked into the training set. For this reason, we also evaluate the model on completely new data in Subsection 3.5.3. For this reason, this evaluation should be considered the least biased and is the most representative of trained model.

### 3.3 Model Architecture

We design our model so that it outputs three different values: the median and the 2.5% and 97.5% quantiles. Doing so allows us to estimate the 95% prediction interval around the predicted median and will be useful for identifying anomalous points in our time series. We set up our general architecture as done in Bahdanau et al. [29]. We have an initial RNN encoder which processes the input time series. Then, we apply the attention mechanism multiple times for each time step in the output time series. The output of this mechanism is then fed into a RNN which acts as a decoder. Finally, the output of the decoder is sent through separate fully-connected layers for each of the three outputs. Note that the weights for these layers are shared across time steps. For our model, we will use GRUs [27] over simple RNN units or LSTM units [66] since GRUs are faster to train and generally perform as well as LSTMs.

In order to come to our final architecture, we use the hyperopt package [86] in Python. The package implements a hyperparameter optimization algorithm called tree-structured Parzen estimator (TPE) [87]. With it, we are able to not only optimize the number of hidden units in our RNN layers, but also whether to use one or two RNN layers, which activation function to use in the attention mechanism (substituting the tanh function in Equation 2.21c), and what kinds of regularization layers we should use, if any. The full list of hyperparameters over which we optimize is presented in Table 3.1. We run the algorithm for 50 trials, each time training a model with the hyperparameters chosen by hyperopt. We train the model as described in Section 3.4 in a Google Colaboratory notebook which is equipped with a Tesla K80 GPU and optimize around the validation loss after four epochs. The hyperparameter choices that achieve the lowest validation loss are presented in Table 3.2 with the order of the columns in Table 3.2 matching the order of the rows in Table 3.1 (we use abbreviated names in the latter case to manage space). A diagram of the final model is shown in Figure 3.3. Note that after each RNN layer in our model, we test whether it is effective to apply batch normalization [88, 89] or Gaussian dropout [90]. Here, we are applying batch normalization between the layers as described by Laurent et al. [89] rather than within the layers (on the hidden-to-hidden connections within the RNN layers) [91].

Of the top ten models, about half of them opted to use two layers of GRU in the pre-attention stage. Almost all models use batch normalization after the first GRU in the pre-attention encoder, but it is never used elsewhere. Many models also employ Gaussian dropout to varying degrees, though the best model does not. Interestingly, it is also essentially never used in the decoder GRU. Another source of regularization is the relatively small batch sizes [95] used in all models, with the best model using the minimum batch size. On the other hand,
Table 3.1: Optimized hyperparameters for hourly metrics model.

<table>
<thead>
<tr>
<th>gru1_units</th>
<th>bn1</th>
<th>gd1</th>
<th>gru2_units</th>
<th>bn2</th>
<th>gd2</th>
<th>actvn_fn</th>
<th>gru_post_units</th>
<th>bn_post</th>
<th>gd_post</th>
<th>batch_size</th>
<th>val_loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>False</td>
<td>0.6583</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>relu</td>
<td>226</td>
<td>False</td>
<td>False</td>
<td>78</td>
<td>0.3184</td>
</tr>
<tr>
<td>256</td>
<td>True</td>
<td>0.0388</td>
<td>146</td>
<td>False</td>
<td>0.0685</td>
<td>selu</td>
<td>214</td>
<td>False</td>
<td>False</td>
<td>89</td>
<td>0.3159</td>
</tr>
<tr>
<td>230</td>
<td>True</td>
<td>0.1437</td>
<td>210</td>
<td>False</td>
<td>0.2861</td>
<td>softplus</td>
<td>219</td>
<td>False</td>
<td>False</td>
<td>56</td>
<td>0.3129</td>
</tr>
<tr>
<td>109</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>relu</td>
<td>194</td>
<td>False</td>
<td>0.0609</td>
<td>33</td>
<td>0.3127</td>
</tr>
<tr>
<td>175</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>tanh</td>
<td>207</td>
<td>False</td>
<td>False</td>
<td>63</td>
<td>0.2968</td>
</tr>
<tr>
<td>112</td>
<td>True</td>
<td>0.5239</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>selu</td>
<td>232</td>
<td>False</td>
<td>False</td>
<td>44</td>
<td>0.2910</td>
</tr>
<tr>
<td>212</td>
<td>True</td>
<td>0.0434</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>softplus</td>
<td>160</td>
<td>False</td>
<td>False</td>
<td>47</td>
<td>0.2905</td>
</tr>
<tr>
<td>251</td>
<td>True</td>
<td>0.0404</td>
<td>146</td>
<td>False</td>
<td>0.7158</td>
<td>selu</td>
<td>217</td>
<td>False</td>
<td>False</td>
<td>49</td>
<td>0.2878</td>
</tr>
<tr>
<td>70</td>
<td>True</td>
<td>0.4736</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>softplus</td>
<td>230</td>
<td>False</td>
<td>False</td>
<td>43</td>
<td>0.2534</td>
</tr>
<tr>
<td>105</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>relu</td>
<td>243</td>
<td>False</td>
<td>False</td>
<td>32</td>
<td>0.2472</td>
</tr>
</tbody>
</table>

Table 3.2: Top ten model architectures based on validation loss for forecasting hourly metrics.

the number of hidden units in the GRUs are on the high end of the range, with most hovering around 150 to 200 hidden units. Finally, activation functions used in the attention mechanisms are dominated by rectifiers such as ELU \[\text{elu}\] and SELU \[\text{selu}\].

### 3.4 Model Training

With the optimized parameters selected, the model is built and trained using the Keras \[\text{Keras}\] library in Tensorflow \[\text{Tensorflow}\]. In order to speed up training on the GPU, we replace the GRU cells in the model with the CuDNNGRU available in Tensorflow which greatly speeds up the training time. The final model contains 408151 trainable parameters. We train the model using a COCOB optimizer \[\text{COCOB}\], an adaptive learning rate optimizer which does not require any hyperparameter tuning.

In order to predict the three outputs—the 2.5%, 50%, and 97.5% quantiles—that we want, we use the quantile loss. The loss function for the \( \tau \)th quantile is given by

\[
\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{M} \sum_{i=1}^{M} \left( y^{(i)} - \hat{y}^{(i)} \right) (\tau - \mathbb{1}_{[y^{(i)} - \hat{y}^{(i)} < 0]}) \quad (3.2a)
\]
where \( \tau \in (0, 1) \) and \( \mathbb{1} \) is an indicator function. This can be written equivalently as

\[
\frac{1}{M} \sum_{i=1}^{M} \begin{cases} 
\tau (y^{(i)} - \hat{y}^{(i)}) & \text{if } y^{(i)} - \hat{y}^{(i)} \geq 0 \\
(\tau - 1)(y^{(i)} - \hat{y}^{(i)}) & \text{if } y^{(i)} - \hat{y}^{(i)} < 0
\end{cases} \quad (3.2b)
\]

\[
\frac{1}{M} \sum_{i=1}^{M} \max(\tau (y^{(i)} - \hat{y}^{(i)}), (\tau - 1)(y^{(i)} - \hat{y}^{(i)})) \quad (3.2c)
\]

for ease of implementation. The total loss of our model is then the sum of the losses for each of the quantiles we output. In this way, we do not have to retrain the model multiple times for each output and the entire system is trained end-to-end.

Again, we are training in a Google Colaboratory notebook. For the hyperparameter optimization, we train for only four epochs. When training the final model as identified in Table 3.2, we instead train for a full 16 epochs. This takes about 17 minutes to train. Because the CuDNNGRU is only usable on a GPU, when using the model on a system without a GPU, the basic GRU class must be used. Luckily, with some minor modifications, we can simply load in the weights trained on the GPU onto the CPU-usable GRUs. The final training and validation losses for the final model are 0.1714 and 0.1749 respectively, with the constituent losses shown in Figure 3.4. These values are not directly interpretable, however, considering that we have scaled the inputs and outputs of the model.
3.5 Model Evaluation

With the model trained, we will now look at how well the model is able to forecast the hourly metrics. For this, we will also evaluate on both the training set and the test set. Finally, we will observe how well the model does on data collected during the first two weeks of November 2018 which would be a time frame which the model has yet to see.

3.5.1 Evaluation on Training and Test Sets

First, we plot the 24-hour forecasts for two random times from both the training set and the test set in Figures 3.5 and 3.6. Due to confidentiality reasons, the scale of the y-axis has been omitted. In all four plots, we see that the predicted curve for the median is relatively smooth compared to the true values of the hourly metrics. This is acceptable as it is hard to expect the model to be able to predict the noise in the data considering that it is not possible to extract all sources of noise given just the six hourly metrics. Additionally, we see that the forecasts follow the same trend as the true values. Considering that these examples come from different times of the day, it would appear that the model has been able to identify daily periodic trends from the data. Furthermore, the area contained within the predicted 2.5% and 97.5% quantiles, which is an estimation of the 95% prediction interval, contains most of the true values.

There are instances, however, where the observed values lie outside this range. The most notable instance is in the first training example in Figure 3.5 where we can visually see a notable dip in all metrics for the evening of November 2, 2016. Generally, we tend to observe irregular points for the purchase- and offer-related metrics, as spikes in particular can be quite random. Regardless, we will refer to any point outside of the 95% prediction interval as anomalies. Anomalies for the new_users and sessions should probably be considered more urgent to investigate than purchase- and offer-related metrics since the latter are expect-
3.5. Model Evaluation

edly noise and prone to random spikes. Additionally, not all anomalies warrant investigation. Rather, the model serves as a sort of filter to quickly allow a human analyst to narrow down on where potential problems might be. However, one could also adjust the size of the prediction interval by simply retraining the model and tuning the loss function if one wanted less or more anomalies being identified.

3.5.2 Attention Maps

To assist in our understanding of how the model arrives to its forecasts, it is interesting to visualize the attention weights of the model for each of the four examples. Each time step in the input contributes to an attention context vector of length 96 (the number of hours in the past on which we are forecasting) which is constructed for each of the 24 forecasted time steps. Thus, each forecast corresponds to a 24 x 96 matrix of attention weights where each row of the matrix should sum to 1. We can then colour the cells of the matrix based on the magnitude of the weights and create a colour map of the weights. We show these in Figures 3.7 and 3.8. We use a red colour scale here, with a larger weight being assigned to a darker shade of red.

We observe an interesting behaviour in the attention maps. Firstly, the model appears to only focus on a limited number of narrow hour ranges to make its forecasts. Secondly, the same hour range is used to make the entire forecast, not just a part of it. Thirdly, we see that the model generally appears to use attention for the first few predictions. For the later predictions, it would appear that the model does not really make use of attention, or at least to a lesser degree than earlier predictions. In other words, as we move up in the attention map, the weights are generally more spread out. For the first training example, the model appears to focus primarily on 12-18 hours before the forecasted hour. For the second training example, there are three ranges of attention: one at about 12 hours in the past, another at about 60 hours, and a final one at about 84 hours in the past. Notably, the attention is more focused on the earlier ranges. We see similar behaviours in the examples from the test set.

It is important to not generalize what we see here across all examples in the training and test sets, or even for all predictive models. For a different model, we might see very different attention maps. In fact, previous models that we trained had attention maps that looked a lot different than the ones shown here. The attention maps are a neat tool to understand how this model makes its forecasts, but not why. As better models are applied to this task, attention maps are likely to change so it is important to only consider attention maps with the model that produced them and to not generalize them as ground truths about the actual data set.

3.5.3 Evaluation on New Data

One concern with our training/validation/test split is that while the test set technically does consist of sequences which the model has not seen while training, the sequences are likely to be very similar to other sequences in the training and validation sets since they all come from the same time range. For this reason, we will also evaluate our model on data collected during the first two weeks of November 2018 which would not contain any hours present in our initial data set. We show two examples from this new data set in Figures 3.9 and 3.10 along with their corresponding attention maps.
We observe similar behaviour and performance as we saw with the test set. Perhaps notably, however, the model appears to mainly focus on a single hour range, and does so to a lesser degree than with the test set examples. Of course, as time goes on, the forecasts may worsen. It is thus advisable to retrain the model every month or so in order to ensure that the model can adapt to recent changes. In particular, in the event of a large game patch or a new marketing strategy or campaign, the model should be monitored carefully. Expectedly, the model should mark more outliers than usual, but if the forecasts begin to deviate considerably, the model should be retrained.

3.5.4 Publication as a Shiny App

A subset of the analyses and reports published internally at Big Blue Bubble are built using R \cite{99} and exist on a Shiny \cite{100} server. Since the framework already exists, we migrate our model over to R and create a web interface so that anyone in the company can interact with the model. Luckily, there is a R library for Tensorflow which allows us to simply load in the model which we have already trained. We also load in the scaler parameters (Equation 3.1) from a file so that we do not need to recompute them in R.

The web interface is a Shiny application which allows for interactivity. We provide an hour slider so that the user can select an hour for which they wish to see the forecasts. We then query the Redshift database for that date and run it through the model. We plot the 24-hour forecasts for that date in six separate plots. For aesthetics, we represent the estimated 95% prediction intervals with a grey area.

Additionally, we show the one-hour forecast (with the corresponding prediction intervals) for each hour six days into the past. Doing this allows the user to see how the model has been doing recently. For clarity, we mark the user-specified date with a red, vertical line. We show a snapshot of the web page in Figure 3.11, omitting the scale on the y-axis. Additionally, we mark any points outside of the prediction interval (the anomalies) with a red cross.

3.6 Discussion

We show here that we can apply recurrent neural networks to the task of forecasting the hourly metrics in *My Singing Monsters*. We use two years’ worth of data to train our model in Tensorflow. The model produces respectable forecasts and appears to be capable of picking up on daily trends. It would not be surprising if the model also learned weekly trends, but it is less likely that the model can learn monthly or yearly trends as the data set is limited in this regard. Additionally, by adding an attention mechanism to the model, we are able to visualize the weights of this mechanism and use it to somewhat interpret how the model is arriving to its forecasts. Combined with the 95% prediction interval that the model also outputs, we can identify anomalous points and focus our manual attempts to identify truly important points. In order to help convey the functionality of the model, we also produce a Shiny application written in R that displays the future and past forecasts of any user-specified time.

Further steps for this project include testing other architectures of models or using more metrics to forecast the six presented here. We tried training a separate model for every hourly feature, each trained on the data for all six metrics. However, we did not see large improvement,
although we did not perform a thorough hyperparameter search for these models. We could also try varying the number of lookback or lookahead (forecasted) hours, or add autocorrelation information to the model, or even just encode the day of week information for the model to incorporate in its forecasts. Additionally, we when we sampled time series from the data, we used an overlapping sampling technique. That is, time series starting at two consecutive hours both exist in the data set. As such, there can be concerns of data leakage among the training, validation, and test sets. Although we demonstrate that model still works well with new data that is sampled from a time range not presented in any of the training, validation, and test sets, further work could include a block sampling approach to mitigate any risks of data leakage.

One could also further explore the attention maps to identify what are the salient metrics that are not included in the current model. One could also extract the internal vector representation of each input (that is, for each hour) after the attention mechanism and cluster them to see if any noticeable clusters appear such as clusters for days of the week or holidays (though the data set is limited for this since some holidays would only appear twice). Finally, one could also expand the functionality of the Shiny application to allow for easier exploration of the data, including displaying the attention maps on the page, adding hover text to the plots, or making it possible to vary the number of hours shown in the plots.
Figure 3.5: Forecasted hourly metrics from training set.
3.6. Discussion

Figure 3.6: Forecasted hourly metrics from test set.
Figure 3.7: Attention maps for examples in Figure 3.5.
Figure 3.8: Attention maps for examples in Figure 3.6
Figure 3.9: First example of forecasts on new data.
Figure 3.10: Second example of forecasts on new data.
Figure 3.11: Snapshot of the Shiny application for the hourly metrics model.
Chapter 4

Predicting Player Conversion in My Singing Monsters

4.1 Motivation

In this section, we apply recurrent neural networks (RNNs) to session data of players in My Singing Monsters (MSM) in order to predict whether a player will make a purchase at some point in their lifetime within the game. We term a player having made a purchase as being converted. We term a player having left the game prior to 30 days of when we query the data as being churned. Such a player is thus inactive and we assume that they will not come back. Otherwise, the player is active. Here, we will focus primarily on converted players, regardless of them being churned. However, we do acknowledge the significance of an active player who is not yet converted, as there is still a chance they will make a purchase. Thus, we develop a ML model that attempts to classify a player, given information about their first few game sessions, as being in one of these three classes: converted, not converted and churned, and not converted but active.

Being able to predict whether a player will make a purchase given only their interactions with the game is no easy task. Why a player will make a purchase depends on a wide variety of factors, including things that cannot be easily or possibly inferred from simply their in-game actions. Furthermore, the problem is exacerbated if we are only looking at their first few game sessions, as the player’s impression of the game and their willingness to spend can vary through time. For example, some players may not wish to make an in-app purchase until they have made significant progress in the game. For the first few sessions, their in-game behaviour may not deviate from that observed in players that will never spend. Despite this, we construct a ML model in the hopes that it will be able to capture some clues in the player session data.

Having these predictions is extremely helpful for various reasons. Firstly, the number of players who do end up spending compared to the overall population of players is quite small. Being able to correctly predict which players will spend allows us to better target these groups. For example, if we knew which players are likely to spend, we might consider targeting them with promotions which encourage them to spend more than they would have already. Alternatively, we might ignore this group and go after players we believe will churn and target them with promotions to encourage them to continue playing the game. Without this segmentation,
we are likely to waste resources by trying to keep players who were already likely to stay or by scaring away players through poorly-targeted purchase promotions.

As we saw in Chapter 3, we can also use attention in our model to identify which of the first few sessions were more important to making the prediction and use that to investigate player behaviours correlated with their spending. Another benefit of the model is that it takes temporal data (the player’s sessions) and encodes it into a feature vector internally. The advantages here are two-fold.

Firstly, common practice in business intelligence is to hand-craft or engineer features out of the data. For example, we might want to look at average spending per session, or number of daily sessions, or total number of monster breeds. While interpretable, this can cause a few problems. First of all, if we are able to look at the entirety of data, we are able to view the player through more lenses than simply our hand-crafted features. If we create a feature that is not very important or meaningful, or if we make an error when calculating the feature, then any model that tries to use that feature will suffer in its ability to learn. On the other hand, having access to all the data allows the model to learn what is important by itself and takes out some of the human bias in engineered features. Furthermore, such features take time to create and require further processing to aggregate. If we are able to use the data in as raw a form as possible, than we can cut away, to an extent, the middle man.

Secondly, as we will see, we can extract this feature vector and use it to project our players into a 2- or 3-dimensional space and thus cluster them based on these encodings. While the encodings are biased to help make the prediction as to whether a player will make a purchase, these clusterings may still provide additional granularity in identifying key player behaviours beyond predicting conversion.

4.2 Data Preparation

Within the game, many player actions trigger events which are recorded and stored in an Amazon S3 bucket. These events are triggered within a player’s session which is defined by when the player starts and stops playing the game. Events contain various fields that vary from event to event, but all events have the unique identifier of the player that triggered it (bbb_id), a timestamp of when it was triggered, and the name of the event. Raw event data is in JSON format. Events are stored as they arrive and are not aggregated by player, at least not immediately. Some commonly-queried events are collected on a nightly basis and put into a Redshift database for ease of querying. Still, in order to get a full view of the player’s history, we need to use both the raw data in the S3 bucket, as well as leverage data that is already aggregated in the database.

First, we identify a subset of the entire player base to look at since looking at the entire player base would be incredibly time-consuming and resource-intensive. Rather, we only consider a random sample of players who started playing MSM between October 1, 2017 and May 31, 2018, inclusive. Additionally, when considering the events of these players, we only consider events recorded on or before August 1, 2018. Next, we identify a few events than are not automatically inserted into Redshift and manually pull them out of S3 for these players. The

https://aws.amazon.com/s3/
specific features include those related to Tribal Island (see Figure A.6) and those related to the
earning and spending of various auxiliary currencies in the game such as keys and starpower.
We also pull out the active_island event which contains information about the amount of
these currencies earned and spent within each session.

Once these are in a Redshift table, we combine this data with the data for other events that
are already present in Redshift. In particular, we look at 25 different player events. In addition,
we can determine the amount of coins, food, keys, and shards earned and spent per session.
Finally, there is a session event from which we can determine the length of a player’s session,
the time between sessions, as well as which events belong with which session. Sessions are
demarcated by disconnects from the MSM server. When the server recognizes that the player
has disconnected from the game, whether it is from a logout, or a force close of the app, or
because of an unstable Internet connection, the session event is fired.

### 4.2.1 Session and Player Feature Vectors

One scheme to organize all this event data is to treat the events individually and use a hierarchi-
cal model to process events within a session, and then sessions within the player’s history. In
this way, the order of the events is kept and the model can learn something from it. However,
we argue that the order of events within a session is not necessarily important and will only add
complexity to the model. Instead, we aggregate the events into session feature vectors which
count up the number of occurrences of each of the 25 events within the session. Thus, a player
with 100 sessions will be represented as 100 feature vectors and the order of the sessions is
still kept, though the order of the events within a session is lost.

The feature vector for each session ends up being a 45-dimensional vector. The first 25
components are the counts for each of the events. The next 8 components represent the start
time of the session; the specific encoding for this is described in Subsection 4.2.2. The next 10
components are the values for coins, diamonds, food, keys, and shards earned and spent during
the session. The last two components represent the length of the session (in seconds) and the
number of minutes since the last session. For the very first session, the latter value is set to 0.

Finally, we also obtain a feature vector relating to a player. That is, a player is represented
in our model as a list of session vectors, plus a player vector. This vector is much simpler in
that it only contains information as to which platform they are using (iOS, Android, other) and
how they were acquired (organically or not). An organically-acquired player is defined as a
player not having been acquired via a user acquisition (UA) campaign. The two features are
one-hot encoded and stored in this vector. The idea here is to use this vector as an auxiliary
input to our model. One other option is that we can append this vector to each of the session
vectors and keep our model as having only one input. However, with minor testing, we found
that this approach actually performed worse.

### 4.2.2 Session Start Encoding

We wish to also include some information of when the session begins in each session feature
vector. For example, it may be useful to the model to know that the session happens on a
weekday, or that it happens in the summer, or that it happens at night. Of course, the model
would have no knowledge of what the time periods are, but any seasonality may be picked up
if we include this information. We take inspiration from the positional encoding in [30] and encode the timestamp of each session using a set of sinusoidal functions.

We want to encode the separate parts of the timestamp for the session event. Specifically, we want encode the month, day, weekday (i.e. Monday, Tuesday, etc.), and the hour. We could simply encode them numerically, having 1-12 represent the month, 1-31 represent the day, 0-6 represent the weekday, and 0-23 represent the hour (this is the scheme that the Python `datetime` library uses). However, in reality, these fields are cyclic and an encoding in which the hour that follows 23 is 0 is unintuitive. Ideally, we want a continuous function where we start at, say, 0, and increase until we reach 0 again. Of course, the intermediate value theorem says that such a continuous function cannot exist.

The solution is to use two functions: sin and cos. Alone, they are not sufficient, but by combining them, they provide a unique and smooth encoding for each of the features. In general, we encode each feature as a tuple (pair of values)

\[
\left( \sin \left( \frac{2\pi x}{d} \right), \cos \left( \frac{2\pi x}{d} \right) \right)
\]

where \(x\) is the value of the feature as an index and \(d\) is the number of possible values for that feature.

Let us consider a concrete example. Suppose we wish to encode “Friday”. Assume we start indexing with Monday = 0, Tuesday = 1, etc. so that Friday would have the index 4 and there is a total of 7 days per week. Thus, we would have \(x = 4\) and \(d = 7\) and our encoding of “Friday” is

\[
\left( \sin \left( \frac{2\pi \times 4}{7} \right), \cos \left( \frac{2\pi \times 4}{7} \right) \right) = \left( \sin \left( \frac{2\pi \times 4}{7} \right), \cos \left( \frac{2\pi \times 4}{7} \right) \right) 
\approx (-0.43388, -0.90097)
\]

We show the plots of these functions in Figure 4.1. Note that each integer value on the x-axis is assigned to a unique pair of values on the y-axis. This would not be the case if we only used one of the functions. Additionally, the values for the position immediately following 6, which would represent Sunday, are the same as those for 0, which represent Monday. This is due to the periodic property of the sinusoidal functions. Thus, we have an encoding that preserves cyclic information. We encode each part of the timestamp as enumerated above and obtain 8 values to identify the start time of each session.

We note that Vaswani et al. [30] mention they do not see much difference between positional encoding and simply using the index. We have not tested the difference on our model, but we believe that the former encoding contains a bit more information than the latter at the small cost of initial intuitiveness.

### 4.2.3 Player and Session Sampling

As stated previously, we are only looking at players who started playing within an eight-month period starting on October 1, 2017. Additionally, we only consider the events for these players.

---

[3]: https://docs.python.org/3/library/datetime.html
which were captured on or before August 1, 2018. Since we are trying to predict player conversion, we only include sessions up to, but not including, the player’s first purchase. That means that if a player makes a purchase in their first session, we do not even consider them at all. This is acceptable as predicting player behaviour before their first session is complete is generally infeasible and beyond the scope of this section. Rather, we are trying to infer behaviour from the first few sessions.

Next, we consider how many sessions we should look at before making a prediction, which we will call $N$ in this section. Ideally, we would want $N$ to be as small as possible so that we can make early, actionable predictions. If we choose a larger $N$, we might be able to learn more about our player. However, it may be too late for most players since many players who will churn do so within the first few days of playing the game. We arbitrarily choose $N = 10$. In our data set, it turns out that 56% of the players have at least 10 sessions. Of these players, 50% of them hit this mark within 3 days and 75% of them hit it within 8 days. However, it should be noted that we are also considering players that have less than 10 sessions. When we consider all players in our data set, 50% reach either their last session or their tenth session (whichever one comes first) within four days, while 63% reach this point after at most eight days. Thus, the data that we are using here consists of data that can be collected from the majority of the players within their first eight days of playing and so, $N = 10$ appears to be a reasonable choice. We may find that $N > 10$ makes better predictions, but the cost is requiring a longer period to wait before making a prediction on those $N$ sessions and we have no way of knowing which players will play less than $N$ sessions, though we know that it will likely be more than for $N = 10$ (certainly it cannot be less). In practice, we would wait either eight days or until 10 sessions have been logged before making a prediction for a player. However, we will also see how well the model performs when making predictions before either these events occur in Subsection 4.5.2. Since RNNs can accept variable-length input, one could also experiment with using the model on more than 10 sessions, though we do not explore that here.
4.2.4 Cost-Based Player Weighing

The full data set consists of 996005 players which we will divide into training and test sets using a 90-10 split. We then investigate the class proportions in our training set. 82% are churned and did not convert, 11% are still active but have yet to convert, and only 7% are converted. The test set follows a similar distribution. We can see that there is quite the class imbalance in our data set. As a result, a naive classifier might classify all players as being in the majority class and it would have an 82% prediction accuracy. Despite a high accuracy, the model will have nothing informative to offer us. In fact, if nothing is done, any classifier we try to train will likely lean towards this result.

There are a few options to counteract this. One option which we initially tried was to under-sample from the majority class. By randomly selecting players while keeping the distribution of class labels approximately uniform, we could make a classifier that would not be biased in selecting a majority class. However, we discovered that this did not generalize well. Our theory is that, by undersampling from the majority class, our model was unable to properly learn trends that would identify these players. Since undersampling results in the exclusion of most of the players in the majority class, it is possible that the subset of players selected from that class was not representative enough to allow the model to learn well. As a result, to a certain degree, we were overfitting on the minority classes. Another alternative is to oversample by generating representative, but “fake” examples from the minority classes. However, this is rather complicated for multivariate sequential data.

An easier alternative is adjust the loss function. The standard loss function we use here when performing a multi-class classification is the categorical cross-entropy loss, which is just a generalization of Equation 2.12 to the multi-class case:

$$L(y, \hat{y}) = \sum_{i=1}^{M} -y^{(i)} \cdot \log(\hat{y}^{(i)})$$ (4.2)

where $M$ is the total number of players in our training set, $y^{(i)}$ is a one-hot encoded class vector and $\hat{y}^{(i)}$ is the corresponding softmax output of the model. When implemented, the values in $\hat{y}^{(i)}$ are clipped to the range $[\varepsilon, 1 - \varepsilon]$ where $\varepsilon$ is some small value to avoid undefined or positive values when taking the logarithm.

This loss function treats all classes as equal, but we can apply a simple tweak to it to make it value correct predictions on the minority classes more. We first calculate a weight for each of the three classes in our data set that is inversely proportional to the size of that class. Denote $C$ as the set of all players in class $c$ and the weight for class $c$ as $\omega_c$. Then,

$$\omega_c = \frac{M}{\sum_{i=1}^{M} \mathbb{1}_C(y^{(i)})}$$ (4.3)

where $\mathbb{1}_C(y^{(i)})$ is the indicator function for class $C$ and is equal to 1 if the player represented by $y^{(i)}$ belongs to class $C$, and 0 otherwise. Now, if $c(y^{(i)})$ returns the class of $y^{(i)}$, we can redefine our loss function as

$$L(y, \hat{y}) = \sum_{i=1}^{M} -\omega_{c(y^{(i)})}(y^{(i)} \cdot \log(\hat{y}^{(i)}))$$ (4.4)
4.3. Model Architecture

Essentially, we have placed a weight on each class, with the majority class having a much smaller weight than the weights on the minority classes. In this way, we force the model to pay special attention to the minority classes. At the same time, we are still exposing the model to all players in these classes. Thus, we have overcome the weakness apparent in undersampling. Finally, note the loss function only affects the training of the model; once the model is trained, we do not have to worry about weighting new, unseen examples.

4.2.5 Feature Transformation

Some of the features in our session feature vector can range wildly in value, particularly the features related to currencies. Again, for nice convergence properties, we look to transform the data such that the values for each feature lie within a similar range. Rather than an involved procedure such as Equation 3.1, we simply apply a logarithmic transformation. Specifically, we transform feature $x_i$ to $x'_i$ using

$$x'_i = \log(x_i + 1) \quad (4.5a)$$

whose inverse is then

$$x_i = e^{x'_i} + 1 \quad (4.5b)$$

This transformation spreads out small values and compacts large differences. We transform all features in this manner, except for the positional encodings for the session start time, whose range is already within an acceptable range ($[-1, 1]$). We also do not apply normalization to the player feature vectors.

4.2.6 Zero-Padding

Finally, we must consider how to practically train our model. Theoretically, RNNs can be applied to sequences of varying lengths. However, with Tensorflow [97], that would require training on one sample at a time, which would take far too long. Our sequences are comprised of session feature vectors, for which we will have at most 10. For players with less than 10 sessions, we can fill in the “missing” feature vectors with zeros vectors. Another alternative is to use the Masking layer in Keras, but they are not compatible with the CuDNNGRU layers which we will use when training the model. A final alternative worth exploring is to group the players based on the number of valid sessions they have and to train the model in batches using these groups. Note that when we evaluate the model after it has been trained, we do not need to zero-pad the input vectors (so long as we are inputting one at a time). We see an application of this in Subsection 4.5.2.

4.3 Model Architecture

For our model architecture, we follow a similar model as presented in Section 3.4. Again, our pre-attention RNN uses GRUs [27] and is bi-directional. However, this time, we do not need to output a sequence but rather a single class label. Thus, we do not require a RNN following the
attention mechanism. Instead, we experiment with replacing it with various fully-connected layers instead. We also introduce the player feature vector at this point in the model. After these layers, we place a final softmax output layer which will output three numbers that can be interpreted as the probabilities of each class. When making a hard prediction on the class label, we simply pick the class with the highest probability.

We use the hyperopt package [86] which we used previously to optimize various hyper-parameters of the model around the validation loss. In our hyperparameter search, we test whether to use one, two, or three GRU layers and the number of hidden units of each layer, whether to use batch normalization [88, 89] or Gaussian dropout [90] after each of these recurrent layers, the activation function in the attention mechanism, the batch size when training, and the number (one or two), size, and activation functions of the fully-connected layers following the attention mechanism. The full list of parameters over which we optimized is presented in [Table 4.1]. We run the algorithm for 75 trials on a laptop with a NVIDIA GeForce GTX 1050 video card. As before, we replace the GRU cells in the model with the CuDNNGRU available in Tensorflow to speed up training. The hyperparameter choices that achieve the lowest validation loss are presented in [Table 4.2] and a diagram of the final model is shown in [Figure 4.2].

Table 4.1: Optimized hyperparameters for player conversion model.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden units in 1st GRU layer(s)</td>
<td>uniformly over [8, 256]</td>
</tr>
<tr>
<td>Use 2nd GRU layer?</td>
<td>True/False and if True, pick hidden units uniformly over [8, 256]</td>
</tr>
<tr>
<td>Use 3rd GRU layer?</td>
<td>True/False and if True, pick hidden units uniformly over [8, 256]</td>
</tr>
<tr>
<td>Batch Normalization (BN) after 1st GRU layer</td>
<td>True/False</td>
</tr>
<tr>
<td>Gaussian Dropout (GD) after 1st GRU layer</td>
<td>True/False and if True, pick rate uniformly over [0, 1]</td>
</tr>
<tr>
<td>BN after 2nd GRU layer</td>
<td>True/False</td>
</tr>
<tr>
<td>GD after 2nd GRU layer</td>
<td>True/False and if True, pick rate uniformly over [0, 1]</td>
</tr>
<tr>
<td>BN after 3rd GRU layer</td>
<td>True/False</td>
</tr>
<tr>
<td>GD after 3rd GRU layer</td>
<td>True/False and if True, pick rate uniformly over [0, 1]</td>
</tr>
<tr>
<td>Activation function in attention mechanism</td>
<td>“elu”, “hard_sigmoid”, “relu”, “selu”, “softplus”, “tanh”</td>
</tr>
<tr>
<td>Use 1st fully-connected (FC) layer after attention mechanism?</td>
<td>True/False and if True, pick hidden units uniformly over [8, 512]</td>
</tr>
<tr>
<td>Activation function for 1st post-attention FC layer</td>
<td>“elu”, “hard_sigmoid”, “relu”, “selu”, “softplus”, “tanh”</td>
</tr>
<tr>
<td>Use 2nd fully-connected (FC) layer after attention mechanism?</td>
<td>True/False and if True, pick hidden units uniformly over [8, 512]</td>
</tr>
<tr>
<td>Activation function for 2nd post-attention FC layer</td>
<td>“elu”, “hard_sigmoid”, “relu”, “selu”, “softplus”, “tanh”</td>
</tr>
<tr>
<td>Batch size for training</td>
<td>uniformly over [512, 1024]</td>
</tr>
</tbody>
</table>

Table 4.2: Top ten model architectures based on validation loss for predicting player conversion.

We note that in [Table 4.2], all but one model use only a single recurrent layer. That same model is the only model that does not use any batch normalization after its recurrent layers. On the other hand, all models here use dropout after at least one of their recurrent layers. The activation function chosen for the attention mechanism appear to heavily favour rectifiers. Most
models also choose to use exactly one fully-connected layer after the player feature vector is introduced to the model. The choice of the number of fully-connected layers, their size, and their activation function, in addition to the choice of batch size, appears to vary quite a bit among the top ten models.

4.4 Model Training

As before, we select the best architecture as identified by hyperas and build it in Tensorflow. The final model contains 236948 trainable parameters. Again, we train the model using a COCOB optimizer [98] for 32 epochs. This time, we use early stopping to stop training if the validation loss does not improve for five epochs. The best model trained for 14 epochs before early stopping and took only about ten minutes to train. We substitute GRU layers for the CuDNNGRU layers used in the hyperparameter search so that we can run the model on a machine without a GPU. The final training and validation losses are 2.0870 and 2.1423 respectively. The losses over training epochs are shown in Figure 4.3.
4.5 Model Evaluation

In order to evaluate our model, we first examine how well the model does on data which it has already seen. In particular, we combine the training and validation sets into a single set and run those players through our model. Of the three predicted probabilities that the model output, we pick the maximum as being the predicted class. We can then compare the predicted and true class labels in a confusion matrix, shown in Figure 4.4. A confusion matrix shows the predicted labels on the x-axis and the true labels on y-axis. For example, our model incorrectly predicts 10809 players as being active but not converted, when they had in fact already converted (and may or may not be churned). The confusion matrix uses a colour scale that is normalized across the rows, with higher values being shown in a darker red to aid in its interpretation. A perfect classifier’s confusion matrix would be diagonal. Our classifier has a clear diagonal, but does make quite a few mistakes. Clearly, it makes more mistakes in misclassifying players from the majority class simply because there are more of them. The normalized colour scales shows, however, that it is still able to correctly predict most of the players in this class.

4.5.1 Precision and Recall

To get a sense of these mistakes, it is useful to consider a few metrics. Accuracy is a commonly used metric but is problematic when applying it to an unbalanced data set. We might define our model’s accuracy as the number of correctly classified examples, divided by the total number of examples. Our data set is approximately 10-80-10 split across classes so a naive classifier that always predicts the majority class would get 80% accuracy. However, it is clear that despite such a high accuracy, our model does not provide any useful information to us. As such, it is better to consider a few other metrics and to use these on a class-by-class basis.

Precision is defined for a class as being the number of correctly-identified examples divided by the number of examples classified as that class. Perfect precision means that when a classifier predicts a class, it is always correct. Recall is defined for a class as being the number of correctly-identified examples divided by the true number of examples in that class. Perfect
recall means that within the subset of examples classified as being in a particular class, it is
certain to contain all true examples for that class. The $F_1$ score is then computed as a harmonic
mean between precision and recall. These values for our combined-training-and-validation set
are shown in Table 4.3, along with the number of examples, or support, for each class. The
average shown in the final row is weighted by the support for each class.

<table>
<thead>
<tr>
<th>Class</th>
<th>Precision</th>
<th>Recall</th>
<th>$F_1$ Score</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert</td>
<td>0.30</td>
<td>0.64</td>
<td>0.41</td>
<td>58114</td>
</tr>
<tr>
<td>no convert + churn</td>
<td>0.95</td>
<td>0.68</td>
<td>0.79</td>
<td>739179</td>
</tr>
<tr>
<td>no convert + active</td>
<td>0.29</td>
<td>0.71</td>
<td>0.41</td>
<td>99111</td>
</tr>
<tr>
<td>Average / Total</td>
<td>0.83</td>
<td>0.68</td>
<td>0.73</td>
<td>896404</td>
</tr>
</tbody>
</table>

Table 4.3: Precision, recall, and $F_1$ scores for the combined training and validation set.

We see that the precision in the minority classes is quite low, but is quite high for the
majority class. This is because the classifier is predicting a lot of players in majority class to
be in the minority classes. This is perhaps to be expected as there are simply so many such
players, but is certainly something that could be improved. Low precision pulls down the $F_1$
score for both minority classes. On the other hand, the recall for all three is reasonable. That
means the within each class, the model is able to correctly classify most of the players. In
particular, the recall scores in the minority classes are quite nice, all things being equal. Even
in the presence of an unbalanced data set, our model still manages to have a decent recall.
Another thing of note is that a lot of confusion lies in between the churned and active classes
within the no-convert class. It may be that discovering a more discriminating feature for these
two classes, or training with $N > 10$, would help our model perform better.

We now investigate the test set, which we had put aside initially. We plot the confusion ma-
trix in Figure 4.5 and show the precision, recall, and $F_1$ score metrics in Table 4.4. We observe
that the metrics here are more-or-less the same as we saw with the training and validation set. As such, we conclude that the model is able to generalize well.

Figure 4.5: Confusion matrix for the test set.

<table>
<thead>
<tr>
<th>Class</th>
<th>Precision</th>
<th>Recall</th>
<th>F₁ Score</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert</td>
<td>0.29</td>
<td>0.62</td>
<td>0.39</td>
<td>6320</td>
</tr>
<tr>
<td>no convert + churn</td>
<td>0.94</td>
<td>0.68</td>
<td>0.79</td>
<td>82092</td>
</tr>
<tr>
<td>no convert + active</td>
<td>0.29</td>
<td>0.70</td>
<td>0.41</td>
<td>11189</td>
</tr>
<tr>
<td>Average / Total</td>
<td>0.83</td>
<td>0.68</td>
<td>0.72</td>
<td>99601</td>
</tr>
</tbody>
</table>

Table 4.4: Precision, recall, and $F₁$ scores for the test set.

Low precision in the minority classes means that we cannot pick a player from these classes at random and correctly classify them. High overall precision means that we can pick any player at random and correctly classify them. However, there is a good chance that this player will belong to the majority class and the model recognizes this.

With good recall rate across all three classes, if we wanted to isolate any class from the whole set of players, we can accomplish that quite well with this model. As an example, suppose we wish to treat the two minority classes as one class and we wish to target these players specifically. Assume we are working on the test set. If we take all players predicted to be in those classes, we would end up with a set of 40194 players. In that set, 64.9% of the players are not players we wish to target due to our poor precision. On the other hand, 80.7% of the players who we do want to target are in that set. Consider the alternative of blindly targeting the entire test set. Now, in this set of 99601 players, 82% are those who we do not wish to target. This model allows us to reduce the number of players we have to target by more than half while keeping recall of desired players high.
4.5.2 Class Probabilities Over Time

In practice, we do not know how many sessions a player will have, nor do we know when they will have them. Our model uses a player’s first 10 sessions, but that tenth session may come well after they first create their account. Even so, we would like to make predictions on what we have on the player so far. To explore whether this is practical, we analyze how the model predicts over time. Specifically, we take a random player and feed them through the model ten different times, each time adding another one of their sessions to the input. So, for the first time, we take a player’s first session vector and send that through the model. Note that we do not pad the sequences so that they are ten vectors long. Then, we take the player’s first two session vectors and feed those through the model, and so on. In Figure 4.6, we choose four players from the test set who have at least ten sessions and plot the prediction probabilities as we add sessions, as well as the attention weights on each of the session. For the attention maps, we use the same colour scale as we did for the confusion matrices. Again, these maps are normalized across their rows.

It is important to note that with each prediction comes uncertainty. However, we have not designed the model to output uncertainties with each prediction. Although the model may report a high probability for a player belonging to a certain class, there is also an implicit amount of uncertainty that the model does not report. As such, just because the model reports a high probability at one time step does not mean that this probability can not decrease over subsequent time steps. Likely, the uncertainty at an earlier time step will be larger than the uncertainty at later time step. Having the model explicitly report these uncertainties and visualizing them is an area for future work.

In Figure 4.6a, we show an example where the classifier correctly identifies an unconverted but active player. We see that the model appears to disregard the first two sessions after it has seen the third. On the plot to the left, we see that the first two sessions correspond to where the model incorrectly predicts that the player will convert. We also see that the model mostly focuses on the most recent session, though other session appear to be important for limited amounts of time. Key sessions to look at here might include the third, fourth, and ninth sessions.

In Figure 4.6b, we see an example where the model is about 50% certain that the player will convert without churning. Interestingly, it is only after the about tenth session that the model reaches this conclusion; the model predicts that the player will remain active without converting for most of the sessions. However, we do see that the confidence in the correct prediction does increase over the course of the ten sessions. From the attention map, we again see that the model mostly focuses on the most recent session, though other session appear to be important for limited amounts of time. Key sessions to look at might include the third, fourth, sixth, eighth, and ninth sessions.

Figure 4.6c shows an example where the model is quite certain in its prediction, but where it is actually incorrect. The model predicts that the player will convert, even though they actually churn without converting. We see that the model is fairly certain in its prediction throughout the first ten sessions. While its confidence does decrease over the first five sessions, it never falls below 60%. The attention map here is also quite simple; for the most part, the model only focuses on the current session.

Finally, in Figure 4.6d, we show a situation where the model ended up fairly split among the three classes and gets the prediction wrong. The model appears to focus on the three most
(a) A very certain and correct prediction.

(b) An uncertain but correct prediction.

(c) A certain but incorrect prediction.

(d) An uncertain and incorrect prediction.

Figure 4.6: Various predictions and their respective attention maps.

recent sessions, with emphasis on the more recent sessions. For the final session, we see quite a large spread of attention over several sessions. This likely reflects the low confidence of the model as it appears that it is unsure of which sessions it should be focusing. From the attention
map, it is unclear if there are any specific key sessions which contribute to the final prediction.

When looking at the attention maps, we see that most of the emphasis is on the most recent
session the model sees. In some cases, the model does appear to identify a set of sessions that
it believes to be more important and others which it thinks are less important.

We showed here examples for players which have at least ten sessions, but when looking at
examples where we do pad the session feature vectors, we find that the model does put some
weight on the zero-padding vectors, though these values appear to be negligible. However,
this may not be the case for all examples and we would ideally not like the model to put any
attention on these vectors. The reason is that it allows the model to “learn” about how the lack
of a session contributes to the prediction. In reality, however, we have no idea about whether
or not another session will come or not. We want the model to learn based solely on what it
is given, rather than what it is not given. As a result, exploring a model which trains without
using zero-padded vectors is an interesting avenue for research in the future.

4.6 Clustering

Finally, we look at how we can cluster the players in our test set. If we are able to identify
meaningful clusters, it may add further information that can be used on top of the predictions
produced by the classifier. In order to cluster the players, we first need to reduce the sequential
session vectors and the player feature vectors into a single vector that can be used in clus-
tering. Rather than manually doing this ourselves, we leverage the internal representation, or
encoding, of the player in our ML model. Before outputting, our model makes use of a fully-
connected layer. The output of this layer is a single vector, which is exactly what we want. As
such, we will use the encoding produced by the final fully-connected layer in the model as our
representation of the player.

This vector itself is not interpretable as it is a 380-dimensional vector whose components
are complex combinations of the initial player features. However, we can assume that it con-
tains some information about the player. In particular, the information should be biased towards
making a good prediction as to which class the player belongs. As we saw, however, the model
is not the most accurate classifier. As such, we would expect that the clusterings that will be
 afforded by this encoding will both be likely clustered based on the player class, and that the
clustering will be extremely noisy. As explained in Section 2.6 the idea of noise is not inher-
et to all clustering algorithms so we will explore using both the HDBSCAN* and k-means
clustering algorithms.

4.6.1 Applying HDBSCAN*

To begin, we take a random sample of 25000 players from the test set. This is done for com-
putational efficiency when running the various algorithms necessary for clustering, as well as
being able to render the visualization of the clusterings quickly. As 25000 is still a large sam-
ple, the distribution of the player classes is very similar to that of the test set. We then extract
the encodings as described above for these players. Since we are unable to visualize in 380
dimensions, we use the popular t-SNE algorithm [101] to bring the encodings down to both
two and three dimensions. The particular implementation of the algorithm we use is provided by Ulyanov [102].

We then use HDBSCAN* [81], as implemented by McInnes and Healy [103], and show the clusterings in Figure 4.7 We leave all parameters to their default settings. Note that any points labeled as noise are shown as grey crosses. Interestingly, only 214 players are labeled as noise. Additionally, though it is extremely hard to discern, the saturation of the colour on each point scales with how probable the point belongs to that cluster. We see that the algorithm identifies four clusters. Two of the clusters are very large, while the other two are relatively small.

![Figure 4.7: Visualizations of the clustering produced by HDBSCAN*.

4.6.2 Cluster Hierarchy

As the HDBSCAN* algorithm iterates, it constantly split apart clusters to create new clusters. Once all clusters have reached a stopping condition, the algorithm then selects which splits will become the output clusters of the algorithm. We can thus visualize this hierarchy of clusters by using a dendrogram, as shown in Figure 4.8. The four main clusters that were selected are circled and in their corresponding colour. We see that cluster 1 splits off first, followed by cluster 2. Then, clusters 3 and 4 are formed by splitting the remaining cluster. We can see that the algorithm actually continues to split even after these clusters are formed, but they are not chosen as stand-alone clusters and are instead merged back with the main cluster.

4.6.3 Cluster Analysis

We can then go into each cluster and investigate the players within each cluster to try and understand how they have been grouped. We have put the plots used to describe each cluster in Appendix B. For each cluster, we first show the distribution of the classes as well as the distribution of both of the categorical features in the player feature vector. We see that cluster
4.6. Clustering

Figure 4.8: HDBSCAN\textsuperscript{*} cluster hierarchy.

2, the smallest cluster, contains mostly players who have not converted but are still active. Similarly, cluster 1 contains mostly players who have converted while cluster 4 contains mostly players who have churned and did not convert. Finally, cluster 3 follows a similar distribution to the actual distribution in the test set. We see from the Acquisition plot that all clusters contain some number of both organic and non-organic players, with there being much more organic players. We also see that the distribution of platforms is also very similar across clusters, with there being a slightly larger ratio of the “other” platform in cluster 2 compared to the other clusters. This might hint that the auxiliary player vectors actually do not contribute much to the overall clustering, and thus to the conversion prediction model.

We also average the event counts for all players and all sessions for each cluster. In particular, we average each event count for each session by the number of valid players for that session. This is necessary as we may have a varying number of valid sessions per player in each cluster. We do not include the zero-padding vectors as they would artificially drag down the averages. We then normalize the averaged event counts across the sessions. That is, for each session in the final plot, the sum of all the averaged event counts for a single session will be one. We then use a colour map to show this matrix of event counts. From this, we can see that clusters 1 and 4 do not have any players with a tenth session.

Looking at averaged event counts, we see some trends we would naturally expect. For example, we would expect that the counts for the \texttt{diamonds\_spend\_speedup} event decrease over sessions as the players will likely have less diamonds to spend as they use them up. This is not quite the case for cluster 1 which contains mostly converted players. On the other hand, we would expect the number of \texttt{speedup\_with\_video} events to naturally increase over time as players will begin to interact with the core mechanic of the game and will have run out of diamonds to spend on the speedups. Again, cluster 1 does not appear to quite follow this trend. We note that the variance of the different types of events is quite small in clusters 3 and 4 which have mostly players that will churn without converting. Meanwhile, cluster 1 and 2 have a wider variance, correlating the ideas that players who interact with multiple elements of the game are likely more interested with the game and more likely to convert or stay in the long run.
Event maps for clusters 3 and 4 are generally quite similar, though we do see a larger occurrence of the `speedup_with_video` event in cluster 3. Clusters 1 and 2 are also quite similar. Again, these clusters contain mostly users who are either converted or who are still active. We see that cluster 2 does appear to have a larger spread of different events. We see that there appears to be more players spending diamonds on the spin wheel mini-game, which has a chance to reward the player with significant currency prizes, in cluster 2. We also note increased activity in the `island_unlock` and `monster_sell` event. Monster sell events are notable in particular as they are likely a result of a failed breeding; that is, the player unsuccessfully bred two monsters in an attempt to produce a rarer monster. This shows that the player understands the core mechanic of the game and likely correlates to their engagement and results in their retention. The increased presence of the `speedup_with_video` also speaks to this.

We then plot the session length against the minutes since the last session for each cluster. Note that each y-axis in this plot uses a different scale. Session lengths are recorded in seconds. In general, we see, and would expect, that the average session length decreases over time, while the average number of minutes between sessions increases. Generally speaking, session times are shorter and time between sessions is longer in clusters 2 and 4. Unusually, cluster 2, which contains mostly players who remain active without converting, has the highest average time between sessions. This might simply be due to the fact that players who have ten sessions and have such large time deltas between session are likely committed to the game (as why would they return after such a long time otherwise?). This is probably a large clue for the model to predict players in this class, rather than an overall observation about the behaviour of such players. With that said, a notable steady increase in the average minutes between sessions can be found in clusters 3 and 4, while being less present in clusters 1 and 2.

Finally, we plot the average number of earned and spent currencies. We can likely neglect the shards and keys currencies since the player is unlikely to interact with these in the first few sessions. The declining trend for diamonds appear consistent throughout the clusters. This makes sense as ways of consistently earning diamonds do not appear until later in the game. Food spending and earning also appear to be quite similar among clusters. However, it should be noted that the ratio of earn-to-spend is much tighter for cluster 1. Finally, an interesting observation is that the amount of coins earned/spent in cluster 1 is lower than the other cluster. The ratio here is also much tighter for clusters 1 and 2 when compared to that of clusters 3 and 4. This may mean that the players who churn do earn coins, they do not seem to spend them. This could be for a wide variety of reasons and it would be hard to pin down the exact reasoning here. However, it does hint at something of which to potentially be aware.

As a final note, we acknowledge that taking the average of the players in the cluster may not be the most correct approach. Another approach might be to take the median of the features. As can be seen in Figure 2.5c, these approaches may not properly distinguish between clusters. Another approach might be pick out individual players to look at from the clusters. However, with the sheer amount of players in our data set, that might not be a desirable option. The best thing to do might be to look at the distribution of the values in the data set. However, we were unable to come up with a clean, compact way to visualize this.
4.6.4 Soft Clustering

Thanks to the implementation by McInnes and Healy [103], we can also produce a soft clustering by assigning all points to a cluster depending on cluster probabilities calculated by the algorithm, thus assigning noisy points to actual clusters. We show these soft clusterings in Figure 4.9. One can identify that a few points appear to have been incorrectly clustered, though it is important to remember that this visualization is only a projection from a very high-dimensional space. However, we can assume that the clustering algorithm is confused as to how to classify certain points.

![Soft clustering visualizations](image)

(a) 2D HDBSCAN* soft clustering visualization. (b) 3D HDBSCAN* soft clustering visualization.

Figure 4.9: Visualizations of the soft clustering produced by HDBSCAN*.

4.6.5 Applying k-means

Finally, we apply the popular k-means algorithm [78] to cluster the players. Since the algorithm requires the number of clusters to be specified beforehand, we choose to look for four clusters, which will allow us to compare k-means to HDBSCAN*. We show the clustering plot in Figure 4.10 and the graphical analyses of the clusters in Appendix B.

We see that k-means produces a very similar clustering to our soft clustering with HDBSCAN*, though this is not true in the general case. In general, the clusterings by both algorithms appear to cluster mainly by class. This goes along with the idea that the initial encoding should be biased towards being usable to make a prediction on conversion. Additionally, we see that there are there are two large clusters which appear to align with the majority class. In both algorithms, the largest cluster follows the distribution of the test set closely. It is possible that this is a group of players for whom the model is uncertain which class they belong, resulting in the poor precision of the model.
4.7 Discussion

We show here how recurrent neural networks can be applied to predict player conversion in *My Singing Monsters*. We first identify players who began playing in the inclusive date range of October 1, 2017 and May 31, 2018. We then collect the raw event data for these players up to and including event data on August 1, 2018. We then group these events into game sessions. These are then fed into our RNN classifier, along with some auxiliary data about the player themselves. The output of the classifier are three probabilities, each identifying the player as belonging to one of three classes: converted, not converted and churned, and not converted but active. Despite an unbalanced distribution of these classes, our trained model achieves a reasonable recall. However, we note that it also has poor precision for the minority classes.

We then investigate how the model performs as we feed it sessions one at a time, mimicking how it might be used in practice. We also observe how the model distributes its attention across sessions. We note that the model generally focuses on the most recent session which it is given, although it does exhibit the capability to focus on a particular session if it chooses to do so. We also explore here how the probabilities of each class change over time and note the model’s transparency can be improved by having it report uncertainties, which are implicit in its predictions, explicitly. Finally, we look at how we can cluster the players by taking the model’s internal encoding of the player. We use two different clustering algorithms and observe that the clusters produced appear to align most with the class of the player. This makes sense as the model is inherently tuned to make predictions about this class. We find that in the clusters containing players who either have converted or are still active, the average player will log a larger variety of events than players who eventually churn. We also observe that players who do churn typically spend less coins than players who do not, or those who churn after converting. This may hint that increased awareness of how to spend coins early on may lead to a lower churn rate.

Further steps for this line of research include attempts to improve on the precision of the
model. Avenues for exploration include changing the architecture of the model, changing the loss function, trying other techniques for tackling the unbalanced data set, or using a different way of organizing the event data. Additionally, one could try to cut down on the number of features fed into the model. We see that certain events, for example, do not appear frequently and can likely be ignored without much consequence. Doing so might allow a model to train faster or identify important trends more easily. One could also try extending the number of sessions to look at. We would assume that looking at more data for each players would allow the model to identify longer-term trends which may be more indicative of the player’s class. However, one should keep in mind how applicable such a model is in practice. It is also important to test the model on data for players who began playing after May 31, 2018 and observe how well the model generalizes to the most current data.

Finally, in order to cluster the players without bias, one could explore other models to encode the player. Alternatively, one could try different parameters on both the clustering and the dimensionality-reduction algorithms than the ones used here. Another option is to use different algorithms than the ones applied here; we explore such a technique in Chapter 6. To identify how the players are clustered, we also recommend investigating various techniques for representing a player. If a good encoding for a player can be produced, we recommend training a standard, non-recurrent neural network for the purposes of predicting conversion and comparing it to the model presented here.
Chapter 5

Predicting Lifetime Customer Value in My Singing Monsters

5.1 Motivation

Lifetime value (LTV) is one measure to determine player value. In our case, we want to investigate the LTV for a particular player, or for a subset of players, in My Singing Monsters. The calculation of LTV is simply the sum of the three sources of revenue from the players: purchases, advertisements, and in-game offers. At any given date, we can calculate the LTV by summing these three daily values and then performing a cumulative sum of these sums over the player’s entire lifetime. Thus, LTV is an inherently cumulative value. When predicting LTV, we might consider player lifetimes of vastly differing lengths and attempt to produce a “in-the-long-run” number for LTV. However, we might also want to consider the LTV for players after a particular length of time.

Here, we consider day 120 (D120) LTV; that is, the LTV for a player after 120 days of playing. We try to predict this value by only looking at the first 14 days of that player’s lifetime. Thus, we are trying to produce an early prediction of that player’s lifetime value. If we can successfully identify high-value players, so called “whales”, in their infancy, we can both (a) try to understand the behaviours that tend to appear among whales, and (b) monitor and promote revenue from these players. While whales make up a small minority of the player base, they contribute to a large portion of the game’s revenue. As such, a model which can predict whales can be extremely useful. For example, the model can be used to evaluate user acquisition (UA) campaigns. Such campaigns are not cheap and we would ideally want a large number of high-value players to be acquired in the campaign. If, after only 14 days, we can predict the long-term success of the campaign, we can stop unfruitful campaigns early while putting more resources into productive ones.

5.2 Data Preparation

Since we want to predict D120 LTV, we can only use players who have played for at least 120 days. As such, we set our cutoff to players whose creation date is between January and April 2018. Thus, the latest day 120 in our data set would be on July 29, 2018 (March 31 plus 120
5.3. Training Multiple Models

Additionally, we limit our data set to a random sample of players from the United States. We set the lower bound in the date range so that we can quickly evaluate our results; a model to be used in production should probably be trained on the full set of players for which we have revenue data.

We also need to consider players who have reinstalled the game after uninstalling at one point or another. This is because we run campaigns where players who had previously left the game for various reasons come back to the game. These players are essentially starting fresh. However, the game might associate them to the same bbb_id that they had before leaving the game. As such, we cannot simply use the creation date, but must also compare it to a reinstall date, if it exists. This would give us a more meaningful LTV calculation. We perform this check and ignore any players who were reacquired on or after April 1, 2018. Note there will naturally be aberrant or unexpected data. There may be cases where the date recorded is incorrect, or the recorded data for that player is incorrect. There are also cases where the player installs on multiple devices and it messes with our creation dates. Overall, however, these cases are few and far between and we hope that the model can deal with whatever noise they may bring. At the very least, it is a problem with which the models that are already put in place must handle.

We run a query to our Redshift database to get the appropriate creation date for the players, as well as to pull out the daily features we want to use as input to the model. We calculate the three daily revenue features which sum up to the player’s LTV. Again, this is the daily revenue (in cents) from in-app purchases, advertisements, and offers. Additionally, we get the daily number of purchases, sessions, and level ups. We also pull the daily total number of seconds spent in-game and the daily number of diamonds spent. Finally, we perform a cumulative sum on the three revenue features to get a cumulative LTV feature. Thus, we use a total of 10 features. Finally, we also look at the LTV on the player’s 120th day, which gives us our target value. We want to use only the features for the player’s first 14 days. As such, we clip the data set so that we only have the data for the first 14 days after the player is created or reinstalls (the maximum of the two). In total, we look at 860065 players.

As in Chapter 4, we apply a log transform on both the input data and the target LTVs. We then split the players into training, validation, and test sets using an approximately 80-10-10 split. We have 696652 players in the training set, 77406 players in the validation set, and 86007 players in the test set.

5.3 Training Multiple Models

Here, we explore the idea that training on more than 14 days’ data will benefit the inference on just 14 days. Specifically, we test training models on 14, 30, and 60 days and see what effect it has when evaluating on just 14 days. The hypothesis would be that letting the model see how the first 14 days might affect the first 30 or 60 days could give it some insight when evaluating on just 14 days. In earlier experiments, we tried to optimize hyperparameters for each of the three models separately. However, this is an expensive procedure in terms of time and computing resources. We found that the models generally had a similar architecture and the loss was not significantly different for the best models. As such, we will use the same architecture for all three models.

The models use a single bi-directional recurrent GRU layer with 256 hidden units...
followed by the same attention mechanism used in Chapter 4. We then use two fully-connected layers both having 128 units, with the first using a linear activation function and the second using the ReLU activation [54]. The output layer of the model is the log-transformed predicted D120 LTV. We show a diagram of the model in Figure 5.1. Recall that we train three separate models on 14, 30, and 60 days as input. However, when evaluating the model, we test on only 14 days. As a result, the input of these models can be of variable length. We train these models using the COCOB optimizer [98] and use the mean-squared error loss function. We train our models in a Google Colaboratory notebook which is equipped with a Tesla K80 GPU. Note that we will train more models in this chapter, all of which are trained in a similar manner.

![Diagram of our D120 LTV prediction model.](image)

We build the models using Tensorflow [97] and train for exactly 16 epochs, using a batch size of 512 for all models. We plot the training and validation losses for all three models in Figure 5.2. We see that the overall loss improves as we increase the size of the input. This makes sense as if we know the D60 LTV, we will likely have a pretty decent approximation of the D120 LTV. This brings up an issue with using too much data as input. If we gave the model up to the day 120 data, the model only needs to return the D120 cumulative LTV feature to achieve 100% prediction accuracy. As a result, when only fed 14 days, the model would only return the D14 LTV, which would likely be quite wrong. As such, there should be a point where we get diminishing or negative returns when using more data. However, we will use a model that simply predicts the D14 LTV as the D120 LTV as our baseline. From the losses, we see that the first epoch loss is quite high since the training loss is calculated as an average over all batches, and early batches are likely to have very high loss. After that, the loss basically plateaus after the second epoch. This could be a sign that our input or our model needs improvement. The loss here is not very interpretable as it (a) corresponds to the log-transformed LTV and (b) we want to see how well the model performs with only the first
There are a few ways we could approach evaluating the performance of the model when moving to the test set. We show these metrics for the entire test set in Table 5.1. First, we consider the sum of D120 LTVs for all test set players and compare it to the sum of the predicted D120 LTVs. For confidentiality reasons, we show the absolute difference between these values rather than their actual values in the first column. Next, we consider the mean absolute error (MAE) and the root-mean-square error (RMSE) for the LTVs across test set players. We perform this comparison after applying the inverse scaling transform on the model outputs. We also show these metrics for our baseline model which simply outputs the D14 LTV as its prediction. We see that, for the sum difference of D120 LTVs, the 14-day model comes the closest while our naive baseline model is the most incorrect. We embolden the lowest sum difference, MAE, and RMSE values. When comparing MAE and RMSE, we find that the 14-day also performs the best. The other two trained models are still better than the baseline, but the 30-day model actually has the worst RMSE between the two.

We can make a quick improvement here before exploring the metrics further. Our model has no understanding of what exactly is an LTV. We know that it is a dollar value and that it cannot be negative. Additionally, we know that it cannot decrease over time. As such, the D120 LTV must be at least the D14 LTV which the model sees. However, we do get some
instances where the model does predict below the cumulative LTV. Rather than making our model more complicated, we can simply perform a manual check with the predictions to make sure that they are at least the D14 LTV. We update the predictions and the evaluation metrics in Table 5.2. All three models benefit from this check, but the 14-day model still has the lowest sum difference and the best RMSE. However, now the 30-day model has the best MAE and the 60-day model has a better RMSE than the 30-day model.

<table>
<thead>
<tr>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>14-Day Model</td>
<td>$14,873.95</td>
<td>$0.38</td>
</tr>
<tr>
<td>30-Day Model</td>
<td>$25,521.56</td>
<td>$0.39</td>
</tr>
<tr>
<td>60-Day Model</td>
<td>$19,914.67</td>
<td>$0.39</td>
</tr>
<tr>
<td>Baseline</td>
<td>$34,334.52</td>
<td>$0.40</td>
</tr>
</tbody>
</table>

Table 5.1: Evaluation of LTV model on the test set.

<table>
<thead>
<tr>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>14-Day Model</td>
<td>$14,457.16</td>
<td>$0.38</td>
</tr>
<tr>
<td>30-Day Model</td>
<td>$22,887.80</td>
<td>$0.36</td>
</tr>
<tr>
<td>60-Day Model</td>
<td>$18,860.45</td>
<td>$0.37</td>
</tr>
<tr>
<td>Baseline</td>
<td>$34,334.52</td>
<td>$0.40</td>
</tr>
</tbody>
</table>

Table 5.2: Evaluation of LTV model, but with a manual check to set predictions to at least the D14 LTV.

5.3.1 Evaluation on Different Value Cohorts

For basically any game with in-app purchases, there will always exist a small minority of players who make up a majority of the revenue coming from these purchases. These high-value players, or “whales”, are what allow games following the freemium model to stay afloat. For the vast majority of players in *My Singing Monsters*, the D120 LTV will be low because they will not have made any purchases and the only revenue comes from ads or offers. Predicting the LTV of the average players is relatively simple compared to predicting the LTV of whales. The first problem comes from even identifying a whale. Some whales will spend a lot in the first 14 days, but others will not make a purchase until much later into the game. For these players, there may be no indication that a large purchase will come down the line and any model is essentially forced to predict much lower than the actual D120 LTV. Since our loss function minimizes squared error (on log-transformed target values) averaged across the entire data set, our model will likely heavily under-predict for whales, particular if they have minimal activity during the first 14 days. To see this, we evaluate our three trained models and consider the baseline models for both whales and the rest of player base, who we will refer to as non-whales.

We split our test set into a whales set and a non-whales set. We strictly define a whale, for our purposes, as a player whose D120 LTV is at least $10. There are 1281 such players in the test set, making up a mere 1.49% of the test set players. We run the two sets through the three trained models and show the results in Table 5.3 and Table 5.4. Note that we again ensure that the predictions are at least the D14 LTV. We also show the baseline model’s performance here.

For whales, all the trained models outperform the baseline. The MAE and RMSE for the 14-day model are the best, and the sum difference is very close to the best. Note that the MAE and RMSE are significantly larger than the average values reported in Table 5.2. This is due in part to the high variance of D120 LTVs among whales. Additionally, the presence of whales
5.4 Adding a Whale Classifier to the Model

We saw that our predictions for non-whales were quite good, but the performance on whales left a bit to be desired. A natural extension to our model would be to train separate models for whales and non-whales. In particular, the whale model could tune itself to do better on the whales than our 14-day model as it would not have to worry about handling the non-whales. However, at evaluation time, it is hard to tell whether a player is a whale or not based off only the first 14 days. Of course, if a player has made at least $10 for the company, they are trivially a whale by our definition. However, not all players will be so discernible. As such, we also would need to introduce a third model to classify players between the two classes. Then, based off its classification, we would then send the player to the appropriate model to predict their LTV. The three models will share a similar architecture to our 14-day model. Again, we use a bi-directional GRU to process the first 14 days. Then, we feed this to our attention mechanism and follow that with two fully-connected layers before the output layer. For the LTV prediction model, the output is the same, but the classifier will instead use a sigmoid activation function to output the probability that the player is a whale.

Additionally, because of this class imbalance, we also use weighing as we did Subsection 4.2.4. Simply put, whales will be assigned a heavier weight in order to balance out the class.

<table>
<thead>
<tr>
<th></th>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>14-Day Model</td>
<td>$14,290.27</td>
<td>$15.65</td>
<td>$43.15</td>
</tr>
<tr>
<td>30-Day Model</td>
<td>$16,563.83</td>
<td>$16.05</td>
<td>$44.37</td>
</tr>
<tr>
<td>60-Day Model</td>
<td>$14,142.22</td>
<td>$16.70</td>
<td>$44.12</td>
</tr>
<tr>
<td>Baseline</td>
<td>$22,815.16</td>
<td>$17.81</td>
<td>$48.60</td>
</tr>
</tbody>
</table>

Table 5.3: Evaluation of LTV model on whales in test set.

<table>
<thead>
<tr>
<th></th>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>14-Day Model</td>
<td>$166.90</td>
<td>$0.15</td>
<td>$0.58</td>
</tr>
<tr>
<td>30-Day Model</td>
<td>$6,323.96</td>
<td>$0.12</td>
<td>$0.56</td>
</tr>
<tr>
<td>60-Day Model</td>
<td>$4,718.23</td>
<td>$0.13</td>
<td>$0.58</td>
</tr>
<tr>
<td>Baseline</td>
<td>$11,519.36</td>
<td>$0.14</td>
<td>$0.63</td>
</tr>
</tbody>
</table>

Table 5.4: Evaluation of LTV model on non-whales in test set.

who do not really do anything aberrant in the first 14 days will greatly increase the loss. For example, a player that appears to be trending towards a low D120 LTV such as $0.10 might make a large ($99.99) purchase at day 111. The model likely has no way of predicting this and the RMSE for this example will heavily bump up the average among whales. Unless a more indicative feature for predicting this is identified, it is hard to expect much better. Such a feature may not even exist.

For non-whales, we see that the MAE and RMSE is actually best on the 30-day model. However, the sum difference is also the worst on this model, ignoring the baseline. In a sense, this model best optimized for our loss function. However, we also see that the 14-day model has a remarkably low sum difference with a similar MAE and RMSE as the 30-day. We would ideally prefer a model that balances performance on all three metrics. Additionally, we want good performance on whales in particular since they are such an important set of players. Since the 14-day model performed well on nearly every metric we have looked at, we conclude that there is no significant benefit to training on more days.
sizes when computing the value of the loss function. We use the `compute_class_weight` function in the `scikit-learn` package to quickly compute these weights. The calculation for these weights is similar to the one described in Subsection 4.2.4. We train the classifier on the training set and use a batch size of 512. When deciding the threshold for classifying a player as a whale, we found that using 0.3 seemed to work well. In other words, a player is classified as a whale if the output of the classifier is greater than 0.3, or if the model is at least 30% confident that they are a whale. As the model is updated, this threshold should be revisited. We show the confusion matrix for the classifier on the test set in Figure 5.3. We see that the performance on non-whales is great, as is the precision on whales. However, the recall for whales is a bit lacking. This means that about 35% whales will go to the regular-valued classifier. To some extent, this is basically what happens in our current 14-day model.

![Figure 5.3: Confusion matrix for the whale classifier trained on training set.](image)

Next, we train the LTV predictors for whales. For the whales, we find that letting the model over-predict rather than under-predict makes the final results better. This is due to the fact that the variance of LTVs among whales can vary greatly and having this over-prediction on individual users helps balance things out across a cohort. Additionally, since a significant number of whales will go to the non-whale predictor, the whale model must overcompensate as the prediction on these misclassified players will likely be quite a bit lower than the true D120 LTV. To do this, we use a custom loss function, as suggested by Stack Exchange user Emre:

$$\mathcal{L}(y, \hat{y}) = (\hat{y} - y)^2 (\text{sgn}(\hat{y} - y) + \alpha)^2$$ (5.1)

where $-1 < \alpha < 1$ controls how much the function should penalize overestimation. For our purposes, we found that $\alpha = -0.3$ works well, though this parameter should be reviewed with future modifications to the model. We train the model only on the whales in the training set and use a batch size of 128 because of the reduced size of this data set.

[1] https://tinyurl.com/compute-class-weight
Finally, our non-whale LTV predictor is trained on the entire training set, not just the non-whale players. Since our classifier does not have great recall on whales, quite a few whales will end up going through this classifier. As a result, we find that the model performs better if we train it on all the training players, as this way, the model has a chance of doing well on these misclassified whales. At the same time, we get to enjoy the good performance on the non-whale players that we saw previously in Table 5.4. In other words, we simply re-purpose the 14-day model as our non-whale D120 LTV predictor.

In summary, we train two new models: a whale classifier and a D120 LTV predictor for whales. We combine these with our 14-day model from before to create a new model for predicting D120 LTV. We first compare the model to our previous results assuming we have full knowledge of the true class label for each player and show the results in Table 5.5. Note that the “Non-Whales” row is the same as in Table 5.4. While the MAE certainly is worse than before, the RMSE is still comparable. Additionally, recall that the sum difference is the absolute error between the predicted sum of LTVs and the actual sum. We see that with true knowledge of the class labels, we get better performance on whales than before (see Table 5.3).

<table>
<thead>
<tr>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whales</td>
<td>$10,722.95</td>
<td>$21.93</td>
</tr>
<tr>
<td>Non-Whales</td>
<td>$166.90</td>
<td>$0.15</td>
</tr>
<tr>
<td>Total</td>
<td>$10,556.06</td>
<td>$0.47</td>
</tr>
</tbody>
</table>

Table 5.5: Evaluation of whale and non-whale models with classes known.

Of course, we cannot expect to have prior knowledge of the true class labels. We feed the players through the classifier and, using the output of the classifier and our threshold of 0.3, send the players through the appropriate model. We show the results in Table 5.6. We see here the price we pay for introducing a classifier that has poor recall on the whales. Clearly, some of the whales are predicted to be otherwise, thus increasing the errors for the non-whale cohort. However, the overcompensation of our whale model comes in handy here, making up a lot of the difference. The net total is a model that comes quite close to the actual total D120 LTV, with the smallest overall sum difference we have seen yet. Additionally, the RMSE is still better than the baseline model we explored previously (cf. Table 5.2).

<table>
<thead>
<tr>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pred. Whales</td>
<td>$11,373.65</td>
<td>$21.71</td>
</tr>
<tr>
<td>Pred. Non-Whales</td>
<td>$10,227.76</td>
<td>$0.25</td>
</tr>
<tr>
<td>Total</td>
<td>$1,145.90</td>
<td>$0.50</td>
</tr>
</tbody>
</table>

Table 5.6: Evaluation of whale and non-whale models using the trained classifier.

It should be noted that a better classifier would not require this overcompensation from the whale model. In this case, MSE could still be used for the whale model and we could set a typical threshold of 0.5 for the classifier. These should be reconsidered if changes are made to the classifier, but we will keep these values as is for the remainder of this section.
5.5 Training the Production Model

We have tested two different approaches to predicting D120 LTV. In order to quickly iterate on these ideas, we have limited our data set to a subset of players. We can, however, provide our model with more data. Specifically, we get the data for a random sample of all players, not just US players, created (or re-installed) between September 2017 and April 2018. We could go out further into April 2018 and beyond but we will leave that data to evaluate our production model. In total, there are 4543830 players in this data set. We use the same idea as before, training a model that uses three different sub-models: the classifier, the whale model, and the non-whale model. Recall that the non-whale model is trained on all players.

As a quick evaluation, we check how well our model does on the test set we have been using up to now. Note that since we are actually training on these players, these metrics are biased. We will perform a less-biased evaluation in the next subsection. We show the confusion matrix for the classifier in Figure 5.4 and the prediction results in Table 5.7. We see that if we do know the actual class labels, our model is actually quite good. MAEs and RMSEs are comparable or better than what we saw previously and the sum differences are quite small. However, when we use the classifier, because quite a number of whales do not end up in the correct predictor, our numbers are worse. We see again that prediction on non-whales suffers due to the misclassification of whales and the whale model must compensate. However, individual predictions (based off the reasonable values for MAE and RMSE) seem not to be affected by this. That is, most individual predictions will likely be just as good as before.

Figure 5.4: Confusion matrix for the whale classifier trained on a larger data set.

5.6 Comparison to Current Production Model

We now compare our trained production model to the one currently in use at Big Blue Bubble. We use two test data sets in our evaluation. The first uses a random sample of global players
5.6. **Comparison to Current Production Model**

<table>
<thead>
<tr>
<th></th>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whales</td>
<td>$16.66</td>
<td>$15.73</td>
<td>$40.60</td>
</tr>
<tr>
<td>Non-Whales</td>
<td>$1,920.16</td>
<td>$0.14</td>
<td>$0.56</td>
</tr>
<tr>
<td>Total</td>
<td>$1,903.50</td>
<td>$0.37</td>
<td>$4.99</td>
</tr>
</tbody>
</table>

(a) Evaluation using actual class labels.

<table>
<thead>
<tr>
<th></th>
<th>Sum Diff.</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pred. Whales</td>
<td>$2,931.71</td>
<td>$14.63</td>
<td>$36.17</td>
</tr>
<tr>
<td>Pred. Non-Whales</td>
<td>$12,400.30</td>
<td>$0.25</td>
<td>$3.67</td>
</tr>
<tr>
<td>Total</td>
<td>$9,468.58</td>
<td>$0.41</td>
<td>$5.29</td>
</tr>
</tbody>
</table>

(b) Evaluation using predicted class labels.

Table 5.7: Evaluation of LTV model trained on more data on the test set.

created (or re-installed) between May 1 and May 7, 2018 and we will refer to it as *World*. The second uses a random sample of US players created (or re-installed) between April 1 and April 14, 2018 and we will refer to it as *US*. *World* contains 150194 players and *US* contains 60504 players. Note that because of the time ranges we have selected, we have not seen these players during the training phase. In these data sets, we only consider players playing on either iOS or Android platforms.

5.6.1 **Evaluation on World Data Set**

The *World* data set contains 150194 players, 638 of whom are classified as whales. We show the predictions on this data set in Table 5.8. First, we look at the predictions assuming that we know the ground truth labels for the players. We see that our model performs better than the current in-production model in all regards. In fact, our model is extremely close to the actual sum for the predictions and has all-around better MAE and RMSE for all cohorts. However, when we switch to using the classifier, our model is not quite as good. Our MAE and RMSE for the cohorts are still better than the current model, but the sum of our predictions for predicted non-whale players is worse than that for the current model. Again, this is because our classifier is misclassifying some whales. With that said, our model still performs much better on predicted whales. Overall, we would consider our model to have a slight edge over the current in-production model.

5.6.2 **Evaluation on US Data Set**

Next, we evaluate on the *US* data set which contains 60504 players, 958 of whom are whales. We show the predictions on this data set in Table 5.9, again evaluating using both the actual class labels and the predicted ones. Again, our model outperforms the current model when using the actual labels. This time, however, our model also does fairly well when using the classifier. The error on the sum of predictions on predicted non-whales is still worse due to the recall of our classifier on whales. However, the difference in this data set is not so large that it allows the total errors our LTV predictions to edge out over the current model.
Table 5.8: Evaluation of the LTV model on the World data set.

<table>
<thead>
<tr>
<th></th>
<th>Whales</th>
<th></th>
<th>Non-Whales</th>
<th></th>
<th>Total</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum Diff.</td>
<td>MAE</td>
<td>RMSE</td>
<td>Sum Diff.</td>
<td>MAE</td>
<td>RMSE</td>
</tr>
<tr>
<td>Pred.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Whales</td>
<td>Current Model</td>
<td>$638.77</td>
<td>$23.82</td>
<td>$46.12</td>
<td>Current Model</td>
<td>$3,234.03</td>
</tr>
<tr>
<td></td>
<td>Trained Model</td>
<td>$546.07</td>
<td>$15.12</td>
<td>$31.72</td>
<td>Trained Model</td>
<td>$665.70</td>
</tr>
<tr>
<td>Pred.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Whales</td>
<td>Current Model</td>
<td>$3,234.03</td>
<td>$0.08</td>
<td>$0.43</td>
<td>Current Model</td>
<td>$665.70</td>
</tr>
<tr>
<td></td>
<td>Trained Model</td>
<td>$546.07</td>
<td>$15.12</td>
<td>$31.72</td>
<td>Trained Model</td>
<td>$665.70</td>
</tr>
</tbody>
</table>

(a) Evaluation using actual class labels.

Table 5.9: Evaluation of the LTV model on the US data set.

<table>
<thead>
<tr>
<th></th>
<th>Whales</th>
<th></th>
<th>Non-Whales</th>
<th></th>
<th>Total</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum Diff.</td>
<td>MAE</td>
<td>RMSE</td>
<td>Sum Diff.</td>
<td>MAE</td>
<td>RMSE</td>
</tr>
<tr>
<td>Pred.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Whales</td>
<td>Current Model</td>
<td>$638.77</td>
<td>$23.82</td>
<td>$46.12</td>
<td>Current Model</td>
<td>$3,234.03</td>
</tr>
<tr>
<td></td>
<td>Trained Model</td>
<td>$546.07</td>
<td>$15.12</td>
<td>$31.72</td>
<td>Trained Model</td>
<td>$665.70</td>
</tr>
<tr>
<td>Pred.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Whales</td>
<td>Current Model</td>
<td>$3,234.03</td>
<td>$0.08</td>
<td>$0.43</td>
<td>Current Model</td>
<td>$665.70</td>
</tr>
<tr>
<td></td>
<td>Trained Model</td>
<td>$546.07</td>
<td>$15.12</td>
<td>$31.72</td>
<td>Trained Model</td>
<td>$665.70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pred.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pred.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>Current Model</td>
<td>$1,299.69</td>
<td>$25.12</td>
<td>$67.29</td>
<td>Current Model</td>
<td>$6,293.47</td>
</tr>
<tr>
<td></td>
<td>Trained Model</td>
<td>$68.39</td>
<td>$16.31</td>
<td>$48.39</td>
<td>Trained Model</td>
<td>$99.96</td>
</tr>
</tbody>
</table>

(a) Evaluation using actual class labels.

(b) Evaluation using predicted class labels.

Overall, our model can be considered to be competitive or better than the current model. Our model has the benefit of introducing a classifier that can be used by itself. We see that improvements to the classifier should lead to immediate improvements to predictions as we
5.7. Evaluation on Individual Players

Here, we take a closer look at the predictions for two random players from the US data set, one whale, who we will call player A, and one non-whale, who we will call player B. We plot the input features and some other related plots for these players in Figure 5.5. The y-axis scales have been omitted, but note that they are different between the two players.

Let us begin with the revenue-related input features. We see that player A makes a purchase on day 7 that automatically qualifies them as a whale. It is hard to discern how the other revenue features for this player behave but it is likely to similar to that of a typical player, as we can see.
Chapter 5. Predicting Lifetime Customer Value in My Singing Monsters

with player B. Perhaps it should be noted that player A does appear to complete an offer on the third day. Moving to the count-based features, we see that both players have more activity for the first few days and are less activity following that. This is to be expected as, at this point, the player would likely be waiting for breeds and incubations to complete without having much else to do in the game. Note that the number of impressions and ad revenue features are closely related, particularly if we are only looking at players from the same country, as we are doing here.

We see that player B has a very long session on day 2. Following that day, player B does not spend quite as much time with the game. Player A does not have a day with quite that many session seconds, but does peak on day 6 and comes down after that, though not quite to the extent that player B does. We also see that player B spends a lot of diamonds on the first two days and not much thereafter as they have exhausted the free diamonds given to them or which are easily earned at the start of the game. On the other hand, player A has quite a few high spikes of diamonds spending, with the first two corresponding to the offer claim and in-app purchase. Player A spends a lot of diamonds on day 13. Interestingly, it would appear that player A is still lower in level than player B which might prompt us to investigate how player A spent their diamonds.

In the LTV plot, we show the cumulative LTV for the first 14 days. We also run the model multiple times on the player, adding a new day each time and plot the how the prediction changes as the model sees more and more of the first 14 days (similar to what we did in Subsection 4.5.2). We observe that both players’ D120 LTVs are higher than their D14 LTVs. For player A, the model over-predicts, while it under-predicts for player B. We also see an increase in the predicted LTV for player A on days 8 through 10 which appear to correspond with the decent activity in the session seconds feature and a decent number of diamonds spent on these days. The large spike in diamonds spent for player A does not seem to affect the prediction very much, however. For player B, the model’s predictions appear to increase during days 3 through 5, which correspond to an increase in ad impressions and, consequently, ad revenue. A small bump upwards is noted on day 11 as well, which appears to correspond a small increase in many of the features for player B.

Finally, we visualize the attention weights our model uses in an attention map, normalized across rows as was done in previous sections in Subsection 4.5.2. We see that the models, in both cases, tend to focus on the more recent days, specifically the three most recent days or so. Of these days, the model appears to emphasize based on recency. Notably, there appears to be slightly more attention placed on day 7 for player A, corresponding to the in-app purchase. A similar pattern is not observed for player B.

Overall, the predictions on these players seems reasonable. Both project that the player’s LTV will increase when it actually does. The predictions do seem to be affected by changes in the input features and appear to make reasonable predictions when we have less than 14 days. As such, we could also use the model earlier than 14 days if we wanted. It is unlikely the current model would perform well past 14 days, though it remains to be tested. If we wanted this behaviour, we can simply train on more days, as we saw previously that it does not seem to hurt day-14 predictions anyway.
5.8 Publication as a Shiny App

As was done previously with the hourly features model (Chapter 3), we create a Shiny application for use internally at Big Blue Bubble. The application is specifically suited to predict individual or small groups of players. Larger evaluations must be performed elsewhere due to computational limitations of the machine on which the application runs. We show a snapshot of the web page in Figure 3.11 which shows the predictions and input features of a My Singing Monsters player. Note that y-axes scales and the player’s actual and predicted D120 LTV have been omitted. Essentially, the application is an interactive version of Figure 5.5. The application also allows the user to select multiple players manually or to upload a CSV file containing the bbb_ids of multiple players and show the plots in aggregate. Additionally, the user can select whether the aggregation function is the sum, mean, or median function.

![Figure 5.6: Snapshot of the Shiny application for the D120 LTV prediction model.](image)

5.9 Discussion

In this section, we demonstrate how recurrent neural networks can be used to predict long-term player value in My Singing Monsters. In particular, we use the first 14 days of player data to predict the player’s LTV at their 120th day. This allows us to evaluate what kinds of players are in a particular cohort and can be used to quickly evaluate things like user acquisition campaigns.

We first explore the hypothesis that a model trained on more than 14 days might perform better, even when evaluated on only 14 days. We trained two additional models to a 14-day model, a 30-day and a 60-day model, but found that the performance of these models was
not significantly different to the 14-day model. Considering that the larger models require more resources to train, we concluded that it would be sufficient to use a model trained on 14 days. However, we note that if one wanted to predict D120 LTV using more than 14 days (for example, re-evaluating campaigns after a month), it may be better or even necessary to train these larger models.

We note that our prediction on high-value players, referred to as whales, is a large source of the error in our model. Thus, we trained a separate model on the whales. However, since we do not necessarily know which players will be whales after 14 days, we also needed to train a classifier to separate out whales and non-whales. Our classifier has poor recall on whales so we train the prediction model for whales with a custom loss function that encourages slight over-predictions. Finally, we argue that the model to be used on non-whales should be trained on all players, rather than just non-whales. We show that the three-model predictor performs better than the single-model predictor.

We then re-train this model using a larger data set and evaluate it on both the test set that was used previously to evaluate the models, as well as two other tests set that contain data which the model has not yet seen. We compare our trained model to the model currently in use at Big Blue Bubble and show that it performs competitively to that model. In fact, the model performs better on an individual prediction basis and can outperform the current production model in aggregate. Finally, we mention the Shiny application that is intended for small explorations using our trained model to be used internally at the company.

Further work might look to improve the classifier. Here, we simply used a similar architecture as the prediction model, but a more robust architecture search might come up with a better model. Additionally, more features can be added as input to any of the models. Particularly, if there is a feature that better identifies whales, including that feature can help the classifier immensely. We noted that the the fast plateauing of the loss functions may be an indication that this could help. Also, we use a custom loss function in the whale model only, but a better loss function than MSE might improve the model for non-whales, or even make a single-model system work better. In Chapter 3, we used a quantile loss function to build prediction intervals around the predicted median. Similar techniques for reporting prediction uncertainties can be explored in this application. Finally, if the model should be pushed to production, then a better way to deploy the model than a Shiny application should be explored.
Chapter 6

Clustering *My Singing Monsters* Players Using Autoencoders

6.1 Motivation

We saw previously in Section 4.6 that we can pull out the internal vector representations of the players that a deep model produces and use that to cluster *My Singing Monsters* players. However, we noted that the vector representations made there appear to be tuned towards making a correct prediction of the player's class. While this makes sense for that application, it might also mean that the representations throws away other information that is less useful for making conversion prediction. For understanding the behaviours of our players, we want a more general representation of our players with which to use for our analyses. To that end, we explore using autoencoders to produce these representations and then overlay some additional information about the players on these encodings and observe any trends which may appear. We also employ an algorithm called archetypal analysis [105], which has been used in previous studies on clustering players [44, 45, 106], to find players around whom to form clusters. These players represent extreme examples of players in our player base and are thus easier to interpret, a problem which we discussed previously in Section 4.6.

6.2 Data Preparation

The data which we will use for this section comprise of a random sample of US players who were created between January 1, 2018 and June 1, 2018 and who have not made a purchase in their first 30 days. We calculate the creation date of the players the same way as we did in Section 5.2 where we also consider any re-install dates which appear in our database. As before, we query the Redshift database for this data. We next consider a few features for our players which are aggregated on a daily basis. These features include the player’s minimum and maximum level on a given day, the number of coins and diamonds spent, and the number of monsters the player bred or sold on that day. In total, we consider 15 total daily features for our players. We take only the first 30 days of features for each player; since the date range for our query is beyond 30 days since when the query was made, there are no players who do not have a valid 30 days since their creation date. Our final data set consists of 46129 players. We
apply a log-transformation to all values to bring them all within a comparable range.

## 6.3 Model Architecture and Training

The architecture we are using here is an autoencoder, which is a special type of neural network where the output is the same as the input. In other words, we train a neural network to reconstruct the input which it is given. With enough memory, this should be a trivial task. However, in autoencoders, we restrict the size of the internal layers so as to force the model to learn how to “compress” the data with minimal reconstruction loss. Our model consists of a GRU \[28\] (uni-directional, as we did not find a large difference with bi-directional ones) to read in the daily features, outputting to a single vector (a many-to-one model). The GRU has 128 hidden units so our vector will have 128 dimensions. This is our player encoding that we will use for clustering, thus this part of the autoencoder is called the “encoder”.

However, in order to train our model, we need to have the second part, called the “decoder”, which takes this 128-dimensional encoding and repeatedly feeds it through another GRU (also with 128 hidden units) for each time step in the input (in our case, this would be 30). This GRU outputs a vector at each time step which then goes through a fully-connected layer to output the reconstructed 15 features. The weights for the fully-connected layer are shared across time steps. For our purposes, we can discard the decoder as we only require the learned encodings for our clustering task. However, decoders are an interesting area for exploration, particularly in architectures such as variational autoencoders.

Because we want to tighten the bottleneck, which is provided to us by the encoder, as much as possible, we do not search for hyperparameters here. Rather, we manually set the size of the latent representation of the player to be 128. This is not to say that a hyperparameter search may not be worthwhile, but simply that it was quicker and easier to not do it. Because we do not perform model selection here, we also discard the use of a validation set and instead train our model on the full data set. We build the model in Tensorflow \[97\] and train in a Google Colaboratory notebook for 64 epochs using the COCOB optimizer \[98\] and a batch size of 128 and plot the loss in Figure 6.1. Here, we use mean squared loss here which corresponds to the mean-squared logarithmic reconstruction error of the original data.

![Figure 6.1: Training loss for the player autoencoder.](image)
6.3.1 Dimensionality Reduction and Clustering

Once we have the autoencoder, and thus the encoder, trained, we can pull out the individual encodings for all of our players. Recall that these are 128-dimensional vectors. Of course, we cannot visualize in such a high-dimensional space, but we can use dimensionality reduction techniques to bring down this encoding into two dimensions, which we can then visualize. To do so, we will use t-SNE [101]. In particular, we are using a multi-core implementation of the algorithm as provided by Ulyanov [102].

Before we plot these points, we will cluster our players using their encodings (note that we are not clustering using the t-SNE-reduced data points). We previously applied k-means [78] and HDBSCAN* [81] to cluster players in Section 4.6 and noted their strengths and weaknesses. One weakness common to both algorithms is the interpretability of the clusterings they produce. For k-means, cluster centroids do not necessarily correspond with a real player. Additionally, the centroids can be quite close to one another, making them appear quite similar and making them hard to interpret [45]. For HDBSCAN*, we are given the cluster labels, but must manually investigate these clusters to understand them. This is difficult as there can be many players to look through in one cluster and averaging their features leads to similar problems as k-means and may destroy any benefits we gained from using a density-based clustering approach.

Instead, we will use an approach called archetypal analysis (AA) [105]. This method requires that we specify k as we do with k-means. However, instead of identifying k centroids, it looks for k archetypes. The archetypes identified represent extreme points in the data set and every other point can then be considered as a convex combination of these archetypes. Analogously, archetypes would be members of the convex hull of a finite set of points. This technique has two key advantages: the first is that the archetypes are actual data points in the data set (in our case, actual players) and thus are directly interpretable and are valid (as an example of invalidity, a centroid in k-means may have a fractional level or gold value, which is not possible); and the second is that because archetypes are extremal points, they are generally quite different from one another, making it easy to distinguish among them.

We use the py_pcha package [107] to perform archetypal analysis. We arbitrarily choose to look for \( k = 6 \) archetypes. Once the archetypes are found, we assign each player to the closest archetype via nearest-neighbour search. We plot the t-SNE-reduced player encodings in Figure 6.2 using different colours to differentiate the different clusters and mark the archetypes with a red cross. We see that there appears to be a central “mass” of players, the majority of whom belong to cluster 3. There are also some fragmented groups of players appearing around the central mass. Larger clumps of players here appear to mostly belong to cluster 6, while the sparse groups, as well as various “chains” of players, belong to cluster 5. We now turn our attention to the “tip” of the central mass which consists of three clusters: 1, 2, and 4. Cluster 4 constitutes the very tip, while clusters 1 and 2 split the space below it. A little bit of cluster 3 appears to be mixed in here, primarily with cluster 2.

Looking at the clusters, we cannot really tell anything about their properties. One cannot reliably trust spaces and sizes in a t-SNE plot. As an example, we see that the archetypes do not generally appear at the edges of the clusters as we would expect nor does it appear that clusters 1 or 4 even have archetypes. This is an effect of dimensionality reduction; in three dimensions, we see more clearly that the archetypes belong in a unique cluster (we do not show these plots...
Figure 6.2: Archetypal analysis clustering of the 30-day player encodings.

In higher dimensions, we would find that the archetypes would also appear to move toward the edges of the clusters. We can however, for the time being, assume that the larger clusters 3, 5, and 6 have relatively more variance in the types of players within them, however one might define “variance”, compared to clusters 1, 2, and 4. To concretely look at the cluster (population) size, we plot them relative to the other clusters in Figure 6.3. We see that cluster 6 is the largest cluster, with cluster 3 following close behind in second place. Cluster 5 is the third-largest, meaning that these high-variance clusters constitute more than two-thirds of the entire data set. Meanwhile, the three “tip” clusters only make up about 14% of the players.

Figure 6.3: Relative sizes of the archetypal analysis clusters.

Next, we overlay the plot with each player’s day-120 (D120) lifetime value (LTV). The calculation for this number is the same as in Chapter 5: we add up revenue collected from advertisements, purchases, and offers for each player up to their 120th day. The input features
do not contain any information about these values and thus, the model should not be aware of it while making the encodings. We use a colour scale for the log-transformed D120 LTV, with blue representing low LTV and red representing high LTV, and show the plot in Figure 6.4. Note that some players had incomplete LTV data and were removed from the plot. Interestingly, we find that many of the high-value players lie in the “tip” clusters of cluster 1, 2, and 4. In fact, the closer a player is to the tip of cluster 4, the greater their D120 LTV generally is. Cluster 3 appears to be a mix of low-to-medium-value players and the outer clusters 5 and 6 are mostly composed of low-value players. There are some high-value players in clusters 5 and 6, but these are quite rare, especially considering the relative sizes of the clusters.

![Figure 6.4: 30-day player encodings overlaid with their D120 LTVs.](image)

We now overlay two other variables onto the encodings plot. First, we look at whether the player was acquired organically or not. A non-organic player refers to a player who was introduced to the game via some user acquisition campaign. Second, we look at the platform on which the player plays. For all intents and purposes, the AFTB platform should be considered as equivalent to the Amazon platform. We show these two plots in Figure 6.5. We see that there appears to be no correlation with these features and the clusters. Perhaps more importantly, there appears to be no correlation with these features and D120 LTV. This shows that these features are not good predictors for LTV.

Finally, we look at player retention. For our purposes, we define a player as being “retained” if they have an active day (a day for which there is an event) after their 60th day, and “lost” otherwise. As an implementation note, we only query for up to the player’s first 180 days to save memory. As such, a player that has no active days between their 60th and 180th day, but do have an active day after day 180 would still be regarded as not being retained, or lost. However, these players are likely extremely rare and are likely not worth considering anyway. We show retained players in our plot in Figure 6.6. We see that there appears to be many more retained players in “tip” clusters of 1, 2, and 4. This also clearly shows that there is a correlation with player retention and LTV.
6.4 Clustering and Analysis of Retained Players

We saw previously that clusters 1, 2, and 4 make up a minority of the players but contain most of the high-value and retained players. We now do a deeper analysis of these players. First, we return to the original sample of US players who were created between January 1, 2018 and June 1, 2018 (not just the sample we took for the analysis up to now). We look only at players
who are retained (again, using our definition), but include those who have made a purchase in the first 30 days, and take a random sample of 46802 retained players. Thus, we also add an additional feature that is the daily number of in-game purchases by the player. We perform the same process as before, training an autoencoder on these players. This time, however, we use 60 days rather than 30. This is to hopefully give the autoencoder more information to squeeze into the latent space and get more useful encodings. However, we keep the architecture the same which means that the latent space is still 128 dimensions. This, along with the fact that there are more values to reconstruct, leads to the higher loss as seen in Figure 6.7. However, this does not mean that the encodings are unusable.

With the encodings produced, we can again send them through both the t-SNE and AA algorithms. Again, we pick $k = 6$ for the AA algorithm. We plot the encoded players and the clusters created in Figure 6.8a and their relative sizes in Figure 6.8b. This time, we see a lot less fragmentation in our plot as most of this was previously associated with the variance in the lost players. Now, there is only one cluster, cluster 1, which contains most of these fragmented groups. We see again that there is a “tip” in the plot, this time assigned to cluster 2, that lies on the opposite end of the plot as cluster 1. Neighbouring cluster 2, we have clusters 4 and 6, followed by the remaining clusters 3 and 5. We also see that the size of the clusters are more evenly distributed here, though cluster 1 containing the fragmented groups is still the largest.

Next, we plot the D120 LTV for the retained players, shown in Figure 6.9. We do not plot the acquisition or platform since we did not find any correlation in those plots. We see in the LTV plot that there is again a pattern where high-value players tend to aggregate in the “tip” cluster. Clusters 1 and 3 have the lowest-valued players, with the average D120 LTV increasing as we move towards cluster 2. This makes sense as we have effectively “zoomed in” on the “tip” clusters from the previous subsection.

Our final analysis will look at the archetypes that AA has identified. For this, we will look at the input features individually. However, in the interest of conciseness, we will only look at the minimum and maximum level features and use them as a proxy for measuring engagement. We show these values over the 60 days for each of the six archetypes in Figure 6.10.

For ease of reading, we will refer to the archetypes as CX where X is the corresponding cluster index. We first look at archetype C1. After 60 days, they have the lowest level of all archetypes and have achieved that on the very first day. As such, we can project that this cluster contains players who left the game very early on with minimal interaction. Of course, they must
Figure 6.8: Archetypal analysis clustering of the 60-day retained player encodings with the relative sizes.

Figure 6.9: 60-day retained player encodings overlaid with their D120 LTVs.

return at some point for them to be included here, but their activity is likely to be minimal. Let us now consider C2 whose cluster contains most of the highest-valued players. We see that they end with the highest level after the 60 days and that their level gradually increases over the course of these days. Thus, we might describe players from cluster 2 as active and engaged. Interestingly, C2 does not start with the highest level, but because of the consistency of their play rate, they slowly surpass the other archetypes. Note again that these players typically have the highest D120 LTV. This does not necessarily mean that high engagement leads to high LTV, as it might instead be that players who have the propensity or capacity to spend will be more
engaged as a result. For what it is worth, we note that all archetypes do not have any in-app purchases during these 60 days.

Two archetypes, C5 and C6, initially have a higher level than C2. This may mean that they are more engaged with the game early on, but become less interested as time goes on. Since C2 is still a non-paying player, it is not possible that this advantage was achieved simply because C2 has paid to get ahead. C2 and C6 appear to be at about the same level a month into the game, but C6 does not appear to progress any further for the second month. This may be due to differences in play styles which causes C6 to not level up as quickly, or it may be that C6 is not quite as active as C2. A quick look at the other input features (not shown here) would point toward the latter case. C5 gets off to an early start but drops off much quicker than C6. After about a week into the game, their level ceases to change as their engagement with the game is minimized. This relationship is reflected in the encodings plot (Figure 6.8a): cluster 6 neighbours cluster 2 whereas cluster 5 is farther away. However, because of the early engagement, cluster 5 is still closer to cluster 2 than cluster 1.

Two archetypes remain to be discussed. C3 has medium engagement early on and drops of just as quickly as C5 did. C4 has even less early engagement than all other archetypes, but consistently plays throughout the first month, ending them at a higher level than C3. This retained engagement might be why cluster 5 is the only other cluster to neighbour cluster 2 while, at the same time, neighbouring cluster 1. If getting players into cluster 2 is the desired goal, then it would appear that it would be more successful working with players who have consistent engagement throughout the whole first month, rather than simply those who have high early engagement. We summarize this analysis in Table 6.1. We use a simple eye-test on the D120 LTV plot to fill in the D120 LTV column.
Cluster | Behaviour | D120 LTV  
--- | --- | ---  
C1 | low early engagement; no consistent activity | Low  
C2 | medium early engagement; high consistent activity | High  
C3 | medium early engagement; little consistent activity | Low-Medium  
C4 | low early engagement; medium consistent activity | Medium-High  
C5 | high early engagement; little consistent activity | Medium  
C6 | high early engagement; medium consistent activity | Medium-High  

Table 6.1: Analysis of six different retained player clusters based on their archetypes.

### 6.5 Discussion

We tackle the task of clustering players using autoencoders. Since the output of the autoencoder is simply the input features, it must be able to make a good representation of the player in its constrained, latent space. We take this latent representation and use it to plot the players using t-SNE and cluster the players using archetypal analysis. We overlay D120 LTV, acquisition status, platform, and retention status on this plot and note a correlation with D120 LTV and retention and no correlation between these features and acquisition status or platform. We then perform a deeper dive with retained players and note similar correlations. Finally, we look at the archetypes produced during clustering to understand what kinds of players exist within each cluster and summarize the behaviours of each archetype by looking at their player level over the first 60 days. Overall, we find that players who are actively engaged with the game throughout the first month, or those who are extremely engaged early on, tend to have higher LTVs. This may be an intuitive statement, but encoding and clustering allows us to see this relationship clearly.

Clustering is an unsupervised machine learning task which means that it is difficult to evaluate. Many different changes to our method here could be made. For example, we pick $k$ arbitrarily but it may be that other values for it make more sense. Additionally, archetypal analysis may not even be the correct clustering algorithm for this task. However, since it produces interpretable clusters and we wish to attempt to evaluate our clustering ourselves, it makes the most sense to use it regardless. Other changes would be to look at the autoencoder and see if there is a better architecture that minimizes reconstruction loss further. It should be kept in mind, however, that the latent space should still be constrained to at most 128 dimensions or any comparisons will be invalid. A better player encoding may lead to more distinct or informative clusterings.

Finally, we look only at the player’s first 30 or 60 days. Other date ranges may be appropriate. Alternatively, we might even want to change the granularity to hourly or weekly and see if that produces better clusterings. We could also take an even deeper dive by only looking at players from cluster 2 in the final clustering and see if we can break up the behaviours of high-value players further. Finally, adding more features as input may augment the encodings, though this requires the autoencoder to compress more information and may not really benefit, or could even hurt, the analysis.
Chapter 7

Conclusion

7.1 Summary of Analyses

In the previous four chapters, we have explored how recurrent neural networks can be applied to player data from the free-to-play mobile game *My Singing Monsters*.

Traditionally, one of the tasks assigned to analysts is to monitor various game metrics to make sure that the game is healthy. We see an example of this in Chapter 3 where we look at various hourly metrics including the number of new players and game sessions per hour. An obvious hourly periodicity is noted in the data and when we train a recurrent neural network to predict the next 24 hours, we see this reflected in the predictions. Additionally, because we set up the model to use attention, we can visualize which hours appear to have the most influence on any particular prediction. Notably in this chapter, we use a loss function that also outputs a lower and upper quantile for each prediction. This allows us to see whether the actual value for that hour might be considered anomalous. Setting up a Shiny app using this model augments the analyst’s task of monitoring metrics behaviour by adding a short-term forecast and anomaly detection.

Then, in Chapter 4, we turn to the problem of player conversion prediction. In particular, we use a recurrent neural network to create a model that predicts a player’s conversion state—converted, not converted but active, and not converted and churned—at an early stage in their life cycle. We aggregate event counts, among other features, for the first ten game sessions for each player into ten vectors which are fed sequentially into the model. We see that even despite an unbalanced data set (over 80% players are churned and did not convert), the recall rate across all predicted classes is good and can be used practically to reduce the number of players required to observe to get most of the players belonging to a particular class (see Subsection 4.5.1). We also see that we can use the model to predict conversion over the course of the first ten sessions, not only after the tenth session. Again, this model can be put into a monitoring tool to look at a single player or a cohort of players. In this chapter, we also introduce how the latent representation of the input data of a RNN model can be used for clustering.

Of course, all players will likely stop playing the game at some point or another. What may be of more interest from a business perspective is how much money the player makes for the company. Even if the player churns early, if they would have spent some money of the game, they may be just as valuable as a player who has yet to churn. For this, in Chapter 5, we use a
recurrent neural network to predicting player value after 120 days. We experiment with various models here and settle on an ensemble of RNN that achieves similar, if not better, performance than the current LTV model in use at Big Blue Bubble. We also provide a Shiny app to monitor these predictions over the first fourteen days of a player life which is the window on which the model predicts. One application of this model might be to monitor user acquisition campaigns and terminate unprofitable ones early since they are expensive to run.

The three previous applications are examples of supervised machine learning where the model learns from training input-output pairs. In Chapter 6, we explore an unsupervised application of recurrent neural networks by training an autoencoder on player data from their first 30 days. The autoencoder is forced to learn a latent representation of this data that minimizes reconstruction loss. We take this representation and apply a clustering called archetypal analysis (AA). We use AA to discover six archetypes of player behaviour in the data (one might think of archetypes as extreme examples of representative player behaviours in clusters). Particularly, we explore clustering of retained players and summarize the archetypal behaviours uncovered in Table 6.1.

7.2 Discussion of Analyses

In all the analyses we have done, we work with sequential data as input and have not needed to spend extra effort into carefully selecting features and transforming them into standard metrics which might be fed into standard feed-forward networks. This cuts out a time-consuming process in the design of these analytical model and could potential remove important information in the data. For example, in Chapter 6 we looked at 60 days worth of data for various features such as in-game currency totals. If we used a standard feed-forward network, how should we present the data? We might want to take averages across the entire interval, but we might lose information if there are some early (first few days) trends in the days. The minimum and maximum at various times might be important and when these important times occur might be affected the player’s history up until that point. As a result, recurrent neural network models can be much quicker to set up because we can skip this feature engineering step while still being capable of producing useful and interesting analysis.

An important thing to note across all analyses is that context is important. For example, when we show the attention maps, they should always be taken under the context of the particular model that created them. In other words, they are not representations of the ground truth and are there to help explain the model, not the data. Of course, if our model is good, then they might be accurate reflections of trends and saliencies in the actual data but this is not guaranteed to be the case. Despite this, they are still useful as a self-reflective tool for analysts to consider.

Another case of context is that the value of the result should be considered under a practicality context. In other words, a model that does not achieve over 90% on a particular metric can still be useful. In fact, a high metric does not necessarily mean that the model is useful because our data is unbalanced. For example, in Table 4.4, we report 0.83 precision on the test set. Simply because this number is fairly close to 1.0 does not make it good; when we break it down into the precision for the three classes, we see precision on the minority class is quite low. However, we note that because the recall is consistent across classes, the model does have
some value even though the recall rate is not in the high 90s (of course, a higher recall rate with all other things being equal would be more desirable). In all our analyses, we have discuss where the practical use of the trained model lie in the context of a business setting.

7.3 Avenues for Future Research

While this thesis presents several examples of utilizing recurrent neural networks to analyze player data, there are still many different avenues for future improvement and research. As it pertains to each particular analysis, such discussions can be found in the Discussion section in each analysis chapter. Here, we will acknowledge more general areas which can be expanded.

When looking at player, we use two different views in this thesis. The first is looking at data chunked into game sessions (Chapter 4) and the second is to chunk the data into days (Chapter 5 and Chapter 6). We do not really go into detail as to whether one way of treating the data is superior to the other as it depended on how we wanted to evaluate the model. However, it may be interesting to go deeper and determine if there is a better way to organize the data. We mention briefly in Chapter 4 that we may also want to go more granular and look at individual events rather than accumulating them into counts. This also raises the option of using a hierarchical model that looks at individual events within a game session or day.

This would also mean that we need to decide how events are represented. One option would be simply one-hot encode each unique event. However, some events may contain information that is helpful to the model. For example, knowing which monsters are bred together or the quantity of diamonds spent in a transaction are arguably more important than the fact that these events occurred. To include this, we might pursue a sparse encoding that includes both the one-hot encoded event and these values. However, with higher dimensionality comes a high computing cost and may cause the model to take longer to converge. However, this would be an interesting area to explore simply to try and improve the performance of the models presented here as they do not really take into consideration this information.

Another avenue for research is to observe how these deep analytical models affect the company in the long term. In other words, if these models are incorporated into day-to-day decisions, we would want to observe how the changes that arise from these decisions affect the company. For example, does using a LTV prediction model help lower costs of user acquisition campaigns or does it have no notable effect? In the worst case, over-reliance on the model might cause good campaigns to be terminated (i.e. false negatives) and end up costing the company more to acquire valuable players. In addition to seeing how the company is affected in the long run, it would be interesting to see how the model stand up to these changes. For example, if a churn prediction model helps reduce player churn, will it continue to perform as it did in the past or might the model’s false negative rate increase over time?

This thesis focuses on recurrent neural networks, but other methods may be viable and practical as well. One option would be to use convolutional neural networks to look at the sequential data. In particular, we can use one-dimensional convolution to look at temporal data. This method also works for sequences of varying length. We could also explore non-neural network techniques and see how well they compare to recurrent neural network. However, to reiterate, the goal of this thesis is not to achieve state-of-the-art results, but to explore the practicality of recurrent neural networks.
Finally, another area to explore would be to expand the scope of these analyses. For one, most analyses presented here deal with only US players since they make up the majority of the *My Singing Monsters* player base. We have seen already, in Section 5.6, that these models might be extendable to other countries if the model is trained on data from players from those countries. It might be interesting to extend the model developed in the other chapters or to see how well a model trained only on US player fares when inferring on players from other countries. Another option is to apply recurrent neural networks to other games produced by Big Blue Bubble. An obvious choice would be to look at the sequel game *My Singing Monsters: Dawn of Fire*. It would also be interesting to see how well models transfer between games. In particular, is there any benefit to training new models with weights pre-initialized by training on *My Singing Monsters* data versus simply starting with default initializations (i.e. can we get new models to converge faster/better by starting with old models)?

### 7.4 Concluding Remarks

In conclusion, we show in this thesis four applications of recurrent neural networks in analyzing data from a mobile video game. We show that RNNs can produce reasonable and usable results in the areas of metrics forecasting and anomaly detection, churn prediction, lifetime value prediction, and player clustering. In most models, we use an attention mechanism to help us understand on what the models appear to focus. For each application, we discuss how the models may be used in a business setting and acknowledge what might be improved or what could be explored further.

The number of games that will be available to the public will only keep growing, as will the number of potential players each game can reach. And as metrics tracking in games will become easier for developers to include, so will the number of ways that analysts can look at how the game is being played. With the amount of data only increasing, analysts will surely turn to more automated algorithms to try and make sense of all the data. Machine learning are, and will continue to be, an important tool for analysts. Recurrent neural networks are one method of looking at game data in its raw, temporal form and we show here that they are simple and effective tools to use for almost any analytical purpose. Continued research and publication in the area of game user research will both introduce new techniques for analysis and refine the tools and methodologies we already have and use.


Appendix A

In-Game Screenshots from *My Singing Monsters*

Figure A.1: Example of the main view in *My Singing Monsters*. Here we show Plant Island populated with many different species of monsters. Note the various currencies displayed in the top-right corner and the various menu buttons at the bottom of the screen. The bag of coins floating over several monsters’ heads means that they have earned the maximum amount of coins they can and prompts the player to click it to collect the coins. Finally, the player can select the buttons in the bottom-left to hide the UI or to record their screen.
Figure A.2: Island select screen. The player can scroll through the selection carousel and then press “GO”, which will move the player to that island. Some islands (the first five) also have a Mirror Island, which is effectively a second copy of that island. These can be accessed by pressing the mirror icon in the bottom-left of this screen.

Figure A.3: Market screen. There are different shops for monsters, decorations, structures, and islands. The Currency tabs allows the player to purchase various currencies in exchange for real-world money. The Gear tab redirects the player to the online merchandise shop. The last tab promotes other Big Blue Bubble games (in this case, Concert Kings) and clicking on it will redirect the player to the appropriate app store. Finally, the button in the bottom-left corner sends the player to the diamond offer wall.
Figure A.4: Breeding screen. Clicking on the Breeding structure and selecting Breed will bring the player to this screen. The player can then select two of the monsters on the island to breed. Here, we select a Bowgart and a Noggin in an attempt to breed the four-elemental monster Entbrat.

Figure A.5: An example of a speedup screen. Breeding will, after some time, produce a monster egg. The egg must then be incubated before it hatches. For both these periods, the time required can be shortened by 15 minutes by opting to watch an ad. Alternatively, it can be completed instantly at a rate of one diamond per hour remaining.
Figure A.6: Tribal Island. We have an example of a Tribal Island here with three members, each represented by one monster. The tribal level is the combined level of the members’ monsters, which they can increase by feeding their monsters coins, treats, or shards. Once tribal level 100 is reached, Kayna, the volcano monster on top of the tree stump, is unlocked for that island. Tribal levels reset weekly.

Figure A.7: Composer Island. Here, the player can write custom melodies for their monsters to sing. They can purchase most monsters with coins and immediately place them on their island. The player may also purchase additional Composer Islands using diamonds. Note that we have hidden the usual UI here.
Appendix B

Clustering from Conversion Prediction Model
Figure B.1: Cluster 1 from HDBSCAN*.
Figure B.2: Cluster 2 from HDBSCAN*. 
Figure B.3: Cluster 3 from HDBSCAN*.
Figure B.4: Cluster 4 from HDBSCAN*. 

Cluster 4 \( (C) = 9691 \) 

Conversion 

Reform 

Acquisition 

Session Length vs. Minutes Since Last Session 

Averaged Event Counts 

Session vs. Minutes Since Last Session 

Coins 

Diamonds 

Food 

Keys 

Shirts 

Figure B.4: Cluster 4 from HDBSCAN*. 

111
Figure B.5: Cluster 1 from $k$-means.
Figure B.6: Cluster 2 from $k$-means.
Figure B.7: Cluster 3 from $k$-means.
Figure B.8: Cluster 4 from $k$-means.
Curriculum Vitae

Name: Jonathan Tan

Post-Secondary Education and Degrees: Western University
London, Ontario, Canada
2012 - 2017 B.Sc.

Related Work Experience: Graduate Teaching Assistant
Western University
2017 - 2019

Business Intelligence Programmer Intern
Big Blue Bubble
2018 - 2019