Machine Learning In Predicting Video Game Match Outcomes

AP Research

Word count: 5088

Machine Learning In Predicting Video Game Match Outcomes

Introduction

With every passing year, the explosive growth of the competitive video game industry manages to outperform the records it set in years, and alongside that growth, game profits, player bases, tournament prize pools, and match viewership have all kept up in pace. Given the massive amount of money coming out of modern games, there exists a real benefit in the form of profit to be gained from any improvements to games that can be felt by players, and currently, problems related to putting players into fair matches are potentially ones that could be addressed. In most competitive sports and games, some sort of ranking system is necessary to determine relative player skill in a way that could be applied to placing players into fair matches. Although ranking an individual player compared to another is a relatively simple task, the issue becomes more complicated when deciding how to accurately determine a player's skill level in the context of the teams they find themselves placed onto. In the field of online video games and esports, being able to properly determine a player's true ranking is vital to the quality of service of those games, as matchmaking systems for such games, or the systems responsible for putting players into matches, generally rely heavily on creating two teams of a similar ranking (Riot Games Support Staff, 2019). In most cases, player ranking systems are based on the ELO rating system, a widely accepted and successful ranking model that originated from the need to rank individual chess players with respect to one another, where all players are given a value representing their rank that changes as they win or lose matches (Wunderlich and Memmert 2018). Despite being widely accepted as a successful ranking system, in the context of video game matchmaking, where how well randomly assigned teammates may work with one another can have a large

impact on match outcomes, ELO-based matchmaking systems simply overlook important details. In doing so, many matchmaking systems ignore factors that can be used as predictors of one-sided stomps, which presents an issue for publishers looking to reduce the amount of frustration the players of their games have.

In an ideal world, any given match would be constructed so that each side has as close to a 50-50 chance of winning as possible; however, current matchmaking systems sometimes create matches that are incredibly one-sided, generally known as "stomps," which can have a severe impact on player enjoyment (Riot Games Support Staff, 2019). Stomps are often incredibly frustrating for both sides of a match, and if player frustration were to mount enough, it is possible that video game publishers see lesser profit as a result of players no longer playing their game out of frustration. However, if stomps could possible be preemptively stopped, this issues could potentially be solved. This presents the question of whether the consideration of factors beyond player rank could lead to improved matchmaking via few stomps by determining if a given match will be highly unbalanced. Hence, this study aims to determine if the consideration of more player and match data by matchmaking algorithms can locate and prevent stomps from happening, which would increase player enjoyment, and by extension, profits.

Review of Literature

Since its creation, machine learning (ML) technology has been a powerful application of artificial intelligence (AI) used to process and make predictions about large datasets in more accurate and efficient ways than man-made predictions ever could. As the name implies, ML refers to the process of computers vigorously processing data, and in doing so, 'learning' or being 'trained' to make connections between various points of data and the expected result of an

event given that data. By training algorithms using sample datasets, machine learning algorithms are able to make relatively reliable predictions about the outcomes of future events. For example, Buscema, Massini, and Maurelli (2015) successfully used machine learning to accurately predict earthquake magnitudes based on prior earthquake data from Italy. In another study, Panjan, Šarabon, and Filipčič (2010) found that machine learning algorithms predicted the future success of tennis players with almost the same accuracy that professional coaches could. In another field, Sinthupundaja, Chiadamrong, and Suppakitjarak's (2017) study analyzing the application of machine learning in predicting the future financial standing of public companies in Thailand using stock exchange information also found that machine learning was highly accurate in its predictions. Even still, there exist many more studies that point to the same great potential of machine learning, such as Sturrock, Woolheater, Bennett, Andrade-Pacheco, and Midekisa (2018) identifying building types (such as schools, houses, etc.) with high accuracy using a number of different machine learning models, or Gutiérrez, Canul-Reich, Zezzatti, Margain, and Ponce (2018) finding that machine learning could accurately predict student sentiment of teachers through lexical analysis done using machine learning models. Surprisingly, despite the large variety of subjects covered throughout machine learning studies, the overwhelmingly common successful findings of machine learning point towards its widespread applicability in most fields and invites consideration of what other problems could be solved through the implementation of machine learning systems.

Rather recently, deep learning neural networks, implementations of machine learning that use multiple processing layers to learn and represent data with multiple levels of abstraction, have emerged as being even more promising than more traditional implementations of machine

learning strategies (Voulodimos, Doulamis, Doulamis, & Protopapadakis, 2018). As a result of a number of improvements in the field of machine learning, deep learning neural networks have been shown to outperform previous state-of-the-art techniques in several tasks, cementing its need for further exploration (Voulodimos et al., 2018). Despite its somewhat recent uprise, the applications of neural networks in machine vision, or analysis of images using AI, have already been both well tested and analyzed. In the medical field, deep learning neural networks have proven to be both highly popular and effective. In 2018, Ari and Hanbay were able to develop a brain tumor classification method that analyzed brain scans, which ultimately outclassed previously studied classification strategies at a 97.18% classification accuracy. Similarly, Burlina, Billings, Joshi, and Albayda (2017) also found that deep learning methods outperformed other implementations of machine learning. By testing a neural network's ability to diagnose if patients had myositis, a type of muscle inflammation, Burlina, Billings, Joshi, and Albayda found that the neural network was able to both outperform other machine learning methods and could do so with less manual effort. Even still, many other studies in the medical field have found similar results pointing to the effectiveness of deep learning neural networks in making accurate predictions, with Grainger, Tustison, Qing, Roy, Berr, and Shi (2018) applying deep learning neural networks to quantify body fat volume and distribution, and Ding, Zhang, Xu, Guo, and Zhang (2015) doing the same with EEG classification serving as two additional examples.

Given the success of deep learning in the medical field and the general success of deep learning neural networks, it is reasonable to question the applications of deep learning in other fields. As such, my research hopes to consider how machine learning can be applied to the field

of video game matchmaking. However, it is important to address the fact that this type of study has already been done before, albeit in the form of a project by an individual instead of a professionally conducted study. For example, Jagtap (2018) found that by using predictive modeling, the winner of a given match of the online game *League of Legends* (LoL) could be found with moderate success (roughly 60%) from the early stages of a match after testing a number of machine learning strategies when applicable data was supplied. Similarly, Yin (2018), also attempted to test the prediction accuracy of various machine learning models in determining the outcomes of LoL matches, and reached similar prediction accuracies to that of Jagtap. However, despite these experiments, the topic of match outcome predictions still deserves further research for a number of reasons. Some of the more significant points include Jagtap not considering a deep learning model as a machine learning strategy, Jagtap and Yin only using a limited number of variables in their models, and, most importantly, neither Jagtap or Yin testing the accuracy of predictions if they were subjected to the same limited set of information that matchmaking algorithms would have access to. To clarify, the information that matchmaking algorithms would have access to would only consist of data available at the moment the matchmaking system has placed all 10 players into teams, but before those players have made any selections related to the game. This would include vital information such as which characters were chosen, which has a significant impact on the direction of the match. By limiting predictions in such a manner, my study hopes to design a deep learning neural network to predict match outcomes before a match begins. If such predictions could be made reliably, it is possible that a similar model could be implemented into matching algorithms so that one-sided matches

could be avoided based on their confidence in the prediction, hopefully reducing player frustrations with games.

Hypothesis

Despite improvements over similar studies done by Yin (2018) and Jagtap (2018), such as considering a great number of data points per match and focusing on designing a deep learning neural network, I expect the model designed in this study to, on average, only be able to make predictions with an accuracy close to 50%, or the expected accuracy from random guessing. This outcome is somewhat realistic, as although more information is being considered compared to other similar studies, in order to mirror what a matchmaking algorithm would have access to, very critical data points need to be excluded. Generally, factors such as the selected characters within a match have been known to be very strong predictors of match outcomes, which is why the expected accuracy is so low. However, I still believe that my model may be able to be somewhat effective in determining matches that are highly biased towards one team, as some factors such as multiple players with losing streaks or no recent matches may still be used as strong predictors for match outcomes regardless of additional factors.

Methods

Because this study looks to evaluate the effectiveness of a deep learning neural network, two sets of data needed to be collected. The first necessary dataset was a large sample of match data that was fed into the algorithm for testing and training purposes. The second was the output of the neural network when tested using that sample match data and how that output compared to a baseline prediction to determine the prediction accuracy of the neural network.

One of the most important questions for this study was from which video game should data be collected. Ultimately, LoL was chosen over other popular games for a number of reasons. These reasons included match data being publically available, the game's popularity, the ability to compare results to similar studies from the same game, and the complexity of matches supporting a deep learning approach. Additionally, the researcher's familiarity with LoL in particular aided in knowing which data points would be significant enough to be fed into the neural network.

Match Data Collection

Given that the match data was being used as an input for the neural network, its collection was necessary before the design and creation of the neural network could occur. One of the major problems of the study at this point was that Riot Games, the developer and publisher of LoL, offers no method for easily obtaining a sample of recent LoL matches, and as a result, that sample had to be manually collected as a part of this study. Therefore, the researcher opted to collect match data by following a discriminative snowball sampling method. This sampling method is generally implemented when members of a population are difficult to locate individually, and operate by having one member of the population point to another, who points to another to create a sizeable dataset. This sampling method aligns with that of this specific study because while there is no method of obtaining a large sample of matches immediately, any given match can point to another match by analyzing the other matches played by the players in the previous match.

In order to collect the match data in accordance with the given sampling method, three main aspects that needed to be addressed: where to gather match data from, where to store that

data, and how to create the process of collection and storing the match data. To solve the issue of where to gather match data from, the Riot Games API was employed. The Riot Games API is a platform created by the developer and publisher of LoL, and it offers access to LoL match and player data given a specified match ID or player ID. Additionally, there is currently no other platform to request LoL match data from, so its usage was necessary. To solve the issue of where to store data, the Google Sheets platform was chosen. The Google Sheets platform was chosen because of its functionality allowing data to be collected and read without a local copy, as well as there being widespread support for its usage in scripts. As for the creation of the script, the programming language Python was chosen. Although virtually any programming language with support for API access could have been utilized, Python was chosen for its simplicity in development, the researcher's familiarity with the language, and its support for the Riot Games API and Google Sheets, all of which made it highly effective as a choice. Specifically, the RiotWatcher and GSpread Python libraries allowed for simplified usage of both Riot Games API and Google Sheet platforms when requests of any type were made, and both were utilized in the collection of data.

Before the process of data collection could begin, developer API keys needed to be requested from the Google Developer Console and the Riot Games API in order to record match information as well as obtain match and player information respectively. Following their acquisition, the match data was collected through the following process: First, a single match was specified as a starting point for the process. That match data was requested from the Riot Games API through the RiotWatcher Library, and the raw match data from the API was stored into a single cell of a Google Sheet using the GSpread Library. For every subsequent match, the

script collected a list of every player from the previous match logged in the Google Sheet, and also requested the match history for every player. The script then searched through the match history of each player by requesting the information from the Riot Games API to find a suitable match, which in the case of this study, was a match that was both not previously recorded in the Google Sheet and had a queue type of 420, specifying a ranked solo 5-on-5 match. Ranked solo matches were preferred for this study because of their more serious and competitive nature compared to other game modes, which would, theoretically, lead to better correlations between a player's expected performance and their historical performance when considered later on. After a suitable match had been found from any of the players in the previous match, that match was logged in the spreadsheet, and the process was repeated until a suitable number of matches had been logged. Figure 1, shown below, displays a section of the spreadsheet and how it appeared following this process. Additionally, the full Python script used for this process can be found in Appendix A.

Current Match Row #:	2003	Initial Match ID:	2986542020	* If the match ID = 0, the match should be ignored (from before p9.4) *								
Match ID	Match Outcome	Raw Match Data	Game Start Time	B1 ID	B2 ID	B3 ID	B4 ID	B5 ID	R1 ID	R2 ID	R3 ID	R4 ID	R5 ID
2986542020	Blue	{'gameId': 29865	1551200720089	t1TTve	KsO-tf	Ay9l0v	A2IbL/	T0NTg	r9UIYF	YkE7i	il7xBX	YrP1y	aHsXE
2986535343	Red	{'gameId': 29865	1551203010013	ooibbjf	4RmjX	t1TTve	CyWM	YTJ-xk	XcHE\	KsO-tf	2kNhC	0lCsV	Ay9l0v
2985929053	Red	{'gameId': 29859	1551100701759	RLhSc	so_idl	EkuzG	p7qD6	4Q091	1pbAo	IYmqG	m_ZE	Smcfo	ooibbjf
2985560318	Red	{'gameId': 29855	1551053856609	5U5Cl	4fUd5	KdOX(	0xFEn	7z2T1	RLhSc	7nO0-	MZpv(	nJB3j	Xe6SM
2985589246	Red	{'gameId': 29855	1551057926268	ci2UB	5U5Cl	03EFE	NN78)	JhHkF	tY7d_[	gyWW	myzDf	OF1K	8Bewv
2985568738	Red	{'gameId': 29855	1551055835218	hOZSt	QMIliL	f59Ntp	L9--c3	nk8dS	ci2UB	Z53oq	FAhgM	-GofA(	kDTg2
2985529677	Blue	{'gameId': 29855	1551053448824	JH4k_	hOZSt	KUz32	1nGP/	mKqfo	cyX3tT	bWhkt	t3Lg2M	YZMw	MAMA
2985641055	Red	{'gameId': 29856	1551060444746	veXZZ	N8Jcfl	r0Kgu	ZNjY-x	Twdc_	Mcilza	51p9i4	JH4k_	eN27v	smJ-jF
2986243313	Blue	{'gameId': 29862	1551148533529	iKReO	n_Hoti	V8fuh	HfBGF	10QhF	Hbr57	LAWB	cUdM	veXZZ	qrGdrF
2986362059	Blue	{'gameId': 29863	1551157425150	Y4o5H	s2w_0	Xq5Bje	3W3A	w8Sh8	iKReO	gYAJr	pXc21	kpvjET	z1Fhvi
2986477989	Blue	{'gameId': 29864	1551181576653	NMvSl	IdkU_x	KeNA	PsxeQ	ptJjevk	Y4o5H	pZadG	ZkpO1	U_ppC	2cad8l
2986508589	Red	{'gameId': 29865	1551197126321	0yhDR	m92-F	ICmN(	hNJyN	NMvSl	NfWjW	60X7z	I1TgV(	oFqCF	0lLpa8
2986528547	Red	{'gameId': 29865	1551199294871	NSOe	0yhDR	RC7s8	6JCVe	LFFMu	81sEiZ	5K80J	G-NrG	OJA7r	Z_LNI
2986582820	Red	{'gameId': 29865	1551209428210	2NFcd	IdhYm	gQEkF	0zdl5_	H72wk	NSOe	6VMFI	3GHzj	y-VLL	ca3zzh
2986440613	Red	{'gameId': 29864	1551166525595	pg2L3	wMHB	2NFcd	ei5WK	aD3CT	64QIfh	Vdsy1_	21jJiB	SASW	3PdSlI
2986432002	Red	{'gameId': 29864	1551163966663	8lutpA	pg2L3	N8X5F	qNdU5	w6ph4	VJx3z	flow9v	DtAWj	RHGc	gMYK(
0	Red	{'gameId': 29751	1549854653151	-FxZo	p3rQw	9st4z_	bk3FL	a_ntQl	aNJot	MKgtJ	Zw1c6	rkwvky	8lutpA
2986200340	Red	{'gameId': 29862	1551145362581	8TD64	qF_5S	ZfMLY	8mWZ	Od9N\	1ckmH	vcc-fci	pWYU	5SLeC	-FxZo

Figure 1 - Match ID and Corresponding Match Data as it would be catalogued in the Google Spreadsheet.

In total, 2000 unique matches were collected and logged, which comes from an expected dataset size of 1000 to 4000 data points to test and train the algorithm suggested by similar studies done by Yin (2018) and Jagtap (2018), as well as complications with Google Sheets being unable to store more than 2000 matches. Out of the 2000 unique matches collected and logged, 54 matches needed to be excluded from the dataset because they were played on a version of the game that was not patch 9.4, which at the time, was the most recent balance update to the game. The patch difference is notable because balance changes to certain characters or core game systems between patches can skew match results, and if the algorithm is unable to account for the changes, its predictions may be skewed.

Match Data Preprocessing

Since the purpose of this research is to determine prediction accuracy using data available prior to the match beginning, most of the data points available from the collected matches could not be used as inputs for the neural network's predictions. As a result, additional data points had to be collected pertaining to the players of each match. Out of the data that could be requested from the Riot Games API, data points from the previous match and historical ranked performance, or ranked statistics, were chosen to be collected. These data were selected because they appeared to be good indicators of potential player success in addition to being selected in similar studies done by Yin (2018) and Jagtap (2018).

Following the selection of the key data points to preprocess, a Python script was created to preprocess the match data by the following steps. First, stored match data in the spreadsheet was be requested, and from the data, the start time of the match, the assigned position of each player, and the team of each player was be stored, for a total of 11 data points from the initial

match. After that, the script went through each of the players in the match and requested that player's previous match and ranked statistics from the Riot Games API. Specifically, 25 data points were taken from the player's previous match, including the time the match started, whether or not the player won that match, how much damage they dealt in the match, among many others. Additionally, 9 data points were taken from the player's ranked statistics, which included their current rank, total wins and losses, and whether or not they were on a win streak, among others. Therefore, in addition to the first 11 data points collected, 35 additional data points were collected per player in the match, resulting in a total of 361 total data points used to predict the outcome of each ranked match.

Since some of the values taken from the match or player data came in the form of a word, they needed to be decoded into a number, as neural networks can only accept numerical values as inputs. For example, if a player's rank was 'Silver,' the value was stored as a '3,' and if a player's rank was 'Gold,' a rank just above Silver, it was stored as a '4.'

After all the values were stored and collected, the values were put into a list and stored into the spreadsheet so that they could be requested again when needed. After a the list was stored, the script repeated the process with the next match in the spreadsheet until it formatted every match. Figure 2 below shows a section of the spreadsheet as it would appear following this process, with every row having its match formatted, excluding those outside of the selected game patch. Additionally, the full Python script used for this process can be found in Appendix B.

Current Formatter Row #:	2003	1946
Is Match Prior to Patch 9.4?	Formatted Match Data	
	1551200720089,100,2,100,2,100,5,100,1,100,4,200,1,200,2,200,3,200,4,200,5,155120072008	
	1551203010013,100,0,100,0,100,2,100,0,100,0,200,4,200,1,200,3,200,2,200,5,155120301001	
	1551100701759,100,0,100,0,100,0,100,0,100,0,200,0,200,0,200,0,200,0,200,0,155110070175	
	1551053856609,100,1,100,5,100,2,100,3,100,4,200,2,200,3,200,4,200,5,200,1,155105385660	
	1551057926268,100,4,100,1,100,5,100,3,100,2,200,5,200,2,200,1,200,3,200,4,155105792626	
	1551055835218,100,4,100,5,100,2,100,1,100,3,200,4,200,5,200,3,200,2,200,1,155105583521	
	1551053448824,100,1,100,4,100,2,100,3,100,5,200,1,200,2,200,3,200,4,200,5,155105344882	
	1551060444746,100,0,100,2,100,1,100,0,100,0,200,1,200,2,200,3,200,4,200,5,155106044474	
	1551148533529,100,3,100,1,100,5,100,2,100,4,200,0,200,2,200,4,200,0,200,5,155114853352	
	1551157425150,100,4,100,5,100,1,100,3,100,2,200,3,200,1,200,5,200,2,200,4,155115742515	
	1551181576653,100,1,100,3,100,5,100,2,100,4,200,3,200,2,200,1,200,4,200,5,155118157665	
	1551197126321,100,3,100,5,100,2,100,4,100,1,200,2,200,1,200,4,200,3,200,5,155119712632	
	1551199294871,100,2,100,3,100,4,100,1,100,5,200,2,200,3,200,4,200,5,200,1,155119929487	
	1551209428210,100,3,100,1,100,4,100,5,100,2,200,1,200,3,200,5,200,4,200,2,155120942821	
	1551166525595,100,2,100,5,100,1,100,4,100,3,200,3,200,2,200,1,200,5,200,4,155116652559	
	1551163966663,100,1,100,2,100,3,100,5,100,4,200,4,200,2,200,0,200,0,200,5,155116396666	
Match is prior to Patch 9.4		
	1551145362581,100,2,100,1,100,3,100,5,100,4,200,4,200,2,200,3,200,5,200,1,155114536258	

Figure 2 - Formatted and decoded match data as it would be catalogued in the spreadsheet.

Algorithm Creation and Design

Following the collection and formatting of the LoL match data, the neural network could be created. At that point in the process, the main concern that needed to be addressed was which machine learning framework would be used to create the model. After consideration of various deep learning neural network frameworks, TensorFlow, a Python-based open source machine learning framework created by Google, was ultimately chosen for the purpose of this study (TensorFlow 2019). TensorFlow stood out among other available frameworks for a variety of reasons. One of the main appeals of TensorFlow is that it is easy to use. Due to the researcher's inexperience with machine learning in general, any framework that reduced the likelihood of errors due to inexperience was highly desirable. This factor was furthered by the fact that TensorFlow is widely popular, which meant that support for issues would be able to be found. Additionally, the widespread use also TensorFlow by large companies such as Google, Twitter,

Coca-Cola, and Ebay among many others is a testament to TensorFlow's effectiveness as a machine learning framework, which was also a large factor used in consideration (TensorFlow, 2019).

Following the selection of TensorFlow as a framework, guidelines available by TensorFlow for designing a neural network that makes predictions between two outcomes were followed. Specifically, the guidelines outlined a neural network model that utilized a binary cross entropy loss calculation in order to determine how to update the model during the process of its training, where the binary cross entropy loss refers to the difference between the expected output and the real output, which would be fed back into the model in order to improve its accuracy. Furthermore, the guidelines outlined a deep learning neural network consisting hidden layers connected sequentially and processed with an "adam" optimizer, where each layer refers to an additional set of calculation done on the input data before reaching an output. The following is the layers and number of output nodes recommended by the guidelines: a 16 input embedding layer, a global average pooling layer, a dense layer with 16 inputs, and another dense layer with 1 input. Following experimental testing of the completed model, the layers were experimentally adjusted to the following to improve prediction accuracy: an 8 input embedding layer, a global average pooling layer, a dropout layer with a keep probability of 0.5, an 8 input dense layer, and a 1 input dense layer.

After the model was designed, the formatted dataset needed to be split into smaller groups to generate training, validation, and testing sample sets. As the names allude to, the training set refers to the data that is fed into a model in order to 'teach' it to make predictions, the validation set refers to data that is tested against the model as it is being trained in order to ensure

proper training, and the testing set refers to the data that is used determine the accuracy of the model after it has completed its training. Following the data splits used by other studies, such as those by Buscema, Massini, and Maurelli (2015) and Ari and Hanbay (2018), roughly 70% of the match data was assigned to the training set (1400 matches), roughly 20% was assigned to the testing set (400 matches), and roughly 10% was assigned to the validation set (146 matches).

Before the model could begin training, the number of epochs, or number of times that model would repeatedly fully process the training data and adjust its biases to ensure a higher accuracy, needed to be decided on. While the guidelines provided by TensorFlow recommended the use of 40 epochs, experimental tuning of the model found that a model running on 20 epochs generally resulted in a higher accuracy model, and was therefore selected.

Following the completed design of the model in a Python script that communicated with the aforementioned spreadsheet used to collect match data, the model was trained and tested on the match dataset to determine its accuracy in predicting the outcome of LoL matches before they began. The full Python script used for this process can be found in Appendix C.

Findings

Following the creation of the neural network, the network was repeatedly regenerated until a model with higher than average accuracy was generated and recorded, as the somewhat random nature behind the process of training machine learning algorithms can lead to slight variations in prediction accuracy between models. Figures 3 and 4 show the performance of the selected model over the training and validation set as generated by TensorBoard, a data visualization tool provided alongside TensorFlow.

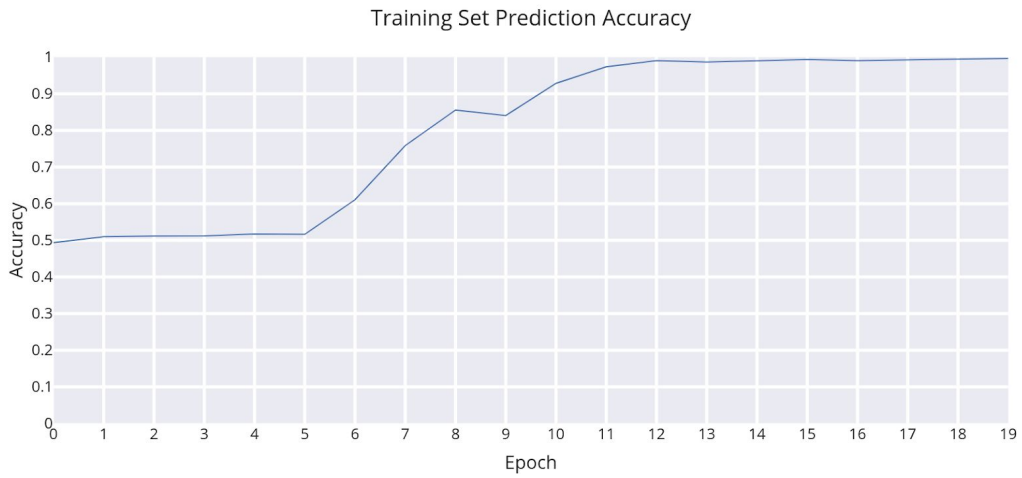

Figure 3 - Prediction accuracy of the model of the training set as the model was being trained on the training set.

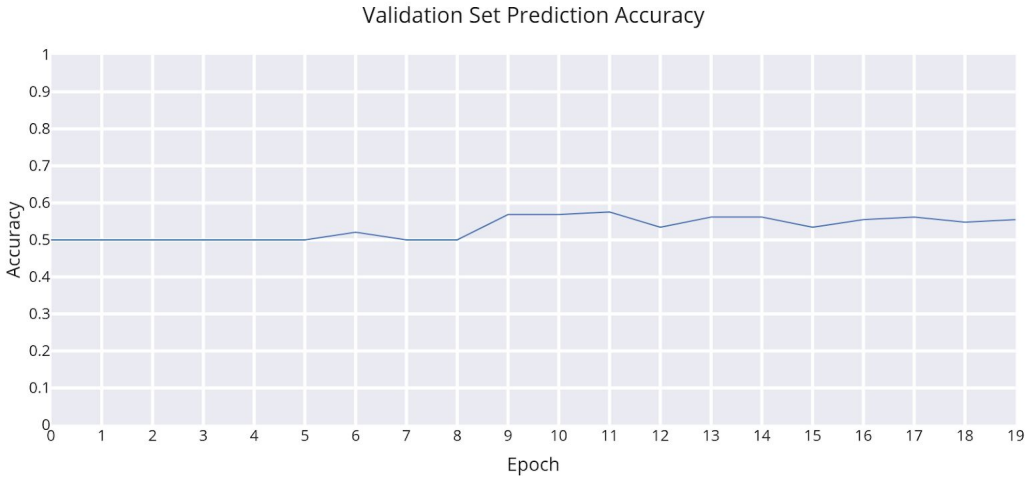

Figure 4 - Prediction accuracy of the model of the validation set as the model was being trained on the training set.

Figure 3 shows the performance of the model on the training set as it was being trained to make predictions on the training set. After 20 epochs, the model's prediction accuracy on the training set was approximately 99.64%, which appears to be a highly desirable outcome for the model. However, rather contrastingly, when computed alongside the training of the model, the accuracy of the validation was ultimately not nearly as successful in its predictions. As shown in

Figure 4, the performance of the model on the validation set never reached or exceeded 60%, and after 20 epochs of training, the model's prediction accuracy on the validation set was approximately 55.48%.

While the prediction accuracies of the model on the training and validation sets are good predictors of the model's success, the true performance of the model is computed by determining the prediction accuracy of the model on the testing set after it has completed its training. As such, after the training of the model was complete, the testing set was run against the trained model to determine an approximate prediction accuracy of 54.25% for the selected model.

Conclusions

Although a 54.25% prediction accuracy may seem to be a successful predictor of match outcomes, when compared to the fact that a 50% prediction accuracy is the expected outcome of randomly guessing which team would win, it becomes apparent, without the need for any statistical tests, that the neural network is both inconsistent and not a great predictor of match outcomes when its information is only limited to what matchmaking algorithm have access to. Specifically, by only ever predicting the blue team to win, a prediction accuracy of 50.87% could have been achieved for predictions over the entire dataset, only further proving the ineffectiveness of the neural network.

Given the low difference in accuracy the model and random guessing, it can be concluded that the consideration of additional player information in the LoL matchmaking algorithm beyond solely considering player rank would have very little, if any, noticeable impact on match quality. Furthermore, if the model were to implemented, the potential benefits would likely be outweighed by the costs of implementing a system to judge matches and system

upkeep, and should therefore not be implemented. Thus, the result of this study only somewhat aligns with the initial hypothesis. While it was correctly hypothesized that prediction accuracy would only sightly be better than random guessing, the results did disprove the hypothesis that one-sided matches would be easy to predict because of the relatively low prediction accuracy.

It should also be noted that the results of this study are not alone in concluding that predictions of LoL match outcomes using machine learning is ineffective. In similar studies about predicting LoL match outcomes using machine learning, Yin (2018) and Jagtap (2018) both cited only finding a prediction accuracy of slightly over 50% when only considering raw match data, reinforcing the conclusions found here. Additionally, both studies also mentioned a significant difference in the training and testing data set prediction accuracies, which can attributed to a lacking sample dataset. Together, these commonalities between studies looking to predict LoL match outcomes appears to reinforce the conclusion found in this study, being that the outcomes of LoL matches cannot be reliably predicted before they occur.

Limitations & Delimitations

However, despite the similarities this study shares with others on the same topic, there are some limitations that need to be addressed. Although a large number of data points were considered for each player, not every piece of information specific to each player was considered, in part due to physical restrictions of the Riot Games API, as some players have no historical ranked information. Additionally, the Riot Games API offers no way to retrieve a player's ranked information as it was at the time of any given match, so the neural network had to process some of the information after the result of the match it was predicting. That issue was only worsened by the fact that the Riot Games API and the Google Spreadsheets API both limit

the number of requests for information that can be made within a specific time frame, causing the initial match collection and preprocessing sections to take longer, which widened the gap between the player's current ranked statistics and what they were prior to the match. This likely caused inconsistencies between the data the network should have been using to make predictions and what it actually used. Most notably, due to the researcher's inexperience with machine learning and neural networks in general, it is very possible that some important considerations were overlooked at various points in the process, which may have been able to contribute to a more accurate model.

There are also delimitations of the study that need to be addressed. The most notable is that ranked information was the only match information considered in the data. While it may have provided better results for the neural network, it would present issues if the network were to be used to improve all matchmaking queues besides only ranked matches. Additionally, as addressed earlier, because there is no way to generate a truly random sample set of matches from the Riot Games API, some bias was introduced into the match data because the first match was manually selected by the researcher, which likely influenced the average rank of players in the dataset. It should also be noted that while the sample size of 2000 was ultimately chosen and shown to be suitable, a larger sample sample size could have been utilized in order to achieve marginally more realistic results.

Implications & Future Directions

With the results of this in consideration, it can be strongly concluded that neural networks cannot accurately predict the outcomes of LoL matches before they begin, barring any significant future developments. Despite the limitations of the study, the prediction accuracy of the model

being somewhat greater than 50% does point to the potential for future success if a similar study were to be applied to an online game outside of LoL, games of a different genre than LoL, or even applied to real-life games or sports. A more conclusive result can be determined regarding the applicability of machine learning in match predictions in general. Regardless of whether or not the prediction accuracies of those studies prove to be more or less effective than that found in this study, any further research may still be highly beneficial for both the fields of matchmaking predictions and machine learning as a whole.

References

Ari, A., Hanbay, D. (2018). Deep learning based brain tumor classification and detection system. Turkish Journal of Electrical Engineering & Computer Sciences, 26(5), 2275-2286.

Burlina, P., Billings, S., Joshi, N., & Albayda, J. (2017). Automated diagnosis of myositis from muscle ultrasound: Exploring the use of machine learning and deep learning methods. PLoS ONE, 12(8), 1–15.

Buscema, P. M., Massini, G., & Maurelli, G. (2015). Artificial Adaptive Systems to predict the magnitude of earthquakes. Bollettino Di Geofisica Teorica Ed Applicata, 56(2), 227-256.

Ding, S., Zhang, N., Xu, X., Guo, L., & Zhang, J. (2015). Deep extreme learning machine and its application in EEG classification. Mathematical Problems in Engineering, 2015, 1–11.

Grainger, A. T., Tustison, N. J., Qing, K., Roy, R., Berr, S. S., & Shi, W. (2018). Deep learning-based quantification of abdominal fat on magnetic resonance images. PLoS ONE, 13(9), 1–16.

Gutiérrez, G., Canul-Reich, J., Zezzatti, A. O., Margain, L., & Ponce, J. Mining: Students comments about teacher performance assessment using machine learning algorithms. International Journal of Combinatorial Optimization Problems & Informatics, 9(3), 26-40.

Jagtap, S. (2018). How We Trained a Machine to Predict the Winning Team in League of Legends. *Medium*. Retrieved from https://medium.com/trendkite-dev/machine-learning-league-of-legends-victory-predictions-8bc6cbc7754e

Panjan, A., Šarabon, N., & Filipčič, A. (2010). Prediction of the successfulness of tennis players

with machine learning methods. Kinesiology, 42(1), 98-106.

Riot Games Support Staff (2019). Matchmaking Guide. *Riot Games Support*. Retrieved from https://support.riotgames.com/hc/en-us/articles/201752954

Sinthupundaja, J., Chiadamrong, N., & Suppakitjarak, N. (2017). Financial prediction models from internal and external firm factors based on companies listed on the stock exchange of Thailand. Suranaree Journal Of Science & Technology, 24(1), 83-98.

Sturrock, H. J. W., Woolheater, K., Bennett, A. F., Andrade-Pacheco, R., & Midekisa, A. (2018). Predicting residential structures from open source remotely enumerated data using machine learning. PLoS ONE, 13(9), 1–10.

TensorFlow. (2019). Retrieved from https://www.tensorflow.org

Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep learning for computer vision: A brief review. Computational Intelligence & Neuroscience, 1–13.

Wunderlich, F., & Memmert, D. (2018). The Betting Odds Rating System: Using soccer forecasts to forecast soccer. PLoS ONE, 13(6), 1–18.

Yin, J. (2018). League of Legends: Predicting Wins In Champion Select With Machine Learning. *Hackernoon*. Retrieved from https://hackernoon.com/league-of-legends-predicting-wins-in-champion-select-with-machine-learning-6496523a7ea7

Appendix A

```
# Initial Match Data Collection Python Script written using the PyCharm IDE

import time
import gspread
from oauth2client.service_account import ServiceAccountCredentials
from riotwatcher import RiotWatcher, ApiError

def get_sheet():
	# GSpread Setup
	scope = ['https://spreadsheets.google.com/feeds',
		 'https://www.googleapis.com/auth/drive']
	credentials = ServiceAccountCredentials.from_json_keyfile_name('[REDACTED].json', scope)
	gc = gspread.authorize(credentials)
	return gc.open_by_key('[REDACTED]').sheet1

def get_riot_watcher():
	# Riot Watcher Setup
	rg_api_key = '[REDACTED]'
	return RiotWatcher(rg_api_key)

def get_match_data(match_id):
	try:
		return watcher.match.by_id('na1', match_id)
	except ApiError as err:
		if err.response.status_code == 429:
			print('We should retry in {} seconds.'.format(err.headers['Retry-After']))
			print('this retry-after is handled by default by the RiotWatcher library')
			print('future requests wait until the retry-after time passes')
		elif err.response.status_code == 404:
			print('Summoner with that ridiculous name not found.')
		else:
			print('Some error with the watch API occurred. Error Code: ' + str(err))
			raise
		time.sleep(50)
		get_match_data(match_id)

def increment_current_match_row():
	global current_match_row

	time.sleep(1)
	sheet.update_acell('B1', int(sheet.acell('B1').value) + 1)
	time.sleep(1)
	current_match_row = int(sheet.acell('B1').value)

def get_current_match_row():
	time.sleep(1)
	return int(sheet.acell('B1').value)

def log_match_data(match):
	global current_match_row
```

```
        #Store the match ID
        time.sleep(1)
        sheet.update_cell(current_match_row, 1, str(match['gameId']))

        #Store the outcome of the match
        time.sleep(1)
        if match['teams'][0]['win'] == 'Win':
                sheet.update_cell(current_match_row, 2, str('Red'))
        else:
                sheet.update_cell(current_match_row, 2, str('Blue'))

        #Store the raw match data in case it is needed later
        time.sleep(1)
        sheet.update_cell(current_match_row, 3, str(match))

        #Store the starting time of the match
        time.sleep(1)
        sheet.update_cell(current_match_row, 4, match['gameCreation'])

        #Store all the ID's of all the players in the match
        time.sleep(10)
        for i in range (0, 10):
                sheet.update_cell(current_match_row, 5 + i,
str(match['participantIdentities'][i]['player']['accountId']))

sheet = get_sheet()
watcher = get_riot_watcher()
current_match_row = get_current_match_row()

repeat_counter = 0
print('Starting...')

ignore_previous_match_id = False
while(current_match_row <= 2002):
        if(current_match_row <= 3):
                match = get_match_data(sheet.acell('D1').value)
                log_match_data(match)
                current_match_row = 4
                increment_current_match_row()
        else:
                print('Current Match Row: ' + str(current_match_row))

                player_counter = 0
                if not ignore_previous_match_id:
                        previous_match_id = sheet.cell(current_match_row - 1, 1).value
                found_next_match = False

                for player_counter in range(0, 10):
                        if not found_next_match:
                                try:
                                        # Queue 420 = Ranked Solo
```

```
                                        matchlist = watcher.match.matchlist_by_account('na1',
sheet.cell(current_match_row - 1, player_counter + 5).value, 420)

                                        for match in matchlist['matches']:
                                                if (int(match['gameId']) != int(previous_match_id)) and
(not (str(match['gameId']) in sheet.col_values(1))):
                                                        if ignore_previous_match_id:
                                                                current_match_row =
get_current_match_row()
                                                        log_match_data(watcher.match.by_id('na1',
match['gameId']))
                                                        increment_current_match_row()
                                                        found_next_match = True
                                                        repeat_counter = 0
                                                        break
                                        except ApiError as err:
                                                if err.response.status_code == 429:
                                                        print('We should retry in {}
seconds.'.format(err.headers['Retry-After']))
                                                        print('this retry-after is handled by default by the
RiotWatcher library')
                                                        print('future requests wait until the retry-after time
passes')
                                                elif err.response.status_code == 404:
                                                        print('Matches of that queue not found for the player id
\"' + str(sheet.cell(current_match_row - 1, player_counter + 5).value) + '\". Moving onto the next
player.')
                                else:
                                        break
                        repeat_counter += 1

                        if repeat_counter > 1:
                                print('No suitable matches found from previous players. Attempting to find
another match using previous match data.')

                                ignore_previous_match_id = True
                                current_match_row -= 10
```

Appendix B

```
# Data Preprocessing and Formatting Python Script written using the PyCharm IDE

import time
import gspread
from oauth2client.service_account import ServiceAccountCredentials
from riotwatcher import RiotWatcher, ApiError

def get_sheet():
	# GSpread Setup
	scope = ['https://spreadsheets.google.com/feeds',
		 'https://www.googleapis.com/auth/drive']
	credentials = ServiceAccountCredentials.from_json_keyfile_name('[REDACTED].json', scope)
	gc = gspread.authorize(credentials)
	return gc.open_by_key('[REDACTED]').sheet1

def get_riot_watcher():
	# Riot Watcher Setup
	rg_api_key = '[REDACTED]'
	return RiotWatcher(rg_api_key)

def increment_current_match_row():
	global current_match_row

	time.sleep(1)
	sheet.update_acell('Q1', int(sheet.acell('Q1').value) + 1)
	time.sleep(1)
	current_match_row = current_match_row + 1

def get_current_match_row():
	time.sleep(1)
	return int(sheet.acell('Q1').value)

def format_current_match_data(match_data):
	output_string = ''

	# match start time
	output_string += str(match_data['gameCreation']) + ','

	# team and role of each player in order
	match_participants = match_data['participants']
	for participant in match_participants:
		output_string += str(participant['teamId']) + ','

		if participant['timeline']['role'] == 'SOLO' and participant['timeline']['lane'] == 'TOP':
			output_string += '1,'
		elif participant['timeline']['role'] == 'NONE' and participant['timeline']['lane'] == 'JUNGLE':
			output_string += '2,'
		elif participant['timeline']['role'] == 'SOLO' and participant['timeline']['lane'] == 'MIDDLE':
			output_string += '3,'
```

```
                elif participant['timeline']['role'] == 'DUO_CARRY' and participant['timeline']['lane'] ==
'BOTTOM':
                        output_string += '4,'
                elif participant['timeline']['role'] == 'DUO_SUPPORT' and participant['timeline']['lane'] ==
'BOTTOM':
                        output_string += '5,'
                else:
                        output_string += '0,'

        return output_string

def format_previous_match_data(match_data, players_id_in_match, found_match):
        output_string = ''
        if found_match:
                player_data = match_data['participants'][players_id_in_match - 1]

                # general inforamtion about the game
                output_string += str(match_data['gameCreation']) + ','
                output_string += str(match_data['gameDuration']) + ','
                output_string += str(match_data['queueId']) + ','
                output_string += str(match_data['mapId']) + ','

                # whether or not the player won the game and their champion
                output_string += '1,' if player_data['stats']['win'] == 'true' else '0,'
                output_string += str(player_data['championId']) + ','

                # player stats
                output_string += str(player_data['stats']['kills']) + ','
                output_string += str(player_data['stats']['deaths']) + ','
                output_string += str(player_data['stats']['assists']) + ','
                output_string += str(player_data['stats']['largestKillingSpree']) + ','
                output_string += str(player_data['stats']['largestMultiKill']) + ','
                output_string += str(player_data['stats']['killingSprees']) + ','
                output_string += str(player_data['stats']['longestTimeSpentLiving']) + ','
                output_string += str(player_data['stats']['totalDamageDealt']) + ','
                output_string += str(player_data['stats']['totalDamageDealtToChampions']) + ','
                output_string += str(player_data['stats']['totalHeal']) + ','
                output_string += str(player_data['stats']['damageSelfMitigated']) + ','
                output_string += str(player_data['stats']['damageDealtToObjectives']) + ','
                output_string += str(player_data['stats']['visionScore']) + ','
                output_string += str(player_data['stats']['totalDamageTaken']) + ','
                output_string += str(player_data['stats']['goldEarned']) + ','
                output_string += str(player_data['stats']['totalMinionsKilled']) + ','
                output_string += str(player_data['stats']['neutralMinionsKilled']) + ','
                output_string += str(player_data['stats']['totalTimeCrowdControlDealt']) + ','
                output_string += str(player_data['stats']['champLevel']) + ','

                '''
                # since not all matches will have deltas or the same number of deltas, they will be
exlcuded to avoid potential issues
                # player's 'timeline' stats
                output_string += str(player_data['timeline']['creepsPerMinDeltas']['0-10']) + ','
                output_string += str(player_data['timeline']['xpPerMinDeltas']['0-10']) + ','
```

```
                output_string += str(player_data['timeline']['goldPerMinDeltas']['0-10']) + ','
                output_string += str(player_data['timeline']['damageTakenPerMinDeltas']['0-10']) + ','
                '''
        else:
                # fills in 0 for all 25 datapoints if the player's previous match was not very recent
                for i in range(0, 25):
                        output_string += '0,'

        return output_string

def format_ranked_stats_data(input_stats_data):
        output_string = ''

        try:
                # for some reason, the input data comes in as a list with the dict we want as a single
entry?
                stats_data = input_stats_data[0]

                tier_dict = {'IRON': '1', 'BRONZE': '2', 'SILVER': '3', 'GOLD': '4', 'PLATINUM': '5',
'DIAMOND': '6'}
                if str(stats_data['tier']) in tier_dict:
                        output_string += tier_dict[str(stats_data['tier'])] + ','
                else:
                        output_string += '0,'

                rank_dict = {'I': '1', 'II': '2', 'III': '3', 'IV': '4'}
                if str(stats_data['rank']) in rank_dict:
                        output_string += rank_dict[str(stats_data['rank'])] + ','
                else:
                        output_string += '0,'

                output_string += str(stats_data['leaguePoints']) + ','
                output_string += str(stats_data['wins']) + ','
                output_string += str(stats_data['losses']) + ','
                output_string += '0, ' if stats_data['veteran'] == 'False' else '1,'
                output_string += '0, ' if stats_data['inactive'] == 'False' else '1,'
                output_string += '0, ' if stats_data['freshBlood'] == 'False' else '1,'
                output_string += '0, ' if stats_data['hotStreak'] == 'False' else '1,'
        except:
                print('\tError occured when handling a player\'s ranked stats. Logging all 0s and
continuing...')
                output_string = ''
                for i in range(0, 9):
                        output_string += '0,'
        return output_string

def log_formatted_data(formatted_data):
        global current_match_row
        sheet.update_cell(current_match_row, 17, formatted_data)

        # Remove old matches after we have processed their info
        if current_match_row > 52:
```

```
                    sheet.update_cell(current_match_row, 3, 'Match data removed from sheet to prevent
data overload issues.')

            # Delay to ensure that API rates are not passed.
            print('\tDelaying before next match...')
            time.sleep(25)

sheet = get_sheet()
watcher = get_riot_watcher()
current_match_row = get_current_match_row()

unexpected_api_error_occured = False

print('Starting...')
if(current_match_row <= 2):
            current_match_row = 3
while current_match_row <= 2002 and not unexpected_api_error_occured:
            print('Current Match Row: ' + str(current_match_row))

            # Check the stored match id. If it is 0, it played before patch 9.4, and it should be ignored.
            current_match_id = sheet.cell(current_match_row, 1).value
            if current_match_id != '0':
                        formatted_output = ''
                        current_match_start_time = sheet.cell(current_match_row, 4).value

                        current_match_player_list = sheet.range(current_match_row, 5, current_match_row,
14)

                        # Store everything important from the current match
                        formatted_output += format_current_match_data(watcher.match.by_id('na1',
current_match_id))

                        for player_counter in range(0, 10):
                                    players_account_id = current_match_player_list[player_counter].value

                                    try:
                                                # Get the match that the player has played before the stored ranked
match
                                                players_previous_match_from_matchlist =
watcher.match.matchlist_by_account(region='na1', encrypted_account_id=players_account_id,
queue=None, end_time=current_match_start_time, begin_time=str(int(current_match_start_time) -
604800000))['matches'][0]

                                                players_previous_match = watcher.match.by_id(region='na1',
match_id=players_previous_match_from_matchlist['gameId'])

                                                # Find what their id is relative to that match and get their summoner ID
                                                players_summoner_id = None
                                                players_id_in_match = None
                                                for participantIdentity in players_previous_match['participantIdentities']:
                                                            if participantIdentity['player']['accountId'] ==
players_account_id:
```

```
                                                players_summoner_id =
participantIdentity['player']['summonerId']
                                                players_id_in_match =
participantIdentity['participantId']
                                                break

                                        # Log that previous match information given the above
                                        formatted_output +=
format_previous_match_data(players_previous_match, players_id_in_match, True)

                                except ApiError as err:
                                        if err.response.status_code == 404:
                                                # API Error code 404: no matches found within the time frame
                                                print('\t404 error from finding a previous match. Logging all 0s
for it...')
                                                formatted_output += format_previous_match_data(False,
False, False)
                                        else:
                                                unexpected_api_error_occured = True
                                                print('Unexpected API error occured! Stopping script. Please
find a way to get around this case.')
                                                print('Error Status Code: ' + str(err.response.status_code))
                                                break

                                # Find and log their specific ranked information. This will return ranked
information from multiple queues, so only take the one for
                                players_ranked_information =
watcher.league.positions_by_summoner(region='na1',
encrypted_summoner_id=players_summoner_id)

                                formatted_output += format_ranked_stats_data(players_ranked_information)

                        if not unexpected_api_error_occured:
                                # Log the formatted data in the spreadsheet, and increment the row counter.
                                log_formatted_data(formatted_output.strip(','))
                                increment_current_match_row()
                else:
                        print('\tMatch played on a patch other than 9.4. Moving to next match.')
                        increment_current_match_row()
```

Appendix C

```
# Algorithm Creation and Design Python Script written using the PyCharm IDE
# Neural Network Setup Guidelines Provided By:
# https://www.tensorflow.org/tutorials/keras/basic_text_classification

from __future__ import absolute_import, division, print_function

import tensorflow as tf
from tensorflow import keras
from keras.callbacks import TensorBoard
import gspread
from oauth2client.service_account import ServiceAccountCredentials
import numpy as np

print('TF Version: ' + tf.__version__)


# Download the Riot Dataset #################
print('\tDownloading Test Dataset...')
def get_sheet():
    # GSpread Setup
    scope = ['https://spreadsheets.google.com/feeds',
             'https://www.googleapis.com/auth/drive']
    credentials = ServiceAccountCredentials.from_json_keyfile_name('[REDACTED]',scope)
    gc = gspread.authorize(credentials)
    return gc.open_by_key([REDACTED]).sheet1
sheet = get_sheet()
match_ids = sheet.range(3, 1, 2003, 1)
input_label_list_from_sheet = sheet.range(3, 2, 2003, 2)
input_data_list_from_sheet = sheet.range(3, 17, 2003, 17)
input_label_list = []
input_data_list = []
for i in range(2000):
    if int(match_ids[i].value) != 0:
        input_label_list.append('0' if input_label_list_from_sheet[i].value == 'Red' else '1')

        input_data_list.append(input_data_list_from_sheet[i].value.split(','))
input_labels_doc = open('input_labels.txt', 'w')
input_labels_doc.write(str(input_label_list))
input_labels_doc.close()
input_data_doc = open('input_data.txt', 'w')
input_data_doc.write(str(input_data_list))
input_data_doc.close()


# Prepare the Data ###########################
print('\tPrepare the Data...')

largest_value = 0
for dataset_index in range(len(input_data_list)):
    for datapoint_index in range(len(input_data_list[dataset_index])):
```

```
        input_data_list[dataset_index][datapoint_index] =
int(input_data_list[dataset_index][datapoint_index])
        input_label_list[dataset_index] = int(input_label_list[dataset_index])
        if input_data_list[dataset_index][datapoint_index] > 1550741400000:
            input_data_list[dataset_index][datapoint_index] =
(input_data_list[dataset_index][datapoint_index] - 1550741400000)/3600000
        if input_data_list[dataset_index][datapoint_index] > largest_value:
            largest_value = input_data_list[dataset_index][datapoint_index]

input_data_np_array = np.array(input_data_list)
input_label_np_array = np.array(input_label_list)

train_data = keras.preprocessing.sequence.pad_sequences(input_data_list[0:1400], maxlen=361,
value=0, padding='post')
test_data = keras.preprocessing.sequence.pad_sequences(input_data_list[1400:1800], maxlen=361,
value=0, padding='post')
val_data = keras.preprocessing.sequence.pad_sequences(input_data_list[1800:], maxlen=361,
value=0, padding='post')


# Build The Model ############################
print('\tBuild The Model...')
model = keras.Sequential()
model.add(keras.layers.Embedding(largest_value + 1, 8))
model.add(keras.layers.GlobalAveragePooling1D())
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(8, activation=tf.nn.relu))
model.add(keras.layers.Dense(1, activation=tf.nn.sigmoid))
model.summary()
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])

tbCallback = keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, write_graph=True,
write_grads=False,
                                write_images=False, embeddings_freq=0, embeddings_layer_names=None,
                                embeddings_metadata=None, embeddings_data=None,
update_freq='epoch')


# Create a Validation Set ####################
print('\tCreate a Validation Set...')
train_labels = input_label_np_array

x_val = val_data
partial_x_train = train_data[0:1400]
y_val = train_labels[1800:]
partial_y_train = train_labels[0:1400]


# Train the Model ############################
print('\tTrain the Model...')
history = model.fit(partial_x_train, partial_y_train, epochs=20, validation_data=(x_val, y_val),
                    verbose=2, shuffle=True, callbacks=[tbCallback])
```

```
# Evaluate the Model ########################
print('\tEvaluate the Model...')
results = model.evaluate(test_data, input_label_np_array[1400:1800])
print(results)

red_win_counter = 0.0
for result in input_label_list:
    if result == 1:
        red_win_counter += 1
print("\nRed Win Percentage: ")
print((red_win_counter/len(input_label_list)))
print("Blue Win Percentage: ")
print(((len(input_label_list) - red_win_counter)/len(input_label_list)))
```