Predicting Outcomes of *League of Legends* Matches

Tung Nguyen
CSP 554
Illinois Institute of Technology
tnguye48@hawk.iit.edu

James Guerrera-Sapone
CSP 554
Illinois Institute of Technology
jguerre5@hawk.iit.edu

Abstract - *League of Legends* (LoL) is one of the most popular online video games in the world. A game match contains 2 teams of 5 players and each player controls a character with a unique set of abilities. This means that there are many factors which can influence which team wins. There are influences both from the characters being used as well as the individual skill of the players. We have created a data pipeline using Microsoft Azure to process data from the LoL developer API and created a machine learning model to predict the winning team based both off of the player and character compositions of each team.

I. INTRODUCTION

League of Legends is a MOBA (Multiplayer online battle arena) game developed by Riot Games where even minor differences can make a big impact on the outcome of the game. It is one of the most played video games in the world and in 2016 it had 100 million monthly players [14]. The game consists of two 5 person teams where each player controls 1 of 146 champions. Every champion has strengths and weaknesses with unique interactions between them. Champions are picked in a draft format with each team having the ability to remove 5 champions from the available pool before picks are made. Champion picks are unique within a game. This means that the composition of the two teams plays a huge role in the outcome of the match. However, the players themselves also have a large impact. Player skill is measured both by a visible rank as well as a hidden matchmaking rating which is used to create balanced matches. There is also an internally calculated mastery score that every player has for each champion. This score measures how much and how well a player plays a champion.

The LoL developer API gives us the ability to query thousands of matches. We have access to all of the players in a match, their personal statistics and the champions that they played. We fit a model to predict which team will win a game based off of the players and champions on each team. We harnessed the power of Riot's developer API as well as Microsoft Azure to build and deploy a machine learning model that predicts the winning team. However, fitting the most complex model that we developed on all of the data was too computationally expensive, so we also created a simpler model which could utilize all of the data rather than a subset of it. Azure HDInsight allowed us to use Apache Spark and Spark ML for our data processing and the building of the model and allowed us to use CosmosDB to store our data [6]. However, during the process of building the pipeline, we found that HDInsight limited our options for deploying the model to the web app, so we adopted another spark platform called Databricks. Deploying the app through Databricks also required the use of Azure Machine Learning [13]. We also decided to create models using both SparkML and Scikit learn to see the differences between the two of them. The web app itself was created using Flask and HTML.

II. RELATED WORK

Given the rise of powerful machine learning libraries and big data, video games are an excellent source for researchers to test and implement their findings. For each LoL game, each player chooses a champion from the champion pool. Combined with a number of items and runes (additional pre-game choices which can influence the game) each champion can buy, each League game has **32 billion** possible team compositions. Hence, predicting the outcome of a LoL match poses an interesting challenge for big data algorithms. Many attempts to this problem have been made. Felton et al. [12] uses a simple 3-layer neural network to predict the outcome of a match with an 80% accuracy. The project's data consisted of 1781 games. The input features for each game were the human players statistics (win rate, champion mastery points), the champion statistics (win rate across the whole server, champion tier), and the sum of all those input values. Jihan Yin [9] compares prediction results across multiple machine learning models by using a dataset of 1700 matches. Each player-champion pair had 14 features: 5 champion statistics and 9 player statistics. In total, there were 140 input features for each match in the feature matrix. The best results achieved were ~63% in sensitivity by using Gradient Boosting. The article concludes that players statistics are more predictive than champion statistics, that the AD-carry role had the least importance in impacting the game, and that a match cannot be decided purely based on champion select.

Defense of the Ancients (DotA) is a similar game to LoL. They are both champions-based MOBA games and DotA is often considered a more complex game than LoL. Wang [18] built prediction models based on 5,071,858 match records and introduced game length as an additional input feature. The author concluded that Neural Networks were not better than Logistic Regression (around 61% accuracy) and that the game length feature had a small positive impact on the performance. Conley et al. [4] built a champion select recommendation engine by using a greedy search on all possible matchups. The winning probability of each matchup was calculated through a Logistic Regression (69.8% accuracy) or a K-Nearest Neighbors (67.43% accuracy) model. Kalyanaraman's work [8] introduces a graph-based approach to predict the outcome of the games. Though the results were promising, we chose not to follow this approach due to the complexity of graph algorithms and the purpose of the project.

Our project will follow the suggestions from these other works. Simple Logistic Regression is a good option since it offers both low time complexity and comparable results. For each match, the target will be a binary indicator showing whether the blue team won or not. The features for each instance are a combination of each player's historical performance measurements and the champion statistics. To perform machine learning on Hadoop, we will make use of Spark ML, Scikit-Learn and Jupyter Notebooks [1]. In our interactive web program, the user will enter the usernames, champions, and roles of all 10 players from both sides. The program will return the winning probability of the blue side (team 100), which is the posterior probability $P(y = 1|x)$ of the classifier.

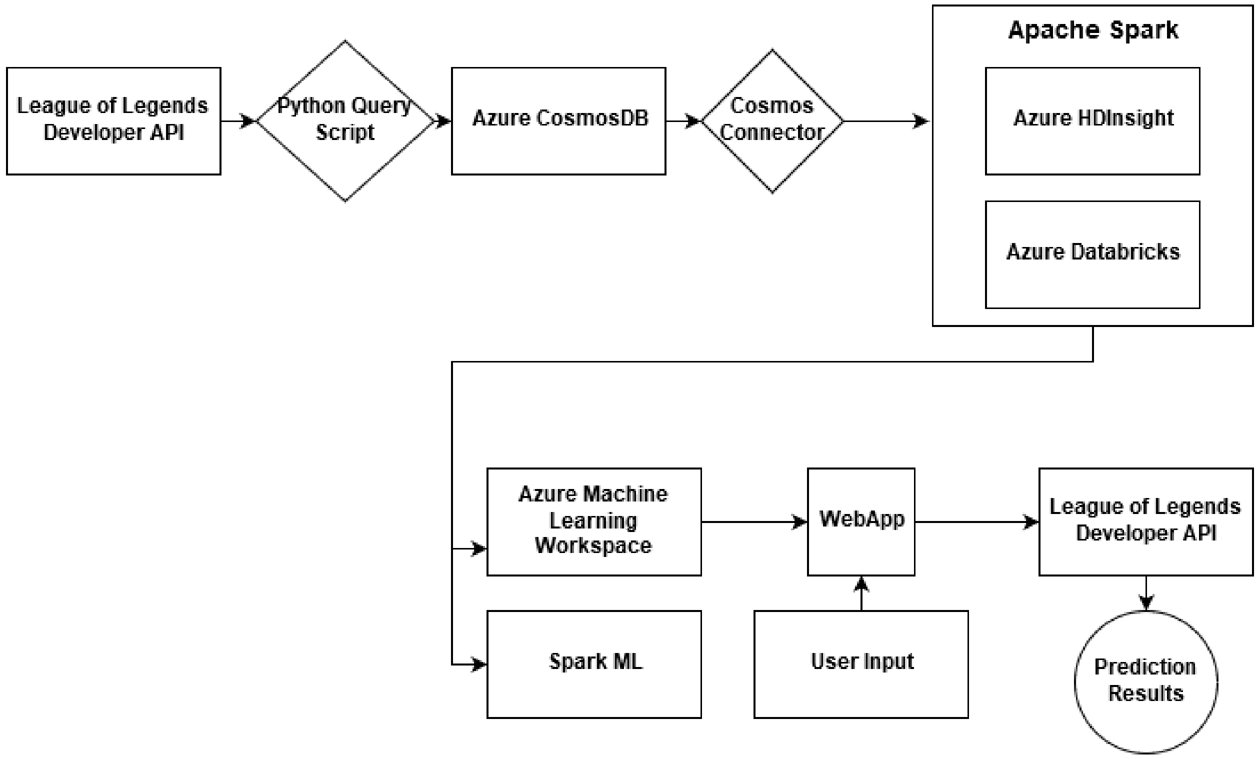


Figure 1: Overview of our pipeline

III. METHODOLOGY

A. Why Azure?

Azure is a platform-as-a-service product that we were already familiar with which meant that it was simply the fastest way to get started with the project. By doing everything within Azure all of the connections between the various pieces of the pipeline were done for us. We chose to use Azure's ecosystem because it saved us a significant amount of overhead. It was also advantageous to have a single hub where we could monitor all of the different resources that we were using.

B. Data Collection

The first step of the project was to query data from Riot Games API onto a database so that we could harness the power of distributed computing to process it. We decided to store the data in Azure CosmosDB. CosmosDB provided an efficient method for storing data. One of the most attractive features of Cosmos is that it is schema agnostic. Each database in Cosmos consists of a collection or ***Container*** [2]. Each Container stores data based on keys called ***ids***. Data sent to each Container is accessed by these *ids.* For our project, the final processed data comes in the form of Python dictionaries with distinct keys for each object. Because Cosmos handles data on keys, we did not have to worry about managing the schema of our data. All we had to do was to create a dictionary with our predefined id and send it to our desired database. Since our raw data came in the form of JSON objects, it is best to utilize CosmosDB for our project compared to Azure Blob Storage[20].

For our project, we needed to get 3 separate contents from the developer API: champion, summoner (what players are referred to as internally), and match data. The first went into a Cosmos container called Champions, which contained base statistics for each champion in LoL. Riot Games provides a static JSON file called *Champion*. It is a file with all of the necessary statistics (damage per level, hp per level, etc.,) for all 146 champions in the game. For each champion, we extract its statistics from the file, summarize them into a dictionary and send the dictionary to the *Champions Container* on Cosmos, keying by championId.

The second Container we created was the ***Players Container*** (players_final), which consists of all of the player data. To collect player data, we utilize Riot Games API to collect a list of players of a certain rank in the game (players' mastery are classified by ranks in the game). We called these players *seed players*, as a separation from *secondary players* which is explained later in the paper. For each player in this list, we query their data and the ids of their previous 10 matches. A player dictionary will then be created and sent to the Players Container, the key is the summoner (player) id.

The last container we use is the ***Matches Container*** (matches_final). Match data is queried from the Riot API based on the match ids collected from *seed players*. The returned JSON object will be the overall match statistics and each player's performance statistics during that match. The object will then be sent to the container, keying by matchId. However, because in our final machine learning model, we predict a match outcome based on the history performance of all 10 players, we will need to collect more data. For now, we only have data of the seed players and their corresponding 10 previous matches' history, called *seed matches*. For each of the collected *seed matches*, we need to collect the history data of the remaining 9 users, we called these users *secondary players*. In order to have the average statistics of the secondary players, we collect the data from their previous 3 matches, called the *secondary matches.* All secondary data will be sent to Matches and Players Container, residing in the same place as the seed data. An overview of the data collection process is visualized in *Figure 2*.

The data collection script was run on a Ubuntu virtual machine deployed on Azure.

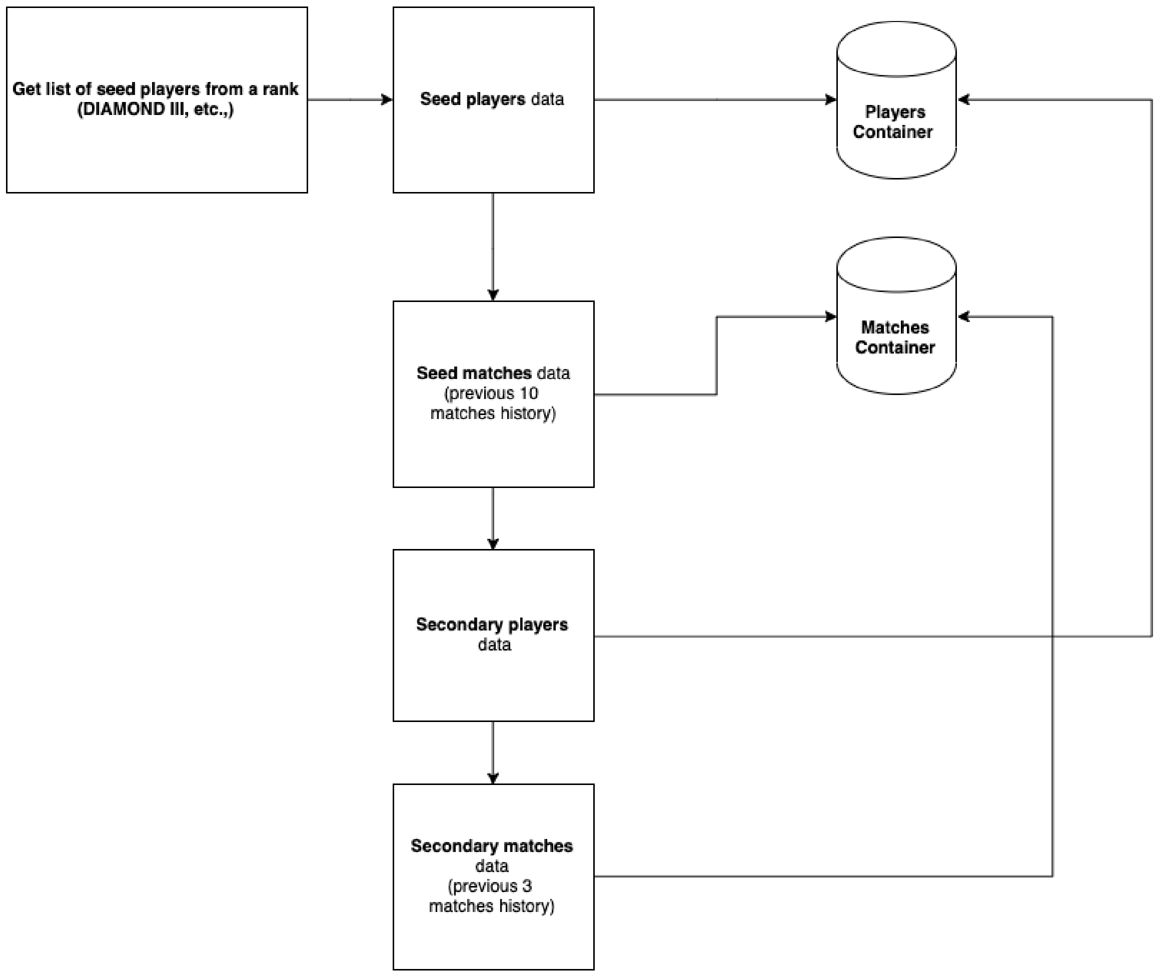

Get list of seed players from a rank (DIAMOND III, etc.,)
Seed players data
Players Container
Seed matches data (previous 10 matches history)
Matches Container
Secondary players data
Secondary matches data (previous 3 matches history)

Figure 2: Overview of data collection process.

C. Data Processing

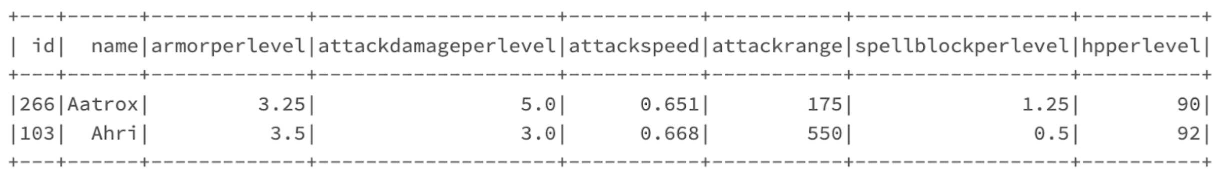
Once the data is on Azure, we begin the process of data cleaning, feature extraction and feature engineering. Spark is the ideal technology for the processing of our data. The most important thing is that it is scalable. LoL is one of the most popular video games in the world which means there are an immense number of games being played every day. The scalability of Spark means that we can choose to collect a huge amount of data and still get good performance. Spark also offers access to other powerful tools which give us access to machine learning libraries. By doing our data processing on Spark, we made it easier to transition into the rest of the project. One of the largest benefits of Spark is that it is easy to set up and use. Traditionally, distributed computing has many challenges associated with it. Compared to MapReduce, Spark runs upwards of 10 to 100 times faster [19] and writing the code is much simpler, as Spark handles a lot of the work behind the scenes. It is also more flexible than MapReduce which is limited to a strict format.

We used an **Azure HDInsight Spark** cluster and **Azure Databricks** for our data processing. HDInsight provides an easily available and easy to configure Spark cluster which has a well documented connector to CosmosDB which allowed for both reads and writes [6]. The Spark cluster got the data from Cosmos using the CosmosDB to HDInsight connector which allowed us to directly load data from *Containers* into Spark DataFrames [16]. From here we could analyze and extract the features that we needed for our model through the built-in **Jupyter Notebook** functionality. In this project, we use HDInsight to create a PySpark's dataframe features matrix for the machine learning model. **Databricks**, with its ability to connect to the **Azure Machine Learning Workspace**, is used to collect the features matrix produced by HDInsight and perform logistic regression on the matrix.

For our feature engineering process, first, we create a champion table. This is simply the process of querying the data from the ***Champions Container***. The champion DataFrame is stored under the name *mast_champ_join.* Overview of the DataFrame is shown in Table 1.

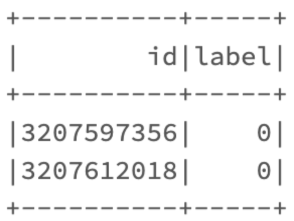
Second, we query match data from the ***Match Container***. One thing that we had to filter out of the data was a special case for LoL matches. Typically in LoL, the only way that a game ends is when one team defeats the other, however, sometimes a player fails to ever connect to the game. In these cases, the game can be ended early, at the 3 minute mark, and the match is effectively a draw. These matches were irrelevant for our data as they contain no meaningful stats due to their short length and lack of a definitive conclusion. Since the draw always happens at roughly the same time every match, it was trivial to filter out games which had a duration value less than 7 minutes. The reason 7 minutes was chosen rather than 3 is quite simple. The team which has a disconnected player gets to choose whether they would like to end the match early or play it out, which might be done because the missing player is simply connecting late and the team doesn't mind the temporary disadvantage. They have a time window in which they can vote to end the game or not. 7 minutes is the minimum amount of time that a LoL match can last and still be counted for the purposes of in-game reward systems and is comfortably past the point of the vote to end early, so it always yields only the relevant matches.

For each match, we collect the match feature that describes which team won and save it to a DataFrame. However, the match result stored in the raw data was very unintuitive, so we had to engineer it into a more reasonable 0 or 1 binary feature. It was originally stored as a string with "WIN" or "FAIL" as the values. There was also a win feature for each team despite this being redundant since only one team can win and the other must lose, which meant that we had an array with these string values in it instead of 1 binary feature like we wanted for our target variable. The team in raw data is called 100 (Blue) and 200 (Red). Our label, in the end, is a binary variable which indicate whether Blue team (100) wins (0 if lose and 1 if win). The data is then saved in a 2 column DataFrame called *label_col.*



```
+---+------+-------------+------------------+-----------+-----------+-----------------+----------+
| id|  name|armorperlevel|attackdamageperlevel|attackspeed|attackrange|spellblockperlevel|hpperlevel|
+---+------+-------------+------------------+-----------+-----------+-----------------+----------+
|266|Aatrox|         3.25|               5.0|      0.651|        175|             1.25|        90|
|103|  Ahri|          3.5|               3.0|      0.668|        550|              0.5|        92|
+---+------+-------------+------------------+-----------+-----------+-----------------+----------+
```

Table 1. Champion table mast_champ_join, id is champion id



```
+----------+-----+
|        id|label|
+----------+-----+
|3207597356|    0|
|3207612018|    0|
+----------+-----+
```

Table 2. Label table label_col, id is match id

The final matrix was to include each player's average stats for the games in their match history, and the base stats for each champion. Since we used PySpark, we broke the DataFrame creation process into multiple small steps. Each step would output one matrix. The output matrices would then be joined together to create the final player history statistics matrix. Spark structured data in a table format, so the process was much more complicated than we anticipated. For example, within a match, each player was referred to by a participant ID ranging from 1 to 10 rather than their unique player ID. The player ID was bound to the participant ID in a different nested struct from the one that contained all of the statistics. The process to create final stats table for each player is described below:

i. First, for each match in the *Match Container,* we query the list of summonerIds, championIds, lanes, roles and game statistics for all players in the game. We save this in a numpy array called *stats_np*.

ii. For each match in the *Match Container,* we collect the match duration (in minutes) and match id. We save these values in 2 lists called *matchIds_list* and *matchDur_list*.

iii. We combine *stats_np*, *matchIds_list and matchDur_list* to create a statistic matrix for each match. The matrix is called *match_sum_stats_agg_df* and its schema is presented in *Table 3*.

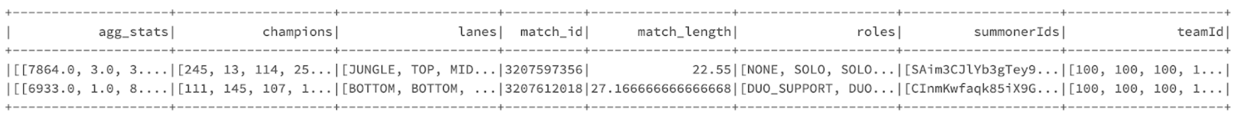
iv. Notice that in this matrix, data in some columns came in the form of an array with length 10. This is the aggregated data of all 10 players in a game. The data is combined due to how Spark handles nested dictionaries. Each value in the array is the stats belong to one player in the game, the index in the array corresponds to the participantId in the game.

v. We then expanded this matrix so each row consists of 1 value for each of those aggregated columns. Because each index corresponds to 1 player, we expand the matrix by looping the zipped list of aggregated columns. Moreover, we will clean the role, lane and team columns, and combine them together. An example of this expanded matrix is shown in Table 4. The expanded matrix is called *match_sum_stats*.

vi. The stats will then be normalized by game duration in minutes.

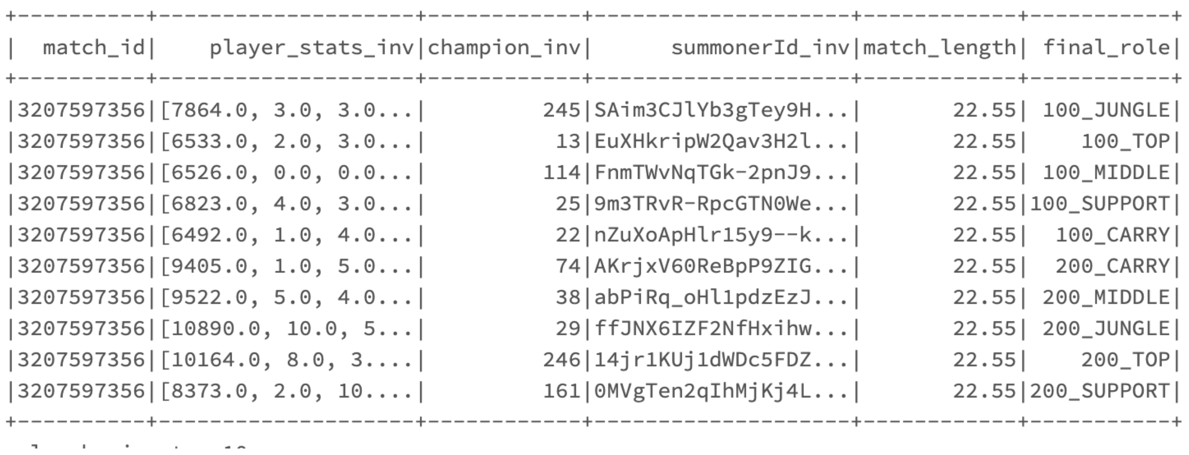
vii. Lastly, we will expand the *player_stats_inv* column into multiple columns for easier data manipulation. Each statistic will be assigned a name. .

viii. We then average the statistics column by grouping the expanded matrix by summonerIds. The final table for average player is called *summoner_avg_stats.*



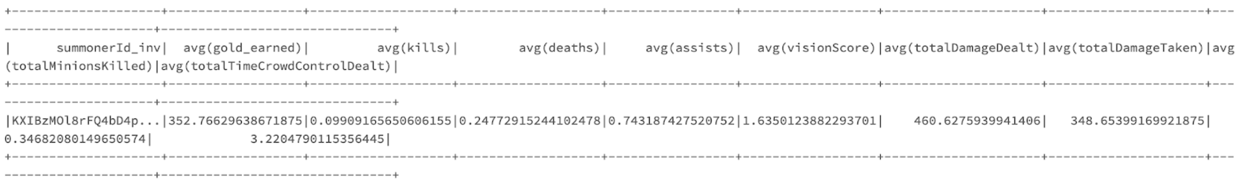
```
+--------------------+--------------------+--------------------+----------+------------------+--------------------+--------------------+--------------------+
|           agg_stats|           champions|               lanes|  match_id|      match_length|               roles|         summonerIds|              teamId|
+--------------------+--------------------+--------------------+----------+------------------+--------------------+--------------------+--------------------+
|[[7864.0, 3.0, 3....|[245, 13, 114, 25...|[JUNGLE, TOP, MID...|3207597356|             22.55|[NONE, SOLO, SOLO...|[SAim3CJlYb3gTey9...|[100, 100, 100, 1...|
|[[6933.0, 1.0, 8....|[111, 145, 107, 1...|[BOTTOM, BOTTOM, ...|3207612018|27.166666666666668|[DUO_SUPPORT, DUO...|[CInmKwfaqk85iX9G...|[100, 100, 100, 1...|
+--------------------+--------------------+--------------------+----------+------------------+--------------------+--------------------+--------------------+
```

Table 3. match_sum_stats_agg_df



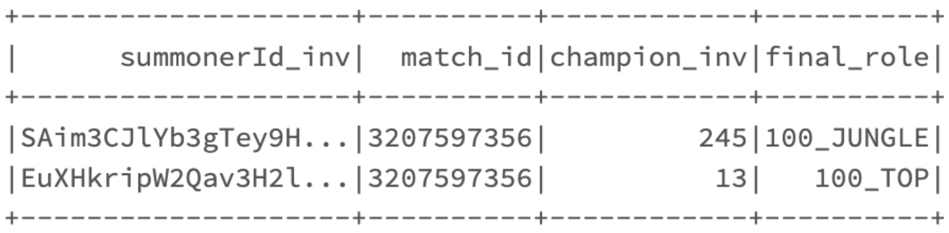
```
+----------+--------------------+------------+--------------------+------------+-----------+
|  match_id|    player_stats_inv|champion_inv|      summonerId_inv|match_length| final_role|
+----------+--------------------+------------+--------------------+------------+-----------+
|3207597356|[7864.0, 3.0, 3.0...|         245|SAim3CJlYb3gTey9H...|       22.55| 100_JUNGLE|
|3207597356|[6533.0, 2.0, 3.0...|          13|EuXHkripW2Qav3H2l...|       22.55|    100_TOP|
|3207597356|[6526.0, 0.0, 0.0...|         114|FnmTWvNqTGk-2pnJ9...|       22.55| 100_MIDDLE|
|3207597356|[6823.0, 4.0, 3.0...|          25|9m3TRvR-RpcGTN0We...|       22.55|100_SUPPORT|
|3207597356|[6492.0, 1.0, 4.0...|          22|nZuXoApHlr15y9--k...|       22.55|  100_CARRY|
|3207597356|[9405.0, 1.0, 5.0...|          74|AKrjxV60ReBpP9ZIG...|       22.55|  200_CARRY|
|3207597356|[9522.0, 5.0, 4.0...|          38|abPiRq_oHl1pdzEzJ...|       22.55| 200_MIDDLE|
|3207597356|[10890.0, 10.0, 5...|          29|ffJNX6IZF2NfHxihw...|       22.55| 200_JUNGLE|
|3207597356|[10164.0, 8.0, 3....|         246|14jr1KUj1dWDc5FDZ...|       22.55|    200_TOP|
|3207597356|[8373.0, 2.0, 10....|         161|0MVgTen2qIhMjKj4L...|       22.55|200_SUPPORT|
+----------+--------------------+------------+--------------------+------------+-----------+
```

Table 4. match_sum_stats



```
+--------------------+------------------+-------------------+-------------------+-----------------+------------------+---------------------+---------------------+---
---------------------+------------------------------+
|      summonerId_inv|  avg(gold_earned)|          avg(kills)|        avg(deaths)|     avg(assists)|  avg(visionScore)|avg(totalDamageDealt)|avg(totalDamageTaken)|avg
(totalMinionsKilled)|avg(totalTimeCrowdControlDealt)|
+--------------------+------------------+-------------------+-------------------+-----------------+------------------+---------------------+---------------------+---
---------------------+------------------------------+
|KXIBzMOl8rFQ4bD4p...|352.76629638671875|0.09909165650606155|0.24772915244102478|0.743187427520752|1.6350123882293701|    460.6275939941406|   348.65399169921875|
0.34682080149650574|             3.2204790115356445|
+--------------------+------------------+-------------------+-------------------+-----------------+------------------+---------------------+---------------------+---
---------------------+------------------------------+
```

Table 5. summoner_avg_stats



```
+--------------------+----------+------------+----------+
|      summonerId_inv|  match_id|champion_inv|final_role|
+--------------------+----------+------------+----------+
|SAim3CJlYb3gTey9H...|3207597356|         245|100_JUNGLE|
|EuXHkripW2Qav3H2l...|3207597356|          13|   100_TOP|
+--------------------+----------+------------+----------+
```

Table 6. match_sum_stats_join

We then join 3 tables: *summoner_avg_stats, label_col,* and *mast_champ_join* together through the use of another table: *match_sum_stats_join.* Our feature matrix, *players_matchs_stats,* now has the schema as follows:

```
root
 |-- match_id: string (nullable = true)
 |-- champion_inv: string (nullable = true)
 |-- final_role: string (nullable = true)
 |-- name: string (nullable = true)
 |-- armorperlevel: double (nullable = true)
 |-- attackdamageperlevel: double (nullable = true)
 |-- attackspeed: double (nullable = true)
 |-- attackrange: integer (nullable = true)
 |-- spellblockperlevel: double (nullable = true)
 |-- hpperlevel: integer (nullable = true)
 |-- summonerId_inv: string (nullable = true)
 |-- avg(gold_earned): double (nullable = true)
 |-- avg(kills): double (nullable = true)
 |-- avg(deaths): double (nullable = true)
 |-- avg(assists): double (nullable = true)
 |-- avg(visionScore): double (nullable = true)
 |-- avg(totalDamageDealt): double (nullable = true)
 |-- avg(totalDamageTaken): double (nullable = true)
 |-- avg(totalMinionsKilled): double (nullable = true)
 |-- avg(totalTimeCrowdControlDealt): double (nullable = true)
```

We then remove matches with less than 10 rows in the expanded matrix. This is because we only want to keep the *seed matches* and valid *secondary matches* for our final matrix. Most *Secondary matches* do not have a large enough player history for all 10 players.

In the end, to create our **final feature matrix**, for each valid match, we create a set of features based on the team and the role of the players. Features were named in the format *team_role_stats*. For example, one feature name is *100_JUNGLE_avg(totalDamageDealt)*. Each instance in *features_matrix,* our final processed feature matrix, is one match. Each match will have 150 features column and 1 label column. Each of the 10 player-champion pairs in the game will have 15 statistic features. The list of features are:

Champion feature	**Players features (average)**
Armor per level Attack damage per level Attack range Attack speed Hp per level Spell block per level	Assists Deaths Kills Gold earn Total Damage Dealt Total Damage Taken Total Minions Killed Total Time Crowd Control Dealt Vision Score

Features_matrix is then used to perform logistic regression to produce a prediction model. The *features_matrix* is saved back to CosmosDB and will be collected again through Databricks to perform machine learning.

Originally, we worked on a sample dataset that was only a few hundred records and we intended to use the champion mastery scores for each player in the model. However, once we were ready to work on our full dataset we ran into trouble. Joining the champion masteries onto the players proved to be too computationally expensive, and even with an increased cluster size we could not process the data in Spark to match our plans. This is because this join operation multiplied the number of records by a huge amount. Since there are 146 champions, every player has a value for each of the 146 (even if the value is 0), which we had to join for all players we had. This operation was incredibly expensive and not feasible with the full dataset. We decided to implement a simpler model which did not include the champion mastery values in order to make sure that we had a deployable model on the full dataset.

D. Machine Learning Model

Problem definition

Given a match ***m***, list of 10 players ***p*** and their corresponding list of 10 champions ***c*,** for each $\{p_i, c_i\}$, $1 \leq i \leq 10$, we generate a feature list $x_{p,c,i}$. Each feature list consists of 9 player history statistics and 6 champion statistic. Match ***m*** features will then be $X_m = \{x_{p_1,c_1}, x_{p_2,c_2}, ...x_{p_i,c_i}\}$, $1 \leq i \leq 10$. Let ***y*** be a binary number indicate whether blue team (team 100) wins the match, our model will try to estimate the posterior probability $P(y_m = 1 \mid X_m)$

We chose to employ a logistic regression model for this project. This was chosen based off of the related works mentioned earlier. It is the correct blend of simple, fast and powerful. As discussed earlier, the features included a number of stats for each player in the match. Since there are 10 players this resulted in a pretty large number of features. As was discussed earlier in the paper, we ended up having 2 models. One complex one which could only be fitted on a smaller data set, and one simple model which can be fit on the entire dataset. The simple model, which is the one that was deployed in the end had 150 features, 15 for each player in the match. For comparison's sake, and also to see which would be easier to deploy to the web app, we used two libraries to fit our logistic regression models. The first model we created was done in SparkML, and the second was done through Scikit Learn and Azure Machine Learning.

Fitting models on Scikit-Learn compared to SparkML wasn't that different. The main difference between the two is that one uses Spark dataframes while the other uses Pandas dataframes. It is relatively simple to fit a model with either one, and both yield similar results. However, the Scikit model proved to be more useful in the end because the integration with Azure Machine Learning Workspace, which is what we used to deploy the model. The advantage of the Spark ML implementation was that we were already working on a Spark cluster. However, switching over to scikit was as simple as converting the Spark dataframe to a Pandas dataframe. Both models were fitted and tested using a 70/30 train test split.

- *HDInsight Vs Databricks*

After fitting the model in Spark ML we had to find the best way to deploy the model to a webapp. Upon doing research we found that it was very hard to deploy from HDInsight. To get around this we had to move our model from HDInsight to another spark platform, Databricks. On Databricks we could deploy the model through Azure Machine Learning Workspace. Compared to HDInsight, we found Databricks to be slightly more difficult to configure initially. However, it allowed greater flexibility and easier management than HDInsight did. HDInsight clusters can't be shut down and instead must be deleted when you are done using them. They also cannot be changed while you are using them. In contrast, Databricks clusters can be modified as you use them if necessary, and you can delete the clusters without deleting everything else you are doing on Databricks [5]. For instance, at one point we increased the number of workers on our Databricks cluster from the default 4 up to 8 to speed up our work. HDInsight persists its data on an Azure Blob Storage system [6], but it is not as convenient or easy to access as it is on Databricks. Since Databricks also runs Spark with Pyspark notebook support, it was easy to transition our code from the HDInsight cluster to the new Databricks one. The only hiccup when transferring from HDInsight to Databricks was the CosmosDB connector. Setting it up for HDInsight was a more intuitive process, but once it was set up everything worked as expected. Thus it was easy to fit a new SparkML Logistic regression on our sample dataset.

E. Deploying Machine Learning model

Deploying the model proved to be one of the most difficult tasks of the entire project. There were quite a few options to choose from, but none of them were as simple as we would have hoped. The solution that we ended up using required us to use Azure Machine Learning Workspace (AMLW) to deploy the model that we had already made. As was mentioned earlier, Azure Machine Learning's integration with Scikit Learn was better than it was with SparkML, so we decided to deploy the model we fit with that. The results of the two machine learning libraries were very similar anyway, so we were not losing anything by choosing to use the Scikit model.

We first integrated AMLW with Databricks and created an experiment on the workspace. Then we created a run in the experiment. The run performs model fitting and tuning using Scikit-learn. After the model is done, Scikit-learn produces a pickled model file. The run then uploads the file to its experiment space.

We then located that model file on AMLW and downloaded it to our local host. We then used this downloaded model to deploy our web application on local host. We once again used

AMLW to create a **Docker Container** to make a Machine Learning server which receives user input, which is a match instance, and outputs the result, the predicted outcome of the match, through REST calls.

The web app that we created is a very simple one to test the deployment of the model. It takes user inputs for all of the players on each team, as well as the champions that they are playing and their roles. It is then process this input to create one match instance. The instance then is sent to the Docker Container and the Container returned the predicted probability. The web app was created using Flask, which is a Python framework for web development. Flask was a good choice because it was quick to learn and setup a basic app. It was also a good choice because it is in Python, which is a language that we have more experience with than the other alternatives for the task.

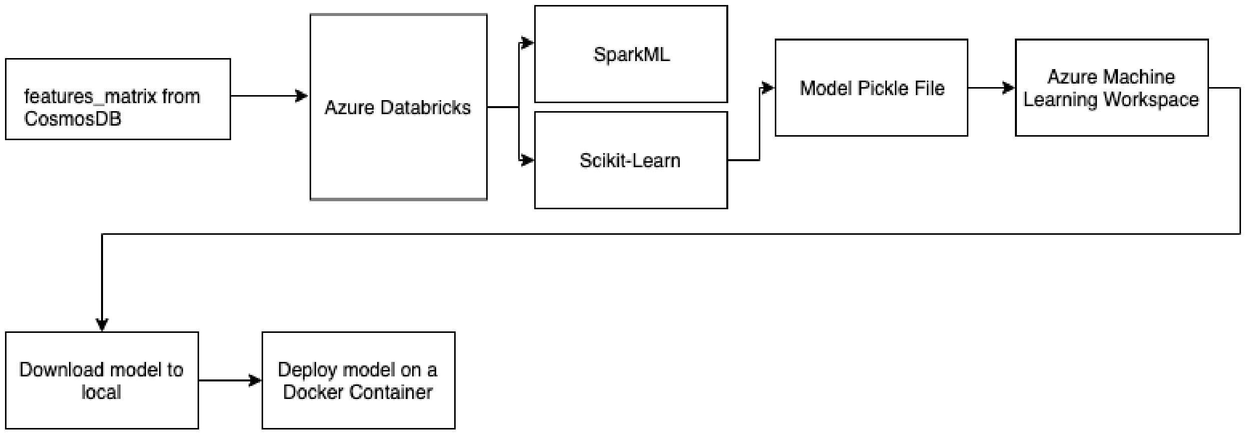

features_matrix from CosmosDB
Azure Databricks
SparkML
Scikit-Learn
Model Pickle File
Azure Machine Learning Workspace
Download model to local
Deploy model on a Docker Container

Figure 3. Machine Learning process till deployment

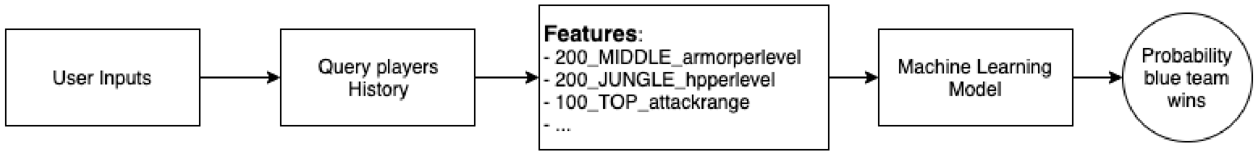


Figure 4. Web app process

IV. RESULTS AND DISCUSSION

In our raw data we queried 25784 matches and 9784 players. This was pruned down to 9112 matches and around 2000 players after data processing. The processing was done on HDInsight was done on a cluster with 2 worker nodes, while the processing done on Databricks was done on a cluster with 8. Each worker node on HDInsight had 8 cores and 56GB of memory. Each worker node on Databricks had 4 cores and 14 GB of memory. The driver nodes for both had 28 GB of memory. There are a few reasons for the sharp drop in the amount of data. The biggest contributor is that in our final model, we only use matches that we have history data of all 10 players, which means it will mostly consist of *seed matches* and some *secondary matches* with enough data. However, despite this large drop we still had a very healthy sample size. The final feature matrix had 150 features. The cost of adding additional features was very high since there needed to be one for each of the 10 players in the match. There were many statistics available to us in the match data that may have yielded some additional predictive power, however the trade-off in model complexity did not seem worth it as we were already dealing with so many features.

Our final model has 6378 records for training data and 2734 records for test data. The best run of our model on Azure Machine Learning resulted in an accuracy score of **83.38%** on test set. We also took other performance metrics of the model. They are included in the following tables and figures:

	Precision	***Recall***	***F1-score***	***Support***
0	*0.84*	*0.82*	*0.83*	*1365*
1	*0.82*	*0.84*	*0.83*	*1369*
avg/total	*0.83*	*0.83*	*0.83*	*2734*

0-actual	*1114*	*251*
1-actual	*214*	*1155*
n = 2734	***0- predicted***	***1-predicted***

log loss	*5.874*
ROC AUC	*0.829*

Table 7: Performance Metrics for the LR model. From top to bottom:
Precision, Recall and F1 on each class and overall
Confusion Matrix of the model results
Log-loss and area under the ROC curve for the model

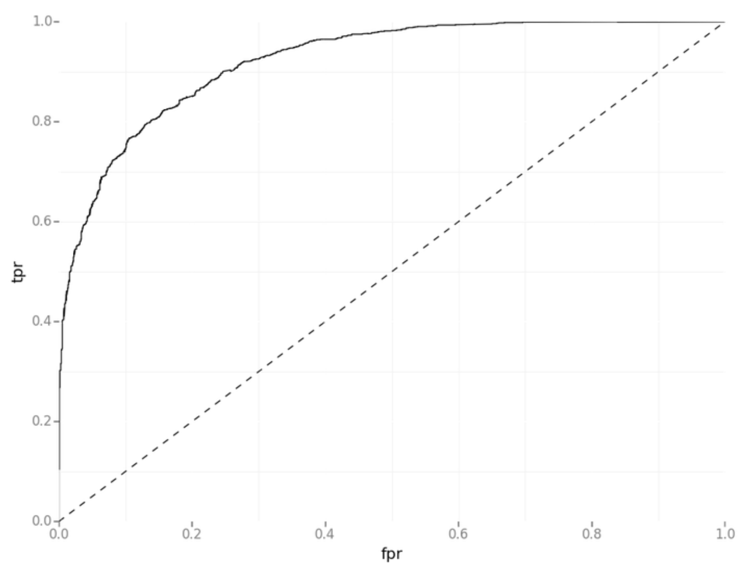
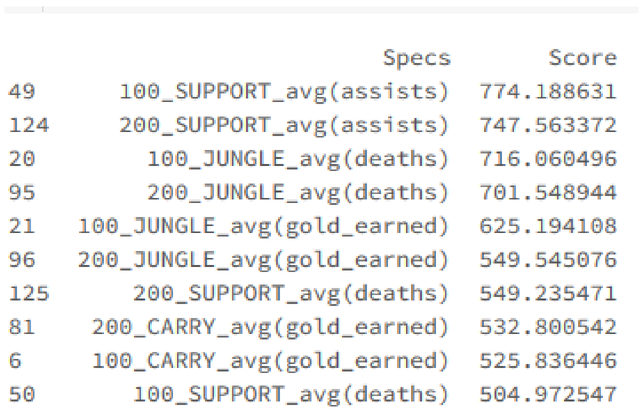


Figure 5: ROC curve for the LR Model

There are a number of interesting findings in these metrics (all on the test set) worth discussing. The F1 score, as well as the accuracy and the area under the ROC curve are all quite good and similar at around 0.83. The Log-loss was 5.874. We are very happy with these results as they are comparable or better to the related works that we viewed and higher than we initially expected. The value of our model is backed up by the fact that all of the metrics yielded similar results. Additionally, on multiple runs of the model on Azure Machine Learning, the results were consistent. The confusion matrix shows that there were relatively few false positives and negatives, and that the numbers of each were similar, which is reflected in the very similar precision and recall values.

These numbers were similar, though slightly weaker, than the numbers from our model made in SparkML. The Spark model yielded a ROC AUC of .93, Precision of .85, and Recall of .85. While these are certainly better than the numbers yielded by Scikit-learn, the difference is small enough that the trade off in ease of deployment made Scikit the superior choice for this project



```
                              Specs       Score
49      100_SUPPORT_avg(assists)  774.188631
124     200_SUPPORT_avg(assists)  747.563372
20        100_JUNGLE_avg(deaths)  716.060496
95        200_JUNGLE_avg(deaths)  701.548944
21   100_JUNGLE_avg(gold_earned)  625.194108
96   200_JUNGLE_avg(gold_earned)  549.545076
125      200_SUPPORT_avg(deaths)  549.235471
81    200_CARRY_avg(gold_earned)  532.800542
6     100_CARRY_avg(gold_earned)  525.836446
50       100_SUPPORT_avg(deaths)  504.972547
```

Table 8: Top 10 Important Features

We also perform features analysis. The most important features are consistent between teams which is a good sign of their predictive power. Though it is interesting that the gap between the importance of support average deaths by team is larger than the gap for the other features. They also open up some interesting discussions about *League of Legends* matches. The prominence of Jungle and Support stats in the top 10 features leads one to believe that these roles tend to have a higher impact on the outcome of a *League* match. On the other hand, the Top and Middle roles are absent from the top 10 features, which could mean that they have a lesser overall impact on the game. It is also interesting that even among the top 10 features, the top 4 are notably higher in predictive power than the rest of the top 10. With the top 2 having around 1.5 times the score of the 10th highest feature.

The total lack of champion statistics in the top 10 shows that the player tends to be more important than the champion pick. This does not come as a huge surprise to us, as a poor performance by a player can make the strengths of a champion wholly irrelevant. Champions have many strengths and weaknesses, but it is on the player to take advantage of them.

V. CONCLUSION

This project proved the predictability of *League of Legends* matches based off of the players and champions within the match. We were able to fit a logistic regression model that yielded good, consistent results on the data. The results were actually better than expected. With further work and a more complex model even better results might be possible.

Over the course of this project we used many different big data technologies and learned a lot about them. With the insight that we have gained from the project it is clear that the initial planning stage of the project could have been a lot cleaner if we had more experience. The fact that we had to switch platforms mid-project due to the limited deployment options on HDInsight was a glaring flaw in our planning that showed our inexperience with big data projects. However, it was a valuable experience as we got to compare the two services and learn more about the benefits of each.

One sentiment which we've often seen expressed both by professors and online is that people misjudge the proportion of a project which will be spent on data preparation compared to actual modelling. This lesson was definitely hammered home by this project. Our data came in a very messy format which required a lot of work to hammer down to features that we wanted. Dealing with multiple nested structs in the JSON hierarchy along with many array based features made querying and cleaning the data a much larger and longer process than was originally anticipated.

The other difficult part of the project was the deployment of the machine learning model once it was fit. This was a task that we originally underestimated when we were planning for the project as well. Part of our struggle was that ease of deployment varies greatly depending on the platform and libraries that you are using. However, this part of the project further proved the value of platform-as-a-service systems like Azure. Building the connections between all of the work that we did would have been even harder without the backing infrastructure of Azure which we used for all stages of the project.

There are a number of improvements and ideas for the future if we continue to work on this project. The most obvious one would be extending the model to be fit on our original full list of features. This was too computationally intensive for our purposes in this project, but given more time and resources we could see the predictive power of champion mastery scores on game outcomes. This would be an interesting look on how experience with a champion improves performance.

Another potential option for the front end of the project would be an improvement to user input. Our current implementation allows the user to input the 10 players, champions and roles within a match. However, an alternative implementation could simply take one player. The API could then query a live match of that player and fetch the other players and champion picks within that live match. We would then be making live predictions.

References

[1] "Apache Spark Tutorial: Machine Learning." *DataCamp Community*, 28 July 2017, https://www.datacamp.com/community/tutorials/apache-spark-tutorial-machine-learning.

[2] *Azure/Azure-Cosmosdb-Spark*. 2016. Microsoft Azure, 2019. *GitHub*, https://github.com/Azure/azure-cosmosdb-spark.

[3] Cojocar, Bogdan. "Realtime Prediction Using Spark Structured Streaming, XGBoost and Scala." *Medium*, 13 Oct. 2018, https://towardsdatascience.com/realtime-prediction-using-spark-structured-streaming-xgboost-and-scala-d4869a9a4c66.

[4] Conley, Kevin, and Daniel Perry. *How Does He Saw Me? A Recommendation Engine for Picking Heroes in Dota 2*. p. 4.

[5] *Getting Started — Databricks Documentation*. https://docs.databricks.com/getting-started/index.html. Accessed 24 Nov. 2019.

[6] hrasheed-msft. *What Are the Apache Hadoop and Apache Spark Technology Stack? - Azure HDInsight*. https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-overview. Accessed 24 Nov. 2019.

[7] *Integrating Kafka and Spark Streaming: Code Examples and State of the Game*. https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/. Accessed 10 Nov. 2019.

[8] Kalyanaraman, Kaushik. *To Win or Not to Win? A Prediction Model to Determine the Outcome of a DotA2 Match*. p. 7.

[9] *League of Legends: Predicting Wins In Champion Select With Machine Learning*. https://hackernoon.com/league-of-legends-predicting-wins-in-champion-select-with-machine-learning-6496523a7ea7. Accessed 19 Oct. 2019.

[10] *Libraries — Databricks Documentation*. https://docs.databricks.com/libraries.html#create-a-library. Accessed 24 Nov. 2019.

[11] Matsuzaki, Tsuyoshi. "Spark ML Serving with Azure Machine Learning." *Tsmatz*, 4 Mar. 2019, https://tsmatz.wordpress.com/2019/03/04/spark-ml-pipeline-serving-inference-by-azure-machine-learning-service/.

[12] *PredictLeague - Approach*. https://www.cs.hmc.edu/~jfelton/approach.html. Accessed 19 Oct. 2019.

[13] sdgilley. *Image Classification Tutorial: Deploy Models - Azure Machine Learning*. https://docs.microsoft.com/en-us/azure/machine-learning/service/tutorial-deploy-models-with-aml. Accessed 24 Nov. 2019.

[14] Tassi, Paul. "Riot Games Reveals 'League of Legends' Has 100 Million Monthly Players." *Forbes*, https://www.forbes.com/sites/insertcoin/2016/09/13/riot-games-reveals-league-of-legends-has-100-million-monthly-players/. Accessed 10 Nov. 2019.

[15] *The Flask Mega-Tutorial Part II: Templates - Miguelgrinberg.Com*. https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-ii-templates. Accessed 24 Nov. 2019.

[16] knandu. *Connect Apache Spark to Azure Cosmos DB*. https://docs.microsoft.com/en-us/azure/cosmos-db/spark-connector. Accessed 24 Nov. 2019.

[17] trevorbye. *Tutorial: Train Your First Azure ML Model in Python - Azure Machine Learning*. https://docs.microsoft.com/en-us/azure/machine-learning/service/tutorial-1st-experiment-sdk-train. Accessed 24 Nov. 2019.

[18] Wang, Weiqi. *Predicting Multiplayer Online Battle Arena (MOBA) Game Outcome Based on Hero Draft Data*. p. 16.

[19] Apache Spark Homepage, https://spark.apache.org/

[20] markjbrown. *Introduction to Azure Cosmos DB*. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. Accessed 25 Nov. 2019.