Riot Games API Challenge 2015

Sjokz to the System:
Building Tools to Analyze Rates of Change in the League of Legends Metagame

Taylor Brent
Jon Dutko
April 17, 2015

Introduction

Abstract

When the Riot Games API Challenge was announced, we chose to turn the opportunity into a science project. As long-time and passionate players of League of Legends, we know that a strong understanding of the "metagame" is a vital facet of a successful strategy. An accurate comprehension of the present state of the metagame can be the difference between victory and defeat on the Fields of Justice; this holds for Bronze-tier ranked games as well as for professional playoff matches and world championships.

But for how much emphasis is placed on the metagame, how much is really known about it? In conversation, the metagame is discussed in relative terms: "Sejuani is strong right now," for example. And there exist services which can quantify this relation objectively as a function of player decision-making, telling us (for example) that Sejuani's pick rate this week is three times greater than it was last week. But looking at singular, disjoint points does not reveal the whole story of the metagame. We believe that insight can be found by viewing the metagame not as an unconnected set of outliers but as a whole body, in motion.

This project aims to quantify the rate of change of the metagame over time. In doing so, we hope that the **volatility** of the metagame can be illustrated. We theorize that the effect of **shocks to the metagame**, both internal and external, can be quantified and plotted. Additionally we argue that trends in metagame change may exist, which can also be quantified, analyzed and, eventually, predicted using our model. Finally we believe that a thorough understanding of this data may have applications for both players and developer and lead to a deeper understanding of the game as a dynamic entity.

Technical Overview

The project presented is comprised of two main applications. The first is **PullScheduler**. PullScheduler is an executable JAR file which, when run, makes a series of calls to the Riot API api-challenge-v4.1 endpoint. These calls are executed to fetch **moments** of match data, representing a statistically significant number of matches played in the URF game mode during a given five-minute period. We parse this document populated with matchIDs and then query the Riot API match-v2.2 endpoint to retrieve the champion selection data for each match. This champion selection data is then gathered into a **ChampSelectData** object, defined as the spread of champion selections over a period of time. At the moment level, a ChampSelectData object represents the total champion selections for each champion in our sample of matches taken over the fetched five minute period. PullScheduler then saves the moment to disk using an ObjectOutputStream. PullScheduler runs in the background, querying the Riot Games API every five minutes after scanning the folder of saved moments, identifying and filling in holes in the data that may exist as far back as eight hours prior to run-time.

The second main application is **Mathman**, which runs aggregation and analysis operations on the moment data gathered by PullScheduler. Mathman first scans the directory of moments stored by PullScheduler and aggregates them into useful pieces of ChampSelectData. For our initial purposes, we have chosen to represent an atomic unit of useful metagame data as a period of three hours. We call these three-hour summative pieces of data **buckets**. Once we have converted the current directory of stored moments into three-hour buckets, Mathman processes the list of buckets into an ArrayList of Date, **ChampSelectUnitVector** pairs. ChampSelectUnitVector is a distinct object type from ChampSelectData; where ChampSelectData measures the count of each champion during a given interval, ChampSelectUnitVector stores the rate of change of each champion's representation between two time-adjacent ChampSelectData objects with a congruent time interval. These values are stored as doubles matched with individual champions, such that the sum total of each champion's vector is the unit vector. Once these buckets have been processed, we can see the extent to which the metagame changed at the time represented by each bucket. We aggregate this data, and then output it as a series of comma separated values to "metaDataStandard.csv" in the application's directory.

Project Architecture
Class Structure

`PullScheduler`

PullScheduler is the method by which we query the Riot API database and gather match data at a constant rate. PullScheduler's primary method is `doorToDoorMoment`, which takes in a time (represented in Long format as an epoch) and makes numerous calls to Whisperer, returning the ChampSelectData for that moment in time. PullScheduler also makes calls to Archiver to store moments as they come in, as well as to scan the directory for holes in the data that may need to be filled in.

`Mathman`

Mathman is the application we run in order to perform aggregation and analysis on our set of data. Mathman makes calls to Processor to convert moments into buckets, and to convert buckets of ChampSelectData into ChampSelectUnitVectors. Mathman writes the data to a spreadsheet locally.

`Whisperer`

Whisperer is a class containing static methods for querying the Riot API. Of primary importance are `getMatchIDsForMoment`, which takes in a time (represented in Long format as an epoch) and returns a list of matchIDs corresponding to live games played during the five minute period starting at the passed-in time; and `getChampIDsFromMatchID`, which takes in a matchID from the list returned by the previous method and returns a list of champions played in the game represented by that ID.

`Archiver`

Archiver is a class that, when instantiated, functions as a mode for both saving moments to and loading moments from the database. These functions are represented by the methods `saveMoment` and `loadMoment`. Archiver also contains a method, `pruneEightHoursList`, to scan the directory and see if there are any holes from the past eight hours in our database that must then be filled.

`Processor`

Processor is a class containing static methods for converting the data we have stored into meaningful buckets, which are later operated upon by Mathman. Processor has only one public-facing function, `getBuckets`, which takes in a start and end date, an Archive object, and the time length of each bucket, represented as a Long value. The function returns a list of Date, ChampSelectUnitVector pairs. At time of writing, each bucket represents three hours of data.

`ChampSelectData`

ChampSelectData is a data structure representing the number of times each champion was played during a given time period. This data is stored in an array of Integers called selections, where each index corresponds to a known champion.

`ChampSelectUnitVector`

ChampSelectUnitVector is a data structure representing the change experienced in the number of times a champion was picked between two adjacent, congruent time periods. This data is stored in an array of Doubles called ratios, where each index corresponds to a known champion.

Data and Analysis
Mathematical Basis

We view the **rate of change of the metagame** at a given point in time, t_i, as a function of the following equation:

$$\Delta meta = distance(\overrightarrow{x_i}, \frac{\sum_{j=1}^{i-1} \overrightarrow{x_j}}{i-1}),$$

where $\overrightarrow{x_i}$ represents a champion selection unit vector at time i. The calculation of the second argument is represented in the code by the getMetaVectorsStandardApproach function in the Mathman class (fig. 1). The distance between a champion selection unit vector and the vectors that precede it is what our application outputs, and what we plot on our graphs. Our distance function takes in two unit vectors and returns a double; it is represented in the code by the getDistance function in the Mathman class (fig. 2). This function calls a "difference constructor", building a unit vector out of the difference of two other vectors (fig. 3).

```
private static List<ChampSelectUnitVector>
getMetaVectorsStandardApproach(List<Pair<Date, ChampSelectUnitVector>> buckets){

	ChampSelectUnitVector baseMetaVector = buckets.get(0).getSecond();
	List<ChampSelectUnitVector> toReturn = new ArrayList<ChampSelectUnitVector>();
	toReturn.add(baseMetaVector);
	toReturn.add(baseMetaVector);

	for(int i = 2; i < buckets.size(); i++){
		ChampSelectUnitVector metaVector = new ChampSelectUnitVector
		toReturn.get(toReturn.size()-1));
		metaVector.multiply(i-1);
		metaVector.add(buckets.get(i-1).getSecond());
		metaVector.divide(i);
		toReturn.add(metaVector);
	}
	return toReturn;
}
```

Figure 1: The function for extracting second-argument values for series of dates, extracted from our original list of unit vectors.

```
private static Double getDistance(ChampSelectUnitVector a, ChampSelectUnitVector b){
	Double squareTotal = 0.0;
	ChampSelectUnitVector difference = new ChampSelectUnitVector(a, b);
	for(int i = 0; i < Constants.champions.size(); i++){
			squareTotal += Math.pow(difference.ratios[i], 2.0);
	}
	return Math.sqrt(squareTotal);
}
```

Figure 2: The function for calculating the distance between the present champion selection unit vector and the one before it.

```
//difference constructor
//returns a champselectunitvector that is a - b
public ChampSelectUnitVector(ChampSelectUnitVector a, ChampSelectUnitVector b){
	ratios = new Double[Constants.champions.size()];
	for(int i = 0; i < ratios.length; i++){
		ratios[i] = a.ratios[i] - b.ratios[i];
	}
}
```

Figure 3: The constructor that returns a unit vector representing the difference between two unit vectors.

Scope, Data Set and Plot

The data we scraped from the API Challenge endpoint took place in the period starting on April 5, 2015 at 4:30PM PST and running up through the end of the URF game mode on April 13, 2015 at 4:30AM PST. In an ideal setting, a more complete data-set would have been compiled with data representing the breadth of the URF game mode from start to finish, beginning with its release on April 1. Further, an outage in collection resulting from a broken connection occurring on the morning of April 8 was able to be mostly recovered from but resulted in the loss of about nine hours of data. We have corrected our code to remain robust in the face of future outages, and the functions through which we pass our data for analysis account for outages such as these.

The duration for which data was collected is significant to the extent that it contains **a shock that we can expect to create a local increase in the volatility of the metagame**. On April 7 the patch notes announcing the deployment of Patch 5.7 were announced, disseminated through the player base through the client itself and through a variety of social networks. These patch notes, in addition to the patch itself, could be interpreted as "shocks" to a metagame at rest, compelling a user-base to experiment with champion selections.

The plotted data (fig. 4) represents the sum total of the data collected by PullScheduler and then analyzed by Mathman. This data is available in the "metaDataStandard.csv" file in this project's repository.

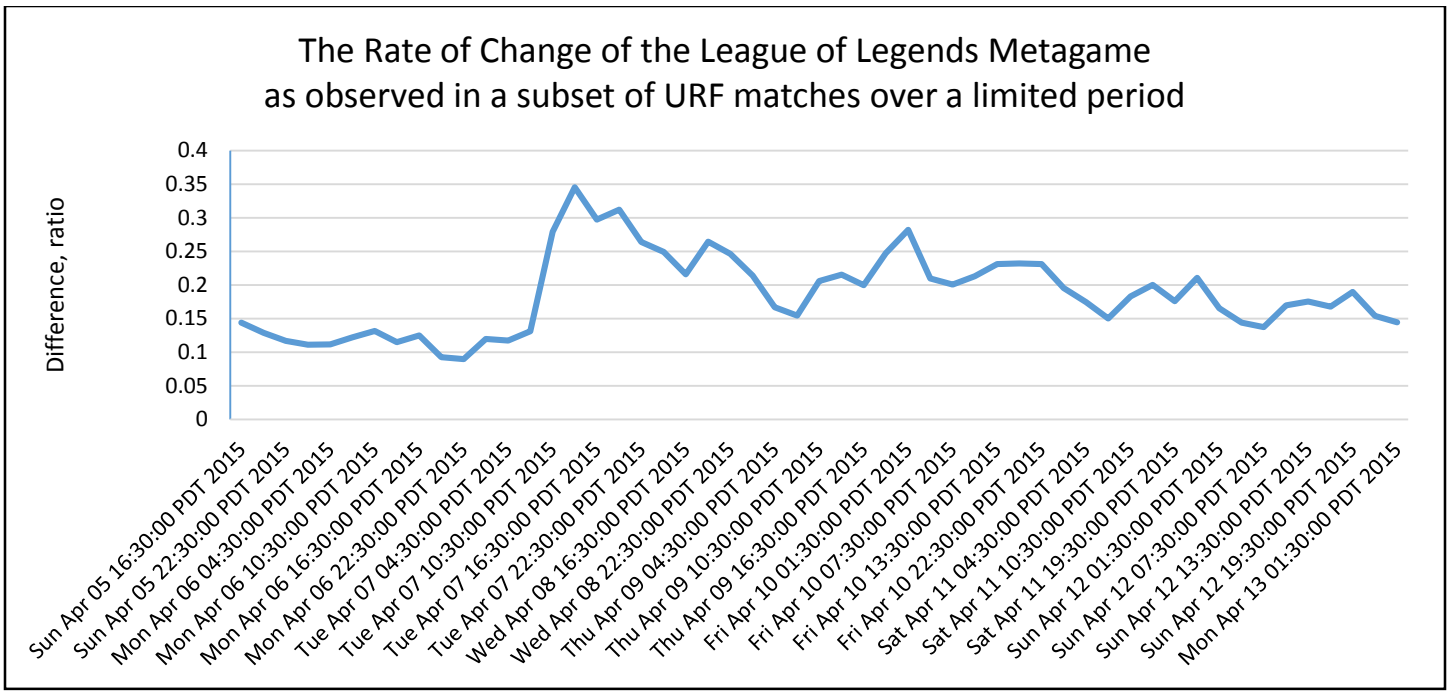

Figure 4: A visualization of the data obtained by PullScheduler and analyzed by Mathman

Conclusion

Results

The plotted analysis appears to confirm our hypothesis that, using the tools discussed here, the volatility of the metagame can be quantified and illustrated. An interpretation of the graph above shows an URF mode whose metagame is cooling five and six days after its release. The Patch 5.7 notes are then released on April 7, and a spike in metagame volatility can be observed throughout the day. Afterwards the rate of change of the metagame is seen to cool again in a downward trend until the game mode is pulled on April 13.

The choice to run our initial data collection and analysis on the URF game mode was one catalyzed by the requirements of the API Challenge but nevertheless provides an interesting lens through which to view the data. Intuitively we can imagine that there is less regularity in the URF game mode – the mode is more prone to a background volatility – because it is inherently experimental. Similarly, because of the casual nature of URF – it is important to remember that this is an unranked mode – it can be presumed that there is less of a social pressure to adhere to a metagame. Despite these considerations, we were still able to observe the effect of an external shock on the metagame.

Next Steps, Remaining Questions and Useful Applications

Our next step is to formally request a developer key with greater allowances for requests from the API. This will allow us to run data collection on a much broader and more interesting set of data: that of historical ranked matches in solo queue and ranked teams. By collecting such historical data we would hope to analyze the resulting plots and compare them against known possible shocks to the metagame, be they internal (such as released patches) or external (such as virulent social network activity or influential professional play).

By having a much larger and more competitive-relevant data set to work with, we can hope to answer a series of questions that we still possess or that have risen as a result of this experiment. Do there exist day-to-day or month-to-month trends in metagame volatility? Are players less willing to stray from the metagame in the middle of the day? How quickly will a new metagame cool down after a shock of some size? To what extent is the metagame as a whole influenced by Keane locking in Urgot? Are players in Gold more or less loyal to the metagame relative to those in Diamond? Are solo queue players more or less predictable than those who play in ranked teams?

We hope that, with this expanded data set, we can offer a web-forward interface for individuals to access, use, and explore what we have discovered with the tools we have built. Ideally we could offer analysis across servers and game-modes, offering analysis across variable time-spans and of scaling moment-length. These findings can be broken down further into champion-by-champion illustration, and the scale of future shocks to the system could arguably be predicted and socially disseminated.

In the act of expanding our software to answer these questions, we believe that the tools we build and the data output by these tools will have many applications. As a gameplay developer, these tools could be used to fine-tune future patches, calibrating champion fixes to shock the metagame to a precise, predictable extent. As a professional coach, these tools could effectively illustrate the extent to which one must change their team's strategies, hoping to gain the competitive edge by keeping abreast of the shifting champion environment.

We imagine that individual players accessing this data would gain all of these insights as well, in addition to one piece of wisdom that we would hope to communicate: that League of Legends is an ever-changing, complex and dynamic system. To quote a favorite Voidborn predator, in order to find success, we all must consume and adapt.