Spring 5-18-2018

Detecting Metagame Shifts in League of Legends Using Unsupervised Learning

Dustin P. Peabody
University of New Orleans, dpeabody@uno.edu

Detecting Metagame Shifts in League of Legends Using Unsupervised Learning

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Dustin Peabody

B.S. University of New Orleans, 2013

May, 2018

Table of Contents

List of Figures

List of Tables

Abstract

Over the many years since their inception, the complexity of video games has risen considerably. With this increase in complexity comes an increase in the number of possible choices for players and increased difficultly for developers who try to balance the effectiveness of these choices. In this thesis we demonstrate that unsupervised learning can give game developers extra insight into their own games, providing them with a tool that can potentially alert them to problems faster than they would otherwise be able to find. Specifically, we use DBSCAN to look at *League of Legends* and the *metagame* players have formed with their choices and attempt to detect when the metagame shifts possibly giving the developer insight into what changes they should affect to achieve a more balanced, fun game.

Keywords: Unsupervised Learning, DBSCAN, Metagame, Game Development, Clustering

Introduction

Video games have come a long way in the roughly forty years since their inception, and with all that change, the complexity of games has increased as well. This complexity, and the variety it brings, are often the very traits players search for when choosing their games. After all, more variety often means more and longer player engagement, and that often means more revenue for game developers. For competitive games, keeping everything balanced is of prime importance. How are players supposed to measure their skill if the underlying game is imbalanced? Did the player win the match or did the game win it for them? Conversely is the player at fault for losing or was it really the fault of the game itself?

Knowing that competitive games require balancing, how do game developers achieve this? With the ubiquity of the internet developers don't need to release a perfectly balanced game. Instead, they can release small updates to the game over-time to arrive at some semblance of balance. They can also utilize patches to release new content for their game. These incremental releases keep players interested and engaged, but also further increase the game's complexity. So now game developers are put in a difficult situation: balance the game they already have, or add new content which will itself change how the game is balanced and played. Often they attempt to preempt this problem using quality assurance teams and test environments, but for some games the players outnumber the developers a million to one. The developers can't possibly find and try every combination of variables to ensure a continuously balanced product. By using unsupervised learning, we can, with relatively little domain or game specific knowledge, build a system that empowers the development team to find and identify problems and patterns more quickly by pointing them in the right direction.

This thesis will show that unsupervised learning can be used to give the developers insight into their own games perhaps showing them areas that need attention or alerting them to problems faster than they would be able to find alone. Unsupervised learning can help find patterns for things that are non-obvious and help guide development toward a more enjoyable experience for everyone.

In the following pages we describe a system that, given some easily obtainable, high level data, can help identify what choices players are making in a multiplayer game and why. These choices shape the game's ecosystem and are ultimately a reflection of the current state of the game and its balance. We use DBSCAN and some novel techniques to find and differentiate between clusters formed by several sequential patches in the game *League of Legends*. We show that our methods can accurately detect when a large shift in player choice occurs. Our results demonstrate the usefulness that unsupervised learning techniques exhibit and contribute some first steps in the domain of continuous game balance and *metagame* detection.

Background

In the following pages, we'll look at a game called *League of Legends* the world's most popular game [1] and one with more than its fair share of complexity. Boasting roughly 130 characters to choose from, a game of 10 players has over 163 trillion possible combinations of characters in it. And this is before the game even begins when players' actions would balloon the complexity even further. *League of Legends* (LoL) is a multiplayer online battle arena (MOBA) style game, developed by Riot Games (Riot), in which two teams of 5 players each face off against each other for control of a map called Summoner's Rift. The objective of the game is to destroy the enemy team's Nexus, a heavily defended structure located deep within the enemy team's territory. Players accomplish this by selecting from over 130 characters to control which all function differently. Before the game begins each team takes turns banning one character at a time, preventing anyone in the game from selecting that character, until there are six total banned characters. Then each team alternates choosing one character at a time until both teams are comprised of five characters each, and then the game begins. In this thesis we mostly concern our selves with which characters each player bans and why. Often this is influenced, and sometimes outright decided, by the current *metagame*.

Metagame is a term used when the actions inside a game or results of a game are affected by or affect things outside of that singular instance of the game. Tournaments are an example of a metagame. The outcomes of each match influence the standings of the tournament and dictate how it proceeds, who wins, and who loses. In this way, the *metagame* of the tournament is affected by the results of each match, but the *metagame* can also affect the matches played inside of it as well. If a player were to discover a dominant strategy, that

strategy would be played more and start to stifle other strategies of play. Then people would attempt to overcome this dominant strategy with yet another strategy, even if it would be worse in a vacuum. This is the style of *metagame* found in LoL and it can be found in many other games as well. Characters or strategies become popular and spread, then players look for ways to overcome these prevalent characters or strategies to better secure victory.

Metagames evolve, grow and change over time naturally as players learn the game and adapt to other players' actions, but they can also be changed in other ways, namely if the game itself changes. While unlikely or unusual in many games such as chess or football, and even many older video games like *Super Smash Brothers Melee*, modern video games can change over the course of their lifespan. Often this is accomplished via *patching*. Patching is the release of small updates to the game after the game's initial release. While patches are sometimes for fixing bugs or other code related issues, they can also be used to modify the balance of a game. Developers might increase how much health a character has or change how much something in the game is worth. These changes also have an effect on the *metagame* and cause it to shift overtime, sometimes drastically. Developer's motivations for doing this can differ greatly. Sometimes certain characters or strategies are overbearing and are harming player engagement. Other times some aspect of a game lies ignored and forgotten, so the developers somehow make it more appealing for players to use. And still other times, the developers just want to keep the game fresh and exciting.

Whatever the reason for patching, if the underlying game is modified, the metagame can shift as well, sometimes in unexpected ways. Riot, the developer of *LoL*, frequently utilizes patching to modify their game. Roughly every two weeks they release a patch with updates that

may strengthen, weaken, or add characters or change various aspects of the game, causing shifts in the prevalent strategies employed by players. This predictable cycle of changes and potential metagame shifts makes League of Legends particularly interesting to study.

Patches are how many developers, Riot in particular, continue to interact with their game after launch. As such, the content of the patches is important. Unsupervised learning can aid developers in making decisions about what will be included in a given patch. If a pattern is observed where players are abusing a certain mechanic, the problem can be corrected, or if a feature is shown to be underutilized, the developers can rework the feature to be more appealing. These sorts of problems can be detected with unsupervised learning based on player behavior, allowing the developers to quickly identify the problems and begin developing fixes, which are then deployed via patches.

Related Work

Video games have been the subject of extensive research. This research can be grouped into either game based research or player behavior based research. Game based research is focused on things in and immediately surrounding the game. Kica et al. [2] studied the effect of patches on *League of Legends* by categorizing the changes present in each patch and relating them to the various champions' performance and Riot's patching habits. While they also examined the effect patches have on *League of Legends*, we choose to focus more on which groups of characters are being banned than on individual characters' win rates over time. Wu, Xiong, and Lida [3] explored features of various Multiplayer Online Battle Arena "MOBA" games and their impact on the balance and fairness of the games. The mechanisms they investigated included which characters appeared in professional games as well as pick and ban order. We chose to focus on which characters were banned rather than include the order, but more information could possibly be gleaned from the ban order as they demonstrated. Claypool et al. [4] examined the *League of Legends* matchmaking system and how players perceive its fairness and effectiveness. They find that games in general are balanced according to LoL's in-game rankings, and that players often derive greater enjoyment from winning games regardless of the fairness of the game itself. These findings help inform us of players' desires to win being of great importance. We can assume the desire to win is a driving factor behind in game decisions and selections. Beau and Bakkes [5] use Monti Carlo simulations to examine the impact of game actions and then modify the highest impact actions until they arrive at a balanced game. In the simple game they designed, they were able to run repeated trials until they discovered the highest impact actions or choices. This iterative approach to discover dominant strategies is

similar to how the millions of players of *League of Legends* can arrive at the strongest possible actions or choices for the current iteration of the game even when the developers may not have found those optimal choices themselves.

Player behavior research is often more concerned with identifying patterns in behavior, often to aid developers in designing games that players will enjoy more. Sifa et al. [6] utilize Simplex Volume Maximization to classify player behavior in *Tomb Raider Underworld* and how it changes across the length of the game. Drachen et al. continue [7] and develop behavior profiles in the Massively Multiplayer Online Role-Playing Game *Tera* and the multi-player strategy war game *Battlefield 2: Bad Company 2* using K-means and Simplex Volume maximization. Their data is cardinal in nature whereas our is nominal, but they are still utilizing clustering techniques to define player behavior. Rather than look at instances of their target games, they instead look at the accumulated stats of the players and try to cluster the players into different behavioral groups. Both of these papers endeavor to use clustering on player behavior, which is similar to our methods because the metagame is formed via players' decisions and behaviors. Drachen et al. [8] continue their research and use several clustering methods to build player behavior profiles across two distinct game modes with a focus on performance and play style in the video game *Destiny*. This time the authors' aim is again to classify players based on their in game behavior, with the future goal being to recommend items that compliment their play style. This again speaks to the utility that clustering has on discovering insights from otherwise unparsable amounts of data. Player behavior can also be utilized in more direct ways like predicting the outcome of a match as shown by Ong, Deolalikar, and Peng [9]. They developed an unsupervised learning framework to find behavior

clusters which they leveraged into a classification algorithm in an attempt to learn an outcome predictor. Again the authors of this paper cluster based on cardinal data gathered about players representing their behavior. Here, however, the authors take those clusters and use them to train various classification models, supporting the idea of an “ideal team” based on player behavior and chosen characters that fit those behaviors. We are more interested in evaluating the metagame which is more concerned with the whole of player behavior than it is with individual player behavior.

Data Collection

League of Legends is a very complex game, and there are many different kinds of data we could target for collection, but we want data which fits a few specific criteria. First we want data that is easy to understand. We're trying to find data that reflects what's going on in LoL's *metagame*, while simultaneously being easy for someone with little-to-no understanding of the game's more intricate systems to use. By utilizing this type of data we make a system that can easily be ported to various games, all while requiring little game specific knowledge. Secondly, we'd like to utilize data that is easy to collect and process. With these criteria in mind, we decided to use the set of characters that are banned in each game. This data is easy to understand as it's just the characters that are disallowed in a given game. The only real game knowledge related to this data is that the disallowing of these characters is done by the players, and it happens because they don't want to have those characters in the game. Additionally, this data is relatively easy to obtain via that Riot API.

Using the API, we collected 60,000 games across 6 patches from season 5 of *League of Legends*, approximately Fall 2015. Each patch consists of approximately two weeks of time, and each patch has its own set of differing ban rates for each character. The patches are labeled as: 5.12, 5.13, 5.14, 5.15, 5.16, and 5.17. These patches were targeted because, based on our experiences, there was a large *metagame* shift that occurred in patch 5.16 when an entire class of character was reworked. This is supported by several gaming news reports on the patch written at the time [10] [11] [12].

The Riot API does not support any random sampling methods, so when actually colleting the data, we began by using a few known high level players as seeds. The choice to only use

high-level players was made with the goal of only working with data generated by players with an exceptional understanding of the intricacies and nuances of both LoL and the *metagame* surrounding it. The hope is that they would quickly adapt to any shifts in the meta and that would be reflected in the data. We further restricted our focus to just the North American region because the author was more familiar with the high-level players making it easier to choose seed players.

We first looked for any games those players played in a given patch and selected one to add. We then looked at the nine other players and acquired their games. We repeated this process until we had around 10,000 games for the patch. We limited the number of games a single player could have per patch to one in order to control for a player having a favorite character or ban and skewing the data around that player. We also omitted any games that had incomplete or incorrect ban information in Riot's systems. All told, we collected approximately 60,000 games over the course of roughly a month of continuous API usage amounting to approximately 500MB of data.

All of the data was stored in a PostgreSQL database using the Active Record Object-Relational Mapping. This enabled us to use simple Ruby code to automate the acquisition of game and player data by creating an easy to use interface with the Riot API.

Methodology

Our goal is to develop a system that can identify and track metagame changes over time, but also requires little game specific knowledge so that it's both easier to use and simpler to port to other games. To that end, we decided to utilize unsupervised learning techniques, namely clustering, to perform the bulk of our investigation. Much of the work in and around the clustering was performed in the R programming language [13] because it has many trusted implementations of various statistical and machine learning algorithms. In this case we're using the FPC package [14] and its DBSCAN implementation for clustering.

DBSCAN [15] was chosen as the clustering method of choice for a few reasons specific to the dataset we have. Firstly, DBSCAN has the concept of noise points making it resistant to misclassification of outlier points. This is important because given our data there is a high likelihood of random and/or noise points (people just banning what they hate for reasons other than balance). Secondly, it can find clusters of arbitrary shape. Since this effort was largely exploratory, we didn't know what shape the clusters would take. The fact that DBSCAN can find non-linearly separable clusters is highly valuable. Finally, and most importantly, DBSCAN does not need to be provided with a number of clusters to find; the algorithm discovers that for us. These are all very important points and advantages over other clustering methods like the centroid-based algorithm k-means, which needs a starting number of clusters and often has difficulty finding clusters that are not linearly separable.

DBSCAN also allows us to use an arbitrary distance function. In many datasets Euclidean distance is used by default, but due to the high dimensionality and nominal nature of our data, Euclidean distance is not sufficient. Given that the bans in each game are essentially a set, we

decided to use the Jaccard index to measure distance between two games' bans. The Jaccard index is defined as $\frac{|A \cap B|}{|A|+|B|-|A \cap B|}$ where A and B are sets. Jaccard distance is defined as the dissimilarity between two sets. It is calculated by subtracting the Jaccard coefficient from one. Jaccard distance is thus defined like so $\frac{|A \cup B|-|A \cap B|}{|A \cup B|}$. We apply the Jaccard distance between every combination of points in a given patch, creating a dissimilarity matrix. The FPC package's DBSCAN takes this matrix as input in lieu of the original dataset. For example, if we have two points $A = [a, b, c, d, e]$ and $B = [a, b, d, f, g]$ their Jaccard index would be:

$$\frac{|[a, b, d]|}{|[a, b, c, d, e]| + |[a, b, d, f, g]| - |[a, b, d]|} \; or \; \frac{3}{5 + 5 - 3} = \frac{3}{7}$$

and the Jaccard distance would be:

$$\frac{|[a, b, c, d, e, f, g]| - |[a, b, d]|}{|[a, b, c, d, e, f, g]|} \text{ or } \frac{7 - 3}{7} = \frac{4}{7}$$

The dissimilarity matrix would look as follows.

	A	**B**
A	0	$\frac{4}{7}$
B	$\frac{4}{7}$	0

The DBSCAN algorithm has two parameters: ε and $minPts$. ε is the radius of the neighborhood around a point and $minPts$ is the minimum number of points that need to be within ε distance from a point for that point to be considered a core point. Any two points within ε distance of each other are reachable in respect to each other. The cluster is formed by

a core point reaching other core points forming a path from core point to core point, each within ε of the previous point. If a point is within ε of a core point, but does not have at least $minPts$ within ε radius of itself, then the point is a non-core point or "edge." The point still belongs to the cluster, however it cannot form paths to other points to grow the cluster. Any points that are not edge points or core points of some cluster are classified as noise.

The algorithm begins by choosing an arbitrary point and checking if there are $minPts$ points within ε distance of the initial point. If enough points are present, a cluster is started and the algorithm examines each point located within ε of the initial point in the same way. This continues until all the edge points of the cluster are found. If enough points are not present within ε of the initially chosen point, then the point is labeled as noise. The next unvisited point is then chosen and the same steps are performed again. After every point has been visited and classified, the algorithm terminates.

To begin, we need to choose ε and $minPts$. Often times this choice is made using domain knowledge and expertise mixed with trial and error. We decided to start by iterating through ε and $minPts$ values until we arrived at a set of clusters that contained some non-duplicate bans. This is due to the data set having many points that are duplicates to one another. Entire clusters could be made with every point being an identical copy of each other.

Figure 1 Possible values for Epsilon and minPts

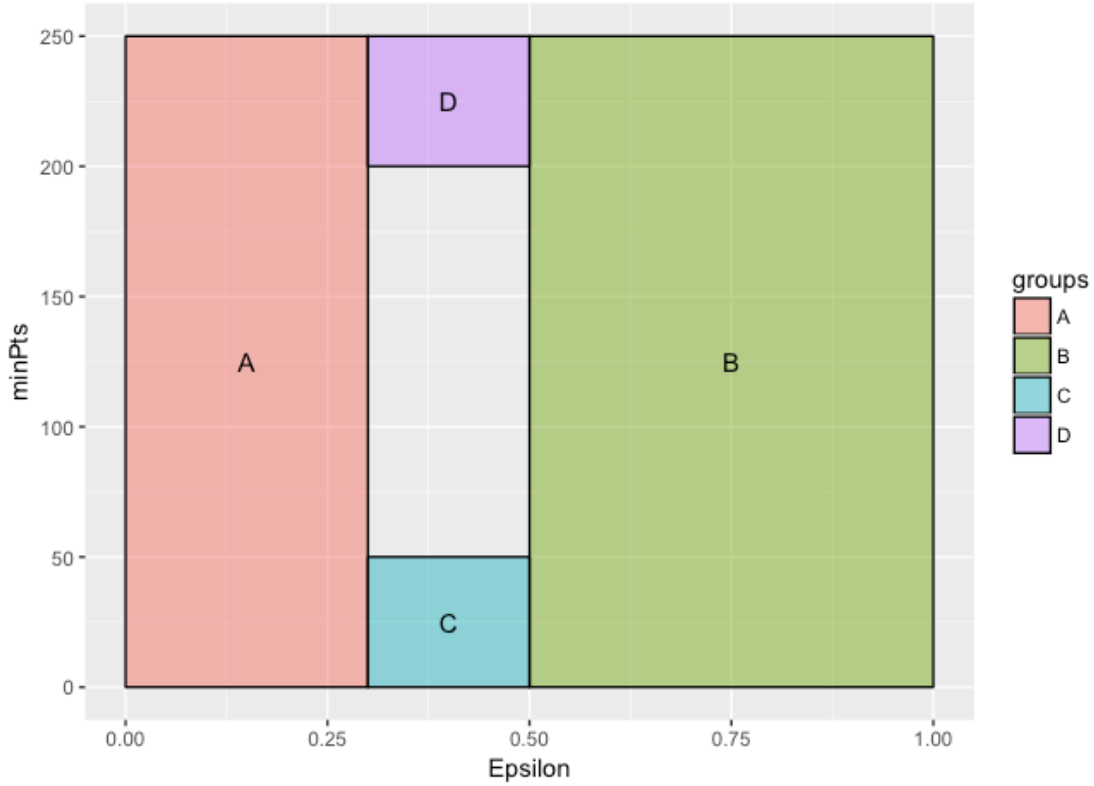


Figure 1 shows the general results produced by different values for ε and $minPts$. Region A (ε below 0.3) always results in all of the data being classified as noise. Region B (ε at or above 0.5) results in only one cluster containing all the data with no noise. Region C (ε between 0.3 and 0.5 and $minPts$ below 50) generates many small clusters, but the clusters just contain duplicate points, meaning that each point in the cluster has a Jaccard distance of 0 with every other point in the cluster. Finally region D (ε between 0.3 and 0.5 and $minPts$ above 200) results in one cluster, but with increasingly large amounts of noise, until the whole dataset is noise. This leaves us with the gray region consisting of ε between 0.3 and 0.5 and $minPts$ between 50 and 200 which produces valid clusters. We found that setting ε to 0.3 and $minPts$

to 100 was a good strategy to achieve defined clusters. We settled on a constant number to make the next step more consistent.

Now that we have some parameters to use, we can use DBSCAN on each patch individually to extract its clusters. Once that is done, we have some number of clusters for each patch representing the *metagame* during that patch. If we can detect a cluster persisting from patch to patch, that means the *metagame* has persisted across those patches as well. Similarly, if a cluster breaks apart going from one patch to the next, we can say that the *metagame* has shifted between those patches. To do this comparison we take the clusters from two different patches and build a new dataset consisting of just the points which comprise those clusters. We then run DBSCAN on the newly created dataset. After DBSCAN is done we look to see if the new dataset separates into different clusters or coalesces into a single cluster. As a final step if the clusters separate, we check the origin patches of the points comprising the clusters. If each cluster is mostly comprised of points from a single patch, we can safely say the patches and therefore the *metagame* have separated. This is done with two patches at a time to make examining the results easier.

After all of the above is done, we're left with clusters which represent the metagame in each patch, but the clusters themselves are still difficult to to parse on a human level. Often the clusters can have over 2,000 points and extracting any actionable insights from them can prove difficult and time consuming. In response to this issue, we employ a final step to give the clusters something of a label which also allows us glean some information quickly. We begin by taking the dissimilarity matrix for each patch, generated as input to DBSCAN, and we separate the matrix by cluster, leaving us with several matrices one per cluster. Next we sum the total

distance for each point and divide by the number of points in the cluster. This gives us the average distance for each point, we call this the “centroid” for the cluster. The centroid is the average set of bans for that cluster, the set of bans with the most similarities to every other set of bans in that cluster. While DBSCAN does not have centroids natively due to the algorithm not assuming the sphericity of clusters, we find the centroids useful for the readability and utility applications discussed above.

Results and Discussion

When DBSCAN is performed on each patch alone we see that exactly one cluster is produced per patch. When comparing each patch to it's immediate chronological successor, we see that almost all of the patches' clusters don't separate. 5.12 and 5.13 clusters coalesce into one cluster. The same is true of 5.13 and 5.14, 5.14 and 5.15, and 5.16 and 5.17. Between patches 5.15 and 5.16 however, the clusters diverge. When the points comprising the clusters from 5.15 and 5.16 are combined and run through DBSCAN, they form two distinct separable clusters. One cluster is made up almost entirely by points from 5.15 and the other by points originating from 5.16. The table below shows that this pattern holds throughout all the patches. Patches 5.16 and 5.17 separate from every other patch except for each other. The only exception is between 5.12 and 5.15.

Table 1. Inter-patch Cluster Comparisons

	5.12	**5.13**	**5.14**	**5.15**	**5.16**	**5.17**
5.12	N/A	No Split	No Split	Split	Split	Split
5.13	No Split	N/A	No Split	No Split	Split	Split
5.14	No Split	No Split	N/A	No Split	Split	Split
5.15	Split	No Split	No Split	N/A	Split	Split
5.16	Split	Split	Split	Split	N/A	No Split
5.17	Split	Split	Split	Split	No Split	N/A

We know from closer inspection that the large change found between patches 5.15 and 5.16 is due to a major game update. The 5.16 patch notes talk about changing an entire class of character, the "Juggernauts." This patch drastically changed several characters and for the duration of the patch these "Juggernauts" saw drastically increased ban rates. This increased ban rate persisted into 5.17 which would explain why those clusters are non-separable. The several patches before the change seem to have rather stable ban rates, the same few characters are banned rather consistently. This seems to suggest that the game was rather stagnant as each patch didn't drastically change the players' banning habits from the previous patch. This doesn't mean that patches 5.12 through 5.15 had no effect, just that the changes between sequential patches were small in comparison. The effects of the patches do seem to build on each other as exemplified by patch 5.12 and 5.15 separating from each other despite the lack of drastic changes in patch 5.12.

The fact that each patch only has a single cluster is strange at first, but it does line up with our experiences in *League of Legends*, where there is usually only one dominant strategy or a small handful of characters that are overbearing at a time. This is especially true following the "Juggernaut" changes described above which shook the *metagame* to its core and warped gameplay for several patches including the *League of Legends* world championship of that year [16]. Our system is general enough that it should handle any instances where several clusters emerge with very little modification, be that in LoL or other games.

Finally, there are some insights and utilities to be gained from examining cluster centroids we calculated before. First, we can use the centroids to quickly estimate the differences between

the clusters. Table 2 is the Jaccard distance measuring the dissimilarity between each of our centroids. We can see that the distance between each centroid closely matches the results of our tests for cluster separation above. Outside of 5.12 being very far apart from each other cluster, the centroid distances perfectly mirror our cluster separation tests. One possible reason for 5.12's distance is that some of the characters in 5.12's centroid are still heavily banned in subsequent patches, but not enough of them and not frequently enough for them to appear in the centroids for those patches. That would still allow for the clusters to merge and remember that the centroids are simplifications of the clusters and not totally indicative of their composition. The centroids simply allow us to get an idea of the cluster's contents quickly.

Table 2. Jaccard Distance between Calculated Cluster Centroids

	5.12	**5.13**	**5.14**	**5.15**	**5.16**	**5.17**
5.12	0.00					
5.13	0.91	0.00				
5.14	1.00	0.67	0.00			
5.15	1.00	0.67	0.29	0.00		
5.16	1.00	1.00	0.91	0.91	0.00	
5.17	1.00	1.00	1.00	1.00	0.29	0.00

Another piece of information illustrated by the centroids is the clusters' relationships to the top banned characters for each patch. One would assume that the top six or so banned characters would be a good indicator of the metagame for that patch, and this is partially true. The top banned characters do represent a large portion of the clusters' compositions, but they don't dictate it fully. This is shown when calculating the distances between a patch's top bans and the

centroids for that patch. For example, patch 5.13's most banned characters are: Ezreal, Ryze, Shyvana, Master Yi, Gragas, and Cho'Gath while that same patch's centroid consists of: Shyvana, Ezreal, Master Yi, Nidalee, Evelynn, and Ryze. So the Jaccard distance would be:

$$\frac{\left|\left|[Ezreal,\ Ryze,\ Shyvana,\ Master\,Yi,\ Gragas,\ Cho'Gath,\ Nidalee,\ Evelynn]\right|-|[Ezreal,\ Ryze,\ Shyvana,\ Master\,Yi]|\right|}{|[Ezreal,\ Ryze,\ Shyvana,\ Master\,Yi,\ Gragas,\ Cho'Gath,\ Nidalee,\ Evelynn]|}$$

or:

$$\frac{8-4}{8}=\frac{4}{8}=\frac{1}{2}$$

Table 3 shows the Jaccard distance between the top 6 character bans per patch and that patch's centroids. We can see that patches 5.12 and 5.17 both have centroids that are equivalent to the top banned characters, but patches 5.13 through 5.16 do not. This shows that while the top bans are worth observing when investigating the metagame, they do not paint the entire picture of what is happening at the time. After all, each game has six unique bans, but the top ban rates do not account for this relationship as they are only calculated per character and not over different combinations of characters so some relational data can be lost if only the top bans are considered.

Table 3. Distance Between Top Bans and Centroids Per Patch

5.12	5.13	5.14	5.15	5.16	5.17
0.00	0.50	0.29	0.29	0.29	0.00

Future Work

Our current system performs well, but a more sensitive system would be of great use. We can see from the various patches that Riot is interested in more than just drastic *metagame* shifts. If that's all they wanted they could just drastically increase or decrease a character's power and guarantee big changes and excitement, but clearly that's not the case. Instead they are concerned not just with intentionally changing things, but also balancing characters that are too strong or too weak and making characters that are played infrequently more appealing to the player base. A more sensitive system would be able to inform Riot when they succeed or fail in these endeavors. A version of our system that takes in various kinds of inputs could also reveal unexpected interactions when changes are made. Perhaps strengthening one character makes another stronger or weaker, or maybe reducing another's power allows an infrequently played character to come back into the *metagame*. Introspection into the formed clusters could show how various components of the game are linked, even if the developers weren't aware of the full breadth of consequences their changes had. All of this data is of great interest to the players as well. The ability to quickly analyze and adapt to changes is of great importance to competitive players who want to win as much as possible. Information about what other players are picking and banning would enable them to create new strategies that work in the current metagame more quickly.

The first improvement that comes to mind would be to use more data from the games played, such as how often a given character is picked or how often a character wins, in addition to the ban data we used here. This would make the system more sensitive to minute changes between patches. Another worthwhile avenue would be the addition of some in-game data like

what items players are purchasing on their characters. Since items are purchasable by all characters, their power and balance is of great import. A single item being too strong can quickly warp the game's landscape, making entire classes of characters incredibly powerful and pushing out others. The previous two points are of particular import because they account for interactions that aren't as apparent when only utilizing pick, win, and ban rates. Because all of these measures are calculated per character with no attention paid to combinations, adding this information to the clustering has the potential to show unknown synergies or interactions both between characters and between characters and items. On the more technical side of things, other clustering algorithms could prove very useful in discerning *metagame* changes. In particular, the OPTICS family of algorithms [17], which are based on DBSCAN, definitely warrant experimentation. These differ from DBSCAN by first linearly ordering the points by their distance from each other, such that the points that are closest to one another are neighbors after the ordering. And finally, as is always the case, a better distance function would increase the clustering accuracy. Jaccard distance was sufficient here but a more sensitive function built specifically for *League of Legends* would improve cluster detection.

One of the biggest improvements we could make, would be to use a better distance function. Finding one that better reflects the differences between a set of bans or a team composition is difficult but may be possible. For example, we know that bans are picked in order, to encapsulate this, we developed a new distance function. The function counts the number of matching ordered pairs in both sets and divides that number by the total amount of ordered pairs, the intersection of the set of pairs over the union of the ordered pairs. If we have

two points $A = [a, b, c]$ and $B = [c, a, b]$ A would have the ordered pairs $[a \rightarrow b,\ a \rightarrow c, b \rightarrow c]$ and B would have $[c \rightarrow a,\ c \rightarrow b,\ a \rightarrow b]$. The total distance would be:

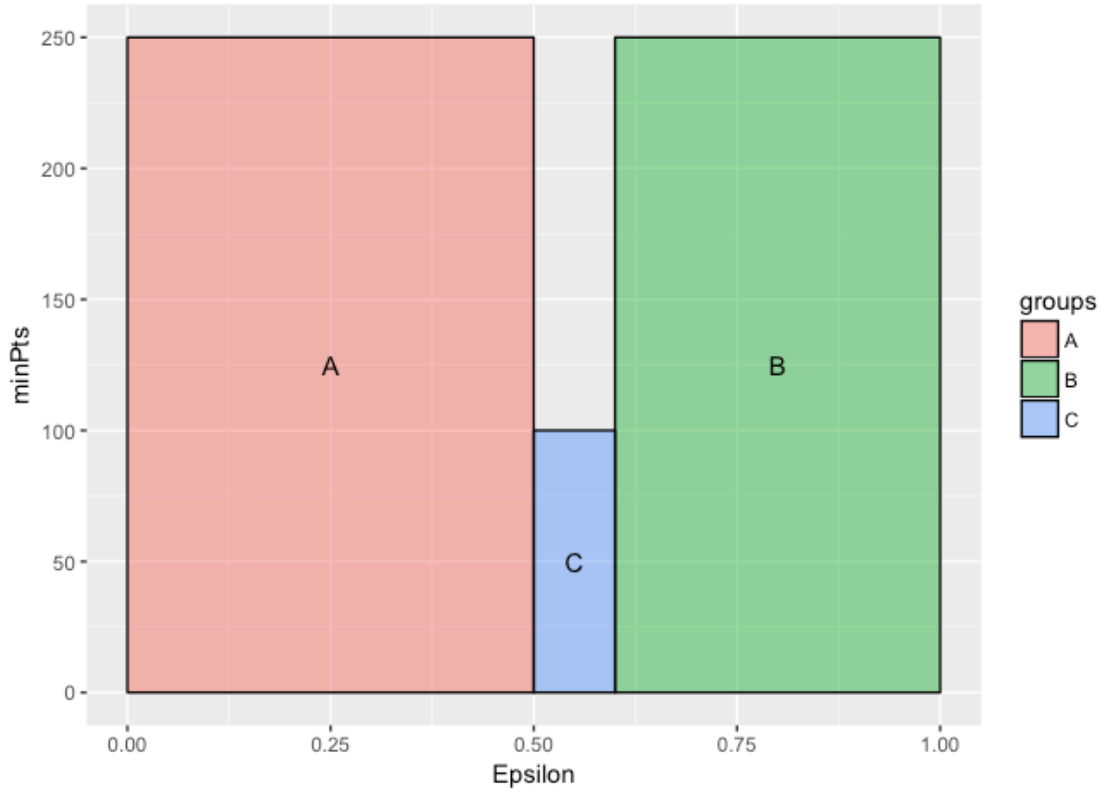
$$1 - \frac{|[a \rightarrow b]|}{|[a \rightarrow b,\ a \rightarrow c, b \rightarrow c, c \rightarrow a,\ c \rightarrow b]|} = 1 - \frac{1}{5} = 0.8$$

This new metric does a good job of accounting for order, but it focuses almost exclusively on that order. These two sets have the exact same points, but because the order changes slightly, the distance between them is very large. To combat this, we combined this new metric with Jaccard distance each equally weighted. This step makes the total distance between A and B 0.4 rather than 0.8, which is much closer.

We used this new distance to run some tests on our data to see how DBSCAN would perform. We specifically looked at patch 5.16 and the first thing we observed is that our old parameters no longer work. ε to 0.3 and $minPts$ 100 simply results in all noise and no clusters. This makes sense given that on the whole our points are now farther apart. Our previous method of iterating through parameter combinations until we discover a cluster or clusters that are not comprised entirely of duplicate points doesn't work as well with this distance metric due to the increased distance between points. We do see, however, that ε lower than 0.5 always produces comparatively small clusters often consisting of less than 800 points and frequently being smaller than 100 points. ε values at or above 0.6 result in a single cluster with varying amounts of noise points depending on $minPts$. ε between 0.5 and 0.6 seems to be the sweet spot as everything between those values always results in at least one significant cluster. This is illustrated in figure 2 below. One interesting observation is that $minPts$ is much less sensitive

than it was with just Jaccard distance. The minimum is around 100 rather than 50, but depending on the exact value of ε we can set $minPts$ much higher, in some cases over 1000, before we start to transform the single strong cluster with noise. It's possible that this new

Figure 2 New possible values for Epsilon and minPts



distance is overvaluing or undervaluing order, but that could be tuned up or down depending on what one believes is a better measure of distance between sets of bans is *League of Legends.*

A potentially more exciting prospect would be adapting our system for use in various other games. Detecting and describing *metagame*s is a challenging problem especially in the general case. Adapting our system could help many other developers find their game's *metagame* quickly and efficiently. This is especially exciting for smaller developers who don't have Riot's resources. With a modified version of our system, small development teams would have a tool

to quickly detect *metagame* shifts and respond if they don't like the way the environment is progressing.

The process wouldn't be too complex, seeing as how most of the system is built on rather game agnostic parts. The distance metric is generic and the data collection only requires a surface level understanding of the game. Given all the complexity of *League of Legends*, results can be gathered with just what characters are banned in the game. These techniques are easily mapped onto other MOBA style games like *Defense of the Ancients 2* and *Heroes of the Storm*, to name two popular examples. Both games have a wide array of characters to choose from and a system of bans to prevent some characters from participating in a given game. A large amount of game specific knowledge wouldn't be required to map our methods onto other genres either. Various competitive shooter games like *Counter Strike: Global Offensive* or *Overwatch* have similar choices players can make. Overwatch has different characters to play as and form teams with, and Counter Strike has a variety of weapons players can use at the start of every round. If the rate at which these things are selected is available, either through a convenient API like Riot has, or via data scrapping, then it should be possible to implement a similar system for those games with minimal changes.

References

[1] "SuperData Research | Games data and market research » Market Brief — 2017 Digital Games & Interactive Media Year in Review", Superdataresearch.com, 2018. [Online]. Available: https://www.superdataresearch.com/market-data/market-brief-year-in-review/. [Accessed: 02- Feb- 2018].

[2] A. Kica, A. L. Manna, L. O'Donnell, T. Paolillo, and M. Claypool, "Nerfs, Buffs and Bugs - Analysis of the Impact of Patching on League of Legends," in *2016 International Conference on Collaboration Technologies and Systems (CTS)*, 2016, pp. 128-135.

[3] M. Wu, S. Xiong, and H. Iida, "Fairness mechanism in multiplayer online battle arena games," in *2016 3rd International Conference on Systems and Informatics (ICSAI)*, 2016, pp. 387-392.

[4] M. Claypool, J. Decelle, G. Hall, and L. O'Donnell, "Surrender at 20? Matchmaking in league of legends," in *2015 IEEE Games Entertainment Media Conference (GEM)*, 2015, pp. 1-4.

[5] P. Beau and S. Bakkes, "Automated game balancing of asymmetric video games," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1-8.

[6] R. Sifa, A. Drachen, C. Bauckhage, C. Thurau, and A. Canossa, "Behavior evolution in Tomb Raider Underworld," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 2013, pp. 1-8.

[7] A. Drachen, R. Sifa, C. Bauckhage, and C. Thurau, "Guns, swords and data: Clustering of player behavior in computer games in the wild," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 2012, pp. 163-170.

[8] A. Drachen et al., "Guns and guardians: Comparative cluster analysis and behavioral profiling in destiny," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1-8.

[9] Hao Yi Ong, Sunil Deolalikar, and Mark Peng, "Player Behavior and Optimal Team Composition for Online Multiplayer Games," *CoRR*, 2015.

[10] James Chen. (2015, August) pcgamer. [Online]. https://www.pcgamer.com/league-of-legends-patch-516/

[11] Rachel Gu. (2015, August) gamespot. [Online]. https://www.gamespot.com/articles/league-of-legends-patch-516-means-drastic-changes-/1100-6429832/

[12] Yannick LeJacq. (2015, August) Kotaku. [Online]. https://kotaku.com/league-of-legends-next-patch-makes-a-lot-of-big-change-1725151910

[13] R Core Team. (2016) R: A Language and Environment for Statistical Computing. [Online]. https://www.R-project.org/

[14] Christian Hennig. (2015) fpc: Flexible Procedures for Clustering. [Online]. https://CRAN.R-project.org/package=fpc

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," , p. 1996.

[16] Daniel Rosen. (2017, July) thescoreesports. [Online]. https://www.thescoreesports.com/lol/news/14464-how-did-this-happen-the-balance-disaster-that-was-worlds-2015
[17] Mihael Ankerst, Markus M. Breunig, Hans-peter Kriegel, and Jörg Sander, "OPTICS: Ordering Points To Identify the Clustering Structure," , 1999, pp. 49--60.
[18] Richard Garfield. (1995) Lost in the Shuffle: Games Within Games. [Online]. https://magic.wizards.com/en/articles/archive/feature/lost-shuffle-games-within-games-2010-06-21-0
[19] Dae-Ki Kang and Myong-Jong Kim, "Poisson Model and Bradley terry Model for predicting multiplayer online battle games," in *2015 Seventh International Conference on Ubiquitous and Future Networks*, 2015, pp. 882-887.

Vita

The author was born in Jefferson, Louisiana and grew up in Chalmette, Louisiana. He obtained his Bachelors' degree in computer science from the University of New Orleans in 2013. He joined the University of New Orleans computer science graduate program and became a member of the Canizaro Livingston Gulf States Center for Environmental Informatics in 2013 and 2015 respectively.