

A machine learning approach for recommending items in League of Legends

Author: Robin Smit

First supervisor/assessor: I.G. (Gabriel) Bucur

Second assessor: prof. T.M. (Tom) Heskes

July 3, 2019

Abstract

League of Legends is the most played video game in the world: its player base has grown from 11.5 million monthly players in 2011 to over 100 million monthly players in 2016. New players have to familiarize themselves with a plethora of game mechanics and concepts, such as strategy, champion abilities and item traits.

Items are important because they provide significant advantages to the players that buy them. They can often mean the difference between winning and losing a match. Because the items available are numerous and have widely varying effects, their presence adds an enormous cognitive load to novice players.

One heavily used tool for learning the basics is the static built-in item recommender, which suggests a fixed set of items for each champion. This system is non-adaptive to specific situations and does not give an indication of what impact an item will have.

This work addresses these flaws by exploring the possibility of creating an item recommender system that *adapts* to specific situations and provides recommendations for items that have the greatest impact on chances of winning the game.

To achieve this, we build a prediction algorithm based on artificial neural networks to *predict the winning team* of a League of Legends game. We will then explore several potential measures of feature importance. Finally, we use LIME as a method locally approximate our classifier and explain its predictions. These explanations are the basis for our recommendations.

1 Introduction

1.1 What is League of Legends?

League of Legends is a multiplayer online real time strategy game, in which ten individuals match up for battle against each other in teams of five. In such a game, each player controls a virtual character called a '*champion*'. The goal of a match is for a player and allied team to destroy a building called the '*Nexus*', located in the enemy base before the allied Nexus is destroyed.

The teams start the game in two opposite corners of the map [Figure 1], where their bases are represented by the large blue and red circles. The teams' bases are connected via three lanes, where multiple defensive structures called '*towers*' are placed (represented by the smaller blue and red dots). In order to conquer the enemy base, all towers in at least one lane must be destroyed. After this criterion has been met, a team can enter their enemy's base to destroy two more towers and ultimately the enemy Nexus.

Located between the lanes are sections of the map called the *jungle*. The jungle contains extra pathways between the lanes, as well as monsters that provide additional resources and bonuses when slain. The gold rewarded by killing jungle monsters can provide an early advantage that could be extended to a significant lead later in the game. The fog of war¹ that covers the jungle poses the threat of an enemy emerging from it. This can create sudden one-versus-two situations known as *ganks*. These aspects play an important strategic role in a League of Legends match.

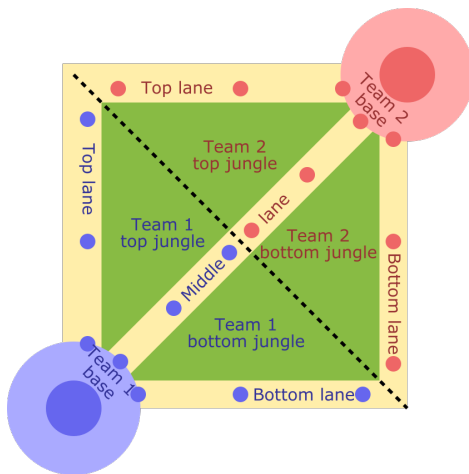


Figure 1: Map of a typical MOBA (multiplayer online battle arena)

¹Fog of war refers to the area that is unseen by a team: the opposite of *vision*. Vision is granted by allied champions, towers and some items.

1.2 Phases

A typical League of Legends match can be divided into three phases: early-game (also called '*laning phase*'), mid-game and end-game [1, 2].

The *laning phase* is the first phase of any League of Legends game. It is where the champions of both the allied and enemy team decide which lane they will primarily defend and exploit until the next phase of the game. The main objective of this first phase is to earn *gold* and *levels* while denying these as much as possible to the opponents. Gold and experience can be obtained by killing minions that spawn in each lane, as well as monsters in the jungle. Usually, a team of five players will take positions across the map as follows:

- One champion in the top lane
- One champion in the middle lane
- Two² champions in the bottom lane
- One champion roaming the jungle

In the *second phase* (mid-game), champions leaving their initial lane (*'roaming'*) and *ganking* play a major role. When given the opportunity, champions will start roaming and seek to gank and kill enemies across the map. This creates a window of opportunity to take small objectives such as towers and jungle monsters (e.g. *Dragon*). Enemies that are most vulnerable to ganks are those that leave the vicinity of the towers in their lane.

The *third phase* (end-game) is used for grouping and *team battles*. Often, all ten players will gather near a major objective (such as *Baron Nashor*³) and fight until one team emerges victorious. The team that wins the battle often gets to take one or more major objectives, thereby making a significant step towards becoming the victors of the match.

1.3 Learning to choose items

During the match, the player has various ways to earn in-game-currency (*'gold'*) that may be spent on in-game items. These items [3] provide positive effects to the player's champion and/or to its allied champions (or negative effects to enemies) and last for the duration of the match. A champion can

²This is done to create a numbers advantage near the pit of the *Dragon*, a jungle monster that gives additional bonuses to the team that slays it.

³Baron Nashor is the most powerful jungle monster and grants a powerful bonus to the team that slays it.

own a maximum of six items at any one time during a match. Different items have different effects, and some items may suit certain circumstances better than others.

New players may ask themselves: *"In which ways do the item choices impact my chances of winning the match?"* Variables that influence the decision-making process for buying items by skilled players often include the interactions between items and champions' abilities, as well as the positioning and behaviour of both allied and enemy champions during the game.

To maximize their champion's chances of winning the match, new players will have to learn what items to buy and when to buy them. They may find guidance from more experienced players, use the static built-in recommender system, or –over the course of multiple matches– learn by trial and error what type of items fit certain situations.

The problem with the first approach is that new players would already need to know someone that could offer this guidance. These more experienced players could provide tailored advice to the newer player as the game progresses and even offer an explanation on the decisions being made. The biggest limitation is availability: not all players have an already knowledgeable group of friends, or their help cannot be solicited when needed.

The second approach is the built-in item recommendation system, which lists a fixed set of items for each champion and is readily available throughout the game. This system, however, does not take into account many of the subtleties in a match – it is not adaptive to specific situations. An ideal set of items does not depend only on which champion is played, but also on the enemy team and *their* items and abilities. The ideal set of items is not static, but continually changes as the game evolves. Another limitation of the current recommendation system is that does not offer any indication of which items can impact the game most at a given time.

In this work we will address these deficiencies by suggesting an recommender system that is *adaptive* and *specifies the effect* that item recommendations have on the chances of winning the match.

In order to give an indication of the importance of the items being purchased within a League-of-Legends match, we will first attempt to answer the question: *'To what extent do items predict the probability of winning a League of Legends match?'* Additionally, we will determine what other (if any) variables of a match are important for predicting the win/lose probability of a team. Finally, we pose the question *'How can we use this information to generate item recommendations?'* and explore a possible solution.

In the process of answering these questions we will learn about designing and implementing an artificial neural network (ANN) [4]. We will also look into feature selection and variable importance in the context of ANNs. Later,

we will use LIME (an explanation algorithm) to explain the behaviour of our ANN locally. We can use these explanations to make item recommendations to League of Legends players.

In section 2 we provide an overview of how we collect and pre-process our data. We show which features are available and how we construct our data sets. In section 3 we explain how and why we use ANNs. We then go on to explore potential ways to determine the importance of features in the data, presenting our results along the way. Section 4 describes a method for recommending items to players at different stages of the game. We conclude with a discussion of our proposed method and offer some suggestions for future work in section 5.

2 Data

2.1 Data Collection

Riot Games, the creators of League of Legends, provide an API [5] through which information on specific matches can be requested. There are three types of requests that are useful for building a data set:

1. A Match-request: Using a `matchId` (a numeric identifier), request general information about that specific match. This includes a list of all ten participants and their `summonerIds` (numeric identifiers for accounts).
2. A Timeline-request: Using a `matchId`, request a second-by-second overview of the events in the game.
3. A MatchHistory-request: Using a `summonerId`, request a list of `matchIds` corresponding to matches played on this account.

While we are downloading the raw data, we switch back and forth between sending Match-/Timeline-requests and sending MatchHistory-requests. When a response to a Match-request is received, we can expand our current database of players with the participants in that match. These participants' `summonerIds` can in turn be used to request new `matchIds`. This process of alternating between types of requests allows us to gradually download match data from the Riot API and grow the data set that we'll use for training a neural network.

In order to be able to train and re-train neural networks often and relatively quickly, we have decided to use only a set of 1000 unique matches and their corresponding 1000 timelines. All of these 1000 matches have been played on the North-American League of Legends server. The patch number is *7.17.200.3955* for each match.

League of Legends has a tier system that groups players with approximately the same level of skill within the same tier. The rank distribution of the League of Legends player base [6] is as shown in [Table 1]. Our data set approximates that distribution, meaning that the majority of our games were played by players in the tiers Bronze through Platinum.

Tier	LoL player base	Our data
Challenger	0.02%	0.00%
Grandmaster	0.05%	0.00%
Master	0.07%	0.10%
Diamond	3.90%	3.64%
Platinum	12.17%	13.82%
Gold	28.08%	27.41%
Silver	35.22%	37.53%
Bronze	17.71%	17.50%
Iron	2.78%	0.00%

Table 1: Comparison between the overall distribution of ranked players in League of Legends and the distribution of our data.

2.2 Data preprocessing

2.2.1 Raw data

Our raw data consists of 1000 `match` objects and their corresponding 1000 `timeline` objects.

`Match` objects contain general information about the match, such as `gameId` and `gameDuration`. Some important variables in the match object pertain to the teams, whereas others are specific to the participants. For instance, each team has the boolean variable `win`, which we will use as a target variable for our prediction algorithm later. Other indicators of the lead of a team over another can be given by variables such as `towerKills` and `baronKills`.

Variables that are significant for each participant of a match are their items and their match performance metrics. The latter consists of variables such as `kills`, `deaths`, `assists`⁴, `goldEarned` and `championLevel`. There are many more variables that pertain to other combat scores, such as the total amount of damage done to and by the player for each different type of damage. This work will not focus on this last category of variables.

⁴The number of `assists` indicates how many times a player has helped an ally kill an opponent.

Timeline objects provide an overview of all the events that happen in a League of Legends match. Each event has a `timestamp` variable and a `type`. We will later use transaction events (`type` \in {`ITEM_PURCHASED`, `ITEM_DESTROYED`, `ITEM_SOLD`, `ITEM_UNDO`}) to calculate a players' inventory at a given time during the game. We will also use the events `BUILDING_KILL` and `ELITE_MONSTER_KILL` to mark when typical phases of the match start/end.

Timeline objects additionally contain some information on each player's progression into the game. Every 60 seconds, variables such as `position`, `currentGold`, `totalGold`, `level`, `xp` and `minionsKilled` can provide insight into where a player stands with respect to the opposing team.

2.2.2 Extracting a game state

After the raw data (matches and timelines) is loaded, we have the final state of the game (in a Match object) and all events from the start leading up to the final state (in a Timeline object). In order to extract the game state at a timestamp $t \in [0, \text{gameDuration}]$, all events in the interval $[0, t]$ have to be considered. As an example: to calculate the items of a player, we start with an empty inventory and record all events occurring up to timestamp t that are of type `ITEM_PURCHASED`, `ITEM_SOLD`, `ITEM_UNDO` or `ITEM_DESTROYED` for that player. We do this for all features we use as input for the neural network.

2.2.3 Splitting the game into phases

We define four particular timestamps based on the events in a game [1, 2]:

- T_{start} is the start of the game. At this stage in the match, nothing has happened yet.
- T_{early} is the timestamp at which the *first tower* is conquered in the match. The loss of a tower and increased roaming potential mark the end of early game, transitioning the match into the mid-game phase.
- T_{mid} is the first timestamp at which *Baron Nashor* (a jungle monster that gives a powerful team bonus) is killed in the match.
- T_{end} is the latest time at which the winner of a match is undecided – within 60 seconds of a match's `gameDuration`.

The data structure we use allows extracting a game state from a (match, timeline)-pair. Doing so requires specifying a timestamp argument $t \in [0, T_{end}]$, which is the in-game time. Using $t = T_{end}$ for each match, we

get a data set for which we expect the neural network to perform well: near the end of the game, it is usually evident which team is winning. From a later result we will see that this is indeed the case.

Let D_{start} , D_{early} , D_{mid} and D_{end} be the data sets that correspond to the timestamps T_{start} through T_{end} . Note that the events that define T_{early} and T_{mid} may not happen in each match: some games end quickly, before the mid/end-game even starts. If, for example, a match is forfeited before the fall of any tower, we do not add a game state for that match to D_{early} . As a result, the data sets D_{early} and D_{mid} are slightly smaller compared to D_{start} and D_{end} . Out of our 1000 matches, in 971 cases at least one tower is conquered. Baron Nashor is slain at least once in 686 of all games. Thus, we have $|D_{start}| = 1000$, $|D_{early}| = 971$, $|D_{mid}| = 686$ and $|D_{end}| = 1000$. In most games, the first tower is killed at approximately 40% of the game duration [Figure 2a] and the first Baron Nashor is killed around 90% into the game [Figure 2b].

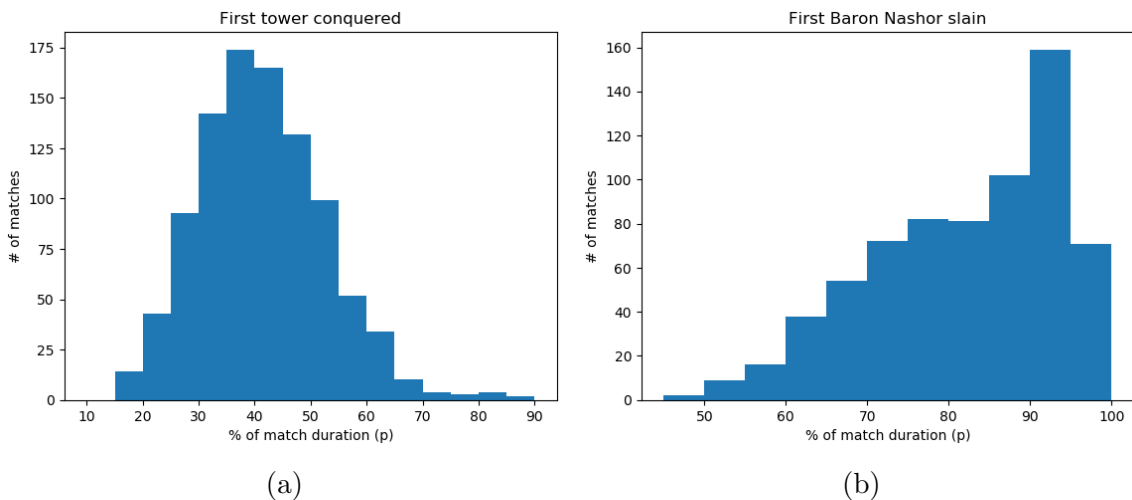


Figure 2: Histograms showing at which percentage of a game significant events (left: tower kills, right: Baron Nashor kills) happen for the first time. The height of a bar indicates in how many matches an event happens first near the percentage (p) of the game shown on the x-axis. Example: when in a 40 minutes long game the first tower falls at 06:48 (=17% of the `gameDuration`), that match contributes to the height of the bar that spans 15%-20% in graph (a).

2.2.4 Feature normalization

The datasets are standardized using the `scale` method from `sklearn.preprocessing`. To initialize the neural network’s weights and biases we use random values. How small these values should be depends on the scale of the inputs and the number of inputs. Without normalization, it might take a long time for the neural network to converge, or it might converge to a suboptimal local minimum. Standardizing the data removes the problem of scale dependence of weight sizes.

3 Neural network setup & results

Our research question necessitates solving the following classification problem: given a game state, which team will win the match? To answer this question, we will build a neural network that gives a distribution of the probability of winning for `team0` and `team1`.

There are various reasons as to why ANNs are well suited to this problem. They are able to generalize by learning patterns from the input that is provided to them. This is important because the number of possible inputs is enormous in comparison to the number of matches that we will provide as a training set.

Neural networks can handle highly non-linear and complex relationships. This is important for our problem because we expect the relation between the inputs and probability of winning the game to be complex and non-linear. ANNs also do not require the data to fit a certain distribution, which is an advantage because of the non-parametric nature of our input data.

Another reason for using neural networks is *scalability*. ANNs are well suited to complex problems with a lot of data. League of Legends is an intricate game and this work will only focus on a very small part of it. Future expansions of this work, however, may need to train classifiers on data sets containing millions of game states. We will discuss possible paths for continuation of this project in the last section.

One of the disadvantages of neural networks is that they are considered ‘black boxes’, meaning that it is hard to understand what causes a neural network to come up with its output. This work will explore some potential measures of variable importance in order to learn which features influence the output of a predictor.

The implementation we will use is the `MLPClassifier` from `sklearn`, where MLP stands for Multi-Layer Perceptron. A perceptron is the basic ‘building block’ of an artificial neural network. It uses *weights*, a *bias* term

and an *activation function* to determine their output from given input. When we create multiple '*layers*' (groups) of perceptrons and connect them, using the outputs of one layer as input for the next layer of perceptrons. The first layer in such a network is called the *input layer*, and the last is called the *output layer*. All layers in between are called *hidden layers*.

The `MLPClassifier` class from `sklearn` implements such a multi-layer perceptron. It is a supervised learning algorithm that can be trained to classify data that is not linearly separable.

The parameters that we use for training the classifier are: `solver="adam"`, `hidden_layer_sizes = (20, 20, 20)`, `learning_rate_init = 0.01`, `tol = 0.01`, `max_iter = 1000`, `alpha = 0.1` and `shuffle = True`.

3.1 Cross-validation

Each of our data sets D_{early} through D_{end} is split into a training set and a test set (sometimes called 'holdout') at a ratio of 6:4. In order to validate our results we use 5-fold cross-validation on the training sets. Five so-called 'folds' are created: we subdivide the training set into five parts. The neural network is then trained five separate times. During each cycle, one of the five parts is assigned the role of *validation set*. The neural network is trained on the remaining four parts and then we compute the accuracies on the validation set.

At the start of a game, both teams should have virtually the same chance of winning the match. As the game progresses we expect one of the teams to build up a lead over their opponents, eventually leading to a victory of the match. As mentioned earlier, our neural network hits an accuracy of approximately 97% using all features for data set D_{end} . Similarly, using D_{start} (corresponding to the start of the game) we get an accuracy of 50% — no better than randomly guessing the winning team. These results indicate that our classifier is working as expected.

3.2 Feature selection

Initially, we looked at only a very small subset of features: `kills`, `currentGold`, `totalGold`, `levels`, `minions`, `building_kills`, `building_deaths`, `tower_kills` and `tower_deaths`. We selected these features because, when significantly different between opposing players/teams, they indicate a lead that could potentially lead to a victory. Training the neural network on just these nine features resulted in an accuracy of 96% (using the D_{end} data set).

To find out to what extent these nine variables have predictive power, we also trained nine classifiers, each using only one feature (instead of all nine

features simultaneously) [Table 2]. Note that the amount of `minions` killed by a player is significantly less important for determining the winner than any of the other eight features.

3.3 Combining features

The first five features mentioned above (namely `kills`, `currentGold`, `totalGold`, `levels` and `minions`) are player-specific statistics. When used for each of the ten players individually, those five features produce 50 separate input variables. We tried combining these variables. Our first step is to group the variables by team (e.g. by calculating `team0_kills` and `team1_kills` instead of using a different `kills` variable for each player). Our second step is to use the relative difference between teams instead of absolute values (such as `team_gold_delta` instead of `team0_gold` and `team1_gold`). We used D_{end} as our data set for training. A comparison of the results is shown in [Table 2].

Both of these steps yielded a slight increase in accuracy for predicting the winning team, even though the total number of input variables was reduced. This shows that the predictive power of the features is given by the relative difference in metrics between the teams instead of absolute values. At the same time only team-level aggregate metrics are sufficient whereas individual player metrics do not improve the accuracy of the classifier.

Feature name (\mathbf{X})	Accuracy using $\text{player}_{j-\{\mathbf{X}\}}$ for $j \in \{0, 1, 2, \dots, 9\}$	Accuracy using $\text{team}_{i-\{\mathbf{X}\}}$ for $i \in \{0, 1\}$	Accuracy using $\text{team}_{-\{\mathbf{X}\}}_{\text{delta}}$
<code>kills</code>	88.9%	90.6%	92.0%
<code>currentGold</code>	87.4%	89.4%	91.9%
<code>totalGold</code>	94.0%	95.8%	96.8%
<code>levels</code>	90.0%	91.6%	93.2%
<code>minions</code>	53.5%	60.5%	65.9%
<code>building_kills</code>	—	93.1%	93.4%
<code>building_deaths</code>	—	93.3%	93.0%
<code>tower_kills</code>	—	93.0%	93.5%
<code>tower_deaths</code>	—	93.8%	93.0%
<i>Number of input variables</i>	10	2	1



Table 2: Predictive power of single preselected features as indicated by the accuracy of neural network classifiers trained on game states from D_{end} . In the first column, we train the classifier on ten variables representing individual metrics of the same feature for each player. In the second column, we aggregate these variables with respect to the two teams, and then train a classifier on these two aggregated variables. Finally, in the last column, we compute the difference (`delta`) between the team aggregate values and then train a classifier on this single variable. For example, in the first row, a classifier was trained initially on ten variables (`player0_kills` through `player9_kills`), then on two aggregated variables (`team0_kills`, which is the sum of `player0_kills` through `player4_kills`, and `team1_kills`, which is the sum of `player5_kills` through `player9_kills`), then finally on a single variable (`team_kills_delta`, which is the difference between `team0_kills` and `team1_kills`). Note that the last four features are team-based, so we cannot attribute them to individual players.

3.4 Prediction using items only

3.4.1 Using variance to select items

When all values of a variable are the same within a data set, we cannot deduce a relation between a predictor and the target variable. In other words: zero-variance variables have no predictive power. As an exploratory step, we will entertain the possibility that conversely, high-variance variables have *more* predictive power.

We had a look at the variance of the input variables and we selected the

twenty items with the highest variance [Table 3]. The nature of these items varies. Some are *basic trinkets*⁵: they are free and everyone gets them [7]. However, players can choose to switch between the different types of basic trinkets. Some high-variance items are cheap starter items, such as  *Cloth Armor*. Others are high-tier ('finished') items, such as  *Blade of the Ruined King*.

Training the neural network on these twenty high-variance items gives us an accuracy of 60%.












Item name	Tier	Variance
 Warding Totem (Trinket)	1	3.21
 Ninja Tabi	2	1.48
 Mercury's Treads	2	1.45
 Total Biscuit of Rejuvenation	1	1.12
 Oracle Alteration	1	0.94
 Farsight Alteration	1	0.92
 The Black Cleaver	2	0.87
 Control Ward	1	0.81
 Sorcerer's Shoes	2	0.75
 Infinity Edge	2	0.74

Table 3: Ten items with the highest variance, sorted by variance in our data. Higher-tier items require one or more item from the previous tier and additional gold.

3.4.2 Using variable importance to select items

Here we take a quick sidestep from training our `MLPClassifier` from `sklearn` and instead use an `ExtraTreesClassifier` [8]. This classifier maintains a list of features ranked by their relative importance. We use this list and again take the twenty highest ranked items [Table 4]. This list still contains some starting items (e.g.  *Doran's Blade*), but consists mainly of finished high-tier items.

Taking these features back to our neural network, it achieves an accuracy of 61% using just these items.

⁵Trinket items are used for managing *vision* around the map, either granting vision to the allied team or showing enemy wards. There are two types of basic trinket items: the 'Warding Totem' and the 'Oracle Lens'. At the start of the game, players get the Warding Totem item for free.





Item name	Tier	Importance
 Infinity Edge	2	0.047
 Lost Chapter	2	0.042
 Rapid Firecannon	3	0.038
 Rabadon's Deathcap	2	0.033
 Liandry's Torment	3	0.031
 Guardian Angel	2	0.030
 Doran's Blade	1	0.026
 Negatron Cloak	2	0.019
 Redemption	3	0.019
 Frostfang	2	0.018

Table 4: Ten items with the highest importance, sorted by importance according to `ExtraTreesClassifier`. Higher-tier items require one or more item from the previous tier and additional gold.

3.4.3 Items owned vs. items bought

Up to this point, we have only considered what is in the players' inventories in a certain game state. In the time before that, it is possible the player buys and uses an item that gets destroyed. The item would not appear in the game state, but still have an effect on the game. Examples of such items are consumables like the  *Health potion* or others like the  *Cull* [9].

For this reason we explore the accuracy of our classifier with the items bought (up to the given timestamp) and compare it to the results we obtained using the items owned (at the given timestamp) [Table 5]. This comparison shows a slight improvement of using the amount of items bought by a team over the amount currently owned, but only for D_{end} .

Data set	Accuracy using Items owned	Accuracy using Items bought
D_{start}	50.25%	49.75%
D_{early}	53.37%	51.16%
D_{mid}	63.35%	59.05%
D_{end}	69.10%	73.30%

Table 5: A comparison of the accuracy of classifier predictions using items owned versus items bought for different data sets

3.4.4 Results using all items in our data

In the best-case scenario our neural network achieved an accuracy of 73% using just items as input features. This is using the data set D_{end} , which is the set of game states at a point where each match is finished. Even though there is some predictive power in the items bought by the players, there is not nearly as much as in the amount of gold a player (or team) has gathered during the match.

In section 4 we will create a measure of variable importance by using LIME, an explanation algorithm.

3.5 Other classifiers

In addition to the `MLPClassifier` class provided by `sklearn`, we tried some other classifiers. These are the results using the full set of features (using item data and other fields) for each of the four data sets (D_{start} through D_{end}) [Table 6]. These results indicate that different classifiers perform very similarly when all features are used as input. When just the items are considered as features, other classifiers perform slightly worse than the neural network [Table 7].

Classifier	D_{end}	D_{mid}	D_{early}	D_{start}
MLPClassifier	97%	81%	67%	50%
GradientBoosting	97%	81%	65%	51%
RandomForest	98%	80%	69%	50%
ExtraTrees	97%	78%	69%	51%

Table 6: Accuracies of classifiers using *all* features (items as well as performance metrics)

Classifier	D_{end}
MLPClassifier	70%
GradientBoosting	65%
RandomForest	68%
ExtraTrees	62%

Table 7: Accuracies of classifiers using *items only*

3.5.1 GradientBoosting

Gradient boosting ([10], chapter 10) is a sequential ensemble ([10], chapter 16) technique where newly added weak learners (decision trees) ‘learn’ from their predecessors’ mistakes. Using gradient descent, this method tries to minimize the loss (given by a *loss function*) when adding trees.

The parameters that we used for training the `GradientBoostingClassifier` are: `n_estimators = 100`, `learning_rate = 1.0`, `max_depth = 1` and `random_state = 0`.

3.5.2 RandomForest

For training the `RandomForestClassifier` ([10], chapter 15) from `sklearn`, a multitude of *decision trees* are constructed. Each tree is built from a sample drawn (with replacement) from the data set. During construction of these decision trees, nodes are split based on the best split from a *random subset* of features (not all features).

The parameters we used for training the `RandomForestClassifier` on the data are: `n_estimators = 3000`, `max_depth = 10` and `random_state = 0`.

3.5.3 ExtraTrees

The `ExtraTreesClassifier` [8] from `sklearn` is similar to `RandomForestClassifier`. The distinction lies in the way thresholds for splits are computed. Thresholds are generated randomly for each feature in the random subset of all features. The best of these is then used as threshold for the split condition.

The parameters we used for training the `ExtraTreesClassifier` on the data are: `n_estimators = 1000`, `max_depth = 3` and `random_state = 0`.

4 Recommending items to players

4.1 LIME as a potential feature importance indicator

LIME [11, 12] stands for Local Interpretable Model-Agnostic Explanations. It is an algorithm that can be used to explain the predictions of a classifier or regressor. By creating an explainer for our neural network, LIME can give insight into which variables are important. The algorithm takes our classifier and a single data point (a game state) as input. It then trains a simplified (linear) model that approximates the classifier. The output of the LIME algorithm is a list of ‘reasons’ and their contribution to the prediction of the classifier. These contributions are based on the weights of the simplified model and we will later refer to them as ‘LIME scores’.

In our case, when we use all available features (not just items) LIME indicates that the amount of penta-/quadrakills in a match is a very important indicator of the winning team. These types of multikills often create windows of opportunity to take important objectives that eventually lead to victory.

When we consider only the items in our data set, LIME gives us an indication of which items play an important role in predicting the outcome of that match. As an example, we take a look at the match with `gameId=2585566722` (which was randomly selected). For this match’s corresponding game state in D_{end} , our classifier predicts that the first team will win with probability $1.9682 * 10^{-10}$. Indeed, the second team is the winner of this match and taking a closer look at the game state used as input tells us why: the winning team is up in items, gold, kills and levels. The wealthiest player on the losing team has less gold than the poorest on the winning team [Figure 3].

Level	Champion	Minions	Lane	K/D/A	Items	Gold (total)	Gold (current)
10		68	TOP	2 / 7 / 1		5608	888
12		21	JUNGLE	2 / 4 / 3		6767	892
11		84	MIDDLE	1 / 2 / 2		5689	339
10		35	BOTTOM	1 / 5 / 4		5631	2481
11		156	BOTTOM	3 / 6 / 1		7643	938
This team loses the match							




Level	Champion	Minions	Lane	K/D/A	Items	Gold (total)	Gold (current)
13		122	TOP	10 / 3 / 3		9799	799
11		44	JUNGLE	3 / 3 / 8		8241	516
12		125	MIDDLE	3 / 1 / 8		8177	477
11		74	BOTTOM	5 / 1 / 9		8310	710
10		6	BOTTOM	3 / 1 / 10		7889	1488
This team wins the match							

Figure 3: Overview of match 2585566722. The column header $K/D/A$ stands for kills, deaths and assists. assists indicate how many times a player has helped an ally kill an opponent.

In this case, according to LIME the items that influence this prediction the most are *Rabadon's Deathcap*, *Liandry's Torment* and *Blade of the Ruined King*. These items are not owned by any of the players in this match, but they are strong late-game items. They can potentially level the playing field or snowball the leading team to victory.

Another item that stands out in the LIME-explanation is that the presence of *Boots of Speed* on the losing team is an advantage to the winning team. *Boots of Speed* are a tier 1 item and can be upgraded by paying extra gold. Four players on the winning team have already done so, versus two on the losing team. LIME picks up on this, pointing out the mobility advantage

of the winning team.

By ‘giving’ the losing team two  *Rabadon's Deathcaps*,  *Liandry's Torments* and  *Blades of the Ruined King* (granted, that is a lot) and running the prediction algorithm again, their chances to win increase all the way to 75.6%.

4.2 General description of the recommender system

Let D be one of our data sets ($D \in \{D_{start}, D_{early}, D_{mid}, D_{end}\}$) and let $x \in D$ be one of the game states. Then calling the `predict_proba` method of our trained classifier (passing data point x as an argument) will return a tuple of probabilities: `predict_proba(x) = (p, 1 - p)`. Here p is the predicted probability that the first team will rise to victory from game state x , and $1 - p$ is the predicted probability that the second team will win the match.

The next step is to use LIME to generate an *explanation* for this prediction. Doing so will give an list of contributing factors to the prediction, paired with the responsible feature. An example of the first five items of such a list is shown in [Table 8].






Reason	LIME score
 Hunter's Talisman $\Delta \leq 0.00$	-0.28
 Trinity Fusion $\Delta > 0.00$	0.25
 Sapphire Crystal $\Delta > 0.00$	0.22
 Maw of Malmortius $\Delta \leq 0.00$	0.20
 Zhonya's Hourglass $\Delta \leq 0.00$	0.17

Table 8: Example of a list of contributing factors to a prediction ($p = 0.99379711, 1 - p = 0.00620289$). LIME scores are weights to those contributing factors: they indicate which feature changes will have the most impact on the prediction. The features are `item_deltas` (Δ), indicating the relative difference between items owned by `team0` and `team1`.

As the final step, for each of the items mentioned in the table we derive two hypothetical game states: one where `team0` has purchased the item (game state x_0) and one where `team1` has purchased the item (game state x_1). We run the prediction algorithm again, producing a two new tuples of win probabilities: `predict_proba(x_0) = (p_0, 1 - p_0)` and `predict_proba(x_1) = (p_1, 1 - p_1)`. This means that when `team0` has bought an extra item in game state x_0 (with respect to x), their ‘gain’ in win chance is $p_0 - p$. Similarly, if the item is purchased by `team1` instead, the winning probability for `team1` increases by $(1 - p_1) - (1 - p) = p - p_1$.

As an example we have taken the items mentioned in Table 8 and calculated the teams’ potential increases in win probability by buying those items [Table 9].






Item bought	team0 win chance gain	team1 win chance gain
 Hunter’s Talisman	-2.92%	-0.50%
 Trinity Fusion	0.60%	7.30%
 Sapphire Crystal	0.61%	40.40%
 Maw of Malmortius	0.61%	18.81%
 Zhonya’s Hourglass	0.60%	9.47%

Table 9: Overview of how much each team stands to gain (in terms of win probability) by buying the item in the first column. Features were selected based on LIME scores [Table 8].

4.3 Usage example

In this example, we consider the match with `gameId=2585584646`. This match was randomly selected from the subset of matches for which both T_{early} and T_{mid} are defined. We give an overview of the game states at these timestamps and the accompanying recommendations that are given at T_{early} and T_{mid} .

At 15 minutes and 27 seconds into this game, the first tower falls: **laning phase ends**. To get a general understanding of the current state of the game, we make a lane-by-lane comparison of both teams using [Figure 4].

Level	Champion	Minions	Lane	K/D/A	Items	Gold (total)	Gold (current)
9		88	TOP	0 / 2 / 1		4056	1205
9		2	JUNGLE	0 / 2 / 5		4648	498
10		120	MIDDLE	3 / 2 / 0		5249	519
9		7	BOTTOM	0 / 1 / 4		3795	1569
8		81	BOTTOM	5 / 1 / 0		5312	1487






This team loses the match

Level	Champion	Minions	Lane	K/D/A	Items	Gold (total)	Gold (current)
10		85	TOP	2 / 1 / 0		4560	885
8		1	JUNGLE	1 / 1 / 2		4144	844
11		130	MIDDLE	3 / 2 / 0		5909	1334
9		108	BOTTOM	2 / 2 / 1		5069	1369
8		20	BOTTOM	0 / 2 / 2		3296	695

This team wins the match

Figure 4: Overview of match 2585584646 after the laning phase. The column header *K/D/A* stands for kills, deaths and assists. *assists* indicate how many times a player has helped an ally kill an opponent.

In the top lane, **team1** seems to have a clear lead: the second player is ahead in levels, kills, items and gold but slightly behind in minions. As we saw from an earlier result though, minions are not as good a predictor of win probability as these other features. In the jungle, the player on **team0** has an advantage. Even though he has no kills and died once more compared to his enemy jungler, the first player is up in levels, items and gold. The mid laners appear to be going even: their levels and kill/death/assist-ratio are equal and the differences in minions, items and gold are minimal. In the bottom lane, players battle two versus two. One of the players on **team0** has five kills, which is a significant lead over the opposing players in bottom lane.

Given this game state, our classifier predicts a win probability of 0.21% `team0` and 99.79% for `team1`. According to LIME, the five features contributing to this prediction the most are those shown in [Table 10]. At this point, we would recommend that `team0` purchases the  *Statikk Shiv* next, and `team1` should purchase either a  *Statikk Shiv*,  *Sunfire Cape* or a  *Ravenous Hydra* as a close runner-up. Both teams should stay away from the  *Runic Echoes* enchantment.










Item	LIME score	team0 gain	team1 gain
 Statikk Shiv	0.39	96.08%	0.21%
 Enchantment: Runic Echoes	-0.26	-0.21%	-68.65%
 Ravenous Hydra	0.26	9.11%	0.20%
 Sunfire Cape	0.24	22.96%	0.21%
 Athene's Unholy Grail	0.24	0.01%	0.01%

Table 10: Evaluation of item options in match 2585584646 at T_{early} . Table shows a list of five items, their *importance* according to LIME and potential *gain in win probability* when bought by the teams. LIME scores are weights to those contributing factors: they indicate which feature changes will have the most impact on the prediction.

The next game state we take a look at is shown in [Figure 5]. We are 32 minutes and 45 seconds into the game and *Baron Nashor* has just been slain: **mid-game ends**. Note that our earlier hypothetical recommendation became reality.  *Statikk Shiv* was purchased by both teams and neither team has the  *Runic Echoes* enchantment. `team1` has also purchased the  *Titanic Hydra*, which is an item quite similar to  *Ravenous Hydra*.

Level	Champion	Minions	Lane	K/D/A	Items	Gold (total)	Gold (current)
15		196	TOP	0 / 4 / 4		10787	437
14		13	JUNGLE	1 / 9 / 15		10689	639
16		220	MIDDLE	7 / 7 / 4		12724	680
14		20	BOTTOM	0 / 3 / 18		10670	764
15		214	BOTTOM	17 / 5 / 2		15783	1833

This team loses the match

Level	Champion	Minions	Lane	K/D/A	Items	Gold (total)	Gold (current)
15		141	TOP	6 / 4 / 10		10460	2110
14		25	JUNGLE	6 / 5 / 13		10787	852
17		283	MIDDLE	7 / 4 / 8		13984	670
16		274	BOTTOM	9 / 6 / 8		14143	908
14		34	BOTTOM	0 / 6 / 13		7936	310

This team wins the match

Figure 5: Overview of match 2585584646 after slaying Baron Nashor. The column header *K/D/A* stands for kills, deaths and assists. **assists** indicate how many times a player has helped an ally kill an opponent.

At this point in the game, our classifier predicts a 1.9% win probability for `team0` and 98.1% win probability for `team1`. The explanation and potential gain for the most important items are shown in [Table 11]. We would recommend to both teams that they should buy *Trinity Force* or *Zhonya's Hourglass*. Both teams should avoid purchasing extra *Control Wards* and the *Enchantment: Bloodrazor* item.






Item	LIME score	team0 gain	team1 gain
 Salvation	0.26	4.35%	0.37%
 Control Ward	-0.20	-1.56%	-43.31%
 Trinity Force	0.20	30.32%	1.68%
 Enchantment: Bloodrazor	-0.19	-1.68%	-39.01%
 Zhonya's Hourglass	0.19	18.25%	1.58%

Table 11: Evaluation of item options in match 2585584646 at T_{mid} . Table shows a list of five items, their *importance* according to LIME and potential *gain in win probability* when bought by the teams.

5 Discussion & Future work

In this work, we have proposed an approach for recommending items in a League of Legends game. The result is a method for recommending that is adaptive to the game state and can provide explanations for choosing items in terms of estimated increases in the probability of winning.

We used ANNs to first build a classifier predicting the chances of winning a match. These ANNs were trained on data sets in which the *phase of the game* is a common factor. We then used LIME to extract the items with the most weight in influencing the prediction of our classifiers. We estimate the impact of buying or selling these items by running the prediction algorithm on a game state that has an item added or removed. We recommend the items that show the greatest improvement of a team's win probability.

This new method is different from the current item recommender that is built into League of Legends in that the new system can adapt to specific situations. Recommendations will change between different phases of the game, whereas the current recommendation system presents a list of items that is the same for each game and fixed throughout a match. The method we proposed can also give an explanation for each item in terms of probability of winning the game. This is potentially useful for novice players that do not yet have a sound understanding of the extent to which each item can influence the game.

One pitfall of our proposed method for recommending items is that it is dependent on the quality of the classifier. Bad classifiers give predictions that have little meaning. As a consequence, recommendations based on those predictions can be less helpful or even steer a team in the wrong direction.

Another thing to note is that the relation we are modeling (items and winning a match) is complex and nonlinear. When we use LIME, the explanations generated are based on a *linear* model that *locally* approximates our

classifier. As a sample data point (a game state) changes, it is probable that the prediction of our classifier diverges from a LIME explainer (learned on perturbations of the game state) very quickly.

5.1 Future work

A path that could be explored is making a distinction between high-rated players and low-rated players. Players in higher tiers are usually more experienced and may be able to better take advantage of the items they buy. It is possible that the importance of itemization is partial to the skill of the player. We currently do not make such a distinction since our dataset contains both high rated and low rated players.

A next step is to focus recommendations on the individual player. Our current recommendation method groups items by team. As a result, our generated recommendations are items that the algorithm thinks a *team* should purchase.

Many of the champions in the game have better synergy with a specific subset of items. How well a champion and an item work together depends largely on the nature of the champion and the *bonuses/effects* an item gives. We did not encode this information into our data sets, but doing so may give further insight into which players specifically can best purchase the recommended items.

The 228 unique items in League of Legends that we use in our data set all have their own set of effects. Some of those properties add bonuses to the same statistic of a champion – for instance: if item A gives +10% attack speed and item B gives +20% attack speed, then purchasing both A and B would increase a champions’ attack speed by 30%. Other effect groups, such as *Armor* and *Armor Penetration* counteract each other when owned by champions on opposing teams. Encoding items in terms of their effects could be the good step towards a deeper understanding of *why* items do (not) work well together. Thinking of items in terms of their properties would make sure the introduction of new items into the game requires little additional training of the neural network.

References

- [1] Auster Delaurant. Advanced game phase breakdown. <https://www.mobafire.com/league-of-legends/build/advanced-game-phase-breakdown-223875>, jun 2012.

- [2] Ehsahn. Beginner's guide to the phases of the game, and explanations of each. <http://forums.na.leagueoflegends.com/board/showthread.php?t=642319>, sep 2011.
- [3] Items in league of legends. <https://leagueoflegends.fandom.com/wiki/Item>.
- [4] Christopher Michael Bishop. *Pattern Recognition and Machine Learning*, chapter 5. Springer, 2006.
- [5] Riot Games. Api. <https://developer.riotgames.com/api-methods/>.
- [6] League of legends rank distribution. <https://web.archive.org/web/20190612021628/https://www.leagueofgraphs.com/rankings/rank-distribution>, jun 2019.
- [7] Trinket (item). https://leagueoflegends.fandom.com/wiki/Trinket_item.
- [8] Louis Wehenkel Pierre Geurts, Damien Ernst. Extremely randomized trees, mar 2006.
- [9] Cull (item). <https://leagueoflegends.fandom.com/wiki/Cull>.
- [10] Jerome Friedman Trevor Hastie, Robert Tibshirani. *The Elements of Statistical Learning*. Springer, second edition, feb 2009.
- [11] Carlos Guestrin Marco Tulio Ribeiro, Sameer Singh. 'why should i trust you?': Explaining the predictions of any classifier, feb 2016.
- [12] marcotcr. lime. <https://github.com/marcotcr/lime>.