DDAM Project – Teamfight Tacting

Alfonso Ferraro, Matteo Garro', Giuseppe Mirko Milazzo, Lorenzo Testa

January 31, 2022

1 Introduction

Teamfight Tacting (TFT) is an auto battler game where each player compete online against seven other opponents by building a team of characters and trying to resist until being the last player standing. Each team is composed of a variable number of characters (also called *units*), depending on the level of the player. Each unit, that has a level measured by its *stars* (from 1 to 3), is semi-randomly associated to a list of 0 to 3 items, that further characterize the unit's capabilities. Moreover, each character has a *trait*, which can be combined with the ones of the other teammates as to create powerful side effects.

The file `TFT_set4_euw_challanger_games.csv` contains records of the teams employed by players in different matches belonging to Set 4 – Fates. Each observation, corresponding to a couple player–match (*puuid–match_id*), is characterized by a series of features, including the composition of the team and the final placement of the player (see Table 1 for the full list of covariates).

The goal of our project is to understand whether the performance of a player in a match can be predicted by the composition of his/her team. In order to do so, we proxy the player's performance using both a discrete response (*placement* and some of its derived features) and a continuous response (*total_dmg_to_players*), given its high absolute correlation (0.894784) with *placement*. Given the different nature of the response variables, we proceed in parallel with both a classification and a regression approach. We perform all the tasks, if not explicitly indicated, using the distributed computing capabilities offered by PySpark.

This report is organized as follows. First, in Sec-

Feature	**Type**
patch_version	string
match_id	string
puuid	string
placement	integer
level	integer
gold_left	integer
last_round	integer
time_in_game	float
total_dmg_to_players	integer
players_eliminated	integer
chosen_unit	string
chosen_trait	string
units	nested dictionary
traits	dictionary

Table 1: List of original features contained in the data set

tion 2 we explain the preprocessing steps that we take in order to prepare our data set for further analysis. Then, in Section 3 we try to understand whether the feature space is naturally separated in groups of observations, i.e. whether clusters of different nature exist and, eventually, how these can be described. At this point, we proceed with the classification task (Section 4) and the regression one (Section 5). Finally, we provide some concluding remarks.

2 Preprocessing

The original data set contains 748586 observations and 14 features. Two of them, *traits* and *units*, are dictionaries. The dictionary *traits* counts the number (value) of traits (key) in each team. We manipulate the feature performing a sort of *one-hot encoding*, by creating as many new columns as the number of distinct traits in the data set. However, as opposed to the one-hot encoding, we mark the presence of a given trait in a team by inserting the number of characters with such a trait (i.e. the dictionary value) within the team. By doing so, we increase the dimensionality of the data set of 26 features. The second dictionary, *units*, characterizes the characters in the various teams. It has the characters' names as keys and nested dictionaries, which include the characters' stars and items, as values. We repeat the one-hot encoding, by pivoting the data set as to "explode" the nested dictionaries. In particular, we create 58 new columns indicating the level of stars of each unit in a team, or 0 if the character is not in the team. For example, we create the column *Nidalee* (a champion's name), filling it with the stars of the champion when it is employed in a team, and 0 otherwise. By doing so, we are aware that we are losing a potentially important piece of information (the items associated to each character), but, unfortunately, accounting for it would make analysis unfeasible. In former trials, we have employed such a strategy by exploding the data set creating as many columns as the number of different couples character–item, ending with more than 3000 features, which are too many to be used in any further analysis. Therefore, we decide to reduce the computational burden at the cost of losing a piece of information. After having unpacked the two dictionaries and having dropped some variables (*units* and *traits*), we end up with a data set of 96 columns. Note that in order to extract the information from the dictionaries, we partially employ Pandas, as PySpark does not offer the possibility to manipulate nested dictionaries such as the ones contained in the column *units*.

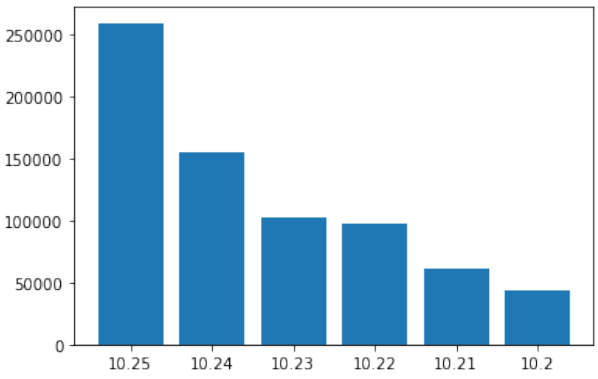

Figure 1: Histogram of *patch_version* after the removal of patches 10.19 and 11.10

Pivoting traits and units creates a considerable amount of missing values, which are imputed by setting them to 0.

Furthermore, as different patches of the game are released over time, we restrict our analysis to the ones 10.20, 10.21, 10.22, 10.23, 10.24 and 10.25 (removing 10.19 and 11.10), in order to work on a consistent sample of data. We end up with 717072 observations, corresponding to 89634 matches, whose distribution over the patches is shown in Figure 1. Similarly, Figure 2 shows the histogram of *total_dmg_to_players*, our response for the regression task.

Finally, we prepare different responses for the classification task. In particular, starting from *placement* we create:

- A dummy indicating the final placement in the first position as 1, otherwise 0 (thus creating an unbalanced situation);
- A variable with 4 classes (first, second and third places, 0 otherwise);

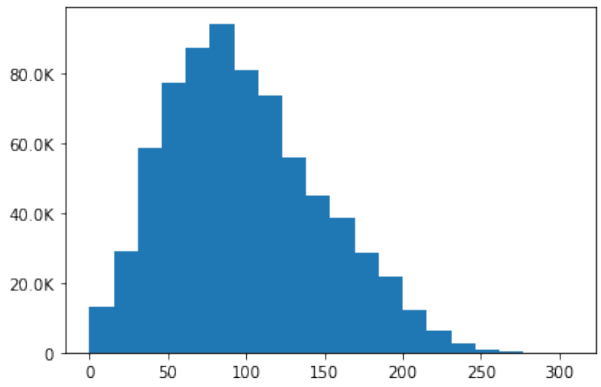

Figure 2: Histogram of *total_dmg_to_players*

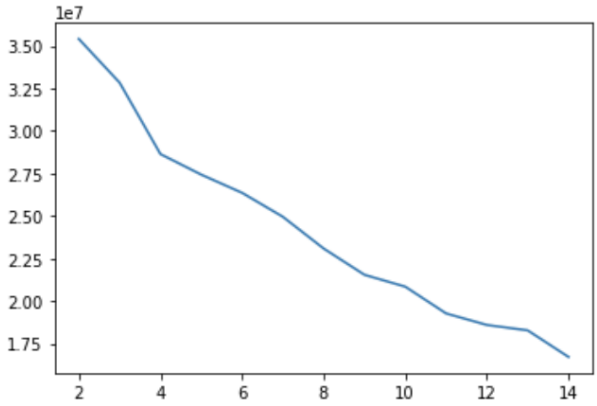

Figure 3: Elbow plot, k-Means clustering, Euclidean distance and SSE error

- The *Placement* column with its 8 classes

We are now ready to proceed with the clustering task, which can help us in understanding the morphology of the space we are investigating.

3 Clustering

We perform clustering analysis on the data set in order to discover interesting patterns in the data. Before proceeding, we drop some columns from the data set (*patch_version*, *match_id*, *puuid*, *chosen_unit*, *chosen_trait*, *placement*, *level*, *gold_left*, *last_round*, *time_in_game*, *total_dmg_to_players* and *players_eliminated*). In doing so, we try to explore the space of *units* and *traits*. We implement the k-Means algorithm, keeping quite low the number of iterations (equal to 10) – a choice that allows us to explore more values of k and different metrics. We tune k with both the Euclidean and the cosine distance. Results achieved with the latter are in line with the ones obtained by using the former distance. Thus, in the following, we refer to the k-Means algorithm with Euclidean distance. With regard to k, we investigate the interval $[2, 14]$ with a step size of 1. Figure 3 shows the *Elbow plot* on the specified range of values. Clearly, there is not an evident k to choose. Anyway, we try to work with $k = 4$ and $k = 9$, which are the levels of k where the rate of decrease of the SSE changes the most.

Cluster	Avg	Max	Min	Med	Size
0	4.41	8	1	4	282699
1	4.525	8	1	5	283012
2	4.638	8	1	5	51069
3	4.595	8	1	5	109452

Table 2: Summary statistics computed on the distribution of *placement* in the four clusters obtained by the k-Means algorithm with $k = 4$

We compute various summary statistics related to the distribution of *placement* in the clusters obtained with k-Means. In particular, we compute the maximum, the minimum, the mean and the median of *placement*, in order to understand whether there are clusters with relevant levels of such a variable. Table 2 and Table 3 show the values of such summary statistics of placement in the clusters obtained with $k = 4$ and $k = 9$, respectively.

It is possible to see that, for $k = 4$, cluster 0 seems to be the one with the lowest average placement, that is the cluster with the best players belonging to it. Among the other clusters, even if they show a higher average placement, cluster 1 has the best value. When $k = 9$, it is still possible to notice that cluster 2 and cluster 4 have an average placement below 4.40, followed by clusters 0, 5 and 8 with an average placement between 4.47 and 4.50.

Cluster	Avg	Max	Min	Median	Size
0	4.498	8	1	5	22274
1	4.615	8	1	5	48343
2	4.376	8	1	4	29604
3	4.619	8	1	5	82027
4	4.389	8	1	4	192068
5	4.479	8	1	4	48168
6	4.543	8	1	5	229594
7	4.549	8	1	5	33450
8	4.485	8	1	4	40704

Table 3: Summary statistics computed on the distribution of *placement* in the nine clusters obtained by the k-Means algorithm with $k = 9$

In order to further characterize the clusters, we analyze the most frequent characters and traits by cluster. By choosing a minimum frequency threshold (the minimum number of matches, in percentage, that a character/trait has to be employed in order to be considered characterizing for the cluster), we can study the most frequent (and thus characterizing) characters and traits for each cluster. Interestingly, there is not much overlapping between groups. Table 4 shows the most frequent characters and traits for the cluster with lowest average prediction for $k = 4$, applying a threshold of 0.3. Similar results can be drown using $k = 9$. In particular, Table 5 shows the most frequent characters of clusters 2, the one with minimum average placement.

As a final remark on the clustering task, it is possible to see that the champions and the traits chosen in both cases are roughly the same, suggesting that they are probably an optimal choice made by the players who want to maximize their chances to win.

4 Classification

As explained during the preprocessing step, we carry out different classification tasks. First, we try to predict the variable *placement* as it appears in the original data set, i.e. with 8 possible values. Then, we focus on a four class response variable (shrinking placements from 4 to 8 towards 0) and a two class

Unit	Trait
Ashe	Adept
Diana	Assassin
Irelia	Brawler
Janna	Cultist
Jax	Dazzler
Kindred	Divine
LeeSin	Duelist
Lissandra	Elderwood
Lux	Enlightened
Morgana	Exile
Pyke	Hunter
Shen	Moonlight
Talon	Mystic
Warwick	Ninja
Yone	Spirit
Yuumi	Warlord
Zilean	

Table 4: Characterizing units and traits of cluster 0 found by k-Means with $k = 4$. Frequency threshold at 30%

Unit	Trait
Fiora	Adept
Irelia	Cultist
Janna	Divine
Jax	Duelist
Kalista	Enlightened
LeeSin	Exile
Shen	Mystic
XinZhao	Ninja
Yasuo	Warlord
Yone	
Zilean	

Table 5: Characterizing units and traits of cluster 2 found by k-Means with $k = 9$. Frequency threshold at 30%

Metric	Training Score	Test Score
Accuracy	0.280	0.267
Precision	0.254	0.227
Recall	0.280	0.267

Table 6: 8-class classification. Results of the Random Forest classifier on *Training set* and *Test set*, without hyperparameter tuning

Metric	Training Score	Test Score
Accuracy	0.288	0.288
Precision	0.256	0.256
Recall	0.288	0.288

Table 7: 8-class classification. Results of the Random Forest classifier on *Training set* and *Test set*, with hyperparameter tuning

variable (shrinking towards 0 everything but the first placement). Of course, the last two cases are unbalanced, in particular the last one having 87.5% (7 over 8 possibilities) of the values equal to 0.

As in the clustering task, we drop from the data set the columns which are highly correlated with the response variable *placement* (otherwise the task would be trivially solved). The rationale of this choice is the following: each feature which can be computed only at the end of the game, e.g. *time_in_game* and *players_eliminated*, does not provide any information that can be exploited by a player prior to participate to a match. Therefore, we use in our prediction tasks only those features which are available to the player before actually entering a match, as to provide meaningful and non-trivial information. A positive side effect of our choice is that PySpark's *Vector assembler* has to store the index of few features, as it only accounts for non-zero values.

4.1 8-class classification

We first employ a Random Forest classifier as to deal with the task of 8-class classification. We randomly split the data in a *Training set* and *Test set* with a proportion of 70% and 30%, respectively. We do not apply any hyperparameter tuning at the beginning, arbitrarily choosing 10 as the number of trees to grow. This simple choice allows us to get a feeling of the complexity of the classification task. Results are shown in Table 6, for both the *Training set* and the *Test set*.

We now try to tune hyperparameters as to achieve better performances, using a k-fold cross validation with $k = 5$. We tune the following hyperparameters:

- Number of trees (10,15,20)
- Maximum depth of each tree (5,7)

The best configuration is the one with 20 trees and maximum depth equal to 7. Results are slightly better than before, as it can be appreciated in Table 7.

In addition, we exploit the Random Forest Classifier to measure the importance of the various features, as shown in Figure 4. It is straightforward to note the similarity between the characters and traits highlighted by the RF feature importance plot and the ones defined as characterizing found during the clustering task.

Moreover, we also try to implement a Naive Bayes classifier, but its results are really discouraging: the accuracy, the precision and the recall on the *Test set* are always below 0.12.

4.2 4-class classification

We now try to perform the classification task with the 4-class variable. As opposed to the previous case, given the unbalanced nature of the response, we random undersample the majority class (the players ranked from 4 to 8, i.e. the observations with value 0) after the split in *Training set* and *Test set*.

We fit again the Random Forest classifier, which performs quite well with and without hyperparameter tuning. With the 4-class output, the best configuration is the one with 10 trees and maximum depth equal to 7. Results are shown in Table 8 and Table 9. Feature importance of 4-class classification can be observed in Figure 5.

We are able to achieve better results even with the Naive Bayes classifier, reaching an accuracy and a recall over 0.48 and a precision of about 0.53.

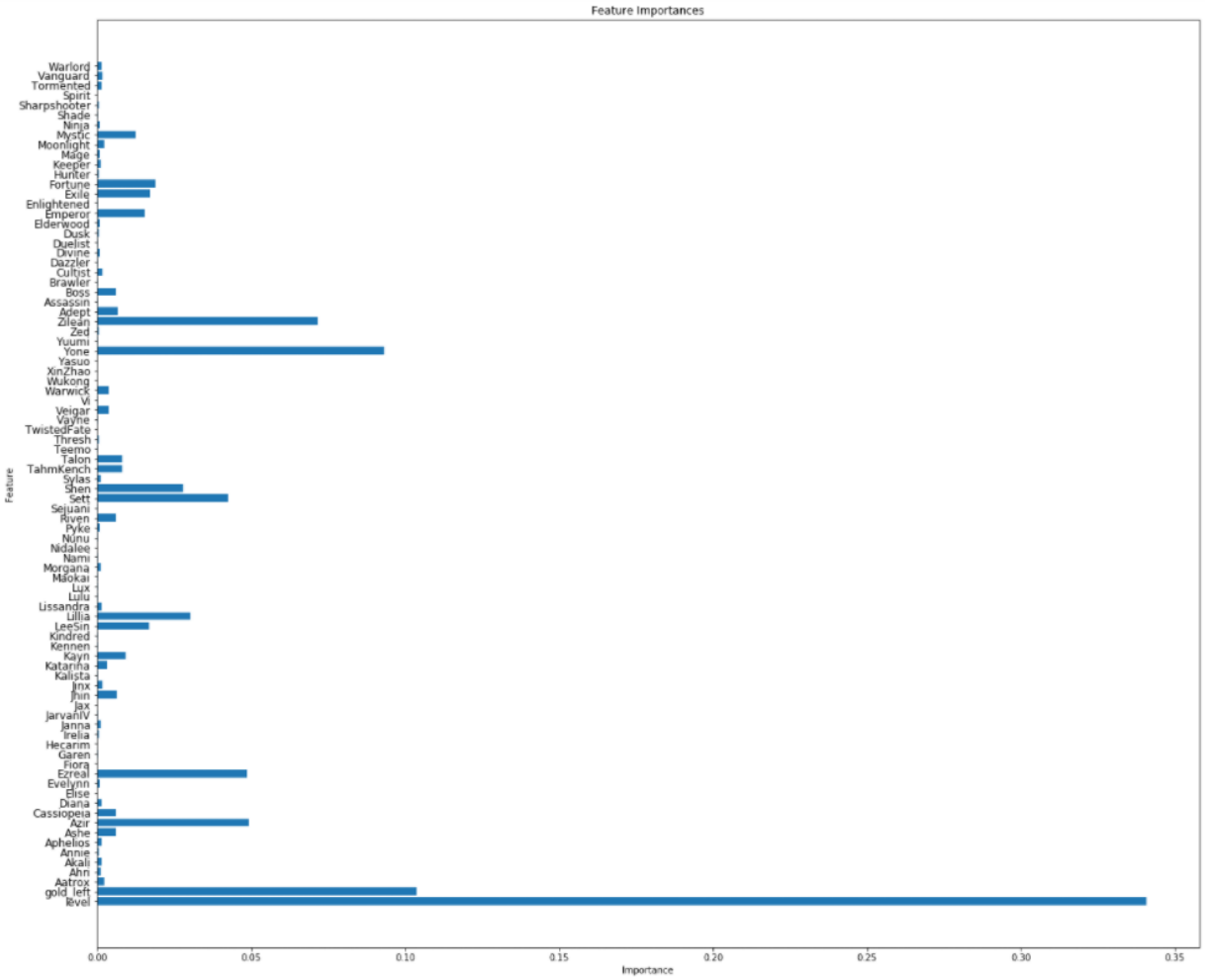

Figure 4: Random Forest feature importance, 8-class classification task

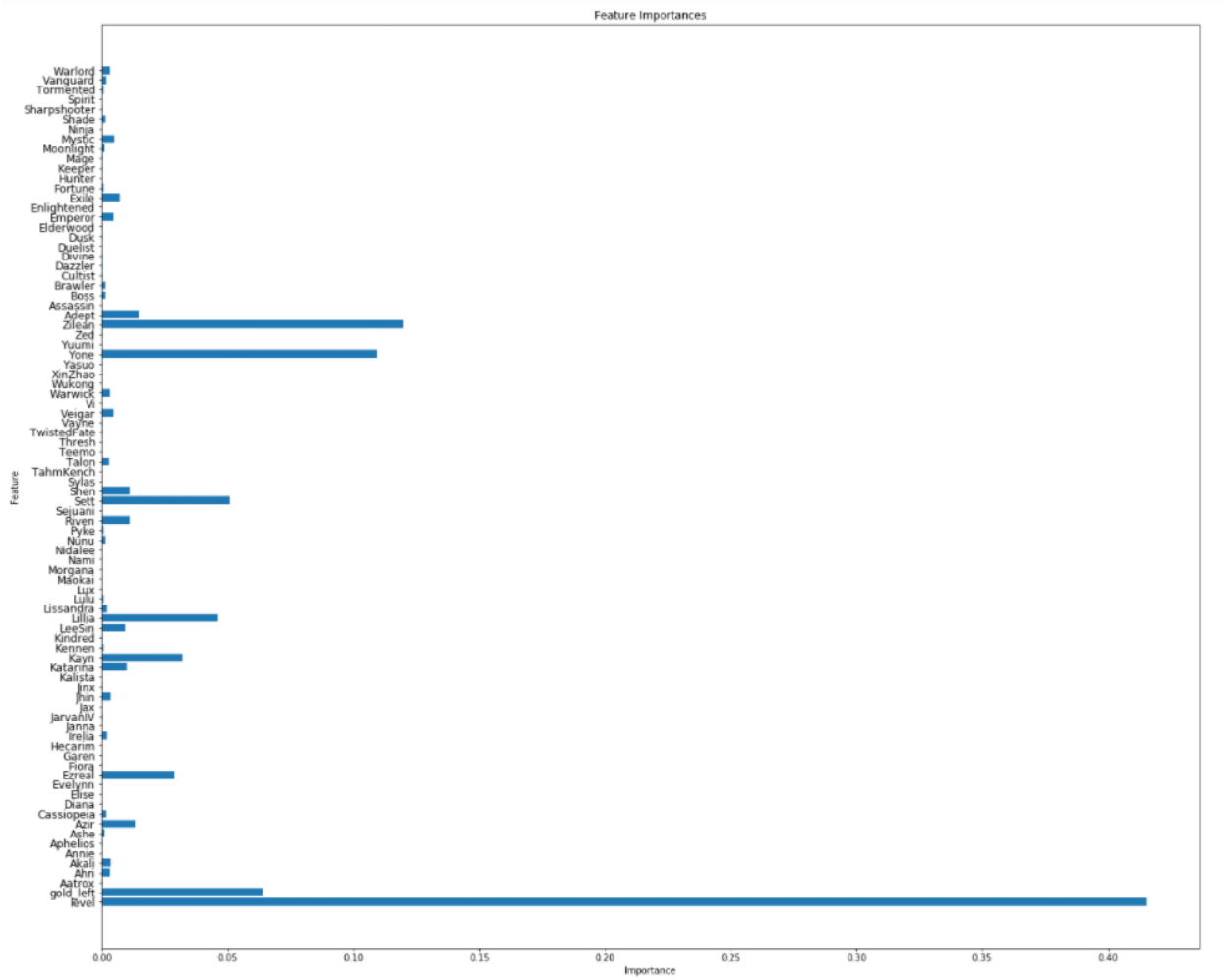

Figure 5: Random Forest feature importance, 4-class classification task

Metric	Training Score	Test Score
Accuracy	0.662	0.664
Precision	0.662	0.664
Recall	0.562	0.563

Table 8: 4-class classification. Results of the Random Forest classifier on *Training set* and *Test set*, without hyper-parameter tuning

Metric	Training Score	Test Score
Accuracy	0.672	0.674
Precision	0.672	0.674
Recall	0.567	0.570

Table 9: 4-class classification. Results of the Random Forest classifier on *Training set* and *Test set*, with hyper-parameter tuning

4.3 Binary Classification

We now turn to binary classification, trying to predict whether a player can get the first place or not. Clearly, the response is highly unbalanced (93616 positive class, 655312 negative class). Thus, after having split the data set in a stratified fashion, we random undersample the majority class.

At this point, we fit a variety of methods, both with and without hyperparameter tuning. In the version with hyperparameter tuning, we use GridSearchCV (k-fold cross validation with $k = 5$) to explore the parameter space. We report results directly for the *Test set*.

We first fit a Random Forest classifier. The results of the version with hyperparameter tuning (optimal parameters: 10 trees and 5 as maximum depth) are shown in Table 10 and the associated confusion matrix is in Table 11.

We exploit the Random Forest classifier once again to appreciate the importance of each feature in determining the output (see Figure 6). The feature importance shows that, overall, the most important predictors for the binary classification task are *level* and *gold_left*. However, there are some champions that are very good predictors, as well as some *traits*.

Metric	Test Score
Accuracy	0.802
Precision	0.958
Recall	0.807
F1	0.828
AUC ROC	0.786

Table 10: Binary classification. Results of the Random Forest classifier on *Test set*, with hyper-parameter tuning

	Class 1 pred	Class 0 pred
Class 1 act	20580	6337
Class 0 act	34824	145852

Table 11: Confusion matrix, Random Forest classifier, binary classification

We also fit a Multilayer perceptron. The optimal architectural parameters, found by GridSearchCV, are 3 as layers (86, 24, 2 units) and 15 as maximum iterations. Table 12 shows the results and Table 13 shows the confusion matrix. Overall, the Multilayer perceptron shows worse performances than the Random Forest classifier and has longer training times.

We finally fit a Logistic Regression. The Logistic Regression model performs almost as the Random Forest classifier with slightly more accuracy and precision, but slightly less recall. Table 14 shows the results achieved with the tuned model ($maxIter = 12$, $regParam = 0.1$, $elasticNetParam = 0.2$), while Table 15 shows the relative confusion matrix.

Metric	Test Score
Accuracy	0.691
Precision	0.953
Recall	0.679
F1	0.741
AUC ROC	0.727

Table 12: Binary classification. Results of the Multilayer Perceptron on *Test set*

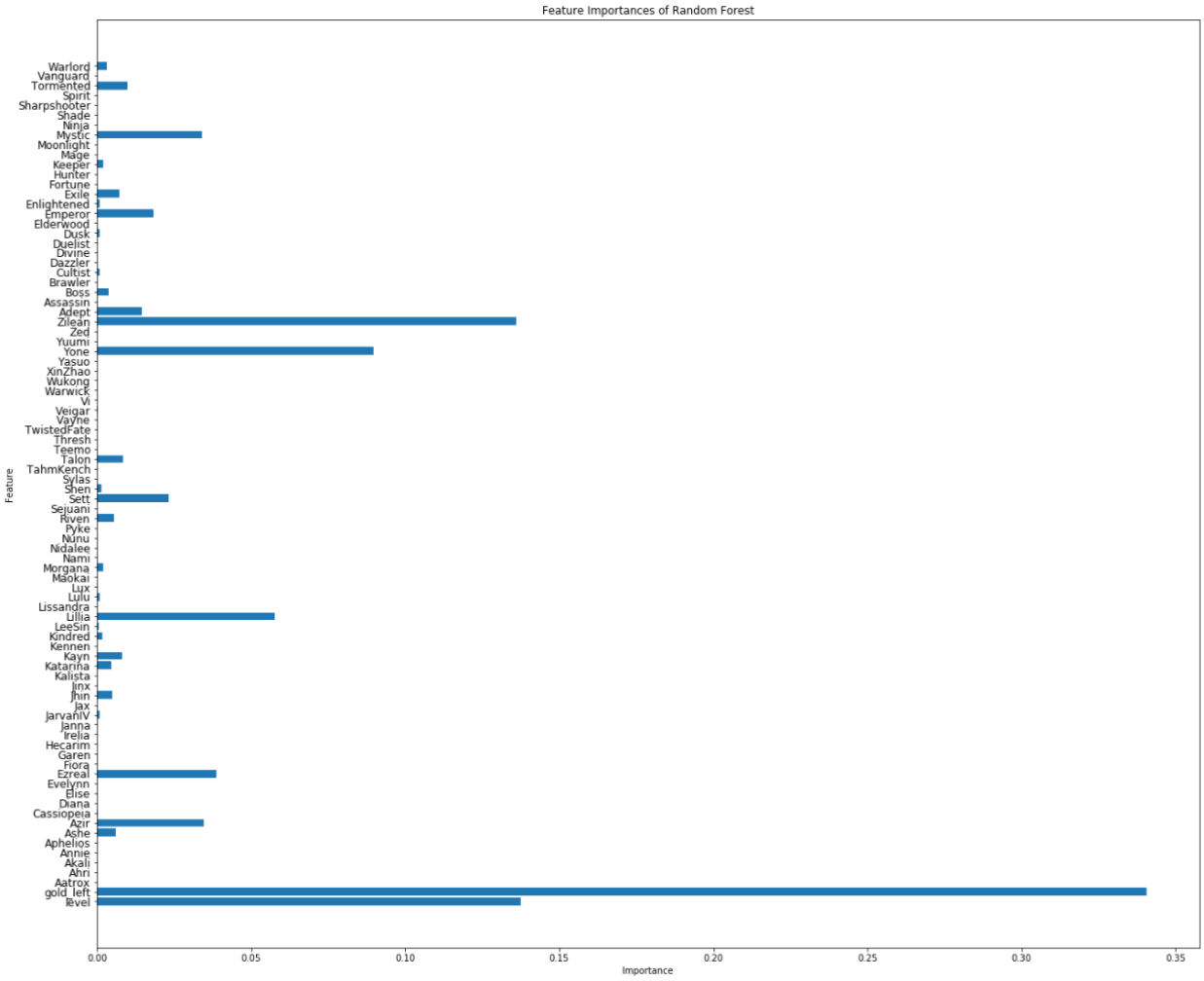

Figure 6: Random Forest feature importance, binary classification

	Class 1 pred	Class 0 pred
Class 1 act	20845	6072
Class 0 act	58066	122610

Table 13: Confusion matrix, Multilayer Perceptron, binary classification

Metric	**Test Score**
Accuracy	0.802
Precision	0.958
Recall	0.807
F1	0.828
AUC ROC	0.786

Table 14: Binary classification. Results of the Logistic Regression on *Test set*

	Class 1 pred	Class 0 pred
Class 1 act	20580	6337
Class 0 act	34824	145852

Table 15: Confusion matrix, Logistic Regression, binary classification

RMSE	**R2**
37.4219	0.422

Table 16: Model summary of the Decision Tree Regressor

5 Regression

We finally fit a Decision Tree for the regression task, where we try to predict the total damage to players depending on all the units and traits features. Results are shown in Table 16. The model is able to explain 42.2% of the variability in the damage to the other players.

6 Conclusion

By distributing the computational burden using PySpark, we are able to perform complex analysis on a massive data set. Although the PySpark's interface is not easy to interpret, we associate the placement (measured in different fashions) of a player to the composition of his/her team. Of course, more analysis could be done to refine our results. For example, a greater search among the hyperparameters could surely increase our scores. Moreover, we could use the cluster labels in the classification and regression tasks, as they could contribute to the prediction quality.