PLAYERUNKNOWN'S BATTLEGROUNDS

FINISH PLACEMENT PREDICTION

Anqi Mao | Guancheng Yao | Shyam Subramanian | Yang Mo

Overview

Our final project is based on the Player Underground Battleground Finish Placement Prediction Competition on Kaggle. It is the winning placement percentage of each player in the match that we need to predict. After preprocessing the data, We conduct feature engineering by transforming the data, adjusting some variables and extracting the important features, and achieve some visible improvements on the results of our models. 8 models are fitted in total, including basic models like zero-R, linear regression and beta regression, and advanced models, such as PCA combined with Lasso, Random Forest, Neural Network, XGBoost and LightGBM. After several trials of tuning the parameters, the final best mean absolute error we achieved is 0.028, which is generated by the model LGBM. Our final ranking is at top 28% on the Kaggle leaderboard.

Content

1. Introduction

1.1 About the competition

'PUBG' is the abbreviation of 'PlayerUnknown's BattleGrounds', which is a popular battle royale-style video game. 100 players are dropped onto an island empty-handed and must explore, scavenge, and eliminate other players until only one is left standing, while the play zone continues to shrink.

1.2 The Dataset

The competition provides 4446966 player records in the given training data set, covering the players' performances in a single match and the general game. We need to predict the 'winPlacePerc', which is a percentile winning placement with 1 corresponds to 1st place and 0 corresponds to last place in the match.

There are 28 predictors and 1 response. The description of the variables is as following.

Table 1 Variable Description

Variable	Description	Type
Id	Player's Id	object
groupId	ID to identify a group within a match.	object
matchId	ID to identify match.	object
assists	Number of enemy players this player damaged that were killed by teammates.	int64
boosts	Number of boost items used.	int64
damageDealt	Total damage dealt	float64
DBNOs	Number of enemy players knocked.	int64
headshotKills	Number of enemy players killed with headshots.	int64
heals	Number of healing items used.	int64
killPlace	Ranking in match of number of enemy players killed.	int64
killPoints	Kills-based external ranking of player.	int64
kills	Number of enemy players killed.	int64
killStreaks	Max number of enemy players killed in a short amount of time.	int64
longestKill	Longest distance between player and player killed at time of death.	float64
matchDuration	Duration of match in seconds.	int64
matchType	String identifying the game mode that the data comes from.	object
maxPlace	Worst placement we have data for in the match.	int64
numGroups	Number of groups we have data for in the match.	int64
rankPoints	Elo-like ranking of player.	int64
revives	Number of times this player revived teammates.	int64
rideDistance	Total distance traveled in vehicles measured in meters.	float64
roadKills	Number of kills while in a vehicle.	int64
swimDistance	Total distance traveled by swimming measured in meters.	float64
teamKills	Number of times this player killed a teammate.	int64
vehicleDestroys	Number of vehicles destroyed.	int64
walkDistance	Total distance traveled on foot measured in meters.	float64
weaponsAcquired	Number of weapons picked up.	int64
winPoints	Win-based external ranking of player.	int64
winPlacePerc	The target of prediction.	float64

1.3 Evaluation Metric

Submissions are evaluated on *Mean Absolute Error* between the predicted ‘winPlacePerc’ and the observed ‘winPlacePerc’.

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}.$$

Where

yi represents the predicted response.

xi represents the observed response.

n is the number of observations in the test dataset.

2. Data Preprocessing

Our data preprocess includes two parts. Firstly, we use Python and R to do the visualization and investigate the data. After looking into the data, we detect the outliers along with NAs in the dataset which can probably cause models to be inefficient.

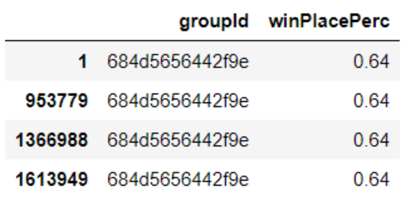
2.1 Data Visualization

In the total 28 predictors, 24 of them are numeric variables and the other 4 are objects, which is Id, groupId, matchId and match type. The three ids identify the players’ information of each group in each match they participated. And the match type indicates that the dataset contains players’ performance and placement from 16 different match type.

matchType			
solo	solo-fpp	normal-solo	normal-solo-fpp
duo	duo-fpp	normal-duo	normal-duo-fpp
squad	squad-fpp	normal-squad	normal-squad-fpp
flaretpp	flarefpp	crashtpp	crashfpp

Table 2 Match types

Players playing solo-match have their own placement, while the players from the same group share the same placement.



	groupId	winPlacePerc
1	684d5656442f9e	0.64
953779	684d5656442f9e	0.64
1366988	684d5656442f9e	0.64
1613949	684d5656442f9e	0.64

Table 3 Players from the same group share the same winning placement

The number of groups in each match type is different. The picture of numGroups tells the feature of match type even if you are not a player of the game, i.e. In solo matches, there are around 100 single players, each of whom is a group. And in terms of Duo, the yellow curve indicates that it is a two-person match type, with around 50 groups in each match. So does the squad.

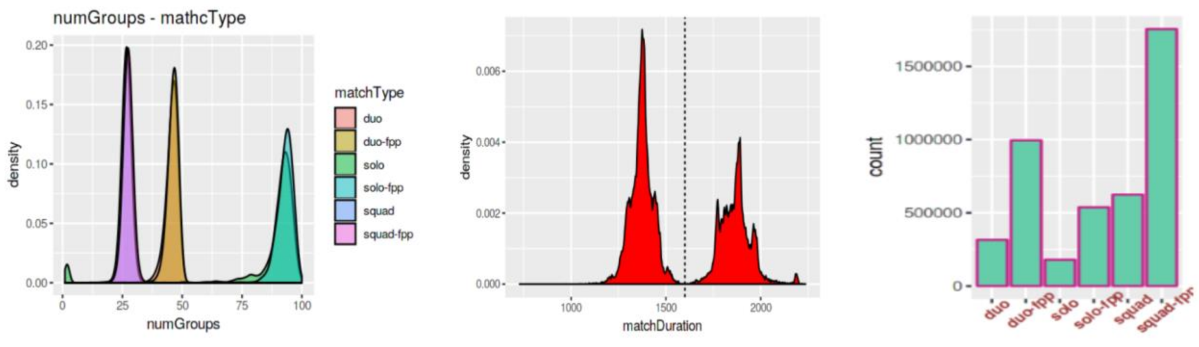


Figure 1 Match type statistics

The players' killing statistics tell an interesting story. These pictures indicate that in the killing game, the majority of the players kills no one in a single match in all forms of killing methods, whether by themselves or by assisting teammates. Hence, the minority of the players who are really experienced and skilled determine the leaderboard at the end of the match.

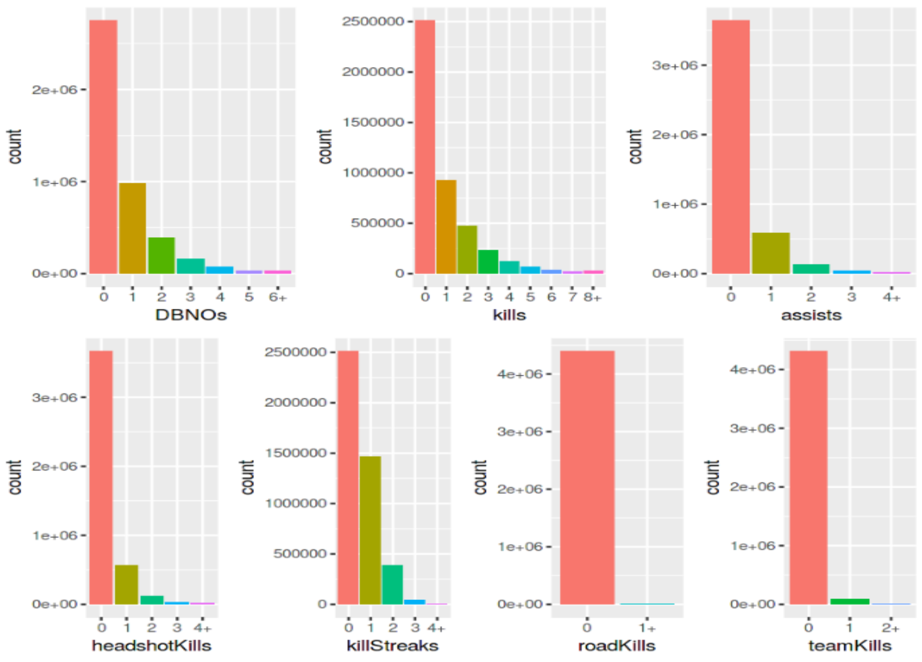


Figure 2 Players killing statistics

2.2 Outlier Dectection

As we plot the variable distributions, the average level of the players in each feature are as the blue bar. And we find some particular players who could kill people without moving (1535), with 100% headshot rate (20), killed someone 1000 miles away (23), travels more than 10000 miles by foot, acquired more than 80 weapons (19) and so on.Hence, we remove these extreme records with a threshold of no more than 99.5% of the whole dataset, ensuring the least damage to the original data.

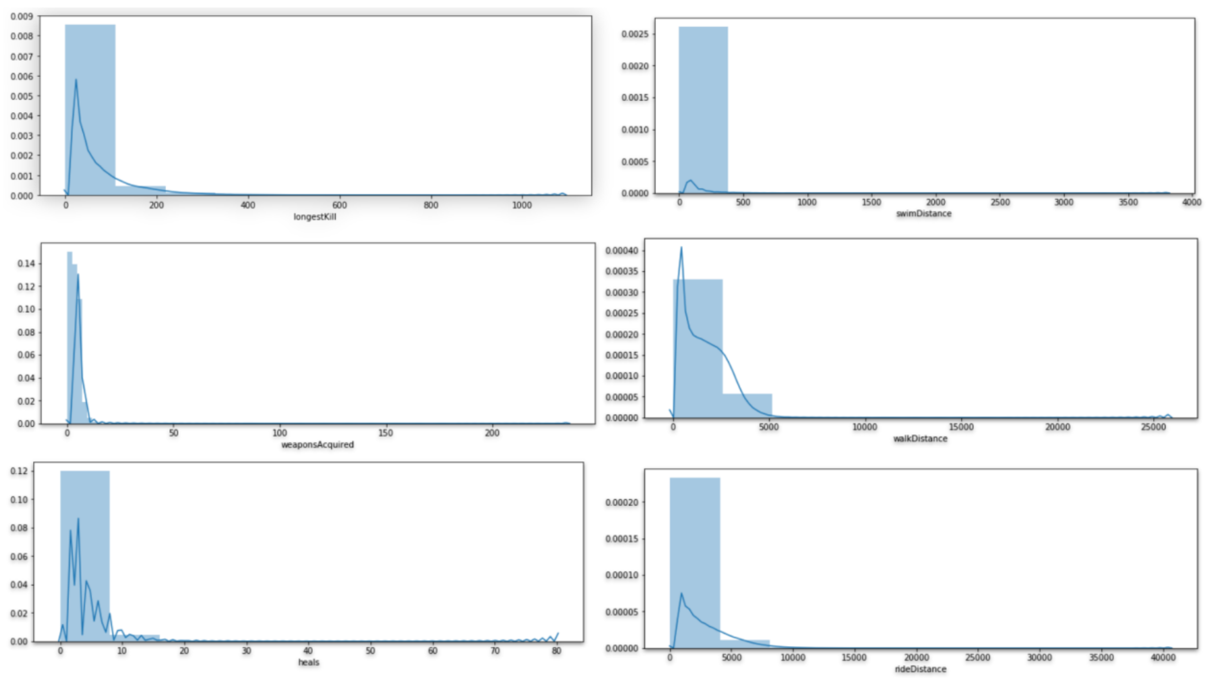


Figure 3 Variable distributions: detect the outliers

3. Feature engineering

3.1 Correlation bettween variables

The correlation matrix indicates that there are a few variables highly correlated with the response, of which the highest positive correlation is the 'walkDistance' and the highest negative is the killPlace.

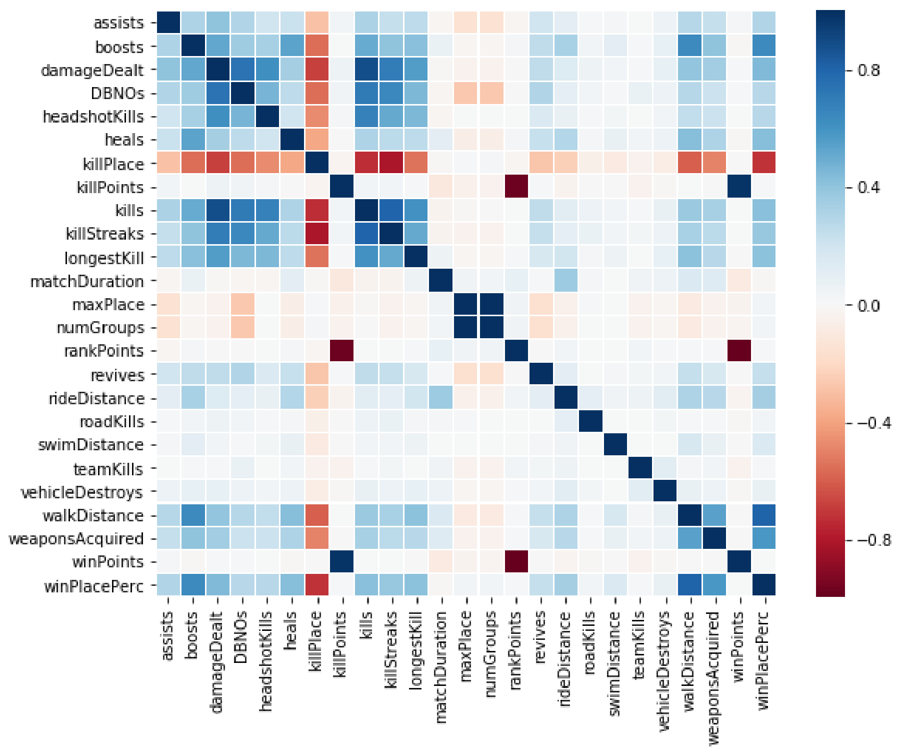


Figure 4 Correlation Matrix

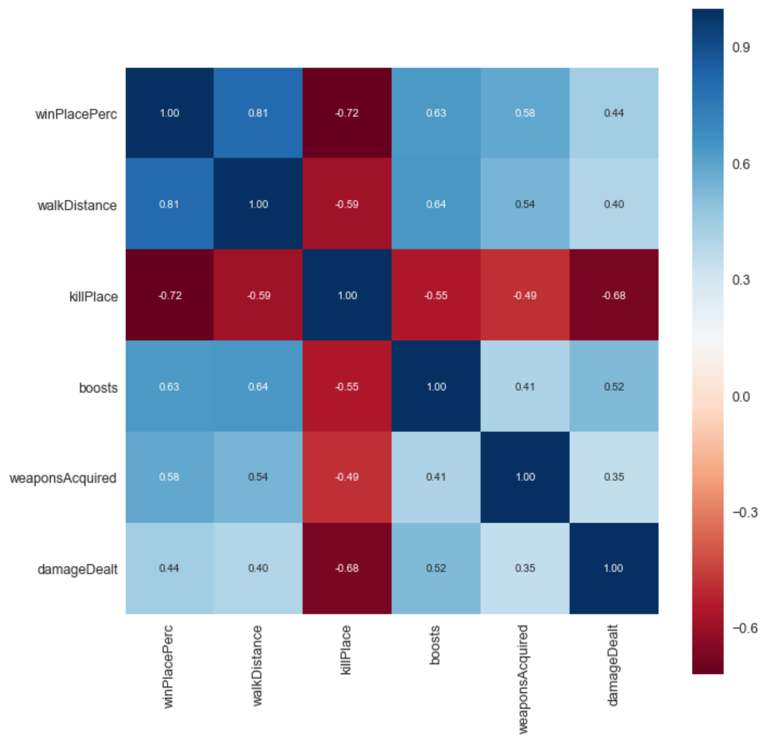


Figure 5 Top-5 most correlated-with-the-response variables

It is possible that the highly correlated variables (walkDistance, killPlace, boosts, weaponAcquired, damageDealt) might be the most important features in predicting winPlacePerc. This corresponds with our intuition, because weapons, supplies, and movements all are very import element on the battlefield.

3.2 Feature Transformation

3.2.1 Add New Features

A game in PUBG can have up to 100 players fighting each other. But in most of the circumstances, a game isn't "full". There is no variable that gives us the number of players joined. So let’s create one and use it to normalize other features.

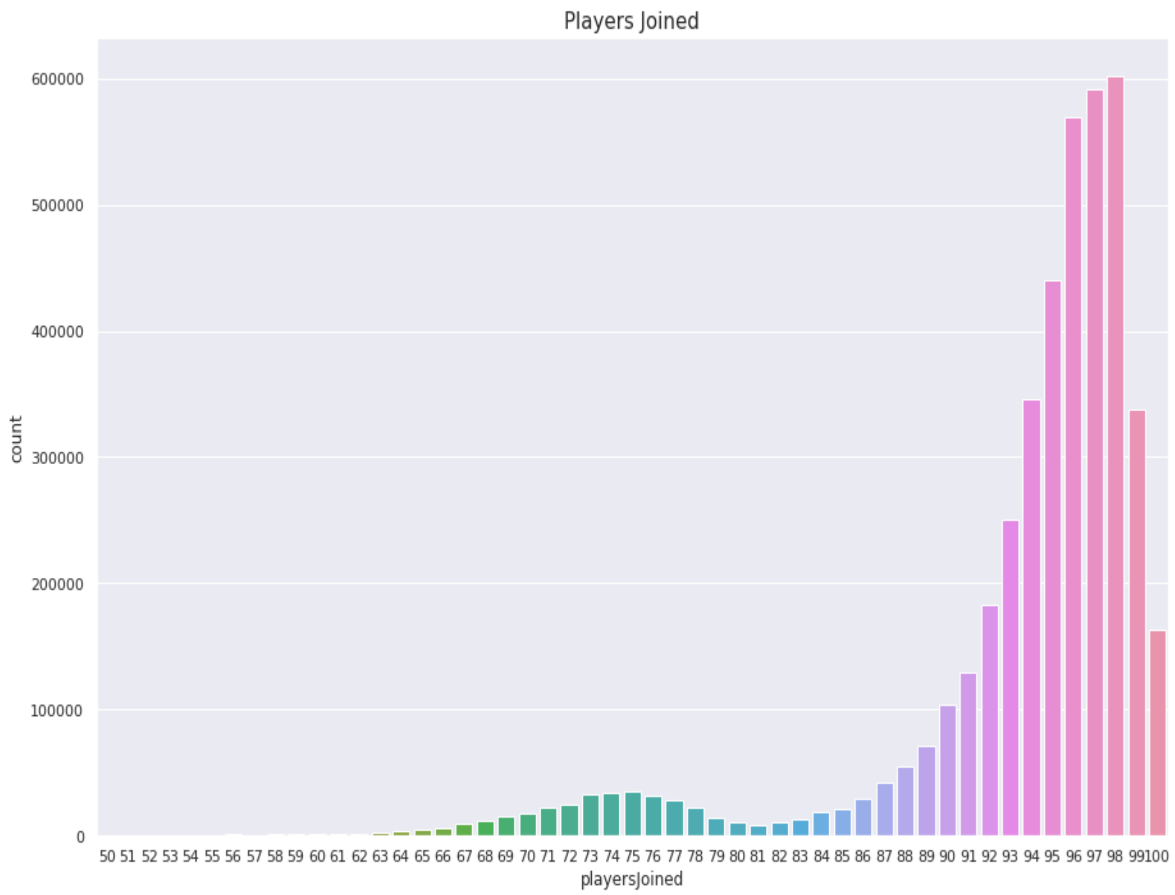


Figure 6 New feature: Players joined

Based on the "playersJoined" distribution, we can create other new features to normalize their values. For example we can create the "killsNorm" and "damageDealtNorm" features. When there are 100 players in the game it might be easier to find and kill someone, than when there are 90 players.

- killsNorm = kills / playersJoined
- damageDealtNorm = damageDealt / playersJoined

Because heals and boost are simlar, so do the swimDistance , walkDistance and rideDistance, thus we add they together

- items= heals + boosts
- total_distance = swimDistance + walkDistance + rideDistance

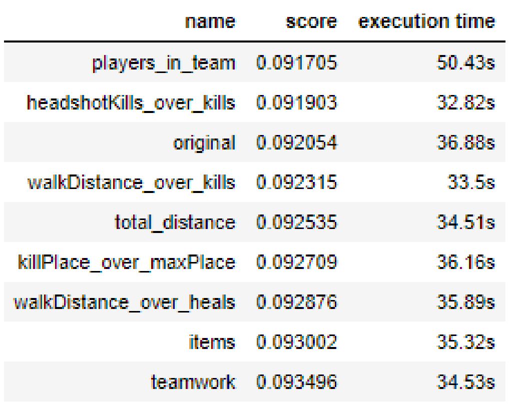
We can also use walkDistance to nomalize other features

- walkDistance_over_kills = walkDistance / kills
- walkDistance_over_heals = walkDistance / heals
- walkDistance_over_boosts = walkDistance / boosts

The other features we created

- teamwork = assist + revive
- headshotKills_over_kills
- players_in_team

We add those features to dataframe and run the most simple linear regression (Because the more complicated the model is the more time it takes, in order to make the feature engineering faster, we always try linear regression first if the result is good then we try some more complicated model)



name	score	execution time
players_in_team	0.091705	50.43s
headshotKills_over_kills	0.091903	32.82s
original	0.092054	36.88s
walkDistance_over_kills	0.092315	33.5s
total_distance	0.092535	34.51s
killPlace_over_maxPlace	0.092709	36.16s
walkDistance_over_heals	0.092876	35.89s
items	0.093002	35.32s
teamwork	0.093496	34.53s

Table 4 New feature performance

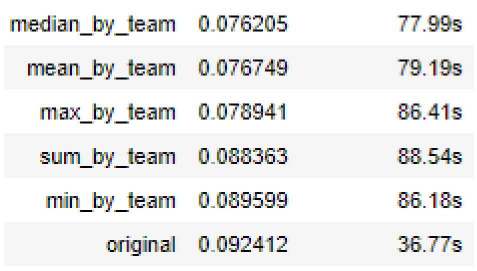
Those features did not make our MAE lower, so we think further.

3.2.2 Group features by teams

Because the game is played by teams (usually there is 2 or 4 players in a team according the the match type). The winPlacePerc is decided by the performance of the whole team, so we try different way to group feature by teams

- min_by_team
- max_by_team
- sum_by_team
- mean_by_team
- median_by_team

For every feature now we have, we create 5 new feature with 5 different way to represent the performance of the whole team. Then we run experiments.



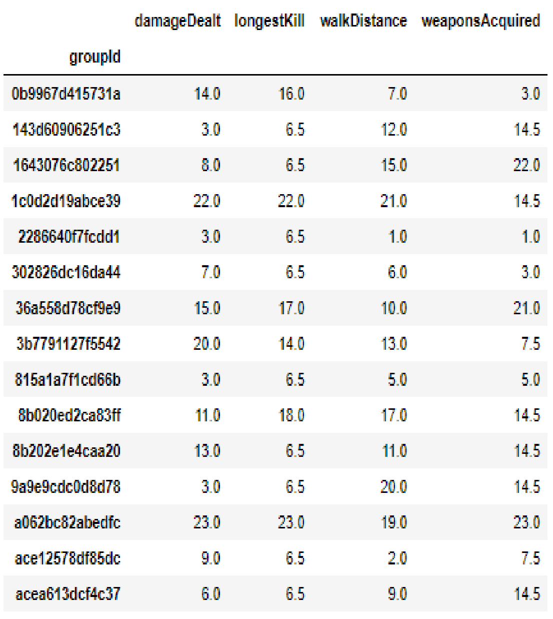
median_by_team	0.076205	77.99s
mean_by_team	0.076749	79.19s
max_by_team	0.078941	86.41s
sum_by_team	0.088363	88.54s
min_by_team	0.089599	86.18s
original	0.092412	36.77s

Table 5 Performances of grouping features by teams

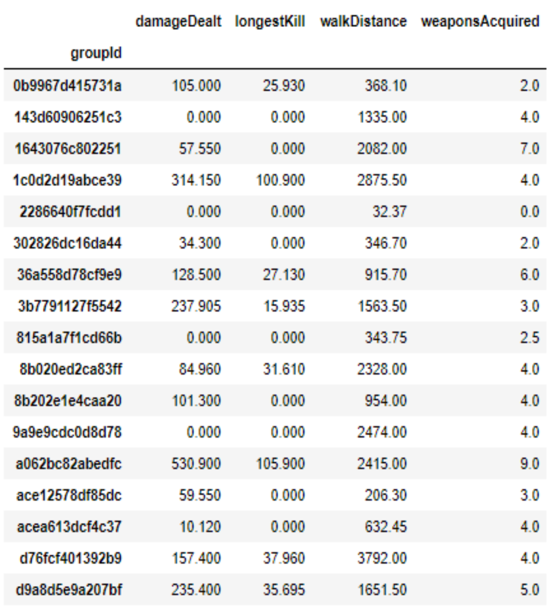
As you can see in the table 5, median_by_team works the best, which makes our MAE go down to 0.076 from 0.092

3.2.3 Ranking in matchs

In reality, the winPlacePerc is decided by the relative performances of a team compared to other teams in a single match but not the absolute performances over all matches. So we tried to transform the original features to ranking features which represent the ranking place of a team in a match. By this method, we not only normalize the data among different matches, but also normalize different features into the same scale.



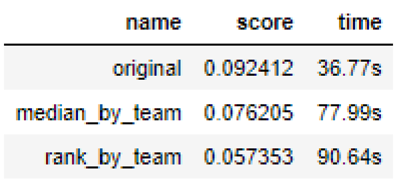
groupId	damageDealt	longestKill	walkDistance	weaponsAcquired
0b9967d415731a	14.0	16.0	7.0	3.0
143d60906251c3	3.0	6.5	12.0	14.5
1643076c802251	8.0	6.5	15.0	22.0
1c0d2d19abce39	22.0	22.0	21.0	14.5
2286640f7fcdd1	3.0	6.5	1.0	1.0
302826dc16da44	7.0	6.5	6.0	3.0
36a558d78cf9e9	15.0	17.0	10.0	21.0
3b7791127f5542	20.0	14.0	13.0	7.5
815a1a7f1cd66b	3.0	6.5	5.0	5.0
8b020ed2ca83ff	11.0	18.0	17.0	14.5
8b202e1e4caa20	13.0	6.5	11.0	14.5
9a9e9cdc0d8d78	3.0	6.5	20.0	14.5
a062bc82abedfc	23.0	23.0	19.0	23.0
ace12578df85dc	9.0	6.5	2.0	7.5
acea613dcf4c37	6.0	6.5	9.0	14.5



groupId	damageDealt	longestKill	walkDistance	weaponsAcquired
0b9967d415731a	105.000	25.930	368.10	2.0
143d60906251c3	0.000	0.000	1335.00	4.0
1643076c802251	57.550	0.000	2082.00	7.0
1c0d2d19abce39	314.150	100.900	2875.50	4.0
2286640f7fcdd1	0.000	0.000	32.37	0.0
302826dc16da44	34.300	0.000	346.70	2.0
36a558d78cf9e9	128.500	27.130	915.70	6.0
3b7791127f5542	237.905	15.935	1563.50	3.0
815a1a7f1cd66b	0.000	0.000	343.75	2.5
8b020ed2ca83ff	84.960	31.610	2328.00	4.0
8b202e1e4caa20	101.300	0.000	954.00	4.0
9a9e9cdc0d8d78	0.000	0.000	2474.00	4.0
a062bc82abedfc	530.900	105.900	2415.00	9.0
ace12578df85dc	59.550	0.000	206.30	3.0
acea613dcf4c37	10.120	0.000	632.45	4.0
d76fcf401392b9	157.400	37.960	3792.00	4.0
d9a8d5e9a207bf	235.400	35.695	1651.50	5.0

Table 6 Data samples before (left) and after(right) ranking transformation

We use ranking features to run a experiment. Fortunately, the MAE goes down to 0.057 from 0.076.



name	score	time
original	0.092412	36.77s
median_by_team	0.076205	77.99s
rank_by_team	0.057353	90.64s

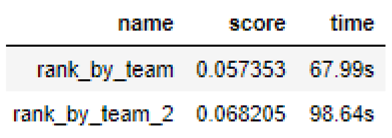
Table 7 Performance of rank-by-team transformation

3.2.4 More Ranking Methods

Considering the ranking method used in the last try destroys the distance, i.e. suppose there are there teams, team A kills 20 enemies, team B kills 10 enemies and team C kills 9 enemies. However the ranking is 1,2 and 3. Apparently the difference between team A and team B is not the same with the difference between team B and Team C, we tries another way to rank.

$$rank\_2 = (x - x.min) / (x.max - x.min)$$

Here we run a cross validation over the new ranking method and find that the performance get even worse, the MAE goes up to 0.068



name	score	time
rank_by_team	0.057353	67.99s
rank_by_team_2	0.068205	98.64s

Table 8 Performance of another rank-by-team transformation method

After we look through the rank, we found out this method actually destroy information even further, because as you can from picture below, the rankings are all squeezed into a small interval(0-0.2) due to the maximum value is to large.

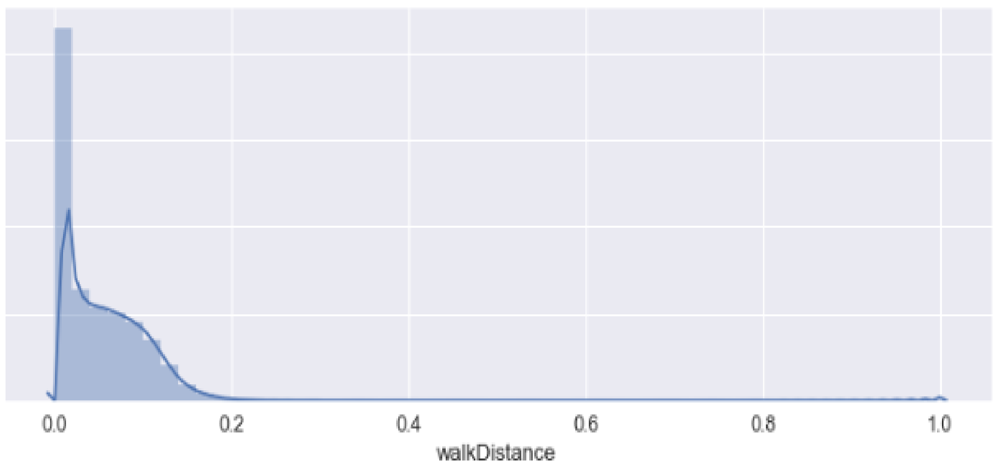


Figure 7 walkDistance distribution (after the rank transformation)

So we came up with another way of ranking, using log() to solve the problem, for the reason that the log function is able to narrow the distances among the dispersed large values by logarithmically scaling the data.

$$rank\_3 = log\ (x+1)\ /\ log\ (x.max)$$

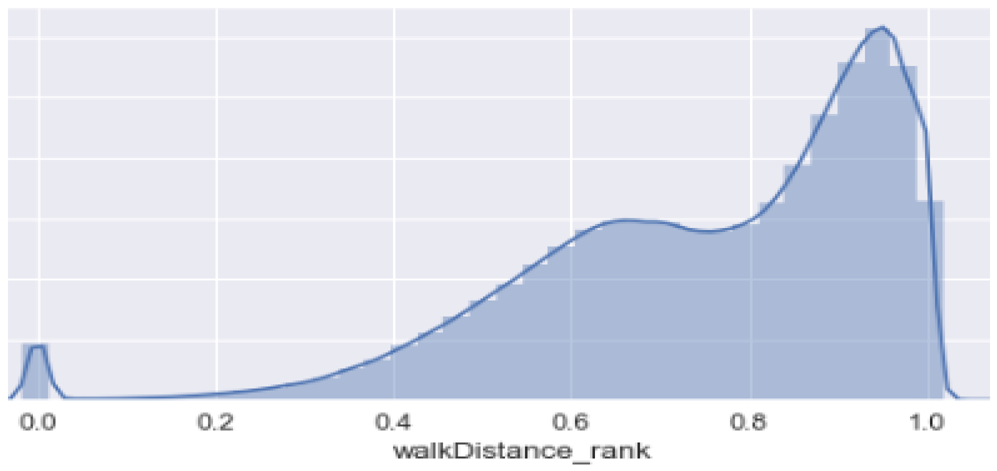
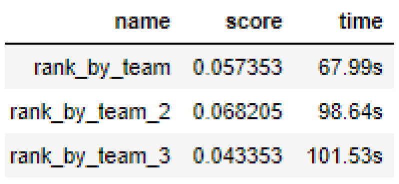


Figure 8 walkDistance distribution (after the logarithmicalized rank transformation)



name	score	time
rank_by_team	0.057353	67.99s
rank_by_team_2	0.068205	98.64s
rank_by_team_3	0.043353	101.53s

Table 9 The performance of the logarithmicalized rank transformation

This time the ranking features imporve our result a lot. The MAE goes down to the 0.04335 from 0.057353. Hence, we add rank_3 features to data frame to fit other more complicated model, to further improve our predict performance.

4.Models

4.1 Baseline Models

4.1.1 Linear Regression

Least Squares Linear regression model fits a linear hyperplane to reduce the Residual Sum of Squares between predicted and actual value. Preprocessing includes one-hot encoding the categorical attribute (*matchType*) and removing the ID attributes (*Id,matchId,groupId*). We used all the features to fit the model. 5-Fold cross validation is used to report errors.

MAE	RMSE	R^2

0.0899	0.1231	0.8397

Table 10 Linear regression performance

As we can see, the linear regression model performs slightly better than the Zero-R model.

4.1.2 Linear Regression on Logit transformation

Our target attribute is a probability and is constrained in the range [0,1]. But, performing linear regression provided values above 1 and below 0. Previously we replaced the output values above 1 as 1 and below 0 as 0.

Next, we used a logit transformation on the output and predicted this value. We then performed an inverse logit on the predicted output to get the probability value in the range [0,1].

Logit transformation of actual y:

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X.$$

Inverse Logit transformation of predicted y:

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}}.$$

MAE	RMSE	R^2
0.0877	0.121	0.8421

Table 11 Logit transformation linear regression performance

As we can see, the linear regression of logit transformation performs slightly better than the Linear Regression model.

4.1.3 Experiments

A. Linear Regression on Logit Transformation

Previously we established a baseline model using Linear Regression on Logic transformation. This section covers experimenting Linear Regression with PCA, Feature Extraction, Forward and Backward Feature Selection.

Pre-processing	MAE	RMSE	R^2
PCA - 95% variance 29-variable model	0.1154	0.1516	0.7568
Lasso - alpha = 0.01 (15 variable picked)	0.0954	0.134	0.8002
Forward Feature Selection	0.0877	0.121	0.8421
Backward Feature Selection	0.0862	0.119	0.8496
Feature Extraction	0.0577	0.0791	0.9337

Table 12 Linear Regression with PCA, Feature Extraction, Forward and Backward Feature Selection

We find that

- PCA with Linear Regression did not provide a better result.
- Lasso picked out 15 varibales
- Both Forward and Backward selection is done with MAE as the metric for feature addition or elimination. Forward feature selection provided a model with all variables and backward subset selection provided a model with all variables except *longestKill*
- Feature Extraction by aggregating features of players belonging to same group provided a much better result. (Rank, Median, Min, Max)

The top 5 important coefficients are

- killPlace_rank: -2.092420443897782
- kills_rank: -1.2711517397371752

walkDistance_rank:	1.19594343199714
roadKills_rank:	0.5414468629295042
boosts_rank:	0.18354260740890907
longestKill_rank:	0.175932170489564

4.2 Beta Regression

Beta Regression is a model that belongs to the class of Generalized Linear Models. In least squares Linear Regression we assume that the distribution of the output follows a normal distribution. If we assume that the output follows a different distribution such as poisson or binomial distribution, we get different Mean and Variance and thus Linear Model is subjected to change. In our case, our output is close to a Beta distribution. Formally, beta distribution is a family of continuous probability distributions defined on the interval [0, 1] parametrized by two positive shape parameters, denoted by α and β. Beta regression assumes a range of (0,1) for the output, but our actual output assumes the extremes 0 and 1. So, an useful transformation in practice is **y = (y . (n-1) + 0.5) / n.** Although this model has significance over linear regression, this model performed similar to linear regression. We believe that the large dataset and in turn a good representation of the population compensated for the error introduced by linear regression.

Mean of Beta distribution

$$\mathrm{E}[X] = \frac{\alpha}{\alpha + \beta}$$

Variance of Beta distribution

$$\mathrm{var}[X] = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$$

MAE	RMSE	R^2
0.0865	0.120	0.8443

Table 13 Beta Regression performance

4.4 Impact of PCA

From the pair-plots it can be seen that there is no linear relationship between predictors. One of the assumptions of PCA is that there is correlation between input predictors. It works best when this assumption holds. Therefore using PCA in our case with all non-linear relations does not improve the model.

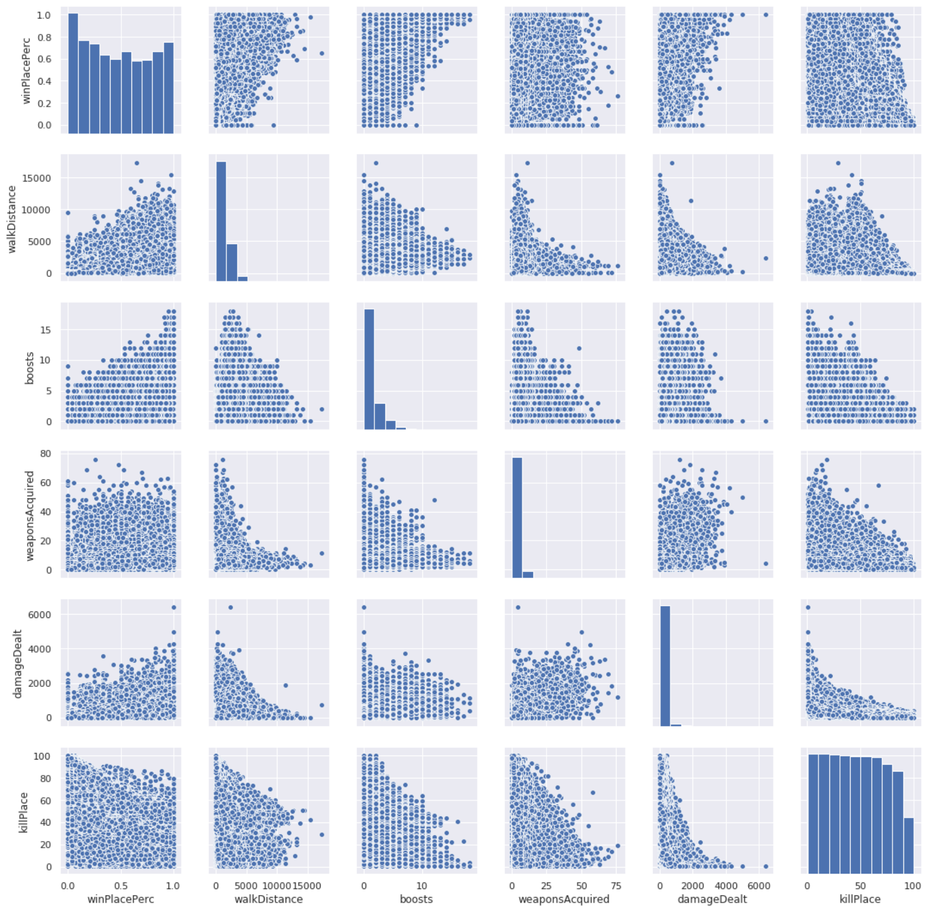


Figure 9 Pair-plots

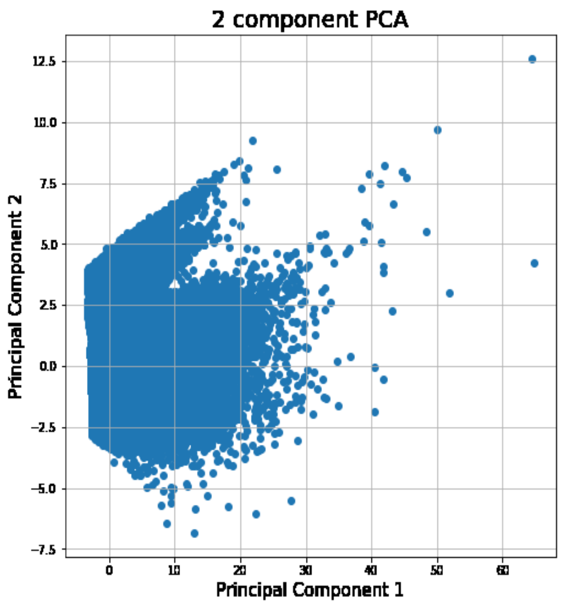


Figure 10 2 component PCA

Also, the plot between first two PCA components do not show much linearity which suggests that fitting a linear regression model might not give a better accuracy.

4.5 Random Forest

Random Forest builds a large number of weak decision trees by bootstrap aggregating the data points for each tree and also randomly selecting a subset of predictors for each tree. It then combines the output of the trees to form a strong learner. A feature is selected to split if the split point gives the lowest MAE. We perform a series of experiments with Feature Engineering and parameter tuning and report the best model from Random Forest Regressor. The execution time of random forest for a simple Random Forest model was around an hour, since the dataset is very large. This restricted us from getting better results in hyperparameter optimization and performing cross-validation.

Model Parameters	Pre-processing	MAE	RMSE
n_estimators: 100 min_samples_leaf: 30	NA	0.0509	0.0704

max_features: sqrt criterion: mae			
n_estimators: 40 min_samples_leaf: 10 max_features: sqrt criterion: mae	Hyperparameter optimization via Grid Search on MAE on n_estimators and min_samples_leaf	0.0436	0.0634
n_estimators: 40 min_samples_leaf: 10 max_features: sqrt criterion: mae	Feature Extraction - Ranking, Median, #Players in Match, #Players in the group	0.0378	0.0581

Table 14 Random forest performance

Feature Importances of only original features

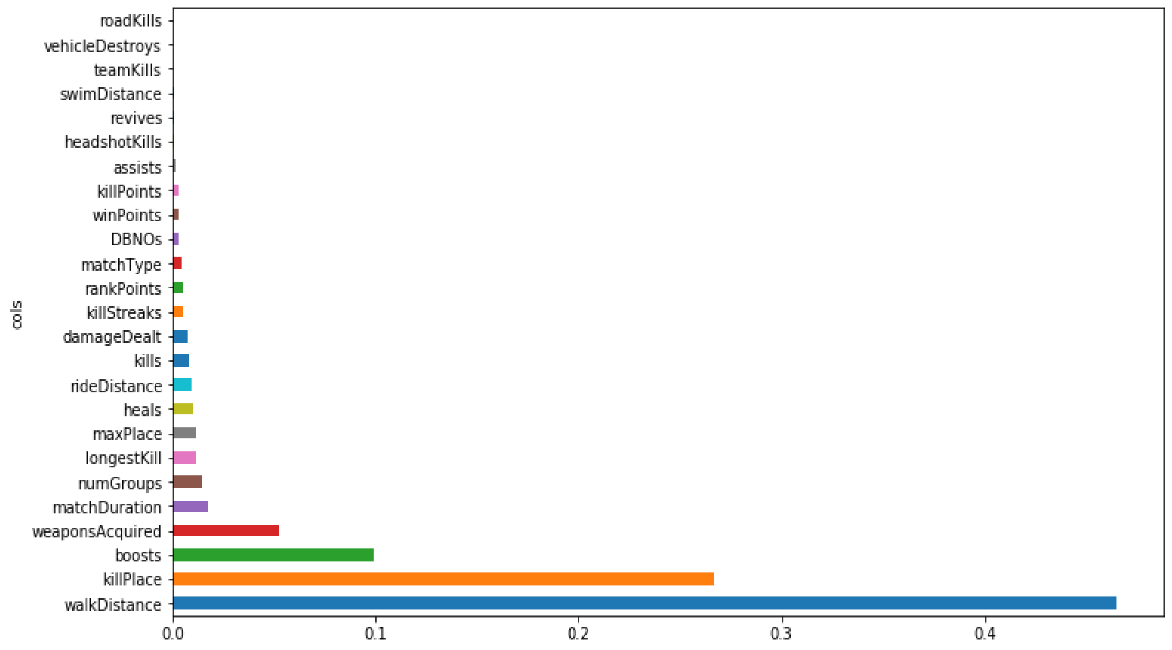


Figure 11 Original Feature Importances

4.6 Neural Network

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function by training on a dataset, where is the number of dimensions for input and is the number of dimensions for output. Given a set of features and a target , it can learn a nonlinear function approximator for either classification or regression.

It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. The figure below shows a one hidden layer MLP with scalar output.

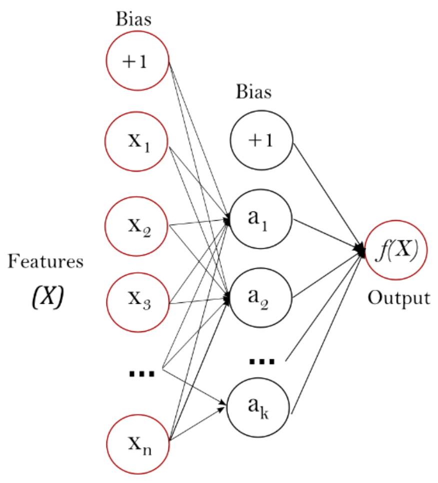


Figure 12 one hidden layer MLP with scalar output

4.6.1 Advantages and disadvantages

The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models.
- Capability to learn models in real-time (on-line learning) using partial_fit.

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- MLP is sensitive to feature scaling.

4.6.2 Fitting process

Firstly, we scale the data and fit the model with default parameters. Then we tune the parameters and conduct cross validation using the grid search method. After we get the result, we compare the results and choose the best model to predict the test dataset getting the final value of MAE.

4.6.3 Parameter Tuning

We adjust the solver, alpha and layer in the Neural Network model and run 5-fold cross validation. The results in the table indicates that the model with solver 'Adam', alpha = 0.00001 and layer = (150, 6) has the best performance of 0.041261.

Solver	Alpha	Layer	MAE
adam	0.00001	(5, 2)	0.048931
adam	0.00001	(5, 3)	0.047044
adam	0.00001	(50, 2)	0.042776
adam	0.00001	(50, 3)	0.042885
adam	0.00001	(100, 3)	0.041668
adam	0.00001	(100, 7)	0.041397
adam	0.00001	(100, 8)	0.041554
adam	0.00001	(100, 9)	0.041414
adam	0.00001	(150, 6)	**0.041261**
sgd	0.00001	(150, 6)	0.052242
lbfgs	0.00001	(150, 6)	0.050487
adam	0.0001	(150, 6)	0.042612
adam	0.001	(150, 6)	0.044283
adam	0.000001	(150, 6)	0.041369
adam	0.00001	(150, 7)	0.041869
adam	0.00001	(150, 8)	0.042204
adam	0.00001	(200, 4)	0.042515
adam	0.00001	(200, 5)	0.042192
adam	0.00001	(200, 6)	0.041414

Table 15 Neural Network Parameter tuning

However, the MAE of 0.041 is not satisfying. The reason why neural network does not good enough might be as following:

- There are still some other parameters that we did not tune;
- The dataset is not big enough for the neural network to learning the proper feature of the real problem.

4.7 XGBoost

4.7.1 Why do we use XGBoost?

It has been proved that in practice Tree Boosting can be effectively used for predictive mining of classification and regression tasks. The XGBoost has re-implemented tree enhancements and has been well-received in Kaggle and other data science competitions, therefore, it is extremely popular now.

4.7.2 Basic component of XGBoost

- Boosting

Boosting is a learning algorithm that uses multiple simpler models to fit data. The simpler models used are also known as base learners or weak learners. The way to learn is to adaptively fit the data using a basic learner with the same or slightly different parameter settings.

- CART

The tree model is a simple and interpretable model. Their predictive power is really limited, but by combining multiple tree models (such as bagged trees, random forests, or in boosting), they can become a powerful predictive model. For XGBoost, it implements a tree model related to CART. CART grows trees in a top-down manner. By considering each split parallel to the coordinate axis, CART can choose to minimize the segmentation of the target. In the second step, CART considers each parallel segmentation in each region. At the end of this iteration, the best segmentation will be selected. CART repeats all of these steps until the stop criterion is reached.

- Xgboost

The objective function of XGBoost:

$$Obj(t) = \sum_{i=1}^{n} L(y_i, \hat{y}^{t-1} + f_t(x_i)) + \Omega(f_t) + constant$$

where the regularization is:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{i=1}^{T} {w_j}^2$$

T is the number of leaf nodes and w_j is the weight of the jth leaf node. And through a series of formulas, the final objective function looks like this:

$$Obj(t) = -\frac{1}{2}\sum_{j=1}^{T} \frac{{G_j}^2}{H_j + \lambda} + \gamma T + C$$

After the objective function is simplified, you can see that the target function of xgboost is customizable, and only the first and second derivatives of it are used in the calculation. After obtaining the simplified formula, the next step is to calculate the gain brought by the selected features, so as to select the appropriate splitting feature.

$$gain(\phi) = gain(before) - gain(after)$$

$$= \frac{1}{2}[\frac{{G_L}^2}{H_L + \lambda} + \frac{{G_R}^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}] - \gamma$$

4.7.3 First try with XGBoost

We split the training data into 80% to train and 20% percent to test. And then we set the parameters for the XGBRegreesor.

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0.3, learning_rate=0.1, max_delta_step=0,
       max_depth=18, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=10, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

The result is that we get 0.055 MAE on the validation dataset.

```
from sklearn.metrics import mean_absolute_error
y_pred=model.predict(X_test)
print('mean absolute error is:',mean_absolute_error(y_test,y_pred))
```

```
mean absolute error is: 0.05556577361869186
```

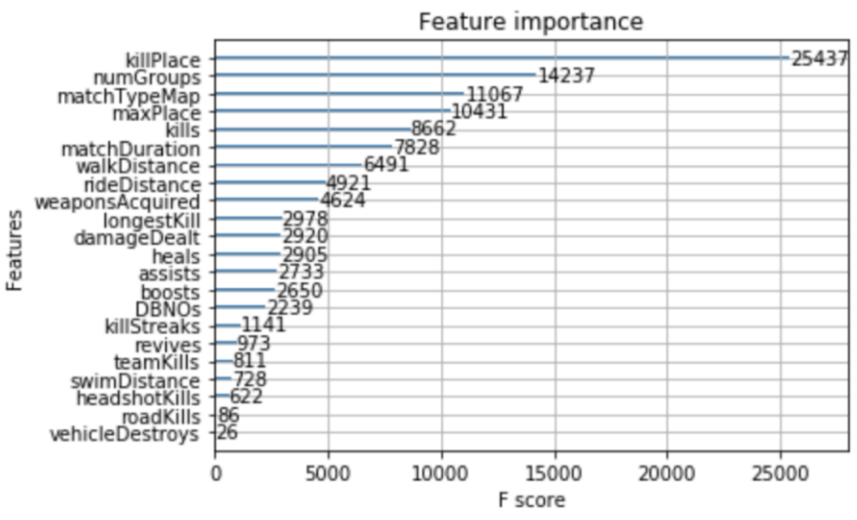
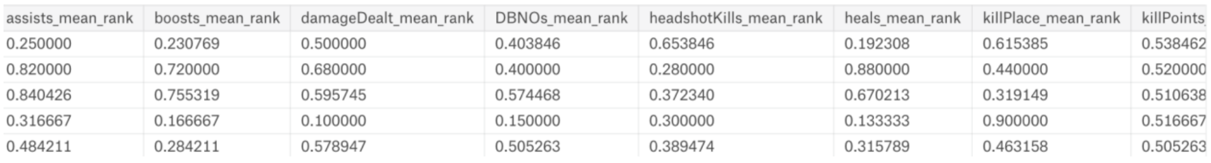


Figure 13 XGB feature importance 1

4.7.4 Second try with XGBoost

The result of our first try is not as good as we expected. So in the second try, we add new feature into the training and testing dataset.



assists_mean_rank	boosts_mean_rank	damageDealt_mean_rank	DBNOs_mean_rank	headshotKills_mean_rank	heals_mean_rank	killPlace_mean_rank	killPoints
0.250000	0.230769	0.500000	0.403846	0.653846	0.192308	0.615385	0.538462
0.820000	0.720000	0.680000	0.400000	0.280000	0.880000	0.440000	0.520000
0.840426	0.755319	0.595745	0.574468	0.372340	0.670213	0.319149	0.510638
0.316667	0.166667	0.100000	0.150000	0.300000	0.133333	0.900000	0.516667
0.484211	0.284211	0.578947	0.505263	0.389474	0.315789	0.463158	0.505263

We fit the XGBoost model having 50 features in total, and get a new MAE:

```
from sklearn.metrics import mean_absolute_error
y_pred=model.predict(X_test)
print('mean absolute error is:',mean_absolute_error(y_test,y_pred))
```

```
mean absolute error is: 0.03962486781381307
```

Again, we plot the important features to see whether the new feature has any effect on the model.

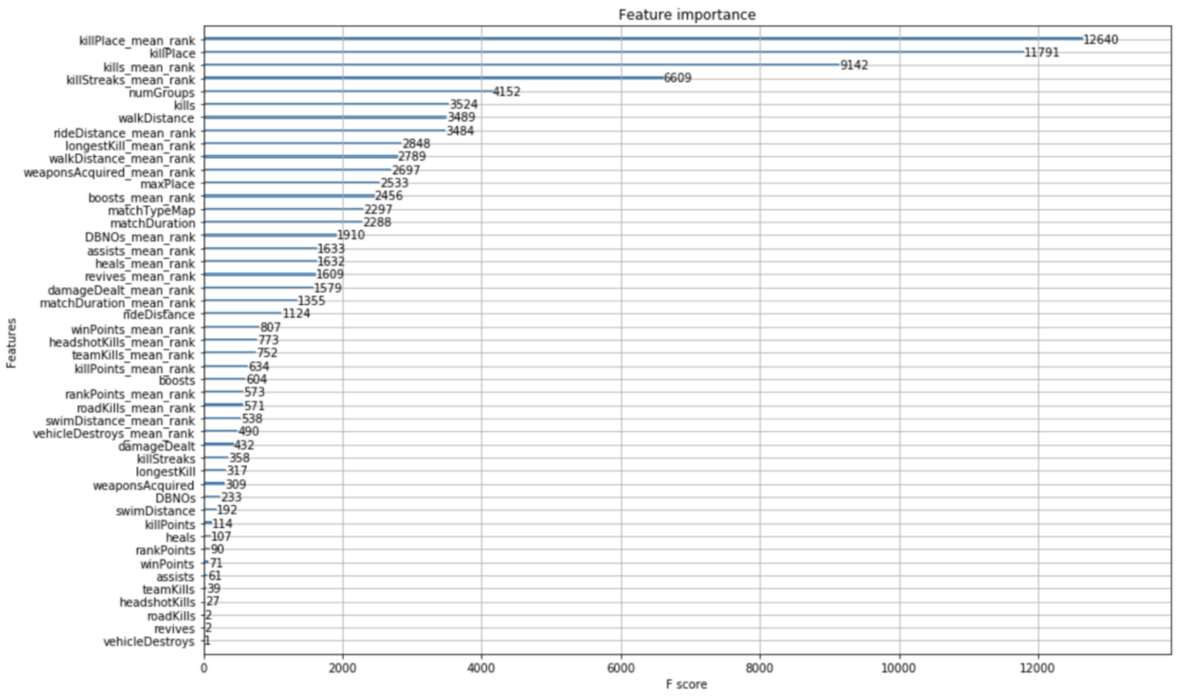


Figure 14 XGB feature importance 2

Apparently, we can conclude that most of new features play an important role in the current, the reduce of MAE is due to the introduce of mean ranks of their original features.

4.7.5 Parameter tuning

1.n_estimators: Number of boosted trees to fit. This is the number of iterations we will set for the XGBoost, when n_estimators is too large, the boosted tree will probably overfitting, so we set the parameter the default value 100.

2. max_depth: Maximum tree depth for base learners. this value is used to avoid overfitting. We cross validate the value and we set the final max_depth as 5.

3. Min_child_weight: Determines the minimum leaf node sample weight sum. When its value is large, the model can avoid to learn local special features. But if this value is too high, it will lead to under-fitting. This parameter needs to be adjusted with cv, and finally we found that the default value which is 1 works better.

4. Gamma: During the node splitting, the node splits only if the value of the loss function drops after splitting. Gamma specifies the minimum loss function drop value required for node splitting. The larger the value of this parameter, the more conservative the algorithm. The best Gamma for this model is 0.1.

4.8 Light Gradient Boosting Model

LightGBM is an alternative to XGBoost model implemented by Microsoft in 2017. Since then the model has gained much appreciation and Kaggle toppers use LightGBM. Motivation behind using LightGBM is that its implementation is much faster than XGBoost. Several parallel and distributed architectures are employed. Major structural difference between XGBoost and LightGBM is that, XGBoost tree grows level-wise (breadth-wise) and LightGBM tree grows leaf-wise (depth-wise). A leaf that minimizes the loss function is expanded. Previously, to split a tree all split points are considered. LightGBM introduced histogram based splitting, thus only ***#bins*** split points are considered instead ***#splitpoints.*** Another improvement in LightGBM is Gradient One-sided sampling. The key idea is that while considering split points, all the points with larger gradients are considered. But, only a sample of data points with smaller gradients are considered. The reason is that data points with smaller gradients are already well trained. Further, Exclusive feature bundling is used to bundle two or more features to avoid the computational cost of performing operations on sparse data features. It also handles categorical variables automatically.

Model Parameters	Pre-processing	MAE	RMSE
n_estimators: 100 min_data_in_leaf: 20 feature_fraction: 0.8 learning_rate: 0.5 metric: mae	NA	0.0495	0.0696
n_estimators: 500 min_data_in_leaf: 20 feature_fraction: 0.7 learning_rate: 0.1 metric: mae	NA	0.0483	0.0688

n_estimators: 500 min_data_in_leaf: 20 feature_fraction: 0.8 learning_rate: 0.1 metric: mae	Feature Extraction - Ranking, Median	0.0382	0.0554
n_estimators: 100 min_data_in_leaf: 20 feature_fraction: 0.8 learning_rate: 0.01 metric: mae	Feature Extraction - Ranking, Median	0.0336	0.0512
n_estimators: 1000 min_data_in_leaf: 20 feature_fraction: 0.8 learning_rate: 0.05 metric: mae	Feature Extraction - Ranking, Median, #Players in Match, #Players in the group	0.0263	0.0394

Table 16 LGBM Performance

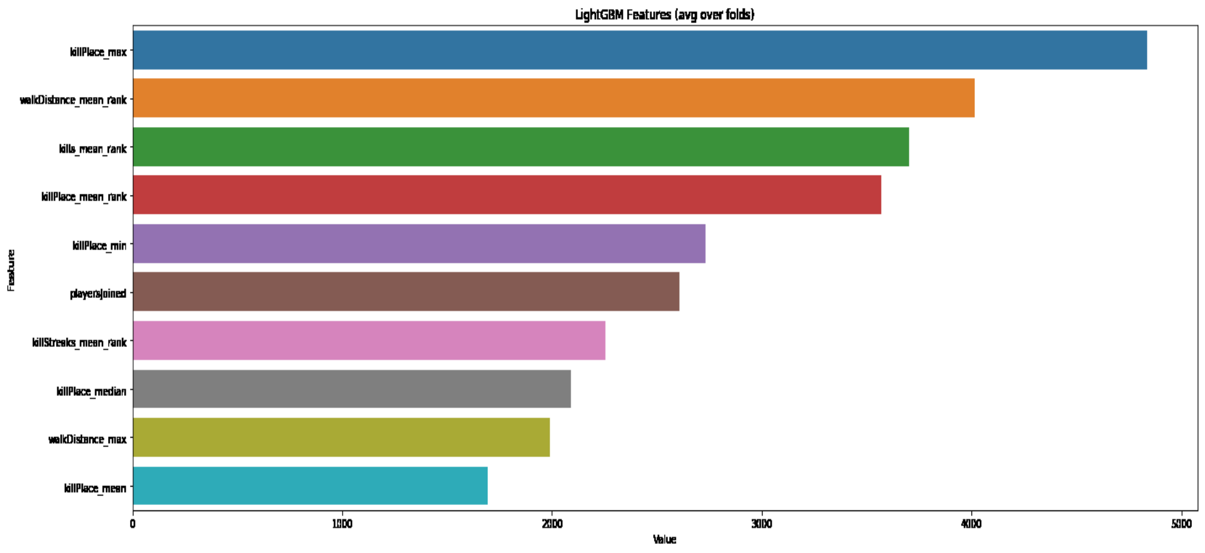


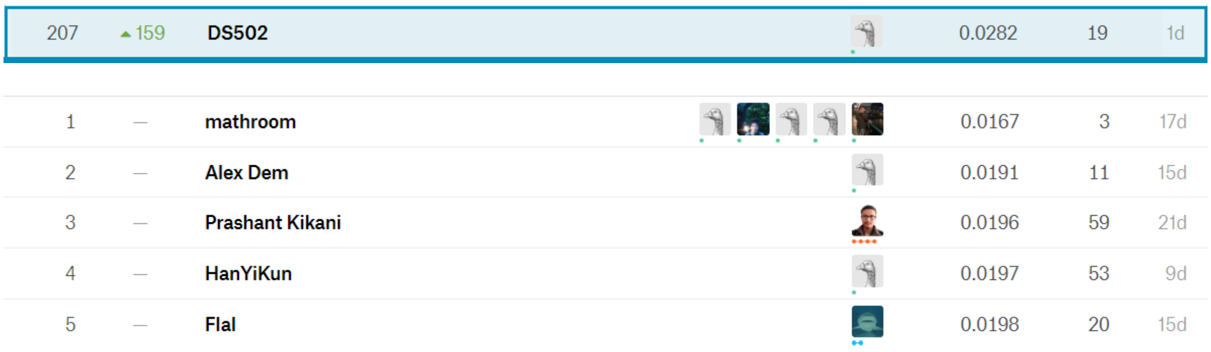
Figure 15 Top 10 important features from LightGBM:

5. Conclusion

After preprocessing the data by getting rid of outliers, We conduct feature engineering and achieved some visible improvements on the results of our models. 8 models are fitted in total, including basic models like zero-R, linear regression and beta regression, and advanced models, such as PCA, Random Forest, Neural Network, XGBoost and LightGBM.

5.1 Final Achievement on Kaggle

Among all the models we have tried, LGBM performs the best, with an outstanding MAE of 0.0263. After we submit the model on Kaggle, we get a final test MAE of 0.0282, which is the top 28% in the leaderboard. For the top 5 ranks, their MAE starts from 0.0167 to 0.0198.



207	▲159	DS502	0.0282	19	1d
1	—	mathroom	0.0167	3	17d
2	—	Alex Dem	0.0191	11	15d
3	—	Prashant Kikani	0.0196	59	21d
4	—	HanYiKun	0.0197	53	9d
5	—	Flal	0.0198	20	15d

5.2 Learnings and Drawbacks

5.2.1 Findings and Learnings

We find that as long as you are a active player in the PUBG, which means that you do practice killing and shooting skills, keep move in the game and use boosting materials often, you will have a great chance winning the game. PUBG is not a difficult game to play.

The trials of models and data transformation is excellent. We conduct nice feature engineering and have significant improvements on our results.

5.2.2 Drawbacks and Limits

The limited time restricts us to experiment some more models which could be useful, such as SVM Regression. And tuning parameters consume too much time, CPU and memory.

The data requires more hit-the-point feature engineering. We firstly run the models on the raw data, which turns out to be a unsuccessful trial.

Also, the models we have tried has pushed us to the advanced part of the leaderboard, but not the top places. The distances between the top of the leaderboard and us is small but difficult to conquer. There is still some part of the problem that we did not figure out.