Analysis upon video game Battle Royale–Prediction of Performance Ranking

Boming Fu, Chao Ni, Laihe Zhang, Huayu Li

January 20, 2019

Abstract

PUBG is a popular online teamwork game. It is of interest how we can rank players in a game based on kills, distances, and other performance indices. Kaggle launched a competition about predicting the ranking of all players given the training data. In this project, we gave three different ranking methods to predict the ranking respectively: linear regression method, tree method and learning to rank method. We used mean absolute error to measure the model performance and utilized ensemble to improve the final result. Our ensemble score achieved top 4% ranking among over 1300 Kaggle teams.

1 Introduction

1.1 The introduction of Battle Royale

Since 2017, the video game–PlayerUnknown's BattleGrounds(PUBG), has taken the world by storm, and has attracted millions of people's attention. In this game, players can choose to enter the match solo, duo, or with a small team of up to four people. The theme of this game is to survive–each match starts with about 100 players, and only one person or one team will survive, who is regarded as the final winner. During the whole match, the players are dropped from a flying plane onto an island empty-handed, and they have to quickly determine the best time to eject and parachuate to the ground. Once they land, the players can explore different buildings to find equipments like weapons, vehicles, armors, to protect themselves. In order to survive, the players should not only collect as many equipments as they can, but also kill and eliminate other players to reduce the existing danger. Every few minutes, the playable area of the map begins to shrink down towards a random location, with any player caught outside the safe area taking damage incrementally, and eventually being eliminated if the safe zone is not entered in time; in game, the players see the boundary as a shimmering blue wall that contracts over time. This results in a more confined area; not only will the process of the match increase a lot, but also will the match become more and more exciting. After the fierce battle, only one people or one team will survive, and the one survive is the final winner.

At the completion of each match, players gain in-game currency based on their performance, which is used to purchase crates which contain cosmetic items for character or weapon customization. And by the way, the system will show each player his ranking in this match. Although there is only one winner in the game, the other players will also be honored due to their performance through the match. Because of the fantastic game setting and great quality, more and more people are attracted by this video game, and PUBG has enjoyed massive popularity. With over 50 million copies sold, it's the fifth best selling game of all time, and has millions of active monthly players.

1.2 Introduction of the kaggle contest

As the introduction has shown, this video game attracts many people to spend time on, and during each match, each player will try really hard to become the last man standing, or gain a high ranking during each match. As a matter of fact, there are many potential parameters that link to the performance in one match, like the number of enemy players killed, the total traveled distance, and so on. So the questions come out: is there a strategy to win the match or to gain good ranking result, or is there a model to predict the rank of the players in one given match?

Based on these rising questions, the contest of dealing with performance data on Kaggle came out. On Kaggle website, over 65,000 games' worth of anonymized player data are given, split into training and testing sets, and asked to predict final placement from final in-game stats and initial player ratings. What aims us to do is to find better and better models to fit the performance data, and give out proper answers to the arising questions. The testing dataset can help to assess the performance of ranking model.

1.3 The purpose of our project

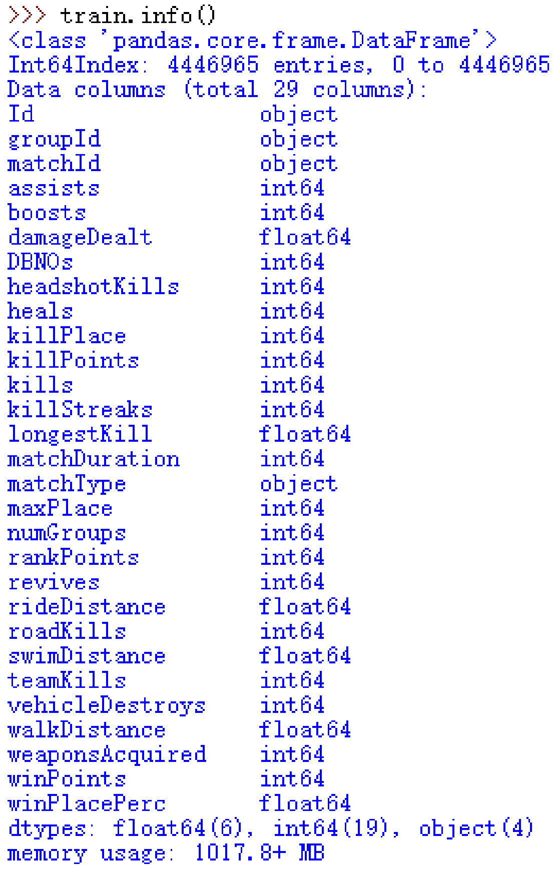
Our purpose is to solve the problems and provide a possible method for players to perform better in the game. In this project, we will mainly focus on the prediction of players raking in one certain round of match based on the given dataset. Here, we emphasize that the prediction is in one certain round, for each round consists of different players, and it is meaningless to judge the players in different match. We will use several different statistical methods to analyze the dataset, to build up the prediction model, and finally, to test to compare these models. In this way, we can make it judging the best way to predict the rank during one certain match. And by the way, this can also reflect the choice of how to gain a good ranking result.

2 Feature Engineering

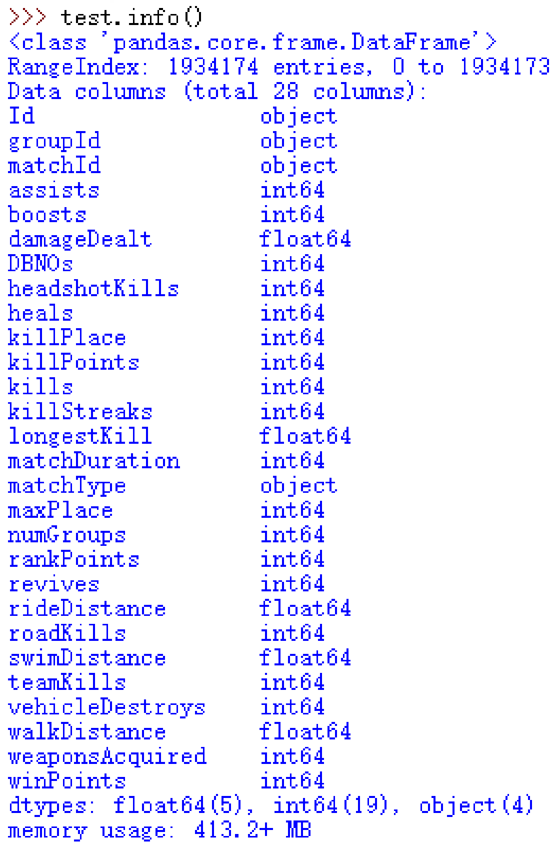
2.1 Data structure and distributions of the dataset

In this part, we will have a description about the data structure and distributions, in order to learn better about the whole datasets.

To begin with, we list out all of the different parameters in order to have a overlook of the whole dataset(the first list is of the training dataset, and the second list is about the testing dataset). From the list given, we can notice that the parameters are divided into two different groups: the ID parameters, and the performance parameters:



```
>>> train.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4446965 entries, 0 to 4446965
Data columns (total 29 columns):
Id                  object
groupId             object
matchId             object
assists             int64
boosts              int64
damageDealt         float64
DBNOs               int64
headshotKills       int64
heals               int64
killPlace           int64
killPoints          int64
kills               int64
killStreaks         int64
longestKill         float64
matchDuration       int64
matchType           object
maxPlace            int64
numGroups           int64
rankPoints          int64
revives             int64
rideDistance        float64
roadKills           int64
swimDistance        float64
teamKills           int64
vehicleDestroys     int64
walkDistance        float64
weaponsAcquired     int64
winPoints           int64
winPlacePerc        float64
dtypes: float64(6), int64(19), object(4)
memory usage: 1017.8+ MB
```



```
>>> test.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1934174 entries, 0 to 1934173
Data columns (total 28 columns):
Id                  object
groupId             object
matchId             object
assists             int64
boosts              int64
damageDealt         float64
DBNOs               int64
headshotKills       int64
heals               int64
killPlace           int64
killPoints          int64
kills               int64
killStreaks         int64
longestKill         float64
matchDuration       int64
matchType           object
maxPlace            int64
numGroups           int64
rankPoints          int64
revives             int64
rideDistance        float64
roadKills           int64
swimDistance        float64
teamKills           int64
vehicleDestroys     int64
walkDistance        float64
weaponsAcquired     int64
winPoints           int64
dtypes: float64(5), int64(19), object(4)
memory usage: 413.2+ MB
```

For the ID parameters, they convey the information of different individuals, different groups, different matches, as well as different gamemodes(solo, duo or so on). In this way, we can make it differentiating different ranking situation under these labels.

For the performance parameters, they convey the achievement of different players or teams in one certain game, and they potentially linked to the final ranking of different players. For example, the parameters include killPoints, longestKill, walkDistance and so on, and using the parameters, the performance of a player will be clear.

What we want to know is the features of the different parameters: the size, the distribution, and so on. And the most important features are the mean, the standard error, the different quantiles and so on. So using the dataset, we give out the important statistical features of different parameters as following:

Table 1: Training dataset

	mean	std	min	25%	50%	75%	max
assists	0.233815	0.588573	0	0	0	0	22
boosts	1.106908	1.715794	0	0	0	2	33
damageDealt	130.7172	170.7806	0	0	84.24	186	6616
DBNOs	0.657876	1.145743	0	0	0	1	53
headshotKills	0.22682	0.602155	0	0	0	0	64
heals	1.370148	2.679982	0	0	0	2	80
killPlace	47.59936	27.46293	1	24	47	71	101
killPoints	505.0062	627.5049	0	0	0	1172	2170
kills	0.924784	1.558445	0	0	0	1	72
killStreaks	0.543955	0.710972	0	0	0	1	20
longestKill	22.9976	50.97262	0	0	0	21.32	1094
matchDuration	1579.507	258.7388	133	1367	1438	1851	2237
maxPlace	44.50468	23.8281	2	28	30	49	100
numGroups	43.0076	23.28949	1	27	30	47	100
rankPoints	892.0103	736.6478	-1	-1	1443	1500	5910
revives	0.164659	0.472167	0	0	0	0	39
rideDistance	606.1158	1498.344	0	0	0	0.191	40710
roadKills	0.003496	0.073373	0	0	0	0	18
swimDistance	4.509323	30.5022	0	0	0	0	3823
teamKills	0.023868	0.167394	0	0	0	0	12
vehicleDestroys	0.007918	0.092612	0	0	0	0	5
walkDistance	1154.218	1183.497	0	155.1	685.6	1976	25780
weaponsAcquired	3.660488	2.456543	0	2	3	5	236
winPoints	606.4603	739.7005	0	0	0	1495	2013
winPlacePerc	0.472822	0.307405	0	0.2	0.4583	0.7407	1

Table 2: Testing dataset

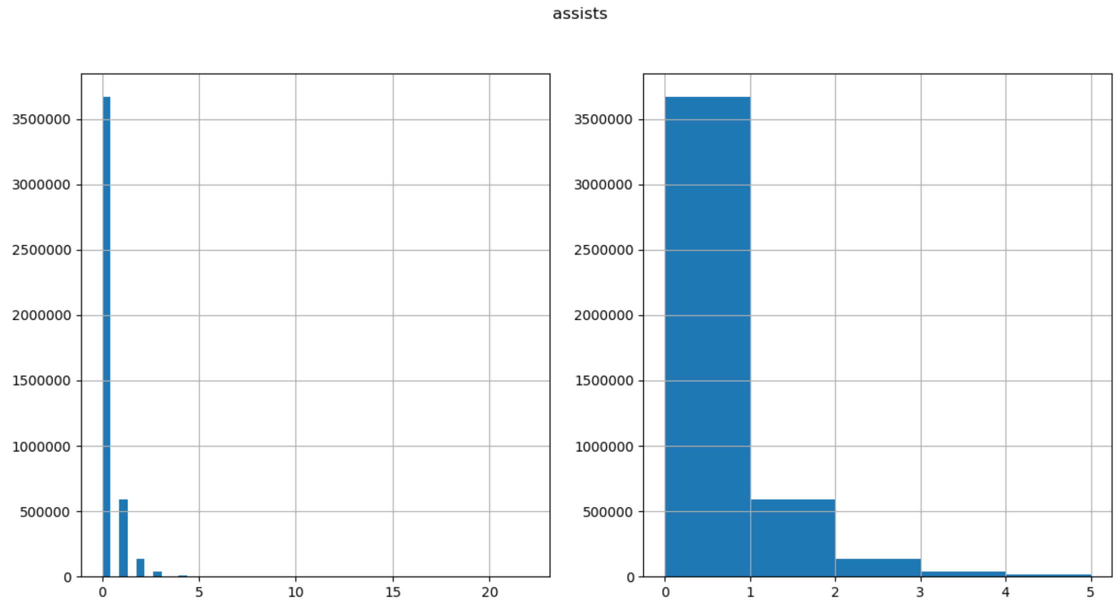
	mean	std	min	25%	50%	75%	max
assists	0.229952	0.578592	0	0	0	0	27
boosts	1.10436	1.713828	0	0	0	2	24
damageDealt	129.7406	167.4371	0	0	84.16	185.4	6229
DBNOs	0.65415	1.13369	0	0	0	1	59
headshotKills	0.225815	0.593392	0	0	0	0	41
heals	1.357999	2.665694	0	0	0	2	75
killPlace	47.81403	27.46275	1	24	48	71	100
killPoints	509.4502	628.8322	0	0	0	1175	2174
kills	0.918357	1.523761	0	0	0	1	58

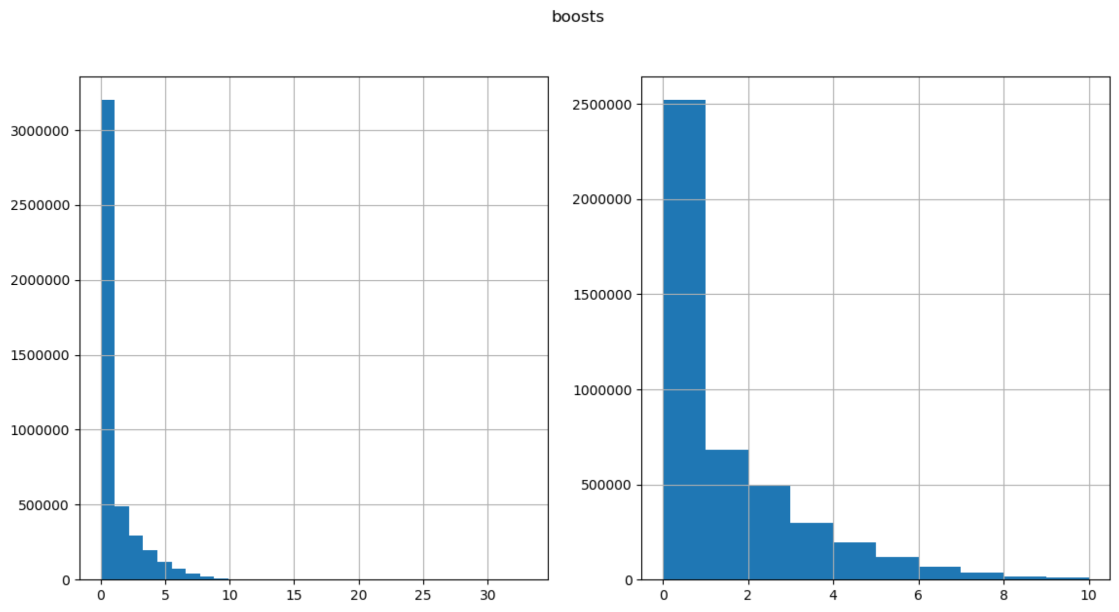
Nextpage

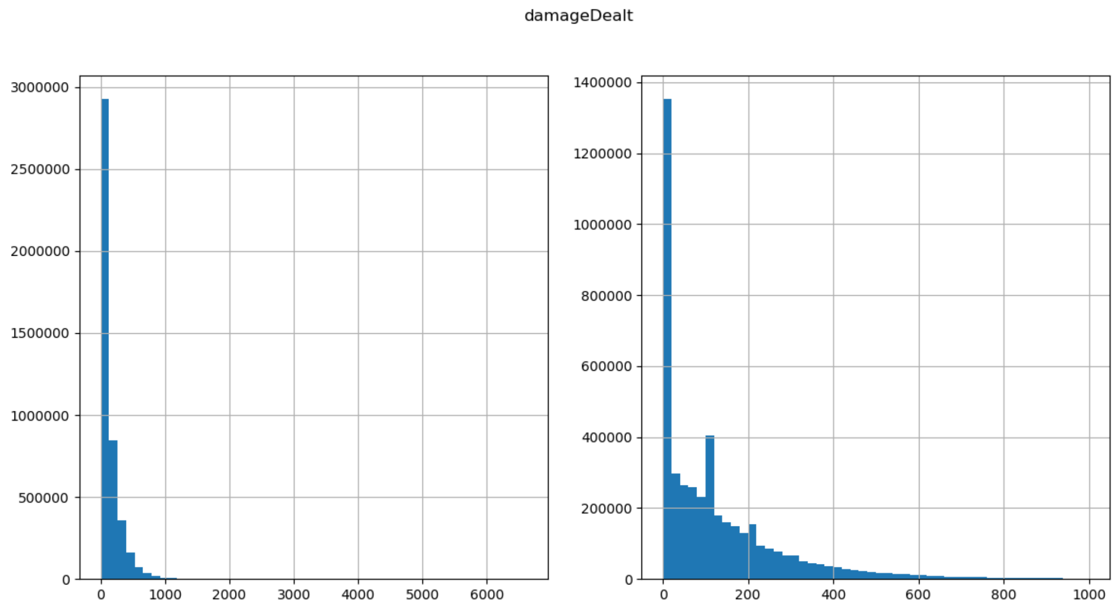
killStreaks	0.543586	0.709011	0	0	0	1	15
longestKill	22.83917	50.57429	0	0	0	21.2	1004
matchDuration	1576.554	256.6299	74	1366	1436	1849	2217
maxPlace	44.88918	23.87529	2	28	30	49	100
numGroups	43.40615	23.29498	1	27	30	47	100
rankPoints	887.3689	737.8231	-1	-1	1442	1500	5742
revives	0.163019	0.468234	0	0	0	0	20
rideDistance	590.9889	1474.899	0	0	0	0.0063	40700
roadKills	0.003409	0.072983	0	0	0	0	15
swimDistance	4.505277	30.26267	0	0	0	0	3271
teamKills	0.023358	0.165087	0	0	0	0	9
vehicleDestroys	0.007681	0.091496	0	0	0	0	5
walkDistance	1149.258	1179.668	0	154.1	679.5	1970	14910
weaponsAcquired	3.62759	2.359042	0	2	3	5	153
winPoints	611.3582	740.6629	0	0	0	1495	2000

Using the list given, we can clearly gain information about different features, and it may help a lot to do some scale change in the later works. But this is not enough, for some features are still hidden: for example, the parameter headshotKills is zero in the first, second quantile, andthe third quantile is 1, but the maximum abount is 64, so it is impossible to get any useful information. So right here, we will do some further analysis upon the parameters and specify the distributon of them. Because the distribution of the training dataset and the testing dataset are similar, we will only focus on the training dataset.

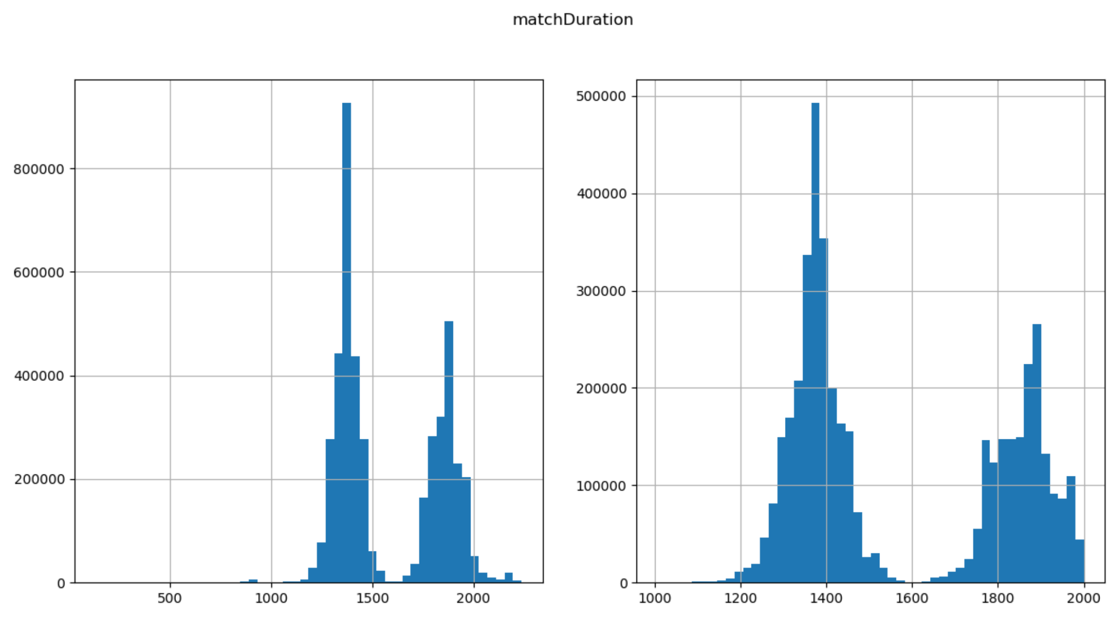
As for the distributions, things are quite different between the different parameters. For most parameters like assists, boosts, damageDealt, nearly all players' stats are around zero, and only very few players has a high record. Some distributions of parameters are as following as examples; here the listed parameters are assists, boosts, and damageDealt respectively:

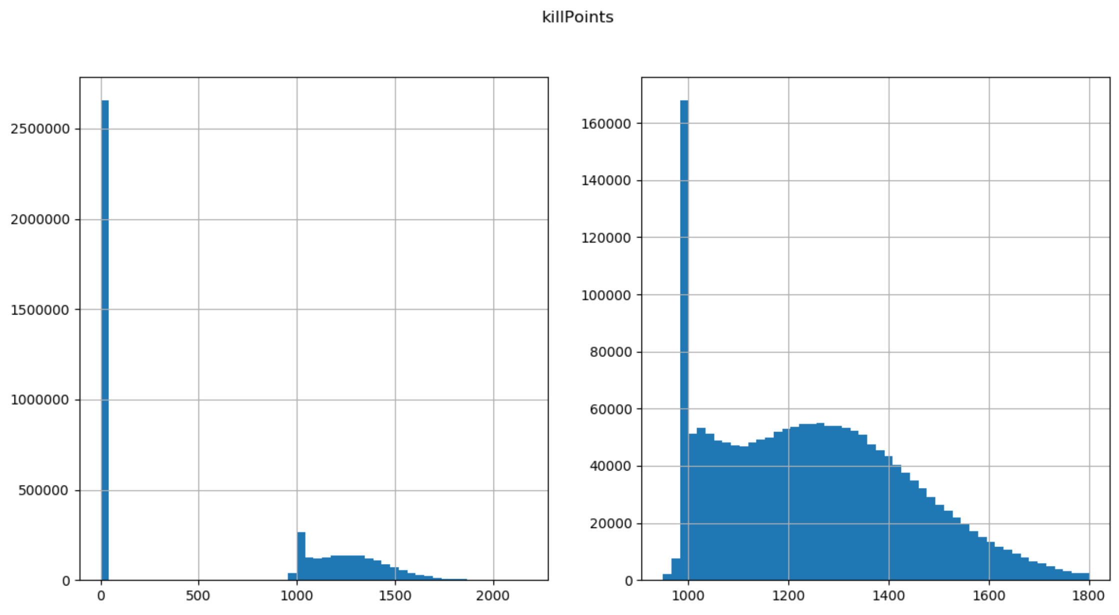




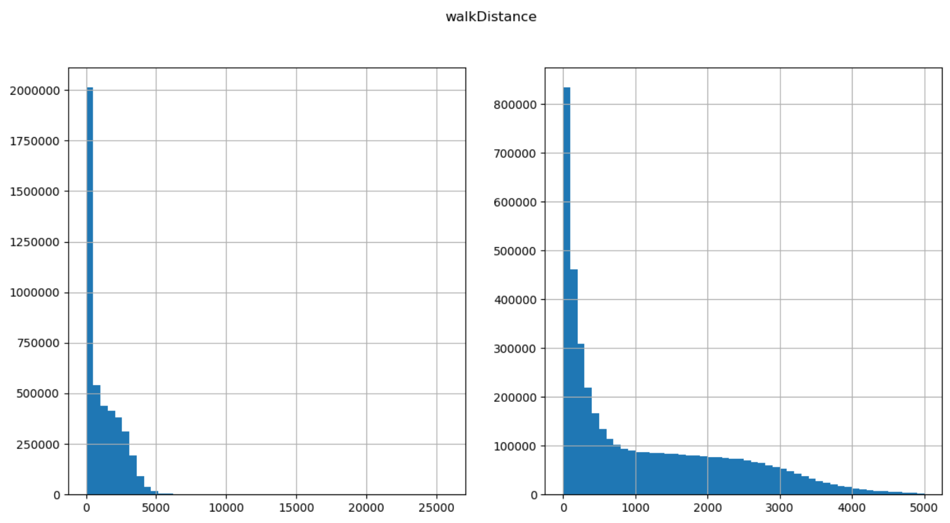


While for some other parameters, the distributions may seem to be a mixture model. That is, the stats of the parameter can be divided into two different parts, and each part has its own distribution. For example, the distribution of matchDuration parameter is approximately a Gaussian Mixture Model, reflects that the maps of these matches may vary. And the killPoint distribution is a combination of Gaussian distribution and a distribution which has only zero as the possible value; and this separated the players into two groups, the first group is of players with really poor performance, and the second group is of players with pretty good performance. By the way, Some distributions of parameters are as following as examples:





By the way, a few parameters are distributed quite scattered so that the stats are not gathered at value zero: for example, the walkDistance. The distribution of walkDistance data is as following:



But this kind of parameters are really rare. Actually, most parameters are clustered at point zero.

2.2 The logic of data preprocessing

To save time and get better models, we tend to use only a part of the training data and do some feature engineering.

2.2.1 Selection of Training Data

When it comes to selecting training data, we try to avoid the matches with lots of missing data. It is known that a match of PUBG contains at most 100 players, so we can count the frequency of each matchId:

```
from collections import Counter
matchIdCounter = Counter(trainDf['matchId'])    # count the matchIds
matchIdCommon = matchIdCounter.most_common()[:]
matchIdUsed = [matchIdCommon[i][0] for i in range(1000)]
trainDfUsed = trainDf[trainDf['matchId'].isin(matchIdUsed)]
```

Note that there is a missing data in column 'winPlacePerc', we should delete the data:

```
trainDf.drop('winPlacePerc', axis=1, inplace=True)
```

2.2.2 Feature Engineering

Sometimes, features that look irrelevant in isolation may be relevant in combination. For example, if the class is an XOR of k input features, each of them by itself carries no information about the class. On the other hand, running a learner with a very large number of features to find out which ones are useful in combination may be too time-consuming, or cause overfitting. By choosing proper features can help us get better models.

We found that players have same 'winPlacePerc' if they are in a same team/group in a match. We view a group but not a single player as a unit.

Our feature engineering can be divided into three parts:

1. Calculate some ratios of existing data and add them into our model, such as:

   ```
   df['killStreakrate'] = df['killStreaks']/df['kills']
   ```

2. Delete some features that looks useless.
3. Calculate some descriptive statistics for a team and drop the data of the single players.

2.3 Data Post-Processing

Our goal is to predict the 'winPlacePerc' of each player, and we found that the value can be viewed as an arithmetic progression in a single match if we regard each group as a unit. For example, the difference of the 'winPlacePerc' between the champion group(=1) and the runner-up group is equal to that between the runner-up group and the second runner-up group, and so on. The last group has a zero 'winPlacePerc', so we can write as:

```
testY1['winPlacePerc'] = ((testY1['numGroups']-testY1['rank'])/
                          (testY1['numGroups']-1))
```

We can also use round function to retain only 4 decimal:

```
testY1['winPlacePerc'] = round(testY1['winPlacePerc'], 4)
```

but it does little help.

3 First choice of model construction–Regression Models

3.1 Methods of linear models

In this part, we introduced some other algorithms to solve this ranking problem. For PUBG dataset, the winPlacePerc represents the relative ranking in the player's corresponding match, so we actually don't care about his actual score, but his ranking. We can use some linear models to solve this problem by giving them 'initial scores' then transfer to the format of their ranking.

By retaining a subset of the predictors and discarding the rest, subset selection produces a model that is interpretable and has possibly lower prediction error than the full model. However, because it is a discrete process—variables are either retained or discarded—it often exhibits high variance, and so doesn't reduce the prediction error of the full model.

Shrinkage methods are more continuous, and don't suffer as much from high variability. Here we use ridge regression (Tikhonov regularization) and LASSO (least absolute shrinkage and selection operator) to do such works. Though originally defined for least squares, they easily extended to a wide variety of statistical models such as proportional hazards models. In our problem it would be am important factor to consider.

3.1.1 Ridge regression

Ridge regression shrinks the regression coefficients by imposing a penalty on their size. The ridge coefficients minimize a penalized residual sum of squares:

$$\hat{\beta}^{ridge} = \underset{\beta}{Argmin} \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 \leq t \tag{2}$$

3.1.2 LASSO

The LASSO is a shrinkage method like ridge with differences on the constraints:

$$\hat{\beta}^{ridge} = \underset{\beta}{Argmin} \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 \tag{3}$$

$$\text{subject to} \quad \sum_{j=1}^{p} |\beta_j| \leq t \tag{4}$$

The LASSO is closely related to basis pursuit denoising.

3.1.3 Localtests

The core of the Ridge regression algorithm is shown below:

```
from sklearn.linear_model import Ridge
resultR = Ridge(alpha=1, max_iter=1e4, tol=1e-4).fit(trainX,trainY)

# predction
predR = np.dot(np.mat(ltX),resultR.coef_)
```

Here 'predR' stands for 'initial scores' predicted by Ridge regression and is the basis for the rank. The Mean Absolute Error on the localtest is about 0.0426.

The core of the LASSO algorithm is shown below:

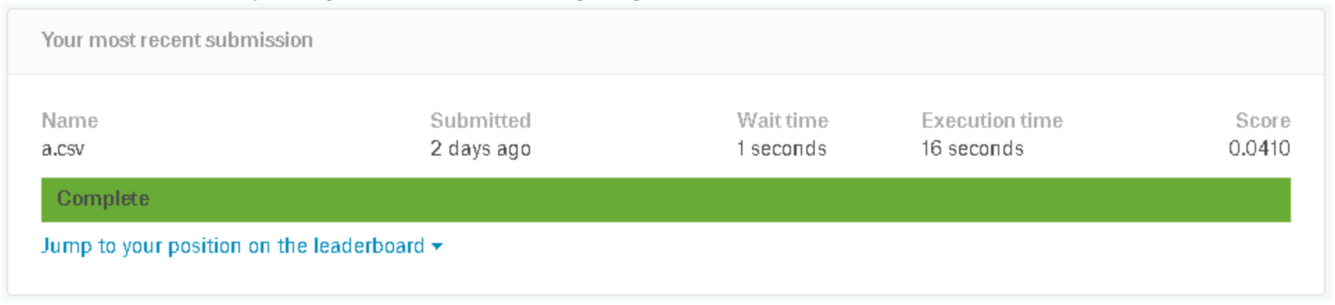
```
from sklearn.linear_model import Lasso
resultL = Lasso(alpha=1, max_iter=1e4, tol=1e-4).fit(trainX,trainY)

# predction
predL = np.dot(np.mat(ltX),resultL.coef_)
```

Here 'predL' stands for 'initial scores' predicted by LASSO and is the basis for the rank. The Mean Absolute Error on the localtest is about 0.0645.

I gave a test on some linear combinations of my models from Ridge regression and LASSO. Merged models have worse performance on the localtest.

3.2 Prediction

Here is the result by using the model from Ridge regression.



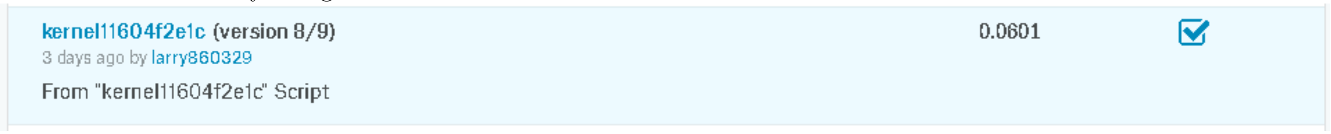
Your most recent submission

Name	Submitted	Wait time	Execution time	Score
a.csv	2 days ago	1 seconds	16 seconds	0.0410

Complete

Jump to your position on the leaderboard ▾

Here is the result by using the model from LASSO.



kernel11604f2e1c (version 8/9) 0.0601 ✓
3 days ago by larry860329
From "kernel11604f2e1c" Script

4 Further attemptation–the XGBoost method

4.1 History of XGBoost

Tree boosting is a highly effective and widely used machine learning method. It works well in tabular-data-based tasks like Anomaly detection, Ads clickthrough, Fraud detection, etc. Tree boosting is based on the theory of Classification and Regression Tree.

The Regression tree (also known as CART) would construct a tree structure to fit the data and predict scores in each leaf. When regression Trees Ensemble and form a forest, its ability to fit training data and predict could be largely enhanced.

There're various algorithms to learn Tree Ensemble. In 1997, Random Forest was proposed by Breiman, it correct regression tree's habit of overfitting to the training set. Later in 1999, Friedman raised Gradient Boost Regression Tree algorithmalgorithm(Abbr: GBRT)[5], which optimize a cost function over function space by iteratively choosing a function that points in the negative gradient direction.

There're some advantages of tree-based methods: First, it has high accuracy and almost half of data science challenges are won by tree based methods. Second it's easy to use because it's invariant to the scale of input data and it can get good performance with little tuning. Third it's easy to interpret and control. However, the conventional tree based methods tend to overfit; also, it's quite a challenge to inprove the training speed and scale up to larger dataset.

To improve the GBRT's generalization ability on test set, Tianqi Chen put forward XGBoost(Extreme Gradient Boosting) algorithm[3]. As a scalable system for learning Tree Ensembles, XGBoost overcome those challenges. First, it set a regularized objective to get better model. Second, it's sparse aware algorithm, which means it weighted the features according to their importance. Third, XGBoost algorithm largely optimizes system and speed through parallelization, cache optimization and distributed computing. In short, it's a faster tool for learning better models.

Initially, XGBoost worked as a terminal application which could be configured using a libsvm configuration file. After winning the Higgs Machine Learning Challenge, it became the secret sauce in Machine Learning circles. Since XGBoost was proposed, it has gained much popularity and attention as it was the algorithm of choice for many winning teams of a number of machine learning competitions.

4.2 XGBoost Algorithm

Assuming we have K trees

$$\hat{y_i} = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F} \tag{5}$$

where $\mathcal{F}$ is the space of regression trees.

Our optimization objective is

$$Obj = \sum_{i=1}^{n} l(y_i, \hat{y_i}) + \sum_{k=1}^{K} \Omega(f_k) \tag{6}$$

where $l(y_i, \hat{y_i})$ training loss, $\Omega(f_k)$ is regularization part, which mearures the complexity of trees. Obviously, this is a bias-variance trade off.

Let's define the complexity of a tree

$$f_k(x) = w_{q(x)} \tag{7}$$

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{8}$$

where w is the weight of the tree, T is the number of leaves, and $\sum_{j=1}^{T} w_j^2$ is the L2 norm of leaf scores.

The objection function is not differentiable, so we can't use methods such as SGD. Actually, the solution is Additive Training (Boosting)

$$
\begin{align}
\hat{y_i}^{(0)} &= 0 \tag{9}\\
\hat{y_i}^{(1)} &= f_1(x_i) = \hat{y_i}^{(0)} + f_1(x_i) \tag{10}\\
\hat{y_i}^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y_i}^{(1)} + f_2(x_i) \tag{11}\\
&\dots \tag{12}\\
\hat{y_i}^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y_i}^{(t-1)} + f_t(x_i) \tag{13}\\
\tag{14}
\end{align}
$$

where $\hat{y_i}^{(t)}$ is the model at training round t, hence we have

$$
\begin{align}
Obj^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t)}) + \sum_{k=1}^{t} \Omega(f_k) \tag{15}\\
&= \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(x_i)) + \sum_{k=1}^{t} \Omega(f_k) \tag{16}
\end{align}
$$

Our goal is finding f_t to minimize $Obj^{(t)}$. Take Taylor expansion of the objective, define $g_i = \partial_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$, $h_i = \partial^2_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$

$$
\begin{align}
Obj^{(t)} &\approx \sum_{i=1}^{n} [l(y_i, \hat{y_i}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \sum_{k=1}^{t} \Omega(f_k) \tag{17}\\
&= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \sum_{k=1}^{t} \Omega(f_k) + Const \tag{18}
\end{align}
$$

Then regroup the objective by leaf

$$
\begin{align}
Obj^{(t)} &\approx \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \sum_{k=1}^{t} \Omega(f_k) \tag{19}\\
&= \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{20}\\
&= \sum_{j=1}^{T} [(\sum_i g_i) w_j + \frac{1}{2}(\sum_i h_i + \lambda) w_j^2] + \gamma T \tag{21}
\end{align}
$$

Let's define $G_j = \sum_i g_i$, $H_j = \sum_i h_i$

$$
Obj^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T \tag{22}
$$

According to the property of single variable quadratic function, we have $w_j^* = -\frac{G_j}{H_j + \lambda}$

$$
Obj = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{23}
$$

Finally, we've got the solution to the Tree model. But there can be infinite possible tree structures. In practice, we grow the tree greedily. Namely, we start from tree with depth 0, then for each leaf node of the tree, try to add a split. The change of objective after adding the split is

$$Gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \lambda \tag{24}$$

This is a trade-off between simplicity and predictiveness, we can either stop split if the best split have negative gain (Pre-stopping) or grow a tree to maximum depth and recursively prune all the leaf splits with negative gain (Post-Prunning).

To recap, XGBoost model optimizes a regularized objective for better generalization and gives out the additive solution. The pseudocode of XGBoost is listed below:

Algorithm 1: Exact Greedy Algorithm for Split Finding

```
Input: $I$, instance set of current node
Input: $d$, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i,\ H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ to $m$ do
    $G_L \leftarrow 0,\ H_L \leftarrow 0$
    for $j$ in sorted($I$, by $\mathbf{x}_{jk}$) do
        $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
        $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
        $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
    end
end
Output: Split with max score
```

4.3 Experiment

XGBoost's clear definitions in algorithm offers extendible modules in software. According to the open-source model, XGBoost provides a gradient boosting framework for C++, Java, Python and it's an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework.

There are some important hyperparameters to be tuned.
1. eta: Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
2. max_depth: Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit.
3. subsample: Subsample ratio of the training instances.
4. colsample_by: Specify the fraction of features to be subsampled.
5. lambda: L2 regularization term on weights. Increasing this value will make model more conservative.
6. alpha: L1 regularization term on weights. Increasing this value will make model more conservative.

To tune the parameter, we firstly choose a bese parameter (mostly default values). Then we take a greedy algorithm to tune the parameters in the order of their importance. For example, we use 5-fold cross-validation method to tune 'eta' and choose the one that gives the best performance, then we tune 'max_depth' in the same way.

We trained our model in a GPU (Nvidia GeForce GTX 108). Assigning different hyperparameters to the model, we got 5 prediction results and computed their average result to ensemble them. Submitting the ensembled result, we got the score of 0.0213(MAE), which ranks 310th in the Competiton.

5 Another possible method–Learning to rank

In this part, we introduced another algorithm to solve this ranking problem. For PUBG dataset, the winPlacePer represents the relative ranking in the player's corresponding match, so we actually don't care about his actual score, but his ranking. We consider a neural network-based ranking method LambdaRank to solve this problem. The ranking try is inspired by an overview about the development of current ranking methods [2].

5.1 RankNet

Gradient descent method can be used for learning ranking functions. In the training process, one specific sample of a query may outweigh another sample with a probability distribution. The corresponding probability cost function is introduced in [1] and a implementation RankNet is put forward. At a given point during training, an output score s_i for each training data x_i is produced. For this pair of data, the predicted probability that x_i ranks before x_j is defined as a sigmoid function:

$$\hat{P_{ij}} = P(R_i > R_j) = \frac{1}{1 + e^{-\sigma(s_i - s_j)}} \tag{25}$$

where R_i denotes the rank of sample i in its comparison group and the parameter σ decides the shape of the sigmoid function.

For each training sample pair, the relative ranking is known, let $S_{ij} \in \{0, 1, -1\}$ to be defined as 1 if sample i ranks before j, -1 if sample i ranks after j, and 0 if they rank tie. Then the label probability can be defined as:

$$P_{ij} = \frac{1 + S_{ij}}{2} \tag{26}$$

The cross entropy cost function is applied:

$$C = -P_{ij} log \hat{P_{ij}} - (1 - P_{ij}) log(1 - \hat{P_{ij}}) \tag{27}$$

Combining above three equations gives:

$$C = \frac{1}{2}(1 - Sij)\sigma(s_i - s_j) + log(1 + e^{-\sigma(s_i - s_j)}) \tag{28}$$

if $S_{ij} = 1$, then the cost function will be:

$$C = log(1 + e^{-\sigma(s_i - s_j)}) \tag{29}$$

while for $S_{ij} = -1$,

$$C = log(1 + e^{-\sigma(s_j - s_i)}) \tag{30}$$

After the cost function is ready, the weights can be updated with stochastic gradient descent:

$$\omega_k \rightarrow \omega_k - \eta \frac{\partial C}{\partial \omega_k} = \omega_k - \eta(\frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial \omega_k} + \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial \omega_k}) \tag{31}$$

where η is the learning rate. Explicitly:

$$\delta C = \sum_k \frac{\partial C}{\partial \omega_k}\delta \omega_k = -\eta \sum_k (\frac{\partial C}{\partial \omega_k})^2 < 0 \tag{32}$$

This shows that the loss is reduced after the gradient descent. The cost function also gives its derivative:

$$\frac{\partial C}{\partial s_i} = \sigma(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}} = -\frac{\partial C}{\partial s_j} \tag{33}$$

let λ_{ij} be defined as:

$$\lambda_{ij} = \sigma(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}} \tag{34}$$

Then the weight update can be rewritten as:

$$\delta\omega_k = -\eta \sum_{\{i,j\}\in I} (\lambda_{ij}\frac{\partial s_i}{\partial \omega_k} - \lambda_{ij}\frac{\partial s_j}{\partial \omega_k}) = -\eta \sum_i \lambda_i \frac{\partial s_i}{\partial \omega_k} \tag{35}$$

where I denote the set of pairs of indices $\{i,j\}$, and λ_i is defined as:

$$\lambda_i = \sum_{j:\{i,j\}\in I} \lambda_{ij} - \sum_{k:\{k,i\}\in I} \lambda_{ki} \tag{36}$$

5.2 Information Retrieval Measures

In lambdaRank algorithm, Normalized Discounted Cumulative Gain (NDCG) [4] is introduced to measure the ranking quality. This measure index is defined as follows. The DCG (Discount Cumulative Gain) for a set of given search results is:

$$DCG@T = \sum_{i=1}^{T} \frac{2^{l_i} - 1}{log(1+i)} \tag{37}$$

where T is the truncation level (for example, if we only care about the first page of returned results, we might take $T = 10$), and l_i is the label of the ith sample in the group. Then NDCG is the normalized version of this:

$$NDCG = \frac{DCG@T}{maxDCG@T} \tag{38}$$

5.3 LambdaRank

LambdaRank is an empirical algorithm, The road from RankNet to LambdaRank is easy. The $lambda_{ij}$ in RankNet is modified to:

$$\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = \frac{-\sigma}{1+e^{\sigma(s_i-s_j)}} |\Delta_{NDCG}| \tag{39}$$

and the gradient descent process remains the same.

5.4 Experiment

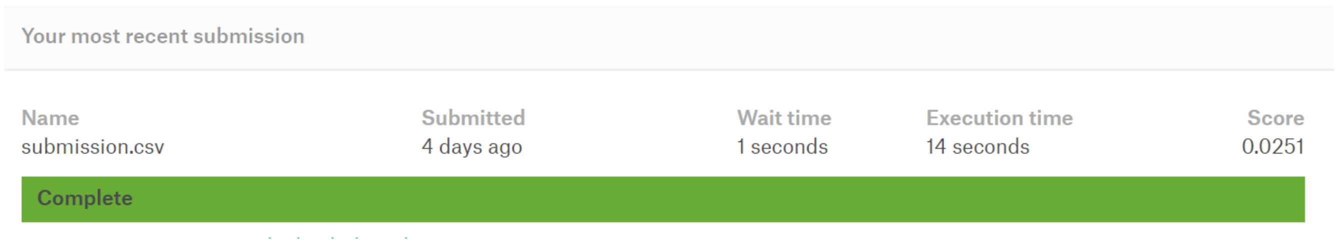
5.4.1 Hyperparameters

We test on our algorithm on a server which has 128GB CPU, we don't use GPUs in the experiment. We use XGboost's embedded LambdaRank algorithm. We have several hyperparameters to tune:

- eta: learning rate;
- gamma: minimum split loss;
- min_chile_weight: Minimum sum of instance weight needed in a child;
- max_depth: maximum depth of a tree;
- eval_metric: loss matrix;
- colsample_bytree: this is a parameter for subsampling of columns;
- subsample: Subsample ratio of the training instances;
- scale_pos_weight: Control the balance of positive and negative weights.

5.4.2 Group the data

One thing we have to do is to set each data to its corresponding group. We have to implement it explicitly. The ranking of each player is a relative ranking in the specific match. So we count the amount of each match and set the group DMatrix for pairwise ranking.


Your most recent submission

Name submission.csv | Submitted 4 days ago | Wait time 1 seconds | Execution time 14 seconds | Score 0.0251

Complete

Figure 1: Learning to Rank: score

5.4.3 Implementation

The complete code can be found in the supplementary materials, here we provide the sketch of the training module.

```python
def rank_train():
    import xgboost as xgb
    from xgboost import DMatrix
    from sklearn.model_selection import train_test_split

    xgb_model = xgb.train(params,
        train_dmatrix, num_boost_round=19,
        evals=[(val_dmatrix, 'validation')])
    pred = xgb_model.predict(test_dmatrix)
    return pred
```

5.4.4 Results

We split the training set and validation set with a 7:3 ratio. Cross-validation is used to obtain the proper hyperparameters. The final parameters can be found in the supplementary materials. In the training process, the final loss is 0.0252, while on the test set, the loss is 0.0251. So our model can generalize very well in novel data environment without over-fitting or under-fitting. With this method alone, we achieve top 30% ranking in the Kaggle competition with more than 1400 teams.

6 Conclusion

We compared these three methods as 3 shows. We can see that XGboost tree method achieves the best score

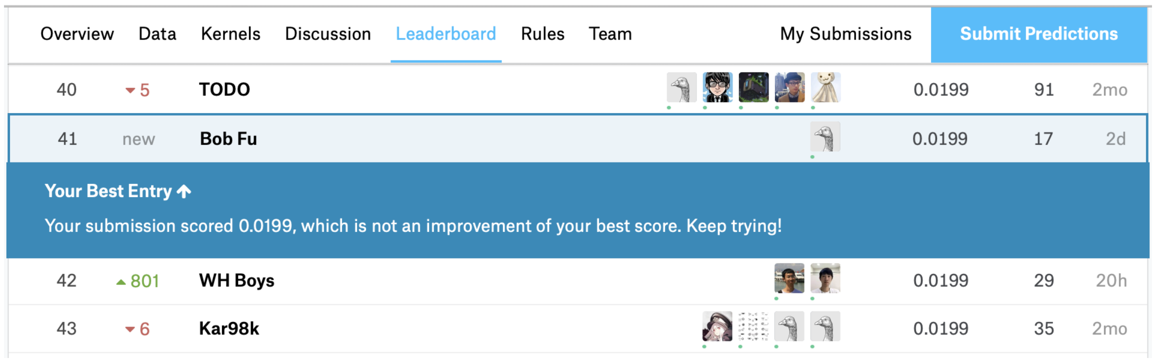
Method	Score
Ridge regression	0.0410
Xgboost tree	0.0212
Learning to Rank	0.0251
Ensemble model	0.0199

Table 3: A Summary: ranking scores with different methods

compared to the rest two. Ridge regression is the simplest model, and its performance is the best. Learning to rank utilized neural networks, and the result is alighlt worse than XGboost tree. When we ensemble these three methods, the result is improved significantly.

7 Acknowledgement

Thanks Laihe ZHANG for his work in Regression ModelsThanks; thanks Mr. Boming FU for his work in XGBoost Algorithm; thanks Chao NI for his works in Learning to Rank; thanks Huayu LI for his work in summarizing the feature information; They completed the final project writing together.



40	▾ 5	TODO	0.0199	91	2mo
41	new	Bob Fu	0.0199	17	2d
42	▴ 801	WH Boys	0.0199	29	20h
43	▾ 6	Kar98k	0.0199	35	2mo

Your Best Entry ↑

Your submission scored 0.0199, which is not an improvement of your best score. Keep trying!

Figure 2: Caption

References

[1] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96. ACM, 2005.

[2] Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

[3] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[4] Kalervo Järvelin and Jaana Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 41–48. ACM, 2000.

[5] Wikipedia contributors. Gradient boosting — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-January-2019].