

# Kaggle PUBG Finish Placement Prediction Report

Team name:

**TODO**

Team member (alphabet order):

Daniel Zhang,  
Shangyu Zhang,  
Qiang Guo,  
Wenxiao Xiao,  
Zijie Wu

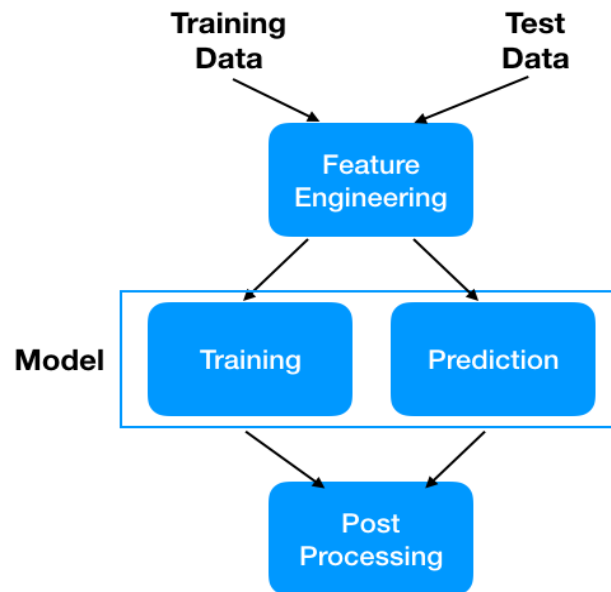
12/02/2018

# Introduction & Outline

PUBG is a very popular video game in which 100 players are dropped onto an island bare-handed and player must find his way to explore, scavenge, and eliminate other players until he is the last one standing. Since it is so popular across many different countries and regions, lots of players discuss about how to win the game, what's the best strategy to win in PUBG? Should you sit in one spot and hide your way into victory, or do you need to be the top shot?

In order to figure that out, We dugged into 65,000 games' worth of anonymized player data and see what kind attributes are highly correlated with the final placement. Also, we tried to build different models to predict the final placement of each player based on their in-game statistics.

The general idea to approach this challenge is to train a **regressor** from given training data to predict a continuous number that represents the predicted finish placement for each team/player. The general structure of a solution is as follow:



During training, first we sample 20% (or 30%) of the training data to test the performance of our model, and fine-tune its parameters. Then we train the model given full training dataset, and predict the result on test data.

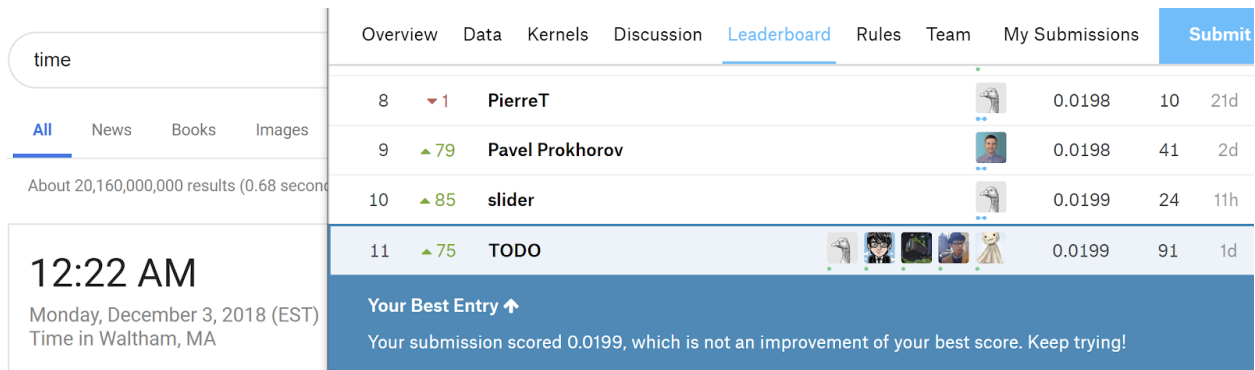
Specifically, after preprocessed the raw data such as dealing with the missing value, we did some visualizations to preliminarily know the distribution of different attributes and the correlation between different attributes and target variable. Then, we spent most of the time on feature engineering and tuning parameter which will be discussed in detail in the feature

engineering section. As for model selection, we tried Random Forest, Neural Network, MLP, Pytorch, LightGBM, etc. based on the type of this task and our final mean absolute error is 0.0199 which ranked 12th across the whole competition.

Regarding to the work of each team member, Qiang, Shangyu, Zijie created our own neural network model from scratch using keras and the lowest mae we get is 0.0209.

In the meantime, Daniel and Wenxiao are standing on the shoulder of the giant and trying to build things up from the model with high score of public leaderboard. They managed to boost the score to 0.0199.

Best ranking we have achieved:



time	Overview	Data	Kernels	Discussion	Leaderboard	Rules	Team	My Submissions	Submit
All News Books Images									
About 20,160,000,000 results (0.68 seconds)									
12:22 AM Monday, December 3, 2018 (EST) Time in Waltham, MA	8	▼ 1	PierreT					0.0198	10 21d
	9	▲ 79	Pavel Prokhorov					0.0198	41 2d
	10	▲ 85	slider					0.0199	24 11h
	11	▲ 75	TODO					0.0199	91 1d
	<b>Your Best Entry</b> ↑ Your submission scored 0.0199, which is not an improvement of your best score. Keep trying!								

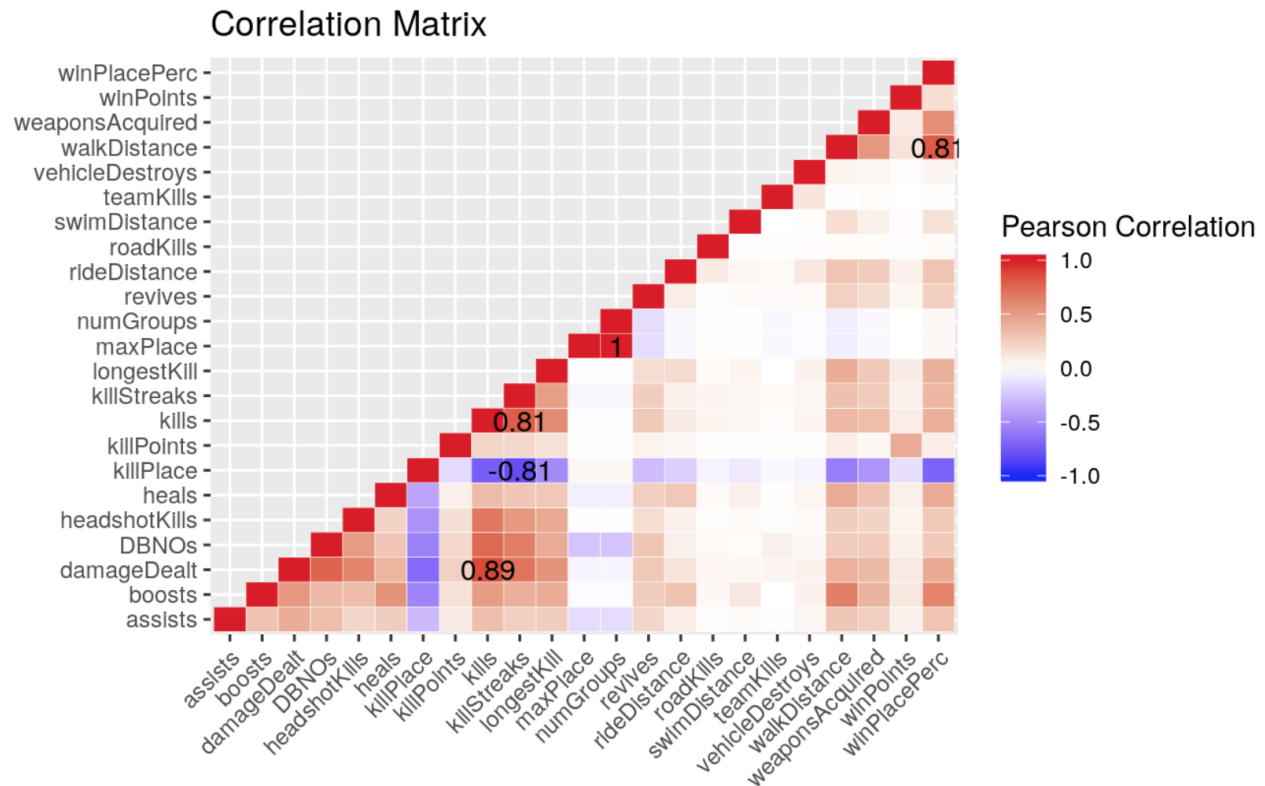
## Feature Engineering

Before any actual feature engineering, we exclude outliers from the dataset that could affect our model badly. We remove the matches with no players or just one player:

```
df = df[df['maxPlace' > 1]]
```

Then, we begin to analyze the data. First, we draw the feature correlation matrix referenced to a report made by Shaun Loong<sup>1</sup>.

<sup>1</sup> <https://www.kaggle.com/slmf1995/exploring-pubg-match-statistics-rankings>



From the matrix, we can find what kind of features strongly relevant to the target winPlacePerc, and what kind of features are highly correlated. For example, we can tell that walkDistance contributes most to winPlacePerc and kills is highly correlated to killStreaks. Then, we can choose the important features and generate new features based on original features.

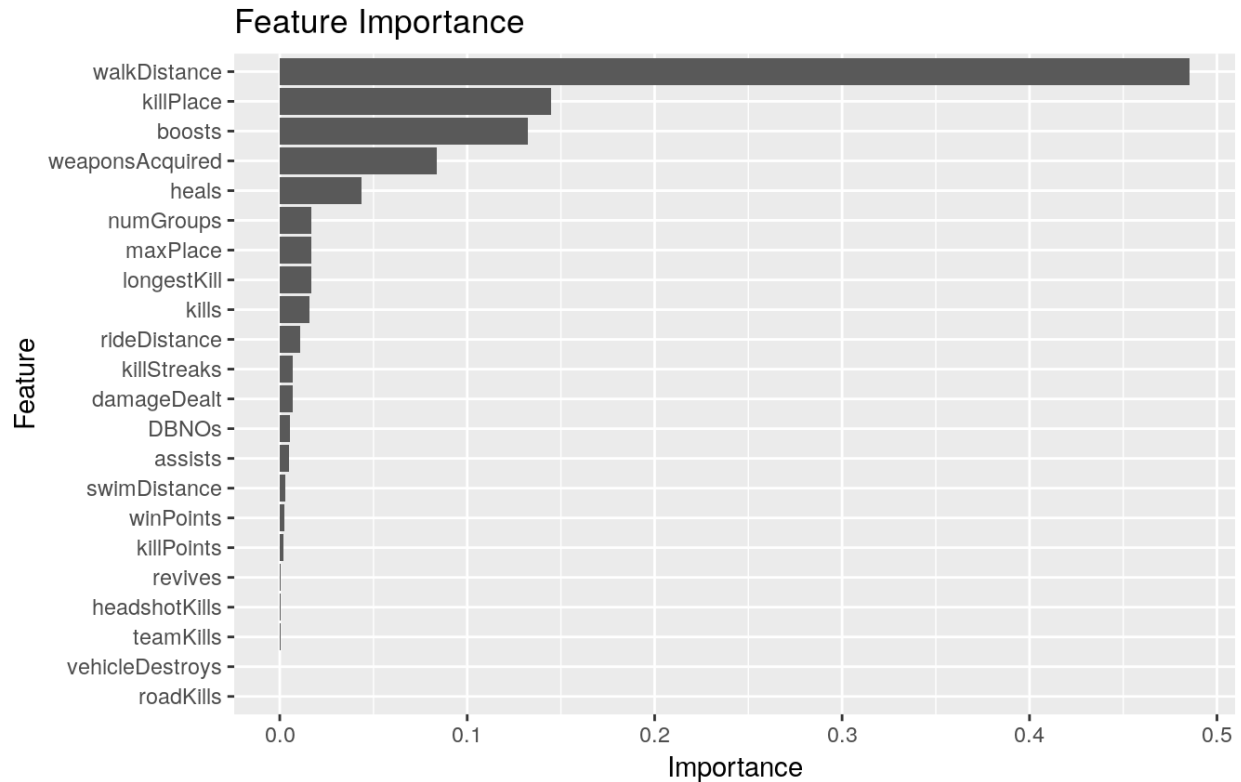
Specifically, we referenced feature engineering codes from here<sup>2</sup> and here<sup>3</sup>. It creates some new features from existing features, like total distance, headshot rate, killStreak rate, kills per walk distance and so on. Such new features are more meaningful than the original ones, thus helping us easily highlight skilled players, who are more likely to achieve higher ranks.

```
df['totalDistance'] = df['rideDistance'] + df['walkDistance'] + df['swimDistance']
df['headshotrate'] = df['kills'] / df['headshotKills']
df['killStreakrate'] = df['killStreaks'] / df['kills']
df['killsPerWalkDistance'] = df['kills'] / df['walkDistance']
```

Besides, we also make some tweaks when generating new features. For instance, let's have a look at the new feature "total distance", which is composed of 3 attributes rideDistance, walkDistance and swimDistance. From a feature importance chart referenced to Shaun Loong below,

<sup>2</sup> <https://www.kaggle.com/anycode/simple-nn-baseline-4>

<sup>3</sup> <https://www.kaggle.com/chocozzz/lightgbm-baseline>



We can tell rideDistance, walkDistance and swimDistance contributes differently to the winPlacePerc, therefore, we accordingly add weights like,

```
df['totalDistance'] = 0.038 * df['rideDistance'] + 0.96 * df["walkDistance"] + 0.002 *
df["swimDistance"]
```

, which makes the new feature totalDistance more reasonable.

Next, we remove meaningless features like matchId, playerId, groupId and matchType. They merely serve as identifiers for some entities, and they have nothing to do with the skill of the player.

```
features.remove("Id")
features.remove("matchId")
features.remove("groupId")
features.remove("matchType")
```

Considering the most popular game mode is squad mode, we have to do some feature engineering jobs based on groups. In squad mode, every player shares the same winPlacePerc. Sometimes the best player determines the final winPlacePerc, while sometimes the weakest player drags down his team. According to this, we generate group features like,

```
# Generate group min features
agg = df.groupby(['matchId', 'groupId'])[features].agg('min')
agg_rank = agg.groupby('matchId')[features].rank(pct=True).reset_index()
```

```
df_out = df_out.merge(agg.reset_index(), suffixes=["", ""], how='left', on=['matchId', 'groupId'])
df_out = df_out.merge(agg_rank, suffixes=["_min", "_min_rank"], how='left', on=['matchId', 'groupId'])

# Generate group max features
agg = df.groupby(['matchId', 'groupId'])[features].agg('max')
agg_rank = agg.groupby('matchId')[features].rank(pct=True).reset_index()
df_out = df_out.merge(agg.reset_index(), suffixes=["", ""], how='left', on=['matchId', 'groupId'])
df_out = df_out.merge(agg_rank, suffixes=["_max", "_max_rank"], how='left', on=['matchId', 'groupId'])
```

Also, in most cases, the average performances of a team determines their final rankings. According to this, we generate group features like,

```
# Generate group mean features
agg = df.groupby(['matchId', 'groupId'])[features].agg('mean')
agg_rank = agg.groupby('matchId')[features].rank(pct=True).reset_index()
df_out = df_out.merge(agg.reset_index(), suffixes=["", ""], how='left', on=['matchId', 'groupId'])
df_out = df_out.merge(agg_rank, suffixes=["_mean", "_mean_rank"], how='left', on=['matchId', 'groupId'])
```

Other group features like group size, group median, group sum may also play important roles. We add these group features to our dataset in a similar way mentioned above.

Finally, we have finished all the feature engineering work.

## Model Selection

Models we have tried:

Type	Framework	Team member	Best score
Neural Network	Keras	Zijie	0.0282
Neural Network	Ultimate MLP	Shangyu	0.0245
Neural Network	Keras	Qiang	0.0209
Neural Network	PyTorch	Daniel & Wenxiao	0.0203
Gradient Boosting	LightGBM	Wenxiao	0.0202

# Model Tuning

## 1. Build from scratch (Keras)

First, we start with a vanilla neural network with 3 layers  $128 * 64 * 32$  and get the result 0.0229. Then we begin to add some decoration.

### 1.1. Dropout layer

To avoid overfitting, two dropout layers are added in the middle of the model and the mae increased drastically to 0.0284. We then think that this might be because we drop out too many neurons which leads to information loss. Then we keep only the second dropout layer which turns out to be even worse (0.0301).

At this time, it occurs to us that we might need to keep the first dropout layer instead of the second. After doing this, the mae is 0.0258, which is still worse than 0.0229. At this time, it is certain that we have not reached the point of overfitting.

### 1.2. Learn more epochs

Since we have not reached the point of overfitting, then we probably are in the state of underfitting. We increase the epoch from 36 to 72(0.0243) and then to 288(0.0234). Though there are some increment, but after reading the log, it occurs to us that after epoch 150 the loss and mae seems to be stagnant already, we change to epoch to 158 and mae is now 0.0217, which is a nice improvement from the vanilla neural network.

### 1.3. New initialization and activation

From this point, we first try some new activation function and go over all the options keras provide us. First try is elu(exponential linear unit), which gives a terrible result 0.0262. And at this time, we think it can be very low efficient to go over all the options of activation function, so we hope that there is some absolute good or bad activations to choose from. Since most comments show that relu is currently the best, then we roll back and begin to try a new way of initialization: glorot\_uniform. This gives the new current best 0.0215. With hope of an even better performance, we try another way of initialization glorot\_normal, but the mae is 0.0257.

At the same time, we know a new relu called leaky relu and take it for granted it should be better than relu, but this gives 0.0234.

## 1.4. New way of scale

We noticed that most teams use  $y = y * 2 - 1$  to scale their data, and wonder why min max scaler is not applied. This trial gives us a result of 0.0305, clearly it's not performing well so we discard it.

## 1.5. Optimal validation split ratio

Andrew Ng offer a ratio of 0.01/0.005 as validation ratio, thus we add that on our training samples (0.0221) and also prelu activation (0.0500). Up to now validation split ratio is set and no new activation function is applied.

## 1.6. Larger brain

We enlarge the neural network from 128\*64\*32 to 256\*128\*64 hoping that more capacity brings more accuracy. This brings us the new best 0.0214. Then for the first time we tune down the learning rate to 0.003 and again a new best 0.0209. Then we increase epoch (0.0209), change the batch size (0.0218), decrease the learning rate (0.0216) and more (0.0251), increase the learning rate (0.0215), add batch normalization (0.0222). Most of them are not supposed to have impact on the result (like normalization is only supposed to make training the model faster) but instead end up negative influence on our model. We think this partially has something to do with the randomness of initialization but mostly because we do not land on a low biased model.

# 2. Stand on the giant's shoulders

Three models' architecture are introduced in this part: MLP (pyTorch) from Ceshine Lee, MLP (ultimate) from harshit and LightGBM from Hyun woo kim.

## 2.1. MLP model with PyTorch

While Qiang, Shangyu, Zijie are building neural network from scratch, Daniel and Wenxiao tried to learn from the excellent work of other competitors. Due to the expensive time complexity of previous MLP models, we hope to find a way to shorten the running time of the neural network learner. That's the reason we choose to build the model with PyTorch, which allows use GPU to speed up the learning process.

We initialize the model with 4 hidden layers,  $398 * 128 * 128 * 128 * 128 * 1$  (398 is the number of features after feature engineering). And thanks to the experience from the keras model, we used Relu as the activation function. The raw mae of this initial run is 0.0237, and after adjusted to percentile, the mea is 0.0204. Because of the large volume of this training data and high dimensional feature space, firstly we want to improve the performance by increase the size of our model. After changing the sizes of hidden layers, we achieve a raw mae of 0.0235(0.0203 after adjustment) with a size of  $398 * 256 * 128 * 128 * 128 * 1$ . We also try to add another



hidden layer or increase the size of second and third hidden layer, but cannot make any improvement.

Secondly, since all features are normalized between -1 to 1, we think changing activation function could improve the performance, especially Tanh and Hardtahn. As expected, after trying different activation functions including ELU, LeakyReLU, LogSigmoid, Sigmoid, PReLU, Tanh and Hardtanh, Tanh turns out to be the best for this model. Also at this step we twisted the feature engineering a little, assigned weight to walkDistance, swimDistance and driveDistance according to their importance, and fill missing values in rankPoints, winPoints and killPoints. Notice that if one element has a missing rankPoints, it must have winPoints and killPoints, so we just fill the average of winPoints and killPoints into the missing rankPoints, and vice versa. After this step, we achieved 0.0234 before adjustment, which suggested we improved the MLP model. However, after adjustment, the result is still 0.0203, which may suggest that the performance of this model could not improve at this stage.

## 2.2. MLP by ultimate

This library is the early one on which we try to build the neural network. The architecture is from harshit, who ranked 3rd on the leaderboard with mae 0.0209. First, we add one more layer to make it 64\*32\*16, which gives a result of 0.0211. Then we try to enlarge it even more by making it 128\*64\*32, which gives a result of 0.0208. Then attempts are made to scale the prediction in another way (0.0228) and eliminate some features with little importance (0.0230), then we decide to stop at this place since training with this library does not achieve lots of improvement and is quite time consuming.

## 2.3. Gradient Boosting model with LightGBM

After several failed attempts to future improve our MLP model, we decided to try other models in order to achieve a better result. And gradient boosting seems to be promising for a large data set like this. We try out the baseline of Hyun woo kim and get a mae 0.0205 after adjustment. In the baseline model, "objective" = "regression", "metric" = "mae", 'n\_estimators' = 10000, 'early\_stopping\_rounds' = 200, "num\_leaves" = 31, "learning\_rate" = 0.05, "bagging\_fraction" : 0.7, "bagging\_seed" : 0, "num\_threads" : 4,"colsample\_bytree" : 0.7

In order to improve the performance, we first applied our feature engineering. And noticed that although in the baseline model early stopping rounds was set to 200, the model did not stop early, which means the validation mae keeps dropping until 10000 rounds. Firstly after setting learning rate to 0.08, we have an early stopping but the performance is actually worse (0.0206). So we set learning rate back to 0.06 and increase the number of estimators to 20000, this time we also have an early stopping, and the performance increases to 0.0204. In our final attempt, we further decreases learning rate to 0.04 and get the best result so far, 0.0202. We also tried

lower learning rate to 0.03 and increase number of estimators to 25000, but the mae after adjustment did not decrease anymore.

## Post processing

After we have the prediction from our model, given test data, the next and last step is to process it, during a process called post-processing. The main idea is to adjust the result to correctly reflect the placement as if in a real game.

We know that there are 3 game modes that this dataset captures: solo, duo and squad. Take solo mode for example, there will be at most 100 teams with exactly one player in each team, so the final placement for players could be any integer range from 1 to 100, and it is represented by a real number range from 0.0 to 1.0 in this challenge, where 0.0 means the last place (100th), and 1.0 means the first place (1st).

But since we are training a regressor, the output from the model may not fall in range from 0.0 to 1.0 at all, and even they do, the value is not going to be exactly what the placement is, converted to real number within range 0.0 and 1.0. For example, for a match with 100 players, the output of our model could be 0.499, we can tell that its predicting a placement of 50th, but the correct prediction for 50th would be 0.5, so there is a tiny amount of error here ( $0.5 - 0.499 = 0.01$ ). Fortunately, we could eliminate such error during post-processing.

The same principle applies for other game modes. To do this, we retrieve the number of teams in the same match i.e. these data with same match ID, then calculate the true ranking based on max number of teams in this match.

Given above idea, we do two things during this step: first, we clamp the output between 0.0 and 1.0. Then, we recalculate the “correct” placement, given the output and the number of teams in that match<sup>4</sup>.

Additionally, we observe an increase in performance when merging two results from different models. That is, we (Daniel) take two results from two different model (MLP and PyTorch in this case), and create a new result simply by taking theirs mean:

```
final_result["winPlacePerc"] = (final_result["winPlacePerc_x"] +  
final_result["winPlacePerc_y"]) / 2
```

---

<sup>4</sup> <https://www.kaggle.com/anycode/simple-nn-baseline-4>

The performance is increased from 0.0202 to 0.02, although tiny, but real. This method is essentially creating an ensemble of several different models and take their best results.

## Conclusion & Future work

We achieved our best single model result using gradient boosting model, which suggests this model is very promising. Unfortunately, due to the large volume of this dataset and high dimension of feature space after feature engineering, we don't have enough time to make sure we get the best result out of this model. Also in this competition we want the best possible result, so all features are kept after feature engineering. However, some of the feature might be insignificant and can be discard to speed up the learning process.

In the future, we would like to reduce the feature space by identifying insignificant features. After that we may be able to use tools like grin search to find the best possible parameters for this model and get a better result with less running time. And we also hope to find more suitable models for this data, and ensemble all results with majority voting, to check if this method can yield a better result.