

BAN 676 - Deep Learning

Project: PUBG Finish Placement Prediction

Project team members:

Saidakbar PardaeV (tu3643)

Navneet Chauhan (fg3276)

Antony Paulson ChazhooR (ca6953)

Ashish Solanki (na5442)

Project Link: <https://www.kaggle.com/c/pubg-finish-placement-prediction>

Publish Date: October 1st, 2018

Data Type: PUBG game players statistics after each game

Prediction: Player's finishing placement based on their final statistics.

“1” is First Place

“0” is Last Place

Submission: the lower the mean absolute error (MAE) on test data set, the better position in leaderboard.

1.1 INTRODUCTION

PlayerUnknown's Battlegrounds (PUBG) is an online multiplayer battle royale game developed and published by PUBG Corporation, a subsidiary of South Korean video game company Bluehole. 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.

The battleground is a player versus player shooter game in which up to one hundred players fight in a battle royale, a type of large-scale last man standing deathmatch where players fight to remain the last alive. Players can choose to enter the match solo, duo, or with a small team of up to four people. In either case, the last person or team left alive wins the match. In the game, up to one hundred players parachute onto an island and scavenge for weapons and equipment to kill others while avoiding getting killed themselves. The available safe area of the game's map decreases in size over time, directing surviving players into tighter areas to force encounters.

The team at PUBG has made official game data available for the public to explore and scavenge outside of "The Blue Circle". We have given over 65,000 games' worth of anonymized player data, split into training and testing sets, and asked to predict final placement from final in-game stats and initial player ratings.

Please note: This competition is not an official or affiliated PUBG site - Kaggle collected data made possible through the PUBG Developer API.

1.2 QUESTIONS AND FINDINGS

- What would be the best strategy for Player to win in PUBG?
- Should a player sit in one spot and hide into victory? Or a player needs to be top shot?

1.3 DATA DESCRIPTION

In a game, up to 100 players start (matchId). Players can be on teams (groupId) which get ranked at the end of the game (winPlacePerc) based on how many other teams are still alive when they are eliminated. In a game, players can pick up different munitions, revive downed-but-not-out (knocked) teammates, drive vehicles, swim, run, shoot, and experience all of the consequences -- such as falling too far or running themselves over and eliminating themselves.

We are provided with a large number of anonymized game stats, formatted so that each row contains one player's post-game stats. The data comes from matches of all types: solos, duos, squads, and custom; Also, there is no guarantee of there being 100 players per match, nor at most 4 players per group.

Datasets:

Training Set

Test Set

Data Fields:

DBNOs: No. of enemy players knocked

assists: No. of enemy players this player damaged that were killed by teammates

boosts: No. of boost items used

damageDealt: Total damage dealt. *Note: Self-inflicted damage is subtracted*

headshotKills: Number of enemy players killed with headshots

heals: No. of healing items used

Id: Player's Id

killPlace: Ranking in match of no. of enemy players killed.

killPoints: Kills-based external ranking of the player.

If there is a value other than -1 in rankPoints, then any 0 in killPoints should be treated as a "None".

killStreaks: Max no. of enemy players killed in a short amount of time.

kills: No. of enemy players killed.

longestKill: Longest distance between player and player killed at the time of death. This may be misleading, as downing a player and driving away may lead to a large longestKill stat.

matchDuration: Duration of the match in seconds.

matchId: ID to identify the match. There are no matches that are in both the training and testing set.

matchType: String identifying the game mode that the data comes from. The standard modes are "solo", "duo", "squad", "solo-fpp", "duo-fpp", and "squad-fpp"; other modes are from events or custom matches.

rankPoints: Elo-like ranking of the player. This ranking is inconsistent and is being deprecated in the API's next version. Value of -1 takes place of "None".

revives: No. of times this player revived teammates.

rideDistance: Total distance traveled in vehicles measured in meters.

roadKills: Number of kills while in a vehicle.

swimDistance: Total distance traveled by swimming measured in meters.

teamKills: No. of times this player killed a teammate.

vehicleDestroys: No. of vehicles destroyed.

walkDistance: Total distance traveled on foot measured in meters.

weaponsAcquired: No. of weapons picked up.

winPoints: Win-based external ranking of the player.

If there is a value other than -1 in rankPoints, then any 0 in winPoints should be treated as a "None".

groupId: ID to identify a group within a match. If the same group of players plays in different matches, they will have a different groupId each time.

numGroups: Number of groups we have data for in the match.

maxPlace: Worst placement we have data for in the match. This may not match with numGroups, as sometimes the data skips over placements.

winPlacePerc: The target of prediction. This is a percentile winning placement, where 1 corresponds to 1st place, and 0 corresponds to the last place in the match. It is calculated off of maxPlace, not numGroups, so it is possible to have missing chunks in a match.

In total there are 29 attributes and 4446966 Records in the dataset. Each row corresponds to a single player and his performance across the 29 parameters. The following 29 attributes were used to predict "winPlacePerc" which is a percentile winning placement, where 1 corresponds to 1st place, and 0 corresponds to last place in the

match. It is calculated off of maxPlace, not numGroups, so it is possible to have missing chunks in a match.

Data Preparation and Exploratory Analysis

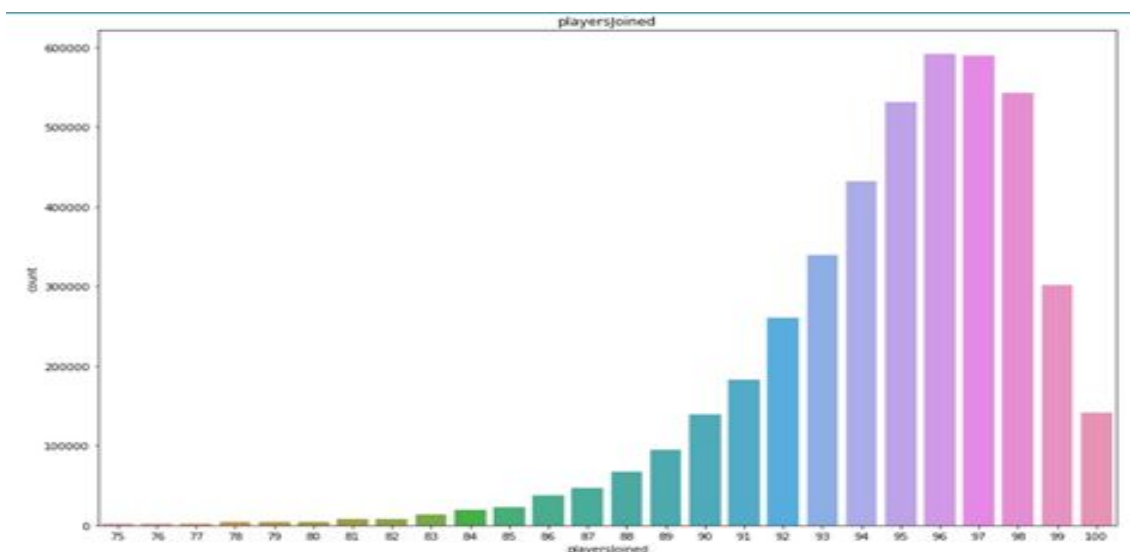
The first step in data preparation was to spot the rows with missing values. Only one row in the training data (2744604) was found with incomplete values and it was eliminated.

```
#To check for missing values
train[train.isnull().any(axis=1)]
```

rideDistance	roadKills	swimDistance	teamKills	vehicleDestroys	walkDistance	weaponsAcquired	winPoints	winPlacePerc
0.0	0	0.0	0	0	0.0	0	0	NaN

Additionally, we also found that there were matches containing only 1 or no players . These records were also removed.

The average number of players who join matches were then explored. This data was important as the total number of players joined would have a significant impact on various other significant parameters like Kills, damageDealt, maxPlace, and matchDuration. For instance if the number of players are less for a match the kills, damage, duration and ranking would be impacted accordingly. The graph showing the number of players joining vs the no. of matches can be seen below.



The results of this analysis showed that the most common number of players joining were in the range of 94-99. The parameters Kills, damageDealt, maxPlace, and matchDuration were accordingly normalized based on the players joined in their corresponding matches.

In the next part of data exploration, the focus was to identify cheaters in matches who had a significant number of kills while not even moving an inch. This was identified based on three parameters, namely walk distance, ride distance, and swim distance.

```
display(train[train['killsWithoutMoving'] == True].shape)
train[train['killsWithoutMoving'] == True].head(10)
```

(1535, 31)

ce	teamKills	vehicleDestroys	walkDistance	weaponsAcquired	winPoints	winPlacePerc	totalDistance	killsWithoutMoving
0	0	0	0.0	8	0	0.8571	0.0	True
0	0	0	0.0	22	0	0.6000	0.0	True
0	0	0	0.0	13	0	0.8947	0.0	True
0	0	0	0.0	7	1500	0.0000	0.0	True
0	0	0	0.0	10	0	0.3000	0.0	True
0	0	0	0.0	8	0	0.8000	0.0	True
0	0	0	0.0	8	0	0.6000	0.0	True

```
# Remove outliers
train.drop(train[train['killsWithoutMoving'] == True].index, inplace=True)
```

We also focused on finding anomalies in the data where there were unbelievable parameter values for features like walkDistance, rideDistance, swimDistance, and weaponsAcquired. In essence these are extreme outliers.

```
# Remove outliers(anamoly in walk distance)
train.drop(train[train['walkDistance'] >= 10000].index, inplace=True)
```

```
# Remove outliers(anamoly in ride distance)
train.drop(train[train['rideDistance'] >= 20000].index, inplace=True)
```

```
# Remove outliers(anamoly in swim distance)
train.drop(train[train['swimDistance'] >= 2000].index, inplace=True)
```

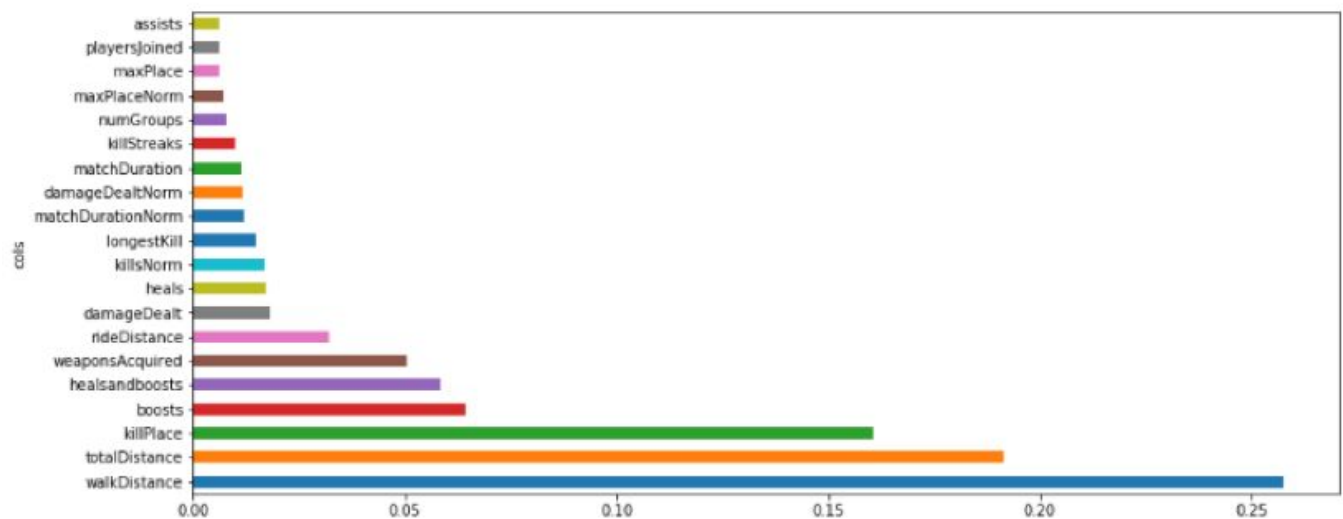
```
# Remove outliers(anamoly in weapons acquired)
train.drop(train[train['weaponsAcquired'] >= 80].index, inplace=True)
```

Feature Engineering

In order to improve accuracy and to achieve better prediction results it is highly necessary to create new features from existing data. Feature engineering therefore is one of the most important tasks of Machine Learning.

This step involved creation of features based on most contributing parameters towards our target variable winPlacePerc.

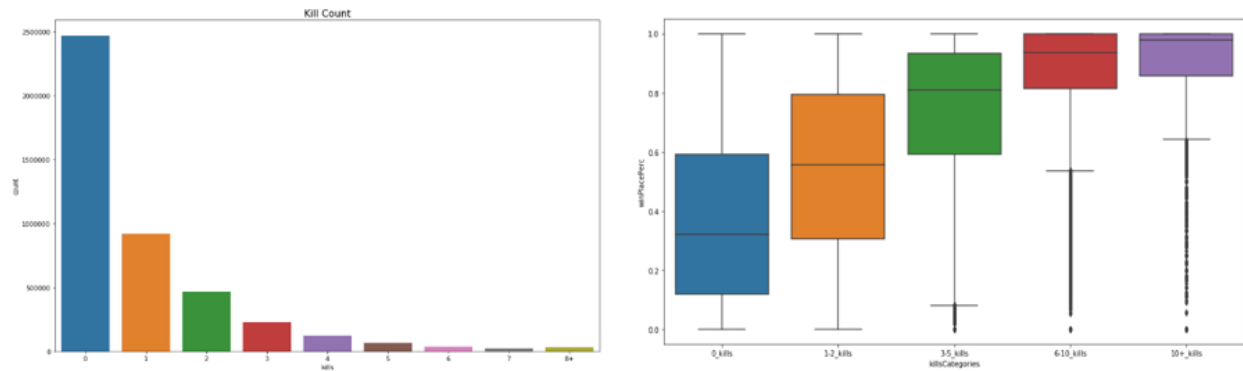
The following image shows the most how much each predictor was responsible for the winPlacePerc:



PUBG finish placement

It can be seen from the above figure that walkDistance, killPlace, weapons, boosts, damage, heals etc, were the main contributors to the main prediction.

Additional analysis into the kills parameter showed that a player was likely to be placed near to 1 when the number of kills were 10 or more.



The above findings was further supported by playing the game hands on. In a particular game our player's group achieved the 1st place. The final screenshot after the game, shows the highlights of the players key performance metrics like kills, heals, and damage etc.



From all our aforementioned findings, to assist this model we created featured attributes namely headshot rate, heals and boost, total_distance, features based on group ID and match ID. Some of the most important synthesised features are explained below

- **Headshot rate:** This parameter was found by dividing the number of headshot kills to the total number of kills. Headshots kill the enemy with just one bullet shot and increases chances of getting resource items and eventually additional kills in the game
- **Heals and Boosts:** Heals and boosts were two different attributes in our database which can be combined into one main category called Heals and Boosts. They in effect increase health and vitality and help a player last longer in a match.
- **Total Distance:** The game progresses in a way such that the map reduces in size between time intervals. It is necessary therefore for players to keep moving to the focus area of the map to stay in the game. The distance a player travels is therefore vital to his survival and victory. The feature total_distance was created by combining walkDistance, rideDistance and swimDistance.
- **Kill Streak rate:** This is the number which would indicate the rate at which a player killed a set of players in a short time frame. This indicates the players skill to act fast and face multiple enemies.
- **Distance over kills:** walk distance per Kills achieved shows the average distance a player travels to confront and attack one player.
- **Distance over weapons:** This feature shows the average distance a player had to traverse for acquiring a weapon.
- **Distance over heals:** A player gradually loses a significant amount of health over the course of the game. To regain it and survive in the game a player regularly uses boost items. The 'distance_over_heals' shows the average distance at which a person used a boost item.

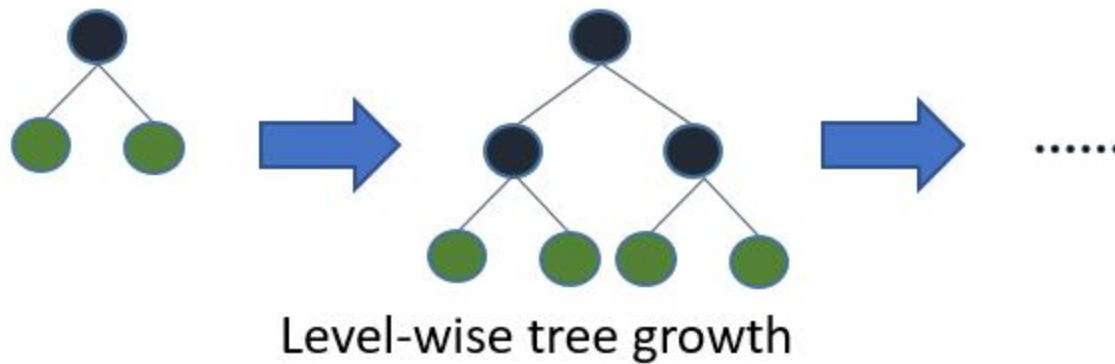
2.1 MODEL DESCRIPTION

The model of choice for our dataset is light Gradient Boosting Machine (GBM). Our choice of model has been primarily determined by the Keras densely connected neural network model that could not improve our prediction metric - mean absolute error (MAE). Light GBM is a high-performance gradient boosting framework based on classical decision trees. The real world experiments indicate that light GBM can speed up the training process by up to 7 times keeping the same accuracy compared to XG Boost, another gradient boosting algorithm that has been known among machine learning enthusiasts for a comparatively longer time (explained.ai, 2018). Origins of gradient boosting machines date back to 1999 when a Stanford University professor Jerome Friedman documented it in his research paper¹. Since then, different types of GBM has been implemented and the first notable implementation is XG Boost that gained wide popularity among competitions in Kaggle (B. Gorman, 2017). Light GBM is the next development in GBM implementation with high efficiency and scalability that was developed by Microsoft in late 2016 (Microsoft, 2016). Light GBM is slowly started gaining popularity in machine learning community because of its comparable prediction accuracy and up to 2-7 times speed gains compared to XG Boost (lightgbm experiment, 2018). The list of machine learning challenge winning solutions using light GBM is growing (github.com, 2018).

Considering the model's efficiency and speed, we decided to try Light GBM in our dataset. Since the model requires and mostly relies on the correct parameter tuning, we were able to get better results after a few runs and improvements led to an improvement from 0.056 to 0.0204 in mean absolute error. This also improved our in leaderboard as well. With Keras based NN, we were at around 600th position in the leaderboard. After implementing our light GBM model, we were able to take a spot in 102nd place in the leaderboard.

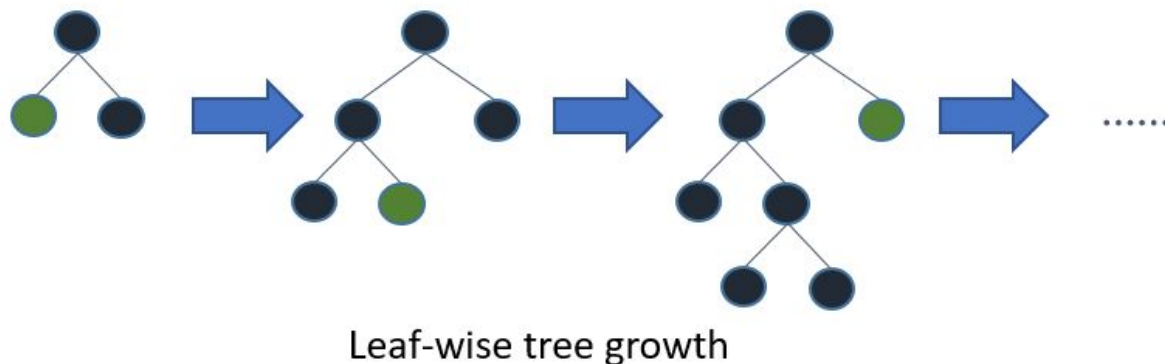
Light GBM model splits the tree leaf wise instead of splitting trees depth wise or level wise as implemented in other boosting algorithms. XG boost uses level wise tree growth approach as illustrated below.

¹ Paper: <https://statweb.stanford.edu/~jhf/ftp/trebst.pdf>



(source: analyticsvidhya.com, 2017)

In comparison, light GBM implements leaf wise tree growth as shown below.



(source: analyticsvidhya.com, 2017)

Since leaf wise splits result in higher complexity of the model, it can lead to overfitting and limiting maximum depth of splits can solve this overfitting issue.

Light GBM has the following advantages over other GBM:

- It uses histogram based algorithm meaning it converts continuous features into buckets of bins. Thus, further training is done using these bins instead of continuous values. This increases the speed of the training process.
- Memory usage is low. Once continuous features are converted into bins, there is no need for these continuous features. Less memory is required to store the bins.
- because light GBM implements leaf wise growth, big dataset and parallel learning are also supported.

2.2 ALTERNATIVE MODEL ANALYSIS

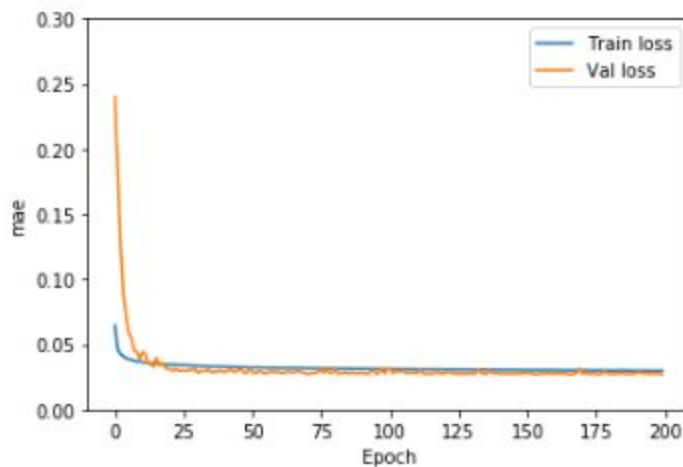
An alternative model is a model with Keras densely connected neural networks implementation. Neural networks are proven to fit into most of the problem and give very accurate results.

Model with 30% dropout.

```
# model construction: forming the network layers using keras
model = keras.Sequential([
    keras.layers.Dense(2048, kernel_initializer='he_normal', activation=tf.nn.relu, input_shape=(X_tr.shape[1],)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.3), # dropout for reducing the overfitting problem
    keras.layers.Dense(1024, kernel_initializer='he_normal', activation=tf.nn.relu), # 2nd hidden layer
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.3),
    keras.layers.Dense(512, kernel_initializer='he_normal', activation=tf.nn.relu), # 3rd hidden layer
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.3),
    keras.layers.Dense(256, kernel_initializer='he_normal', activation=tf.nn.relu),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.3),
    keras.layers.Dense(64, kernel_initializer='he_normal', activation=tf.nn.relu),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.3),
    keras.layers.Dense(1, kernel_initializer='normal', activation='sigmoid')]) # output layer

model.compile(loss='mse', #this loss method is useful for numeric prediction
              optimizer=tf.train.AdamOptimizer(learning_rate=0.001), metrics=['mae'])
model.summary()
```

Above model gave us following result:



Which indicates that validation loss converges after a few epochs and reaches to 0.0315 as the minimum error:

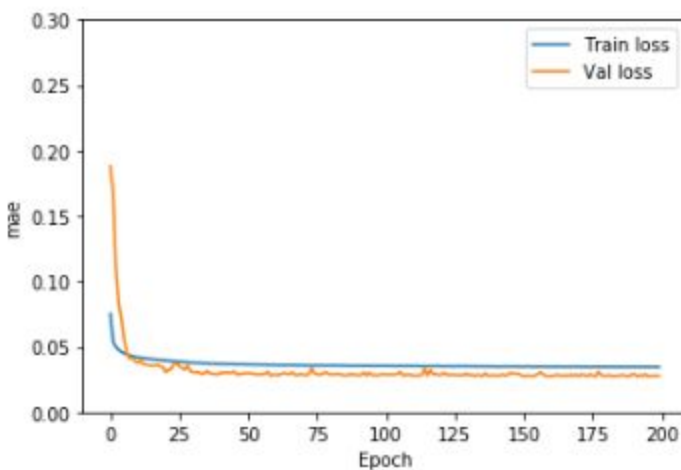
```
Epoch 73/200
1621395/1621395 [=====] - 42s 26us/step - loss: 0.0019 - mean_absolute_error: 0.0315 - val_loss: 0.0017 - val_mean_absolute_error: 0.0281
Epoch 74/200
655360/1621395 [=====>.....] - ETA: 23s - loss: 0.0019 - mean_absolute_error: 0.0315
```

Model with 0.5 as dropout.

```
# model construction: forming the network layers using keras
model = keras.Sequential([
    keras.layers.Dense(2048, kernel_initializer='he_normal', activation=tf.nn.relu, input_shape=(X_tr.shape[1],)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5), # dropout for reducing the overfitting problem
    keras.layers.Dense(1024, kernel_initializer='he_normal', activation=tf.nn.relu), # 2nd hidden layer
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(512, kernel_initializer='he_normal', activation=tf.nn.relu), # 3rd hidden layer
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(256, kernel_initializer='he_normal', activation=tf.nn.relu),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, kernel_initializer='he_normal', activation=tf.nn.relu),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1, kernel_initializer='normal', activation='sigmoid')]) # output layer

model.compile(loss='mse', #this loss method is useful for numeric prediction
              optimizer=tf.train.AdamOptimizer(learning_rate=0.001), metrics=['mae'])
model.summary()
```

Changing the dropout value to 0.5 , helped us to improve error by a significant amount shown below:



```
Epoch 199/200
1621395/1621395 [=====] - 40s 25us/step - loss: 0.0022 - mean_absolute_error: 0.0345
- val_loss: 0.0017 - val_mean_absolute_error: 0.0281
Epoch 200/200
1621395/1621395 [=====] - 40s 25us/step - loss: 0.0022 - mean_absolute_error: 0.0344
- val_loss: 0.0017 - val_mean_absolute_error: 0.0277
```

Thus we reached to mae value to 0.0277 and improved our rank further by 50 ranks.

We continued changing other hyper parameters such as activation functions and without batch normalization but were not able to get better results, thus we moved on to another model as described earlier - light GBM.

2.3 MODEL OPTIMIZATION

Running the model without any hyper-parameter tuning is only a starting point. When we look at the leaderboard in Kaggle, we observe that top competitors have a mean absolute error of around 0.02 for predictions. Our model scored MAE of 0.07 which landed our team in 600th position in the leaderboard. Of course, our aim is to land in the top 10 of the leaderboard. For these purposes, we have to fine-tune the hyperparameters of our light GBM.

One of the ways to improve the prediction accuracy of the model is to manually change the hyper-parameters one by one and train the model to check each model's mean absolute error. If we have the following set of parameters to tune, we will need to check $3 \times 4 = 12$ different models.

'learning_rate': [0.03, 0.04, 0.05],

'num_leaves': [33, 36, 39, 50]

At first glance, this manual parameter search technique seems manageable. However, as the parameter space gets bigger, manual parameter searching becomes exhaustive. For this reason, we are going to implement the GridSearchCV function that searches for the minimum mean absolute error within the specified grid of parameters. In order to verify if each of the models is performing better than the other, GridSearch also performs 3 fold cross-validation on the dataset (scikit-learn, 2018). This means that dataset is divided into 3 parts and while the first part is kept as a test, the rest is used as training. This is the first fold. Then the first and third parts are used for training while the second part of the dataset is used for testing. Third fold holds the last part of the dataset for testing while training with the first and second part of the dataset. There are total 3 runs are performed then, the average of their MAEs are returned to GridSearchCV. This 3 fold cross-validation verifies that there is no overfitting in the model and returned MAE is a more reliable indicator of the model's accuracy in comparison to a single model that performs predictions for 30% dataset instead of 100% dataset. One pitfall of such method is that it requires three times more resources (RAM if run in parallel) or longer run time compared to one training set. Since Kaggle offers 16 gigabytes of RAM with 4 CPU cores and one K80 GPU, we need to perform a random split on the dataset to prevent crashes of Jupyter Notebook. Our train dataset takes 4 GB of RAM space. If we launch 4 parallel instances of GridSearchCV, the process would be over 16 GB with system libraries. Therefore, we take only 40% of the dataset for GridSearchCV:

```
_, X_test, _, y_test = train_test_split(x_train, y, test_size=0.4, random_state=46)
```

Does this reliably represent the entire dataset if we want to check the overall performance of the model? Considering the data set is not stratified, random sampling from the dataset should give us the same MAE as it is for the whole dataset (StackExchange, 2012).

GridSearchCV accepts a set of parameters as follows:

```
gridParams = {  
    'learning_rate': [0.03, 0.04, 0.05],  
    'num_leaves': [33, 36, 39, 50] }
```

Why these two parameters? The official documentation of lightGBM states small learning rate and a large number of leaves increase the prediction accuracy of the model. If we do not see any improvement, we can specify another set of parameters depending on whether the grid search identified lower/higher numbers return better predictions.

We could increase the parameter space, but time constraint of 6 hours paired with only CPU computation for lightGBM (GPU is not supported in Kaggle version of lightGBM) we limited our grid parameters.

Before running grid search, we define our model:

```
mdl = lgb.LGBMRegressor(metric='mae',  
    objective="regression",  
    n_estimators=20000,  
    bagging_fraction=0.7,  
    bagging_seed=0,  
    num_threads=3,  
    colsample_bytree=0.7,  
    num_boost_round=3000)
```

Above parameters ensure that model only changes its learning rate and num_leaves, not other parameters.

Instantiate the grid search:

```
grid = GridSearchCV(mdl, gridParams, verbose=5, cv=None,  
    scoring='neg_mean_absolute_error')
```

After instantiating the grid search, we can run it by calling a fit method:

```
grid.fit(X_test, y_test)
```

To obtain the best parameters after fit, we simply call:

```
grid.best_params_
```

The only limitation of this exhaustive search is that, lightGBM may not return the lowest MAE for num_boost_round=3000. In fact, MAE may lower after 30000 iterations of boosting, which is prohibitive to perform under current hardware limitations.

Nonetheless, this grid search should give us a hint on which direction the model parameters should be tuned.

The final results were `learning_rate=0.04` and `num_leaves=36`. Of course, we need to verify by training the model with these parameters. So, we run two models with a learning rate of 0.04 and 0.03 and the number of boosting rounds were set to 50000 for both. Why two different models instead of the best parameters from grid search? As mentioned before, with more boosting rounds one model can converge better than the other at the ending few thousand rounds.

2.4 IMPLEMENTATION (PREDICTION, OPERATIONALIZATION)

After feature engineering of the dataset and tuning the hyperparameters of the model, we can start to train it. One important fact to note is that gradient boosting algorithms do not require normalization of datasets. Before realizing this fact, we spent around a week trying to tune the model and were not able to break into top 100. However, training the model without normalizing the data gave us immediate boost in prediction accuracy. Therefore, we kept the dataset unchanged. Before training the model, there are few parameters that one should be aware of:

Objective: regression. Since we are doing numerical prediction, model's objective is regression. It has also binary and multiclass options.

Num_iterations: (also called 'num_boost_round') number of boosting iterations to be performed. By default the parameter is set to default=100. However, with our huge dataset, the model needs to train for longer iterations. We set this parameter to 50000. This is enough to finish just before hitting the time limit in Kaggle for python scripts.

num_leaves : number of leaves in one tree ; default is 31. This is one of the most important parameters of lightGBM. In optimization part we found 35-36 to be a good candidate for better prediction accuracy, so we set this parameter to 35.

device : default = cpu. We have a GPU option as well. However, the python that was running on Kaggle platform did not have the latest version of light GBM that supported GPU, so we were limited to only CPU, which was one of the main bottlenecks limiting our boost rounds to 50000.

feature_fraction: (also called 'colsample_bytree') set to 0.7 for our model. It specifies the fraction of features to be taken for each iteration. This is helpful in dealing with overfitting.

bagging_fraction: set to 0.7. This specifies the fraction of data to be used for each iteration and is generally used to speed up the training and avoid overfitting.

num_threads: number of CPU cores to be used for light GBM. 4 CPU cores were available and so, it was set to 4. One fact to note, it does not speed up if cores are not physical (e.g. hyper threads for Intel processors). Thus, only actual physical cores should be specified. Otherwise, set to system detected maximum.

Early_stopping: 320. If the model's MAE does not improve on validation dataset for 320 rounds, the model will stop training. This is also helpful in preventing overfitting. After specifying the parameters, we need to specify training and validation dataset:

```
lgtrain = lgb.Dataset(train_X, label=train_y)
```

```
lgval = lgb.Dataset(val_X, label=val_y)
```

This type of data representation will convert continuous features into bins, so the dataset is memory efficient (lightGBM, 2018).

Now the model can be trained:

```
model = lgb.train(params, lgtrain, valid_sets=[lgval], verbose_eval=1000)
```

Here, params has a list of parameters specified above in a dictionary format. Then training set, validation sets are specified. Verbose_eval=1000 makes sure that metrics of training and validation are shown as a message for each 1000 iterations.









After the training is done, which takes 7 hours in 4 core CPU of Kaggle, we can now do model predictions:

```
pred_test_y = model.predict(x_test, num_iteration=model.best_iteration)
```

For prediction, we do not need to create special binned dataset for test. However, we need to make sure to apply all the feature engineering steps to test dataset as well.

Otherwise, the model will not be able to do predictions. Additionally, we can choose which iteration of the model to use with num_iteration. In our case, we use model.best_iteration.

There is a trick that should be applied during the submission of the test dataset. This trick is related to feature 'MaxPlace'. When 'MaxPlace' is 0 or 1, then predictions should also be 0 or 1, respectively. Therefore, the submission section of the script has few more lines of code. The best MAE so far for our model is 0.0203, which landed us in top 5% (39th place out of 1488 competitors) in the time of writing this project².

32	▲ 36	Koi King		0.0203	17	3d
33	▼ 15	Harshit Sheoran		0.0203	69	3d
34	▲ 87	Khanh Tran Cong Phu		0.0203	15	3d
35	▼ 16	Meomeo		0.0203	5	10d
36	▼ 14	powerboy		0.0203	60	3d
37	new	xiaohao1234		0.0203	5	3d
38	▼ 17	Happy Wind Man		0.0204	34	17d
39	new	PUBG project		0.0204	28	10h
Your Best Entry ▲						

² New better entries from competitors may decrease our place. For up-to-date leaderboard, check: kaggle.com/c/pubg-finish-placement-prediction/leaderboard

2.5 Potential Improvements, Challenges

Since this was our first Kaggle Challenge, we spent our most of the time building and testing the model. By reading top 1% kagglers, we learned that trying multiple models at once and choosing the best model after that for parameter tuning is the best strategy. It may be the case that light GBM is not the best suitable model to get to top 10. For this reason, it is very likely that we need to train densely connected neural networks as well with more powerful hardware. Thus, we could improve our position in the leaderboard by doing these following steps:

- Use AWS Cloud Jupyter Notebook with at least 32GB RAM and NVidia K80. It is possible that 16GB RAM is enough. However, for multithreaded GridSearchCV, we definitely need at 32 GB of RAM. we must make sure that libraries are GPU enabled. Otherwise, CPU calculations take longer time.
- Use Keras NN. There are other NN enabled libraries available for Python such as Pytorch or Ultimate MLP. By using Keras built NN we ensure that simple NN layers without any additional code can do better job at predicting than light GBM.
- Use GridSearchCV with 5 fold cross-validation on the entire dataset to search in a wider range of parameter. In Kaggle, we are limited by time, memory and some libraries that do not support GPU. These restrictions will be removed if we move to AWS and GridSearch with wider range of parameters ensures that we obtain the best model for our predictions.

After these improvement, our possible outcomes include being in top 10 of the leaderboard.

Bibliography

Explained.ai, 2017, source: <https://explained.ai/gradient-boosting/index.html>

Ben Gorman, 2017, source:

<http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/>

Light GBM, 2016, source: <https://www.microsoft.com/en-us/research/project/lightgbm/>

Light GBM experiment, 2018, source:

<https://lightgbm.readthedocs.io/en/latest/Experiments.html>

Github.com, 2018, source:

<https://github.com/Microsoft/LightGBM/blob/master/examples/README.md#machine-learning-challenge-winning-solutions>

'Light GBM vs XGBOOST?', analyticsvidhya, 2017, source:

<https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/>

'GridSearchCV', Scikit-learn, 2018, source:

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV

Stackexchange, 2012, source:

<https://stats.stackexchange.com/questions/34939/k-fold-cross-validation-strategy-for-large-data-set-in-statistical-learning>

lightGBM, 2018, source:

<https://lightgbm.readthedocs.io/en/latest/Python-Intro.html>