PUBG Finish Placement Prediction

EE 660

Project Type: Collaborative (Kaggle) – Individual

Muthulakshmi Chandrasekaran

muthulac@usc.edu

USC ID – 4486180242

December 6, 2018

Table of Contents

1. ABSTRACT

This project models a good machine learning model for the ongoing Kaggle competition PUBG(Players Unknown Battle Ground) Finish Placement Prediction. A PUBG Game starts with 100 players, and the goal of the game is to eliminate all other players, until a single player remains, while there is a continual shrinkage of the playing area. The objective of the project is to predict the finishing placement (or the winning percentage) of the given players using the provided very large data. The problem is non – trivial because of the large number of points on the training and test set. This is a Regression problem, wherein the baseline model is chosen to be simple Linear Regression. To solve complex regression models, Random Forest Regressor and other Gradient Boosting Trees(Light GBM, XGBoost and CAT Regressor) are implemented. It is observed that Light GBM outperforms all other models, since the number of data points are very large. Submission to the Kaggle gives a public score of 0.0247. Further improvements have also been suggested.

2. INTRODUCTION

2.1 Problem Type, Statement and Goals

The dataset is based on the trending game – Player's Unknown Battle Ground(PUBG). In a PUBG Game, each match starts with 100 players onto an island. The goal of the game is to eliminate all players, as the game progresses, until only one player is left, while the playing area continually shrinks. The goal is to model the best strategy to win in a match – by predicting the placement of players when the game ends, or the winning percentage of each player, using the given data. The placement of players is on a scale of 0 to 1, which ranks the players based on the given model.

This is a Regression problem, since the prediction is a real – value in the range [0,1]. The variable that is predicted is the 'winPlacePerc', that ranks the players. This problem is particularly interesting because this is based on a real – time data, and is a currently on going Kaggle Competition. The final scores are based on the Mean Absolute Error. This problem is non – trivial because of the following reasons :

- Dataset contains statistics of real – time data, which are used in winning the games
- Dataset is very large – it has about 4.44 million data points on the training set, and about 1.93 million points on the test set.
- Data visualization part is mandatory and challenging due to very extensive and diverse features.
- Though the number of features are related, most of the features are highly correlated with each other, that help in predicting the winning percentage.

2.2 Literature Review

None.

2.3 Prior and Related Work

This project is an ongoing Kaggle competition. Hence there is no prior or related work.

2.4 Overview of Approach

The goal is to find the winning percentage of the players in the range [0,1] – the output is a real value between 0 and 1. Hence, this is a regression problem. The first try is an implementation of a simple baseline classifier – Linear Regression Model. Then improved models like Random Forests and

Gradient Boosting Algorithms – like lightGBM and XG Boost are tried. Since the baseline model is chosen to be a simple Linear Regression model, the which apparently performs very poor without any optimizations. The model is best chosen based on the accuracy it gives on the initial test set, which is split from the training set. For choosing the best model, R2 Score and Mean Absolute Error are chosen as the performance metrics. Model selection is interesting here, because the labels of the test set are actually unknown to us, like any other real – time machine learning problem, and they are to be predicted. The implementation is done in Python.

3. IMPLEMENTATION

Prediction of winning percentage of the players is achieved using the above approach. This is coded in Python, the details of implementation are given below.

Coding Language	Libraries Used
Python	Numpy, Sci-kit learn, catboost, xgboost,lightgbm matplotlib , seaborn (for plots)

3.1 Data Set

This is a Kaggle competition, and hence the dataset is provided in the competition.

Table 1 gives information about the input files.

Table 1 Information about the data files

	Train Data	Test Data
Number of data points	4446966	1934174
Number of columns	29	28

There are 28 features in the input file, to predict one output variable $\boldsymbol{winPlacePerc}$**.**

The percentage of prediction is given under the label '$\boldsymbol{winPlacePerc}$', that is present in the training set, but not in the test set, that explains the difference in numbers.

Table 2 gives the explanation of what each column is. [Reference:6]

Table 2 Description of columns

S.No	Feature Name	Description of each feature
1	Id	A unique id to identify the entry
2	$groupId$	ID to identify a group in a match.
3	$matchId$	ID to identify a match.
4	$assists$	Count of enemies damaged by a player, but killed by team mates.
5	$boosts$	Number of boost items used.
6	$damageDealt$	Total damage dealt, excluding self – inflicted damage
7	$DBNOs$	Number of enemy players knocked.
8	$headshotKills$	Number of enemy players killed by headshots.
9	$heals$	Number of healing items used.
10	$killPlace$	Ranks of number of enemy players killed in a match.
11	$killPoints$	Ranking of a player based on the number of kills.
12	$killStreaks$	Max number of enemy players killed in a short time.
13	$kills$	Number of enemy players killed.
14	$longestKill$	Longest distance between player and player killed at time of death.
15	$matchDuration$	Duration of a match (in seconds)
16	$matchType$	Type of match being played.
17	$maxPlace$	Worst case placement of data in the match.
18	$numGriups$	Number of groups we have data for in the match.
19	$rankPoints$	The points based on a player's rank
20	$revives$	Number of times a player revived teammates.
21	$rideDistance$	Total distance traveled in vehicles(in meters).
22	$roadKills$	Number of kills while in a vehicle.
23	$swimDistance$	Total distance traveled by swimming (in meters).
24	$teamKills$	Number of times this player killed a teammate.
25	$vehicleDestroys$	Number of vehicles destroyed.
26	$walkDistance$	Total distance traveled on foot(meters)
27	$weaponsAcquired$	Number of weapons picked up.
28	$winPoints$	Win-based external ranking of player.

The last column in the training set is winPlacePerc, which is the target of prediction. This 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. The details of each feature are given in Table 3.

Table 3 Details of each feature and the target variable

S.No	Feature Name	Type of feature	Data type	Range of Values Min – Max		Number of unique values
1	***Id***	These three columns are not used as features, they are to group the statistics. They also aid in finding patterns among other features.				
2	***groupId***					
3	***matchId***					
4	***assists***	int64	Numerical	0	22	20
5	***boosts***	int64	Numerical	0	33	27
6	***damageDealt***	float64	Numerical	0.0	6616.0	29916
7	***DBNOs***	int64	Numerical	0	53	39
8	***headshotKills***	int64	Numerical	0	64	34
9	***heals***	int64	Numerical	0	80	63
10	***killPlace***	int64	Numerical	1	101	101
11	***killPoints***	int64	Numerical	0	2170	1707
12	***kills***	int64	Numerical	0	72	58
13	***killStreaks***	int64	Numerical	0	20	18
14	***longestKill***	float64	Numerical	0.0	1094.0	28284
15	***matchDuration***	int64	Numerical	133	2237	1266
16	***matchType***	object	Categorical	squad-fpp, duo, solo-fpp, squad,duo-fpp, solo, normal-quad-fpp, crashfpp, flaretpp, normal-solo-fpp, flarefpp, normal-duo-fpp, normal-duo, normal-squad,		16

				crashtpp, normal-solo		
17	***maxPlace***	int64	Numerical	1	100	99
18	***numGroups***	int64	Numerical	-1	5910	100
19	***rankPoints***	int64	Numerical	0	39	2262
20	***revives***	int64	Numerical	0	40710	25
21	***rideDistance***	float64	Numerical	0.0	18.0	33562
22	***roadKills***	int64	Numerical	0	3823	14
23	***swimDistance***	float64	Numerical	0.0	12.0	28345
24	***teamKills***	int64	Numerical	0	5	11
25	***vehicleDestroys***	int64	Numerical	0	25780	6
26	***walkDistance***	float64	Numerical	0.0	236.0	38599
27	***weaponsAcquired***	int64	Numerical	0	2013	97
28	***winPoints***	int64	Numerical	0	1	1447
29	***winPlacePerc***	float64	Numerical	0.0	22.0	3000
		This is the target to be predicted.				

3.2 Pre – Processing, Feature Extraction and Dimensionality Adjustment

Before pre – processing, the data is understood carefully. The algorithm is coded in Python.

3.2.1 Exploratory Data Analysis

The training data is first analysed to explore the relationships between the variables.

Checking for missing data points :

The initial version of the data given did not have any missing points. In the recent version of the dataset to be used for the competition, there are missing data points in the training set.

The initial size of the input sets are shown below.

```
Training Set Size :  (4446966, 29)
Test Set Size :  (1934174, 28)
```

The number of missing data points are found to be 1. This is a very small value compared to the number of data points $\left(\frac{1}{4446965} = 2.24872469\text{E} - 7\right)$. This is found at the row with $\boldsymbol{Id, groupId}$ **and** $\boldsymbol{matchId}$ as shown below.

	Id	groupId	matchId
2744604	f70c74418bb064	12dfbede33f92b	224a123c53e008

Hence this row can be deleted without any loss of data points. Hence, the size of the training data will become

```
Training Set Size :  (4446965, 29)
Test Set Size :  (1934174, 28)
```

These sets are further used for analysis.

In the training set, target variable is $\boldsymbol{winPlacePerc}$**,** and the columns $\boldsymbol{Id}$, $\boldsymbol{matchId}$ and $\boldsymbol{groupId}$ are not counted as features.

First, the target is checked for outliers, and Figure 1 explains it.

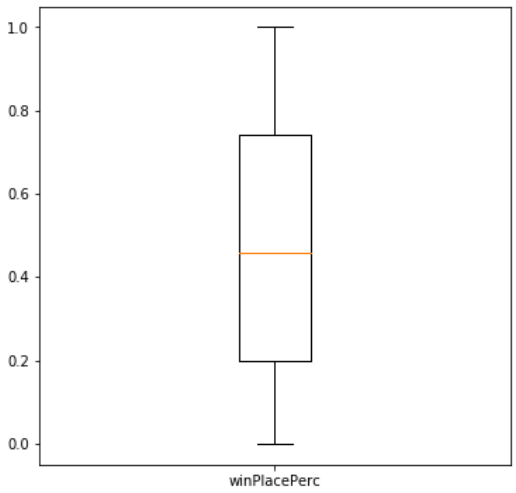

Figure 1 Checking for outliers in Target

It is found that there are no outliers, and hence there's no processing required.

The correlation between all the other features is obtained using heat map as shown in Figure 2.

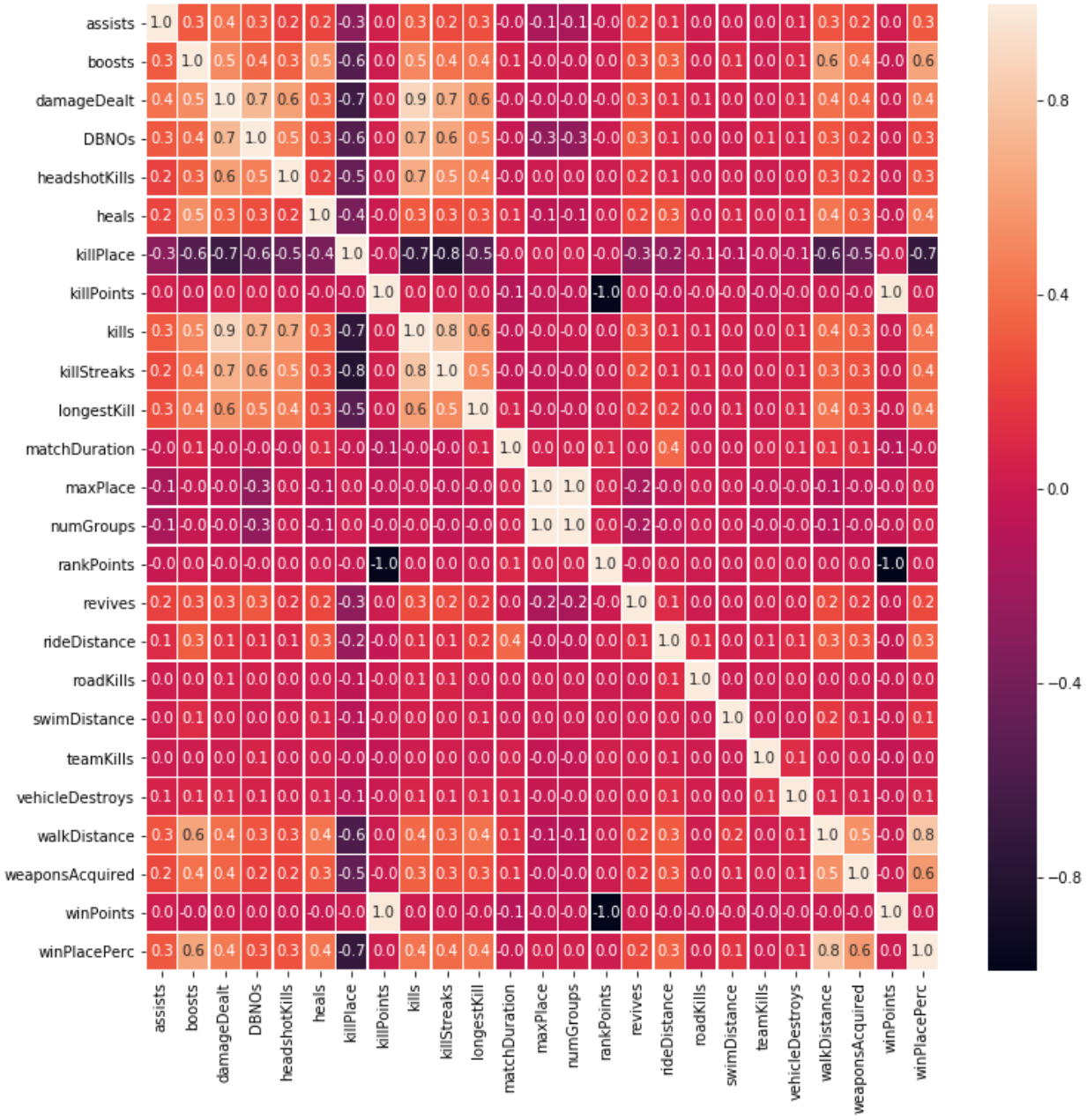

Figure 2 Correlation Map for all features

From the correlation map, it is observed that the feature $\boldsymbol{walkDistance}$ has highest correlation with $\boldsymbol{winPlacePerc}$, while $\boldsymbol{killPlace}$ has the least correlation, according to the heat map. Some features have zero correlation as well. It is also observed that the highest collated features to $\boldsymbol{winPlacePerc}$ are $\boldsymbol{walkDistance}, \boldsymbol{boosts}, \boldsymbol{weaponsAcquired}, \boldsymbol{damageDealt}$.

In general, the game revolves around killing and surviving. Hence, additional features that were added were based on these parameters. The following figures show the feature importance, as in the level of correlation of the corresponding variable to the target variable $\boldsymbol{winPlacePerc}$.

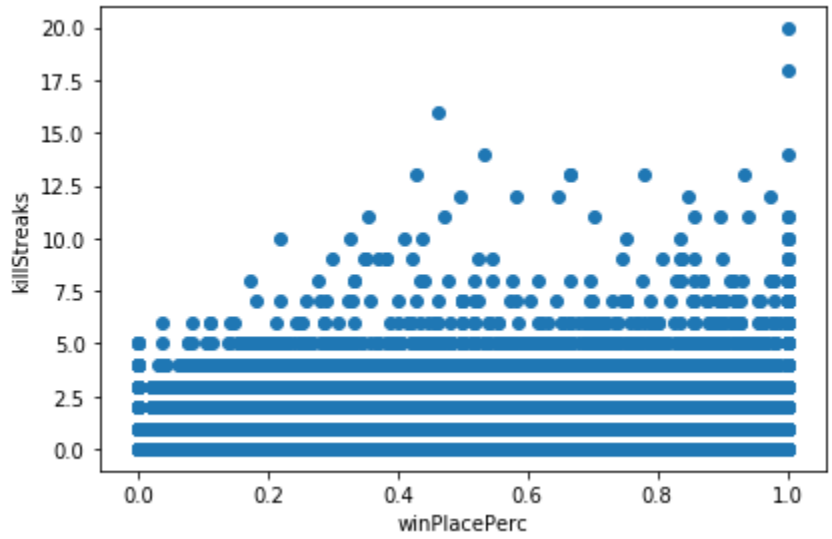

Figure 3(i) – Plot of killStreaks vs winPlacePerc, that shows that killStreaks are correlated. This can be observed from the heat map as well.

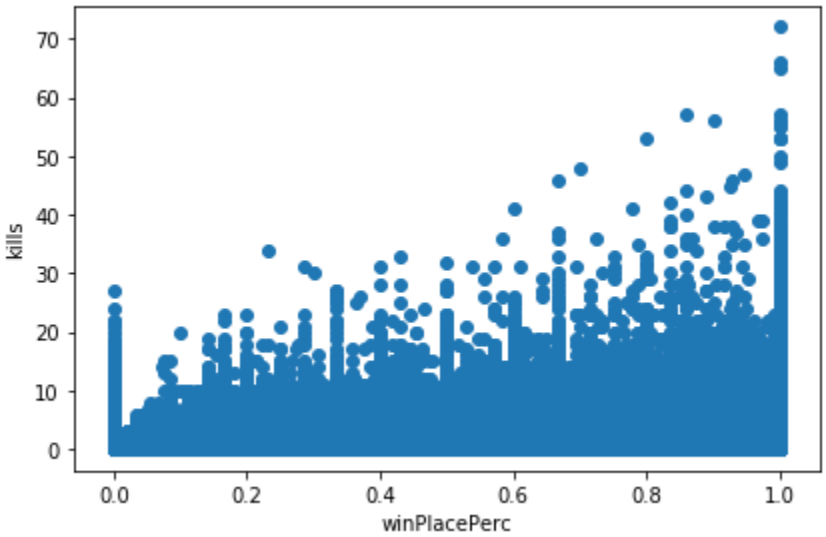

Figure 3(ii) – Plot of kills vs winPlacePerc, that shows that the distribution of the number of kills across different winning percentages. For a higher winning percentage, the number of kills should be more.

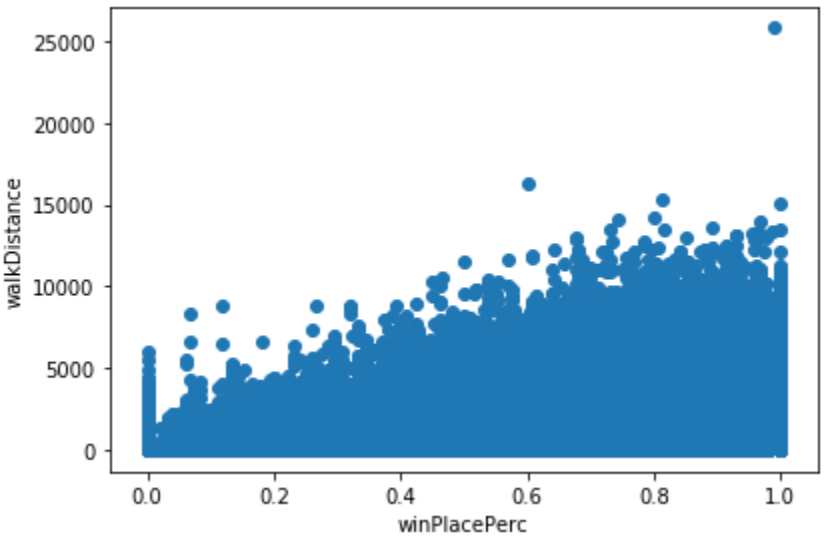

Figure 3(iii) – Plot of walkDistance vs winPlacePerc,showing this is not a so useful feature, but somewhat correlated with winPlacePerc.

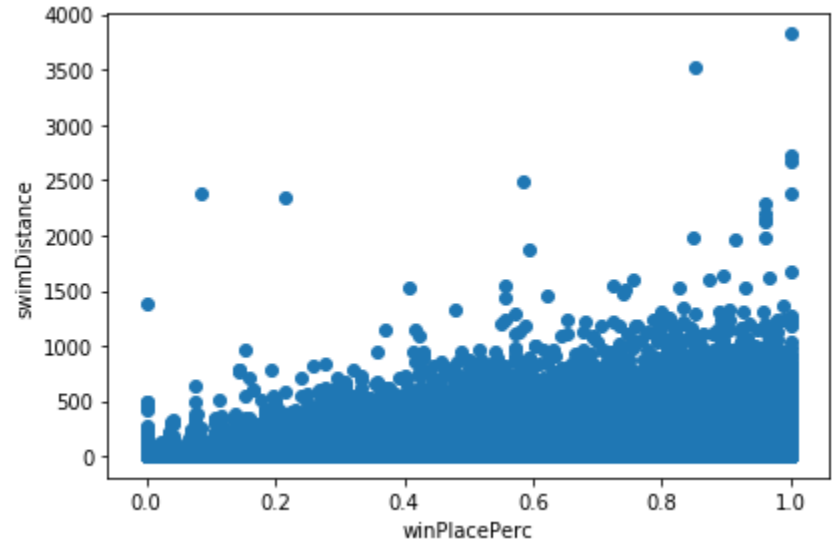

Figure 3(iv) – Plot of swimDistance against the winning percentage, that shows a very little correlation.

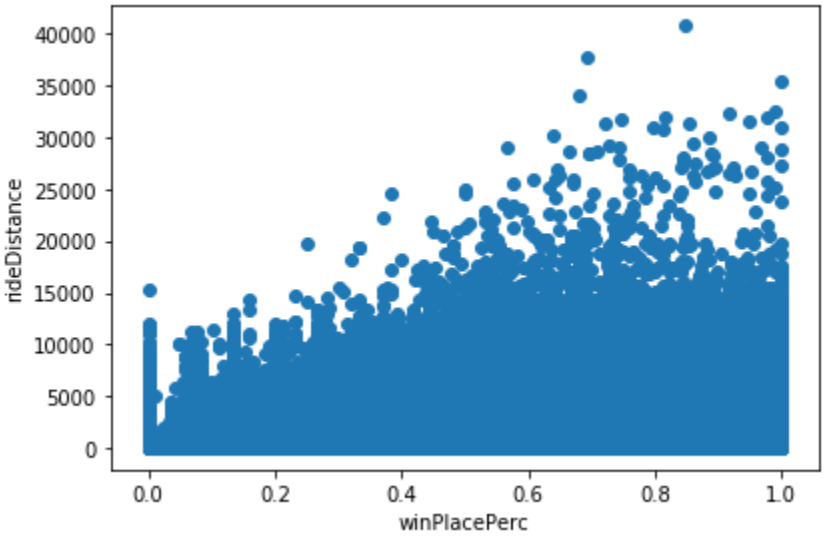

Figure 3(v) – Plot of rideDistance against the winning percentage, that shows a reasonable correlation for lower winning percentages, but very scattered at higher winning percentages. Hence this might be a useful feature.

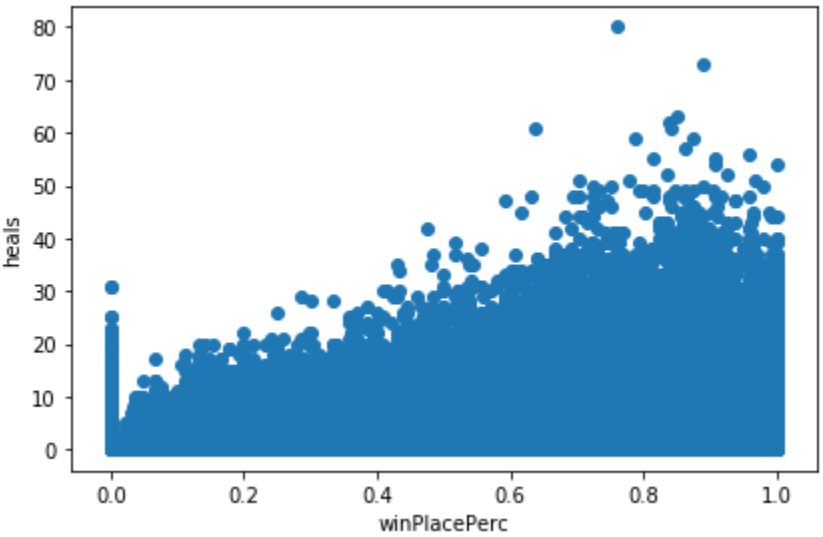

Figure 3(vi) – Plot of heals vs winPlacePerc, that shows that good correlation.

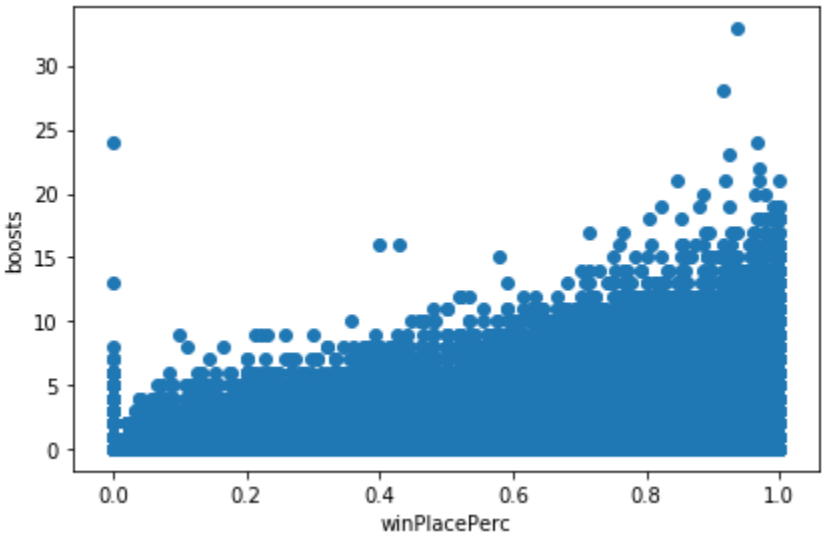

Figure 3(vii) – Plot of boosts vs winPlacePerc, that shows that good correlation.

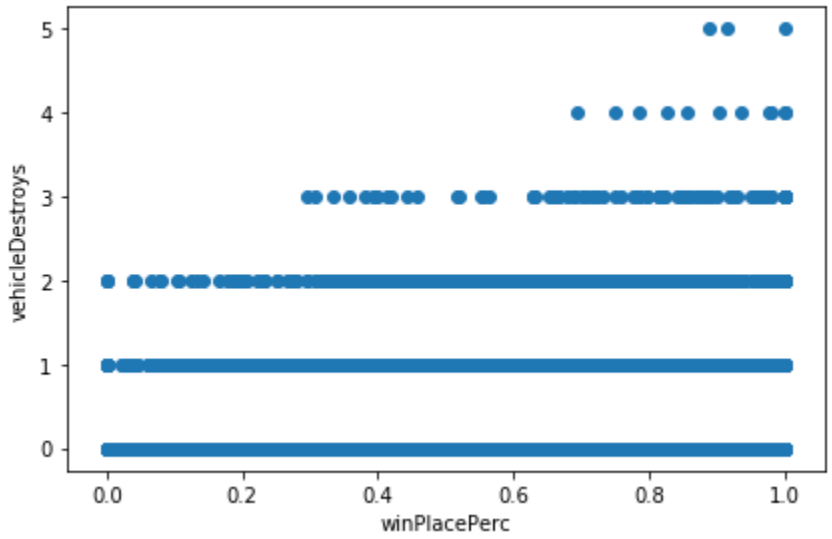

Figure 3(i) – Plot of vehicleDestroys vs winPlacePerc, that shows that this is a bad feature.

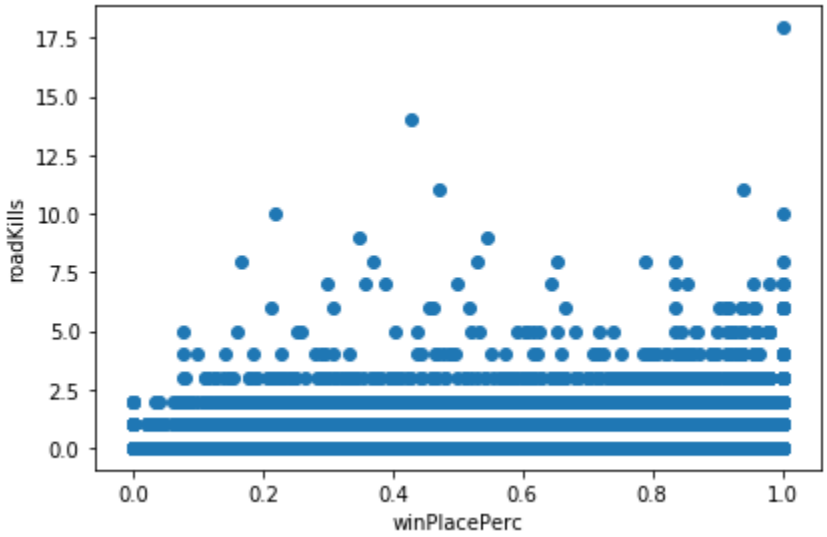

Figure 3(i) – Plot of roadKills vs winPlacePerc, that shows that this feature is not that important.

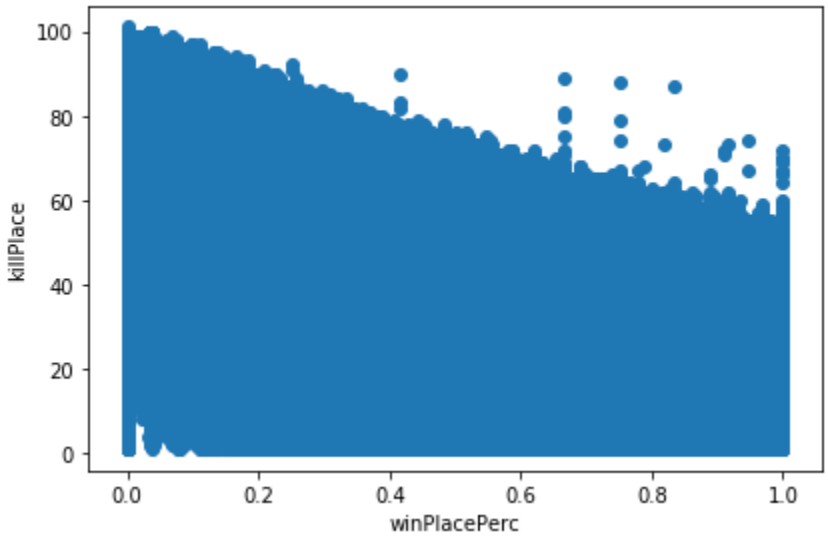

Figure 3(i) – Plot of killPlace vs winPlacePerc, that shows that winning percentage decreases as the killPlace value decreases.

This shows a highly negative correlation between the two variables.

The histogram of the target variable $\boldsymbol{winPlacePerc}$ is visualized as in Figure 4, that shows the distribution of values across the range. This shows that the values are distributed across all ranges.

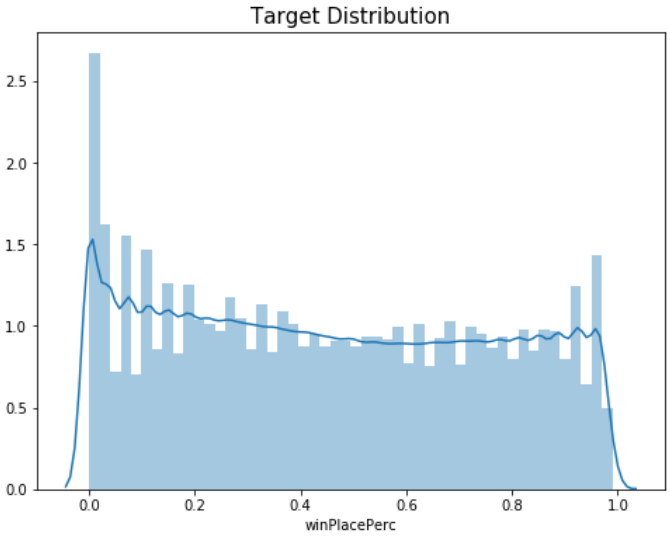

Figure 4 Distribution of the target variable

In addition to the actual features, a combination of features can be used. For example, the following figure 5 shows the relationship of heals and boosts across the target variable.

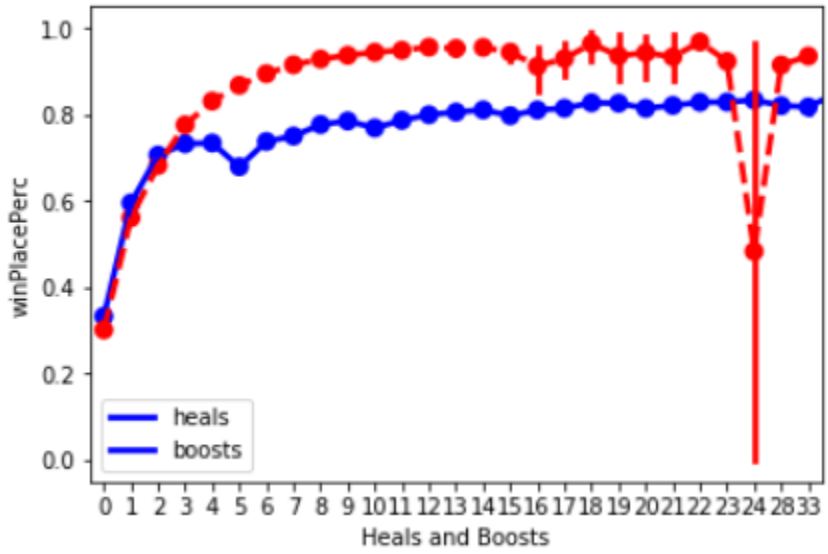

Figure 5 Heals and boosts across target variable.

In some cases, the correlation between two feature variables can also be observed. For example, in Figure 6(a), the relation between walkDistance and Heals is shown, Another example would be Figure 6(b), that shows the relation between kills and Damagedealt, since these are the features that are highly correlated with the target.

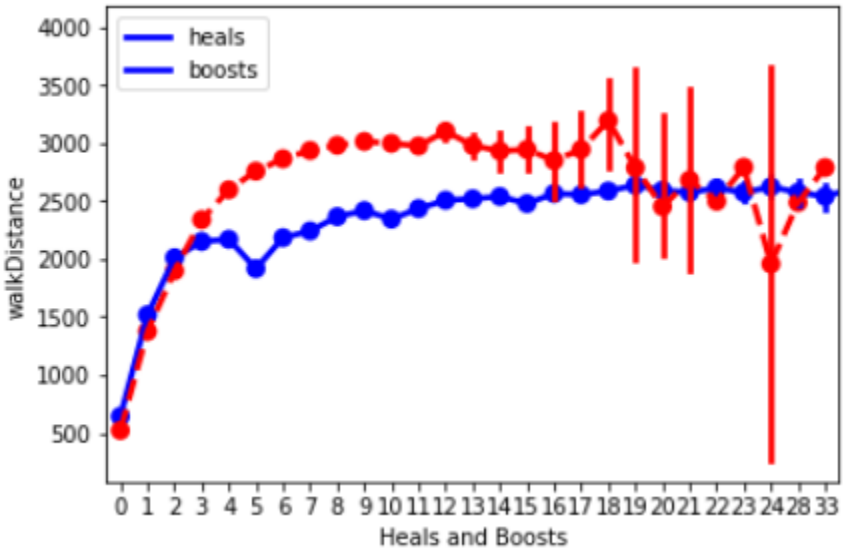

Figure 6(a) Relation between walkDistance and Heals

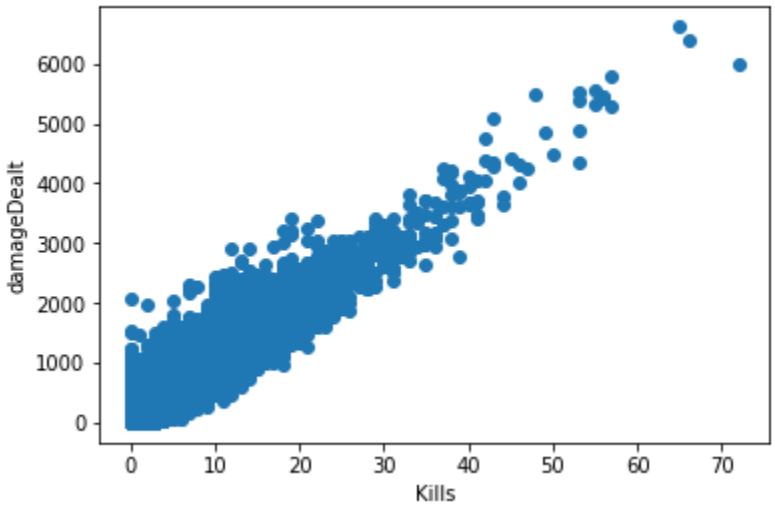

Figure 6(b) Relation between kills and DamageDealt

Through this analysis, it is observed that almost all variables are in a certain range only. Some values are highly correlated with the output target variable, which are observed through histogram plots and box plots. It is also observed that group id and match id are important in determining features relating to a certain group. Some extra features were added.

The number of each feature that affects the result is also found, to find the importance of each feature, as shown below.

```
Average no.of healing items used : 1.370
Average no.of kills : 0.925
Average no.of assists : 0.234
Average walking Distance : 1154.218
Average swim Distance : 4.509
Average Ride Distance : 606.116
Average no.of head shot kills : 0.227
Average damage dealt : 130.717
```

3.2.2 Pre – processing

Raw input data can have different range of values. Hence, all data are scaled to the same scale before further processing. Standard Scaler is used here. The following command is used to include this function.

```
from sklearn.preprocessing import StandardScaler
```

Standard Scaler removes the mean and normalizes to unit variance. Only the numerical features are standardized using Standard Scaler.

3.2.3 Feature Extraction

Based on EDA, relation between various features are observed. Some features were dropped, and some were added. The following features were added in addition to the actual features :

- Head Shot Rate, that gives the number of kills per headshotkills
- Kill Streak Rate, that gives the ratio of kills and headshot kills
- The total of swim, walk and ride distance
- The number of weapons acquired compared to total distance
- Total number of heals and boosts
- Total number of boosts or heals, and their sum, per walk distance
- Total number of kills and assists
- Mean, max and min of each feature, and their ranks according to groupId and matchId

Some features are added according to the group and match statistics. Some features like swimDistance, walkDistance, rideDistance, vehicleDestroys, roadKills and headshotKills were dropped, since they were already included as a single feature, and have less correlation.

3.2.4 Dimensionality Adjustment

The total number of useful features after all suitable pre – processing methods were found to be around 200. Since this is comparable(or vey less) to the number of data points, there is no dimensionality reduction needed.

3.3 Dataset Methodology

The training data and test data are given separately as input files.

The training data is divided into test_train and train_train sets, where the test_train set acts like a sample test set (with labels), when the actual test set is not known. This set is used only for evaluating the model, and calculating the final performance metrics.

The train_train set, that contains 85% of the actual training set, is split into pre – training set and training set, with pre – training set having 15% of the total number of points on the training set, randomly sampled.

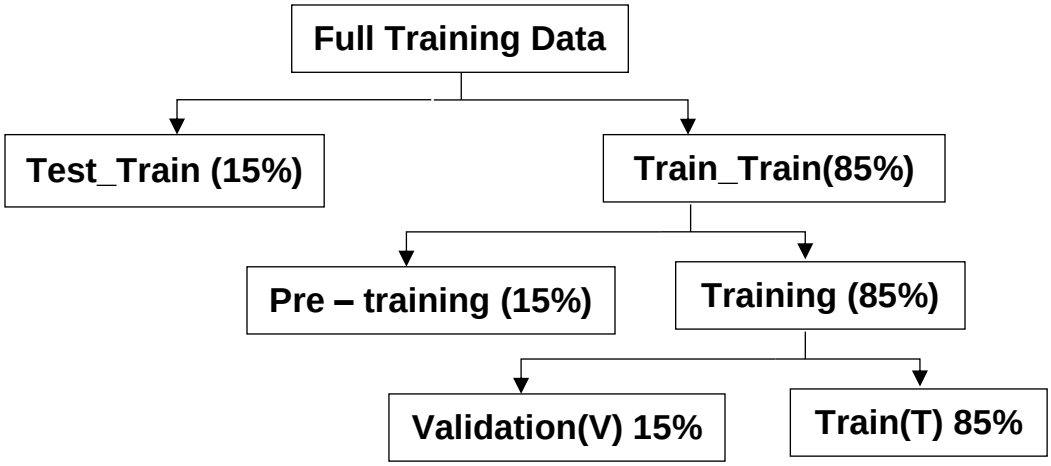

Figure 7 Dataset Methodology

In general, pre – training data is used to analyse the pre – processing to be done, extraction of features, and dimensionality analysis. Here the entire train_train set is used for exploratory data analysis, ti get better insights of the data.

The training set is further divided into validation set, and a set for training, wherein validation set is 15% of the training set. This methodology is followed only in certain cases, since the time constraints on the Kaggle platform was limited to 6 hours, and the dataset was large. This is given in Figure 7.

Cross – validation was used, which were helpful in some models, that are explained as the model is being explained. Wherever cross – validation was used, it was sequential.

The best – performing model is decided based on the performance of each of the model on the test_train set. The unknown test data(given data) was validated using the final model, and the winning percentage was predicted. This is explained further for each model chosen in the sections to follow.

3.4 Training Process

Once the datasets are properly divided, various models (or algorithms) are applied on the suitable dataset, to find the best – performing model. The following models are implemented.

i. Linear Regression
ii. Random Forest Regressor
iii. Gradient Boosting – Light GBM
iv. Gradient Boosting – XG Boost
v. CAT Regressor

Each of these models is implemented after pre – processing and feature engineering, on the training data, the dataset marked as T in Figure 7 in Section 3.3. The highlighted row in each model shows the optimal parameters for that model, chosen based on the lowest MAE.

3.4.1 Linear Regression

Since this is a regression model, and the goal is to predict the percentage winning of players, the simplest regression model is chosen as the baseline classifier. Linear Regression is implemented, after addition of extra features.

Linear Regression aims to model the relationship between two variables by fitting a straight line between the two variables. This method assumes that the two variables are linearly related. The equation of a linear regressor can be given as

$y = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$, where

w_0 is the intercept, y is the response, and $w_1, w_2, \ldots, w_n$ are the weights of the input variables $x_1, x_2, \ldots, x_n$.

The model is trained to find out the linear model that best fits the given data.

Ordinary Least Squares Regression is used here, since this is already available in the package sklearn. The output of a linear regression model is a linear line, with a constant slope.

The already available package in sklearn uses Ordinary Least Squares Linear Regression, which means the error measure is calculated based on ordinary least squares. The following scikit – learn function is used to implement Linear Regression.

```
class sklearn.linear_model. LinearRegression (fit_intercept=True,
normalize=False, copy_X=True, n_jobs=None)
```

3.4.2 Random Forest Regressor

Random Forest is an additive model wherein decisions are made based on the additive sum of base models, wherein each base model is a decision tree. Random Forests are particularly advantages when choosing large data with numerical features, as well as categorical features. They work well on data with dense features, and hence they are tried for this dataset. Random forests are also helpful that they calculate the importance between the features. Random forests are better than decision trees, because, it prevents overfitting by creating random subsets of trees, and clustering them.

Figure 8 shows a pictorial explanation of random forest[Reference 4].

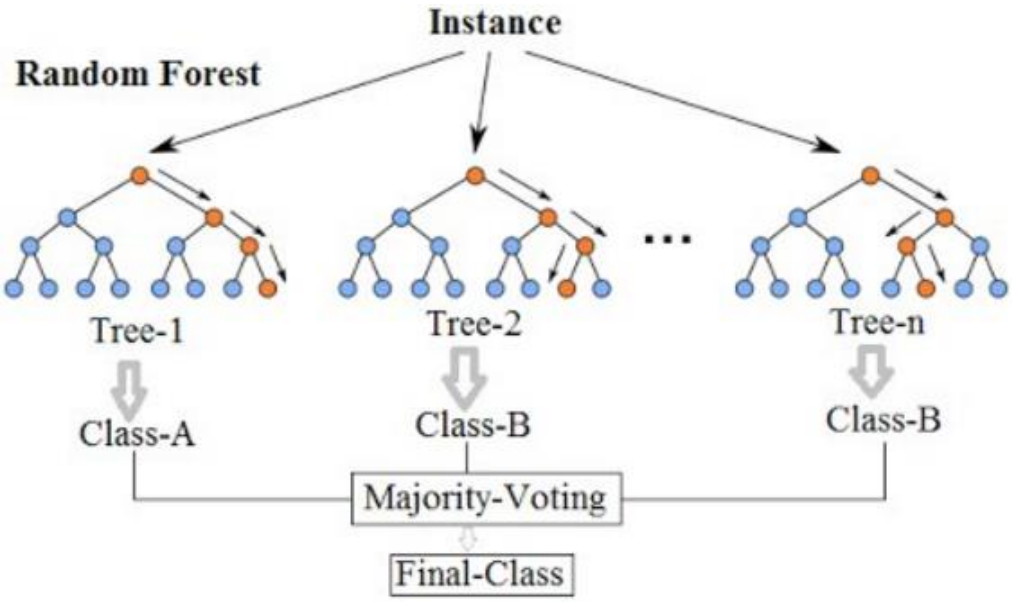

Figure 8 A simple representation of Random Forests

The following scikit – learn function is used in the implementation of a Random Forest Regressor.

class `sklearn.ensemble.` **`RandomForestRegressor`** (*n_estimators='warn', criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False*) [source]

The parameters used to model the classifier in the Random Forest Regressor are

- n_ estimators – that gives the number of trees
- max_depth – that defines the maximum depth of the tree
- max_features – the maximum number of features in a tree

The number of estimators, and the max_depth are the features varied to calculate the performance of the model. Table 4 gives an estimate of the performance of the model on the Validation set V, that is split from the training Set. The performance metric used here is the Mean Absolute Error(or the MAE Score).

Table 4 Performance metric by changing the parameters of Random Forest Regressor

Number of Estimators	Maximum depth	MAE on Validation Set
10	7	0.0694
30	7	0.0423
10	9	0.0530
30	9	0.0488

Hence the model with max_depth = 7 and n_estimators = 30 is chosen as the better model using Random Forest Regressor.

3.4.3 Gradient Boosting Algorithms – Light GBM

Light GBM is a gradient boosting framework based on decision tree algorithms.

The general algorithm of decision trees is each node is added to a level below it, starting from left to right. A new level is added for each node below it. Light GBM differs in the way that the nodes are added from left to right. The following figure 9 gives an overview[Reference 2].

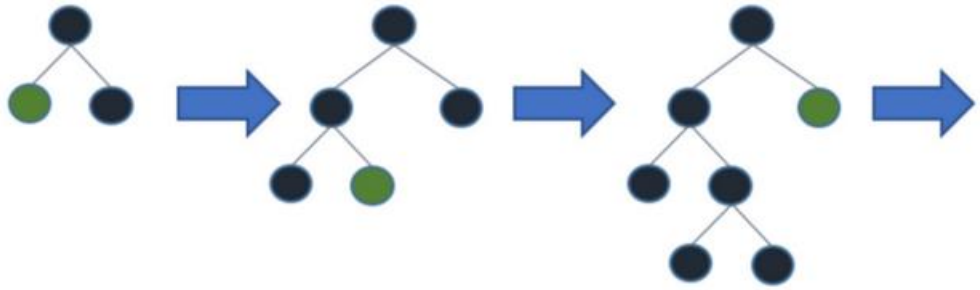

Figure 9 Growing a Light GBM Tree

Light GBM trains a Gradient Boosted Decision Tree, and supports large scale datasets. Hence Light GBM was tried for this dataset. GBM is also a fast implementation of the gradient boosting algorithms. Boosting is used for training a model sequentially, and the model learns from the previously run iterations. In each iteration, a value is predicted, which is finally summed up over all iterations of all models. With reference to the CART algorithms discussed in the class, boosting can be formulated as follows.

The implementations of the Gradient Boosting Algorithm are XGBoost and Light GBM.

Light GBM is fast because it reduces the loss when moving from left to right, rather than the normal deciison trees.

There are a large number of parameters for Light GBM, and hence finding the optimal parameters would be extensive. The parameters used are explained as follows.

- Max_depth – that controls the maximum depth of the tree. This parameter can be used to avoid overfitting,
- Min_data_in_leaf - the minimum number of leaves in a Light GBM decision tree. This parameter can also be used to avoid overfitting.

- Feature_fraction – fraction of parameters chosen randomly in each iteration.
- Lambda – Specified the type of regularization.
- Early_stopping_round – to reduce excessive iterations, if the metric of validation doesnot improve in consecutive rounds.

The parameters chosen for the light GBM model are as follows, given in Table 5.

Table 5 Performance metrics for the light GBM model, with variation in parameters.

Number of estimators	Number of leaves	Learning Rate	MAE Score
3000	27	0.05	0.0382
10000	27	0.07	0.0357
20000	27	0.07	0.0288
20000	40	0.07	0.0481

Since the model takes a long time to train, and give the results, limited tries were made to selected the best – performing model.

3.4.3 Gradient Boosting Algorithms – XGBoost Model

XGBoost is an implementation of the gradient boosting for the trees. CART is basically trees, wherein they have to be optimized over different iterations. To learn the trees, a model has to be defined, and optimized. If each function is represented as f_i , and prediction as $\hat{y}_i(t)$, the model can be respresented as in Figure 10 [Reference 3]

$$\hat{y}_i^{(0)} = 0$$
$$\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)$$
$$\hat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i)$$
$$\dots$$
$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

Figure 10 Mathematical representation of XGBoost Model

L2 regularization is used.

MAE Score calculated by varying some parameters using XGBoost Model are shown in Table 6.

Table 6 Performance metrics for the XGBoost model, with variation in parameters.

Max_depth	Learning Rate	MAE on Validatuion Set
5	0.1	0.0447
5	0.07	0.0359
7	0.1	0.0405
7	0.07	0.0387

Validation was done based on the maximum accuracy score, as calculated by the accuracy_score function of sklearn. Any higher value of max_depth resulted in overfitting of the model on test_train.

3.4.5 CatBoost Regressor

Catboost is also an open – source library that implements Gradient Boosting models. This is a type of implemenatation of GBMs. Cat Boost is actually **Cat**egory **Boost**ing, that works good, preferable for categorical data. It used gradient boosting on Decision Trees.

The parameters that are modified in catboost to control overfitting are :

- Learning Rate
- Maximum Depth of the tree
- Handling categorical variables using the parameter cat_features
- L2 Regularization
- Number of leaves : taken as a constant from the reference models already available.

The model is coded using the already available function imported form the catboost library.

MAE Score calculated by varying parameters for Cat Boost Regressor are shown in Table 7.

Table 7 Performance metrics for the Cat Boost model, with variation in parameters.

Iterations	Learning Rate	MAE Score
10000	0.05	0.0601
10000	0.08	0.0568
20000	0.08	0.0510
20000	0.05	0.0582

3.5 Model Selection and Comparison of Results

Since this is a Kaggle competition, and the test labels are not explicitly available, a portion of the data in the training set is used for evaluation of the model. That is the reason for splitting the data into train and test initially.

From each of the model above, Table 8 gives the best performing models on the Validation Set only.

Table 8 Best performance of each model

S.No	Model	MAE on the Validation Set
1	Linear Regression	0.0857
2	RF Regressor	0.0423
3	Light GBM	0.0288
4	XG Boost	0.0359
5	CAT Regressor	0.0510

In the validation sets, the light GBM is found to have the least MAE value. Each of these models is tested on the train_test data, and the performance metric used to evaluate the model is the R2 Score and Mean Absolute Error.

R2 Score is a statistical performance metric, that shows how close the actual and predicted values are, particularly in regression models. This can be a negative or a positive value, and the best model that actually predicts the actual values has a R2 Score of 1.0.

MAE Score is the mean absolute error, that tells the mean of the error between the true value and predicted values of the winning percentage.

The following table 9 compares the R2 Score and MAE obtained on the test_train data,shown in Figure 7.

Table 9 Performance Metrics of the Models against the Test_Train Set

S.No	Model	R2 Score	Mean Absolute Error
1	Linear Regression	0.7523	0.0877
2	RF Regressor	0.9374	0.0410
3	Light GBM	0.9481	0.0265
4	XG Boost	0.9320	0.0510
5	CAT Regressor	0.9201	0.0591

A good model has R2 Score close to 1.0, and a possibly less MAE. The following observations can be made.

- Linear Regression is a simple baseline model, there is no control of overfitting or any other optimization and hence has a high error.
- RF Regressor also gives a higher error rate, since the datapoints are very large – and there are rare occurrences of many data points. Unless a reasonable number of trees are specified, a large number of trees are needed for a large number of datapoints, overfitting occurs and the model performs bad.
- XG Boost is a Gradient Boosting Tree, wherein each tree is built is based on the previous tree. Error propagation could be easier, and overfitting occurs if the data is noisy.
- Memory is a major constraint in CatBoost. Hence, the number of data points has to be significantly reduced. Hence size of training set is comparatively reduced, and the model may not be fully trained.
- Light GBM is memory efficient, and it builds trees leaf – wise, rather than level – wise. Overfitting is taken care of by the max_depth parameter.

Of all these, the model that gives the best positive R2 Score and MAE Score is Light GBM and hence this model is chosen to be the optimal model.

4.FINAL RESULTS AND INTERPRETATION

The best – performing model on the training set is Light GBM. The parameters of this best – chosen model are :

```
"objective" : "regression", "metric" : "mae", 'n_estimators':20000,
'early_stopping_rounds':200,"num_leaves" : 27, "learning_rate" : 0.07,
"bagging_seed" : 0, "num_threads" : 4,"colsample_bytree" : 0.7,"max_depth" : 7
```

Out – of – Sample error for this model on the training set, is the error of the model on the test_train set. This is found to be the least, from Table 8, and hence this model is chosen to be the optimal model. Hence, this model is submitted on Kaggle, and checked for the final score. The final public score obtained was 0.0247. If a simple model was implemented, using only the given features, the MAE Score was found to be as high as 0.0929. The addition of useful features is a key factor in lowering the MAE Score.

The other models were also submitted, and checked for the final MAE Score. It was observed that all other models had higher MAE Scores. This is expected performance of the model, since this gave the best performance on the training set. Since the training set was initially divided into sample test and sample training set, the model was positively evaluated on the sample test set, and could be generalized to larger test sets.

To still get a lower MAE value, the following are to be carried on until the competition ends.

- Get accurate measure of all features, and find the feature importance, and by using a weighted combination of features.
- Use PCA if needed, to avoid overfitting.
- Use a different classifier model altogether.

[Some parameters in the code are to be modified for improvements and checked by submitting until the competition ends].

5.SUMMARY AND CONCLUSION

This Kaggle competition to find the winning percentage of players is implemented in Python. The dataset was divided into different small sub – datasets, that helped in each part of the algorithm modeling. The relationship between different features are explored in the EDA Step, and proper pre – processing is done. Some less important features are removed, and many features are added. One of the important features added was the ranking based on the mean, minimum and maximum values. The performance metric used to evaluate the classifier is MAE Score. Of all the classifier models, light GBM is found to give the least MAE on the sample test set as well as on the actual test set. The public score of the model is obtained as 0.0247.

REFERENCES

[1] https://ml-cheatsheet.readthedocs.io/en/latest/linear_regression.html

[2] https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc

[3] https://xgboost.readthedocs.io/en/latest/tutorials/model.html

[4] https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d

[5] Scikit – learn documentation

[6] https://www.kaggle.com/c/pubg-finish-placement-prediction

[7] Kaggle – already available kernels