EE 660 – Machine Learning for Signals and Systems
Final Project
PUBG Finish Placement Prediction (Kaggle)

Project Type: Collaborative (Kaggle)
Name: Thiyagarajan Ramanathan
Email: tramanat@usc.edu
USC ID: 4973341255

Abstract

PUBG is a mobile game with a number of players of players being dropped on an island and the objective being, to emerge victorious among the lot by killing the other players in the game. The sooner you get eliminated, the lower the rank you get. The developers of the game have collected the statistics of around 65,000 games and have made it available to the public on Kaggle to analyze the features and predict a player's win position given the statistics of the match as well as the player. There are about 29 features originally, with 4 million datapoints. Effective feature engineering appeared to be the key for this project. Gradient boosting methods have been providing excellent results for datasets with a lot of numerical features. In this project, XGBoost and LightGBM have been used along with other regression models to predict the win percentage of a particular player. The error on the unseen test data using LightGBM is about 0.0255. One main problem is the quantity of data. Since it is very high, it requires machines with high computational power. For all the work done on this project, Kaggle kernels were used. More feature engineering and better hyper parameter tuning can lead to better results.

Introduction

PUBG stands for Player Unknown's Battlegrounds. It is a game played on the mobile phone over the internet, the objective being to win the game, obviously. You will be dropped on an island along with another set of players. The ultimate aim is to finish in a better position when compared to other players in the game. This is a shooting game and hence you need to kill the other players, and if you are the last man alive you are the winner of the game. This is the summary of the game though there are many options such as playing as a team to win the game, etc.

The aim of this project is to predict the win percentage of any given player through the statistics of the player. If you have to finish in a higher position, you will definitely have to have did a high number of kills. Similar to the number of kills, there are many other factors such as walk Distance, headshotKills etc, using which we have to predict what position the player will finish in. Hence, this is a regression problem, with the values ranging from 0 to 1. The features contain both the in-game statistics of a particular player as well as the initial statistics (history) of the player. This is a really interesting dataset and part of an ongoing Kaggle competition. This dataset has a huge number of datapoints, around 4 million data-points in the dataset. Since the dataset is really huge, it gives a better chance of creating a very good model as well as make it easy for us to understand the data, to make better predictions.

Overview of approach:

As said earlier, the dataset is very large. The dataset contains details about 65,000 matches, with each match having around 85-90 players. We have lots of different ways to proceed with the dataset methodology. In this project, pre-train sets are used for exploratory data analysis, train and validation for choosing different parameters for a particular model and doing this for different models. The best model (regressor in this case) is chosen based on the performance on the test data we have split. The overall performance of the model is evaluated on the entirely unseen dataset, provided on Kaggle. The Kaggle system will give us the mean absolute error on the test set. The different regression models used in this project are Linear Regression, Ridge Regression, Multi-layer Perceptrons, Random Forest, XGBoost and Light GBM. The tree-based algorithms are very good for numerical feature set which makes the XGBoost and Light GBM the best candidates for this type of task. Another important point to be noted here is that, the error metric followed is mean absolute error and not mean square error like in other regression problems.

The report is organized as follows: The next section contains the implementation part where the features are explained first, how the test, train, pre-train and validation sets are created, the explanation of the different models and how they work along with the validation error results of the same model with different parameters. The last section shows how the best model is chosen. The section 'Final Results and Interpretation' contains the errors on the test set are presented and the best model is chosen for evaluation on the final unseen test set. The last section contains the conclusion and future work.

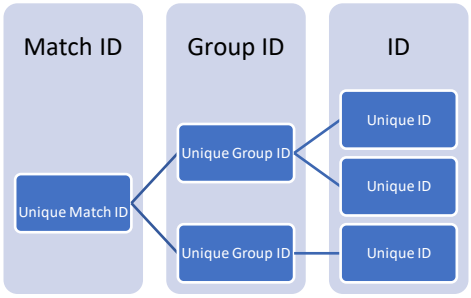
Implementation

Dataset

The dataset is obtained from a number of PUBG games played. The objective is to predict the winPlacePerc of a player (or a team) based on the stats. Since this is part of a Kaggle competition, we are given 2 files namely *test.csv* and *train.csv*. The train.csv contains around 4,446,966 datapoints and the test.csv file contains around 1,934,174. Each datapoint has a dimension of around 29 with the last column being the 'winPlacePerc' which is the target value. The test set has all features except the true values.

All the features along with their datatypes and a brief introduction is given below:

1) ID: This is the unique identification for a single player in any game. Each player gets his own unique ID whenever he joins a game. They represent an individual player no matter if he plays solo or in a squad. The total number of IDs is equal to the number of datapoints in the training dataset. The datatype is strings.
2) GroupID: This is the ID given for a particular group. Each group may have only one ID or more than one ID depending on the type of the game. The number of groups is lesser than the number of IDs, as a number of groups have more than one ID depending on the type of the game. This datatype is also strings.
3) MatchID: This is the ID given for every match. A single matchID may contain a number of different groupIDs based on the number of teams (or solo) players joining the match. The unique number of match IDs will be lesser than the number of groupIDs as there are different numbers of players in every group. This is also a string. The hierarchy between the MatchID, groupID and ID is given in the figure below. We can see that one Match ID can have a number of groups, which may have a team of players (IDs) or a single player.



4) Assists: This feature is present matches where there are groups with multiple players playing the game. Assists is related to killing the enemy. If the enemy doesn't die after you attack, you have injured the player. When someone else from your team kills that previously injured player, your team member gets the kill and you will receive the assist. The datatype is int.

5) Boosts: This denotes the number of boosts used by a player. Boosts are items that increases your health. When you receive boosts, your ability to run faster improves for a limited time. The health increase process takes some time and is not instant. The data-type is int.
6) Heals: This feature is also very similar to Boosts. The main difference between boosts and heals is that, it takes shorter time to increase health when compared to Boosts, but the movements of the player must be reduced when heals are used, otherwise there will be no effect. This feature is also int.
7) DamageDealt: This denotes the damage inflicted to the opposition. Only Headshotkills inflict immediate deaths. In these cases, the Damage Dealt is not considered. In other cases, for every bullet fired at an opponent, their health decreases. This value is denoted by this feature. The datatype is float.
8) DBNOs: DBNO stands for number of opponents knocked out. This does not mean that the opponents are killed. When you knock an opponent, they are not killed but injured and they can be revived through boosts or heals by their team members or themselves. This feature is int.
9) Headshot kills: This feature denotes the number of times a player had killed his/her opponents by shooting at his/her head. Head shots result in instant death and will not be considered in any of the category such as Damage Dealt or Assists. Head shot kills are counted in total number of kills. The feature is int.
10) Kills: This denotes the total number of kills that a particular player has done in a match. This feature is very important, and a higher kill number can convey higher win Place Percentages. This of int datatype.
11) Kill Place: This denotes the rank that is obtained based on the total number of opponents you have killed. This will be an important feature since it ranks the players that have the greatest number of kills, which may be directly related to the win Place Percentage of that player. This feature is also of the int datatype. The maximum kill place can be the total number of players in that particular game.
12) Kill Points: This feature denotes the sum of points obtained by killing an opponent. If you kill an opponent who has full health, you get varied points based on whether it is a solo, duo or a squad match type. If the opponent you have killed has a reduced health, then the amount of Kill points you are awarded varies. For instance, if you a kill an opponent with full health, you get 20 points in a solo match. This number reduces corresponding to the health of the opponent. If the opponent already has less health, then the Kill points obtained will be less.
13) Match-Type: A single match may contain multiple number of groups which may have one pl ayer or more than one player. The number of players in each group is conveyed by this featur e. This is a categorical data type and has the following values:
'squad-fpp', 'solo-fpp', 'duo-fpp', 'squad', 'solo', 'duo', 'crashfpp', 'normal-squad-fpp', 'flarefpp', 'normal-duo-fpp', 'normal-squad', 'normal-solo-fpp', 'flaretpp', 'normal-duo', 'crashtpp', 'normal-solo'.

14) NumGroups: This feature denotes the number of groups in a match. The groups may be solo, duo or a squad. This is not equal to the number of unique players in a match. This feature is of int datatype.
15) MaxPlace: This feature denotes the least rank in each match. This may not be necessarily equal to the number of groups in that particular match as the maxPlace may have skips over some groups. This feature is also of the int datatype.
16) Killstreaks: This feature denotes the number of enemies killed in a short span of time. If the Killstreak is high, then we can understand that the player has been killing a lot of people in a very short amount of time. This feature is also of the int data type.
17) Longest Kill: This feature denotes the amount of time between attacking a player and when the player dies. For example, when an opponent is killed by riding a vehicle over, the death may occur after a very long amount of time. This time is denoted by Longest Kill. This feature is of the data type float.
18) matchDuration: This feature denotes the total duration of a match in seconds. This feature is of the datatype float.
19) RankPoints: This feature denotes the rank points obtained by every player. This may be the total rank points that a player has got. This denotes the consistency of players over his previous matches. Number of kills and overall position are used to calculate the rankpoints. This feature is not specific to each game and is an external feature providing a history of the player. This feature is of datatype int.
20) Revives: This feature denotes the number of times a player has revived the health of teammates. By revive, it means to give a heal or a boost item you have to your teammate. This feature is also of the int datatype.
21) Ride Distance: This feature denotes the distance covered on a vehicle by a player. This feature is of the float datatype.
22) Swim Distance: This feature denotes the distance covered by a player through swimming. This feature is also of float Datatype.
23) Walk Distance: This feature denotes the distance covered by a player on walk. Larger walking distance means that the player has covered most of the area by walk as all players cannot easily get a vehicle. This feature is of float datatype.
24) Team Kills: This denotes the number of team mates a player has killed. This may have occurred due to a mistake, or any other reason. This feature is of int datatype.
25) RoadKills: This denotes the number of a kills a player has committed by attacking from inside a vehicle or running over the opponent on a vehicle. This feature is of int datatype.
26) Weapons Acquired: This denotes the total number of weapons acquired by a player in a single match. This is of int datatype.
27) Vehicle Destroys: This denotes the number of vehicles destroyed by a player in a particular match. This is of int datatype.

28) Win points: This denotes the total number of a points a player has obtained in all the matches played. This is an external feature and does not depend on that particular match. This feature provides a good understanding of the history of player. This feature is of int datatype.
29) Win Place Perc: This is the target variable that has to be predicted. This denotes the finishing position of a player. It ranges from 0 to 1, with 0 meaning the last position and 1 being the first position. One important observation is that, all players in the same group receive the same win place percentage. The datatype is float and the objective of the problem is to predict the win place percentage based on the other features given.

Libraries Used:

There are a number of libraries used for this project. Matplotlib for visualizations, Pandas for reading the data and processing them, scikit-learn for using the regression models such as linear regression and ridge regression, keras for multi-layer perceptron (As sklearn MLP is very slow), XGBoost library for XGBoost regression and LightGBM library for LGBM implementation. All the experiments of this project were run on Kaggle Kernels for faster training.

Exploratory Data Analysis for the features:

In this section we will analyze some of the important features. We will be seeing the distribution of some of the features along with their relation to win Place Percentage. All this analysis is done on the pre-train set extracted from the whole training set. The datapoints are not randomly split. The datapoints are split based on their matchIds. Analysis on some of the features is given below:

1) Match Size:
   The distribution of number players in every match is given in the histogram below:

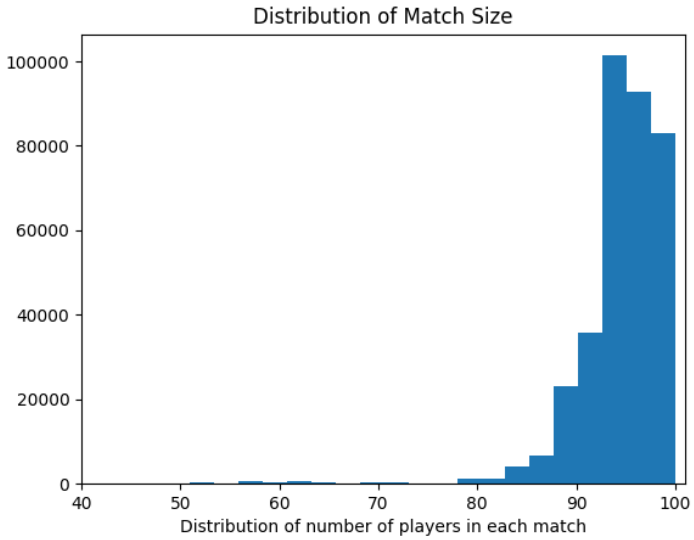


Figure 1

From this graph, we can see that the number of players in each match is in the range of 80-100. This is number of players in every match. This number may be divided into a number of groups based on the type of the match. The average number of players in each match is around 94.

2) Team Size:
The distribution of the number of players in all the groups is given below:

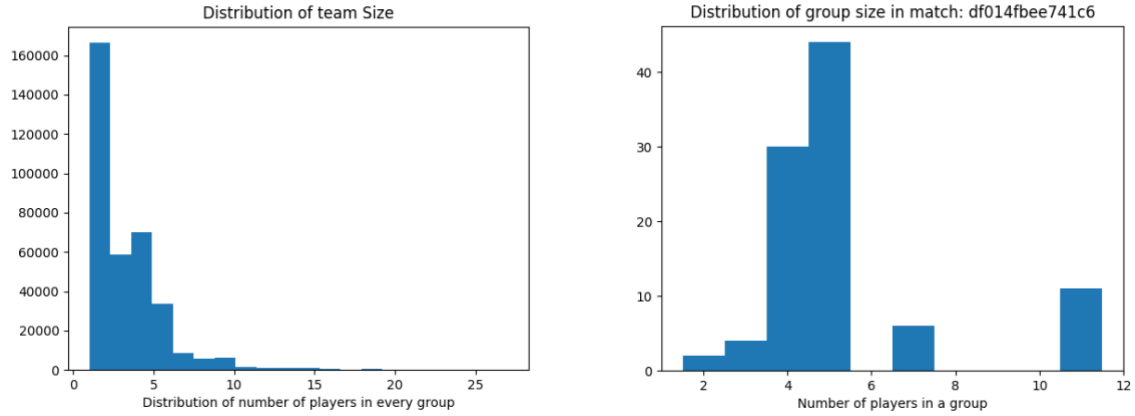


Figure 2

The graph on the left side gives the histogram of the number of players in every group, obtained from all the matches. The plot on the right gives the distribution of players in a team in a single match (match Id given above the plot). From the graph on the right, we can see that all teams need not have the same number of players in them. So, all the matches have different groups, which can have varied number of team members. This is one of the main reasons why the aggregate for each group is chosen as features. This will be explained in the pre-processing section.

3) Heals and Boosts:
These 2 features are very similar and hence they are combined together as a single feature. We can understand how these features are similar with a help of a histogram of these 2 features.

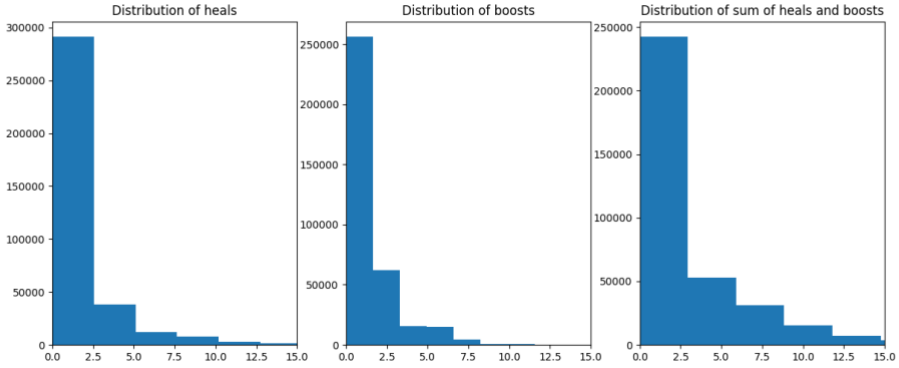


Figure 4

From the distributions, we can see that the heals and boosts are similar features and can be combined to get a new feature which is the sum of heals and boosts.

4) Distribution of win place percentage:

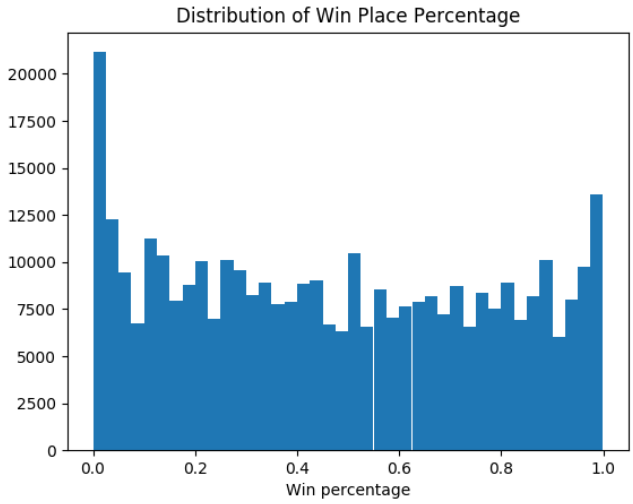


Figure 5

The above distribution is obtained for win place percentage of all players in the pre-train set. We can see that the distribution is very similar to the U-distribution with more density in the lower and higher regions of the value. We might get a better a visualization of the graph when the win place percentage of all the groups are plotted as this is plotted for all individual players.

5) Match Duration

The figure given below denotes the match duration of each game played in seconds. There are three main maps in PUBG and hence the duration of the match depends on how large the maps are. When the maps are bigger, the match duration also increases. This feature has a correlation of 0 with the win place percentage. Hence this feature is completely irrelevant and not required.

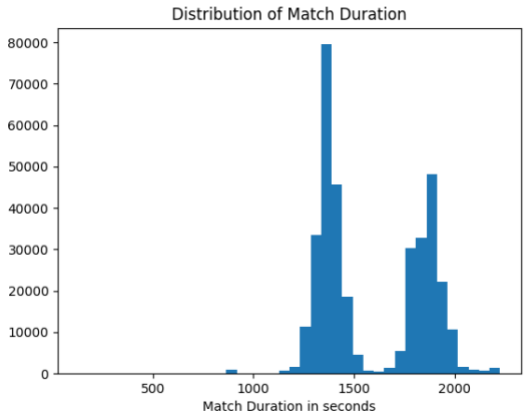


Figure 6

We can see that there are 2 distributions with different mean match durations. This clearly explains the fact that there are 2 main battle fields and this feature will be of no importance.

6) Headshot kills and Road kills:
These two features can be considered as special skills of a player. When a player performs a head-shot kill or a road kill, the count of kills also increase. Hence, finding the headshot rate and road kill rate can help giving importance to these features. The histograms of these 2 features are given below:

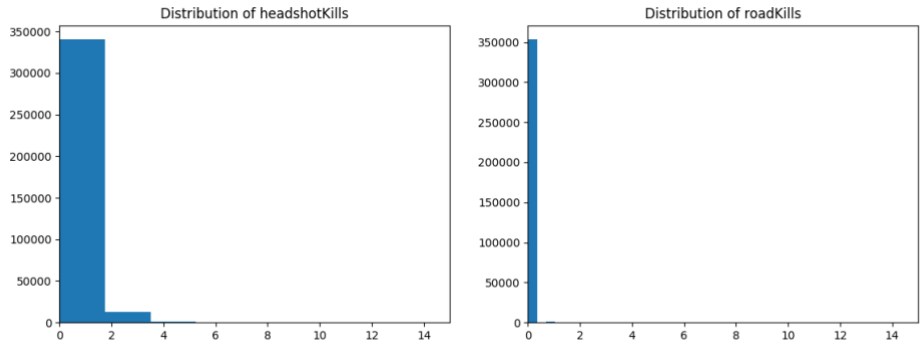


Figure 7

The distribution of these features is similar, and a very small number of players have a non-zero attribute for these features. Only 16% of the players have a non-zero value for headshot Kill and only 0.3% of players have a nonzero value for road kills and hence these 2 features are very sparse.

7) Kills vs Damage dealt
The following plot is a scatter plot between the number of kills and damage dealt. We can see that there exists a direct relation between these 2 features. We can also see that there is a high correlation between these 2 features. These 2 features can also be combined in some way to know the best players in every game. Since, players who have a good kill place have higher number kills, which is directly proportional to the damage dealt.

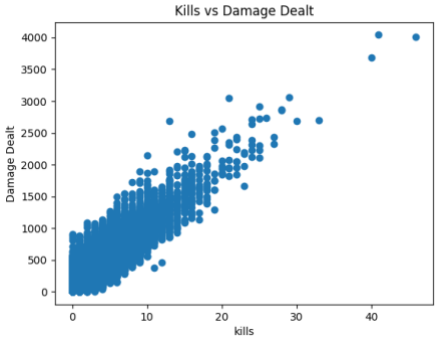


Figure 8

8) Walk Distance vs Win Place Percentage

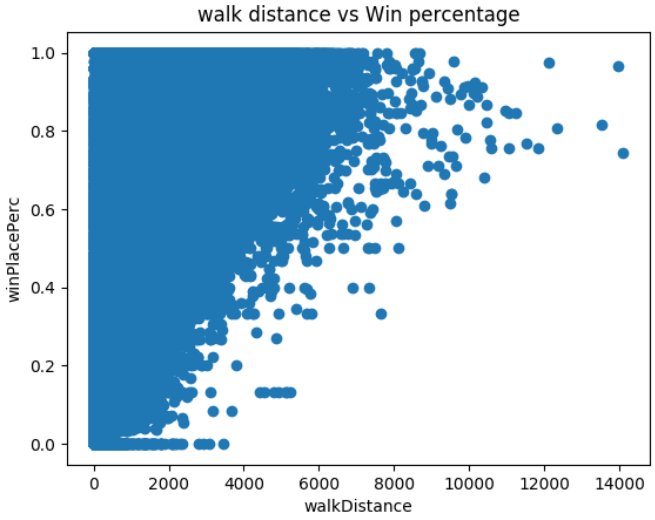


Figure 9

The above plot shows the relationship between walk Distance and win percentage. We can see that these 2 features are highly correlated. We can see that higher walk distances correspond to higher win percentages. If the player has covered a lot of distance, it corresponds to the idea that he has a higher win percentage. This can be confirmed with regression model that uses only one feature.

9) Kill Place vs Win Place Percentage

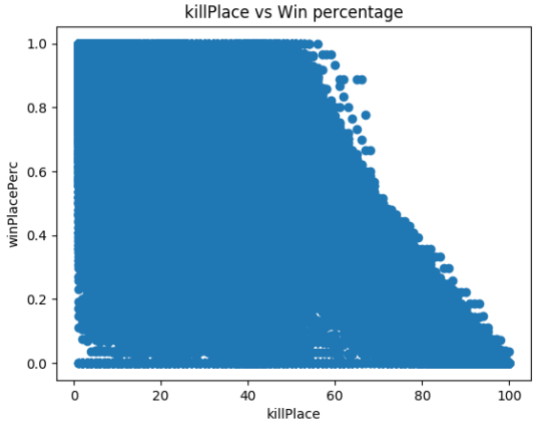


Figure 10

The scatter plot is obtained using Kill place on the x axis and Win percentage on the y-axis. We can see that there is an inverse relation between Kill place and Win percentage. Players with higher kill ranks will have higher win percentages. This can be understood with the above plot as well. Players with lot of kills get higher win percentages.

The following section contains the correlation between all features of the dataset

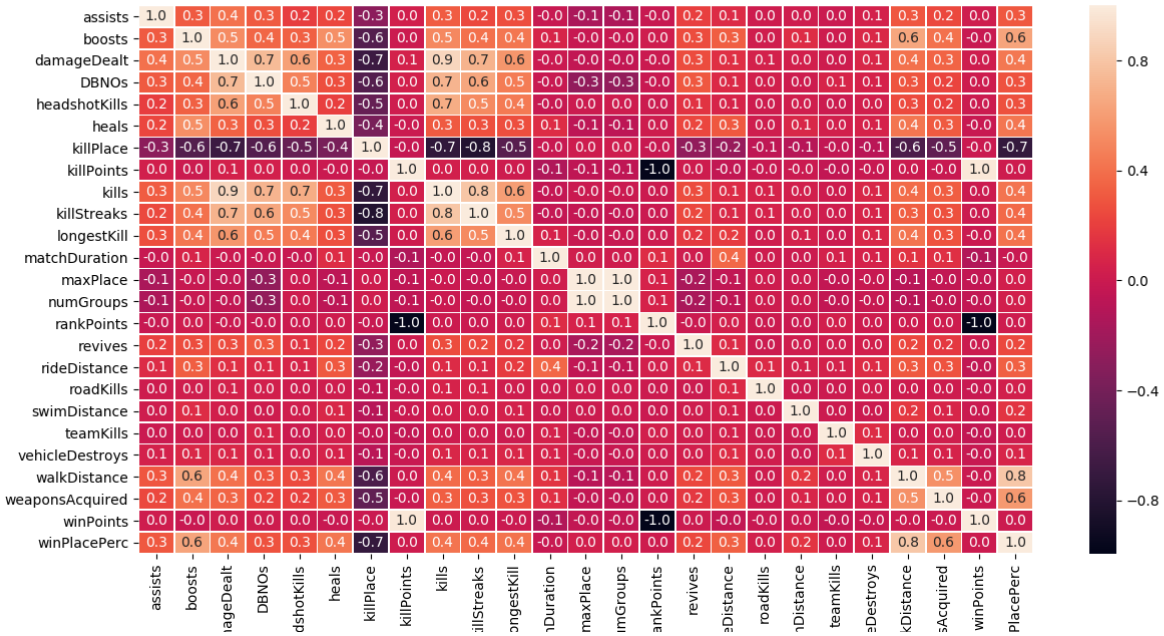


Figure: Correlation between all the features in the original dataset (without any grouping)

From the above plot, we can see the correlation between all the features. +1 means that there is a high positive correlation and -1 means that there is a very high negative correlation. And 0 means that there is no correlation between the features. Observations from the correlation plots are as follows:

a) The correlation between win Place Percentage and walk Distance is equal to 0.8 which conveys that walk distance is a very important feature for predicting the win Place Percentage.

b) We can also see that there is a high negative correlation between Kill place and win Place Percentage. When a simple regression model was trained using only one feature at a time, we can understand the importance of Kill Place more.

c) We can also see that there is a very high negative correlation with Kill place and features such as Damage Dealt, kills, kill streaks, walk distance and weapons acquired as well. This is mainly because Kill Place is a rank. So, players with higher kills, get higher ranks (meaning, in the top 10). They will have the least ranks but all the other features such as damage dealt, kills, kill streaks etc., will be higher. This is the main reason why there is a high negative correlation between kill place and other features.

d) There is a high correlation between the following features as well:

I) walkDistance and boosts
II) Damage dealt and headshot kills
III) headshotkills and kills
IV) weapons acquired and distance
V) kills and killstreaks
VI) walk distance and kills

The following results are obtained by running a simple linear regression model with just one of the mentioned features without any other pre-processing. The error values are obtained on the pre-train set.

Feature	Mean absolute Error	Feature	Mean Absolute Error
Assists	0.252	Damage Dealt	0.2353
Boosts	0.1974	DBNO	0.254
Heals	0.236	Walk Distance	0.142
Rank Points	0.267	Swim Distance	0.2636
Kill Points	0.267	Ride Distance	0.246
Kills	0.238	Road Kills	0.2673
Vehicle Destroys	0.266	Revives	0.267
Head-shot kills	0.254	Team Kills	0.267
Longest Kill	0.239	Weapons acquired	0.202
Kill Place	0.164	Kill streaks	0.244

From the results in the table above, we can confirm our observations with correlation between walk Distance-Win Place Percentage and Kill Place-Win Place Percentage. A simple linear regression model with just one feature from these 2 can give an error of about 0.15 whereas other features have an error of about 0.26. The features with better correlation with the win place percentage have a lower error.

These highly correlated features can be combined together to give a better representation of the data. Some new features are added based on the correlation between the mentioned features. We will see more about adding new features in the next sections.

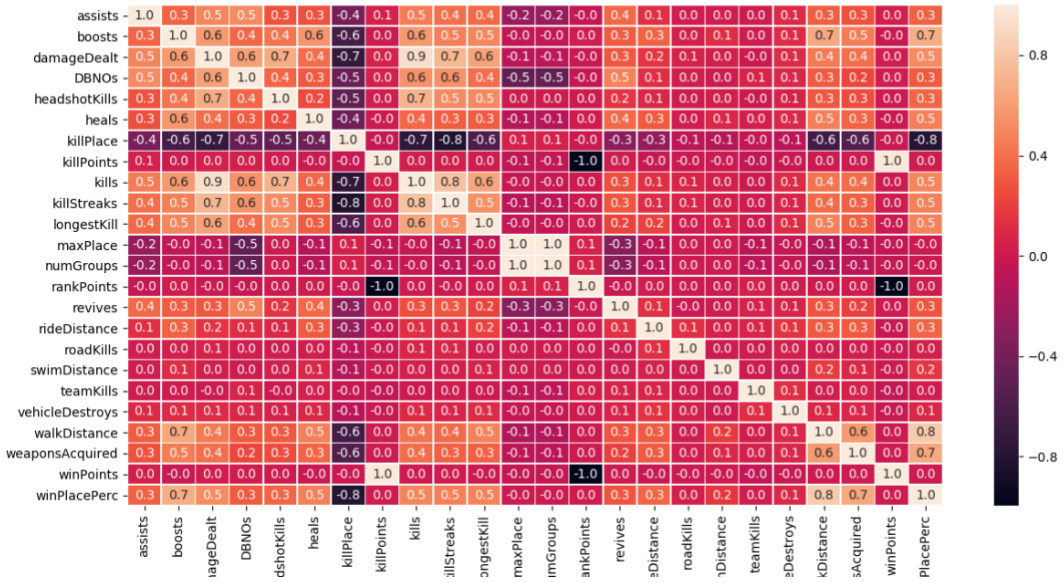


Figure: Correlation between all the features after aggregating based on mean

The above correlation heatmap is obtained after aggregating all the groups with the mean of the player stats. We can see that the same correlation values between all the features and win Place Percentage has increased. For instance, consider the features boosts, damage dealt, heals, killPlace, kills, kill streaks, longestkill and weapons acquired the correlation has increased by value 1. This is the main reason why the player stats are aggregated to get the feature vectors of a while group rather than individual datapoints.

This can be explained in an easier way. The stats of a player are important, but the win Place percentage mainly depends on the stats of the opponents. When you are in a squad where your teammates have very good stats, you still win the game with a better win percentage. When you are in a game where there are lesser number of better players when compared to you, you will definitely have a better finish percentage. This is the main reason to aggregate the features for every group. Ultimately, we need to consider the stats of a group as a whole to predict which position the group will finish in. This will be explained more in the next section.

<u>Preprocessing, Feature Extraction and Dimension Reduction:</u>

The dataset is all numerical except for the features such as Id, groupId, matchType and matchId. One of the main ideas that has to be understood from the dataset, in every match there are a lot of groupIds, with each groupId containing a lot of player Id's. All the Id's in a group in every match have the same win Place Percentage. Hence, this becomes more of estimating the win percentage for every group rather than estimating it for every player in group. But we have different sets of statistics for every player in a group, even though the win percentage for every player in the group is the same.

To handle this, the stats of all the players in a particular group have to be combined in some way. In this project, different methods have been used to handle the stats of the players belonging to the same group. The methods of combining the player stats are given below:

1) Mean: This is the first idea that comes to our mind. Calculating the mean of the stats of all the players in the group and assigning the mean values to each of the different groups. This might lead to a lot of information loss, as the group members may have diverse stats and hence we might get misleading information. Hence the first set of features for every group are the means of the all the features for every player in a particular group.
2) Maximum: Once the mean of the features is added, to retain the maximum values in each group, the max of all features among all the players of the group are added as new sets of features for every group.
3) Minimum: Similar to the maximum values being added as new set of features, the minimum values among all the features for every group are calculated and can be created as a new set of features for every group.
4) Sum: Adding the sum of all the stats of all the members of a group can be also added as features to every group.
5) Standard Deviation: The standard deviation of all the stats of the members of every group can be calculated and added as new set of features for every groupId.

Thus, we have created new sets of features for every group. Ultimately, the number of datapoints we have is going to be equal to the number of groupIds in total dataset with each datapoint having multiple features calculated from all the members of a particular group. For groups which have only one player, the final group stats are going to be the same as the stats of the solo player. For obtaining the win Place percentages for all the players in a group, the win Place percentage of the particular group is first calculated and all the players in the group will be assigned the same win place percentage.

Apart from adding the new sets of features for all the groups, the percentiles of all the new added features are also added. For every match, there will be a number of groups. After adding the new features that were obtained by combination of all stats of the players in each group, we need to know where these values stand in terms of the all groups in one match, otherwise it would be difficult to figure out the differences of all the groups in a match. So, a percentile for all the newly added features are calculated, and new sets of features are created for percentiles for each of the new stat. For instance, if mean, min and max features of all the members of the groups are added as new features for every group, then 3 more columns having the percentile of values specific to the range of values in every match are added for every group corresponding to mean, min and maximum.

This might be a bit difficult to understand. For instance, let the number of matches be 10. Let each match have around 10 groups and let each group have members ranging from 1-20. There will be a total of $matchnum * groupnum * playernum$. Let the number of features per player be equal to 10. The following steps have to be added for calculating the group specific features:

a) Find the mean, minimum and maximum for all the features among all the players in each group. After this is done, there will be a dataset containing only the matchId and the groupId. So, the new set of datapoints will be of dimension 30 (since there are 3 sets of features added for each group). The number of datapoints is equal to $matchnum * groupnum$ (since all the Ids of one group have been combined to form a single value for every group).
b) The next step is to find the percentile values of all the features respective to groups of the same match. So, for three sets of new features each of dimension 10, another set of features with dimension 10 containing the percentile values corresponding to the 10 features will be added. In this specific case, there are 30 features originally and after adding the percentile values, there will be a new set of 30 features. So, the final dimension of the dataset will be $matchnum * groupnum * 60$. This is how the individual IDs are combined to get the final stats for every group. And this is done only because all the IDs in the same group have same win Place Percentage.

This marks the completion of the first set of feature extraction steps.

Adding new features and their effect:

There are many features that are highly correlated between themselves and also highly correlated with the win Place percentage as well. These features can be combined to get better features that have a high correlation with the win place percentage and actually help the regressor to predict the results better. The following features are added:

1) Percentage of head shot kills = headshotkills/kills
2) totalDistance=walkDistance+swimDistance+rideDistance
3) skills=headshotkill+roadkill
4) killstreakrate=killstreaks/kills
5) totalHealth=heals+boosts
6) distance_over_weapon=totalDistance/weapons acquired
7) Distance_over_heals=totalDistance/totalHealth
8) Walkdistance_over_kills=walkDistance/kills
9) Killplace_over_maxplace=killplace/max_place

The correlation heatmap confirms the correlation of these newly added features with the win place percentage. We can see that some of the newly added features provide high correlation with the win place percentage, especially total distance, total health and kill place over max place.

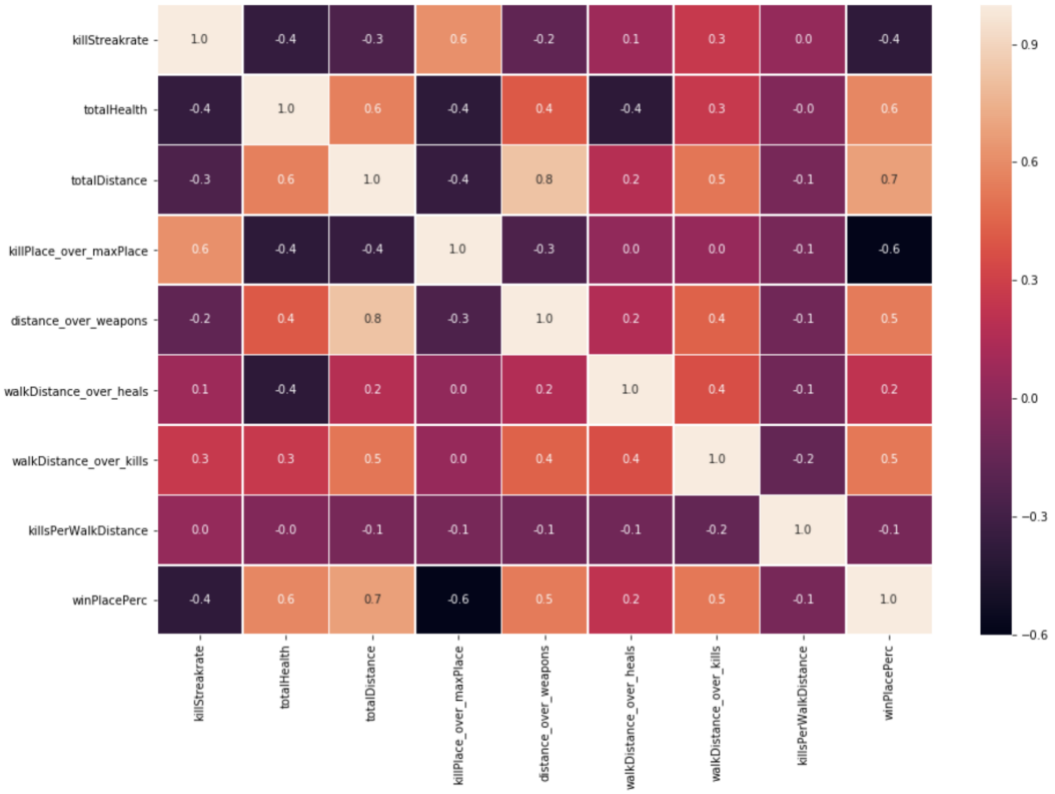


Figure 13

Apart from these new set of features, there are 2 other features added. One is the team size feature which represents the number of players in every group. Since all the player stats in one group are combined together, this new feature would tell us the number of players in each group. Another feature is match size, which represents the number of players in each match. These are the final set of features. These new features are combinations of the original features that had a high correlation with the win place percentage. The above graph contains the correlation values obtained with the new set of features and the win Place Percentage. We can see that the correlation between these new features and the win place percentage is very high. Also, the correlation between the new features is mostly less. These were obtained on the original set of features and not on the aggregate mean values. The correlation between these features and the win place percentage may further increase when they are represented for every group individually. Some features used to create these new sets of features are removed. Examples include swim distance, ride distance and walk distance as they are combined to get the total distance.Since the size of the dataset is very high, even with the increase in the feature dimension there is no much need for a dimensionality reduction technique to be used. We can see that some of the features are very highly correlated and hence reducing the dimension may decrease the model complexity, but the training data set is very large, and the number of features is very less. We are explicitly removing some of the features with low correlation without using PCA. Hence, the increase in the feature space does not necessarily require us to a use dimension reduction technique like PCA to be used.

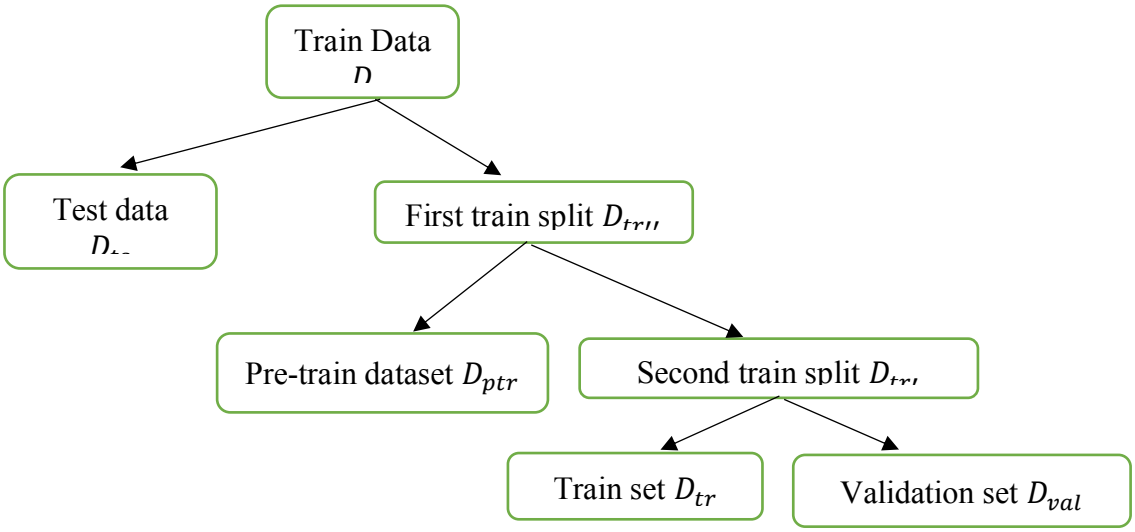
Dataset Methodology:

The training dataset is really huge with around 4 million datapoints. We have lot of flexibility in choosing the test and train sets. In this project, all the splits are based on the **matchID**s. As told earlier, the whole idea of predicting the win place percentage depends on the match you are playing in. It depends on your teammates as well as the quality of the other players you are playing the match with.

We cannot randomly split dataset for this project. For instance, consider one match ID which has about 90 players. If we are randomly splitting the datapoints into the train and the test set, 50 players may go into the train set and 40 players go into the test set. When we are aggregating the groups to get the feature vector for each group, some groups may be present in both the test and the train set. The bad players of one group may go into the test set and good players may good into the train set. If this happens, we will have the same win Place Percentage as label for both these categories. This is a problem because, we want the regressor to predict a low as well as a high win place percentage for one group with good members on one set and low stat members on the other. Hence, to resolve this issue, we need to split the test, train and validation sets based on the matchIds. This is also the idea behind aggregating the features based on the characteristics of members of the group.

The win place percentage of a group depends on the quality of the other groups in the match as well. Hence, one match cannot be split into training and test sets as some groups will be missing in the train set and viceversa. Hence, the splits will be made based on the matchId. This results in unique matchIds present in the dataset splits and there shall be no common matchIds in any of the dataset splits.

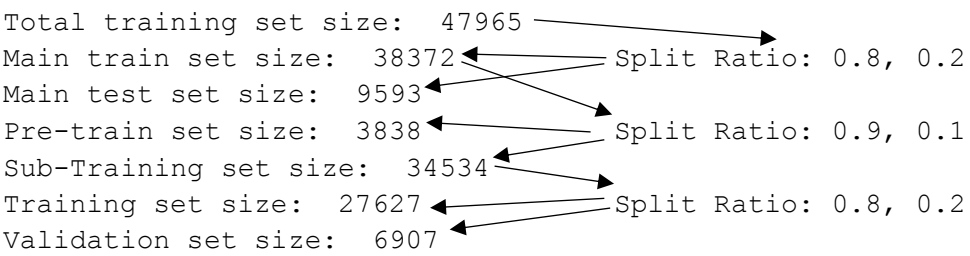
The method of dataset split in this project can be explained with the flowchart below:



Flowchart: Dataset Methodology

After performing the splits based on the flow charts, we will have 4 main datasets namely, Pre-train set, train set, test set and validation set. This test set will be used to evaluate out best models chosen based on results obtained on the validation set. The validation set will be used for evaluating the different models, and the best model for each different type of regressor will be evaluated on the unseen test set. The pre-train set was used for Exploratory data analysis and other calculations on correlation as well. Once this is done, the complete training set will be used to train the final classifier which will be used to predict the unseen main test data, whose predictions we can submit to obtain the final out of sample regression error.

The splits and number of datapoints in each of the splits are given below:
All the splits are based on the number of unique matchIds.



The number of datapoints may not be always constant, since different matches have different number of groups in them. The dataset is really huge with a large number of datapoints. Hence, to choose the best parameters using cross-validation through grid search is an impossible task. Also, the number of features is really high and hence cross-validation is very hard to be performed on the entire training sets. From the D_{tr} we have after following the dataset split methodology, we can use tiny subsets from this data to be used for cross validation even though this is a humungous task considering total the number of data points in the entire D_{tr}. The cross-validation procedure also depends on the type of regressor that has been used. In this project, the main regressors used are XGBoost, Random Forest Regressor, MLP Regressor, Linear Regression, Ridge Regression and LightGBM regressor. Linear Regression is also used in some cases for performing the data analysis.
To choose the best parameters, specified sets of parameters for each of the models are tried and the performance on the validation sets are noted. Once this is done, evaluation for each model is performed in the original split test dataset. Based on the results of all the performance of all the models on the test set, the final model is chosen and the out of sample error is calculated with this model. Number of usages of all the datasets is given below:

1) *Validation and Train set:* Depends on the number of parameters for each regressor and the total number of different regression techniques used.
2) *Known test set*: The number times this test set is used is equal to the number of regression techniques we are using.
3) *Unknown test set:* This set will be used only once to calculate the out of sample error.

The best set of parameters from each model will be chosen and trained with the entire $D_{tr''}$ and then evaluated on the test set we have split. Once the best model is chosen, it is used to predict the unseen test set which is submitted to Kaggle to get the final out of sample error.

Training Process:

As mentioned earlier, there are 5 different regression models that are used in this project. They are Linear Regression, Ridge Regression, Random Forest Regressor, Multi-Layer Perceptron Regressor and Light GBM Regressor. The principles of each of the different models will be explained below. The results of regression with different settings of the same model are also given with the explanation of the classifier.

Linear Regression:

Linear Regression is one of the simple Regression models. In this approach, the weights for each feature are calculated. The feature weights are obtained using training process. The main aim of Linear regression is to reach at the optimal weights for each feature. This is a Maximum Likelihood estimation for the parameters of the model. In this case, the parameters are the weights for the model. If we add a constraint or assume that the prior for the weights is not uniform, the problem becomes more like a Maximum A posterior (MAP) estimation.

The output for linear regression is given by: $y = mx + b$ where x is the feature vector and b being the bias. 'm' is the set of weights corresponding to each of the feature in the input vector x. The cost function for linear regression is mostly MSE, which is used for training the weights through the Gradient Descent optimization procedure. There are no parameters to be chosen for linear regression. The only two parameters include fit_intercept and normalize.

1) Fit_intercept: This parameter decides whether we should include the weight term w0
2) Normalize: This parameter decides whether we should normalize the features before we should normalize the features before we train the model.

Reason for choosing Linear Regression: This is a linear model and hence was chosen as a baseline model. It is really fast even for a large amount of data.

The following table contains the performance of linear regression with different parameters on the validation set. The error metric used is mean absolute error which is mean of the absolute error between the test and the predicted errors. For all these experiments, a subset of the final training set was used, and the evaluation was done on the whole training and the validation set since it is impossible to run cross validation for the entire training set.

Linear Regression Parameters		Error on Train set	Error on Validation set
Fit Intercept	Normalize		
True	True	0.057436	0.057415
True	False	0.0554	0.05543
False	True	0.05567	0.05569
False	False	0.0556	0.05570

Table: Linear Regression results with various metrics

We can see that the best parameters for Linear Regression are Fit intercept being true and normalize being false. If we normalize all data points separately, there is not point in doing that. If a normalization has to be applied, all the data points have to be normalized based on the group or the match. Since we already calculating the ranks before training a regressor, re-normalizing those features is not a good idea.

Ridge Regression:

Ridge Regression is very similar to Linear Regression except that there is a constraint on the weights unlike Linear Regression. The output equation for ridge regression is similar to that of the equation for linear regression. The main difference is in the error function. Instead of MSE, the cost function for Ridge regression is going to be MSE plus the penalty on the sum of the squares of the weights used. The equation for the cost function for ridge regression is given below:

$$Cost\ (W) = RSS(W) + \lambda * (sum\ of\ squares\ of\ weights)$$

$$= \sum_{i=1}^{N} \left\{ y_i - \sum_{j=0}^{M} w_j\, x_{ij} \right\}^2 + \lambda \sum_{j=0}^{M} w_j^2$$

The only tunable parameter for ridge regression along with fit intercept and normalize is:

1) Lambda (λ): The parameter controlled to adjust the penalty on the weights. The regularization parameter has to be tuned to get the best parameter setting.

Reason for choosing Ridge Regression: This is also a linear model and a little advanced when compared to Linear Regression. This is chosen to differentiate between different linear models.
The performance for different settings for Fit intercept and Normalization are given below. For these experiments, default regularization values were used.

Linear Regression Parameters		Error on Train set	Error on Validation set
Fit Intercept	Normalize		
True	True	0.07574	0.07578
True	False	0.05541	0.05544
False	True	0.05567	0.05569
False	False	0.05567	0.055695

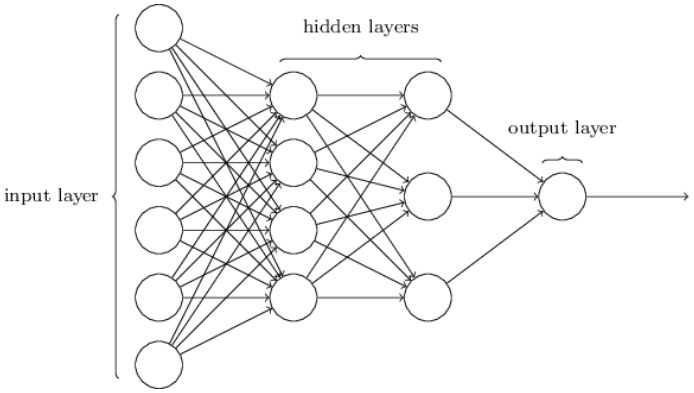
The performances for different values of regularization penalty terms is given below. For all these experiments fit intercept was True and Normalize was False, since that gives the lowest error on the Validation set.

L2 Regularizer Value	Train Error	Validation Error
0.0001	0.05541	0.05546
0.001	0.05541	0.05546
0.01	0.05541	0.05546
0.1	0.055418	0.055470
1.0	0.055490	0.055499
10.0	0.055505	0.05510

We can see that the regularization penalty does not play any importance since there is not any over-fit. We can see that the results of ridge regression are very similar to linear regression. Hence. For this dataset 'l2' regularization does not give any advantage.

Multi-Layer Perceptron Regressor:

This is a class of artificial neural networks. The objective is to learn the patterns in the data through the use of multiple hidden layers. The working principle of can be extended from simple Regression models. But we can control how many nodes otherwise called as neurons the network can have. No matter how many features there are in the input data, we can decide the number of neurons and the number of hidden layers. So, this more like a controlled regression that gives us more flexibility on the weights.



The figure above demonstrates a simple MLP model. This model has one 2 hidden layer and one output layer. The number of neurons in each hidden layer can be controlled. The output of each layer is fed to an activation function. There are many activation functions such as 'relu', 'tanh', 'sigmoid' etc. The activation functions act as squishing functions. They convert the entire range into the desired range mostly between -1 and 1 depending on the type of the activation function. The weights are learnt through a technique called back propagation which uses Stochastic Gradient descent for optimization. The only difference is that, the gradient descent algorithm is used to

change the hidden weights, by propagating the error obtained from the outputs. For the hidden layers, the outputs of hidden layers are differentiated with respected to the weights and the training process is repeated. There are many hyper parameters for any MLP Regression network. The main parameters that have to be tuned are explained below:

1) Number of hidden layers and number of nodes in each layer: To control the number of layers we want the network to have along with the number of neurons per layer.
2) Activation Function: This denotes the activation functions to be used for hidden layer outputs. Some popular values include 'relu', 'sigmoid' and 'tanh'.
3) Learning rate: This parameter specifies the learning rate to be used for optimizer
4) Number of epochs: This parameter decides the number of epochs for training

These parameters have to be chosen using cross-validation or other heuristics. For this project, subsets of the training data are chosen for cross validation and deciding the best sets of hyper parameters. Also, since this is a regression problem, there will be only one output node and the output can also be applied with 'Relu' activation.

Reason for choosing Multi-layer Perceptron: MLP can bring in high non-linearities into the model and understand the model better than Linear and Ridge Regression.

The performance of Multi-Layer Perceptron for different number of nodes and hidden layers is given in the table below. For all the experiments, default learning rate (0.001) was used with activation function being 'relu'.

Hidden layer with nodes	Train Error	Validation Error
(1000, 500)	0.05622	0.0567
(1000, 500, 100)	0.0511	0.0521
(1000, 700, 500, 100)	0.0505	0.0585

The best choice for MLP is a 3-layer network with 1000, 500 and 100 neurons respectively. This is chosen based on the validation error. The error on the test data will be given below in the next section. We can see that, when the number of layers increase, the training error has decreased but the validation error has increased very much. Thus, the networks with higher number of layers cannot always have a very good performance. To attend to this issue of over-fitting, '11' or 'l2' regularization has to be used. But for this project, we have not tried regularization for MLP, and the best model would be the 3 hidden layer network with 1000, 500 and 100 neurons respectively.

Random Forest Regressor:

Random forest is a type of additive bias model which learns the kernel functions from the data itself. In classifiers like SVM, we have to decide the kernel functions. But in Random forest, the splits (otherwise kernel functions) can be learnt from the data itself. This is the main reason why Random forest regressors are called as Adaptive Basis Model functions. Random forests are built from a varied number of decision trees. Decision tree is a simple tree function, which predicts the output by creating different splits in all features one at a time. Decision trees are slightly better than chance, and hence called as weak learners. When we combine a number of decision trees and get the average of the prediction, it is called as bagging. But in bagging, all the features are considered in all the decision trees. The main difference between bagging and random forest is that, in random forest all the different trees are not built with all the input features. The features are also randomly chosen to build the trees. The idea of random forests can be explained with the figure given below.

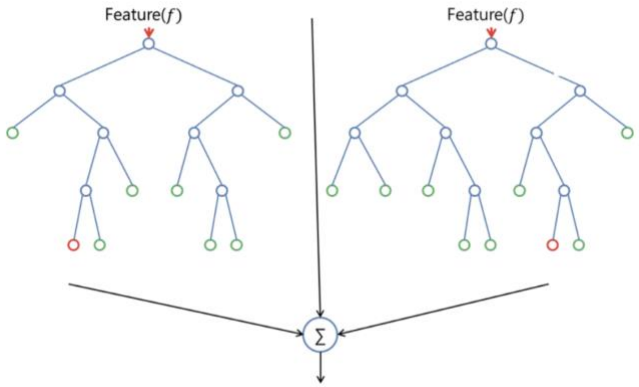


Figure: Simple Idea for Random Forest Regressor

The figure above contains trees built with different sets of features and both trees give a particular output. The average of the outputs is taken and predicted as the output of the entire model. Thus, Random forest classifiers are a bunch of decision trees with different sets of features put together. This can be understood with the equation give below:

$$\hat{f} = \frac{1}{B}\sum_{b=1}^{B} f_b(x')$$

This is the general equation for Bagging. The only difference in this equation for random forest is that, all sets of features are not used for all the trees like in bagging. So, the feature dimensions of input vector x will not be a constant in random forest. This is the working of the random forest regressor. There are many hyper parameters that require tuning in a random forest model. The most important parameters are explained below:

1) Number of estimators: This parameter specifies the number of different trees to be built
2) Depth of tree: Usually, all trees are grown till last split needed to make the prediction correct. This results in overfitting. To avoid this, the trees are completely built till the end and pruned based on the depth of tree parameter. This can have any integer value and has to be chosen carefully.
3) Minimum Loss Reduction: The splits in the decision rules are made based on the reduction in the loss made by creating a split in a particular feature. This parameter sets the minimum value required for a split to be created. This is also another parameter to prevent overfitting. If splits are created with just a small error decrease, then the we are overfitting the model.

There are many other parameters for the random forest regressor, but they are not as important as the above-mentioned features.

Reason for choosing Random forest Regressor: When the input dimension increases, the number of parameters that are to be learnt in MLP increase which increases the complexity of the model. Also, tree based methods have a hood performance history for tabular data with a lot of numerical features. Trees based methods do not require any weight learning and hence are faster than traditional methods.

The results with different settings of number of estimators for the random forest regressor is given below. For obtaining the table below, default number parameters for max depth and max features were used. In the case of sklearn, the default value for max depth is None and for max features it is 'auto' (it is equal to the total number of features).

Number of Estimators	Train error	Validation error
1000	0.0516	0.0518
2000	0.0515	0.0519
3000	0.05122	0.05124
4000	0.05118	0.0511
5000	0.0512	0.0514

Some results with other parameters are given below. For the number of estimators, it was 1000 for all the experiments.

Max depth	Max features	Train error	Validation Error
5	10	0.0856	0.0860
5	20	0.0821	0.0845
5	40	0.0829	0.0837
10	10	0.0663	0.0670
10	20	0.0638	0.0640
10	40	0.062	0.0627
None	10	0.0624	0.0628
None	20	0.0608	0.0615
None	40	0.0601	0.0609

From the tables above we can see that the best parameters for Random forest regressor are number of estimators being 4000, max depth of None (the nodes will be split till the end) and the max features being 40. After feature engineering, we have a dataset of dimension around 170. Hence, when the max number of features that are used for the tree increases, the performance of the model also increases. Also, the dataset is really large and random forest regressor takes a huge amount of time for training. Hence all these experiments were performed on very small amounts of data (smaller when compared to the amount of data used for other models). This was just used to compare the performances of XGBoost and LGBM with Random Forest Regressor, to compare their speeds and the performances.

XGBoost:

XG boost stands for Extreme Gradient Boosting. It is otherwise called as a regularized boosting technique. This algorithm is also based building a number of decision trees. XG boost is also an adaptive basis model function with and is a modification of the Adaboost technique. In random forest, we simply take the average or sum of all the outputs of the different trees. In other words, we calculate the average of the outputs of all the trees. But in Adaboost, the outputs of all the different trees are not added with equal weights. Here, a weighted average of outputs of all the trees are taken. This is the main difference between Adaboost and Random Forest. The trees are created in a sequential manner rather than a parallel manner like in random forest. The future trees are built in such a way that the next tree reduces the total error. This is done by giving higher weights to datapoints that are misclassified previously. The future trees are built to reduce the total cost. The equation for Adaboost model is given below:

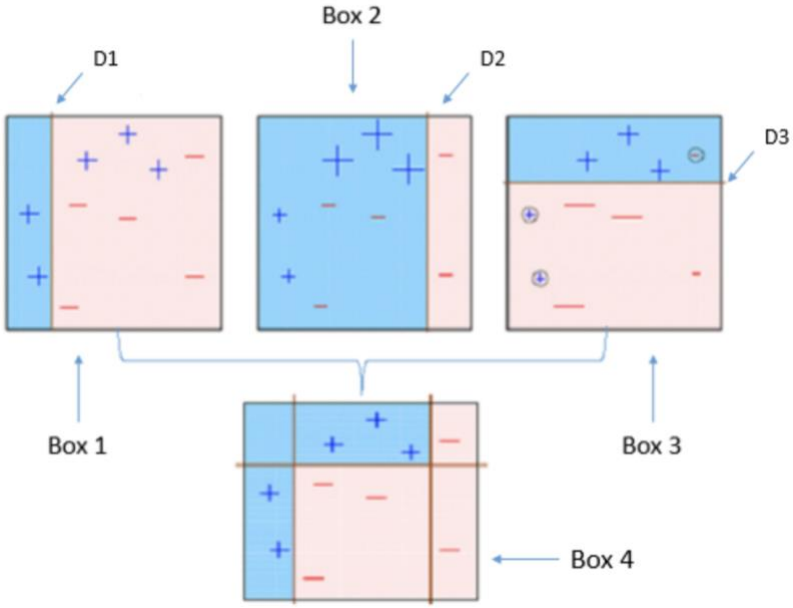
$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Here, alpha denotes the weight for each tree and 'h' denotes the trees. Adaboost is very difficult to be used for Regression problems because the loss function used for Adaboost is exponential categorical loss function (mainly used for classification problems). Even though adaboost can be used for regression problems, it was mainly derived for exponential loss functions. And they are very slow. This is when XGBoost comes in handy. Also, there are constraints on the weights for each trees, making this a regularized boosting mechanism. XGBoost has a lot of customizable parameters and it is much faster when compared to other tree-based techniques. In XGBoost, the gradient of the loss function is calculated iteratively, and the new trees are fit based on the gradients.

$$-\alpha\left[\frac{\partial(L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, \text{ where } \alpha \text{ is the learning rate}$$

The above equation is used to find the gradient of the loss function. This idea is the main difference between random forest and gradient boosting. Even in Adaboost, the same function is performed

but it is more optimized for the exponential loss function whereas XGBoost works for generic loss function.



The figure given above gives a sample decision boundary generated using XGBoost. We can see that, in the initial iterations the misclassified points are given high weights when compared to the other set of points. This is an important aspect of XGBoost.

For this project, we will be using the mean absolute error as specified in the project description. There are many important parameters for XGBoost. Some of them are explained below:

1) Metric: 'mse', 'mae' etc depends on what error we want to minimize. In this case, it will be the mean absolute error.
2) Number of Estimators: Similar to the number of estimators in Random Forest, this also specifies the number of trees to be built.
3) Maximum depth: This determines the depth of each tree similar to random forest. This feature prevents overfitting the tree and effectively prunes the tree.
4) Learning rate: This parameter is the alpha value used when calculating the gradients
5) Gamma: This is otherwise called as the minimum loss reduction. This decides the minimum loss required for creating a split.

All these parameters have to be chosen by cross validation or by other heuristics. In this case, since the dataset is very large, cross validation is difficult to be performed.

Reason for choosing XGBoost: XGBoost is faster and better than random forest and has a very good performance on tabular data. It is a regularized boosting technique and can use parallel processing for tree building.

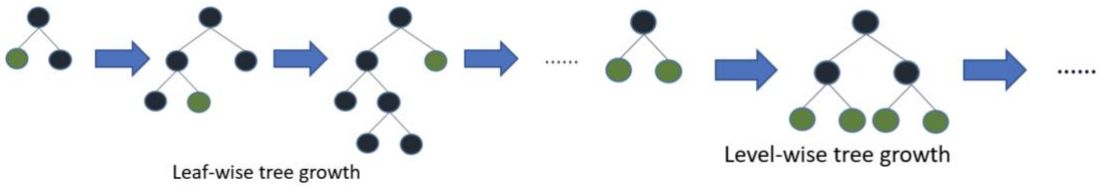
The performance with a few different settings for XGBoost is given below. All the experiments were done on subsets of training data as the time taken for training using the entire training model is very difficult. For all the other parameters, default settings were used.

Depth	Number of Estimators	Train Error	Validation Error
3	500	0.0329	0.0331
3	1000	0.031	0.03155
3	10000	0.0283	0.02986
7	500	0.0284	0.0302
7	1000	0.0292	0.0312
7	10000	0.0277	0.0315
10	500	0.0252	0.0301

From the results, we can see that the best parameters for XGBoost are depth of 3 and number of estimators as 10000. The same set of parameters in Random Forest Regressor would take a huge amount of time. This clearly explains why Gradient boosting methods are much more optimized when compared to vanilla tree-based methods. We can also see that, when the depth increases the model begins over-fit. When the depth is 10, even with lesser number of estimators, the train error is 0.0252 but the validation error is higher than that of the error obtained with a depth of 3. Hence, these are important parameters for preventing over-fitting and have to be carefully chosen.

Light Gradient Boosting (LGBM):

Light GBM is a unique boosting algorithm. This algorithm is also based on building decision trees. But the method in which the trees are built is totally different from the other boosting techniques, the main difference being the growth of trees in horizontal direction rather than the vertical directions like in other algorithms. The following figure can help us understand this better:



LGBM Other boosting methods

From the figure above, we can see that the splits are based on the leaf nodes. It will choose the leaf where the gradient loss is maximum and perform further splits on it to reduce the error. This is the main difference between LGBM and other boosting technqiues. The main advantages of LGBM is that the it is very light and requires less computational power. It can be implemented on parallel processing techniques and has GPU compatibility as well. The working apart from the split on the leaf node is similar to that of the other boosting techniques. There are many parameters in LGBM.

We will see some of the important parameters that have not been previously explained in the other 2 tree based methods.

1) Bagging Fraction: This parameter decides the fraction of datapoints to be taken for each iteration
2) Number of leaves: This parameter decides whether to continue splitting on this node, based on the number of datapoints that node may already have. This parameter is mainly used to prevent over-fitting.
3) Max depth: This controls the depth of each tree. After a tree is built, it is pruned to prevent over-fitting.
4) Lambda: This determines the regularization value for the weights. Ranges between 0-1.
5) Learning Rate: This is factor that is multiplied with error gradient values
6) Feature fraction: This decides the number of features for each tree

These are some of the important parameters in LGBM. In this project, most of the parameters are default values except for some parameters which are chosen based on different experiments.

Reason to choose LGBM: Very light and fast when compared to XGBoost. It is very different from other boosting techniques, and very fast to train. It can handle large amount of data and at the same time take lesser memory.

The following results were obtained with different settings of LGBM Regressor. For the experiments below, the number of estimators used were default. Learning rate was 0.05, max depth was 6, and the number of estimators was 1000.

Maximum Number of Leaves	Bagging Fraction	Train Error	Validation Error
25	0.5	0.0344	0.0350
25	0.7	0.0344	0.0351
25	0.9	0.0344	0.0350
30	0.5	0.03386	0.03485
30	0.7	0.03390	0.03476
30	0.9	0.03387	0.03489
35	0.5	0.033347	0.03487
35	0.7	0.03338	0.03482
35	0.9	0.033398	0.03488

We can see that the best parameters for LGBM are maximum number of leaves being 30 and bagging fraction of 0.7. We can also see that the performance is very stable with different numbers for these parameters. Hence, they can be chosen in any of the ranges. To consider the model complexity and optimal features, the parameters mentioned above are chosen.

The following results were obtained with different number of estimators. The maximum number of leaves was 30 and the bagging fraction was 0.7. These parameters are chosen carefully to reduce overfitting.

Number of Estimators	Train Error	Validation Error
1000	0.03929	0.0399
2000	0.0376	0.0388
5000	0.0356	0.0380
10000	0.0338	0.0378
20000	0.031	0.037

The best number of estimators for LGBM is 10000. We can see that, for 20000 estimators the train error is going down, but the validation error remains the same. We can go with either 10000 estimators as well as 20000 estimators. But the complexity of the model is doubled if 20000 estimators are used. Hence, we will choose 10000 estimators as the best parameter.
Thus, these are the regression models used in this project. Some of the models perform very well when compared to the others. We will study that in the next section.

Model Selection and Comparison of Results:

We have seen the performance of different models with different parameters (though we cannot call that cross validation, a number of parameters were tried to choose the best set). The best setting for each model has to be chosen based on the validation error. Once the best model is chosen for each setting, we can test it on the test set we split in the beginning. In this section we will evaluate the different models based on their performances on the test set we created. We will use the best model from this section in the final results section to get the error on the unseen test data, essentially giving us the out-of-sample error.

The following table contains the validation errors of the different models with the best settings from the previous section.

Model	Parameters	Train Error	Validation Error
Linear Regression	Fit Intercept: True Normalize: False	0.0554	0.05543
Ridge Regression	Fit Intercept: True Normalize: False Lambda: 0.0001	0.05541	0.05546
MLP	Hidden Layers: 3 Neurons: (1000, 500, 100)	0.0511	0.0522
Random Forest	No. Estimators: 4000 Max Depth: None Max features: 40	0.0601	0.0609
XGBoost	Depth: 3 No. Estimators: 10000	0.0353	0.0386
LGBM	Max. Leaves: 30 Bagging Fraction: 0.7 No. Estimators: 10000	0.0338	0.0378

From the results, we have to choose the best models to get the final error on the unseen test data. The best parameters for each model were chosen based on the validation errors. When two or more parameter settings for a model gave higher accuracies, then the model with a lesser complexity was chosen. For instance, for the XGBoost model both number of estimators 20000 as well as 10000 gave the same results. In this case, the model with lesser number of estimators was chosen. With the above models with each of their parameters, we may get unreliable results because the models were trained on subsets of training data and tested on the entire training and the validation sets. This may have a problem on generalizing to the entire training dataset. Hence, we have trained these models using the entire training set and we will evaluate the results on the test data we had split in the beginning. The results on the test set are given below in the table:

Model	Train Error	Test Error
Linear Regression	0.04496	0.04507
Ridge Regression	0.04474	0.04484
MLP	0.0511	0.0520
XGBoost	0.0350	0.0367
LGBM	0.0247	0.0266

From the results, we can see that the best model is LGBM as the error on the test data is lesser than the errors obtained with the other methods. The next step is to train a LGBM model using the entire training data, and then use it to evaluate the unseen test data. The results for Random forest regressor are not given here because Random Forest takes almost 21 hours to train on the entire dataset and hence, it was not trained on the entire dataset. Anyways, all the other models work better than the Random Forest Regressor on the same unseen test dataset. The errors on the unseen test data are given in the section below. The errors on XGBoost are much higher than the error rates obtained previously using the validation sets mainly because of the subsets of data that were used for performing the validation analysis. Thus, the best model is LGBM with a lowest test accuracy of 0.0286.

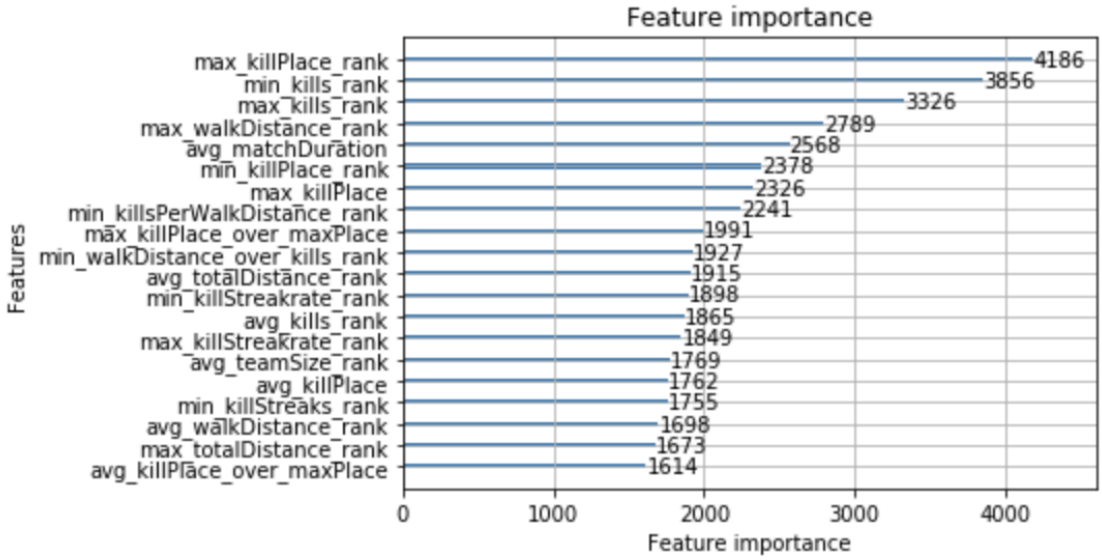
Final Results and Interpretation

From the previous section, the best model on the test created we created is Light GBM with the parameters of maximum number of leaves equal to 30, bagging fraction equal to 0.7 and number of estimators being 10,000. LGBM is the best among a number of other models tested previously. The next step is to evaluate the performance of this model on the unseen test data which is available without the output values.

The final out of sample error obtained with LGBM is: 0.0255

The best LGBM model gives an error of 0.0255 on the unseen test data. One important problem to be handled is, allocating the output values predicted for a particular group to be allocated to the corresponding members of the group. Once we predict the values for the test set, we will have a win percentage for all the groups. The same win percentage has to be allocated to all the members of that particular group. One user to subjected to 10 submissions per day, to get the evaluation of the unseen data. Also, there was a small post processing done on the output predicted values. For output values less than 0, they were changed to 0 and the output values greater than 1 were reduced to 1 as the output range for win Place percentage is between 0 and 1.

Using LGBM, we can interpret the importance of each of the features that are very important for the prediction of win place percentage. The following figure contains the importance of each feature given by the trained LGBM.



We can see that the feature max_killPlace_rank is the most important feature given by the LGBM model. This feature denotes the rank of a player based on the number of kills. Thus, players with a high Kill place will definitely have a higher win percentage. This is the idea. The feature max Kill place contains the maximum rank for a specific group and the feature max kill place rank contains the percentile values for that specific group. Thus, the most important features for the final model are obtained using the 'kill place' feature. This also confirms the fact that the Kill place feature has a high correlation with the win place percentage (shown in the previous correlation graphs). Also, we can see that the newly created features such as total distance, kills per walk distance, kill place over max place, kill streak rate and team size being present in the top 20 important features predicted by LGBM. Thus, the feature engineering has helped very much in the good prediction of win place percentage. Also, we can see that the 'ranks' (the percentile values) are the most important features when compared to the original set of features. As another modification, we could just try the percentile values instead of the original features and that would be an interesting analysis as we do not know the range of each feature.

The final out of sample error using the best model is 0.0255. The lowest error on Kaggle is 0.0167 which is around 0.08 lower than the error obtained using these set of features and this model. The main reason for this difference in performance must be due to feature engineering. Most of the leaderboard results are obtained using Gradient Boosting Methods. GBMs are optimized via Gradient Descent, and at the same time they are tree-based methods. Thus, the main advantages of GBM are as follows:

1) They can automatically do feature selection
2) They are scale invariant
3) They can capture the non-linearities in the data
4) They can also easily handle the outliers in the data

Hence, GBMs are a whole package that can be used for any type of data and at the same time the complexity is also. Of the heavy GBM algorithms such as XGBoost, LGBM, Catboost etc, LGBM seems to be lightest of all. LGBM uses highly optimized methods for splitting the trees and hence it is very fast when compared to the other methods. Also, it gives us the flexibility to control most of the hyper parameters for the tree building process unlike other GBMs. Although XGBoost can have better performances than LGBM, they are slow when compared to LGBMs and contain special design methodology for sparse data. Since the PUBG dataset is not that sparse, the effect of using XGboost is not realized. Also, splitting leaf-wise can lead to better performances when compared to the traditional algorithms. It can also lead to

Summary and Conclusions

Thus, the error of prediction is 0.0255. The Kaggle competition ends in another 2 months, and hence there is a lot of scope for improvement as the lowest error is 0.0167. The main issues must be with the feature engineering. We have also seen that the newly added features are very important for the prediction of win place percentage assuring the ideas used in feature engineering. Now, the aggregate feature used are mean, min, max and standard deviation, along with the rank for all of these features. Now, the aggregates were calculated only for all the groupIDs, the next thing can be to calculate the aggregates for all matches. Also, we can build an ensemble of models with XGBoost, LGBM etc and take the average of the outputs of these models. We can also try more sophisticated neural networks as we do not have a constraint on the data. Thus, these are some of the things that can be tried in the future. Through this project, we have understood the fundamentals of boosting algorithms and accepted the idea that a bunch of weak learners combined with different weights can actually give us a very good output. Also, boosting techniques work very well for numerical data. And the main realization from the results, is that effective feature engineering is a key for any machine learning task.

References:

1) The memory reduce function used in the code was obtained from :
https://www.kaggle.com/gemartin