PUBG KAGGLE COMPETITION

EE 660 Project Type: Collaborative (Kaggle)

Akash Mohan Das, amohanda@usc.edu

1. Abstract

Player Unknown Battle Ground (PUBG) is one of the famous games played across the world currently. The project aim is to predict the Winning Percentile of the players playing in the game. There are various skillsets possessed by the players and various game features that associate to winning the game. Huge amount of data of players who have played the game is given which needs to be analyzed and used to predict the win Percentile of the games to be played or unfinished games. I takes lot of feature processing to predict the best model.

2. Introduction

2.1. Problem Type, Statement and Goals

The problem is to predict the win Place Percentile of the players participating in the PUBG game. Given various stats about the players in the ongoing games and given the stats and win Place Percentile of the players in the already finished games, my goal is to predict the win Place Percentile of the player in ongoing games as accurate as possible and achieving minimum mean absolute error for the predictions. Since the Percentile takes a floating value between 0 and 1, it is a Regression problem. In the approach to solve the problem, one can understand what strategy is useful to stay alive for longer duration in the game and also, possibly, win the game.

Sources of difficulty are:

(1) A physical model that's inherently complicated and hard to abstract.
(2) Handling high memory data
(3) Fit Complex models and time constraints for cross validation
(4) Nonlinear behavior of the features with respect to target
(5) Significant amount of preprocessing required.
(6) Newly add features to get good correlation with target.

2.2. Literature Review (Optional)

No literature survey

2.3. Prior and Related Work (Mandatory)

-None

2.4. Overview of Approach

To begin with, I started my approach with fitting several models for the given data set without altering any of the features to check the error obtained with each model. 'Mean Absolute Error' metric was used to measure the performance between Linear Regression, Logistic Regression, Random Forest Regressor, Light Gradient Boosting, Extreme Gradient Boosting models that were fit on the train set which was split into train and validation set with split ratio 70:30. Further, the metrics of each model were used to consider just Random Forest, LGBM and XGBM for feature engineering and improving the model and correspondingly the score. Features when then analyzed using techniques like Pearson's coefficients and modified to achieve the best score for LGB model.

3. Implementation

Machine Learning problems are always challenging and hence there are various powerful tools and packages that are built for the efficient usage of memory and GPU. For my project, I have used packages like sklearn, numpy, pandas, lightgbm, xgbm that are very robust in mathematical calculations and model fitting and packages like seaborn, matplolib which are standard packages for plotting graphs. The 'mean_absolute_error' method of 'metrics' class of sklearn provides easy access to find the error for any given predicted and actual values. The different models are fit using sklearn package which needs parameters of the models to be specified that are explained in detail in further sections.

3.1. Data Set

Kaggle problem has provided the train and test data sets. The train data set has 4446966 samples with 28 features while the test data set has 1934174 samples with the same number of features. The target values for only the train data set is provided. So, we consider the error obtained after submitting the test data set predictions to Kaggle is considered the out of sample error. The features as per the training data set are described as follows:

- DBNOs - Number of enemy players knocked.
- assists - Number of enemy players this player damaged that were killed by teammates.
- boosts - Number of boost items used.

- damageDealt - Total damage dealt. Note: Self-inflicted damage is subtracted.
- headshotKills - Number of enemy players killed with headshots.
- heals - Number of healing items used.
- Id - Player’s Id
- killPlace - Ranking in match of number of enemy players killed.
- killPoints - Kills-based external ranking of player. (Think of this as an Elo ranking where only kills matter.) If there is a value other than -1 in rankPoints, then any 0 in killPoints should be treated as a “None”.
- killStreaks - Max number of enemy players killed in a short amount of time.
- kills - Number of enemy players killed.
- longestKill - Longest distance between player and player killed at time of death. This may be misleading, as downing a player and driving away may lead to a large longestKill stat.
- matchDuration - Duration of match in seconds.
- matchId - ID to identify match. There are no matches that are in both the training and testing set.
- matchType - String identifying the game mode that the data comes from. The standard modes are “solo”, “duo”, “squad”, “solo-fpp”, “duo-fpp”, and “squad-fpp”; other modes are from events or custom matches.
- rankPoints - Elo-like ranking of player. This ranking is inconsistent and is being deprecated in the API’s next version, so use with caution. Value of -1 takes place of “None”.
- revives - Number of times this player revived teammates.
- rideDistance - Total distance traveled in vehicles measured in meters.
- roadKills - Number of kills while in a vehicle.
- swimDistance - Total distance traveled by swimming measured in meters.
- teamKills - Number of times this player killed a teammate.
- vehicleDestroys - Number of vehicles destroyed.
- walkDistance - Total distance traveled on foot measured in meters.
- weaponsAcquired - Number of weapons picked up.
- winPoints - Win-based external ranking of player. (Think of this as an Elo ranking where only winning matters.) If there is a value other than -1 in rankPoints, then any 0 in winPoints should be treated as a “None”.
- groupId - ID to identify a group within a match. If the same group of players plays in different matches, they will have a different groupId each time.
- numGroups - Number of groups we have data for in the match.
- maxPlace - Worst placement we have data for in the match. This may not match with numGroups, as sometimes the data skips over placements.
- winPlacePerc - The target of prediction. This is a percentile winning placement, where 1 corresponds to 1st place, and 0 corresponds to last place in the match. It is calculated off of maxPlace, not numGroups, so it is possible to have missing chunks in a match.

Features	Data type	Range/ Cardinality

Id	String	-------
GroupId	String	-------
matchId	String	-------
Assists	Integer	0-22
Boosts	Integer	0-33
DamageDealt	Float	0.0 – 6616.0
DBNOs	Integer	0-53
HeadshotKills	Integer	0-64
Heals	Integer	0-80
KillPlace	Integer	1-101
KillPoints	Integer	0-2170
Kills	Integer	0-72
KillStreaks	Integer	0 – 20
LongestKills	Float	0.0-1094.0
MatchDuration	Integer	9-2237
MatchType	String	16
MaxPlace	Integer	1-100
NumGroups	Integer	1-100
RankPoints	Integer	1-5910
Revives	Integer	0-39
RideDistance	Float	0.0-40710.0
RoadKills	Integer	0-18
SwimDistance	Float	0.0-3823.0
TeamKills	Integer	0-12
VehicleDestroys	Integer	0-5
WalkDistance	Float	0.0-25780.0
WeaponsAcquired	Integer	0-236
WinPoints	Integer	0-2013
WinPlacePerc	Float	0.0-1.0

3.2. Preprocessing, Feature Extraction, Dimensionality Adjustment

Missing Data:

The Kaggle PUBG train data had just one missing value in the label ‘winPlacePerc’ that corresponded to the player who was the only one playing the game without enemies. Hence, this data is not useful for the model and hence the data was dropped.

Feature Engineering:

Part1:

Number of preprocessing techniques were done to increase the feature set. To achieve this, firstly the Pearson’s correlation coefficient was found for each feature with every other feature. The heat map for the same is shown in fig 3.1.

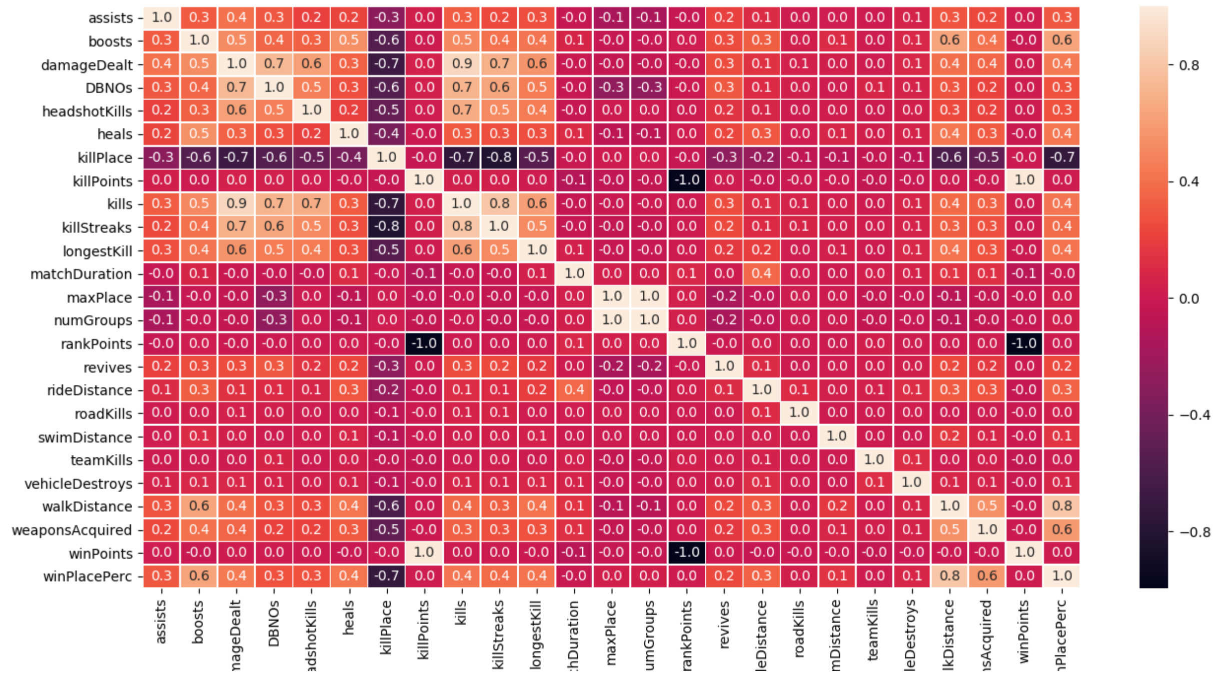


fig 3.1

From the fig 3.1, I analyzed the feature correlation with each other and further adjusted the features according to high correlation with the label. As can be seen, winPlacePerc has good correlation with Assists, Boosts, DamageDealt, DBNOs, HeadshotKills, Heals, KillPlace, Kills, KillStreaks, LongestKill, rideDistance, walkDistance, weaponsAcquired. Further, I narrowed down to top 5 high correlation with the label as shown in fig 3.2.

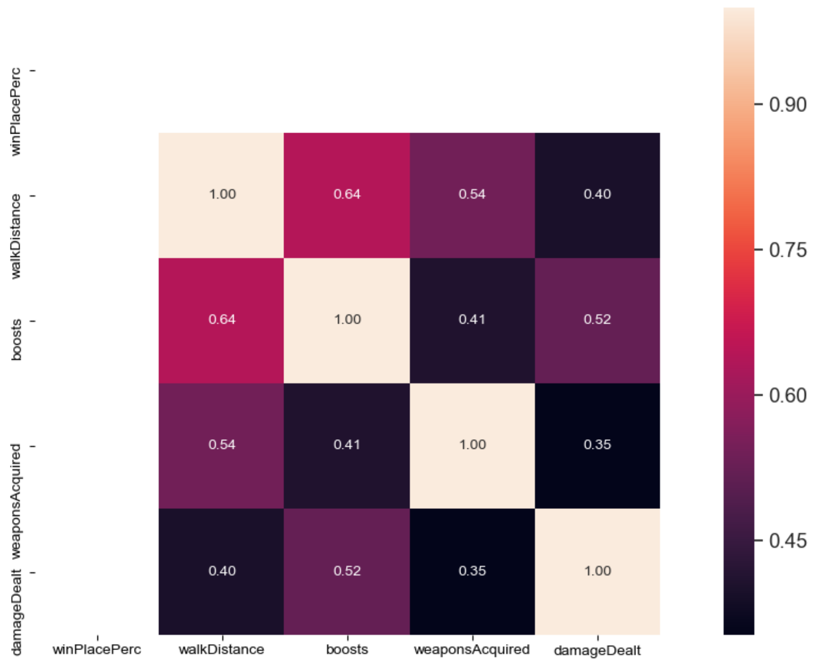


Fig 3.2

Fig 3.2 shows the correlation between top 4 features with the label 'winPlacePerc'. We find that there are several features which have higher correlation with these four features but have similar correlation with the label. So, adding them up gives a better feature measure to the model. Features that are added are:

(i) Total distance = walkDistance + rideDistance + swimDistance (having corr 0.2 and 0.3 with one another)

(ii) HealthItems = Heals + Boosts (having corr of 0.5 with each other)

(iii) Skills = HeadshotKills + roadKills

Also, I made use of the knowledge of the game to take ratios of certain features that would help decide which player is playing well and is more likely to win. For instance, walkDistance alone won't speak much about the killing skill of the player as he may not have encountered any enemy and just keep running around whereas, the ratio of walkDistance to heals says the distance he has walked for every heal he takes up. Other features added are:

(i) Headshotrate = kills/headshotkills (Player's skill in aiming head)

(ii) Killstreakrate = killStreaks /kills (Player's kill in killing number of enemies without getting shot)

(iii) killPlace_over_maxPlace = killPlace/maxPlace (How well the player is killing his enemies with respect to others)

(iv) headshotKills_over_kills = headshotKills/Kills (Negative correlation as headshotrate)

(v) distance_over_weapons = totaldistance/ WeaponsAcquired (Player's skills in distance walked with for change in weapons)

(vi) walkDistance_over_heals = walkDistance / heals (Distance walked for every heal)

(vii) walkDistance_over_kills = walkDistance/Kills (Distance walked for every kill)

(viii) killsperwalkDistance = kills/walkDistance (Negative correlation with walkDistance_over_kills)

Finally, I added the most important feature 'totalPlayersJoined' which shows the number of enemies for each player and is used to normalize other features. So, it is of interest to know the player stats depending on how many enemies he has in the game. The feature totalPlayersJoined was added by grouping together the data set by 'matchId' and count the number of players in each match.

Further, features that speak about player skills such as kills, headshotkills, killPlace, killstreak, LongestKills, DamageDealt etc were normalized with totalPlayersJoined.

Part 2:

Game stats has more weightage when it is grouped by each match and each group so that the effort of each player in the group is known. The fact that every player in the group gets similar winPlacePerc makes it more important to analyze by every matchId and every groupId. Hence, I have taken all the features which includes the features given in the data set and the features created in Part1 and grouped them by matchId and GroupId and made new features by finding attributes of the group like min, max, mean and median and also found rank index for each column. Furthermore, I have added match size and group size as additional two features along with their rank indexes. The overall features are now 326.

3.3. Dataset Methodology

The Kaggle data set has train and test csv files. Initially, the train set was used for selecting the model with 70:30 split into train and validation sets. The train set was used to fit the models with features as it was in the original data set. The validation set gave the error measures for each model and then LGBM was chosen as final model for its lowest error measure.

Further, the same split of 70:30 was used to check for the parameters of LGBM, specifically n_estimators for the values 100, 500, 1000, 5000 and 10000.

After the best parameter was chosen, the LGBM model was fit on the entire train data without any split and used to predict the labels for the test data.

3.4. Training Process

Linear Regression :

Linear Regression is the simplest algorithm to be trained on. The model tries to find the the best hyperplane that best fits the given data points with respect to least mean squared error. The model then predicts the new data points with minimum distance from the hyper plane. The model resulted in sufficiently larger error to not move forward with this model for further considerations.

The general code used by Sklearn is :

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

All the default parameters were considered for fitting the model.

Logistic Regression :

Logistic Regression is usually preferred in the problems where probability of success has to be predicted. Hence, we made use of this concept to predict the win probability of the player given this data. Logistic Regression model from sklearn was used to implement this. Unfortunately, the logistic regression model in sklearn doesn't take continuous valued numbers for the label features. Hence, we categorized the 'winPlacePerc' column to 0 and 1 depending on whether it was less than or greater than 1. Further, logistic regression with lasso (L1 regularization) was also implemented to check the feature importances and eliminated the least important feature to fit the model again. The error obtained was greater than the Linear regression model with lasso. Hence, this model was not made use further. The general equation for Logistic Regression is:

The general code used by the sklearn is :

LogisticRegression(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='warn'*, *max_iter=100*, *multi_class='warn'*, *verbose=0*, *warm_start=False*, *n_jobs=None*)

In our model, penalty = 'l1', C=1.0 was used. Further, model.predict_proba(X) was used on the validation set to get the probabilities of winning which was considered as predicted label and then compared to true label to get the error.

Random Forest :

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement.

Random Forest was chosen because it is a baseline model and it adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

The general code of Random Forest by sklearn is :

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]¶

In my model, I have used n_estimators = 1000, criterion = 'mae' , n_jobs = -1 and default parameters for the rest. The model gave a better error measure comparatively.

Extreme Gradient Boosting :

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance that is dominative competitive machine learning.

The general code using xgbm classifier is :

class xgboost.XGBRegressor(*max_depth=3*, *learning_rate=0.1*, *n_estimators=100*, *silent=True*, *objective='reg:linear'*, *booster='gbtree'*, *n_jobs=1*, *nthread=None*, *gamma=0*, *min_child_weight=1*, *max_delta_step=0*, *subsample=1*, *colsample_bytree=1*, *colsample_bylevel=1*, *reg_alpha=0*, *reg_lambda=1*, *scale_pos_weight=1*, *base_score=0.5*, *random_state=0*, *seed=None*, *missing=None*, *importance_type='gain'*, ***kwargs*)

The parameters given was n_estimators = 1000, learning_rate = 0.05, n_jobs = -1, nthread=3, colsampleby_tree = 0.7 and default parameters for the rest.

The model gave a better error measure compared to Random Forest and hence Random Forest was dropped for further considerations.

Light Gradient Boosting Model :

Many boosting tools use pre-sort-based algorithms (e.g. default algorithm in xgboost) for decision tree learning. It is a simple solution, but not easy to optimize. LightGBM uses histogram-based algorithms, which bucket continuous feature (attribute) values into discrete bins. This speeds up training and reduces memory usage. Advantages of histogram-based algorithms include the following:

- Reduced cost of calculating the gain for each split
- Use histogram subtraction for further speedup
- Reduce memory usage
- Reduce communication cost for parallel learning

The general code for LGBM using lightgbm package is :

*class*lightgbm.Dataset(*data*, *label=None*, *reference=None*, *weight=None*, *group=None*, *init_score=None*, *silent=False*, *feature_name='auto'*, *categorical_feature='auto'*, *params=None*, *free_raw_data=True*)

lightgbm.train(*params*, *train_set*, *num_boost_round=100*, *valid_sets=None*, *valid_names=None*, *fobj=None*, *feval=None*, *init_model=None*, *feature_name='auto'*, *categorical_feature='auto'*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval=True*, *learning_rates=None*, *keep_training_booster=False*, *callbacks=None*)

The params used was: {"objective" : "regression", "metric" : "mae", 'n_estimators':20000,"num_leaves" : 45, "learning_rate" : 0.1, "bagging_fraction" : 0.7,"bagging_freq": 10, "bagging_seed" : 3, "num_threads" : 4,"colsample_bytree" : 0.7}

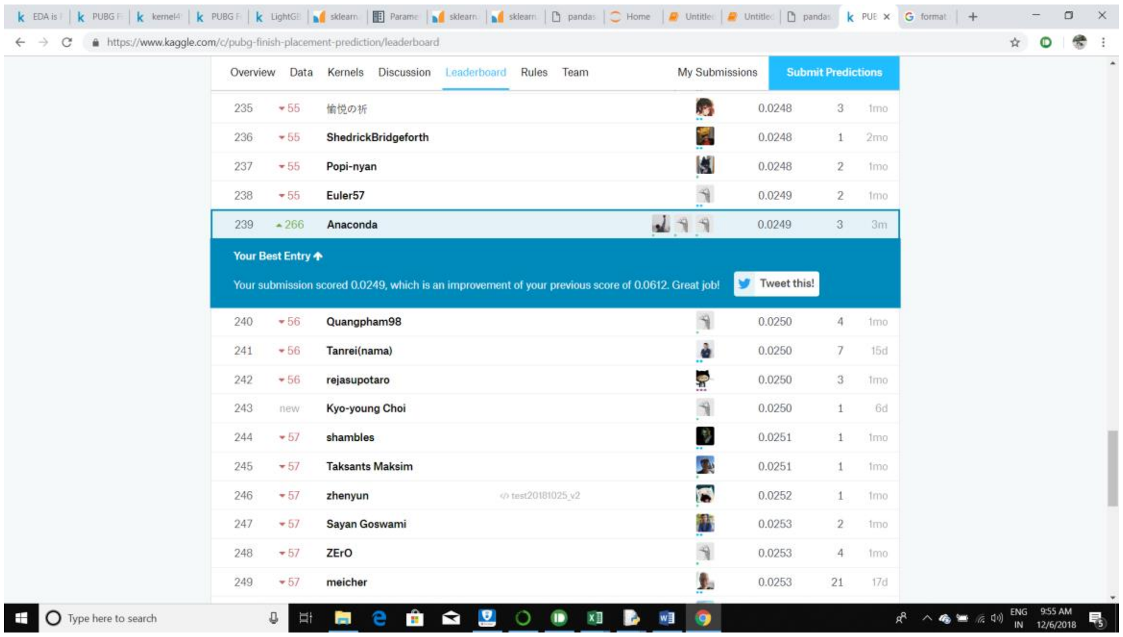
The lgbm model gave the best error measure of all and hence was considered for further considerations and cross validated for n_estimators for the values (100,500,1000,5000,10000). The best score was obtained for 10000 and hence trained the entire train data on this model to obtain predictions for test data.

3.5. Model Selection and Comparison of Results

Models	Mae Error	Model Selection
Linear Regression	0.0913	No
Logistic Regression	0.0736	No
Random Forest	0.0476	No
XGBM	0.0387	No
LGBM	0.02275	Yes

4. Final Results and Interpretation

The final model that was fit was using Light Gradient Boosting Model. The parameters given is mentioned in the section 3.5 . The out of sample performance as given by the Kaggle is 0.0249 and was ranked 239 in the Kaggle Competition.



235	▾55	愉悦の祈	0.0248	3	1mo
236	▾55	ShedrickBridgeforth	0.0248	1	2mo
237	▾55	Popi-nyan	0.0248	2	1mo
238	▾55	Euler57	0.0249	2	1mo
239	▴266	Anaconda	0.0249	3	3m

Your Best Entry ↑

Your submission scored 0.0249, which is an improvement of your previous score of 0.0612. Great job!

240	▾56	Quangpham98		0.0250	4	1mo
241	▾56	Tanrei(nama)		0.0250	7	15d
242	▾56	rejasupotaro		0.0250	3	1mo
243	new	Kyo-young Choi		0.0250	1	6d
244	▾57	shambles		0.0251	1	1mo
245	▾57	Taksants Maksim		0.0251	1	1mo
246	▾57	zhenyun	</> test20181025_v2	0.0252	1	1mo
247	▾57	Sayan Goswami		0.0253	2	1mo
248	▾57	ZErO		0.0253	4	1mo
249	▾57	meicher		0.0253	21	17d

LGBM makes use of the feature at its best to continuously create Boosted Decision Trees which is very useful in high dimension data. Increasing the features to higher dimension has helped this discern the different samples as increasing the dimension increases the distance between them and trees become more efficient. Using higher values for n_estimators could improve the model but committing to the Kaggle kernel was time consuming. Also, extracting more relevant features could further improve the model.

5. Contributions of each team member

Individual Project

6. Summary and conclusions

The hypothesis set initially consisted of 5 models : Linear Regression, Logistic Regression, Random Forest, XGBM, LGBM. The second set of hypothesis consisted of different parameters for n_estimators parameter of the LGBM model. The final model that was fit was LGBM on the feature engineering explained in section 3.2. Also, submitting this to the Kaggle Competition yielded out of sample error of 0.0249 and rank of 239 (at the time of writing report). The game could be further understood to extract more relevant features and error can be further reduced.

7. References

Referred public kernels on Kaggle.