Who wins the game? Machine learning approaches predicting PUBG players ranking

Final Report

Jingling Yang, Jingming Wei

1 Introduction

PlayerUnknown's Battle grounds (PUBG) is a first/third person shooter battle royale style game where over 90 players in a single match try to survive to the end to win the game. Each player can play solo, duos, or squads. Players parachute from an airplane to land on distinct resource points to scavenge for equipment, weapons, first-aid, and vehicles. A circle, the boundary of the safe zone will appear a few minutes after the start of the game, and people outside the safe zone will suffer chronic damage. The safe zone will shrink as the game time elapses to enforce encounters of players. Players in the game can adopt different strategy of survival. They can choose to battle against any encounters, or evade hot battle grounds and take a zigzag path approaching the safe zone.

The interesting part of the game is that there seems no dominant strategies in the game because of some random features of each single match. For example, if a player is so lucky that he lands on a spot that happens to be the exact location of the last circle (the last safe zone), he doesn't need to walk for long distance for better locations. And thus, sticking around and waiting to be placed in top 10 seems to be a good take (though less fun). For another case, if a player's landing spot is far away from the safe zone, he needs to scavenge heavily to get advantages in weapons and boosts in case he encounters some enemies who have hidden for quite a while in the safe zone. Therefore, it would be interesting to see what are the most determinant factors affecting player rankings in the game.

Typically, each player is ranked based on survival time in a match after the end of the game. This project aims to compare across different supervised machine learning methods learned from this course to find the best model that generates the most accurate predictions of PUBG player rankings. In particular, we use regularization methods including ridge, lasso, and elastic net. We also apply tree based methods including decision trees, bagging, random forests, and boosted trees. The structure of this project is as follows. Section 2 describes our empirical strategies.

Section 3 presents our analysis and results. Section 4 discusses the possible underlying conceptual ideas leading to the results and limitations of this project. Section 5 concludes.

2 Empirical strategy

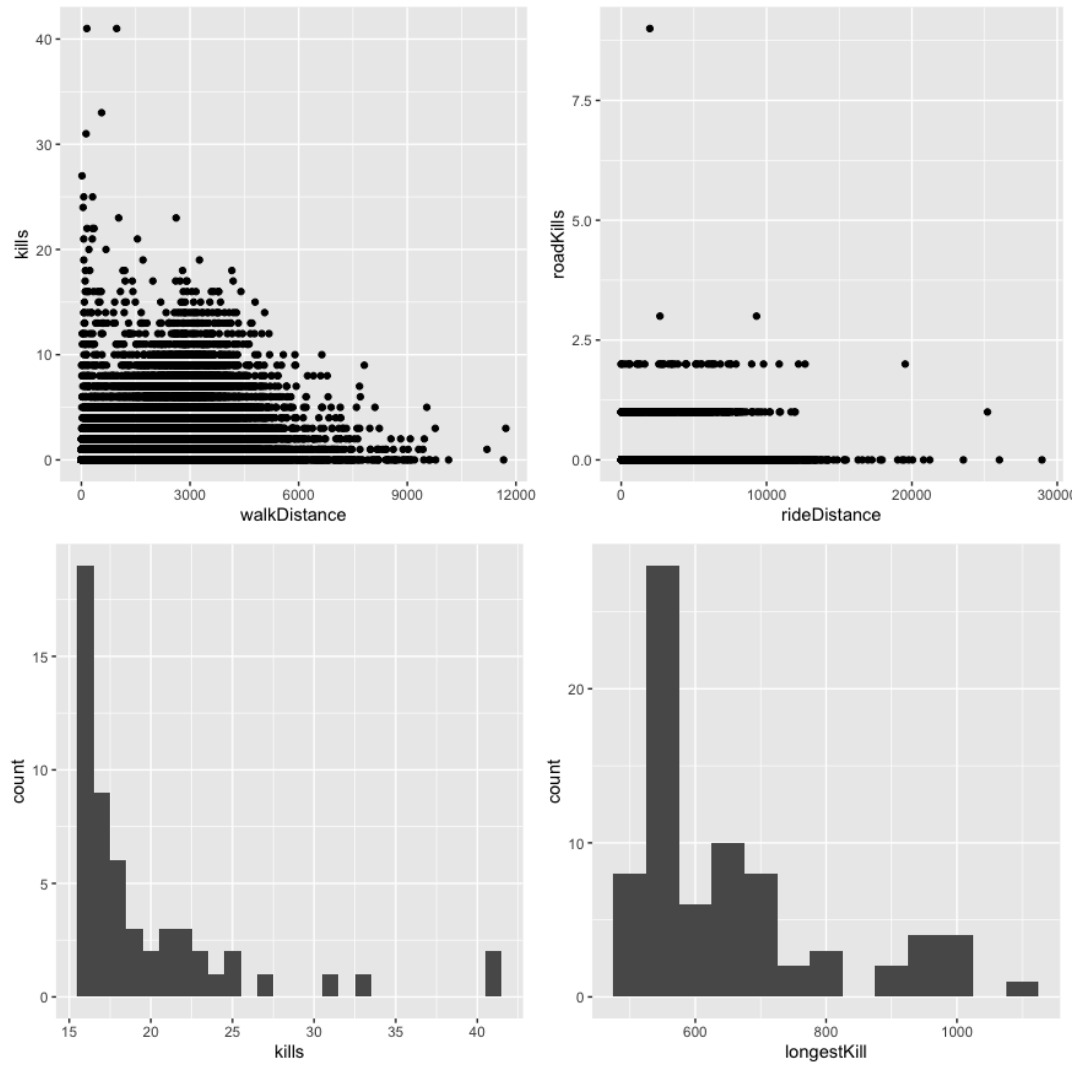
2.1 Data description

The data titled *PUBG Finish Placement Prediction* is extracted from Kaggle. The data provider scraped the data from the PUBG API and well compiled them into two data sets, a train set with 1,934,174 observations and a test set with 1,048,575 observations. However, since it takes "forever" to train such large number of observations for our tree based methods, we take a sub sample of 1,000 matches from the original data set, yielding 185,731 observations in total. Each row of the data set contains relevant statistics of each player in a single game. Table 2 in appendix introduces the columns of the data set, where we use non-ID features and response variable to train and score models. Our target variable of interest is "winPlacePerc", the player rankings scaled from 0 to 1 by number of players in the game. 1 corresponds to the first place, and 0 corresponds to the last place in the match.

2.2 Preprocessing

First, we drop the missing values in the data set. Since the data source is from PUBG developer API, we get most data in good shape. There is only one row of the original large data set with all missing values. Dubiously, this row belongs to a match that contains only one player. So, we drop this observation and draw our sample described above. Also, we classify the messy "matchType" categorical variables which contains 14 detailed matchTypes into three basic categories: "solo", "duo", and "squad".

Secondly, there might be anomalies in PUBG games. For example, there might be some custom games, where players can add AI players (always rookies), or cheaters in the match (using third party software destroying the game balance). We try to detect and exclude those observations from our analysis if they strongly influence our results. Specifically, we define the following four categories as "cheating": 1) many kills without moving; 2) Many road kills while driving a very short distance; 3) suspiciously high number of kills; 4) suspiciously long range kills. From the



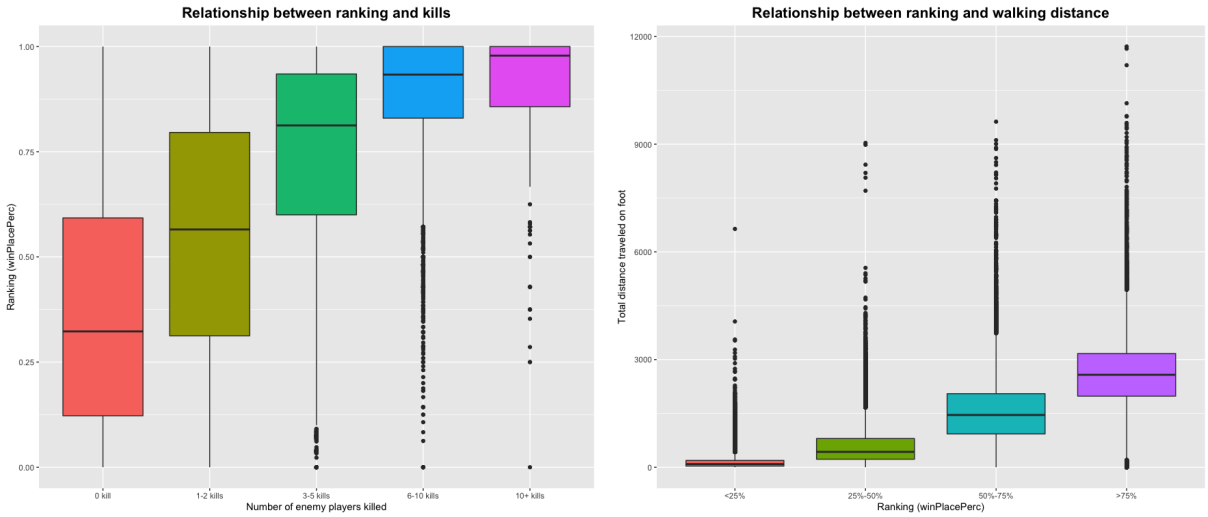
above graphs, we can see that the number of anomalies are actually limited. The number of outliers (> 20 kills when walkditance=0 in the first graph; the players in the upper left corner in the second graph) are below 20 for each. Then, we keep the rows unchanged since we cannot rule out the possibility that if the skills of players are normally distributed, then we might mistakenly classify some highly skillful players into cheaters.

Thirdly, we do two feature space transformations to better capture the player's ability (or strength) during a match. The first set of transformations is based on the variable "Nplayer" (number of players in the match) that we generate, and the second set of transformations is based on the feature "walkDistance".

The motivation for the first change is the following. it's easier for players to do more kills or damage in a 100-people game than in a 60-people game. As a result, even if we find more damage dealt is associated with a better ranking, we don't know if it comes from players' true

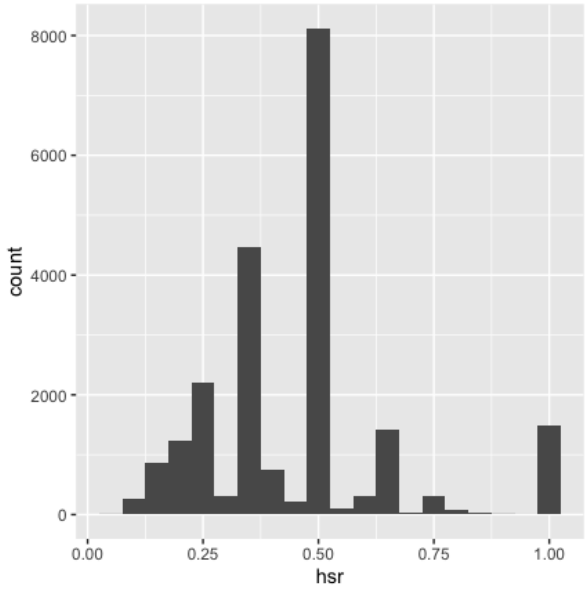
abilities or natural advantages of more players available. Therefore, we add a new variable, the number of players ("Nplayer"). First, this feature enables us to transform other variables to address the above concerns. In particular, we normalize some features by the number of players. We divide the values of the following features by the number of players: 1) the number of kills, 2) total damage dealt; 3) percent of total knocks in the match each player gets. By taking the measures on a per person unit, we avoid possible misinterpretations described above. Secondly, this variable captures the competitiveness of the game to some degree though there are probably other unobserved heterogeneity across matches. More players in a game might increase the difficulties of getting a higher rank in general.

For the second set of modifications, we can see from the graphs below that player rankings are highly correlated with both the walking distance and the number of kills. Such high correlation is not surprising since players are entirely ranked by survival time. For most cases, players need



to travel closer and closer to the safe zone. Traveling longer actually proxies longer survival time. Similarly, as safe zone gets smaller, players are forced to battle against each other. Thus, a larger number of kills are typically associated with higher rankings. We actually first feed these features into the training but yield uninteresting results described above. So we do the following transformations that convert these (almost) proxies of our response to the variables that reflect players' efficiencies in the game. We divide "kills" (number of kills), "boosts" (number of boost items used), and "heals" (number of first-aids used) by the total walking distance. For players walking zero distance (die immediately after landing), we assign their per distance measures as 0's.

An additional feature we add is "headshot rate", calculated as the number of headshot kills divided by the number of kills. We assign 0's to the players with no kills at all. On one hand, this feature is especially an important measure of skills in shooting games since making a headshot kills are extremely difficult. On the other hand, if some players have outstanding headshot rates with a large number of kills, we might suspect that those players are cheating. 83.2% players in our data have zero headshots, where 68.2% of them commit 0 kills. If we restrict our attentions to players with at least 2 kills, we can see from the graph below that most people have headshot rates less than 50% while less than 2000 players have headshot rates of around 1. Of those players, only 40 of them commit kills more than 1. Therefore, this can be another suggestive evidence that cheating is not a serious issue in our analysis.



Finally, we drop the variables that become redundant after the feature transformations and ID variables. Specifically, we drop "Id", "groupId", "matchId", "killPlace", "walkDistance", "kills", "damageDealt", "DBNOs", "heals", "boosts", and "headshotKills". Our cleaned data has 185731 observations with 27 columns (including response). We then scale the features and the response before feeding the data into training models. Though scaling is not actually required for our tree-based models, we keep the units of the response consistent for comparability. We split 70% of the data into the train set and the other 30% into the test set.

We pick the linear methods and tree-based methods for two reasons. First, there is a trade-off between interpretability and prediction accuracy. Though our main goal is to make accurate predictions, we are still curious about the contributions of each features to player rankings. Linear

models enable us to explore the relationship between each feature and the response in a straight forward way. Secondly, without knowing the true data generating process, we want to reduce bias by adding more flexibility to the model. While the parametric linear methods assume a linear structure between the features and the response, tree-based methods don't make any explicit assumptions about their functional relationship. Admittedly, because of the variance bias trade-off, we are unsure about which set of methods performs well. We expect that tree-based methods have higher variance and lower bias due to higher flexibility, while linear regularization methods have reverse properties.

2.3 Linear regularization methods

We first apply ridge, lasso, and elastic net methods to perform training. Specifically, these regressions contain all of our features using a technique that constrains or regularizes the coefficient estimates, or equivalently, that shrinks the coefficient estimates towards zero. We drop OLS-based methods including subset selection, forward and backward selections because it's shown that with the number of features very large, the OLS estimator has particularly poor properties, even if the conditional mean of the outcome given the covariates is in fact linear. Even with number of features modest in magnitude, the predictive properties of the least squares estimator may be inferior to the estimators that use some amount of regularization (Athey and Imbens, 2019).

A. Ridge regression

For ridge regression, we try to estimate $\hat{\beta}^R$ which minimizes

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2,$$

where $\lambda \geq 0$ is the shrinkage rate. As $\lambda \to \infty$, all the coefficients approach 0. We use 10-fold cross validations to tune λ on the train set. After retrieving the best λ that generates the lowest cross validation mean squared error, we fit the corresponding model to the entire train set and score the model using the test set by MSE.

B. Lasso regression

Here we estimate $\hat{\beta}^L$ which minimizes

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta|.$$

Since Lasso changes from β_j^2 to $|\beta_j|$ in the penalty term, we can see some of the coefficients exactly reach 0's for some large value of λ. Therefore, Lasso automatically picks the variables of importance in some sense. Similarly, we perform 10-fold corss validations to tune the shrinkage rate.

C. Elastic net

Elastic net is a mixture of the previous two models. It combines the penalties of ridge and lasso to get the best of both worlds. For an α strictly between 0 and 1, and a nonnegative λ, elastic net minimizes

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\left(\frac{1-\alpha}{2}\sum_{j=1}^{p}\beta_j^2 + \alpha\sum_{j=1}^{p}|\beta|\right).$$

Here, we use 10-fold cross validations to tune α and λ.

2.4 Tree-based methods

A. Regression Tree

We first make a skeleton regression tree model T_0 by applying the recursive binary splitting approach. However, this approach may overfit the data and perform poorly on test set performance. To overcome the overfitting, we apply cost complexity pruning algorithm. In this strategy, we want to find a subtree $T \subset T_0$ which minimize

$$\sum_{m=1}^{T}\sum_{x_i \in R_m}(y_i - \hat{y}_{R_m})^2 + \alpha|T|$$

Here, α is the tuning parameter, controlling the trade-off between the subtree's complexity and its fit to the training data. Therefore, the choice of α is crucial to the model performance.

So first, we apply the cost complexity pruning to the large tree in order to obtain a sequence of

best subtrees, as a function of α. Then we use 10-fold cross-validation to choose the best α^*, which minimizes the average error. At last, we return the subtree corresponding to the optimal α^* as our final regression tree model.

B. Bagging

The regression tree discussed above suffered from high variance. Bagging is a general-purpose procedure for reducing the variance of a statistical learning methods. Specifically, we construct B regression trees using B bootstrapped training sets, and average the resulting predictions, that is,

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

Here, the trees are grown deep, and are not pruned. Bagging has been demonstrated to give substantial improvements in accuracy by combining together hundreds of trees into a single procedure.

C. Random forests

Because in the bagging method, we use the bootstrapping to generate B training sets, the trees built on these training sets are actually correlated. Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees.

Specifically, in this algorithm, we also build a number of decision trees on bootstrapped training samples, as in bagging. But each time when we are considering a split in a tree, a random sample of m features is chosen as split candidates from the full set of features. As usual, we also choose $m = \sqrt{p} = 5$ in this case.

D. Boosting

In the boosting, we also first generate B bootstrapped training sets. And the trees are grown sequentially, and each tree is grown using information from previously grown trees. Specifically, the boosting algorithm works as follow,

1. Initially, set $\hat{f}(x) = 0$ and $r_i = y_i, \ \forall i$.

2. For each training set $b = 1, 2, ..., B$, repeat the following steps: (a) fit s tree $\hat{f}^b$ with d splits; (b) update $\hat{f}$: $\hat{f} \leftarrow \hat{f} + \lambda \hat{f}^b(x)$; (c) update r_i: $r_i \leftarrow r_i - \lambda \hat{f}^b(x)$.

3. Output of the boosting model: $\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$.

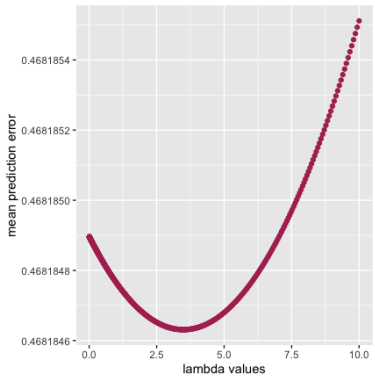
Therefore, in the boosting, there are three parameters we need to tune: the number of trees B, the learning rate λ, and the interaction depth d. As before, we use 10-fold cross validation approach to choose the optimal parameters.

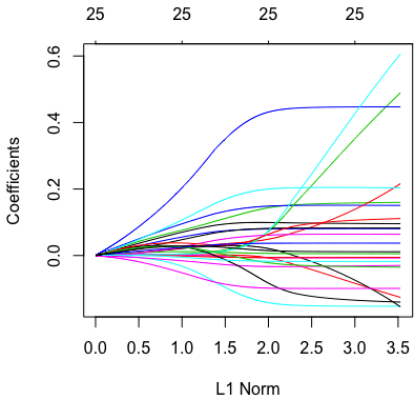
3 Analysis & results

Since we make several major changes to our feature space and address the error of not scaling the response, the results now look quite different from those during our presentation.

A. Ridge regression

For both of our ridge and lasso regressions, we perform a grid search of λ from 0.01 to 10. We plot the cross validation MSE for each iteration of λ as shown below. The best value of λ in this case is $\lambda = 3.496$. We also plot coefficients against L1 norm to see the shrinkage speed of each feature by setting the range of $\lambda \in [0.001, 1000]$. As we can see, some lines approach pretty slow to 0, suggesting that their corresponding features have significant impacts on player rankings. We further rank the coefficients of the variables associated with the best λ by absolute values





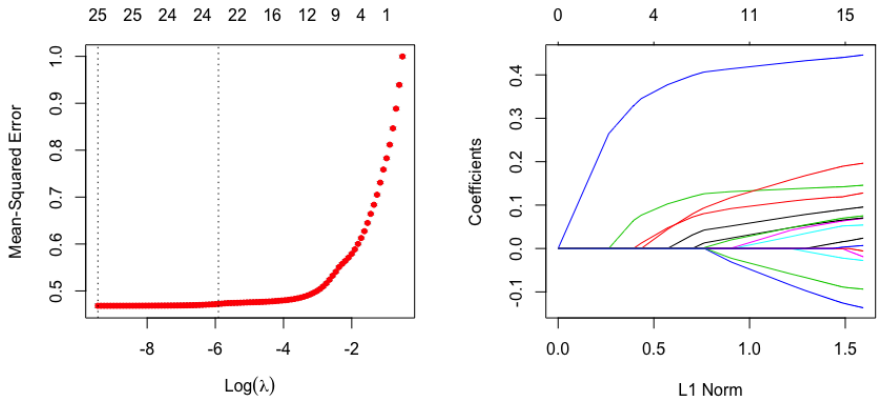
(in Appendix Table 3). We can see that the most important variables are "weaponsAcquired", "rideDistance", "longestKill", "killStreaks", "assists", "revives", and "damage_ave". The above

results suggest that we might need to pick up and switch weapons frequently to cope with different situations in the game. Also, finding a vehicle and driving in the car seems safer.

The test MSE for the ridge regression is 0.6999. This seems pretty large given the original ranking is between 0 and 1. But actually, our response is now between −1.546 and 1.714 after scaling. Therefore, the prediction result seems not too bad.

B. Lasso regression

Similarly, we first perform grid search of λ values over the same range. But it turns out lasso hits a corner solution of λ. So, we restrict the grid into much smaller range as suggested by the "cv.glmnet" function. The cross validation results are shown below. The best λ is 0.00007917509.

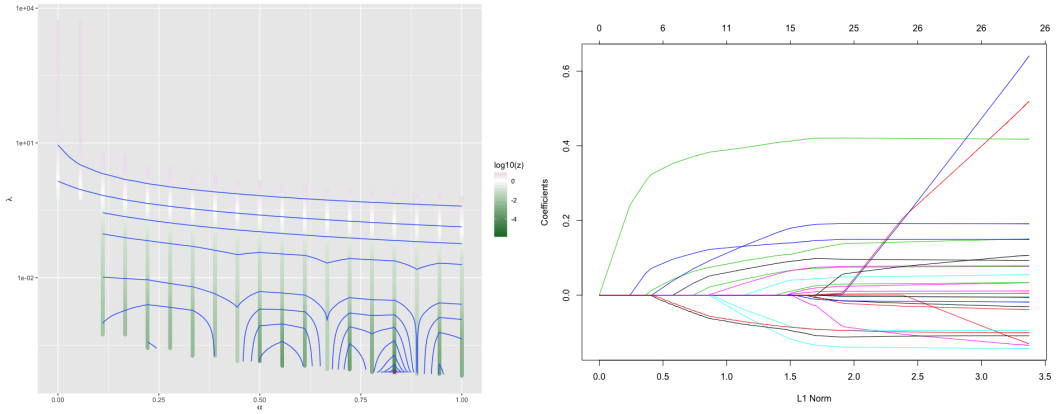


As expected, some of the coefficients drop to 0 after some iterations. The most important variables (ranked by absolute values) are "winPoints", "rankPoints", "weaponsAcquired", "numGroups" "rideDistance", and "maxPlace". Consistent with our previous results, "weaponsAcquired" and "rideDistance" appear to be critical predicting players' rankings. The test MSE is now 0.4902, smaller than the one generated by ridge.

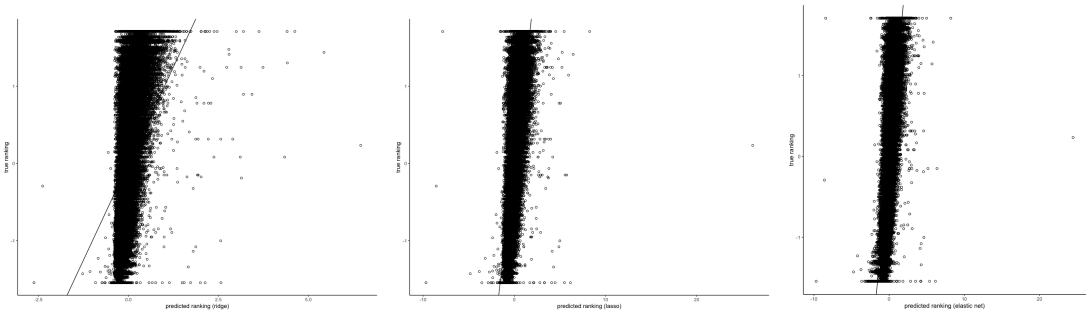
C. Elastic net

For the elastic net, we perform a combination of grid and random search using the function "ensr" to 10-fold cross validate the two parameters α and λ. To save computational power, we keep the grid of α to the default range from 0 to 1 with 18 steps and randomly search more than 940 λ's from the function's default lists. We first plot the level curves to find the best combination of the two parameters. The blue lines represent indifference curves of cross validation errors. The green

vertical lines with different shades represent different level of cross validation errors. Therefore, any points on the same blue line should cross green lines of the same darkness. The red point represents the best combination of our two parameters, where $\alpha = 0.8333$ and $\lambda = 0.00007917479$. Then we plot coefficient graph corresponding to the best α. Since α is close to 1, the graph looks



more similar to the lasso's, where some coefficients shrink exactly to 0's. The test MSE of elastic net is 0.4563186, which is better than both ridge and lasso with no surprise since the tuning parameters include cases for both lasso and ridge.

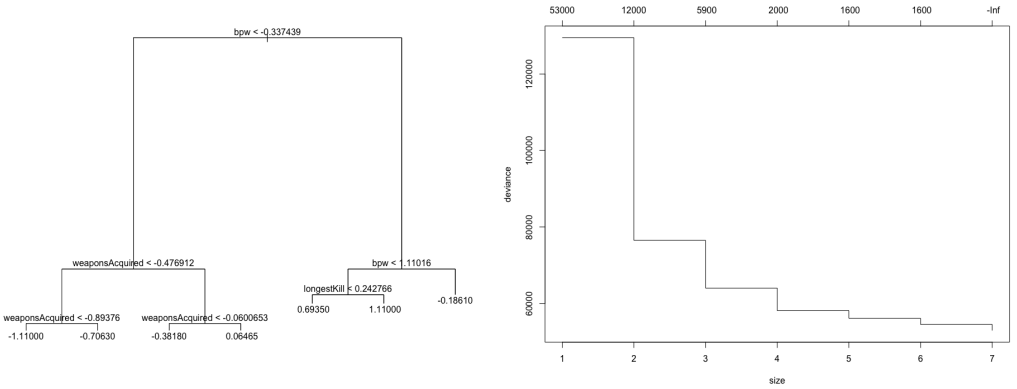


Visually, elastic net yields better calibrations than the previous two methods as we can see the observations fit tighter to the line in the third graph above.

D. Regression tree

We first grow full tree from the train set. The resulting tree has 7 nodes (leaves). Note that only three variables are included for the tree, suggesting that these three variables have strong predictive powers on player rankings. Since the tree is pretty neat already, we don't need to worry too much about over fitting. But to testify our belief, we further perform 10-fold cross validation to see if we can get a even simpler tree by cost complexity pruning. The result suggests we can

keep the original tree since the number of nodes=7 yields the smallest deviance. From the tree, we can see that our new variable boost items used per unit of walking distance has the most impact on player rankings. "weaponsAcquired" again shows up as an important factor. Interestingly, "longestKill" becomes an internal node of the tree. Intuitively, the longest distance kill measures a player's ability of sniping. It's extremely hard for a player in the game to hide in a good location



and snipe accurately at a person far away. Good snipers can always rank higher in the game.

The test MSE of the regression tree is 0.40784. The model performs even better than the elastic net.

E. Bagging

For bagging, we choose the tree size as $B = 500$ without any validation methods. Given that our data has more than 100,000 observations, it takes so long to train the data even for a single value of B. Iterating over many values of B are computationally heavy. When we grow a tree inside the iteration (one of the 500 trees), we consider all of our 26 features for node splitting. As it turns out, the model performs pretty well. The test MSE of bagging is 0.2663795, a huge improvement compared with regression tree. Below (after the random forest section) is the true response vs. predicted response of the test set. As we can see, vast majority of observations stick closely to the 45 degree line.

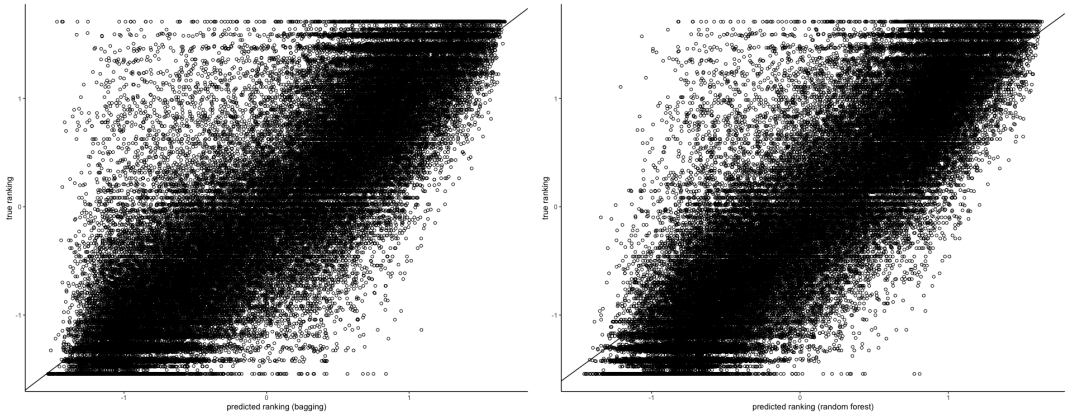
We display the variable of importance for bagging in table 1 of appendix. We use two measures of variable importance, "%Increase in MSE" and "increase in node purity". The first measure is computed from permuting OOB data: For each tree, the prediction error on the out-of-bag portion of the data is recorded (MSE in our case). Then the same is done after permuting each predictor variable. The difference between the two are then averaged over all trees, and normalized by the

standard deviation of the differences. If the standard deviation of the differences is equal to 0 for a variable, the division is not done (but the average is almost always equal to 0 in that case). The second measure is the total decrease in node impurities from splitting on the variable, averaged over all trees.

We can see from Table 4 in the Appendix that, consistent with our previous estimates, "bpw", "longestKill", and "weaponsAcquired" continue to be important. Another two kills relevant features also show up in the top 5 entries.

F. Random forest

The only difference between random forest and bagging is that at each split of the node, we consider a proper subset of the original feature set. As suggested by many literature, we randomly take $\sqrt{26} \approx 5$ features from the original set. Ideally, we would get more robust results if we can cross validate both "mtry" (number of features to consider at each split) and "B" (number of trees to grow). We fail to do so for similar reason discussed above.

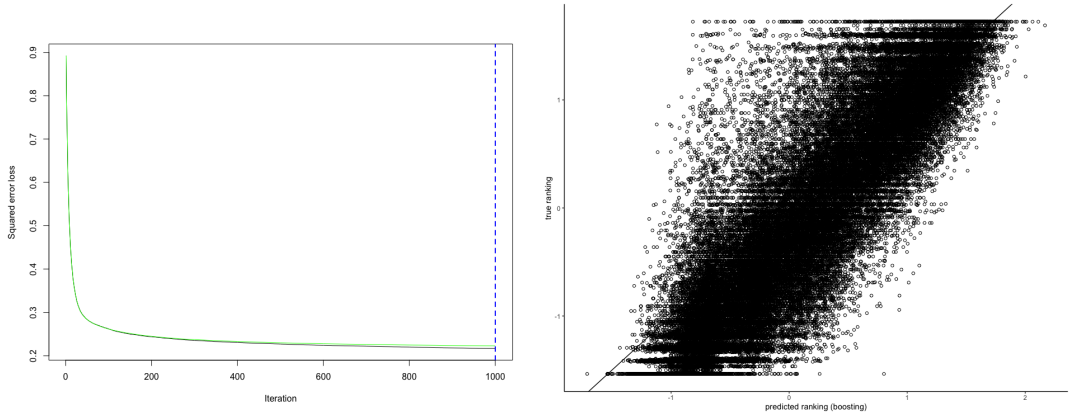


The test MSE for the random forest method is 0.261126, slightly smaller than that of bagging. As we can see from the above graph, the calibration results of both bagging and random forests are similar.

G. Boosted tree

For boosted trees, we apply grid search for all of the three hyper parameters. We search the learning rate $\lambda \in \{0.001, 0.01, 0.1\}$, interaction depth in $d \in \{1, 2, 3, 4\}$, and number of trees in $B \in 1 : 1000$. By 10-fold corss validations, we get the best parameters as $\lambda = 0.1, d = 4, B = 1000$. Fixing the optimal values of λ and α, we plot squred error loss by each iteration (B). As we can see

from the graph, this is actually a corner solution, which suggests we might not hit the best number of trees. But given that the line is so almost parallel to the x-axis, the marginal improvement of the model performance would be limited. The test MSE is 0.2744383, which is quite similar to those of random forest and bagging.



4 Discussion

If we compare across test MSEs across all the models (table below), we can see that tree-based models outperform linear regularization models. One possible reason is that the underlying relationship between features and the response might not be linear. Then, linear estimators in general will yield more biased results.

Table 1: **Test MSEs across different methods**

	Tuned parameter(s)	Test MSE
Linear regularization methods		
Ridge regression	$\alpha = 0,\ \lambda = 3.49578$	0.6999
Lasso regression	$\alpha = 1,\ \lambda = 0.00008$	0.4902
Elastic net	$\alpha = 0.83,\ \lambda = 0.00008$	0.4563
Tree-based methods		
Regression tree	$d = 6$	0.4078
Bagging	$B = 500$	0.2664
Random forests	$B = 500$	0.2611
Boosting	$d = 4,\ B = 1000,\ \lambda = 0.1$	0.2744

Notes: d is the number of splits, which means there are $d + 1$ terminal nodes in the trees. B is the number of trees. λ is the shrinkage rate. α is the mixing parameter between ridge ($\alpha = 0$) and lasso ($\alpha = 1$).

Random forests and bagging perform the best even without cross validation. This is probably due to the nature of the algorithm. Athey and Imbens (2019) concludes random forests require

relatively little tuning and have great performance out-of-the-box. Random forests are particularly effective in settings with a large number of features that are not related to the outcome, that is, settings with sparsity. This is true in our setting, where only few variables seem important determining player rankings.

Even though the models in this project perform well overall, there are many limitations of this project. First, the feature space is pretty limited. We actually tried to recover more information for each match and player from PUBG API (Python scraping code is attached in the appendix), but we failed since the limit is 10 requests/minute. But it would be more interesting to have history information of a player and use his/her previous experience to make predictions on the next match. The biggest drawback of the features in our data set is that all the features are realized values after a match. Thus, we cannot make conclusions about how a player would perform in his/her future games. Secondly, the parameter tuning for the tree-based methods is not thorough. The range of search is somewhat arbitrary to save computational costs, and thus we can't rule out the possibility that there might be other good combination(s) of parameter(s) generating even better results.

5 Conclusion

This paper applies seven machine learning methods to predict PUBG player rankings based on in game features. Though there are possible cheaters in the game, we conclude that they will not affect too much of our analysis. Due to the possible non-linearity between the features and the response, tree-based methods generally perform better than linear regularization methods.

Though this project cannot be used to make future ranking predictions, results do suggest some game strategies for future players. For example, "weaponsAcquired" appears to be important across models. Then it might be a good idea to scavenge more weapons, rather than to stick around one place. Also, it might be a good idea for players to practice sniping skills to secure a better ranking.

Cheating can be a great deal in game settings since it destroys "legal" players' experience and harms gaming companies' profits. Though we mention a bit in our preliminary analysis, it would be interesting to extend this project to detect possible cheaters in a more systematic way using unsupervised machine learning methods when more features are available.

6 Appendix

6.1 Variable dictionary

Table 2: **Summary statistics**

	Type	Meaning
Identificators		
ID	factor	Player's ID
group ID	factor	ID to identify a group within a match
match ID	factor	ID to identify a match
Responses		
winPlacePerc	numeric	Percentile winning placement, where 1 corresponds to 1st place, and 0 correspond to the last place in the match.
Preditors		
assists	interger	Number of enemies this player damaged that were killed by teamates
killPoints	integer	Kills-based external ranking of a player
killStreaks	integer	Max Number of enemies killed in a short amount of time
longestKill	numeric	Longest distane between the killer and the victim at time of death
matchDuration	integer	Duration of match in seconds.
matchType	factor	the game mode: squad, duo, or solo
numGroups	integer	Number of groups in the match
rankPoints	integer	Elo-like ranking of a player
revives	integer	Number of times the player revived teammates
rideDistance	numeric	Total distance traveled in vehicles measured in meters
roadKills	integer	number of kills while in a vehicle
swimDistance	numeric	Total distance traveled by swimming measured in meters
teamKills	integer	Number of times the player killed by a teammate
vehicleDestroys	integer	Number of vehicle destroyed
weaponsAquired	integer	Number of weapons picked up
winPoints	integer	Win-based external ranking of the player
Nplayer	interger	Number of players in a match
kill_ave	numeric	kills/Nplayer
damage_ave	numeric	damageDealt/Nplayer
dbno_ave	numeric	DBNOs/Nplauer
bpw	numeric	boosts/walkDistance
hpw	numeric	heals/walkDistance
kpw	numeric	kills/walkDistance
hsr	numeric	headshotKills/kills
Dropped variables		
boosts	integer	Number of boost items used
damageDealt	numeric	Total damage dealt. (Self inflicted damage is subtracted)
DBNOs	integer	Number of enemy players knoecked
headshotKills	integer	Number of enemy players killed with headshots
heals	integer	Number of healing items used
kills	integer	Number of enemy players killed
walkDisrance	numeric	Total distance traveled on foot measured in meters

Notes: The variables in the "Dropped variables" are used for feature space transformation and some preliminary analysis but are dropped before training.

6.2 Ridge, lasso, elastic net coefficients

Table 3: **Coefficients in the linear regularization methods**

	Ridge	Lasso	Elastic net
assists	0.0450	0.0953	0.0932
bpw	0.0164	0.0371	0.0336
damage_ave	0.0340	0.1119	0.1070
dbno_ave	0.0261	-0.0352	-0.0386
hpw	-0.0004	-0.0177	-0.0185
hsr	0.0228	0.0112	0.0114
kill_ave	0.0292	-0.1410	-0.1344
killPoints	0.0019	-0.1510	-0.1318
killStreaks	0.0512	0.1602	0.1506
kpw	-0.0256	-0.0986	-0.0943
longestKill	0.0613	0.1503	0.1493
matchDuration	-0.0093	-0.1522	-0.1430
matchType	-0.0083	-0.0288	0.0331
maxPlace	0.0112	-0.1706	-0.0311
Nplayer	0.0110	0.0638	0.0547
numGroups	0.0118	0.2309	-0.0008
rankPoints	0.0022	0.5780	0.5092
revives	0.0358	0.0807	0.0771
rideDistance	0.0614	0.2032	0.1913
roadKills	0.0028	-0.0073	-0.0067
swimDistance	0.0271	0.0824	0.0785
teamKills	0.0005	-0.0053	-0.0052
vehicleDestroys	0.0095	0.0052	0.0053
weaponsAcquired	0.1114	0.4466	0.4177
winPoints	0.0022	0.7193	0.6319

6.3 Variable relative importance

Table 4: **Bagging**

	%Increase in MSE	Increase in node purity
bpw	0.29881	61008.2750
longestKill	0.28546	4122.1271
kpw	0.27694	6557.3114
weaponsAcquired	0.22117	17562.9379
kill_ave	0.21219	2961.7773
rideDistance	0.11006	4393.6831
hpw	0.09942	2268.8489
matchDuration	0.05361	6803.9525
damage_ave	0.04766	3667.2386
numGroups	0.04670	2942.7664
maxPlace	0.03895	2156.0980
winPoints	0.02591	1921.6599
killPoints	0.01790	1794.4495
swimDistance	0.01512	852.8461
killStreaks	0.01410	155.4487
rankPoints	0.01302	3445.8873
dbno_ave	0.01094	886.5663
Nplayer	0.00951	2227.6367
assists	0.00878	580.8848
matchType	0.00456	162.9328
revives	0.00294	360.6603
hsr	0.00112	170.1629
teamKills	0.00042	113.2031
vehicleDestroys	0.00003	26.5950
roadKills	-0.00001	4.6408

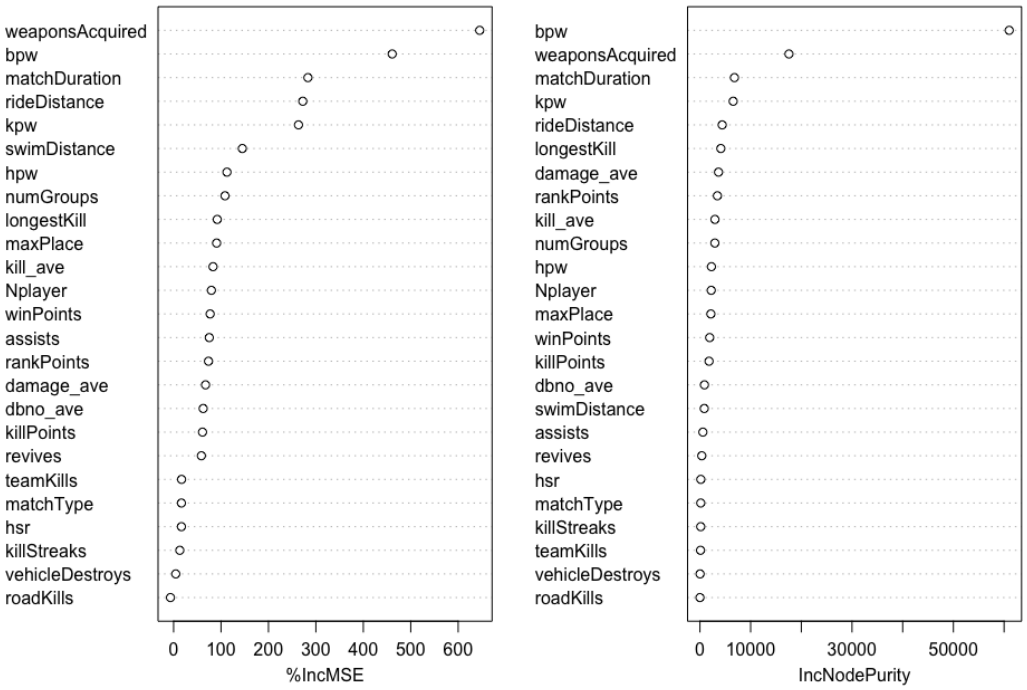
Notes: The variables are sorted in descending order by %Increase in MSE.

Table 5: **Random forests**

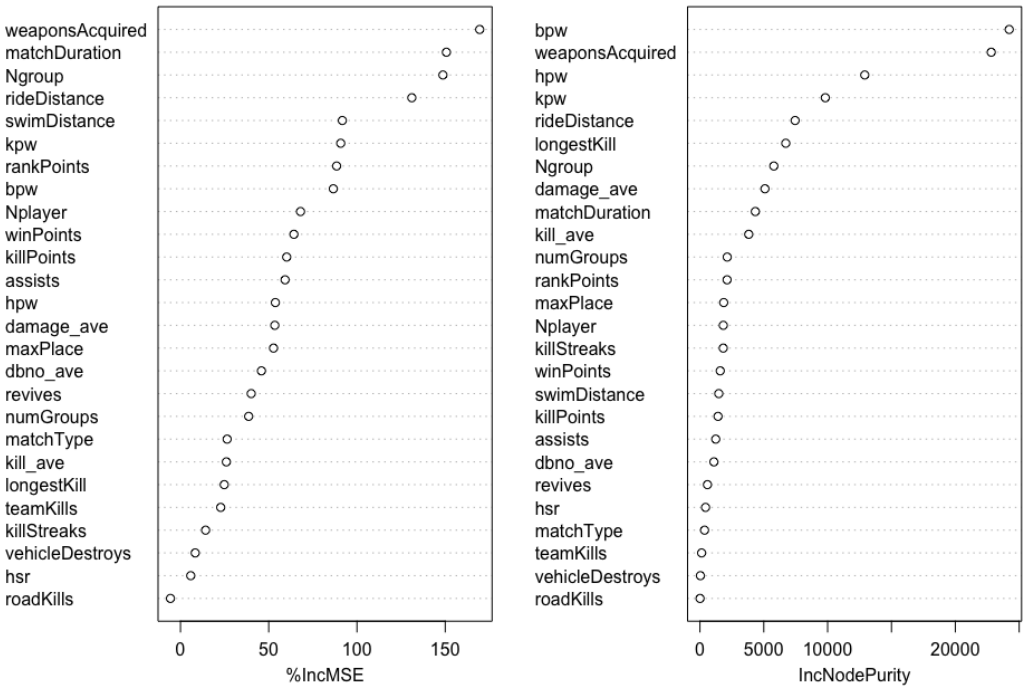
	%Increase in MSE	Increase in node purity
kpw	0.22915	9818.0290
bpw	0.22619	24219.2282
weaponsAcquired	0.19540	22813.3848
kill_ave	0.13514	3820.8496
longestKill	0.13155	6722.0995
hpw	0.11838	12907.7312
rideDistance	0.09106	7454.8169
Ngroup	0.08849	5786.4198
killStreaks	0.08808	1824.9469
damage_ave	0.05732	5091.9399
numGroups	0.04967	2136.7439
maxPlace	0.04270	1862.0402
matchDuration	0.03774	4339.0643
matchType	0.02886	361.2851
swimDistance	0.01768	1476.2695
winPoints	0.01607	1589.2840
dbno_ave	0.01518	1094.5002
rankPoints	0.01200	2125.0599
killPoints	0.01173	1415.8193
assists	0.01033	1237.9372
Nplayer	0.00735	1828.7321
hsr	0.00592	438.8919
revives	0.00485	587.2702
teamKills	0.00057	131.0544
vehicleDestroys	0.00018	38.4467
roadKills	-0.00001	8.2531

Notes: The variables are sorted in descending order by %Increase in MSE.

Variable relative importance (Bagging)



Variable relative importance (Random forests)



6.4 R code

```
# library packages
library(ggplot2)
library(dplyr)
library(magrittr)
library(knitr)
library(MXM)
library(MASS)
library(glmnet)
library(gbm)
library(tree)
library(randomForest)
library(rfUtilities)
library(caret)
library(pastecs)
library(ensr)

## load the data
pubg <- read.csv("/Users/jinglin0828/Downloads/Final-Project/Final-Project
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣/final_pubg.csv",header = T)

## preliminary analysis
# bar graph for kills
pubg$kill.level <- cut(pubg$kills,c(-1,0,2,5,10,max(pubg$kills)))
ggplot(pubg, aes(x=kill.level, y=winPlacePerc, fill=kill.level)) +
  geom_boxplot() +
  theme(legend.position = "none")+
  labs(x = "Number␣of␣enemy␣players␣killed",y="Ranking␣(winPlacePerc)")+
  scale_x_discrete(labels=c("0␣kill","1-2␣kills","3-5␣kills","6-10␣kills", "10+␣kills"))
  ggtitle("Relationship␣between␣ranking␣and␣kills") +
  theme(
    plot.title = element_text(size=16,face="bold",hjust=0.5)
  )

# distribution of weaponsAcquired
```

```
ggplot(pubg, aes(x=weaponsAcquired)) +
  geom_histogram(aes(y=..density..), colour="black", fill="white",binwidth = 5)+
  geom_density(alpha=0.5, color="darkblue", fill="lightblue") +
  labs(x="Number of weapons picked up")

# distribution of the match duration
ggplot(pubg, aes(x=matchDuration)) +
  geom_histogram(aes(y=..density..), colour="black", fill="white",binwidth = 60)+
  geom_density(alpha=0.5, color="darkblue", fill="lightblue") +
  labs(x="Duration of match (seconds)")

# ranking vs walking distance
pubg$win.level <- cut(pubg$winPlacePerc,c(-1,0.25,0.5,0.75,1))
ggplot(pubg, aes(x=win.level, y=walkDistance, fill=win.level)) +
  geom_boxplot() +
  theme(legend.position = "none")+
  labs(x = "Ranking (winPlacePerc)",y="Total distance traveled on foot")+
  scale_x_discrete(labels=c("<25%","25%-50%","50%-75%",">75%")) +
  ggtitle("Relationship between ranking and walking distance") +
  theme(
    plot.title = element_text(size=16,face="bold",hjust=0.5)
  )

## data transformation
# fill NA with zeros
pubg[is.na(pubg)] <- 0

# feature engineering
pubg$kill_ave <- pubg$kills/pubg$Nplayer
pubg$damage_ave <- pubg$damageDealt/pubg$Nplayer
pubg$dbno_ave <- pubg$DBNOs/pubg$Nplayer

pubg$bpw <- pubg$boosts/pubg$walkDistance
pubg$hpw <- pubg$heals/pubg$walkDistance
pubg$kpw <- pubg$kills/pubg$walkDistance
```

```
pubg$hsr <- pubg$headshotKills/pubg$kills

# change NaN to 0
is.nan.data.frame <- function(x){
  do.call(cbind, lapply(x, is.nan))}
pubg[is.nan(pubg)] <- 0

# drop the obs who var has Inf
pubg <- pubg[is.finite(pubg$bpw)&is.finite(pubg$hpw)&is.finite(pubg$kpw),]

# classify match type into three basic categories (solo, duo, squad)
pubg <- subset(pubg,matchType != "crashfpp"& matchType != "flarefpp"
        & matchType !="flaretpp")
pubg$matchType <- ifelse (pubg$matchType %in% c("squad-fpp","squad","normal-squad-fpp",
                          "normal-squad"),"squad",ifelse(pubg$matchType %in% c("duo-fpp","

# change macth type into factor
pubg$matchType <- factor(pubg$matchType)

# exploring headshot rates
# the histogram of hsr given kills > 1
ggplot(pubg[(pubg$hsr!=0 & pubg$kills>1), ], aes(x=hsr)) +
  geom_histogram(binwidth=0.05)

# detecting possible cheaters
# kills vs walking distance
ggplot(pubg, aes(x=rideDistance, y=roadKills)) +
  geom_point()

# roadkills vs road distance
ggplot(pubg[(pubg$kills>1),], aes(x=walkDistance, y=kills)) +
  geom_point()

# the histogram of high kills > 15
ggplot(pubg[(pubg$kills>15), ], aes(x=kills)) +
  geom_histogram(binwidth=1)
```

```
# the histogram of longkills > 500
ggplot(pubg[(pubg$longestKill>500), ], aes(x=longestKill)) +
  geom_histogram(binwidth=50)

## train and test dataset splitting
set.seed(1234)
n <- nrow(pubg)
train.index <- sample(1:n, n*0.7)
train <- pubg[train.index, ]
test <- pubg[-train.index, ]

tr <- subset(train, select = -c(Id, groupId, matchId, killPlace, walkDistance,
                                kill.level, win.level,kills,damageDealt,DBNOs,
                                heals,boosts,headshotKills))
te <- subset(test, select = -c(Id, groupId, matchId, killPlace, walkDistance,
                               kill.level, win.level,kills,damageDealt,DBNOs,
                               heals,boosts,headshotKills))

# scale the variables
xtr <- scale(data.matrix(tr[, !(colnames(tr) %in% c('winPlacePerc'))])) # scale features
ytr <- scale(data.matrix(tr[c('winPlacePerc')])) #response

xte <- scale(data.matrix(te[, !(colnames(tr) %in% c('winPlacePerc'))])) # scale features
yte <- scale(data.matrix(te[c('winPlacePerc')])) #response

tr_s <- data.frame(xtr,ytr)
te_s <- data.frame(xte,yte)


## Linear regularization methods

# 1. ridge regression
# preset tuning parameters
grid <- 10^seq(1,-2,length=1000) # choose lambda from 0.01 to 10
```

```
# fit 10-fold cv to ridge
cvridge <- ridgereg.cv(ytr, xtr, K = 10, lambda = grid, auto = FALSE,
                       seed = TRUE, ncores = 1, mat = NULL )
cvridge_mspe <- data.frame(cvridge$mspe)

ggplot(cvridge_mspe, aes(x=as.numeric(rownames(cvridge_mspe)), y=cvridge.mspe)) +
  geom_point(color="maroon") +
  labs(x = "lambda values",y="mean prediction error")
dev.copy(png,'/Users/jinglin0828/Downloads/Final-Project/graphs/cvridge.png')



# plot the variable importance graph and charts (coefficients vs. lambda)
FI_ridge = glmnet(xtr,ytr, alpha =0, lambda = 10^seq(3,-3,length=1000))
ridge_coefs <- data.frame(FI_ridge$beta)
plot(FI_ridge)

# use the best lambda to fit the data
best_ridge = glmnet(xtr, ytr, alpha = 0, lambda = cvridge$lambda)
ridge_coef <- data.frame(coef(best_ridge)[order(abs(coef(best_ridge)), decreasing=TRUE)])

# score the model by calculating the test mse
yhat.ridge <- predict(best_ridge, newx=data.matrix(xte), s=cvridge$lambda, type = "link")
ridge.mse <- mean((yhat.ridge - yte)^2)
ridge.mse

# plot the true ranking vs. predicted ranking for the test set
test.ridge <- data.frame(y_true=te_s$winPlacePerc, y_hat=yhat.ridge) %>% rename(y_hat=X1)
ggplot(data=test.ridge, aes(x=y_hat, y=y_true)) +
  geom_point(shape = 1) +
  geom_abline() +
  labs(x = "predicted ranking (ridge)", y= "true ranking") +
  xlim(min(yhat.ridge),max(yhat.ridge))+
  ylim(min(yte),max(yte)) +
  theme_classic()

# 2. lasso regression
```

```
# preset tuning parameters
grid <- 10^seq(1,-2,length=1000) # choose lambda from 0.01 to 10

# fit the model with different lambda
lasso.mod <- glmnet(xtr, ytr, alpha=1, lambda=grid)
plot(lasso.mod)

# fit 10-fold cv to lasso
set.seed(1)
cvlasso <- cv.glmnet(xtr, ytr, alpha=1)
plot(cvlasso)
best_lasso_lam <- cvlasso$lambda.min

# use the best lambda to fit the data and calculate the test mse
newX <- model.matrix(~.-winPlacePerc,data=te)
yhat.lasso <- predict(lasso.mod, s=best_lasso_lam, newx=data.matrix(xte),type="link")
lasso.mse <- mean((yhat.lasso-yte)^2)
lasso.mse

# plot the true ranking vs. predicted ranking for the test set
test.lasso <- data.frame(y_true=te_s$winPlacePerc, y_hat=yhat.lasso) %>% rename(y_hat=X1)
ggplot(data=test.lasso, aes(x=y_hat, y=y_true)) +
  geom_point(shape = 1) +
  geom_abline() +
  labs(x = "predicted ranking (lasso)", y= "true ranking") +
  xlim(min(yhat.lasso),max(yhat.lasso))+
  ylim(min(yte),max(yte)) +
  theme_classic()

# 3. Elastic net
# search a grid of values of alpha and lambda for the optimal elastic net model
set.seed(1234)
cven <- ensr(y=ytr, x=xtr, standardize = FALSE)
cven.summary <- summary(cven)
plot(cven)
```

```
# get the optimal alpha and lambda
best <- cven.summary[cvm == min(cvm)]
best

# get the optimal model
str(preferable(cven), max.level = 1L)
plot(preferable(cven), xvar = "norm")

# calculate the test mse
en.best <- glmnet(xtr, ytr, alpha = best$alpha, lambda = best$lambda)
yhat.en = predict(en.best, newx=xte)
en.mse <- mean((yhat.en-yte)^2)
en.mse

# plot the true ranking vs. predicted ranking for the test set
test.en <- data.frame(y_true=te_s$winPlacePerc, y_hat=yhat.en) %>% rename(y_hat=s0)
ggplot(data=test.en, aes(x=y_hat, y=y_true)) +
  geom_point(shape = 1) +
  geom_abline() +
  labs(x = "predicted ranking (elastic net)", y= "true ranking") +
  xlim(min(yhat.en),max(yhat.en))+
  ylim(min(yte),max(yte)) +
  theme_classic()


## Tree-based methods

# 4. Regression Tree w/ cost complexity pruning
# fit the tree
set.seed(1)
tree <- tree(winPlacePerc~., tr_s)
summary(tree)

# plot the original tree
plot(tree)
text(tree,pretty=0)
```

```
# cost complexity pruning (find the best tree is the original one)
prune_tree=prune.tree(tree)
plot(prune_tree)

# return the best regression tree model
best.tree <- prune.tree(tree, best =7)

#get the test mse for tree pruning w/ cost complexity pruning
tree.pred <- predict(best.tree, newdata = te_s)
tree.mse <- mean((tree.pred-te_s$winPlacePerc)^2)
tree.mse

# 5. Bagging
# fit the bagging model by setting mtry=24, n.tree=500
bag = randomForest(formula = winPlacePerc ~. , data=tr_s, mtry=25,
                   n.tree = 500, importance=TRUE)

# get predicted value
yhat.bag = predict(bag, newdata=te_s)
bag.mse <- mean((yhat.bag-te_s$winPlacePerc)^2)
bag.mse

# plot the true ranking vs. predicted ranking for the test set
test.bag <- data.frame(y_true=te_s$winPlacePerc,y_hat=yhat.bag)
ggplot(data=test.bag, aes(x=y_hat, y=y_true)) +
  geom_point(shape = 1) +
  geom_abline() +
  labs(x = "predicted ranking (bagging)", y= "true ranking") +
  xlim(min(yhat.bag),max(yhat.bag))+
  ylim(min(yte),max(yte)) +
  theme_classic()

# get the variable relative importance
bag$importance
varImpPlot(bag, main="Variable relative importance (Bagging)")
```

```
# 6. Random forests
# fit the random forests model by setting mtry=5, n.tree=500
set.seed(1234)
rf = randomForest(formula = winPlacePerc ~., data=tr_s, mtry=5,
                  importance=TRUE, n.tree=500)

# get predicted value
yhat.rf = predict(rf, newdata=te_s)
rf.mse <- mean((yhat.rf-te_s$winPlacePerc)^2)
rf.mse

# plot the true ranking vs. predicted ranking for the test set
test.rf <- data.frame(y_true=te_s$winPlacePerc,y_hat=yhat.rf)
ggplot(data=test.rf, aes(x=y_hat, y=y_true)) +
  geom_point(shape = 1) +
  geom_abline() +
  labs(x = "predicted ranking (random forest)", y= "true ranking") +
  xlim(min(yhat.rf),max(yhat.rf))+
  ylim(min(yte),max(yte)) +
  theme_classic()

# get the variable relative importance
rf$importance
varImpPlot(rf, main="Variable relative importance (Random forests)")

# 7. Boosting
# preset tuning parameters and using the "gbm" package, we could get the best tree size
# from [1,1000] by setting n.trees=1000
Lambda <- 10^seq(-3,-1,by=1)
Depth <- c(1,2,3,4)

# fit the boosted tree and cv
set.seed(1234)
cverr <- data.frame(inter.depth = numeric(0), lambda=numeric(0),
             n.trees = numeric(0), cv.error = numeric(0))
```

```
for (depth in Depth){
  print(depth)
  for (lam in Lambda){
    gbm <- gbm(winPlacePerc ~ . , data = tr_s, distribution = "gaussian",
               n.trees = 1000, shrinkage = lam, interaction.depth = depth, #tuning
               cv.folds = 10, keep.data = FALSE, verbose = FALSE, n.cores = 1)
    n.tree <- gbm.perf(gbm)
    new <- data.frame(inter.depth = depth, lambda = lam,
                      n.trees = n.tree, cv.error=gbm$cv.error[n.tree])
    cverr <- rbind(cverr,new)
    print(lam)
  }
}

# get the best parameters
best.pars <- cverr[cverr$cv.error == min(cverr$cv.error),]
best.pars

# get the boosting model with the optimal paramters: d=4, B=1000, lambda=0.1
b_gbm <- gbm(winPlacePerc ~ . , data = tr_s, distribution = "gaussian",
             n.trees = 1000, shrinkage = 0.1, interaction.depth = 4)

# get predicted value
yhat.gbm <- predict(b_gbm, newdata = te_s,n.trees = 1000)
gbm.mse <- mean((gbm.pred-te_s$winPlacePerc)^2)
gbm.mse

# plot the true ranking vs. predicted ranking for the test set
test.gbm <- data.frame(y_true=te_s$winPlacePerc,y_hat=yhat.gbm)
ggplot(data=test.gbm, aes(x=y_hat, y=y_true)) +
  geom_point(shape = 1) +
  geom_abline() +
  labs(x = "predicted ranking (boosting)", y= "true ranking") +
  xlim(min(yhat.gbm),max(yhat.gbm))+
  ylim(min(yte),max(yte)) +
  theme_classic()
```

6.5 Python code for scrapping

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 20 12:42:48 2020

@author: Jinglin Yang & Jingming Wei
"""

import urllib, json, time, requests, os

os.chdir("/Users/Mark/Desktop/Intro to Machine Learning/Proposal/")

base = r"https://api.pubg.com/shards/pc-na/players/"
addP = "{"+ "0fed8189-4f77-4831-90ca-e2f1834f6b5d" + "}"
url = base + addP + "/seasons/lifetime"
header = {
  "Authorization": "Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI1MjllNjMzMC0zMjU4L
  "Accept": "application/vnd.api+json"}
response = requests.get(url, headers=header)
jsonData = response.json()
resu1 = jsonData['data']['attributes']['gameModeStats']['solo-fpp']
added_features = list(resu1.keys())


def PUBGAPI(playerID,delay=0):
  base = r"https://api.pubg.com/shards/pc-na/players/"
  addP = "{"+ playerID + "}"
  url = base + addP + "/seasons/lifetime"
  header = {
  "Authorization": "Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI1MjllNjMzMC0zMjU4L
```

```
  "Accept": "application/vnd.api+json"}
  response = requests.get(url, headers=header)
  if response.status_code == 200:
      jsonData = response.json()
      resu = jsonData['data']['attributes']['gameModeStats']['solo-fpp']
      finlist = list(resu.values())
  else:
      finlist = [None for i in range(len(list(resu1.values())))]
  time.sleep(delay) #in seconds
  return finlist
PUBGAPI("0fed8189-4f77-4831-90ca-e2f1834f6b5d")
```

```
import csv
names = ('Add',"lat","lon")
for i in range(7):
    csvfile = open('chunk.train{}.csv'.format(i),'r')
    reuslts = []
    result_csv = open('train{}.csv'.format(i), 'w')
    result_csv_writer = csv.writer(result_csv)
    result_csv_writer.writerow(added_features)
    reader = csv.DictReader(csvfile)
    n=0
    for row in reader:
        player = row['Id']
        pubgR = PUBGAPI(playerID = player)
        reuslts.append(pubgR)
        n = n + 1
        print('Successfully analyzed',n,'player(s)')
    print('END{}'.format(i))
```

```
    result_csv_writer.writerows(reuslts)

    result_csv.close()
```

References

Athey, Susan, and Guido Imbens. 2019. "Machine Learning Methods Economists Should Know About."