BATTLE ROYALE FINISH PLACE PREDICTION

Kyle Wojtaszek

IST 707 – Final Project

Introduction

PlayerUnknown's Battlegrounds (also known as PUBG) is one of many recently popular 'battle royale' style video games where a large number of players (typically 100) begin the game, and players are eliminated until the last one standing is declared the winner. Players can be eliminated in a number of ways, the most common of which is by being killed by another player via a number of weapons that are scattered across the game's map. Players can also be eliminated by accidentally killing themselves and by being disconnected from the game due to an interruption in their network service. This style of games has become very popular with young adults and children over the past 2-3 years due to their fast paced action and competitive nature. Recently, a competition was posted on Kaggle to challenge users to predict the finishing order of players in a match based on statistics from that match. Our objective in this paper is to craft a model for this competition and perform as well as we can.

Data Set Basics

The data set is available here, and I have also included a link to it in the References section at the end of the paper. The training set has 4.45 million rows and the test data set has 1.93 million rows. As we will see later on, there are some unique challenges in dealing with a data set this large. Each row contains 28 properties and the training set also has a 29th column with the match results in it. There is a complete data dictionary on the Kaggle site and we will cover specific definitions later on when we explore which features came out as important. However, there are a couple concepts that warrant some early coverage:

- **GroupID/MatchID** – These columns will be important for grouping. The MatchID allows us to identify players from a given match. This is important as a player's rank will be determined by comparing them to others in their match. The GroupID identifies a group of players who received the same finishing position in a given match. This can be because they are a team or because they all were disconnected from the game at the same time. The disconnection scenario becomes an important nuance for understanding the results.
- **killPoints/WinPoints/rankPoints** – The game implements an Elo system[1] to rank players for fair matchmaking purposes. Based on the description of the data, there are two different ranking systems in this data. The 'old' system in which each player has a killPoints and winPoints rating based on their career number of kills and wins respectively. The 'new' system has one unified 'rank' points, based on match results. Some matches only have the 'old' system and some matches only have the 'new' system, so we'll have to deal with this in preprocessing. We combine them into a single column called 'points', which represents a players average Elo rating across whichever of these columns is available for that match.
- **Boosts** – Many of the basic stats from the matches are straight forward (kills, weapons acquired, etc.), however 'boosts' is a concept that is somewhat unique to these type of video games. In this game, boosts refer to a class of items that are found on the battle field which give the player two advantages: They can run faster for a period of time, and they also will regenerate some of their health.

[1] If you are unfamiliar with Elo ranking systems, you can read more about them here: https://en.wikipedia.org/wiki/Elo_rating_system . However, the system itself is not consequential to this paper.

What’s our target?

The goal of our prediction is to accurately predict the **winPlacePerc** value for the test data (perc is short for percentage). This value is basically a percentage of the people in the match that the player beat. The player/team that finishes in first place will get a winPlacePerc of 1.000, last place is 0.000, and middle of the pack is 0.5000. WinPlacePerc is calculated using the following formula:

$$winPlacePerc = \frac{(maxPlace - winPlace)}{(maxPlace - 1)}$$

WinPlace is the absolute finishing place for the player or group in their match (winPlace = 1 for first place, 2 for second place, and so on). MaxPlace is the worst possible finishing place in a given match. The maxPlace is provided as part of the data set. The winPlace column can be reverse engineered from the winPlacePerc column in our training set.

For the competition, our accuracy will be determined by taking the Mean Absolute Error of our test data set answers. This is calculated by taking our guess, subtracting the real value, and taking the average of the absolute value of these differences across the entire test data set. Thus, a mean absolute error of 0 would mean that we predicted every finishing position perfectly.

Benchmarking

Before we start putting together our machine learning algorithm, we should establish a good benchmark so that we can determine if our algorithm turns out to be any good. There are two very simple things we can do with this dataset to make an ‘easy’ benchmark.

Linear Regression Benchmark

Our first benchmark will be just taking all the data and throwing it into a linear regression. After throwing out the columns which are not significant due to their high p-values, we get a model that gives us a mean absolute error of 0.0720. This will be our first benchmark.

Using “KillPlace”

There is a column in the data set called killPlace, which ranks all the players in a given match by the number of kills they had. For example, the player with the most kills in a given match will have a killPlace of 1, second most kills will be 2, and so on. Since this column already ranks the players, what if we just use this rank and calculate a winPlacePerc from it? This gets us a mean absolute error of 0.0628. This performs a little better than our linear regression baseline and will function as our second benchmark.

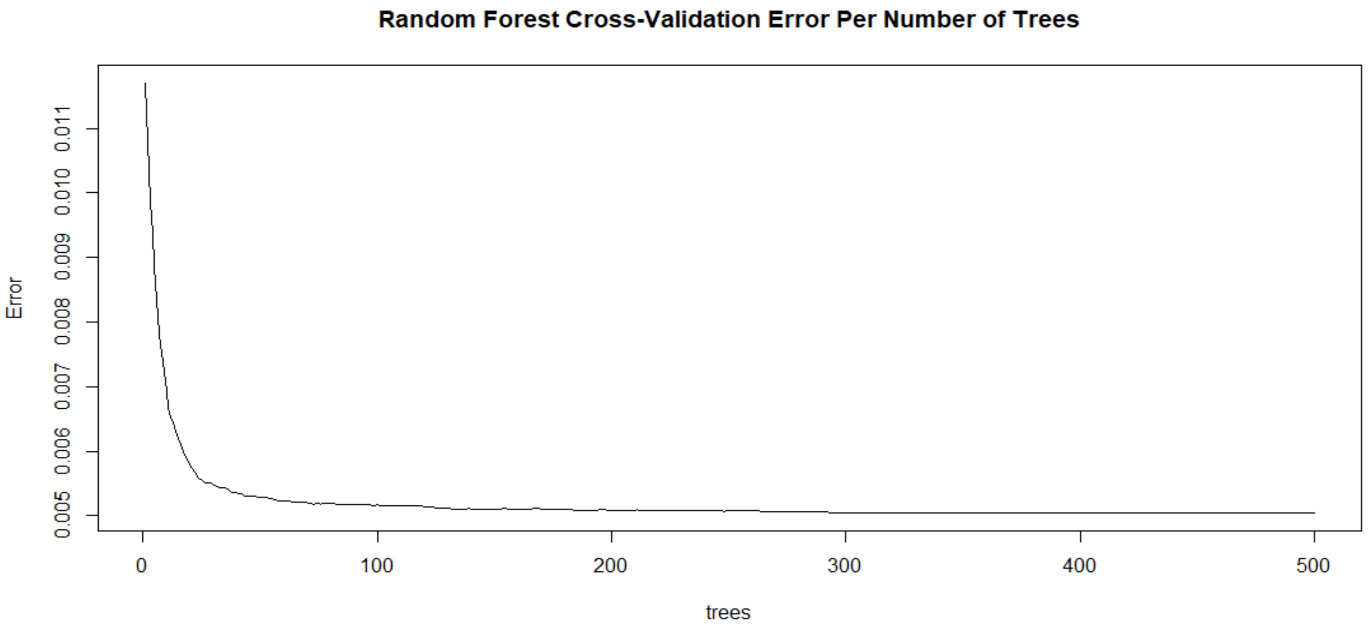
Any machine learning algorithm we put together should be able to beat both these benchmarks, preferably by a good margin.

Model Selection and Tuning

For this project, we are going to use a random forest model. We can quickly rule out a linear regression model, as this problem is not linear and, as we found in our benchmarking phase, the linear regression performs poorly. I also ruled out kNN quickly, as we are not really looking to classify things here. A random forest can be used to predict a continuous variable (such as our winPlacePerc), by taking the average of the ‘votes’ from the decision tress that it is assembled from. I also considered using an SVM model for this problem. I ran a few trials where I gave the same data to both a random forest and an

SVM model and the SVM model consistently was ~0.0075 worse even after some tuning. This is a minor difference, and it is possible that the SVM model could perform on equal footing and I hadn't found the right parameters. However, we're going to stick with random forest for this project and try to maximize our accuracy. Another important factor is that I could fit approximately 10 times the amount of training data into a random forest model before running out of memory when compared to an SVM model. This extra training data can get us some small wins in accuracy.

For tuning our random forest model, we have two main knobs to turn: The number of trees and the mtry variable, which tells the algorithm how many variables to randomly pick as candidates at each tree split. For number of trees, the default value is 500. We can use the returned model to graph the cross-validation error over those 500 trees, as seen below.



Based on this graph, it looks like we actually are maxing out our cross-validation accuracy before we hit 500 trees, and in fact I found that I could decrease this number to 400 trees with no impact on our prediction accuracy. Based on the graph, it appears we might be able to reduce it further, but I found that in practice, when I reduced the number to 375 I started to lose a small amount of accuracy. By reducing our number of trees from 500 to 400 we help keep our model running quickly and cut down on memory usage, both of which are important with our massive data set.

For mtry, the default value is one third of the number of variables used for prediction. Based on research around the internet, the consensus is that this is generally a good number and messing with it rarely provides any positive results. For my own education, I increased mtry by one and decreased it by one and ran these two models. As expected, they both performed worse than when using the default value. So, we will stick with the default value here and use 400 for our number of trees.

First Run Accuracy

Now that we've tuned our parameters, we can run the random forest model on our raw data and get a starting point for our accuracy. Doing so gave me a mean absolute error of 0.0573 on the test data. This is definitely an improvement over our two benchmarks, however the data is still raw, and I think we can do better by massaging a bit more meaning out of our training data set.

Feature Engineering

One of the key phases for increasing our model's accuracy is to pre-process our data via feature engineering. As the old phrase goes, garbage in, garbage out. If we can make our data cleaner and easier for our model to interpret, the model will perform better. I tried a number of things on this data set, many ended up being of little to no value, but we'll take a look at some of the wins that brought our mean absolute error down.

Larger Wins

The following two pre-processing steps gave us large increases in accuracy and provide interesting insights into the data set.

Scaling the Data by Match

Each match in a video game is different. Sometimes you are paired up with a lot of skilled players and this leads to a longer, more strategic game. Sometimes you may be paired up with a number of players who are just messing around and play a quicker, looser game. Since each game is unique, it doesn't make sense for our model to take raw numbers from each match and compare them to each other. In order to combat this issue, I scaled all the variables on a per match basis. By scaling, a mid-tier performer in *any* match should have approximately the same scaled variables, even if the raw variable itself could be very different.

This dataset also contains a number of matches which we can consider 'outlier' matches, where the statistics are wildly different than the majority of matches. The consensus on the Kaggle discussion is that these games are typically custom games where a group of friends get together and mess around for fun, playing in ways that you would not see in a standard competitive match. Not only does the scaling make matches more comparable, it also helps soften the impact of these outlier matches.

Z-score scaling vs. Min Max scaling

How do we scale the data? Originally, I used R's built in scale function, which uses Z-score scaling. However, after some thought I decided to try Min Max scaling, as doing so results in the best player in a given variable in a given match always having a 1.0 and the worst always having a 0.0, which should make things more consistent for the modeling process. The switch from Z-score to Min Max gave a minor, but worthwhile ~0.005 decrease in the mean absolute error.

Doing this scaling was the single biggest improvement of model accuracy that I saw throughout the entire process. Overall, it brought my mean absolute error down by approximately 0.010.

Dealing with Teams

Approximately half of our training data comes from team games, where a team of four players works together and is ranked together at the end of the match. If we just consider each player as an individual, we are missing out on this important team interaction, making our predictions less accurate. But how do we deal with teams? Do we sum up the four players performances and rank them as a unit? Or do we just take the four players and average them together at the end?

Surprisingly in my experimentation, averaging the team members final ranking together was consistently more accurate than any scheme I could come up with to combine teams into one statistical entity. I tried many combinations of summing certain stats and averaging others based on knowledge of how the game works, and none could match the accuracy of simply running the model on four individuals and averaging their result. While unexpected, this result is actually good for us, as it makes our model simpler and makes our post-processing simple as well.

Having a consistent way of dealing with teams, instead of leaving them as individuals, improved our mean absolute error by ~0.0050.

Smaller Wins

After the two improvements above, further increases in accuracy became more difficult.

Adding Some Team Stats

When we were predicting a team game, I found that there was minimal, but significant value to adding a handful of team-wide variables on top of the already established player variables. The variables include total team kills, damage, walk distance, weapons acquired, and boosts. Adding these columns increased accuracy by about 0.0010.

These columns provided an accuracy increase because, while we are ranking each player as an individual, giving the model some info on the team performance allows it to 'cheat' a bit by having this knowledge about the team.

Post-Processing Results

Up to this point, we've been treating winPlacePerc as a continuous variable, and over the entire data set it basically is. There are a large range of games with different numbers of groups, so the winPlacePerc falls all over the place between 0 and 1. However, for any given match there are a set number of values that winPlacePerc can be and we can calculate those values using the formula provided earlier in this paper. For example, if a match has 11 groups, we know those groups will finish with winPlacePerc values of 0, 0.1, 0.2, 0.3….0.9, and 1.0. Our model doesn't know this and will just predict any value between 0 and 1. This seems like a good place for us to get some improvement.

Rounding

My first intuition was to just round the scores to the nearest possible score. So, using the 11-group example above, if the model predicted a group to get a winPlacePerc of 0.3689, I can just round that to the nearest valid value of 0.4. This simple rounding gave us a slight boost in accuracy, in the vicinity of 0.0005, but it turns out we can do better.

Ranking

After inspecting the distribution of predicted winPlacePerc, I noticed something interesting. The model was not picking very many first-place finishers or last place finishers. Instead, there was a clump of players around 0.2 and 0.8, and then a drop off as you moved towards 0 and 1. Intuitively, this makes sense for a random forest model. For a player to be assigned a 1.0 winPlacePerc prediction, all 400 trees would have to unanimously pick them to finish with a 1.0, which would be a pretty rare event. This non-uniformity meant that when rounding the group scores, we would often end up with multiple groups rounded to the same scores, which is automatically wrong as each group is guaranteed to have a unique score.

In order to combat this non-uniformity of results, I decided to change my script to essentially *use the prediction as a ranking instead of a raw prediction*. So, I would take whatever the highest prediction is, whether it was 1.0 or not, and assign that player to 1st place; the 2nd highest prediction to 2nd place, and so on. This looks like a valid move, because we can interpret the prediction result essentially as 'polling' the trees in the forest. The player or group with the highest prediction finished in first-place in the poll. By interpreting the data this way, we ensured not only that each team was assigned a valid value, but that each valid value was only used once. This gave us the best chance of getting a match completely correct, which would contribute a valuable 0 to our mean absolute error. After implementing this post-processing strategy, we gained an additional 0.0010 mean absolute error on top of the gains we received from rounding.

Performance Optimizations

As mentioned previously, this data set is extremely large, millions of rows. Due to this, it was very expensive to work with in terms of memory usage and processing time. There were two key optimizations I found which allowed me to greatly increase my training data size, which helped increase my model's accuracy.

Ranger Package

The Ranger package and ranger function are a high-performance implementation of the random forest algorithm. Using this package, I saw processing time that was orders of magnitude faster compared to the standard randomForest function. Due to this, I was able to put in a lot more training data and get results in a reasonable time. This package is fast enough that my new performance barrier was memory usage. I could put as much data as could fit into memory into the function and it would create a model in reasonable time. Now I needed to find a way to work around memory consumption issues.

Splitting Our Model

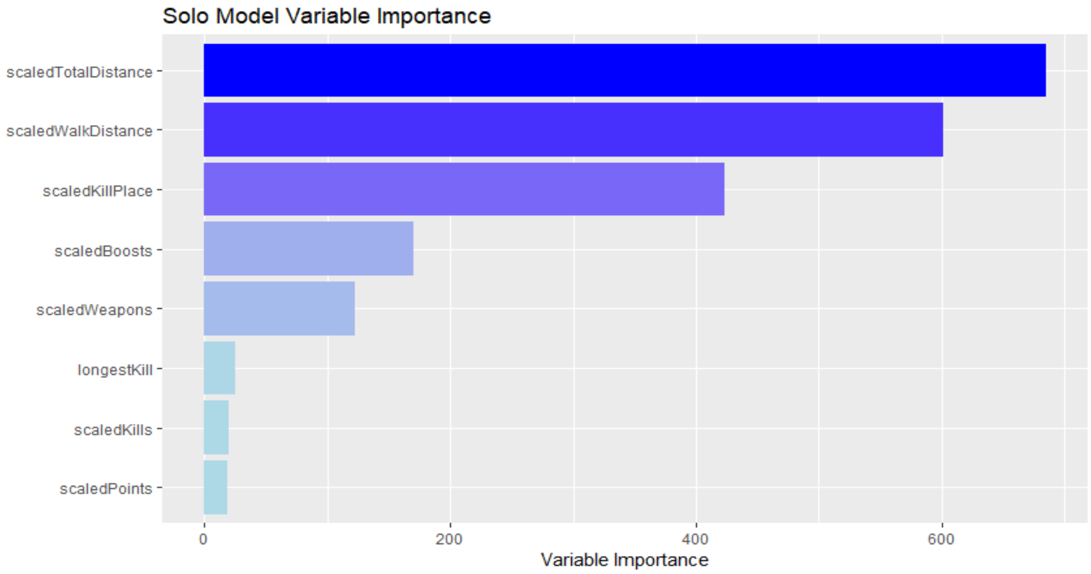
Through using the Ranger package, I was able to run my model on about 250,000 rows before hitting the cap on memory usage in the Kaggle VM. That was a pretty good number, but I was hoping to be able to get a little more out of it. To do this, we can split our data set up by game mode.

There are three primary game modes in the data set: solos, duos (teams of 2), and squads (teams of 4). Each type of game is played slightly different, in solos you have no friends and it is every person for themselves. The larger the team, the more coordination that can occur and the more important coordination becomes. Based on this domain knowledge, I split the data up into three partitions, one for each game type. I ran separate models on each game type and, in doing so, I could customize which variables were passed in for each game type. For instance, the team-based stats were worthless to the solos model, so I could remove those with no penalty on accuracy. By doing this, I not only tripled my sample size from 250,000 to 750,000, as each model could get its own 250,000 samples before running out of memory, but by scoping the models to a game type I saw a slight increase in accuracy, in the ballpark of 0.0005.

Results

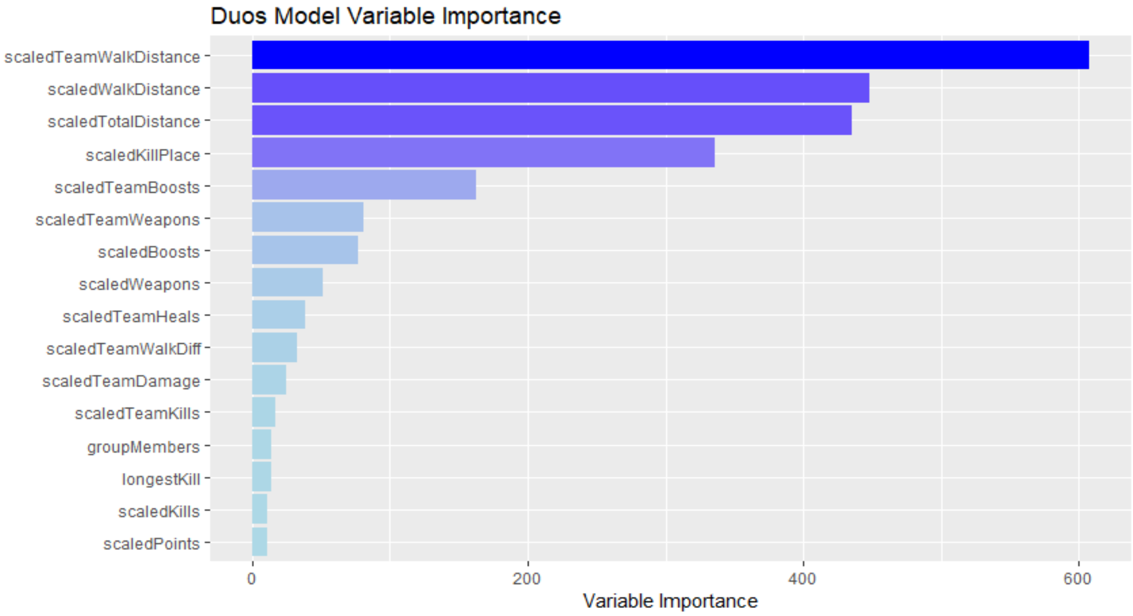
After completing all of the above steps, I submitted my script to Kaggle for official scoring. I ended up with a final mean absolute error of 0.0381, which is currently 352nd out of 960 submissions, approximately 63rd percentile. Overall, I think this is a good score, given that the majority of top submissions are using more advanced models like neural networks or boosted algorithms like XGBoost, both of which I am not currently familiar enough with to use effectively.

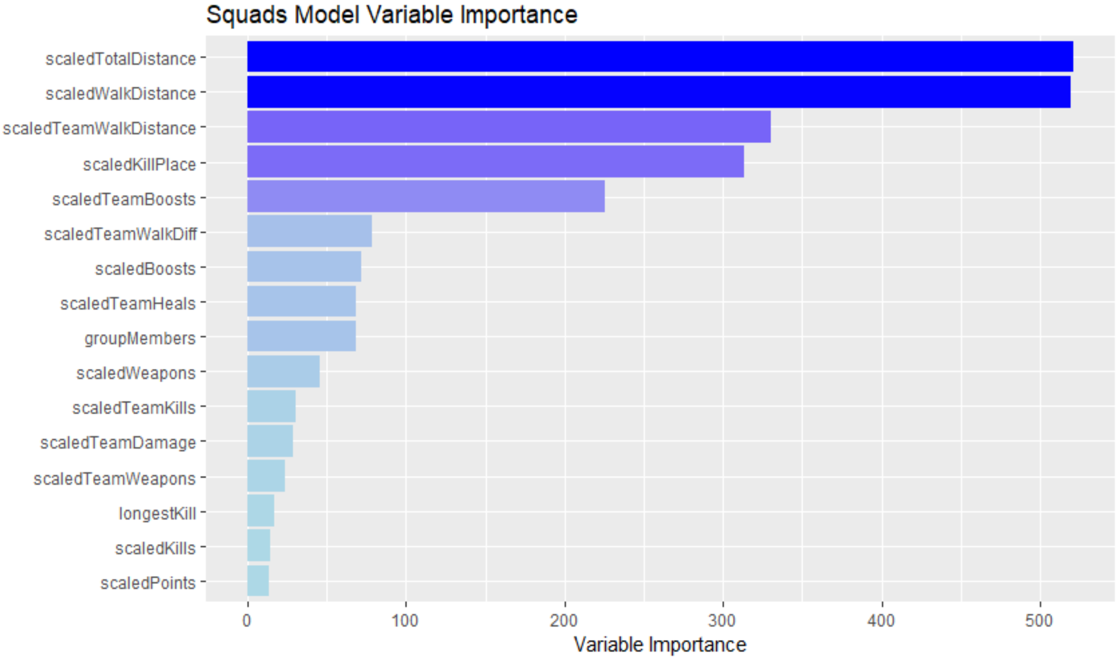
Our final script ended up using three different models, one for solos, duos, and squads respectively. For each model, we looked at the importance of each variable using the random forest's built in importance matrix. We removed any column which were relatively unimportant, and then checked our new model to ensure no loss in accuracy. Following is the importance of the variables we included in each model along with an explanation of the columns used and why they are important. For all columns in the graphs, the term 'scaled' indicates we used our Min Max scaling on this column. First, we'll look at the solo model:



- **scaledTotalDistance** and **scaledWalkDistance** are our most important features: WalkDistance refers to the distance a player's character walked in the game. TotalDistance is the sum of walking distance, swimming distance, and vehicle riding distance. These columns will consistently be the most important variables in every importance calculation. The reason for this is that in a battle royale type game, you are being ranked by how long you survived in the game. Walk distance and total distance are extremely good indicators of how long a player survived, because generally a player is walking around for most of the game. Thus, the longer you were in the game, the more distance you walked.
- **KillPlace:** We used this column for our benchmark, and it also turned out to be important in our random forest models. This is intuitive, because the more players you killed, the more likely it was that you finished higher.
- **Boosts**: If you used more boosts, you were able to regenerate health (which will help you live longer) and also allows you to walk faster, thus perhaps allowing a player to escape a bad situation. Using boosts also indicates that a player has a better understanding of how the game works than a player who ignores boosts.
- **Weapons Acquired**: Similar to walk distance, the longer a player is in the game, the more opportunities they have to pick up more weapons, so this is intuitive. Also, the more weapons a player picks up, the more likely it is for them to acquire a powerful weapon, which will give them an advantage.

- **Longest Kill**: This variable that tells us what the longest distance was a player killed another player from. It is an indirect measure of a player's accuracy, and more accurate players are more likely to stay alive longer.
- **Raw Number of Kills**: While it is less powerful than the killPlace, removing this column has a significant impact on accuracy. This column helps the model see gaps between players, which may be covered up by killPlace.
- **Points**: We talked about this column in the Data Set Basics section. This is a measure of player skill, which offhand I would've expected to be more important. It perhaps indicates that being a skilled player will help, but ultimately there is a lot of variability in matches.


Duos Model Variable Importance


Squads Model Variable Importance

Above are the models for duos and squads. As it turns out, we see the same variables as important in both graphs, however their rankings are slightly different. Also, all columns from the solo dataset are seen again here. In addition, we have some newcomers.

- **Team walk Distance:** The sum of the walk distance for all players on a team. It is not surprising that this is important, given our discussion on walk distance above.
- **Team Boosts and Team Weapons**: The sum of the number of boosts and weapons acquired for the entire team. Again, these are not surprises based on our solo model explanation.
- **Team Heals**: A new concept we haven't hit on before. In team games, it is possible to heal your teammates. Not only does this help players live longer, but it is also an indicator of teamwork.
- **Team Walk Diff**: A feature I engineered in which we look at the difference in walk distance between the max walk distance on the team and the minimum walk distance on the team. This is a measure of how uniform the team's performance was. A team where one member goes down early is generally going to perform worse than a team that sticks together.
- **Group Members**: Another interesting addition. This is where the previously mentioned 'disconnect' scenario comes into play. A large number of matches have at least one 'group' of players who disconnected early in the game (perhaps they were lost when being transferred to the game server). So typically, if you have a large number of group members (larger than the standard number for the given match's type), it is highly likely that your group finished in last, because you were disconnected from the game.

Conclusion

Overall this project was very interesting and opened my eyes to a number of challenges in data science, and interesting solutions to them. We were able to go from some very simple baselines and build up a more complex model with plenty of interesting feature engineering and optimizations. In doing so, we we're able to cut our mean absolute error essential in half when compared to our baseline simple models.

There are still places we can go with this model though to look for improvements. The next step would be to start to investigate more complex models. Using a neural network or a boosted algorithm could probably provide some easy gains over our existing solution, using our existing inputs.

References

Competition data: https://www.kaggle.com/c/pubg-finish-placement-prediction/data

Reference for code to create feature importance graphs:
https://stackoverflow.com/questions/49105358/plot-feature-importance-computed-by-ranger-function