Machine Learning Engineer Nanodegree Capstone Project: PUBG[1] Finish Placement Prediction

Paco Hobi

January, 2019

I. Definition

Project Overview

The *PUBG Finish Placement Prediction* is a Kaggle competition about the highly successful battle royale game PUBG, which has also become a big esports[2] game. As the esports industry continues to grow its unsurprising that machine learning will be used to train and increase the performance of esports players.

A high profile machine learning project in the game industry is OpenAI Five, which is creating Dota 2[3] bots that already have consistently beaten the top human players. The machine learning bot that the OpenAI team has created has not only learnt advanced game mechanics and strategies, but it also has come up with new strategies that were unknown to the most experienced human players. Projects like this one and others can be used by esports players to train and to discover new strategies.

Problem Statement

In a PUBG game players compete against each other and get ranked at the end of the game based on how many other players are still alive when they are eliminated. In game, players can pick up different munitions, drive vehicles, swim, run, shoot, etc. For this project we create a model that predicts player ranks based on their final game stats, on a scale of 1 (first place) to 0 (last place).

To solve the problem we will build a Random Forest regressor model that is able to predict the `winPlacePerc` from the match statistics. To train the model and measure the performance of the predictions we will compare its predictions with the actual `winPlacePerc` of part of the dataset.

The dataset has 4,446,966 data points with 29 variables each. The dataset includes data from different game types, but we will be only working with the data of the game type `solo-fpp` (no teammates and first person), because it is one of my favorite modes and also one of the modes use in competitive mode.

[1] PlayerUnknown's Battlegrounds is an online multiplayer battle royale game developed and published by PUBG Corporation, a subsidiary of South Korean video game company Bluehole.

[2] Esports is a form of competition using video games. Most commonly, esports takes the form of organized, multiplayer video game competitions.

[3] Dota 2 is a multiplayer online battle arena video game developed and published by Valve Corporation.

We will remove variables that are no longer needed when only checking solo games: `matchType`, `teamKills`, `revives`, `assists`, `numGroups` and `DBNOs`. We will also remove other variables that are not necessary, like `id`, `matchId`, `groupId`; and also the ranking variables as we know that these are being deprecated from the API because of their unreliability: `killPoints`, `rankPoints`.

By only considering the `solo-fpp` games and removing the unnecessary variables we are left with 536,761 data points with 17 variables each.

Many of the variables are skewed (see Figure 1), so we will experiment with feature scaling using Box-Cox test and feature normalization. We will also check if we benefit from removing outliers.

Once we have the dataset ready we will train a Random Forest regressor and do hyperparameter tuning using an exhaustive grid search.

Finally we have some ideas for engineered features, and we will check if we obtain better results by adding this new features.

Metrics

We will use the Mean Absolute Error (MAE) as the evaluation metric. MAE is a linear score function and therefore it penalizes huge error less than, for example, the Mean Squared Error. Therefore it is less sensible to outliers. [4]

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i|$$

II. Analysis

Data Exploration

For this competition Kaggle provides a dataset including over 65,000 games' worth of player anonymized data. This dataset was built from official publicly available data provided by the PUBG developers through the PUBG Developer API.

As discussed before, we only work with a subset of the data, specifically only the data points corresponding to solo first person matches. This also allows us to remove multiple columns only relevant for group matches: `Id`, `groupId`, `matchId`, `matchType`, `killPoints`, `rankPoints`, `teamKills`, `revives`, `assists`, `numGroups`, `maxPlace` and `DBNOs`.

[4] "How to select the Right Evaluation Metric for Machine Learning Models: Part 1 Regression Metrics"

```python
data = pd.read_csv('data/train_V2.csv')

# Filter the solo fpp matches
data = data_raw[data_raw.matchType == 'solo-fpp']

# Drop unnecessary variables
unnecessary_cols = ['Id', 'groupId', 'matchId', 'matchType', ‘killPoints', ‘rankPoints',
                    'teamKills', 'revives', ‘assists’, 'numGroups', 'maxPlace', 'DBNOs']
data = data.drop(unnecessary_cols, axis=1)

# There is a row with winPlacePerc set to NaN
data = data.dropna()

data.to_csv('data/pubg_dataset.csv', index=False)
```

The remaining variables in the dataset are: [5]

- `boosts`: Number of boost items used.
- `damageDealt`: Total damage dealt, with self inflicted damage subtracted.
- `headshotKills`: Number of enemy players killed with headshots.
- `heals`: Number of healing items used.
- `killPlace`: Ranking in match of number of enemy players killed.
- `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.
- `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.
- `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”.
- `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.

In Table 1 we can see that for all variables except `matchDuration` and `killPlace` the mean and median vary significantly, indicating a large skew. We can also see this visually in Figure 1. To address this we will apply feature scaling.

[5] Variables descriptions provided by Kaggle

	count	mean	std	min	25%	50%	75%	max
boosts	536761	1.059	1.792	0	0	0	2	28
damageDealt	536761	118.919	158.346	0	0	75.79	171	2305
headshotKills	536761	0.252	0.633	0	0	0	0	16
heals	536761	1.010	2.383	0	0	0	1	63
killPlace	536761	48.026	27.580	1	24	48	72	100
kills	536761	0.935	1.496	0	0	0	1	23
killStreaks	536761	0.482	0.556	0	0	0	1	5
longestKill	536761	21.701	45.717	0	0	0	22.06	940.1
matchDuration	536761	1577.082	249.372	1044	1377	1439	1854	2199
rideDistance	536761	403.210	1270.646	0	0	0	0	40710
roadKills	536761	0.003	0.062	0	0	0	0	4
swimDistance	536761	4.825	32.658	0	0	0	0	1974
vehicleDestroys	536761	0.003	0.061	0	0	0	0	5
walkDistance	536761	945.309	1075.043	0	97.09	467.7	1577	25780
weaponsAcquired	536761	3.487	2.328	0	2	3	5	153
winPoints	536761	563.691	730.822	0	0	0	1491	1922
winPlacePerc	536761	0.492	0.295	0	0.2366	0.4891	0.7474	1

Table 1. Statistics of the reduced dataset.

Exploratory Visualization

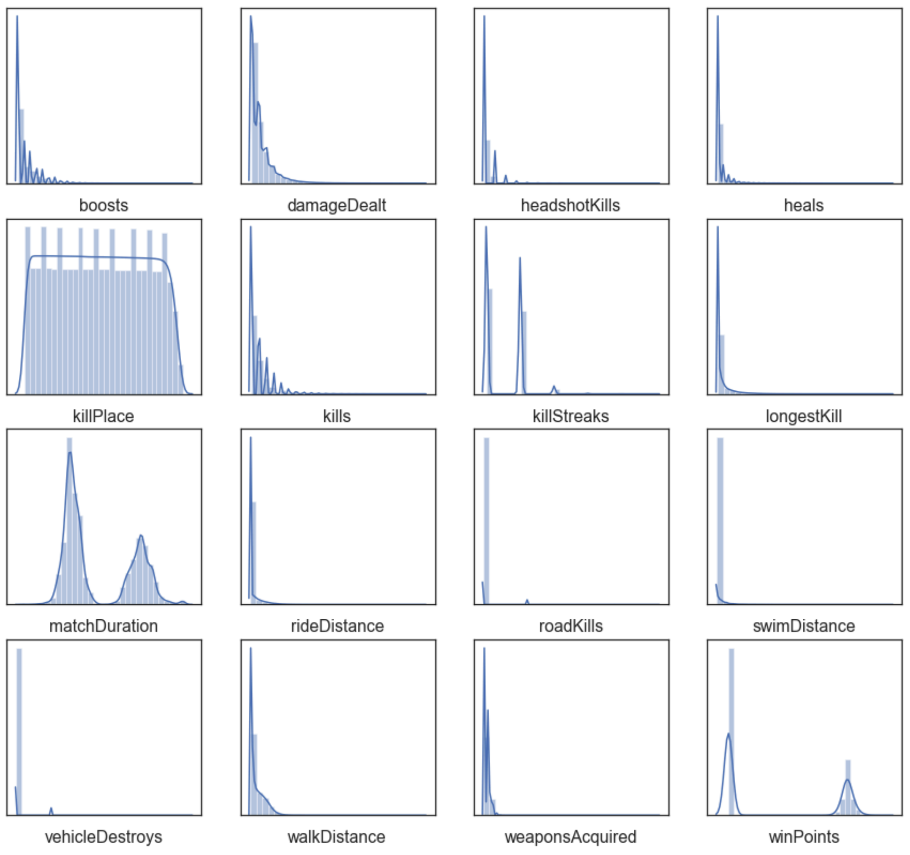

Figure 1. Distribution of the 16 features in the reduced dataset.

In Figure 1 we can see the distribution plot of each of the variables. It shows that most of the variables seem to be skewed towards the lower bound. This makes sense with the domain of the data: in PUBG matches the majority of players perform poorly or normally, but there is always a set of players that vastly outperforms the majority the rest. It is also not abnormal for 10-40% of the players to be eliminated in the first 10 minutes of a match, therefore producing close to 0 in all the variables for their data points.

Figure 2 shows box plots for each of the variables. Every point outside of the whiskers could be considered an outlier and, as we see in the plot, there are many for most of the variables. The reasons for this are the same as for Figure 1: a big percentage of players die very early on the match, the majority of the rest perform poorly or normally, and a small group perform vastly superiorly.

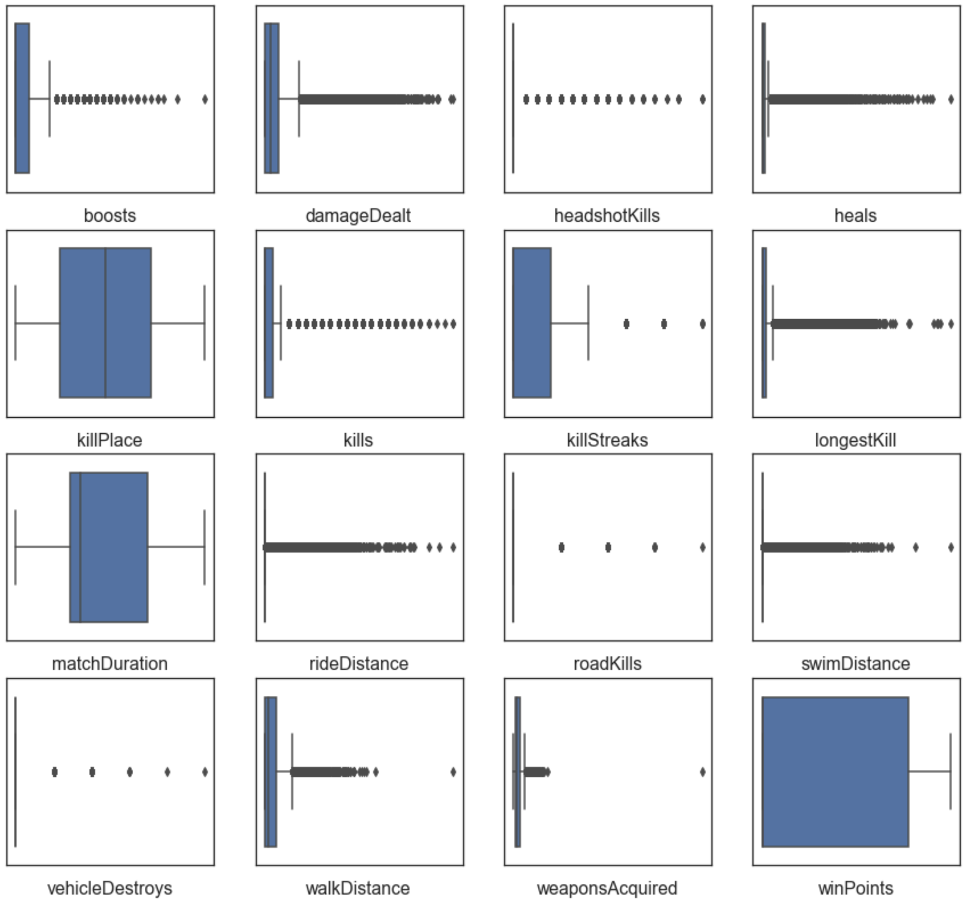

Figure 2. Box plot of the 16 features in the reduced dataset.

Algorithms and Techniques

In the previous sections we concluded that most variables have many outliers. To find and remove them we will apply the IQR[6] rule, which is consider a good rule of thumb. The IQR rule considers a point to be an outlier if the variable's value is greater than $Q_3 + 1.5 \times IQR$ or smaller than $Q_1 - 1.5 \times IQR$, where Q_3 and Q_1 are the third and first quartiles, respectively.

We also saw how the variables present a large skew toward the lower bound, to address this we will apply a Box-Cox power transformation[7] to each of the variables using the SciPy implementation with the default arguments.

Our final model will be a random forest regressor. Random forest is an ensemble method, which means that it combines multiple weak learners to form a strong learner. In the case of the random forest the weak learners are decision trees. To train the weak learners the general technique of bagging is used: for each tree a sample of the dataset is randomly selected with replacement[8] and used to train the decision tree, the trees are then combined by averaging their output. Random forest uses a slight variation in the decision trees: a random subset of the features if selected at each candidate split in the learning process. This technique is called feature bagging, and it attempts to reduce the correlation between the weak learners.

Benchmark

As a benchmark model we are using an AdaBoost regressor with the default scikit parameters and using the reduced dataset without preprocessing. AdaBoost gives us a good benchmark to which to compare our own model.

[6] Interquartile Range is a measure of statistical dispersion, being equal to the difference between 75th and 25th percentiles.

[7] Box-Cox power transformation article on Wikipedia

[8] "With replacement" means that a data point may appear multiple times in the one sample.

```
from sklearn.ensemble import AdaBoostRegressor
from sklearn.model_selection import train_test_split

import config
from utils.model import train_predict


X = data_raw.drop('winPlacePerc', axis=1)
y = data_raw['winPlacePerc']

regressor = AdaBoostRegressor(random_state=config.RANDOM_STATE)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=config.RANDOM_STATE
)

results = train_predict(regressor, X_train, y_train, X_test, y_test)

print(
    f'Train time:      {results["train_time"]:f}\n'
    f'Prediction time: {results["pred_time"]:f}\n'
    f'MAE train:       {results["mae_train"]:.4f}\n'
    f'MAE test:        {results["mae_test"]:.4f}'
)
```

```
Train time:      31.4793
Prediction time:  1.7218
MAE train:        0.0730
MAE test:         0.0731
```

As we see in the code sample AdaBoost gives us a MAE of 0.0731 for the test split. We will use this score as our benchmark.

III. Methodology

Data Preprocessing

a. Remove outliers

As mentioned before, we will use the IQR rule to remove the outliers. To prevent removing too many data points, we only remove the ones considered outliers for 5 or more features.

```
def remove_outliers(df):
    new_df = df.copy().reset_index(drop=True)

    outliers_counter = Counter()

    # For each feature find the data points with extreme high or low values
    for col in new_df.columns[:-1]:
        # Calculate Q1 and Q3 for the given feature
        Q1 = np.percentile(new_df[col], 25)
        Q3 = np.percentile(new_df[col], 75)

        # Calculate the outlier step
        step = 1.5 * (Q3 - Q1)

        # Calculate outliers
        outliers_table = new_df[(new_df[col] < Q1 - step) | (new_df[col] > Q3 + step)]
        outliers_counter.update(outliers_table.index)

    # Select the indices of the outliers
    outliers = [idx for idx, count in outliers_counter.items() if count >= 5]

    # Remove the outliers
    new_df.drop(new_df.index[outliers], inplace=True)
    new_df.reset_index(drop=True)

    return new_df
```

b. Feature scaling

To remedy the skewed features we use SciPy's implementation of the Box-Cox power transformation. To apply Box-Cox the values have to be greater than 0, and we know that many of our features can have 0 as values, but that they never are negative. Therefore to be able to apply Box-Cox we add 1 to each value.

```
SKEWED = ['boosts', 'damageDealt', 'headshotKills', 'heals', 'kills', 'killStreaks',
          'longestKill', 'rideDistance', 'roadKills', 'swimDistance', 'vehicleDestroys',
          'walkDistance', 'weaponsAcquired', 'winPoints']

def scale_skewed_features(df):
    new_df = df.copy()
    new_df[SKEWED] = new_df[SKEWED].apply(lambda x: stats.boxcox(x + 1)[0])
    return new_d
```

c. *Normalize features*

The different features have wildly different value ranges: on one hand `roadKills` varies from 0 to 4, and on the other hand `rideDistance` ranges between 0 and 40,710. To prevent possible problems from this very different ranges we will scale all features to have values between 0 and 1 using SciKit's MinMaxScaler.

```
from sklearn.preprocessing import MinMaxScaler

def min_max_scale(df):
    new_df = df.copy()

    # Don't scale the winPlacePerc
    columns = new_df.columns[:-1]

    scaler = MinMaxScaler()
    new_df[columns] = scaler.fit_transform(new_df[columns])

    return new_df
```

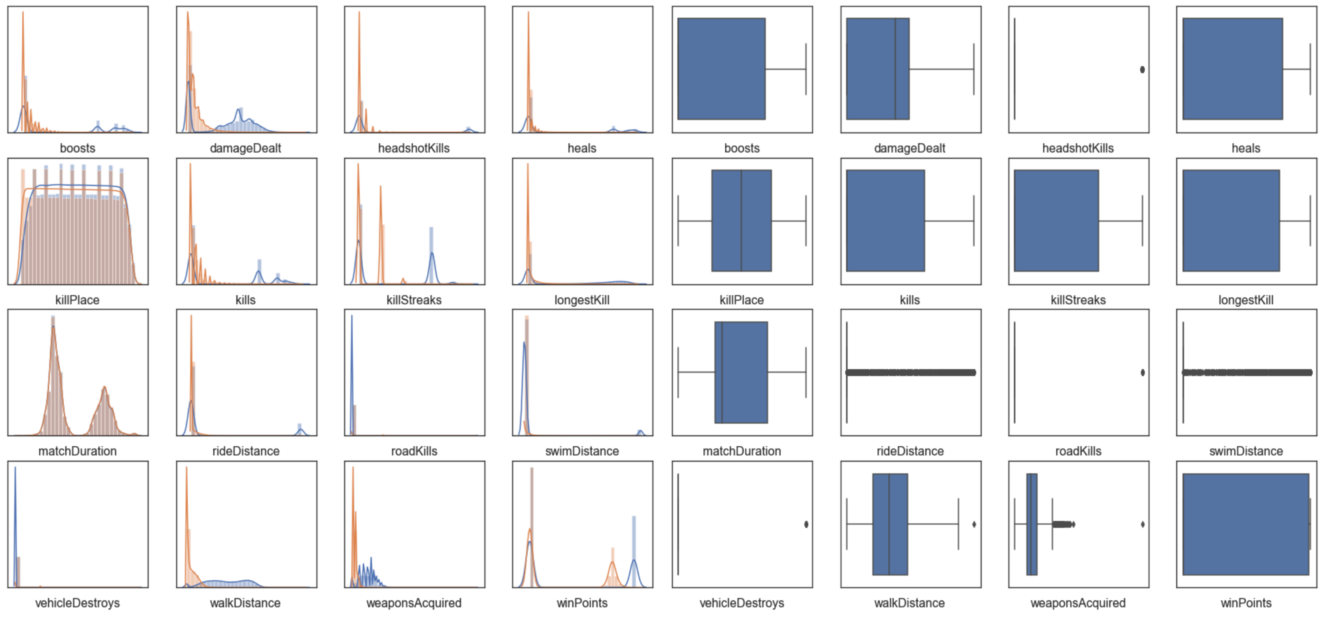

Figure 3. Here we can see both the distribution and box plots of the 16 features after the preprocessing steps. In the distribution plots the blue lines are after the preprocessing, and the orange lines are before the preprocessing.

Implementation

The code for the different preprocessing steps are shown above. Some complications the appeared when coding the preprocessing steps are:

- For the removal of outliers we used the IQR rule. Many of the data points have values that are considered outliers using this rule, but if we removed all the data points with outliers in any of their features we would be removing the majority of points. Therefore we decided to only remove the data points that had at least 5 features considered outliers by the IQR rule. This is a safer approach, but it has the down side that some data points that actually are outliers might no be removed.

- For the feature scaling we used SciKit's Box-Cox implementation, which has a gotcha: it requires the values to be positive. To work around this we shifted all values by one, i.e. we added one to all the values. Doing this allowed us to apply Box-Cox to our dataset.

After creating the smaller dataset and passing it through the described preprocessing steps, it was time to train our model. To do this we used SciKit's RandomForestRegressor as the regressor and mean_absolute_error to score the predictions. The process of training, predicting and scoring with SciKit's provided classes and functions is well defined and no problems arose coding this part.

Using the preprocessed dataset and a RandomForestRegressor we obtained the base performance for this regressor:

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

import config
from utils.model import train_predict


X = data_normalized.drop('winPlacePerc', axis=1)
y = data_normalized['winPlacePerc']

regressor = RandomForestRegressor(random_state=config.RANDOM_STATE)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=config.RANDOM_STATE
)

results = train_predict(regressor, X_train, y_train, X_test, y_test)

print(
    f'Train time:      {results["train_time"]:7.4f}\n'
    f'Prediction time: {results["pred_time"]:7.4f}\n'
    f'MAE train:       {results["mae_train"]:7.4f}\n'
    f'MAE test:        {results["mae_test"]:7.4f}'
)
```

```
Train time:      28.9262
Prediction time:  3.1824
MAE train:        0.0191
MAE test:         0.0484
```

We can see that the MAE is already better than our benchmark: 0.0484 compared to 0.0731. We should still be able to increase out score by refining our regressor.

Refinement

The first step we will take to improve our model is parameter tuning. We will do an exhaustive grid search using SciKit's GridSearchCV and 3 KFold cross validation. We are testing a total of 72 parameter combinations, that together with the 3 folds total to 216 fits.

```
from sklearn.metrics import mean_absolute_error, make_scorer
from sklearn.model_selection import GridSearchCV

# Separate features
X = data_normalized.drop('winPlacePerc', axis=1)
y = data_normalized['winPlacePerc']

# Init grid
param_grid = {
    'n_estimators': [50, 100, 150, 200],
    'max_features': ['auto', 'sqrt'],
    'min_samples_leaf': [2, 4, 8],
    'min_samples_split': [2, 4, 8],
}

regressor = RandomForestRegressor(random_state=config.RANDOM_STATE)
scorer = make_scorer(mean_absolute_error)
grid = GridSearchCV(
    estimator=regressor,
    param_grid=param_grid,
    scoring=scorer,
    cv=3, verbose=50, n_jobs=-1
)
grid = grid.fit(X, y)

print(f'Best parameters are {grid.best_params_}')
```

```
Best parameters are {‘max_features’: 'auto', 'min_samples_leaf': 8, 'min_samples_split':
2, 'n_estimators': 200}
```

Using this parameters we obtain a MAE of 0.0456, compared to the 0.0484 we obtained before parameter tuning. That is an improvement of 0.0028, or almost 6%.

Next we are going to try adding some engineered features which I think could be significant based on my personal experience with the game. In my opinion the best players have high mobility and high amount of precision kills. We already have three different distance features about distance traveled, and the headshots and kills features, but the model might benefit from combining them. We will add the following features:

- `headshotPerc`: percentage of the kills that were headshots (`headshotKills / kills`).
- `totalDistance`: total distance traveled (`walkDistance + rideDistance + swimDistance`).

```
# Add engineered features
data['totalDistance'] = data['walkDistance'] + data['rideDistance'] +
data['swimDistance']
data['headshotPerc'] = data['headshotKills'] / data['kills']

# Fix NaN from division by zero
data['headshotPerc'].fillna(0.0, inplace=True)

# Move the winPlacePerc column to the end
cols = data.columns.tolist()
cols.remove('winPlacePerc')
cols.append('winPlacePerc')
data = data[cols]
```

And after applying the same preprocessing steps and using the same tuned parameters for the random forest we obtain a MAE score of 0.0451, compared to the 0.0456 we obtained before adding the engineered features. That is an improvement of 0.0005, or almost 1.1%.

IV. Results

Model Evaluation and Validation

Our final model is a SciKit Random Forest Regressor with parameters:

- `n_estimators = 200`
- `max_features = 'auto'`
- `min_samples_leaf = 8`
- `min_samples_split = 2`

We settled on this parameters after they gave the best MAE score during an exhaustive grid search of possible parameter values.

Before data points can be used to train or can be given to the regressor to predict the `winPlacePerc`, two engineered features are added and some preprocessing is done.

The engineered features are:

- `headshotPerc`: percentage of the kills that were headshots.
- `totalDistance`: total distance traveled.

And the preprocessing steps are:

- Removing outliers: in Figure 2 we can see in the Box plots that many points can be considered outliers. We removed many of those points using the IQR rule.
- Scaling features: in the distribution plots of Figure 1 we see that many of the features are skewed towards the lower values, and to address this we apply a Box-Cox power transformation.
- Feature normalization: to address the wildly different ranges of values seen in Table 1 we normalize all the features using a min max converter.

We performed a Variance-based sensitivity analysis with the Sobol method[9] to see how changes in the data affect the result. As we can see in Table 2 only a few of the features seems to have an effect on the variance of winPlacePerc. Some of the total-effects are considerably higher to the respective firs-order indices, which indicates that there are relevant higher-order interaction occurring.

[9] Sobol method article on Wikipedia

feature	S1	ST
killPlace	0.588	0.678
walkDistance	0.302	0.370
matchDuration	0.001	0.033
kills	-0.002	0.012
killStreaks	-0.002	0.008
totalDistance	0.002	0.007
boosts	0.003	0.003
weaponsAcquired	-0.000	0.001
longestKill	-0.000	0.000
damageDealt	0.001	0.000
winPoints	0.001	0.000
swimDistance	0.000	0.000
heals	-0.001	0.000
rideDistance	-0.000	0.000
headshotPerc	0.000	0.000
headshotKills	-0.000	0.000
roadKills	0.000	0.000
vehicleDestroys	0	0

Table 2. Sobol method's First-order indices (S1) and Total-effect indices (ST) for the features.

Justification

Our final model together with the engineered features and preprocessing we obtain a MAE of 0.0451, compared to the benchmark MAE of 0.0731. That is a difference of 0.028, or ~38%.

Because `winPlacePerc` is a percentage, a MAE of 0.0451 is equivalent to saying that on average the predicted place will be 4.5% from the actual place. Considering that matches usually have anywhere from 70 to 100 players, this would be an average error of 3-5 places. While I would prefer having an average error of only 1-2 places, I would still consider 3-5 places off a satisfactory solution for the problem.

V. Conclusion

Free-Form Visualization

In Figure 4 we can see a scatterplot comparing the predicted and actual `winPlacePerc` values. The points have been color graded to facilitate the visual recognition of the absolute error: the higher the absolute error the darker the points. Also, the closer the points are to the blue diagonal line the smaller the error. Knowing this we can instantly see that the predictions of the Tuned Random Forest model have smaller errors than the ones of the benchmark AdaBoost model, as they are closer to the diagonal line and the points are much lighter.

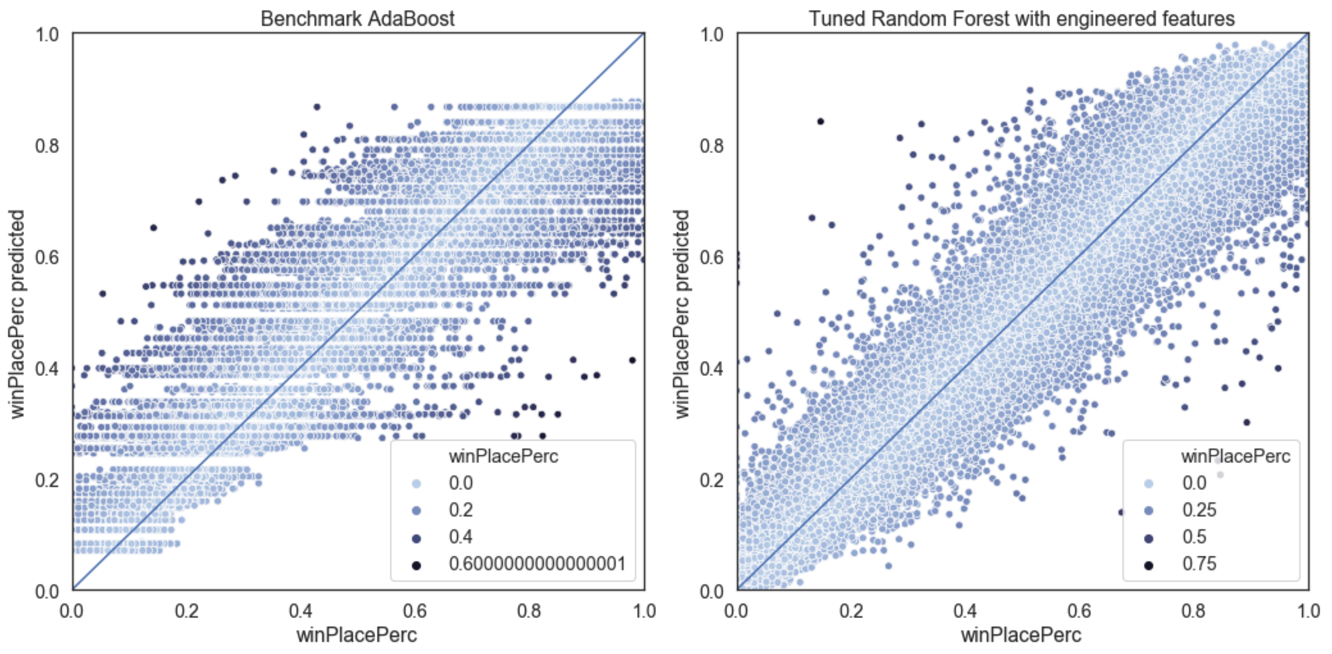

Figure 4. Scatter plots comparing the predicted winPlacePerc with the actual values for both the benchmark AdaBoost model and the tuned Radom Forest model with engineered features. The darker the color the greater the prediction absolute error.

Reflection

I found this project really interesting and also the the domain very fascinating. As gaming is one of my interests the process of exploring and resolving this problem was very enjoyable. I am thankful to PUBG for providing the data and to kaggle for providing the prepared dataset and the interesting challenge.

I found the dataset itself very interesting, as it gave me insight into the general performance and play style of other players in the game. I found myself creating multiple plots and exploring the different features, trying to place myself in the different distributions of players.

On the other hand there were also some difficulties. For example dealing with outliers and the long running times to tune the parameters. I will talk more about this in the next section.

While I would have preferred to obtain a lower MAE final score, I am still satisfied with the results we obtained with our model. I think the final model could be applied to similar problems, but it should be trained again and a deeper outliers identification method could be beneficial.

Improvement

Two areas were I am quite certain we could improve in our model are the removal of outliers and on the fine tuning of the random forest hyperparameters.

While I think that our approach to outliers removal with the IQR rule is a good rule of thumb, I am not sure it is the best approach for this use case. The distribution of the players is naturally skewed to the bottom because the huge gap of skill between the average player and the good players, and I think our approach is removing many of the high performing players, which I think is negatively impacting the model. Another thing to consider is the presence of cheaters in the game (and therefore in the dataset), i.e. players that use malicious programs to gain an advantage in the game like aiming for you, the ability to fly or to see all other players. I think it could be beneficial to instead of using something like the IQR rule, use our domain knowledge to try to only exclude suspicious players.

Another area of improvement is the hyperparameter tuning. Due to the low computational power at my disposal I was only able to do a quite reduced search for optimal hyperparameters. With faster hardware it would be possible to do wider searches in a faster time, allowing for more experimentation.

Apart from this areas, I think that with more advanced knowledge the model could be optimized even further. Therefore I would say that if we used our final model as a new benchmark it would be possible to produce even better solutions.