Analyzing and Predicting PUBG Player Ranking

By

Soumya Das

Under guidance of

Mohit Gupta

and trainers at GeeksforGeeks

Date: 4th July, 2024

Abstract – This report performs statistical analysis on the PUBG player performance dataset released by Kaggle in 2018. Key insights into the data were identified before employing the CatBoost model for regression to predict match outcomes. Each player's ranking at the end of a match is forecasted with high accuracy, achieving an RMSE of 0.08 and an R^2 score close to 1. The prediction model's performance is discussed in detail, and key findings from the analysis are highlighted.

TABLE OF CONTENTS:

1. Introduction

To understand the significance of PUBG, one must be familiar with the idea of a 'battle royale' game. 'Battle royale' is a concept originating from a novel by Takami published in 1999 that was adapted into a film in the following year. It refers to combat between a large number of individuals or teams where the objective is to eliminate all others and become the only player or team left. The movie was controversial, but overall received well in Japan, inspiring other movies, writ-ing and manga. However, it is the gaming community that has strongly embraced the concept due to the deep sense of immersion that video games allow working extremely well in a competitive battle royale setting. a virtual arena where up to 100 players engage in fierce combat. The objective? Be the last one standing. Think of it as a digital Hunger Games, minus the dystopian future.

In a PUBG game, up to 100 players start in each match (matchId). Players can be on teams (groupId) which get ranked at the end of the game (winPlacePerc) based on how many other teams are still alive when they are eliminated. In game, players can pick up different munitions, revive downed-but-not-out (knocked) teammates, drive vehicles, swim, run, shoot, and experience all of the consequences -- such as falling too far or running themselves over and eliminating themselves.

Origins: Brendan “PlayerUnknown” Greene birthed PUBG from an Arma 3 mod. The name? A blend of his nickname and the Japanese film “Battle Royale.”

Steam Sensation: PUBG stormed onto Steam in 2017, selling over 15 million copies during early access. It even set a record with one million simultaneous players!

Influence: PUBG’s success spawned countless battle royale games. It’s the trendsetter at the gaming party.

2. Game Mechanics

The key components of a game of PUBG are:

- **The Map:** Four unique maps are available as of writing this report. Each map has a mixture of terrains with key settlements/towns that tend to have a higher con-centration of weapons, items and thus players.

- **Weapons & Items:** A wide array of weapons, healing items and boosts are scattered around the map in a semi-random distribution. Weapons are more concentrated around certain locations on each map.

- **Vehicles:** A wide array of drivable land and sea vehicles are available. Each player has a limited number of slots for weapons and items; although finding 'bag' items like backpacks and allow players to increase inventory size.

- **The Players:** Up to 100 players are deployed per match, either solo or in teams of 2, 3 or 4.

- **The Playzone (The Circle):** The main driving force of a PUBG match. This is a random circular area of the map that players must remain in. If outside the circle, players take damage and eventually die. The playzone shrinks in 8 stages, each stage is faster and more damaging than the previous. Before the actual playzone (represented by a blue circle) shrinks, a white circle on the map shows players where the next playzone will be. Each circle representing the new playzone is randomly generated.

A standard PUBG match begins with up to 100 players joining a lobby. The players are then transported in a plane that travels in a randomly generated straight line across an island. Players can choose to exit the plane and parachute down to the surface of the map at any point; if any player does not jump out, they are automatically ejected from the plane at the far boundary of the map.

Upon leaving the plane, players do not have any items or weapons; just their bare hands for combat. Thus, selecting where to land at the start of the match is a crucial. Some players/teams tend to avoid certain settlements, as they tend to have a higher concentration of weapons and players; thus, avoiding a highly competitive and dangerous initial few minutes. Some are more

aggressive and prefer to race to these locations in search of better loot. They often have to engage opponents immediately; this is called 'hot drop-ping'.

Once on the ground, the initial priority is to scavenge for weapons and amour whilst getting an idea of where enemy players might be and where the playzone is shrinking to. The playzone begins with a very large circle; diameter of around 4500m. Players are shown the next stage; around 3000m with a white circle, before the actual zone begins to shrink. This repeats with smaller and smaller playzones, the smallest being just 46.4m, until just one player or team is left.

As the playzone gets smaller, players are forced to stay on the move. They are pushed into a more concentrated area, resulting in more encounters. Combat is mainly based around firearms, although hand-to-hand combat may also take place. Once a player takes sufficient damage they may be 'downed'; immobilized until a team-mate revives them. If no revival takes place, or if a player is playing alone, they are 'killed' and given their ranking according to how many players are left alive. They are not allowed to rejoin the match.

Evidently there are countless strategies players can adopt, due to the wide variety of items and vehicles available, the vast and diverse maps as well as the unpredictability that comes with 100 competitive human players. For more information, Chris Carter from Polygon has an insightful article on the basic concepts and strategies of PUBG.

3. Problem Statement

The objective of this project is to predict the performance of PUBG players based on their in-game statistics. By leveraging machine learning techniques, we aim to forecast a player's ranking at the end of a match. This can help in understanding the factors contributing to higher performance and strategizing gameplay for better results.

4. Data Description

The dataset is provided with a large number of anonymized PUBG game stats of **4.5 million** players formatted so that each row contains one player's post-game stats. Each record is made up of 29 attributes that describe one player's statistics throughout one match.

The following is a short description of the attributes used in this report:

- **DBNOs** - Number of enemy players knocked.
- **assists** - Number of enemy players this player damaged that were killed by teammates.
- **boosts** - Number of boost items used.
- **damageDealt** - Total damage dealt. Note: Self inflicted damage is subtracted.
- **headshotKills** - Number of enemy players killed with headshots.
- **heals** - Number of healing items used.
- **Id** - Player's Id
- **killPlace** - Ranking in match of number of enemy players killed.
- **killPoints** - Kills-based external ranking of player. (Think of this as an Elo ranking where only kills matter.) If there is a value other than -1 in rankPoints, then any 0 in killPoints should be treated as a "None".
- **killStreaks** - Max number of enemy players killed in a short amount of time.
- **kills** - Number of enemy players killed.
- **longestKill** - Longest distance between player and player killed at time of death. This may be misleading, as downing a player and driving away may lead to a large longestKill stat.
- **matchDuration** - Duration of match in seconds.
- **matchId** - ID to identify match. There are no matches that are in both the training and testing set.
- **matchType** - String identifying the game mode that the data comes from. The standard modes are "solo", "duo", "squad", "solo-fpp", "duo-fpp", and "squad-fpp"; other modes are from events or custom matches.
- **rankPoints** - Elo-like ranking of player. This ranking is inconsistent and is being deprecated in the API's next version, so use with caution. Value of -1 takes place of "None".
- **revives** - Number of times this player revived teammates.
- **rideDistance** - Total distance traveled in vehicles measured in meters.
- **roadKills** - Number of kills while in a vehicle.
- **swimDistance** - Total distance traveled by swimming measured in meters.
- **teamKills** - Number of times this player killed a teammate.
- **vehicleDestroys** - Number of vehicles destroyed.
- **walkDistance** - Total distance traveled on foot measured in meters.-
- **weaponsAcquired** - Number of weapons picked up.
- **winPoints** - Win-based external ranking of player. (Think of this as an Elo ranking where only winning matters.) If there is a value other than -1 in rankPoints, then any 0 in winPoints should be treated as a "None".

- **groupId** - ID to identify a group within a match. If the same group of players plays in different matches, they will have a different groupId each time.
- **numGroups** - Number of groups we have data for in the match.
- **maxPlace** - Worst placement we have data for in the match. This may not match with numGroups, as sometimes the data skips over placements.
- **winPlacePerc** - The target of prediction. This is a percentile winning placement, where 1 corresponds to 1st place, and 0 corresponds to last place in the match. It is calculated off of maxPlace, not numGroups, so it is possible to have missing chunks in a match.

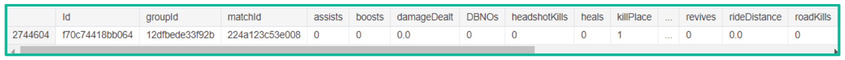
5. Project Methodology

1. Importing Libraries
2. Reading the Data
3. Data Wrangling
4. Feature Engineering
5. ML - CatBoost Model

6. Data Wrangling

6.1 Removing Null Values

The first steps taken involve preprocessing the data in order to clean and better organize the records. Any null values not in essential attributes are identified using the NumPy ‘**.isnull()**’ function.



	Id	groupId	matchId	assists	boosts	damageDealt	DBNOs	headshotKills	heals	killPlace	...	revives	rideDistance	roadKills
2744604	f70c74418bb064	12dfbede33f92b	224a123c53e008	0	0	0.0	0	0	0	1	...	0	0.0	0

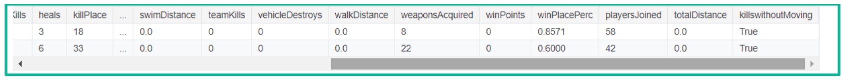
1 such instance was found

6.2 Removing Instances

a. Kills without moving

Players spend between 1-3 minutes [16] at the start of each match parachuting down to the surface. This dataset does not capture the distance covered during this time, and thus, players who somehow get eliminated before landing, or land and fail to move at all have a totalDistance of 0. It is not possible to kill even 1 player if you do not move by atleast 1 unit. Following is mostly used practices by cheaters (ones who interfere with the game's genuine natural processes):

- Aimbots
- Wallhacks
- Triggerbots
- ESP (Extra Sensory Perception)
- Silent Aim



ills	heals	killPlace	...	swimDistance	teamKills	vehicleDestroys	walkDistance	weaponsAcquired	winPoints	winPlacePerc	playersJoined	totalDistance	killswithoutMoving
	3	18	...	0.0	0	0	0.0	8	0	0.8571	58	0.0	True
	6	33	...	0.0	0	0	0.0	22	0	0.6000	42	0.0	True

1535 such instances were found

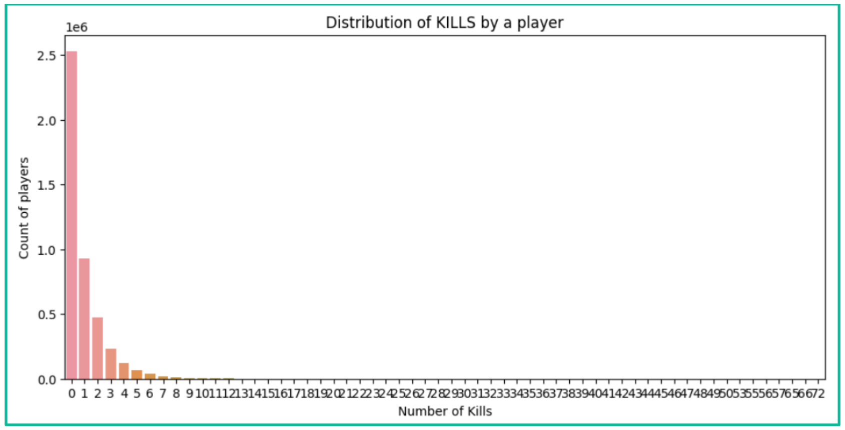
b. Having more than 5 road kills in a match

It takes to be expert among the other players in a match to kill by vehicles only. This data cannot be considered while training a model. Hence dropping the instances from data frame.

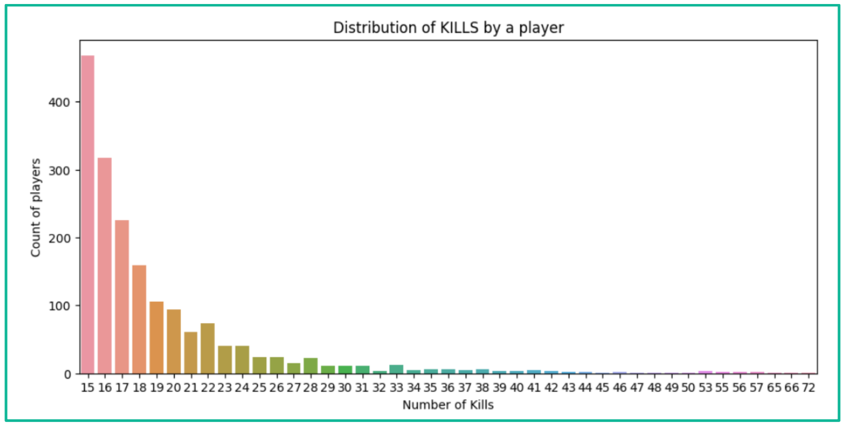
46 such instances were found

c. Kills greater than 20 in a match

Maximum people could kill up to maximum 12 players in a match.



Not many players could kill more than 20 players. Hence, these instances cannot be considered for modelling and hence are dropped.

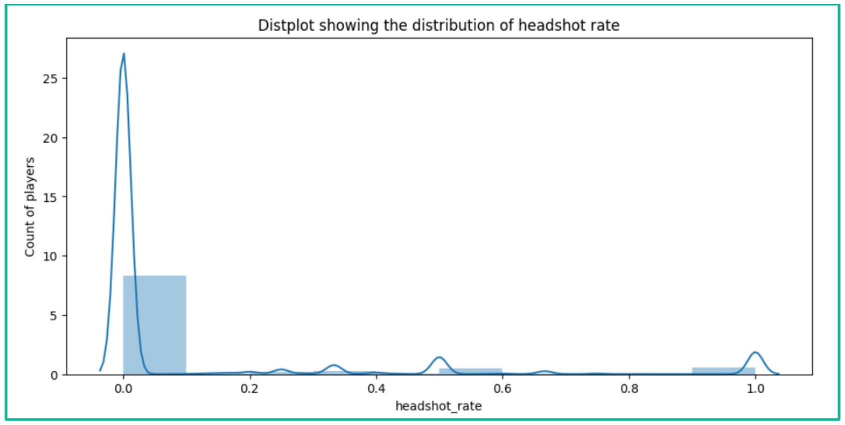


417 such instances were found

d. Kills with headshot rate = 100% and total kills > 5

Headshot rate = Headshot kills / total kills

Killing more than 5 people as headshots where all the shots in a match are headshots is mostly not a general case.

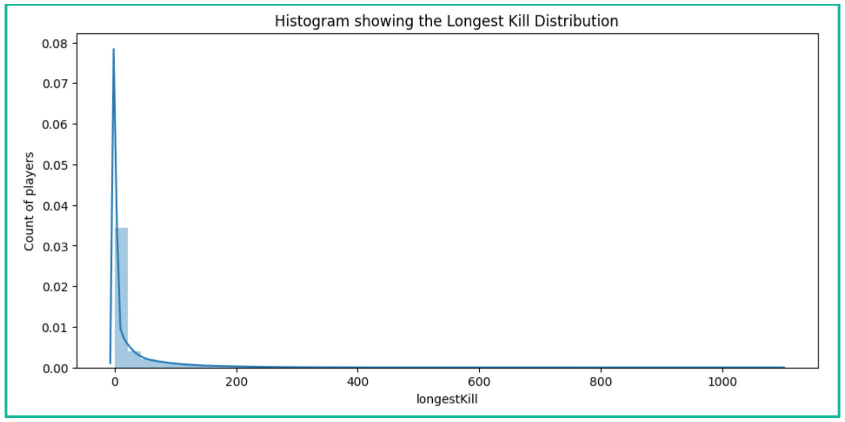


187 such instances were found

e. Kills with longest shot from 500 meters or more

The maximum possible distance that is made possible to snipe from in PUBG is 1km or 1000 meters. However, this is not general case and most of the times, hackers use either of the following to take advantage and win a match:

- Sniper Aimbots
- Bullet Speed/Trajectory Hacks
- No Recoil/No Spread
- Zoom Hacks

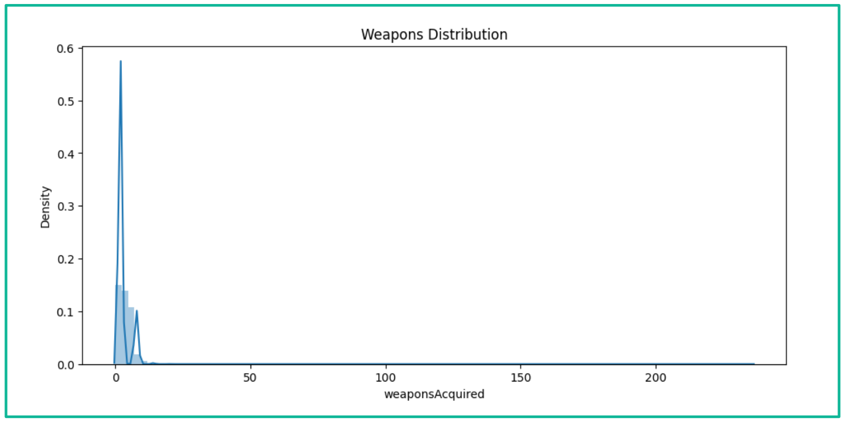


1747 such instances were found

f. Weapon changed more than 15 times in a match

In general, people change up to 10 guns in match (avg. being 5 to 6). But cheaters sometimes use either of the following for unlimited recoil/ guns in a single match:

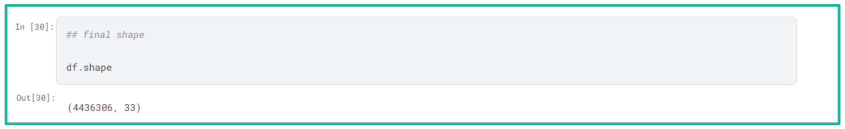
- *Macro Scripts*
- *Rapid Fire Hacks*
- *Input Spoofing*



6809 such instances were found

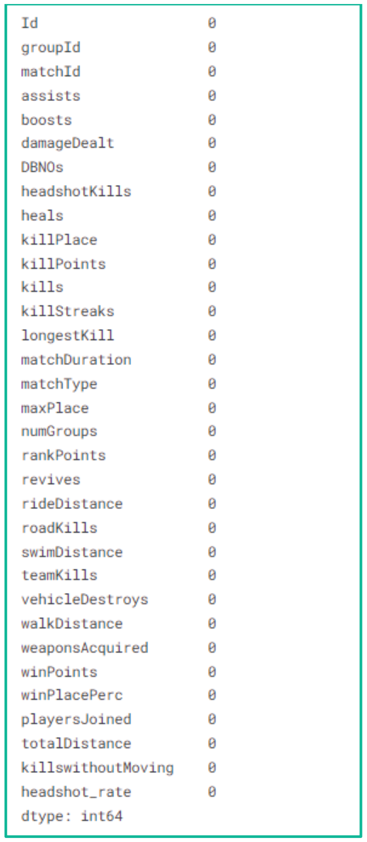
6.3 Exploratory Data Analysis

There were finally 4436306 records picked out with 33 attributes for EDA.



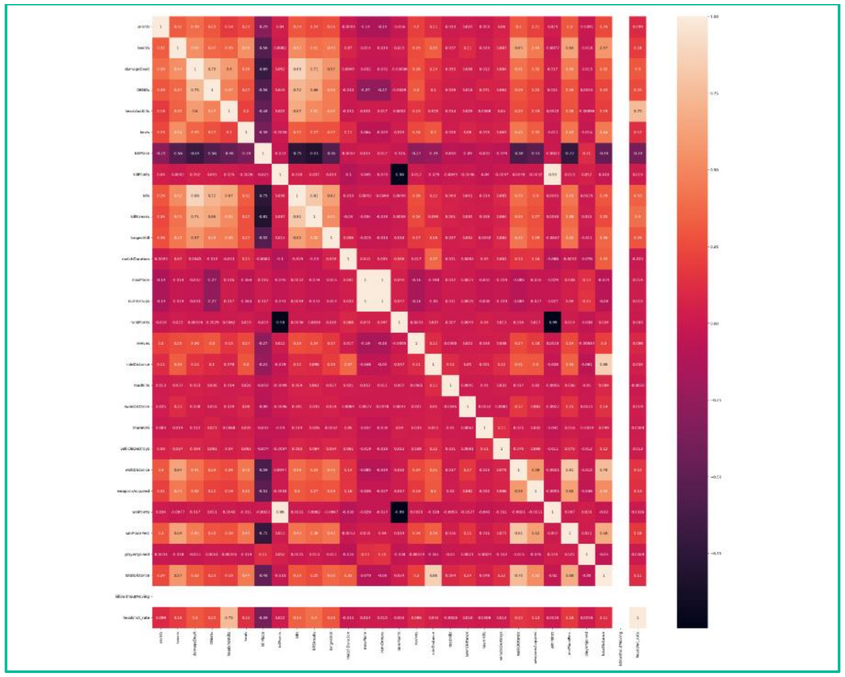
```
In [30]: ## final shape

         df.shape

Out[30]: (4436306, 33)
```

No null values were found in the cleaned dataset.



```
Id                  0
groupId             0
matchId             0
assists             0
boosts              0
damageDealt         0
DBNOs               0
headshotKills       0
heals               0
killPlace           0
killPoints          0
kills               0
killStreaks         0
longestKill         0
matchDuration       0
matchType           0
maxPlace            0
numGroups           0
rankPoints          0
revives             0
rideDistance        0
roadKills           0
swimDistance        0
teamKills           0
vehicleDestroys     0
walkDistance        0
weaponsAcquired     0
winPoints           0
winPlacePerc        0
playersJoined       0
totalDistance       0
killswithoutMoving  0
headshot_rate       0
dtype: int64
```

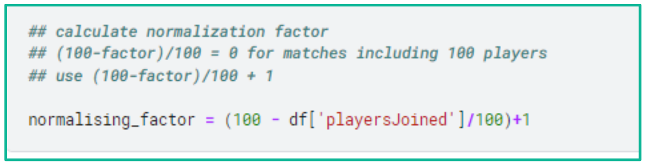
To gain a better understanding of the correlations between attributes, we generate a heatmap of r values. From the heatmap, as shown in the figure below, it appears that playersInMatch does not significantly correlate with the other attributes, which somewhat confirms the intuitive hypothesis.



7. Feature Engineering

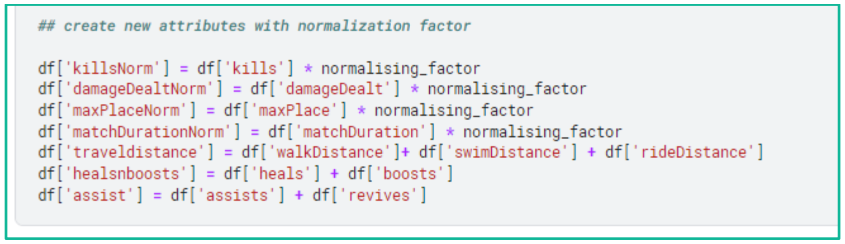
The provided dataset contained a large amount of data with a wide range of values. Hence, new attributes are created from existing features in order to gain further insight into, better organise, and capture the different aspects of each player's performance.

The maximum number of players in a match can be 100. Some attributes for matches where 100 players have joined becomes 0 (100-100). Hence, when normalization is done for such scenario using a normalization factor given below:



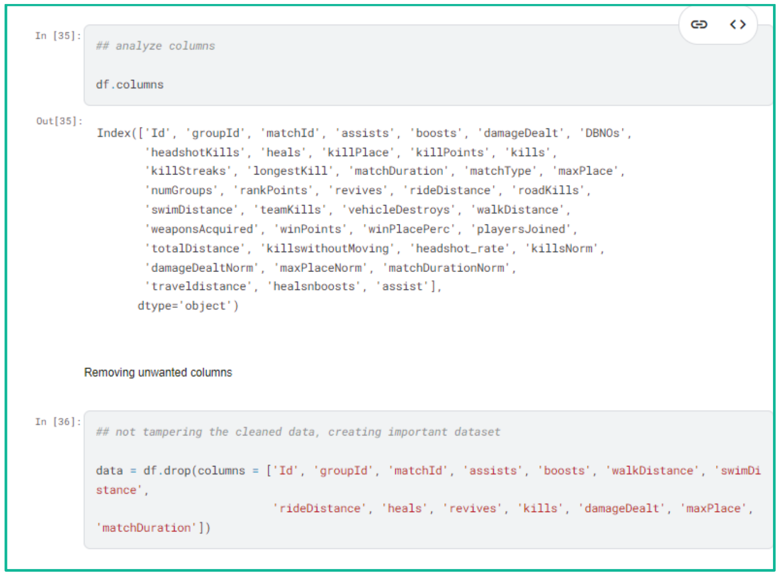
```
## calculate normalization factor
## (100-factor)/100 = 0 for matches including 100 players
## use (100-factor)/100 + 1

normalising_factor = (100 - df['playersJoined']/100)+1
```

New attributes are created with this factor:



```
## create new attributes with normalization factor

df['killsNorm'] = df['kills'] * normalising_factor
df['damageDealtNorm'] = df['damageDealt'] * normalising_factor
df['maxPlaceNorm'] = df['maxPlace'] * normalising_factor
df['matchDurationNorm'] = df['matchDuration'] * normalising_factor
df['traveldistance'] = df['walkDistance']+ df['swimDistance'] + df['rideDistance']
df['healsnboosts'] = df['heals'] + df['boosts']
df['assist'] = df['assists'] + df['revives']
```

To train the model effectively, it is wise to remove any unwanted columns before training our dataset. Hence, I have created a new data frame from the existing data frame to ensure that no data is lost, and also, we do not train our model with excessive unwanted attributes.



```
In [35]: ## analyze columns

         df.columns

Out[35]:
Index(['Id', 'groupId', 'matchId', 'assists', 'boosts', 'damageDealt', 'DBNOs',
       'headshotKills', 'heals', 'killPlace', 'killPoints', 'kills',
       'killStreaks', 'longestKill', 'matchDuration', 'matchType', 'maxPlace',
       'numGroups', 'rankPoints', 'revives', 'rideDistance', 'roadKills',
       'swimDistance', 'teamKills', 'vehicleDestroys', 'walkDistance',
       'weaponsAcquired', 'winPoints', 'winPlacePerc', 'playersJoined',
       'totalDistance', 'killswithoutMoving', 'headshot_rate', 'killsNorm',
       'damageDealtNorm', 'maxPlaceNorm', 'matchDurationNorm',
       'traveldistance', 'healsnboosts', 'assist'],
      dtype='object')
```

Removing unwanted columns

```
In [36]: ## not tampering the cleaned data, creating important dataset

         data = df.drop(columns = ['Id', 'groupId', 'matchId', 'assists', 'boosts', 'walkDistance', 'swimDi
         stance',
                                   'rideDistance', 'heals', 'revives', 'kills', 'damageDealt', 'maxPlace',
         'matchDuration'])
```

8. ML – Catboost Model

We will have to train our model on varied categorical variables. Columns like matchType are in form of labels but such data cannot be used to train models, including CatBoost model. Hence the data needs to be converted into numerical values. For this, I will use One-hot encoding.

8.1 One-Hot Encoding:

One-hot Encoding is used to convert categorical variables into a format that can be provided to machine learning algorithms to improve predictions. This method transforms categorical data into binary vectors, where each category is represented by a unique vector with all elements set to zero except the one corresponding to the category.

On applying One-hot encoding, the categorical variables were converted into binary vectors. For instance, if the "match type" feature includes categories such as "solo," "duo," and "squad," one-hot encoding creates separate binary columns for each type.

Example:

If the "match type" feature had values "solo," "duo," and "squad," after one-hot encoding, the data might look like this:

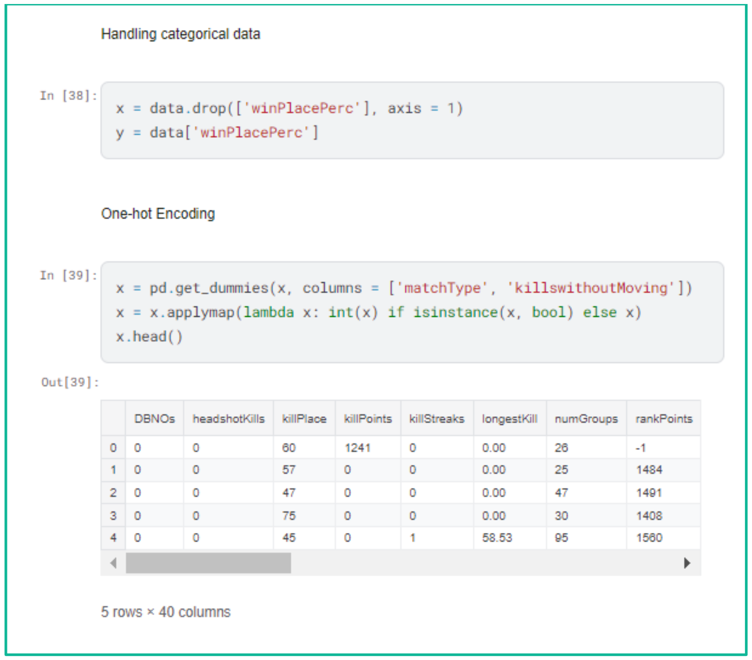
match_type_solo	*match_type_duo*	*match_type_squad*
1	0	0
0	1	0
0	0	1

This transformation enables the model to interpret the categorical data accurately, leading to better performance in predicting player rankings based on various in-game statistics.

Benefits:

Improved Model Performance: One-hot encoding ensures that the model can utilize categorical data effectively.

Avoids Misinterpretation: Prevents the model from misinterpreting categorical values as ordinal, which could otherwise lead to incorrect assumptions.



Handling categorical data

```
In [38]:
x = data.drop(['winPlacePerc'], axis = 1)
y = data['winPlacePerc']
```

One-hot Encoding

```
In [39]:
x = pd.get_dummies(x, columns = ['matchType', 'killswithoutMoving'])
x = x.applymap(lambda x: int(x) if isinstance(x, bool) else x)
x.head()
```

Out[39]:

	DBNOs	headshotKills	killPlace	killPoints	killStreaks	longestKill	numGroups	rankPoints
0	0	0	60	1241	0	0.00	26	-1
1	0	0	57	0	0	0.00	25	1484
2	0	0	47	0	0	0.00	47	1491
3	0	0	75	0	0	0.00	30	1408
4	0	0	45	0	1	58.53	95	1560

5 rows × 40 columns

8.2 Scaling the data:

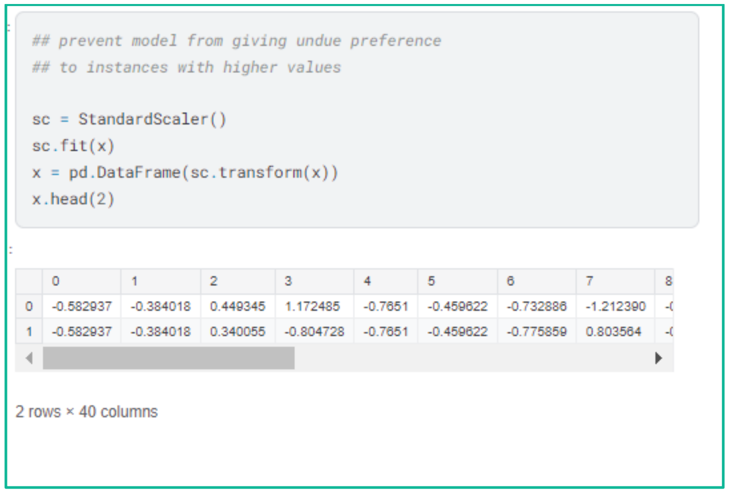
Scaling is a preprocessing step in machine learning that adjusts the range of the data features to ensure that they have a comparable scale. This is important because many machine learning algorithms perform better when numerical features are on a similar scale. If features are not scaled, those with larger ranges can dominate the learning process, leading to suboptimal models.

Types of Scaling:

- **Min-Max Scaling (Normalization)**:
  - Scales the data to a fixed range, typically 0 to 1.
  - Best for algorithms that do not assume any distribution of the data (e.g., k-nearest neighbors, neural networks).
- **Standard Scaling (Z-score Normalization)**:
  - Scales the data so that it has a mean of 0 and a standard deviation of 1.

    - Best for algorithms that assume normally distributed data (e.g., linear regression, logistic regression).
- **Robust Scaling**:
    - Scales the data using statistics that are robust to outliers (median and interquartile range).
    - Useful when data contains outliers.
- **MaxAbs Scaling**:
    - Scales the data to the range [-1, 1] by dividing by the maximum absolute value.
    - Useful for sparse data were preserving zero entries is important.

I have followed standard scaling (Z-score normaliztion) to replace some attributes like killPlace, killPoints, and more have shown some high values compared to the rest of the data.



```
## prevent model from giving undue preference
## to instances with higher values

sc = StandardScaler()
sc.fit(x)
x = pd.DataFrame(sc.transform(x))
x.head(2)
```

	0	1	2	3	4	5	6	7	8
0	-0.582937	-0.384018	0.449345	1.172485	-0.7651	-0.459622	-0.732886	-1.212390	-
1	-0.582937	-0.384018	0.340055	-0.804728	-0.7651	-0.459622	-0.775859	0.803564	-

2 rows × 40 columns

I have applied the formulae,

$(X_i - X_{mean})/X_{std}$

8.3 Splitting the data

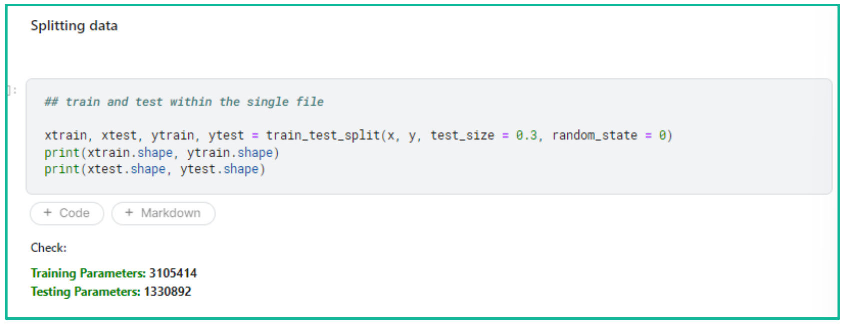
Splitting the data is a crucial step in preparing a dataset for machine learning modeling. It involves dividing the data into distinct subsets to ensure that the model can be trained and evaluated properly. The most common splits are the training set and the test set, and sometimes a validation set is used as well.

How should we split the data?

Training Set: This subset of data is used to train the machine learning model. The model learns the relationships between the input features and the target variable from this data.

Test Set: This subset is used to evaluate the performance of the trained model. It provides an unbiased assessment of how well the model generalizes to new, unseen data.

We have a single dataset to both train and test. Hence, I had to split it into two with X_{input_output} and $Y_{input\text{-}output}$ parameters (Input = Train, Output = Test).



Splitting data

```
## train and test within the single file

xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.3, random_state = 0)
print(xtrain.shape, ytrain.shape)
print(xtest.shape, ytest.shape)
```

Check:

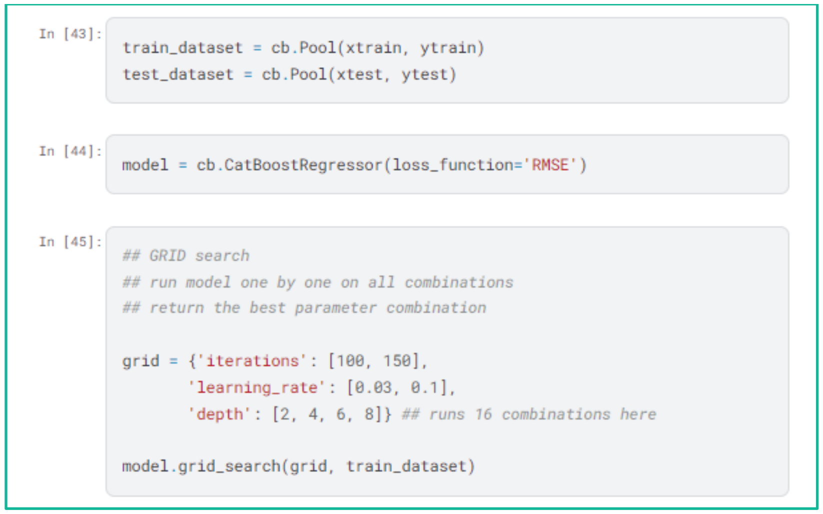
Training Parameters: 3105414
Testing Parameters: 1330892

We have successfully separated it into two parts:

- Training Parameters: 3105414
- Testing Parameters: 1330892

8.4 CatBoost Model

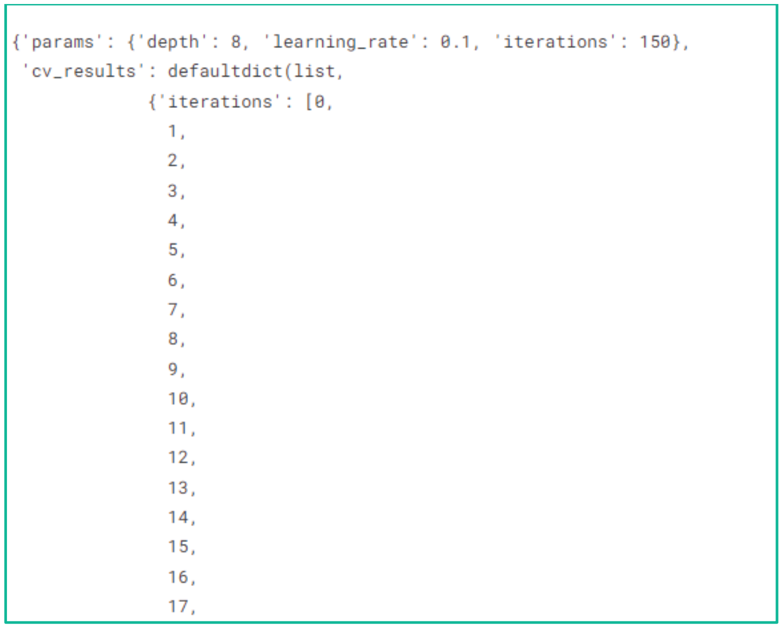
CatBoost (Categorical Boosting) is a powerful gradient boosting algorithm developed by Yandex. It is designed to handle categorical features efficiently, making it particularly useful for datasets with a mix of numerical and categorical data. CatBoost builds an ensemble of decision trees in a sequential manner, where each tree aims to correct the errors of the previous ones.



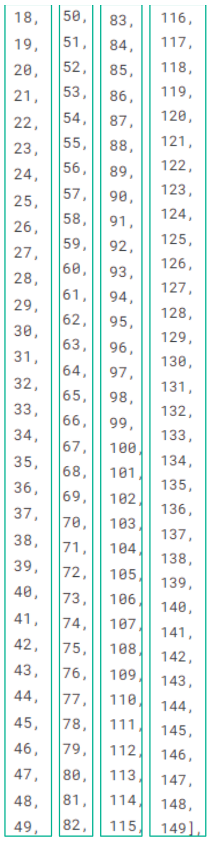
```
In [43]: train_dataset = cb.Pool(xtrain, ytrain)
         test_dataset = cb.Pool(xtest, ytest)

In [44]: model = cb.CatBoostRegressor(loss_function='RMSE')

In [45]: ## GRID search
         ## run model one by one on all combinations
         ## return the best parameter combination

         grid = {'iterations': [100, 150],
                 'learning_rate': [0.03, 0.1],
                 'depth': [2, 4, 6, 8]} ## runs 16 combinations here

         model.grid_search(grid, train_dataset)
```

Implementation Details:

- **Data Preparation:** Training and testing datasets were prepared using `cb.Pool`.
- **Model Initialization:** A CatBoostRegressor was initialized with loss_function='RMSE'.
- **Grid Search:** A grid search was conducted over 16 combinations of hyperparameters (iterations, learning_rate, depth) to find the best parameters.



```
{'params': {'depth': 8, 'learning_rate': 0.1, 'iterations': 150},
 'cv_results': defaultdict(list,
             {'iterations': [0,
               1,
               2,
               3,
               4,
               5,
               6,
               7,
               8,
               9,
               10,
               11,
               12,
               13,
               14,
               15,
               16,
               17,
```



```
18,
19,
20,
21,
22,
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
```

```
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
```

```
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99,
100,
101,
102,
103,
104,
105,
106,
107,
108,
109,
110,
111,
112,
113,
114,
115,
```

```
116,
117,
118,
119,
120,
121,
122,
123,
124,
125,
126,
127,
128,
129,
130,
131,
132,
133,
134,
135,
136,
137,
138,
139,
140,
141,
142,
143,
144,
145,
146,
147,
148,
149],
```

Best Parameters:

- 'depth': 8
- 'learning_rate': 0.1
- 'iterations': 150}
- 'iterations': [0, ... 0.00010630253266510458]

'test-RSME-mean': [0.510447139364591, 0.46282374239418944 .... 0.08233400605849632, 0.08230362954383401]

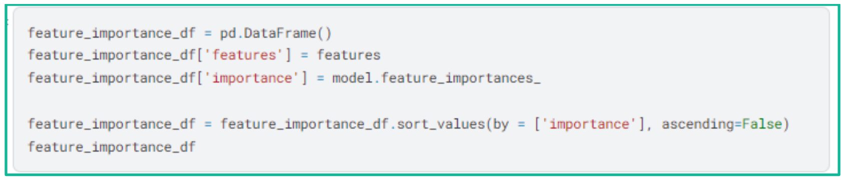
'test-RSME-std': [0.00010276533926709174, 5.6173786800379114e-05 .... 8.785279530946299e-05, 9.618468370615735e-05]

'train-RSME-mean': [0.5104470507382596, 0.4628209532856265 .... 0.08216624885893413, 0.0821341659974452]

'train-RSME-std': [5.377279669221499e-05, 0.00017673718230797576 .... 0.00011721379872657394, 0.00010630253266510458]

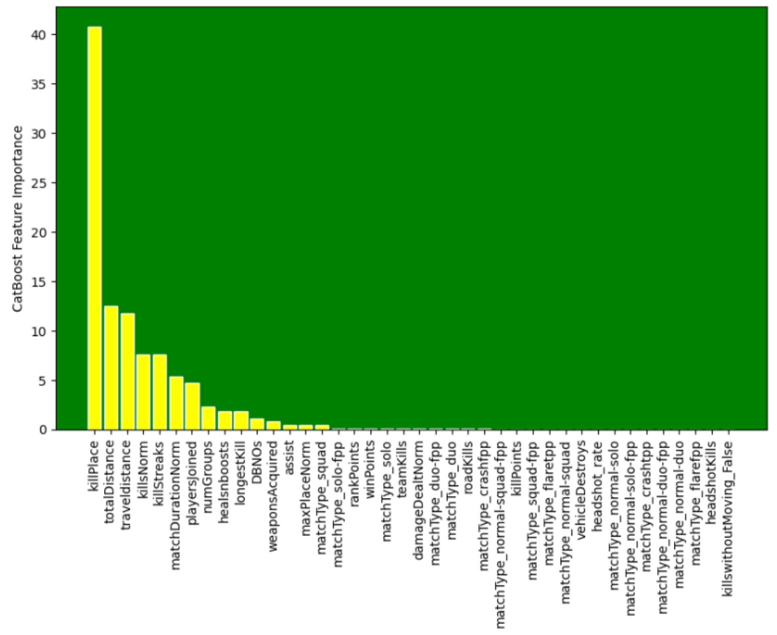
Feature Importance:

After training our data with CatBoost model, feature importance helps to understand the contribution of each feature to the model's predictions.



```
feature_importance_df = pd.DataFrame()
feature_importance_df['features'] = features
feature_importance_df['importance'] = model.feature_importances_

feature_importance_df = feature_importance_df.sort_values(by = ['importance'], ascending=False)
feature_importance_df
```

Visualizing the results:



Hence, the model can be trained dropping the following parameters (as their contribution is zero):

- matchType_normal-squad
- vehicleDestroys
- headshot_rate
- matchType_normal-solo
- matchType_normal-solo-fpp
- matchType_crashtpp
- matchType_normal-duo-fpp
- matchType_normal-duo
- matchType_flarefpp
- headshotKills
- killswithoutMoving_False

8.5 Model Prediction and Evaluation:

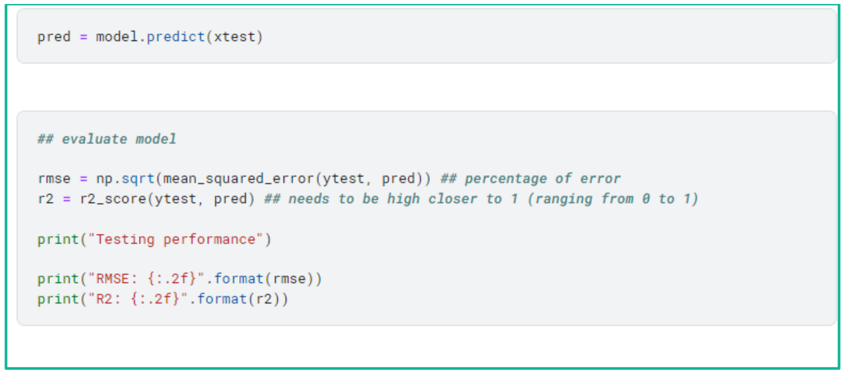
Model Prediction and Evaluation are crucial steps in understanding how well your machine learning model performs on unseen data. In the PUBG game ranking prediction project, after training the CatBoost model, the performance was evaluated using the Root Mean Squared Error (RMSE) and the R² score.

Implementation Details:

- **Making Prdictions:** The trained CatBoost model is used to make predictions on the test dataset.
- **Evaluating the Model:** The model's performance is evaluated using RMSE and R² score.

Root Mean Squared Error (RMSE): RMSE is calculated to measure the average magnitude of the prediction errors. A lower RMSE indicates better model performance.

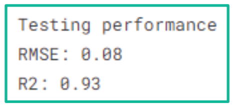
R² Score: The R² score is computed to determine the proportion of variance in the dependent variable that is predictable from the independent variables. An R² score close to 1 indicates a high level of accuracy.



```
pred = model.predict(xtest)
```

```
## evaluate model

rmse = np.sqrt(mean_squared_error(ytest, pred)) ## percentage of error
r2 = r2_score(ytest, pred) ## needs to be high closer to 1 (ranging from 0 to 1)

print("Testing performance")

print("RMSE: {:.2f}".format(rmse))
print("R2: {:.2f}".format(r2))
```

Results:



```
Testing performance
RMSE: 0.08
R2: 0.93
```

- **RSME:** The model achieved a Root Mean Squared Error (RSME) of 0.08.
- **R² Score:** The R² score was close to 1, indicating excellent predictive performance and high accuracy in ranking predictions without being overfitting.

Conclusion

This report is a successful baseline for the analysis and prediction of player performance during PUBG matches. It begins by investigating the key characteristics and patterns in the data, before identifying anomalies and thus possible cheaters and AFK players. We can then carry a feature selection stage to then predict where each player ranks at the end of the game to three degrees of specificity. Several different machine learning algorithms are evaluated and reasonable accuracy is achieved in all three cases. Finally, the results are discussed along with possible improvements are future work.