Winning strategies in team play battle royale game: Insight from Playerunknown's Battleground Game Data

Ruixuan Li, Zeping Tao

December 13, 2019

1 Introduction

Battle royale-style video games have taken the world by storm. It is the type of the game that around 100 players are dropped onto an island empty-handed and must explore, scavenge, and eliminate other players until only one is left standing, all while the play zone continues to shrink. In the team play mode of these games, players form teams and play corporately against the other teams.

PlayerUnknown's BattleGrounds (PUBG) has enjoyed massive popularity as a Battle Royale-style video game. With over 50 million copies sold, it's the fifth best selling game of all time, and has millions of active monthly players.

Taking 'squad' mode PUBG as an example, we applied k-means clustering and decision tree methods to study how players in each match behave on attacking, supporting, cooperation and moving, the essential components of team play battle royale games as well as how these performances lead them to win or lose.

2 Data

2.1 Data preprocessing

The team at PUBG has made official game data available for the public. *Kaggle*, the online community of data scientists and machine learners has collected and documented these data through the *PUBG Developer API* and made it public for a data competition. In our analysis, we use the training dataset (they also provide a testing

dataset) downloaded from *Kaggle* competition website. For analysis, we preprocess the downloaded data in the following ways and our analysis data has 98,351 observations in the end.

2.1.1 Random sample 2,000 matches for analysis

In *Kaggle* training dataset, there are 444,6966 observations for analysis. Each observation is a player in a specific match defined by a 'match ID'. For computation efficiency, we took a random sample of 2,000 unique matches from the data set, which gave us 186,444 observations.

2.1.2 Focus on 'squad' game mode

The game has 3 kinds of basic mode: 'solo', 'duo and 'squad'. In 'solo' mode players play as individuals, in 'duo' mode players form two-member teams to play while in 'squad' mode players form four-member teams to play. The game also have 'flare' and 'crash' as special event modes.

	Count	Percentage
solo	28242	15.10%
duo	57414	30.80%
squad	100273	53.80%
event	515	0.30%
total	186444	100.00%

Table 1: Frequency of game modes in the 186,444 sample

Since we focus on team play part of this game, 'squad' mode would be the best population for our analysis. Therefore, we only kept the 100,273 'squad' mode observations in our analysis sample.

2.1.3 Eliminate observations with less player joined in each match

PUBG game is designed to have 100 players joined in each match, but it does not have exactly 100 players joined in a match every time. In our data, there are 4480 observations in a match with less than 85 players joined, consisting 2.4% of all the observations. Since the gaming pattern might be different and the quantity is so small, we excluded these observations from our analysis so that our analysis data set has 98,351 observations left.

2.1.4 Calculate headshot kill rate

'Headshot kill' is a kind of killing behaviour that the player shoot another player dead at head. Once getting shot at head, the player dies immediately and has no chance to revive. It reflects the accuracy of shooting of a player. Therefore, we created the variable "headshot kill rate" by dividing the number of headshot kills by the total number of kills.

2.2 Variables

2.2.1 Attacking

The ability to attack enemy players is crucial to the success of PUBG game. Players who can kill enemies quickly, accurately and quietly would have higher chance to win. There are 5 measurement variables that could define the latent attacking characteristics of a player:

- damageDelt: Total damage a player dealt on other players excluding self inflicted damage.
- DBNOs: Number of enemy players knocked, which is a state that are about to die but can be revived by teammates in a short time.
- kills: Number of enemy players killed.
- killStreaks: Max number of enemy players killed in a short amount of time.
- longestKill: Longest distance between player and player killed at time of death.
- headshotRate: Rate of headshot kills, which reflect the accuracy of shotting. (Number of enemy players killed with headshots divided by total number of enenymy players killed.

Since the values of these variable are highly right-skewed, we transformed them by taking their natural logarithm.

2.2.2 self-supporting

In order to survive to last in PUBG game, players need to acquire multiple items by exploring different places and use them in order to support the players in various occasions. Items for boosting, healing and multiple weapons are useful for recovering from damages and killing in tough and varied circumstances. Here are 3 measurement variables that could define the latent self-supporting characteristics of a player:

- boost: Number of boost items used. Boost item will increase self-healing ability so that player's health will recover itself in a period of time.
- heals: Number of healing items used. Healing item will immediately increase player's health.
- weaponAcquired: Number of weapons picked up.

Since the values of these variable are highly right-skewed, we transformed them by taking their natural logarithm.

2.2.3 Cooperation

In the 4-member 'squad' mode, players need to play corporately for efficient killing and team surviving by reviving and assisting teammates, and of course not to kill teammates. Here are 3 measurement variables that could define the latent cooperation characteristics of a player:

- assists: Number of enemy players this player damaged that were killed by teammates.
- revives: Number of times this player revived teammates.
- teamKills: Number of times this player killed a teammate.

Since the values of these variable are highly right-skewed, we transform them by taking their natural logarithm.

2.2.4 Moving

In the game, the playable area of the map shrink down towards a random location every several minutes, with any player caught outside the safe area taking damage incrementally, and eventually being eliminated if the safe zone is not entered in time. Therefore, moving by walking, running, swimming and riding with vehicles would be a common behavior that players do for survival. Here are 3 measurement variables that could define the latent moving characteristics of a player:

- rideDistance: Total distance traveled in vehicles measured in meters.
- swimDistance: Total distance traveled by swimming measured in meters.
- walkDistance: Total distance traveled on foot measured in meters.

Since these variables are measured in meters, their range are hugely different from each other. Therefore, we rescaled them using the min-max scaling method.

2.2.5 Win or lose

Whoever survive at last is the biggest winner of this game. Teams that are completely annihilated later have a higher ranking in the end than other teams that are completely annihilated earlier in a match. The measurement for winning is a continuous percentile winning placement, where 1 corresponds to 1st place, and 0 corresponds to last place in the match.

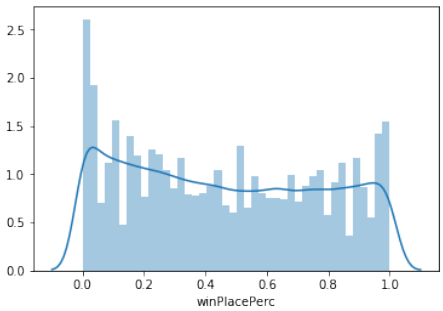

Figure 1: Distribution of winning measurement

3 Model

Due to the large number of variables and certain latent variables that we want to measure, dimension reduction would be necessary for a clearer summary of patterns of player behaviors. After we have the clustered performance patterns, we can use them to predict game outcome, which is to win or to lose.

3.1 Dimension reduction: k-means clustering

Here are the requirements for dimension reduction method in this analysis:

- Clustering: Our input variables are mostly continuous and might not be consistent in predicting winning, so we need categorical outcome without direction.
- Computational efficiency: The method should be able to deal with our large data set of 98,351 observations with an acceptable efficiency.
- Assumption free: Since the independence or association between our measurement variables are unclear and varied, the method should tolerate both independent and correlated variables.
- Interpretability: The clustering result should reflect instinct patterns and be easy to interpret.

Considering these requirements, k-means clustering method would be the best for this analysis. K-means clustering uses an iterative algorithm to partition n observations into k pre-defined distinct non-overlapping clusters. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The objective function (distortion) is:

$$J = \sum_{k=1}^{K} \sum_{i=1}^{n_k} \|X_i^{(k)} - \mu_k\|^2 \tag{1}$$

where K is the number of clusters, n_k is the number of cases in the cluster k, x_i is the case i in cluster j, μ_k is the centroid of cluster k.

We applied k-means method to each set of measurement variables: attacking, self-supporting, cooperation and moving. We choose the number of clusters based on its distortion reduced and the interpretability of clusters. K-means method is sensitive to unbalanced data, so we rescaled or tranformed certain variables according to the balance of the data. We describe the clusters based on the means of each measurement variable for each cluster.

3.2 Prediction: regression tree

Here are the requirements for prediction method in this analysis:

- Accounting for interaction: Gaming patterns interact to each other. For instance, an all-round attacker might have higher probability of being an better self-supporter since he/she needs to survive longer in the combats in order to kill more enemies.
- Continuous outcome: Our winning measurement is a continuous variable.
- Computational efficiency: The method should be able to deal with our large data set of 98,351 observations with an acceptable efficiency.
- Multiple predictors: Clusters are ordinal, so they would be treated as dummy variables which result in a large number of predictors.
- Interpretability: The outcome should tells us how and how much different gaming patterns can lead to win or lose.

Considering these requirements, regression tree method would be the best for this analysis. Regression encompasses a nonparametric regression technique that 'grows' a decision tree based on a binary partitioning algorithm, which recursively splits the data until groups either were homogeneous or contained fewer observations than a user-defined

threshold (Aertsen et al. 2010). The predicted value of a 'terminal' node is the average of the response values in that node (Breiman et al. 1984).

We use Gini index to measure the homogeneity of the target variable within the splitted subsets. Gini index measures how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. If there are k classes of the target attribute, with the probability of the ith class being p_i, the Gini Index is defined as (Bramer 2007 Bramer 2007):

$$Gini\ Index = 1 - \sum_{i=1}^{k} p_i^2 \tag{2}$$

The splitting attribute is the attribute with the largest reduction in the value of the Gini Index.(Shouman, Turner, and Stocker 2011)

We tuned max tree depth and minimum node size to avoid overfitting and for interpretability.

4 Result

4.1 Player Behavior Patterns

4.1.1 Attacking

Based on the identified variables related to attacking, we tried to find players' attacking behavior patterns and player types using K-means clustering method. Clustering players into 3 classes is reasonable for distortion reduction.

The following figure 2 shows the mean value of each variables across 3 classes. Note that the values here are normalized in order to bring in variables with different scales. We named each class based on its characteristics on attacking variables. 'All-round attackers' are highly skilled with enormous damage dealt, more kills than other players, and able to kill from very long distance. 'Weak attackers' are rookies or players who they perform bad on almost all attacking variables. 'Damage dealers' perform bad on killing, but they actually deal damage to enemy players. These are the players who are getting experienced with this game but missing the critical strike to secure a kill, which explains their relative high knock down numbers (DBNOs) but low kills.

4.1.2 Self-supporting

After applying k-means clustering method to self-supporting variables, we chose to cluster players into 4 classes which is the optimal choice for distortion reduction.

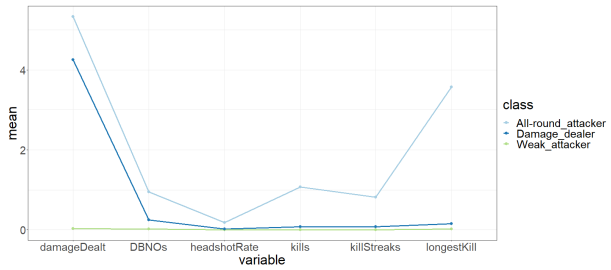

Figure 2: Attacking variable means

Figure 3 shows different features among 4 classes players. Compared to other kinds of players, 'all-round self-supporter' use the largest number of boosting and healing items as well as collect most weapons. 'Moderate self-supporters' act similarly with 'All-round self-supporter', except that they use less boosting items. 'Weak self-supporter' use or collect the lowest number of all self-supporting items. 'Weapon collector' players are similar with 'Weak self-supporter', except that 'Weapon collector' are weapon fanatics with lots of weapon acquired.

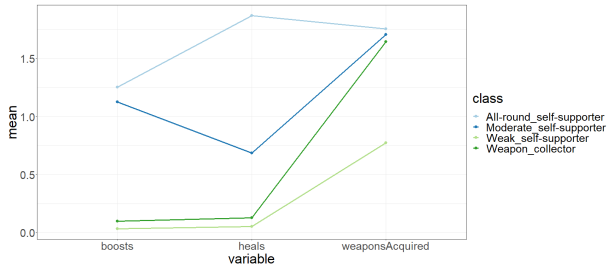

Figure 3: Self-supporting variable means

4.1.3 Cooperation

After applying k-means clustering method to cooperation variables, we chose to cluster players into 4 classes which is the optimal choice for distortion reduction and interpretability.

The following graph 4 shows the differences between players on cooperation activities. 'Weak assistants' are throwers who seldom interact with teammates. 'All-round assistants' are those of high cooperation activities on both assisting teammates to knock down enemies and taking risk reviving teammates. 'Attacking assistants' assist teammates on killing enemies but not on reviving dying teammates. 'Supporting assistants' are more likely to revive teammates than engage in the battle.

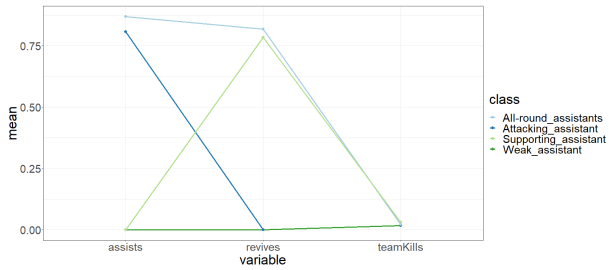

Figure 4: Cooperation variable means

4.1.4 Moving

We applied k-means clustering method to moving variables. For distortion reduction and interpretability purpose, 4 classes of players were identified.

The characteristics of 4 classes players are shown in the figure 5. 'Drivers' like using vehicles to move while 'walkers' move on their legs most of the time. 'Swimmers' spend a lot of time in water and move through water rather than land. 'Weak movers' players move limited distance during the game.

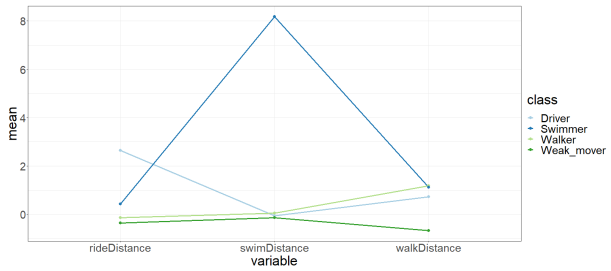

Figure 5: Movement variable means

4.2 Regression Tree on Final Placement

We fitted a regression tree on all players to predict their final win placement percentage and explore what these player clusters are able to suggest on performing better in the game. As mentioned before, higher win placement percentage means higher placement and 1 means the final winner of the game. To fulfill the interpretation purpose, the depth of the tree is limited to 5, the minimum node size is set as 2000. The regression tree is shown in figure 6.

The color saturation indicates the final placement percentage. Deeper the color is, higher the mean final placement percentage the cell has. Here are 4 of the most significant conclusion from the result.

The first thing we find is that being a 'swimmer' is harmful to the final placement, as the left tree of the 'swimmer'

nodes generally have higher win placement than the right tree. Swimming is a very dangerous choice to make in the game, since players cannot shot while swimming, the moving speed in water is very limited and there is nowhere to hide if someone shots you.

Second, we find that being a 'weak attacker' has higher prediction of win placement than others (0.81 versus 0.7). This might be counter-intuitive at first, but actually, it indicates that players with high desire to attack and also put themselves into a risky situation of battle. Considering large amount of players in one match, being caught into a fight might attract potential enemies and make the combat more dangerous and complicated.

Following the nodes with high win placement prediction, we find that if the player is a 'weak attacker', being a 'weak assistant' does not help increase final placement (prediction 0.76 versus 0.84). This suggests that if the player is not good at combat, they should at least help teammates. Since the 'squad' mode emphasizes team cooperation, it is crucial to help teammates especially when players are not active attackers.

For self-supporting items, 'all-round self-supporter' has clear advantages over other supporting type players. The survival purpose of the game makes collecting and utilizing resource critical, and 'all-round self-supporter' type of players utilizes resources better than other players, which increases their survival possibilities and gives them backup to fight for high placement.

5 Conclusion

The result of our analysis partly explains the popularity of PUBG and Battle Royale-style video games. It requires players to carefully choose and balance their strategies on multiple aspects in a time-limited, rapidly-variant and partly random match. Unlike traditional online team combat mode shooting game that largely rely on shooting ability, excellence on shooting would be the far less important in PUBG than a wise strategy on moving, collecting and cooperation, making the game to be diverse and challenging that worth exploring and attempting.

References

Aertsen, Wim et al. (2010). "Comparison and ranking of different modelling techniques for prediction of site index in Mediterranean mountain forests". In: *Ecological modelling* 221.8, pp. 1119–1130.

Bramer, Max (2007). *Principles of data mining*. Vol. 180. Springer.

Breiman, Leo et al. (1984). "Classification and regression trees. Wadsworth Int". In: *Group* 37.15, pp. 237–251.

Shouman, Mai, Tim Turner, and Rob Stocker (2011). "Using decision tree for diagnosing heart disease patients". In: *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*. Australian Computer Society, Inc., pp. 23–30.

A Regression Tree on All Players

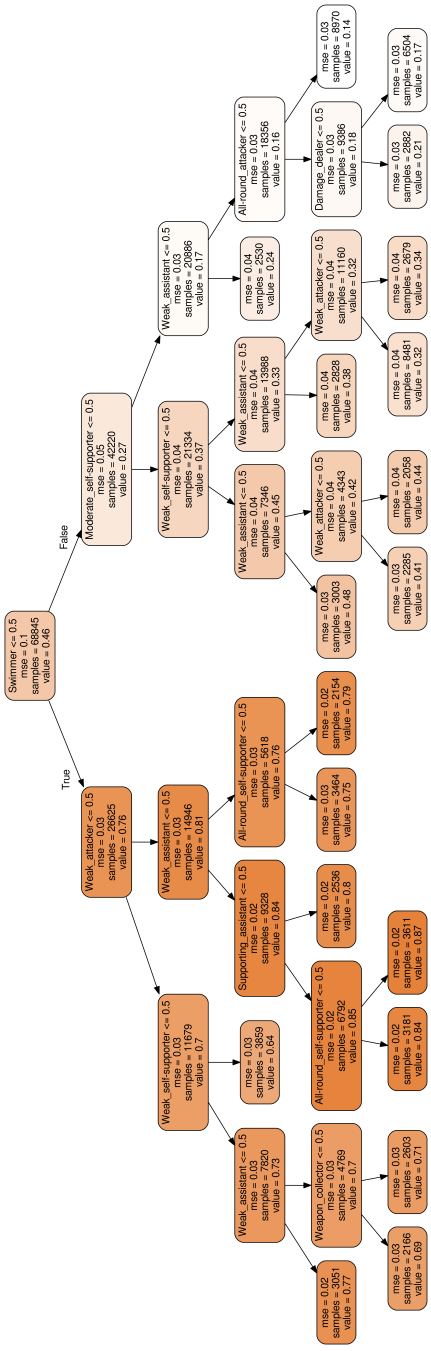

Figure 6: Regression Tree

B Appendix Data Processing and Analysis Python code

```
import numpy as np
import pandas as pd
import random as random

# import k means package
from sklearn.cluster import KMeans
from time import time
import seaborn as sns
import matplotlib.pyplot as plt

# Import xgboost classifier
from xgboost import XGBClassifier
from xgboost import plot_tree
import xgboost as xgb
from xgboost import plot_importance

import os

# Normalize the data attributes.
from sklearn import preprocessing
from sklearn.preprocessing import FunctionTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from sklearn.tree import DecisionTreeRegressor

# visualize decision tree
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
```

```
import graphviz


# Read in dataset
dt = pd.read_csv('/kaggle/input/pubg-finish-placement-prediction/train_V2.csv')

# Random sample data by matchId
random.seed(3)
match_id_list = random.sample(dt["matchId"].unique().tolist(),2000)
dts = dt[dt['matchId'].isin(match_id_list)]
dts.info()

# Collapse fpp and tpp match type. Only classify match type by single, dual, squad and event.
# Only take match type of our intrest, that is squad
def match_3(row):
    if 'solo' in row['matchType']:
        return 'solo'
    elif 'duo' in row['matchType']:
        return 'duo'
    elif 'squad' in row['matchType']:
        return 'squad'
    else:
        return 'event'
dts['gameMode'] = dts.apply(lambda row: match_3(row), axis=1)
dts = dts.loc[dts['gameMode'] == 'squad']
dts.shape

# fpp variable
# Should we include fpp variable even if it not used as any measurement variable in SEM?
def fpp_gen(row):
    if 'fpp' in row['matchType']:
        return 'yes'
    elif 'fpp' not in row['matchType']:
        return 'no'
```

```
dts['fpp'] = dts.apply(lambda row: fpp_gen(row), axis=1)

# Count number of players joined in one match
#players joined
dts['playersJoined']=dts.groupby('matchId')['matchId'].transform('count')
dts[['playersJoined', 'matchId']].head()
dts=dts[dts['playersJoined']>=85]
len(dts)

# Merge winpoint and rankpoint
#    winPoints      and      rankPoints
def merge_ranking(row):
    return max(row['rankPoints'], row['winPoints'])

dts['newRank'] = dts.apply(lambda row: max(row['rankPoints'], row['winPoints']), axis=1)
dts['newRank'].isna().sum()
dts=dts.drop(columns=['rankPoints', 'winPoints'])

# headshot rate
def replace_nan(row):
    if np.isnan(row['headshotRate']):
        return 0
    else:
        return row['headshotRate']

dts["headshotRate"]=dts["headshotKills"]/dts["kills"]
dts['headshotRate'] = dts.apply(lambda row: replace_nan(row), axis=1)

# Select variable of interest.

attack_var = ['damageDealt', 'DBNOs', 'kills',
              'killStreaks', 'longestKill', 'headshotRate']

support_var = ['boosts', 'heals', 'weaponsAcquired']
```

```
coop_var = ['assists', 'revives', 'teamKills']

mov_var = ['rideDistance', 'swimDistance', 'walkDistance']

def optimal_cluster_num(latent, normalized_X):

    # Function show elbow plot for different cluster number options.
    # (default 1 - 10 clusters)

    distortions = []
    for i in range(1, 11):
        t0 = time()
        km = KMeans(
            n_clusters = i, init = 'random',
            n_init=10, max_iter=300,
            tol=1e-04, random_state=0)
        km.fit(normalized_X)
        distortions.append(km.inertia_)
        print('number of cluster = ' + str(i))
        print('Time: ' + str(time() - t0))

    plt.plot(range(1, 11), distortions, marker='o')
    plt.xlabel('Number of clusters')
    plt.ylabel('Distortion')
    plt.show()

    return 0

def kmeans_output(num_class, normalized_X, var):

    # Create mean for each cluster, each variable

    km = KMeans(
```

```
        n_clusters = num_class, init = 'random',
        n_init=10, max_iter=300,
    tol=1e-04, random_state=0)

    normalized_X['class'] = km.fit_predict(normalized_X)



    prediction = pd.DataFrame(normalized_X)
    prediction.columns = var + ['class']

    features = pd.DataFrame([])
    for i in range(0, num_class):
        temp = pd.DataFrame(prediction[prediction['class'] == i].describe())
        temp = pd.DataFrame(temp.loc['mean', :])
        temp.columns = ['class' + str(i)]
        features = pd.concat([features, temp], axis=1)

    return features

def get_prediction(normalized_X, num_class):

    # Get prediction classes

    km = KMeans(
        n_clusters = num_class, init = 'random',
        n_init=10, max_iter=300,
        tol=1e-04, random_state=0)
    return km.fit_predict(normalized_X)



# Normalize attack ability variables
selected = attack_var

X = dts[selected]
```

```
X.reset_index(drop=True)

transformer = FunctionTransformer(np.log1p, validate=True)

normalized_X = transformer.transform(X)

normalized_X = pd.DataFrame(normalized_X)
normalized_X.columns = selected

# Show class number plot
optimal_cluster_num(selected, normalized_X)

# create variable mean for each cluster
features = kmeans_output(3, normalized_X, selected)

attack_class = get_prediction(normalized_X, 3)
features

# Normalize support ability variables
selected = support_var

X = dts[selected]

X.reset_index(drop=True)

transformer = FunctionTransformer(np.log1p, validate=True)

normalized_X = transformer.transform(X)

normalized_X = pd.DataFrame(normalized_X)
normalized_X.columns = selected

# Show class number plot
```

```
optimal_cluster_num(selected, normalized_X)

# create variable mean for each cluster
features = kmeans_output(4, normalized_X, selected)

support_class = get_prediction(normalized_X, 4)

features

# Normalize cooperation ability variables
selected = coop_var

X = dts[selected]

X.reset_index(drop=True)

transformer = FunctionTransformer(np.log1p, validate=True)

normalized_X = transformer.transform(X)

normalized_X = pd.DataFrame(normalized_X)
normalized_X.columns = selected

# Show class number plot
optimal_cluster_num(selected, normalized_X)

# create variable mean for each cluster
features = kmeans_output(4, normalized_X, selected)

coop_class = get_prediction(normalized_X, 4)

features

# Normalize mobility variables
```

```
selected = mov_var

X = dts[selected]

X.reset_index(drop=True)

normalized_X = preprocessing.scale(X)

normalized_X = pd.DataFrame(normalized_X)
normalized_X.columns = selected

# Show class number plot
optimal_cluster_num(selected, normalized_X)

# create variable mean for each cluster
features = kmeans_output(4, normalized_X, selected)

mov_class = get_prediction(normalized_X, 4)

features

# Subset original data and get training data set.

temp = pd.DataFrame(dts['winPlacePerc']).reset_index(drop=True)
attack_class_dt = pd.DataFrame({'attack_class': attack_class})
support_class_dt = pd.DataFrame({'support_class': support_class})
coop_class_dt = pd.DataFrame({'coop_class': coop_class})
mov_class_dt = pd.DataFrame({'mov_class': mov_class})

X_dt = pd.concat([temp, attack_class_dt, support_class_dt, coop_class_dt, mov_class_dt], axis=

def binary_win(row):

    # Label top 10% players
```

```
    if (row['winPlacePerc'] > 0.9):
        return 1
    else:
        return 0

X_dt['win'] = X_dt.apply(lambda row: binary_win(row), axis=1)
X_dt.head()

# One hot encoder latent classes

enc = OneHotEncoder(handle_unknown='ignore', sparse = False)

feature_col = ['attack_class', 'support_class', 'coop_class', 'mov_class']

X_init = X_dt[feature_col]

enc.fit(X_init)

X = pd.DataFrame(enc.transform(X_init))
X.columns = enc.get_feature_names(feature_col)
print(X.shape)
print(X.columns)
X.head()

# Rename the classes as new names
alternative_names = ['All-round_attacker', 'Weak_attacker', 'Damage_dealer',
                     'Moderate_self-supporter', 'All-round_self-supporter', 'Weak_self-suppor
                     'Weak_assistant','Supporting_assistant', 'Attacking_assistant', 'All-rou
                     'Driver', 'Walker', 'Swimmer', 'Weak_mover']
print(alternative_names)
X.columns = alternative_names
X.head()
```

```
# Train test split
y = X_dt.win

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)

# Visualize decision tree
data = export_graphviz(clf,out_file=None,feature_names=X.columns,class_names=['lose', 'win'],
                           filled=True, rounded=False,
                           special_characters=True)
graph = graphviz.Source(data)
graph

# Create regression Tree data (whole data, all players)
y_2 = X_dt.winPlacePerc
X_train, X_test, y_train, y_test = train_test_split(X, y_2, test_size=0.3, random_state=1)

rgt_1 = DecisionTreeRegressor(max_depth=5, min_samples_leaf = 2000)
rgt_1.fit(X_train, y_train)

# Visualize regression tree
data = export_graphviz(rgt_1,out_file=None,feature_names=X.columns,
                           filled=True, rounded=False,
                           special_characters=True)
graph = graphviz.Source(data)
graph
# Export decision tree plot
export_graphviz(rgt_1, out_file='whole_tree.dot', feature_names = X_train.columns,
                rounded = True, proportion = False, precision = 2, filled = True, max_depth =
!dot -Tpng whole_tree.dot -o whole_tree.png -Gdpi=600

# Draw variable importance plot
rgt_importance = dict(zip(X_train.columns, rgt_1.feature_importances_))
pd.DataFrame([rgt_importance]).transpose().sort_values(by = 0, ascending = False)
```