LONDON'S GLOBAL UNIVERSITY

UCL

Investigation of the Use of Player Type Clustering in Predicting the Finish Placement of PlayerUnknown's Battlegrounds

Candidate Number: MPVW9[1]

BSc Computer Science

Supervised by Dr. Denise Gorse

Submission date: 03/05/2021

[1] **Disclaimer:** This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This project uses a rich data set provided by Kaggle to investigate the impact of player type clustering on PlayerUnknown’s BattleGrounds finish placement prediction. Several machine learning regressors are used for finish placement prediction, including Linear Regression and XGBoost, but the focus of the project is on the use of K-means clustering both to provide insight into the game and to investigate the hypothesis that models trained separately on different 'player types' might lead to an overall better prediction result than either a model trained on all players or models trained on 'match types' (defined within the data set and a property of the game scenario not the players). It is discovered that while player type clustering does not in fact lead to better finish placement prediction, it gives a surprising amount of insight into what makes a good player of the game.

Contents

Chapter 1

Introduction

1.1 Outline

Esports, also known as electronic sports, is a form of competition using video games.[1] The evolution of Esports had a rough but inspiring history, from two dozen students crammed into a semi-dark console room at Stanford University in October 1972 for competing in a video game called Spacewar,[2][3] to recent years in which some popular Esports competitions have been streamed worldwide to millions of people.[4] Esports has grown from an underground culture to a mainstream, multi-billion dollar industry.[5] It creates its unique ecosystem of opportunities, including live streaming, game development, the fan base of players, and brand investment in sponsorships and advertising.[5] This lucrative ecosystem has created an increasingly competitive field, and as a result there is a greater emphasis on data-based game analytics to hone and improve the skills of professional players and help them find the right way to win.

This report will focus on PlayerUnknown's Battlegrounds:

BEST MULTIPLAYER GAME
PLAYERUNKNOWN'S BATTLEGROUNDS
KRAFTON, PUBG CORPORATION, XBOX GAME STUDIOS, TENCENT GAMES

Figure 1.1: Best Multiplayer Game achieved by PlayerUnknown's Battlegrounds in The Game Awards 2017.[1]

Abbreviated also here as PUBG, this is a video game released in 2017 by Bluehole,[6] that

[1] 2017 | History | The Game Awards. 2021. *2017 | History | The Game Awards*. [online] Available at: <https://thegameawards.com/history/year-2017> [Accessed 12 April 2021].

has achieved several awards such as PC Game of the Year of 35th Golden Joystick Awards and Best Multiplayer Game of The Game Awards 2017.[7][8]

Due to the high complexity of this game, it is often difficult to predict which finish placement each player is more likely to get. Therefore, an extraordinarily rich data set collected by Kaggle has been used in this project to identify and pinpoint the winning conditions or patterns that lead the players to achieve higher finish placements. [9]

1.2 Aims and Contributions

The aim of this project was to investigate whether using player type clustering, and training separate models for each discovered 'player type', would better predict the finish placement for each player of PlayerUnknown's Battlegrounds matches.

The results from this research provide substantial insights, using K-means Clustering, Linear Regression and XGBoost models, into how a game of PlayerUnknown's Battlegrounds is best played as well as addressing the primary aim of discovering whether a game’s outcome can be better predicted by using player file clustering.

1.3 Report Structure

The rest of the report is structured as follows: Section 2 gives background information on PlayerUnknown's Battlegrounds, examines the literature on Esports prediction, and reviews the machine learning models used; Section 3 discusses design considerations for data extraction, data analysis and data transformation applied; Section 4 describes the results; and Section 5 concludes the report with discussion and thoughts about further work.

Chapter 2

Background and Related Work

2.1 PlayerUnknown's Battlegrounds

Figure 2.1: 2019 PlayerUnknown's Battlegrounds Global Championship Winner GEN.G[1]

As described in Chapter 1, the goal of the project was to investigate whether the finish placement of a player in a PlayerUnknown's Battlegrounds match could be better predicted based on player type clustering. In this first section of the project background, the history of Esports will first be briefly introduced, followed by introducing the popularity of PlayerUnknown's Battlegrounds in Section 2.1.2. Finally, the gameplay of PlayerUnknown's Battlegrounds will be explained in Section 2.1.3.

[1] ArrowDixoN. 2019. PlayerUnknown's Battlegrounds GLOBAL CHAMPIONSHIP WINNER GEN.G - FINAL GAME - Pio, Loki, Esth3r & Taemin. [Online] Available at: https://www.youtube.com/watch?v=N7DlCZlPpbU [Accessed: 13 April 2021].

2.1.1 A Brief History of Esports

As mentioned in Chapter 1, the first Esports event dates to October 1972, when Stanford University students played the video game Space Wars. The winner received a one-year subscription to Rolling Stone magazine.[1] The website "Bountie Gaming"[10] is an excellent source of information about the history of Esports and the remainer of the information in this subsection has been derived from it. Eight years after 1972, the first video game competition was held, the Space Invaders Championship, which had 10,000 contestants and received extensive media attention. As the 1980s went on, companies like Twin Galaxies promoted video games and posted high scores and records in publications like the Guinness Book of World Records. In the 1990s, companies like Nintendo and Blockbuster had started sponsoring game world championships, as the popularity of computer games grew with the advent of the Internet. In 1997, the Red Annihilation Tournament was held for the formerly famous "Quake game", which is widely considered to be the world's first Esports event, with about 2000 participants. The winner received the prize of driving John Cormack (lead developer of Quake) in his Ferrari. A few weeks later, the Cyber Athletes Professional League (CPL) was formed, and they held their first tournament later that year.[10] Later in the decade, the rise of the Defense of the Ancients (Dota), a mode derived from Warcraft III: Reign of Chaos, led to the real birth of the MOBA (Multiplayer online battle arena). For now, popular games like League of Legends continue to follow the MOBA model, with legions of fans and players on a daily basis.

2.1.2 The Popularity of PlayerUnknown's Battlegrounds

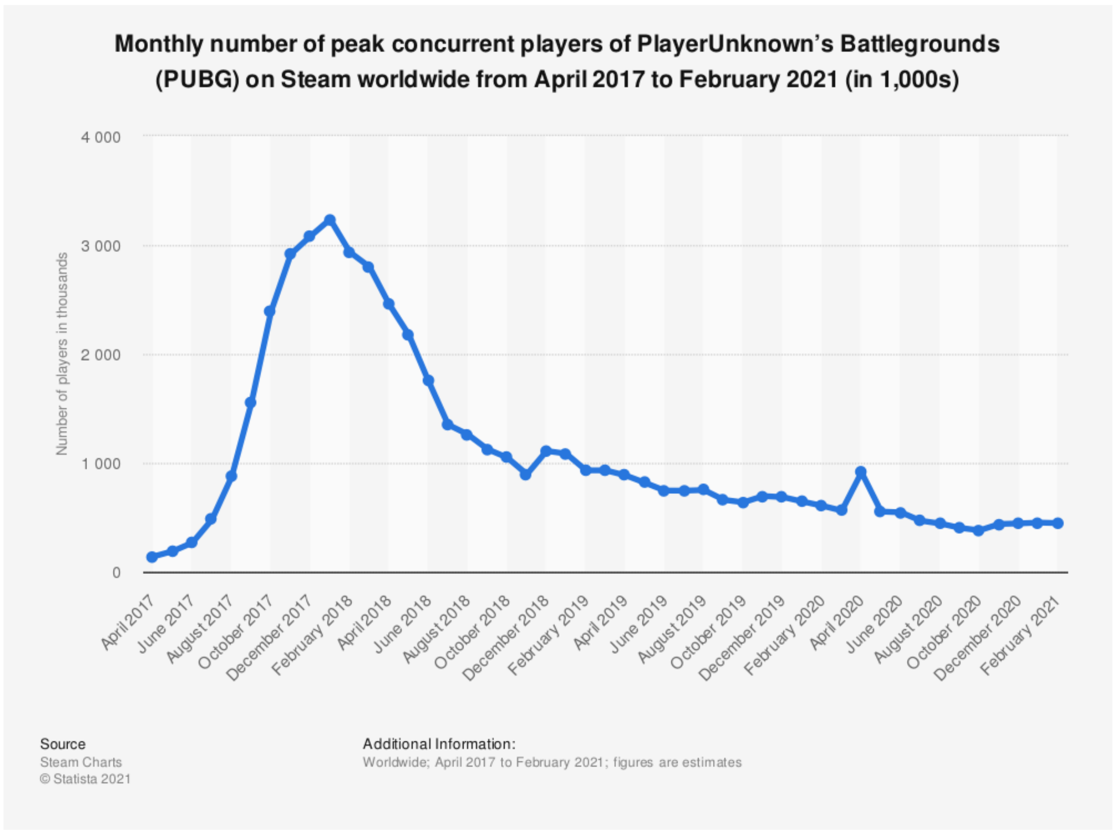


Figure 2.2: Number of Peak Concurrent Players on The Steam Platform[2]

In the first three days of Windows Early Access in March 2017, PlayerUnknown's Battlegrounds grossed US$11 million.[11] By the second week of April, the game had sold more than 1 million copies and reached a peak of 89,000 players,[12] with SuperData Research estimating that the game sold more than $34 million in April 2017, making it one of the top 10 games in the world with monthly revenue that surpassed that of Overwatch and Counter-Strike: Global Attack.[13] As of May 2017, the game had sold more than 2 million copies and grossed an estimated $60 million.[14][15] Within three months of Early Access's release, it had sold more than 5 million copies,[16] and Bluehole announced that it had grossed more than $100 million.[17] PlayerUnknown's Battlegrounds reached

[2] Statista. 2021. PlayerUnknown's Battlegrounds number of players on Steam 2021 | Statista. [online] Available at: <https://www.statista.com/statistics/755111/PlayerUnknown's Battlegrounds-number-players/#:~:text=First%20released%20at%20the%20start,January%202018%20at%203.24%20million.> [Accessed 14 April 2021].

its four million sales mark faster than Minecraft, which took more than a year during its paid beta development to reach similar sales.[18]

In January 2018, the number of peak concurrent players on the Steam platform was the highest, with 3.24 million.[19] However, due to the lack of attention to cheating and the poor optimization of the game itself, the popularity and number of players of PlayerUnknown's Battlegrounds on the Steam platform have been declining since then.

2.1.3 How PlayerUnknown's Battlegrounds Works

The websites “WIKIPEDIA”[20] and “GAMEPEDIA”[21] are fantastic sources of introducing the in-game elements of PlayerUnknown’s BattleGrounds and the information in this subsection has been derived from them.

A brief overview

PlayerUnknown's Battlegrounds is a player-versus-player shooter game where up to 100 players fight in a battle royale, a massive last-person-standing death match where players fight to keep the last survivor alive. Players can compete individually or in small teams. The last person or team alive wins the match.

Gameplay

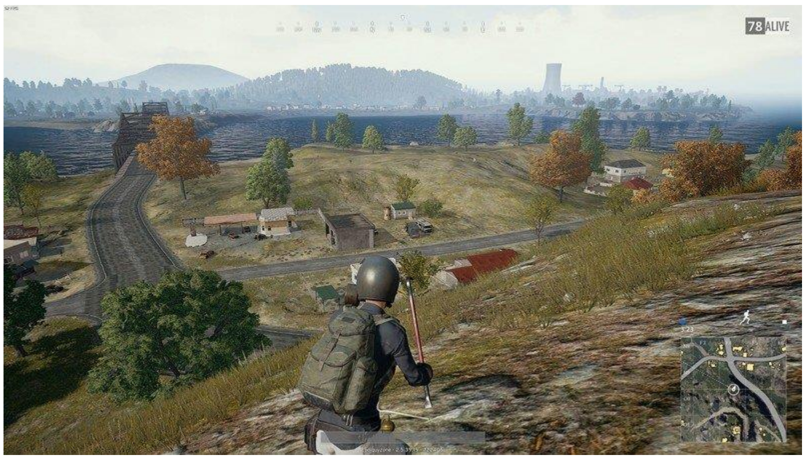


Figure 2.3: Game View of PlayerUnknown's Battlegrounds[3]

[3]Windows Central. 2017. How to make the most money on PUBG cosmetics. [online] Available at: <https://www.windowscentral.com/these-pubg-cosmetics-will-earn-you-most-money-steam-trades> [Accessed 14 April 2021].

Before each match, the system will select one of the four available maps as the match map and players will start by parachuting from a plane onto the map. The plane's flight path across the map is random, requiring the player to quickly decide the best time to eject and land. Players start out with no equipment, except for the choice of custom clothing, which does not interfere with gameplay. Players can search buildings, ghost towns, and other locations for weapons, vehicles, armour, and other equipment after landing. These items are randomly distributed on the map by the system at the start of a match, and high-risk areas usually have better equipment. Dead players can be looted to obtain their equipment. Players can choose to play the game from either a first-person or third-person perspective, each with its own strengths and weaknesses in combat and situational awareness.

Every few minutes, the safe zone on the map starts to shrink to a random location, and players who get stuck outside the safe zone gradually take damage, and if they do not reach the safe zone in time, they will eventually be eliminated. In the game, players are presented with a shimmering blue wall that shrinks over time. This leads to a more limited playable map, which increases the chance of encounters. During a match, random areas on the map will be highlighted in red, and after a certain period of time the area will be bombed, causing great danger to the players in that area. In both cases, the system warns the players a few minutes before the events, giving them enough time to move to a safe area. Occasionally, a plane will randomly fly over certain safe areas, or places where a player uses a flare gun, and drop a loot bag containing items that are not available on the map. These packs emit visible red smoke, attracting interested players to compete for them, thus creating further conflict. Generally, it takes less than 30 minutes to finish a round.

Game modes

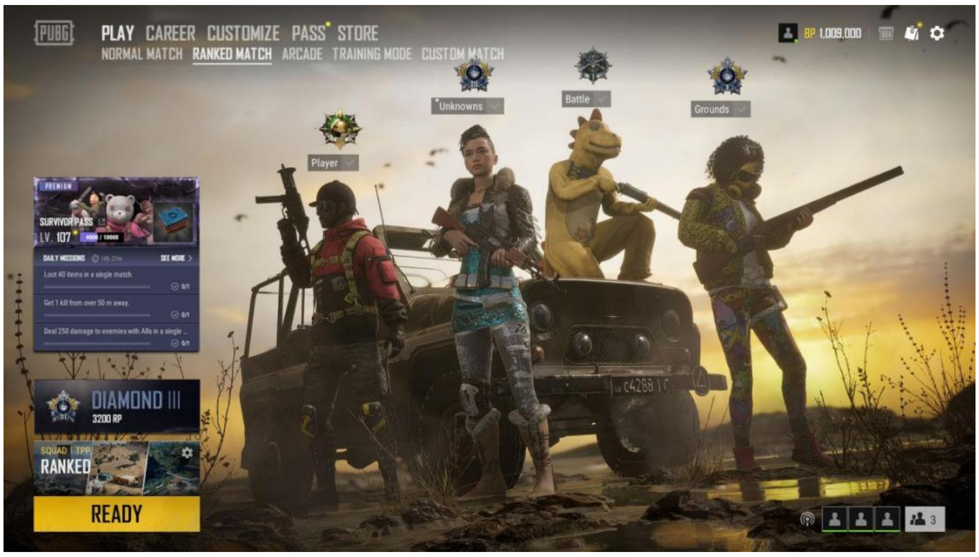

PUBG
PLAY CAREER CUSTOMIZE PASS STORE
NORMAL MATCH RANKED MATCH ARCADE TRAINING MODE CUSTOM MATCH
BP 1,009,000
Player
Unknowns
Battle
Grounds
PREMIUM
SURVIVOR PASS
LV. 107
DAILY MISSIONS
SEE MORE
Loot 40 items in a single match.
Get 1 kill from over 50 m away.
Deal 250 damage to enemies with ARs in a single ...
DIAMOND III
3200 RP
SQUAD TPP
RANKED
READY

Figure 2.4: PlayerUnknown's Battlegrounds Game Mode Selection Page[4]

Servers: Ranked vs Unranked

Ranked

Players increase their "ELO points" by winning a match and/or killing the most enemies in a match. "ELO points" are ranked in the leader board, and in most cases, professional teams recruit professional players based on "ELO points".

Unranked

Unranked servers are mainly for players who just want to have fun and study the game. Winning or losing in an unranked server does not affect the leader board in a ranked server.

Classic modes

Solo

[4] PLAYERUNKNOWN'S BATTLEGROUNDS. 2021. Dev letter: Ranked Mode. [online] Available at: <https://www.pubg.com/2020/05/20/dev-letter-ranked-mode/> [Accessed 14 April 2021].

The player is spawned alone in the game world, relying on his own tactics and techniques to survive, and aiming to be the last player alive to win the game.

Solo FPP (First Person Perspective)
Solo FPP is basically the same as *Solo* mode, except that it changes the third-person perspective to the first-person perspective.

Duo
Two players are teamed up in the game world to compete with others. All members of the last surviving team (including those killed) become the winners.

Duo FPP (First Person Perspective)
Duo FPP is basically the same as *Duo* mode, except that it changes the third-person perspective to the first-person perspective.

Squad
Four players are teamed up in the game world to compete others. All members of the last surviving team (including those killed) become the winners.

Squad FPP (First Person Perspective)
Squad FPP is basically the same as *Squad* mode, except that it changes the third-person perspective to the first-person perspective.

Event modes

Flare Gun
Squads of four will fight in Erangel, which is a game map that will be introduced shortly. Flare guns will be randomly generated on the map, but exceedingly rarely. The player who finds the gun can fire flares; a plane will drop a special package if the flares are fired within the safe zone, otherwise a special vehicle will be dropped. There will be an icon on the map that tells all players where the special care package has dropped.

Crash Carnage

Teams of duos will fight in Erangel. No firearms will be spawned throughout the game map, so the players need to focus on melee weapons, throwables, and driving skills to win this mode.

In multiplayer mode (e.g. duo, squad), when a player runs out of his life, he will not die instantly if he has a still-alive teammate, but be knocked out. The player will lose the ability to shoot, move fast and hold a gun. The player's health will decrease but his teammate will have a chance to revive him before his health runs out.

Maps

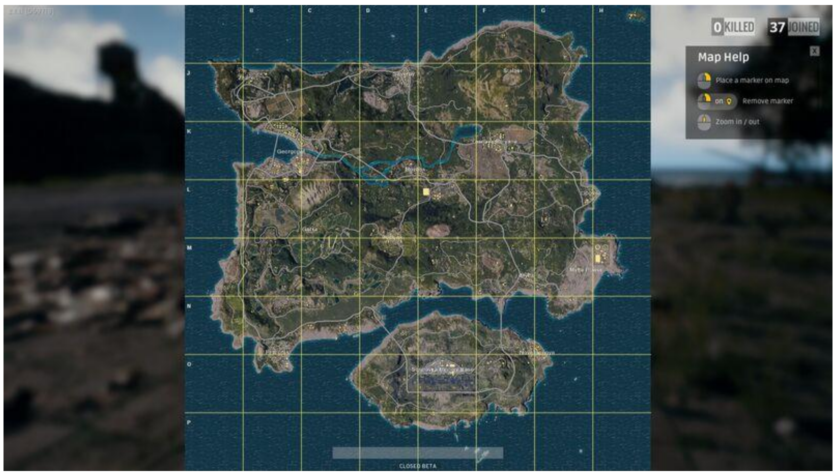


Figure 2.5: PlayerUnknown's Battlegrounds Mini-Map[5]

A map is a playable area for players to play against each other on the battlefield. In these maps, a red area periodically spawns onto random locations, causing explosions within the area, and periodically a C-130 flies over the map to drop some package.

[5] PLAYERUNKNOWN'S BATTLEGROUNDS Wiki. 2021. Maps. [online] Available at: <https://pubg.fandom.com/wiki/Maps> [Accessed 14 April 2021].

Figure 2.6: Erangel[5]

Currently Erangel (a Russian map), Miramar (a Central America sand map), Sanhok (a Southeast Asian forest map) and Vikendi (an island snow map) are used for all the game modes.

Weapons

Players can attack enemies and defend themselves by using weapons. PlayerUnknown's Battlegrounds contains more than 50 different weapons that are categorized into 11 different kinds. Each weapon can only load the corresponding type of ammunition which means the player must not only collect more ammo but also collect the correct type. For example, M16A4, one of the most popular assault rifles, can only load 5.56mm ammo.

Equipment

More than 100 items of equipment exist in PlayerUnknown's Battlegrounds, ranging from protective gear to various tools designed to help players throughout the game. They can be classified as default equipment, backpacks, glasses, gloves, goggles, hats, helmets, jacket, masks, pants, shirts, shoes, multi-slot, and vests. Backpacks, helmets, and vests are more important as they can increase the players' defense and capability to carry more stuff.

Items

Items range from munitions to consumable items which can be classified as crates, consumables, food. Consumables are incredibly significant in this game, being used for example for restoring health (medical items such as bandages), fixing a vest (vest repair kit), or refuelling a vehicle (gas can).

Vehicles

Vehicles are especially important to map movement, mainly because of the distance between players. Once the initial dense grouping of players has been handled by the system, it is important to move to the centre of the map, as the external limits will shrink. In addition, unless the player is hit directly in the head or body, the player will not take significant damage while moving the vehicle, which will greatly increase their life span. There are 16 different kinds of drivable vehicles in PlayerUnknown's Battlegrounds. Players can drive not only the on-road vehicles like vans and motorcycles, but also boats and planes. Each kind of vehicle has an independent occupant limit, speed limit and health limit (defines the ease with which the vehicle is damaged).

2.2 Use of Machine Learning in Esports Prediction

Compared to conventional sports, far more data are available in Esports. In addition the data can be precisely measured. For example, the PUBG Developer API provides several in-game statistics for each player, such as swimming distance, which can only be observed and speculated upon in a physical competition. The access to richer data in Esports opens up an opportunity to use machine learning to build powerful models that can uncover patterns that human experts would struggle to see, and leads professional Esports teams to invest millions of dollars in data companies to discover tactics that will ensure their players perform at their best.[22]

Even Kentucky Fried Chicken (KFC) have become involved with the use of ML for Esports prediction. KFC are sponsors of LPL, the top-level League of Legends (a popular MOBA game, as mentioned in Section 2.1.1) professional competition in mainland China, and are also reported to have worked with a data company to create a machine learning algorithm based on all the historical and real-time data to predict the win rate of each team during the process of a LPL competition.[23]

The remainder of this subsection will examine the ways in which different machine learning models have been used in Esports prediction, and will assess the potential value of each type of ML model for the problem of this project.

2.2.1 Recurrent Neural Network

Recurrent neural networks (RNN) are often used for time series analysis. In a recurrent neural network, the output of the previous step is used as the input of the current step, which means that it predicts a value over a set of time based on past data.[24]

A match of League of Legends usually lasts 15-30 minutes, making it reasonable to use RNN to predict outcomes. Silva et al.[25] used a Long Short-Term Memory (LSTM) network, Gated Recurrent Unit (GRU) and a simple RNN to predict the winner of a League of Legends match using data between minutes 0 and 25. When predicting for the time interval 10-15 minutes, Silva et al. found that simple RNN could be better than the LSTM and GRU, possibly because this was a small data set.

In a UCL CS final year project that ran last year, Tobias Edwards[26] used a single hidden layer LSTM classifier for the continuous outcome prediction of League of Legends matches. Edwards hypothesised the LSTM would achieve better results than other models, but he finally achieved a prediction accuracy of 83.44% at time interval 20-25 minutes which is similar to the results from Silva et al, and it turned out that some other models performed better. This may be due to the small data set and the necessary simplicity of the LSTM model (one single hidden layer).

For PUBG, a game that lasts about 30 minutes, provided that temporal data were appropriate and available, a recurrent neural network might work well. However, the data collected for this project contains only pre-match data and final in-game stats. Therefore, a Recurrent Neural Network was inappropriate for this project.

2.2.2 Decision Tree-Based Models

Decision tree-based models, such as Light GBM, XGBoost and Random Forest, are known to work very well on many hard problems and can give interesting insights into feature importance. It was for this reason that XGBoost was one of the models selected for investigation in the work of this project.

Tobias Edwards[26] also investigated the performance of XGBoost and Random Forest for the continuous outcome prediction problem mentioned above. It turned out that the decision tree-based models performed better than the LSTM when dealing with the 20-25 minutes interval, with the highest prediction accuracy of 89.02% being achieved by Random Forest.

Unger et al.[27] used Random Forest and Light GBM to predict the finish placement of PlayerUnknown's BattleGrounds using the same data set as this project. After exploratory data analysis, 21 new features were engineered based on the gameplay and the original features, resulting in a mean absolute error of 0.0205 by LightGBM. They concluded that a Gradient Boosting model can generate the best results for regression type problems, at least in this one. Also, that creating new features that better represent the player's metrics can improve the overall prediction score.

As already mentioned, XGBoost was one of the models chosen for investigation in this project, due to its excellent performance on many other problems. However unlike in the work of Unger et al., here the original raw data will be maintained and the investigation will focus on seeing if player type clustering can improve the overall prediction MAE.

2.3 Machine Learning Models

2.3.1 K-means Clustering

K-means clustering is a kind of unsupervised learning and its goal is to find K groups within the data.[28] The algorithm iteratively assigns each data to one of the K groups based on feature similarity. After performing a K-means clustering, K centroids are generated to label the original data and new data.[28]

However, K-means clustering is sensitive to the initialisation of centroids. Thus, if centroids are initialised as very biased points, it may result in poor clustering.[29]

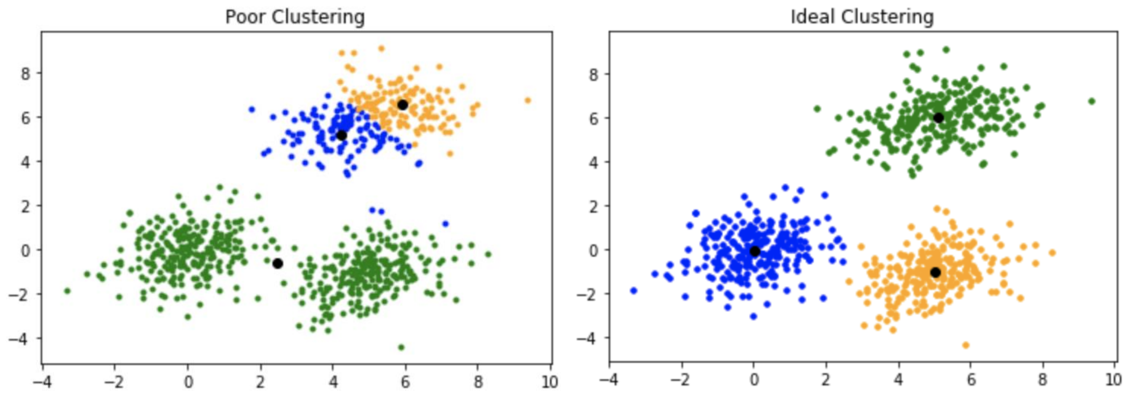


Figure 2.7: Poor Clustering and Ideal Clustering of K-means[6]

In order to overcome this drawback, K-means++ algorithm is used in this project. The algorithm uses an intelligent strategy to reliably initialise the centroids, followed by applying the remaining normal K-means algorithm. [29]

Calinski-Harabasz Index, also known as the Variance Ratio Criterion, is used in this project for evaluating the performance of K-means clustering. When using this metric, the higher the index, the denser and better separated the clusters.

[6] GeeksforGeeks. 2021. ML | K-means++ Algorithm - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/ml-k-means-algorithm/> [Accessed 14 April 2021].

2.3.2 Linear Regression

Linear regression is one of the basic supervised machine learning models. It assumes a linear relationship between the input variables and the single output variable. Linear regression seeks to find a set of weights w that minimize a loss function L, which is the sum of the squared residuals:[30]

$$L(y,x) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} w_j x_{ij})^2$$

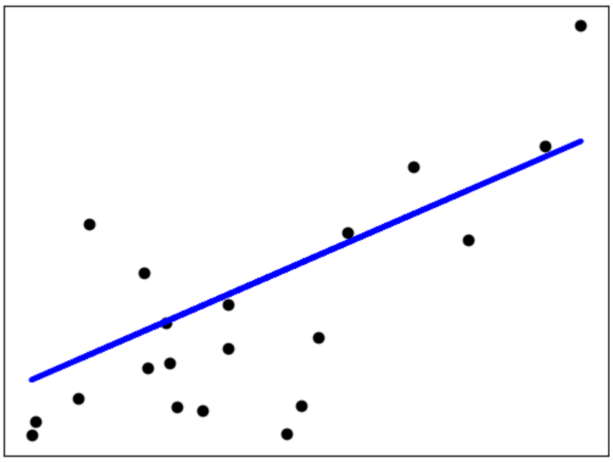


Figure 2.8: A Fitted Linear Regression[7]

L1 and L2 regularisation

A process of regularisation is important in order to avoid overfitting the model to the training data. The penalty applied for *L1 regularisation* is equal to the absolute value of the magnitude of the coefficients. Adding L1 regularisation to linear regression will result in the following loss function:

[7] Scikit-learn.org. 2021. Linear Regression Example — scikit-learn 0.24.1 documentation. [online] Available at: <https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html#sphx-glr-auto-examples-linear-model-plot-ols-py> [Accessed 14 April 2021].

$$L(y,x) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} w_j x_{ij})^2 + \lambda \sum_{j=1}^{p} |w_j|$$

L1 regularisation allows a form of feature selection, that is, an appropriate value of λ can make some weights become zero. The greater the value of λ, the more features are reduced to zero. This can eliminate some unimportant features and result in a subset of predictors that help mitigate multicollinearity and model complexity. Predictors that do not shrink to zero indicate their significance.[31]

In *L2 regularisation*, the added term is the sum of the square of the weights. Adding L2 regularisation to linear regression will result in the following loss function:

$$L(y,x) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} w_j x_{ij})^2 + \lambda \sum_{j=1}^{p} {w_j}^2$$

This constraint results in smaller weights, which tend to zero as the value of lambda increases. Shrinking the weights leads to a lower variance, which reduces the error. [31]

Elastic Net

Elastic Net consists of penalties from both L1 and L2 regularisation, which will result in the following loss function when added to linear regression:

$$L(y,x) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} w_j x_{ij})^2 + \lambda \left(\frac{1-\alpha}{2} \sum_{j=1}^{p} {w_j}^2 + \alpha \sum_{j=1}^{p} |w_j|\right)$$

Elastic Net reduces the impact of different features while not eliminating all features. It combines feature elimination from L1 regularisation and feature coefficient reduction from the L2 regularisation to improve the quality of model's predictions.[32]

2.3.3 XGBoost

Gradient boosting is a supervised machine learning technique which produces a prediction model in the form of a set of weak prediction models, usually decision trees.[33] Gradient boosting focuses on fitting each new predictor to the remaining error produced by the previous one. It aims to diminish the loss function and make the weak prediction closer to the actual value when new models are added.

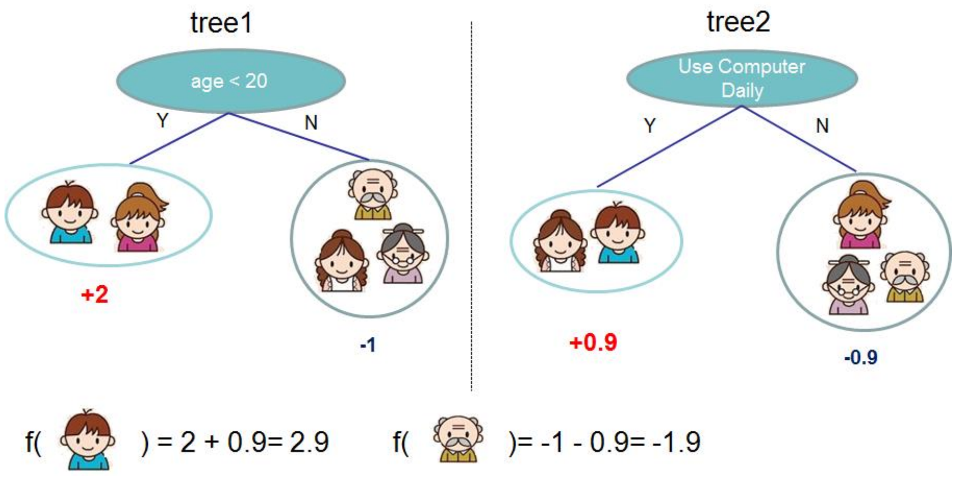


Figure 2.9: An Example of a Decision Tree Ensemble[8]

XGBoost is the abbreviation of eXtreme Gradient Boosting. It is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable.[34] It uses a more regularised model formalisation to control overfitting and provides a parallel tree boosting process, which solves many data science problems in a fast and accurate way.[35]

[8] Xgboost.readthedocs.io. 2021. Introduction to Boosted Trees — xgboost 1.4.0-SNAPSHOT documentation. [online] Available at: <https://xgboost.readthedocs.io/en/latest/tutorials/model.html> [Accessed 18 April 2021].

Chapter 3

Design and Methodology

3.1 Data Acquisition

As has been previously mentioned, the data used in the work of this project was acquired from Kaggle[9]. Kaggle collected about 6,500,000 anonymous player data from more than 65,000 games through the official PUBG Developer API. The data sets were originally collected for a prediction competition 3 years ago, but according to the Kaggle competition specification, once the competition rules are accepted, the data sets can also be used for other non-commercial purposes, including academic research. Lots of the final in-game statistics and pre-match data are provided in the Kaggle data set to predict the finish placement of players. The Kaggle data was split into a training set and a test set, and the prediction targets were provided only for the training set. This project mainly uses the provided training set, with a train-test splitting (internal to the work of the project) that will be explained in Section 3.3.2. The detailed nature of the data that will be used in this project will be discussed below, in Section 3.2.

3.2 Data Description

This data set consists of 21 kinds of final in-game statistics, 7 kinds of pre-match data and the finish placement for each player, which is the prediction target. The data fields and their definitions are as follows:

- DBNOs - Number of enemy players knocked out by a player.
- assists - Number of enemy players damaged by a player but killed by the teammates of that player.
- boosts - Number of boost items used by a player.

[9] https://www.kaggle.com/c/pubg-finish-placement-prediction

- damageDealt - Total damage dealt by a player. Self-inflicted damage, caused by weapons like frag grenade, is subtracted for calculating this quantity.
- headshotKills - Number of enemy players killed with headshots by a player.
- heals - Number of healing items used by a player.
- Id – A player's Id
- killPlace - Ranking in match of number of enemy players killed. This is an in-game statistic which is different to KillPoints.
- killPoints - Kills-based "ELO ranking" (ranking by "ELO points" that introduced in Section 2.1.3) of this player. This is an item of pre-match data which is displayed in the leaderboard of the ranked server.
- killStreaks - Maximum number of enemy players killed by a player in a short amount of time.
- kills - Number of enemy players killed by a player.
- longestKill - Longest distance between a player and the enemy player killed at time of death of the enemy player.
- matchDuration - Duration of match in seconds.
- matchId – ID to identify a match.
- *matchType* - String identifying the game mode that the data comes from. 16 different game modes are collected in this data set, with definitions in line with the game background presented in Section 2.1.3. For convenience, the strings and their corresponding definitions are listed again here as follows:
    - "solo" - solo mode in a ranked server.
    - "duo" - duo mode in a ranked server.
    - "squad" - squad mode in a ranked server.
    - "solo-fpp" - solo FPP mode in a ranked server.
    - "duo-fpp" - duo FPP mode in a ranked server.
    - "squad-fpp" - squad FPP mode in a ranked server.
    - "normal-solo" - solo mode in an unranked server.
    - "normal-duo" - duo mode in an unranked server.
    - "normal-squad" - squad mode in an unranked server.

    - “normal-solo-fpp” - solo FPP mode in an unranked server.
    - “normal-duo-fpp” - duo FPP mode in an unranked server.
    - “normal-squad-fpp” - squad FPP mode in an unranked server.
    - “crashfpp” - Crash Carnage mode that is played in first person.
    - “crashtpp” - Crash Carnage mode that is played in third person.
    - “flarefpp” - Flare Gun mode that is played in first person.
    - “flaretpp” - Flare Gun mode that is played in third person.
- rankPoints – “ELO ranking” based on wins and kills of a player. This is an item of pre-match data which is displayed in the leaderboard of the ranked server.
- revives - Number of times a player revived teammates.
- rideDistance - Total distance travelled in vehicles by a player, measured in meters.
- roadKills - Number of enemies killed by a player while in a vehicle.
- swimDistance - Total distance swum by a player, measured in meters.
- teamKills - Number of times a player killed a teammate.
- vehicleDestroys - Number of vehicles destroyed by a player.
- walkDistance - Total distance walked by a player, measured in meters.
- weaponsAcquired - Number of weapons picked up by a player.
- winPoints - Win-based “ELO ranking” of player. This is an item of pre-match data which is displayed in the leaderboard of the ranked server.
- 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 collected in the match by Kaggle.
- maxPlace – The worst placement in the match.
- winPlacePerc - The prediction target. This is the percentile winning placement, where 1 corresponds to 1st place, and 0 corresponds to last place in the match.

According to a tip from Kaggle’s data collector, *RankPoints* is inconsistent and will be deprecated in the next version of the official PUBG Developer API.[31] Therefore, *RankPoints* was not used for carrying out this project.

3.3 Data Preprocessing

3.3.1 Data Analysis

From Section 2.1.3 and Section 3.2, it can be seen that all players in a team must share not only the same *groupId*, but also the same *winPlacePerc*. While this would seem common sense, it was verified by counting the number of groups that contain the same *winPlacePerc* and calculating the percentage based on the total number of groups. It was discovered that, as expected, all groups shared the same *winPlacePerc*.

3.3.2 Parsing

Handling Missing Data

By searching "NaN" throughout the training set, it was discovered that only one piece of data has a missing target value. The game mode of that match was *solo-fpp* which would not affect the assumption made in Section 3.3.1 because the player by definition does not have any teammates. Removing one piece of data from the *solo-fpp* mode is obviously of little consequence in such a large data set and so the data with missing value was deleted.

Train-Test Splitting

As mentioned in Section 3.1, *winPlacePerc* was given only for the training set. Therefore, to evaluate the performance of the machine learning models developed for this project, the original training set must be split into a new training set and a new test set.

The train-test split should ensure that the new training set and test set have the same properties as the original data sets. In this case the need to keep all instances with the same *groupId* in either the training set or the test set means that the data cannot be assigned completely randomly. The splitting was based on *groupId*. This was important because any single team, identified by its *groupId*, should end up wholly in either the training or the test set. (This was checked in relation to Kaggle's train-test split and found to be true.) The data here were therefore split into 80% training and 20% testing, subject to the above necessary restriction in relation to *groupId*.

3.3.3 Preliminary Data Analysis and Visualisations

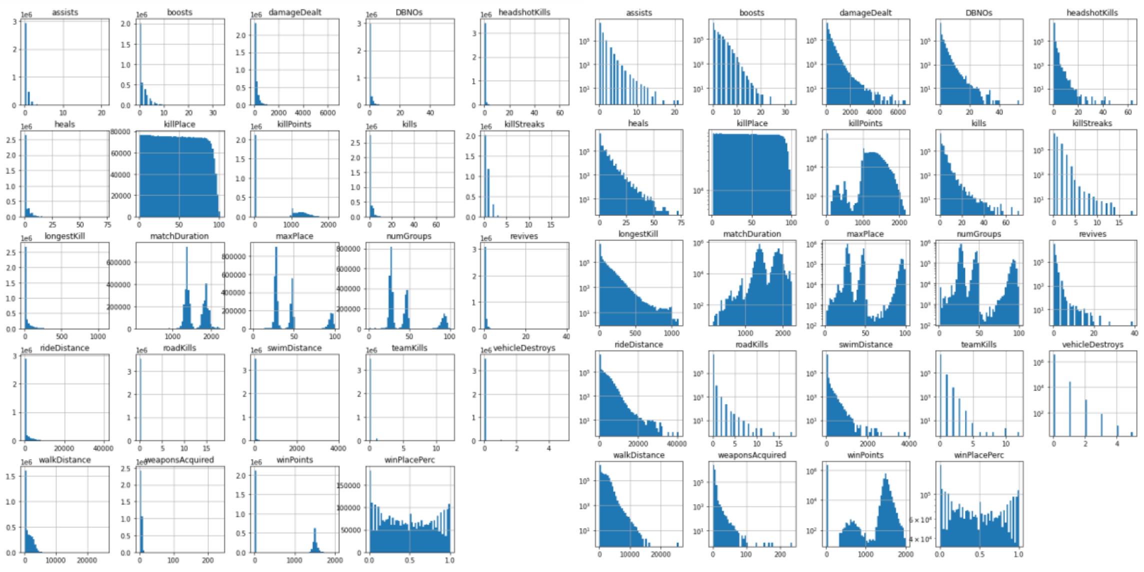


Figure 3.1: Visualisation of data fields

To get a deeper understanding of the data, the distributions for each data field excluding *Id, matchId, groupId and matchType* were examined in Figure 3.1, where the x-axis displays the value, and the y-axis displays the number of players who achieved that value.

The left-hand side plots show the distribution without any scaling; in this case the presence of rare values is difficult to see because each data field is dominated by a single value or a small set of values. Therefore, another set of plots which set the y-axis to a log scale are displayed on the right-hand side. The rare values are much clearer in these plots (e.g. the maximum *damageDealt* is more than 6000, but it's hard to identify from the normal-scaled plot), and their existence will be significant when in further analysis.

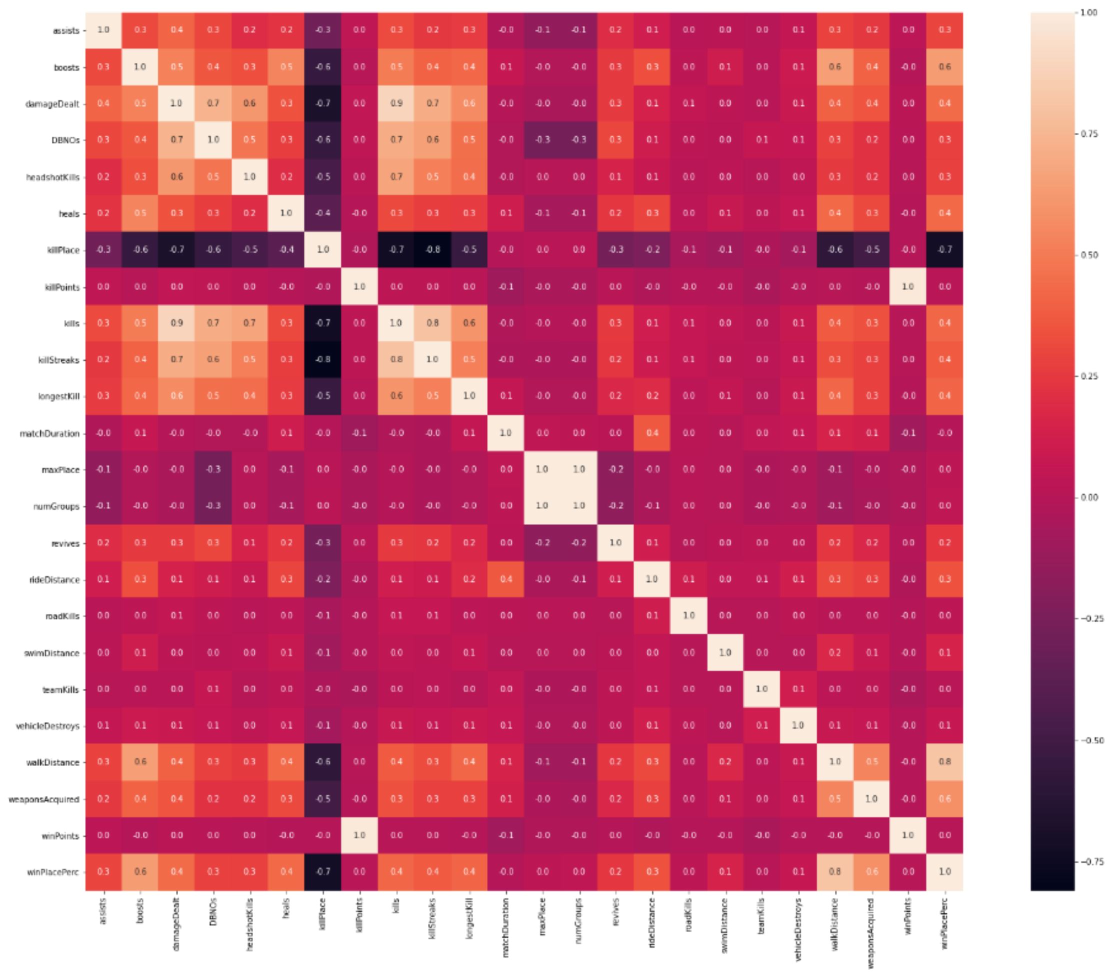


Figure 3.2: Correlation Heat Map

Figure 3.2 shows the correlation heat map between the attributes. By focusing on the correlation between the features and the output (i.e. the bottom row), it can be seen that features like *assists, boosts, damageDealt, walkDistance* are positively correlated to *winPlacePerc*. The only negatively correlated one is *KillPlace*, which is by definition inversely proportional to *kill*s. Therefore, it was hypothesised that the *winPlacePerc* may be highly dependent on the players' performance.

3.3.4 Data Transformation

Scaling

A tree-based model does not require scaling of the inputs since features are binary split. XGBoost is essentially an ensemble algorithm comprised of decision trees, thus, the

XGBoost algorithm does not require scaling either. Typically, also, linear regression does not require feature scaling, unless it is regularised. However Elastic Net is used in this project, a regularised form linear regression in which the penalty for each coefficient depends largely on the scale associated with the feature.[37] K-means Clustering, which is used centrally in this project, is also highly sensitive to scaling since it uses Euclidean Distance to form the cohorts.[38] Thus it was necessary to choose a scaling method.

As can be seen from the plots on the right-hand side of Figure 3.1, since there are very few extreme large values in each data field, the use of min-max scaling will cause a large number of values to be very small, except the rare values. Therefore, min-max scaling is not appropriate for these data sets. And clearly, according to the left-hand side figures of Figure 3.1, none of the features follow a normal distribution. Therefore, standardisation is also inappropriate. Instead, scaling was performed on a per-feature basis, using a simple power-of-ten division. The scaling of each feature is displayed in Table 3.1 (for those features that are not shown, no scaling was done).

Table 3.1: Scaling

Before	After
damageDealt	damageDealt*1/100
killPlace	killPlace*1/100
killPoints	killPoints*1/1000
longestKill	longestKill*1/100
matchDuration	matchDuration*1/1000
maxPlace	maxPlace*1/100
numGroups	numGroups*1/100
rideDistance	rideDistance* 1/1000
swimDistance	swimDistance* 1/100
walkDistance	walkDistance * 1/1000
weaponsAcquired	weaponsAcquired * 1/10
winPoints	winPoints * 1/1000

Categorical Data

The only item of categorical data among the features used here is *matchType*. It can be seen in Figure 3.3 and Table 3.2 that some *matchType*s are much more popular than others; *squad-fpp* is the most popular (over 30% of instances) and *normal-duo* the least (less than 0.005% of instances).

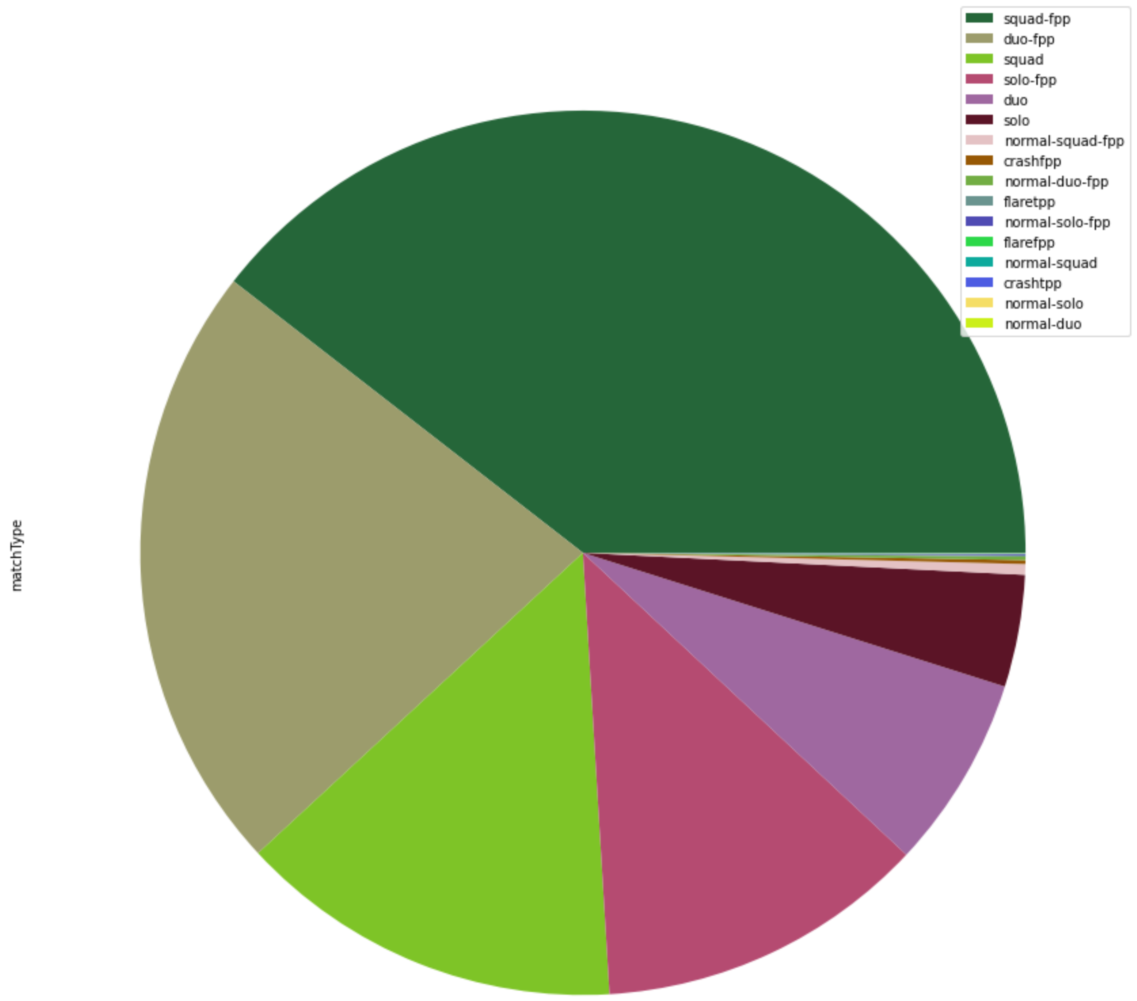


Figure 3.3: Visualisation of *matchType* in a pie chart

Table 3.2 *matchType* Counts and Percentage

matchType	Counts	Percentage
crashfpp	4972	0.1118%
crashtpp	282	0. 0063%
duo	251120	5.6470%
duo-fpp	796987	17.9220%
flarefpp	516	0.0116%
flaretpp	2041	0.0459%
normal-duo	163	0.0037%
normal-duo-fpp	4428	0.0996%
normal-solo	243	0.0055%
normal-solo-fpp	1336	0.0300%
normal-squad	398	0.0089%
normal-squad-fpp	13959	0.3139%
solo	146105	3.2855%
solo-fpp	429670	9.6621%
squad	500754	11.2606%
squad-fpp	1403320	31.5568%

However for training purposes the actual numbers of instances of each *matchType* will be at least as important as the proportions. It can be seen from Table 3.2, that game modes like *normal-duo, normal-solo, crashtpp* contain only a few hundred pieces of data, which may lead to a high risk of overfitting if training a separate model on these individual *matchTypes*.

Combining all the modes with small numbers of members into a new category was considered but due to the independent gameplay of each game mode, it was decided instead to apply one-hot encoding to each of the *matchTypes*. One-hot encoding is used to quantify categorical data, generating a vector whose dimension is equal to the number of categories in the data set. If the data belongs to the i-th category, then the vector is assigned to 0 except for the i-th component, which is assigned to 1.[39]

3.5 Data Clustering

3.5.1 K-means: How Many Initialisations?

As mentioned in Section 2.3.1, K-means Clustering is highly dependent on the quality of centroid initialisations, so that the algorithm needs to run multiple times in order to find the optimal interpretation of the data (that with the least sum of squared distances of samples to their closest cluster centre). In Scikit-Learn this number of initialisaitons is referred to as *n_init*, and is by default set to 10. Although the Kmeans++ algorithm can use a smarter way to generate the initial centroids, the assumption that setting n_init to 10 is sufficient should still be checked. Each clustering was therefore run two times with different seeds for checking if n_init=10 was enough. If the best-clustering results were different, one could then conclude the number of initialisations was not enough and that the number of initialisations should be increased.

3.5.2 Evaluation of Clusters

As mentioned in Section 2.3.1, the evaluation of cluster quality will be based on the Calinski-Harabasz Index. This choice of metric was made because others such as the Silhouette Index were too time-consuming to compute on this large data set.

3.5.3 What Value of K?

To investigate how many clusters can lead to the best clustering, a range of K-values from 2 to a decided maximum were chosen. Then K-means clustering was performed with each of them, and the one with the best Calinski-Harabasz Index taken to be the best number of clusters. Should K=2 be seen to be apparently optimal, some further investigation is necessary as this observation could be compatible, also, with a data set in which there were no distinguishable clusters. The features of each of the two centroids are mapped out in a line chart. If there are indeed two clusters there should be two easily visually distinguishable lines. This form of visualisation can be used for more than two apparent clusters but it is especially useful to distinguish between K=2 and no clusters.

3.6 Model Training, Validation and Evaluation

3.6.1 K-Fold Cross-Validation

The choice of hyperparameters (which will be introduced in Section 3.6.2) should obviously not be done on the basis of test set performance. The choice should be made using a subset of the training set known as a validation set. It is usual in fact to do this on multiple subsets of the training set using a process known as *cross-validation*.[40]

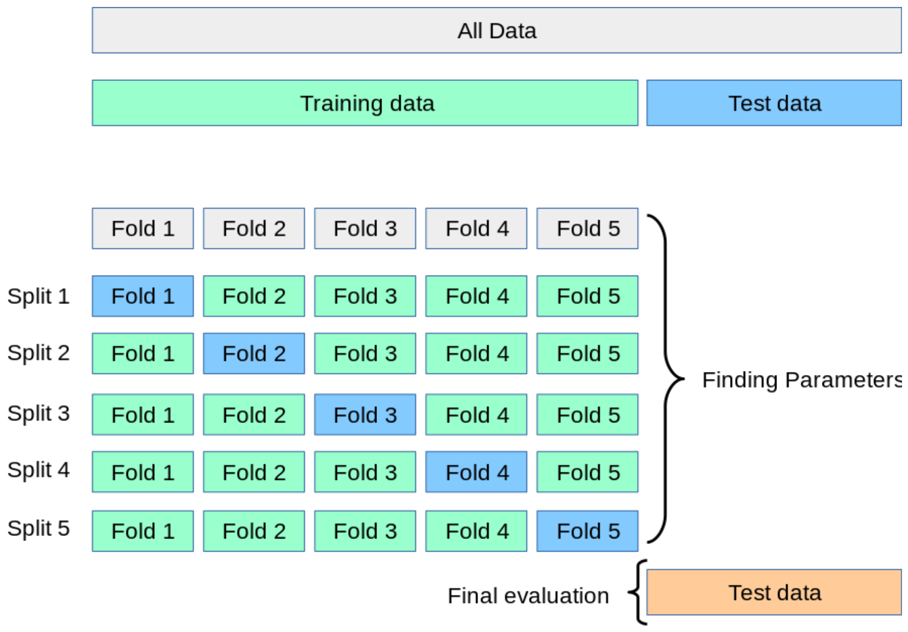


Figure 3.4: 5-Fold Cross-Validation[10]

In *k-fold cross-validation*, the training set is split into k smaller sets, and the model is trained on the $k-1$ folds and evaluated on the rest of the training data. Figure 3.4 shows a 5-fold cross-validation in which the training set is split into 5 folds and 5 different splits carried out to get all possible evaluations. The performance measure reported by a *k-fold*

[10] Scikit-learn.org. 2021. *3.1. Cross-validation: evaluating estimator performance — scikit-learn 0.24.1 documentation*. [online] Available at: <https://scikit-learn.org/stable/modules/cross_validation.html> [Accessed 21 April 2021].

cross-validation is the average of the values computed in the loop.[40]

3.6.2 Hyperparameter Tuning

A hyperparameter is a parameter that is external to the model and which cannot be learned from data. Default choices of hyperparameters may result in a model performing suboptimally, requiring these values to be chosen on the basis of model performance on a validation subset or subsets. Finding the optimal model, especially if using cross-validation, can be extremely time-consuming, and would be expected to be so in this case, with an extremely large data set. Therefore, a set of the most influential hyperparameters was selected for each model for optimisation.

Scikit-Learn's *Randomised Search* with 5-fold cross-validation of the training set was mainly used to evaluate model performance on random combinations of hyperparameter values specified; the resulting hyperparameters might not be the optimal but the computational time was reduced significantly. *Grid Search* can fairly evaluate model performance on all possible combinations but the time consumed for the optimisation in this huge data set could be several times longer than a *Randomised Search*.

Custom Randomised Search was also developed for splitting the validation set using the same method as train-test splitting (mentioned in Section 3.2.2), which is based on *groupId*. *Custom Randomised Search* extracts one validation set for evaluation, and the models are trained on the remaining data with random combinations of hyperparameter values specified. This method was only used if overlapped *groupId* could exist in both validation set and the remaining data when using cross-validation.

Listed below is the selection of hyperparameters tuned for each model.

SGDRegressor with loss = 'squared_loss' and penalty = 'elasticnet' (therefore it is essentially a linear regression with elastic net regularisation):

- alpha – a constant that multiplies the regularisation term. The higher the value, the stronger the regularisation.
- l1_ratio - the elastic net mixing parameter.
- max_iter - the maximum number of epochs over the training data.
- eta0 - the initial learning rate.

XGBoost:

- learning_rate – the learning rate.
- max_depth - maximum depth of each tree.
- min_child_weight - minimum weight required in order to create a new node in the tree.
- colsample_bytree - the fraction of features that will be used to train each tree.
- n_estimators - number of gradient boosting trees.

3.6.3 Model Evaluation

The prediction errors were calculated using MAE, short for *mean absolute error*.

One model for the entire data set

The error in this case is calculated by the following equation,

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

where *n* is the number of samples, *y* is the true value and *x* is the predicted value.

Separate models for separate subsets

When dealing with separate subsets, the error will be calculated based on the number of subsets. The *combined MAE* was calculated as the following equation,

$$MAE_{combine} = \frac{1}{\sum_{i=1}^{k} n_i} \times \sum_{i=1}^{k} (MAE_i \times n_i)$$

where n_i is the amount of data in the i-th subset, MAE_i is the MAE in the i-th subset and k is the number of subsets.

Chapter 4

Results

4.1 Investigation of *groupId*-Based Prediction

According to the discussion in Section 3.3.1, instead of using the whole training set to train the models, it may be beneficial to train the model on a per-group basis. Two experiments were conducted to see if there was a benefit to *groupId*-based training:

1. Train the models in a traditional way.
2. Train the models based on *groupId*.

In the first experiment, as *groupId* should not be overlapped in this situation, the hyperparameter optimisation was carried out by using *Custom Randomised Search*. The optimised models were then trained on the whole training set and tested on the test set.

For the second experiment, the data in the training set were grouped by the *groupId* and each group was replaced by a new piece of data, which was generated by averaging the values of each feature. Therefore, there was no chance to have overlapped *groupId* when performing cross-validation, and so *Randomised Search* was used for optimising the hyperparameters. The optimised models were then trained on the training set. To apply the effect of *groupId* on the test set without reducing the size of the test set, the data in the test set were also grouped by the *groupId*, but instead of replacing by a new piece of data, each of them was transformed into the average data. Finally, the trained models were used to predict on this transformed test set.

The results are summarised in Table 4.1. To assess the statistical significance of the results p-values were also calculated by using a two-sided test for the null hypothesis that the two sets of absolute errors have identical mean (i.e. the two MAE are the same). A large p-value indicates the null hypothesis cannot be rejected while a low p-value means the null hypothesis can be rejected. The p-values are summarised in Table 4.2.

Table 4.1: Results

Index	Model	Scenario	MAE
1	SGDRegressor	Train on all data	0.09027
2	SGDRegressor	Train on all data – groupId based	0.07481
3	XGBoost	Train on all data	0.05752
4	XGBoost	Train on all data – groupId based	0.04910

Table 4.2: p-values

Significantly different?	p-value
1 vs 2	0
3 vs 4	0

From the results of Table 4.1, it can first be concluded that XGBoost, as a strong gradient boosting algorithm, performed much better than SGDRegressor for this problem. Moreover, it is apparent from Table 4.2 that *groupId*-based training and testing give a statistically significant improvement of the model performance. (Note: p-values reported as 0 are zero to within machine precision.) Therefore, it was decided that all further analysis would all be *groupId*-based since it was obviously beneficial.

4.2 Clustering Analysis

After training on the entire training set, the next step would be to train separate models for separate subsets. But first these subsets needed to be generated. Therefore, K-means clustering was used to investigate the possible clusters within the training set.

Calinski-Harabasz Index vs number of clusters for two different sets of 10 initialisations (given the caution about n_init discussed in Section 3.5.1) are displayed in Figure 4.1.

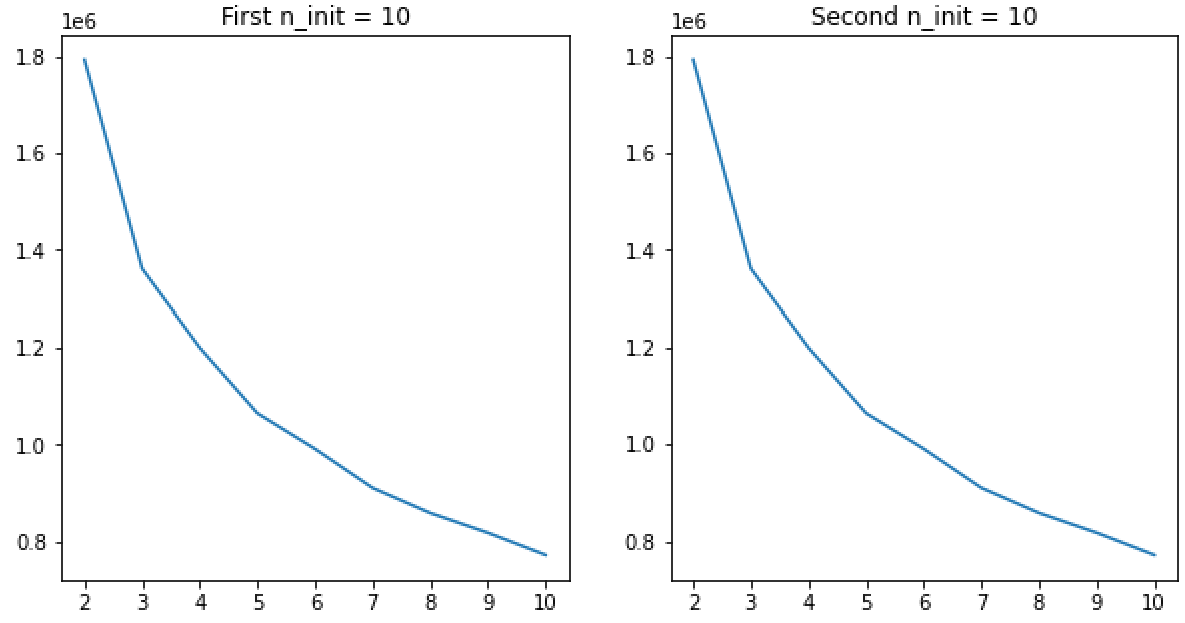


Figure 4.1: Calinski-Harabasz Index vs Number of Clusters

First, it is clear that Figure 4.1 shows almost identical results for each set of initialisations. Therefore, one can conclude 10 initialisations is sufficient for this K-means clustering. (While this same check was performed for all other cases in which a clustering analysis was done, because n_init=10 was found always to be sufficient dual figures as above will not be presented again in this report.)

Second, it is clear also that, the best number of clusters is two clusters. As mentioned in Section 3.5.3, a seemingly-optimal result for two clusters may in fact indicate no clusters. Therefore, a visual test to distinguish between two clusters and no clusters was done. The features of each centroid were plotted in Figure 4.2 (following page), with features laid out on the x-axis as in Table 4.3 (following page), with the tabulated abbreviations.

Table 4.3 x-axis Abbreviations

'as' = 'assists'	'mP' = 'maxPlace'
'bs' = 'boosts'	'nGs' = 'numGroups'
'dD' = 'damageDealt'	'rs' = 'revives'
'DBNOs' = 'DBNOs'	'rD' = 'rideDistance'
'hKs' = 'headshotKills'	'rKs' = 'roadKills'
'hs' = 'heals'	'sD' = 'swimDistance'
'kP' = 'killPlace'	'tKs' = 'teamKills'
'kPs' = 'killPoints'	'vDs' = 'vehicleDestroys'
'ks' = 'kills'	'wD' = 'walkDistance'
'kSs' = 'killStreaks'	'wA' = 'weaponsAcquired'
'lK' = 'longestKill'	'wPs' = 'winPoints'
'mD' = 'matchDuration'	

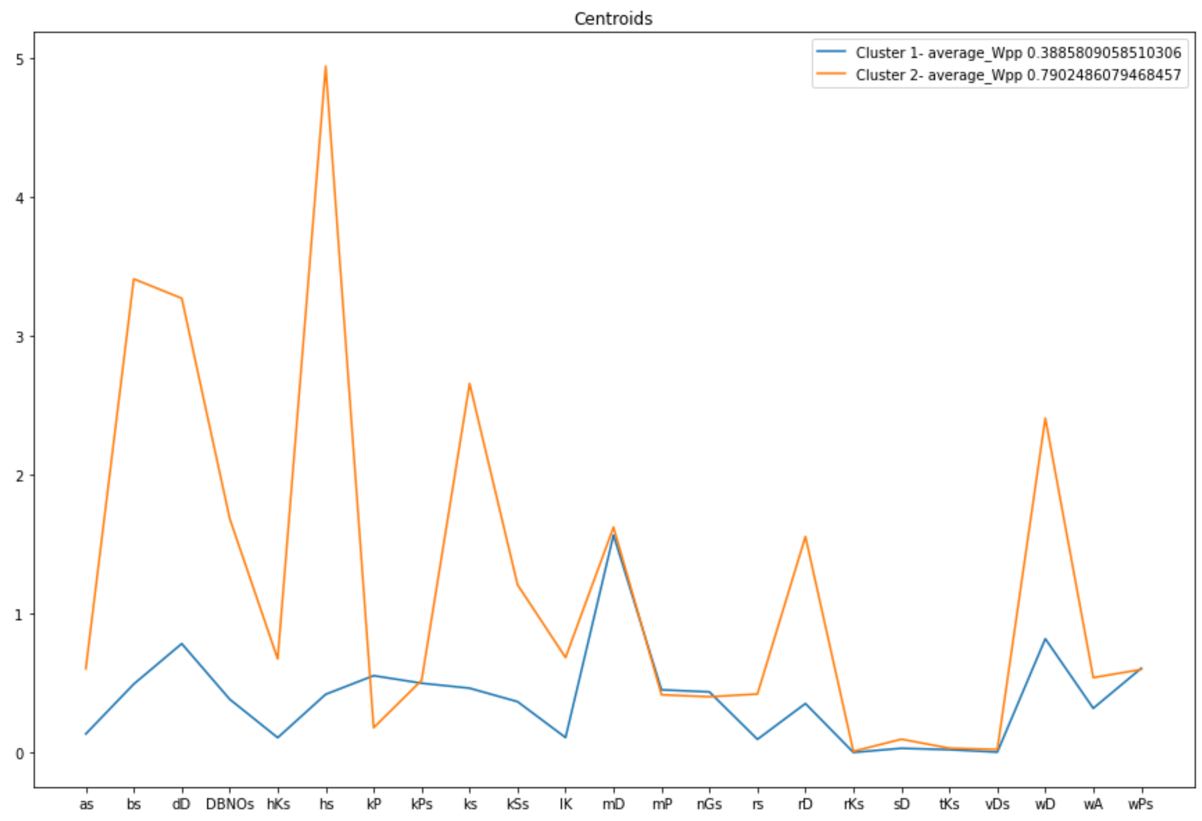


Figure 4.2: Distinguishing Between Two Clusters and No Clusters

The two centroids in Figure 4.2 are clearly different, thus the two clusters are different. In addition, the figure presents a very interesting result. The y-axis represents the centroid-average number of times an activity, such as *heals* (hs, with in this case an average around 5) was performed, and it is clear that the main thing that distinguishes the clusters is the level of activity of the players within it. Moreover, the cluster containing the more active players has the higher average *winPlacePerc* (shown in the upper right hand corner of the figure). It is thus apparent playing as an 'active' player instead of being a 'lazy' one is the best way to win this game. The activity must also be continuous through the game, as large values for (for example) *swimDistance* and *walkDistance* could not be acquired otherwise. This discovery about player activity is by no means an obvious one as, given the battle royale nature of the game, an alternative strategy might have been to be relatively inactive until most other players had been eliminated, focusing instead on stockpiling weapons and other resources. Finally, simply in its display of two meaningfully different clusters, Figure 4.2 also gives a hint that the prediction may be improved by using player type clustering and building a separate prediction model for each type of player. This possibility will be explored below.

4.3 Predicting Using Player Type Clustering

As mentioned in Section 4.1, all following experiments, including this one, would be based on the transformation by *groupId*, and use *Randomised Search* for hyperparameter optimisation. For this experiment, a K-means clustering with two clusters (as evidenced by Figure 4.2) was first applied to the training set, followed by the training of separate models for each of the two clusters.

The summarised results and calculated p-value are presented on the following page in Table 4.4 and Table 4.5, respectively.

Table 4.4: Results

Index	Model	Scenario	MAE
1	SGDRegressor	Train on all data – groupId based	0.07481
2	SGDRegressor	Train on player type clustering	0.06898
3	XGBoost	Train on all data – groupId based	0.04910
4	XGBoost	Train on player type clustering	0.05051

Table 4.5: p-values

Significantly different?	p-value
1 vs 2	0
3 vs 4	7.26015e-90

Inspection of the tables shows the use of player type clustering improved the performance of SGDRegressor but decreased the performance of XGBoost. Hence, the value of the clustering was seen only for the weaker model.

4.4 Predicting Using Game Mode (*matchType*)

The results of Section 4.3 show that "multiple models training" (in that case based on player type clustering) benefits at least the results of SGDRegressor. However the game itself provides a kind of splitting, which is *matchType*, and hence a data-splitting experiment that built separate models for each *matchType* was carried out. The details were similar to those of Section 4.3. The *matchTypes* were transformed into an array and separate models were trained for each *matchType*.

The summarised results and calculated p-values are presented on the following page in Table 4.6 and Table 4.7, respectively.

Table 4.6: Results

Index	Model	Scenario	MAE
1	SGDRegressor	Train on all data – groupId based	0.07481
2	SGDRegressor	Train on player type clustering	0.06898
3	SGDRegressor	Train on each *matchType*	0.07302
4	XGBoost	Train on all data – groupId based	0.04910
5	XGBoost	Train on player type clustering	0.05051
6	XGBoost	Train on each *matchType*	0.04944

Table 4.7: p-values

Significantly different?	p-value
1 vs 2	0
1 vs 3	1.2538e-75
2 vs 3	0
4 vs 5	7.26015e-90
4 vs 6	9.96342e-07
5 vs 6	3.52778e-52

It is apparent from these tables that SGDRegressor performed worse when training on each *matchType* than training on player type clustering but better than training on the whole training set. XGBoost gave opposite results, in which training on each *matchType* was slightly better than training on player type clustering while slightly worse than training on the whole training set. However, the overall winner is still XGBoost trained on the full data set, with no form of splitting.

4.5 Prediction Using Player Type Clustering Inside Game Mode (*matchType*)

From the previous results, it could be seen that for SGDRegressor training on player type clustering was beneficial compared to training on the whole training set. Although XGBoost gave opposite results, favouring training on the whole data set, the difference of the MAE is less statistically significant compared to the improvement of SGDRegressor. Therefore, this final experiment sought to investigate whether the prediction could be enhanced by using player type clustering within separate *matchType*s.

4.5.1 Clustering Analysis for Each *matchType*

The clustering was again conducted by K-means clustering with 10 initialisations. The resulting Calinski-Harabasz Index vs Number of Clusters plots for each *matchType* are shown in Figure 4.3.

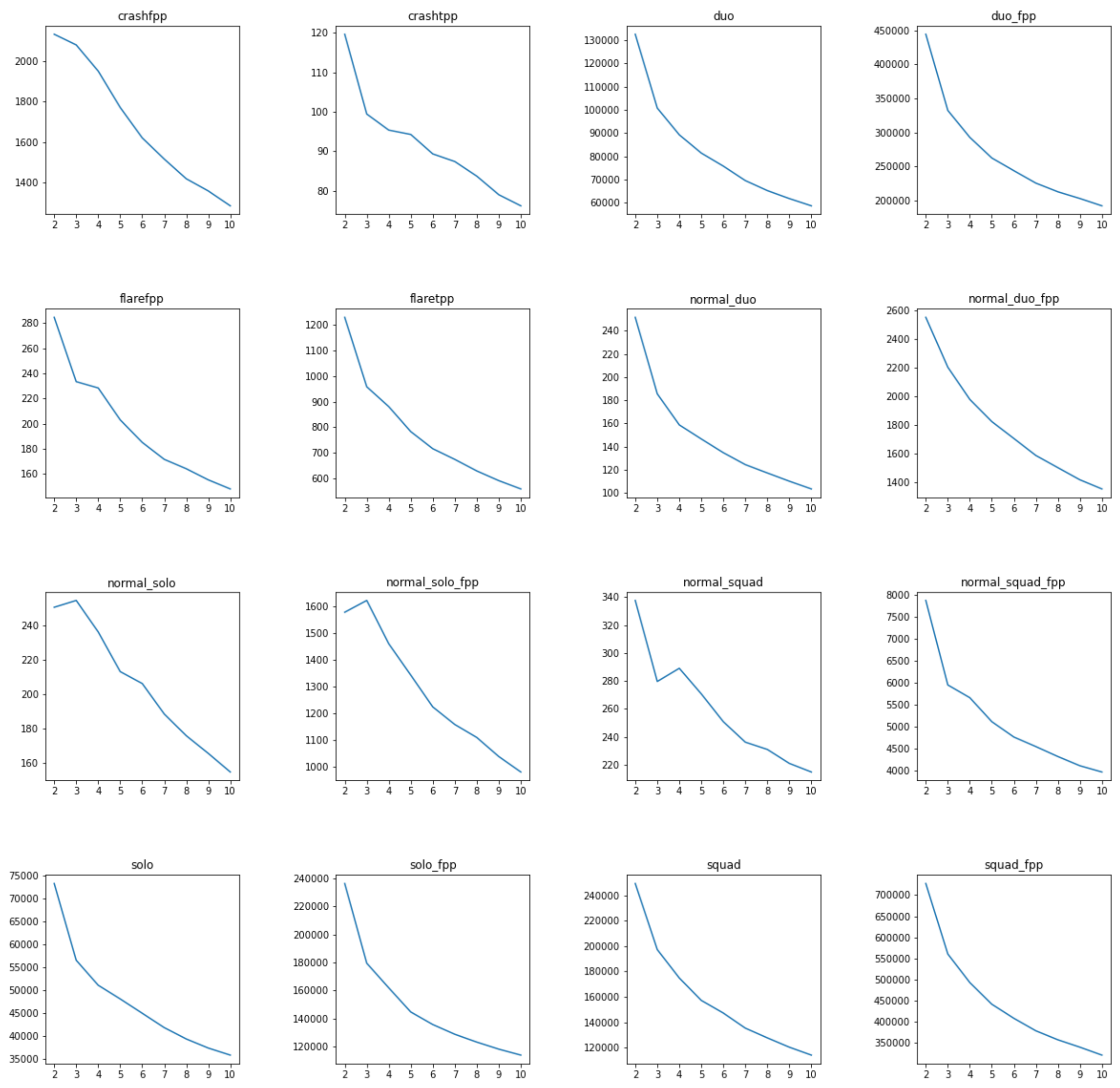


Figure 4.3: Calinski-Harabasz Index vs Number of Clusters for Each *matchType*

From Figure 4.3, it is apparent that each *matchType* except *normal-solo* and *normal-solo-fpp* got its highest Calinski-Harabasz Index when there were two clusters. The exceptions were *normal-solo* and *normal-solo-fpp*, which achieved their highest Calinski-Harabasz Indices for three clusters.

As previously, when two clusters appeared optimal, further analysis was needed to distinguish between no clusters and two clusters, for each case. It was also of interest to investigate what the three centroids of the exceptional cases of *normal-solo* and *normal-solo-fpp* looked like. An example plot for a two-cluster case, *crashfpp*, is shown in Figure 4.4 (the other two-cluster cases show a similar shape). The investigation of the three clusters of *normal-solo* is shown in Figure 4.5. *Normal-solo-fpp* follows the same trend, and so a plot for the other three-cluster *matchType* is not shown here.

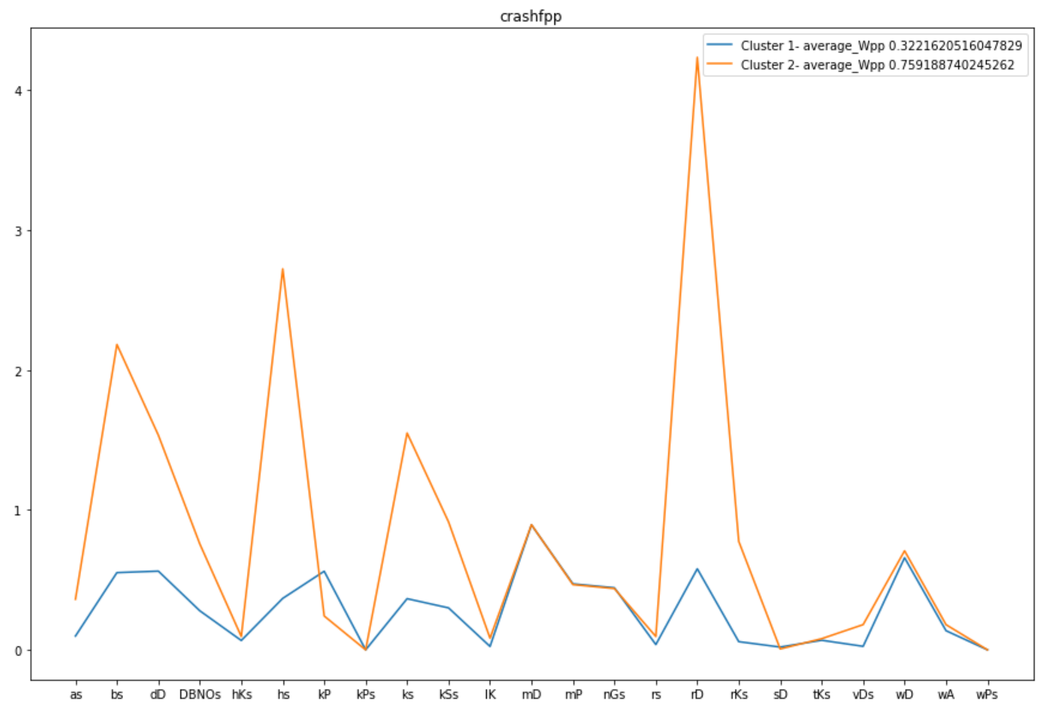


Figure 4.4: Distinguishing between no clusters and two clusters for *crashfpp*

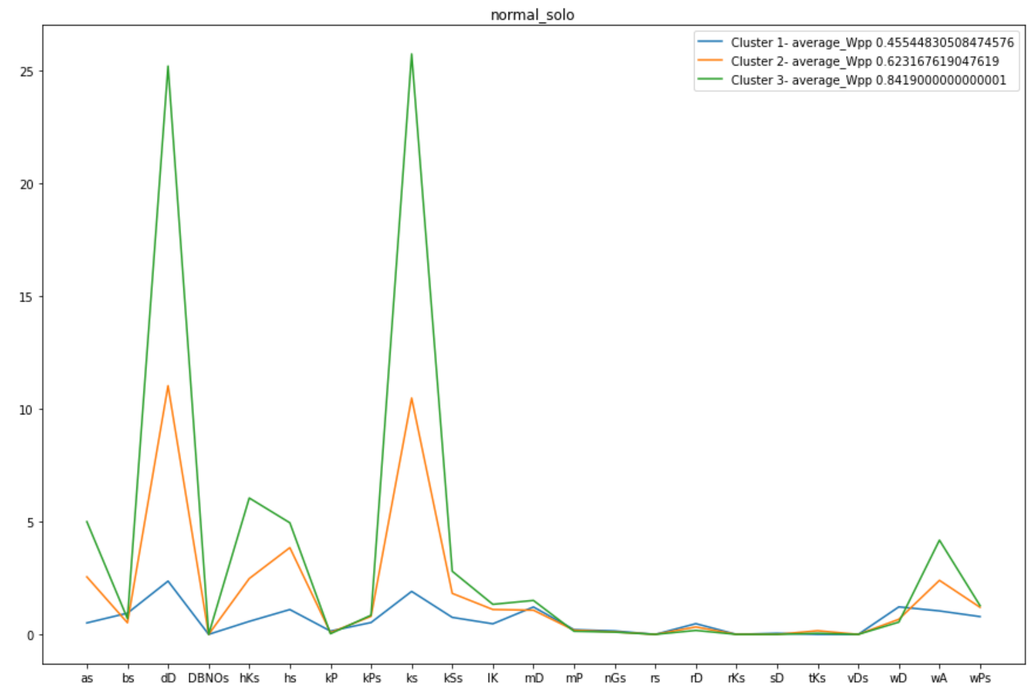


Figure 4.5: Investigation of the difference between three clusters of *normal-solo*

As with the cluster analysis for the full data set, it is apparent that within *matchType*s players are again being separated in relation to their activity levels, with higher activity being correlated with a higher average *winPlacePerc* (shown on the upper right of the plots). This is true also for the two *matchType*s for which the optimal number of clusters is three: here, the players are being grouped into 'active' (with the highest chance of winning), 'lazy' (with the lowest chance), and now, for these two *matchTypes*, a third group of 'moderately active' players with an intermediate chance of winning. However it can been seen that the most important activities differ between the game modes (and this is likely to be true for any pair of game modes): *damageDealt* and *kills* are the most important features for *normal-solo*, while for winning a *crashfpp* match, it is most beneficial to use boosted items, use heal items, kill and move frequently.

Prediction Results

Since some *matchType*s contain only very limited data, as shown in Table 3.2, subsequent clustering might result in very small subsets which would have higher chances to overfit. Thus, two different prediction strategies were designed for this experiment:

1. Train on player type clusters within each *matchType* independently of the size of the *matchType*.
2. Train on player type clusters within each *matchType* only when the size of the *matchType* is above a certain threshold.

For the second strategy, the size-threshold (number of *matchType* members) was set to be 2000. The results and p-values are presented on the following page in Table 4.8 and Table 4.9.

Table 4.8: Results

Index	Model	Scenario	MAE
1	SGDRegressor	Train on all data – groupId based	0.07481
2	SGDRegressor	Train on player type clustering	0.06898
3	SGDRegressor	Train on each *matchType*	0.07302
4	SGDRegressor	Train on the clusters within each *matchType*	[29114313.90932][11]
5	SGDRegressor	Train on the clusters within each *matchType* – with threshold	0.06849
6	XGBoost	Train on all data – groupId based	0.04910
7	XGBoost	Train on player type clustering	0.05051
8	XGBoost	Train on each *matchType*	0.04944
9	XGBoost	Train on the clusters within each *matchType*	0.04979
10	XGBoost	Train on the clusters within each *matchType* – with threshold	0.04968

Table 4.9: p-values

Significantly different?	p-value
1 vs 2	0
1 vs 3	1.2538e-75
1 vs 5	0
2 vs 3	0
2 vs 5	9.71743e-08
3 vs 5	0
6 vs 7	7.26015e-90
6 vs 8	9.96342-07
6 vs 9	4.04797e-23
7 vs 8	1.63548e-45
7 vs 9	3.52778e-52
8 vs 9	5.33341e-07
9 vs 10	0.104321

[11] Too few training instances produced an anomalous result. This model was therefore not compared with any others in Table 4.9.

SGDRegressor

When training on the player type clusters within each *matchType*, SGDRegressor did not produce a meaningful result when training on one extremely small subset, a cluster from *normal-solo* that contained only 20 pieces of data. Therefore, exceedingly biased predictions were produced. Nevertheless, SGDRegressor got its best performance using this subsection's "player type clusters within *matchType*s" method after setting a threshold such that the worst of the overfitting was prevented.

XGBoost

XGBoost did not misbehave in the way that SGDRegressor did when dealing with the one very small data set, producing anomalous predictions for the test set. There is no significant difference between training on player type clusters within any *matchType* or doing this only when the size of the *matchType* is above a certain threshold. In addition, training on *matchType* only is better than training on player type clusters within each *matchType*. Most importantly, however, training on all data is again the overall winner, better than training on player type clusters within each *matchType*.

Chapter 5

Conclusions and Evaluation

5.1 Discussion

The use of property of *groupId*, forcing all members of a team to have the same predicted finish placement (since they would achieve the same actual placement) was crucial in this project, the prediction results being greatly improved in this way. The hypothesis that placement prediction could be improved by splitting the data into subsets and training a separate model for each subset was supported only for a weaker model, such as SGDRegressor. For a powerful model such as XGBoost, neither splitting into *matchType* nor grouping by player type improved the prediction result. Overall, it therefore appears best to use a more powerful model and train on all available data.

5.2 Future Work

This project used XGBoost and Linear Regression with Elastic Net Regularisation to investigate the influence of player type clustering on the quality of the final prediction. Several other machine learning models could be tried in future, such as a Neural Network or Support Vector Machine. The hyperparameter tuning was performed using *Randomised Search* while if applicable, *Grid Search* could be applied to get better hyperparameter settings. Finally, given sufficient computational resources, the Silhouette Coefficient could be used for clustering to check if the results are the same as for the Calinski-Harabasz Index.

5.3 Concluding Thoughts

While it was not the case that player type clustering could improve the overall placement prediction result, this clustering even so gave a valuable insight into the workings of PlayerUnknown's BattleGrounds and into how to win the game. It was seen that 'active' players (though the most useful activities appeared to depend to some degree on the game mode (*matchType*)) were more likely to win, with the importance of high values relating to swimming, walking, or driving implying that this activity should be continuous throughout the game. Conversely, ‘lazy’ players who hide without fighting, moving or using items, were found more likely to get a bad finish placement.

Bibliography

[1] Hamari, J. and Sjöblom, M. (2017), "What is Esports and why do people watch it?", Internet Research, Vol. 27 No. 2, pp. 211-232. https://doi.org/10.1108/IntR-04-2016-0085

[2] Medium. 2021. *The History and Evolution of Esports*. [online] Available at: <https://bountiegaming.medium.com/the-history-and-evolution-of-Esports-8ab6c1cf3257> [Accessed 8 April 2021].

[3] Historyofinformation.com. 2021. SPACEWAR: Fanatic Life and Symbolic Death Among the Computer Bums : History of Information. [online] Available at: <https://www.historyofinformation.com/detail.php?id=2514> [Accessed 8 April 2021].

[4] Kelly, M., 2021. *The most watched Esports events of 2020 | Dot Esports*. [online] Dot Esports. Available at: <https://dotEsports.com/streaming/news/the-most-watched-Esports-events-of-2020> [Accessed 8 April 2021].

[5] Visual Capitalist. 2021. *How the Esports Industry Fares Against Traditional Sports*. [online] Available at: <https://www.visualcapitalist.com/how-the-Esports-industry-fares-against-traditional-sports/> [Accessed 8 April 2021].

[6] Krafton.com. 2021. KRAFTON GAME UNION. [online] Available at: <https://www.krafton.com/en/m/studios/bluehole.html> [Accessed 8 April 2021].

[7] Polygon. 2021. The Game Awards forgot to give PlayerUnknown's Battlegrounds its only award last night. [online] Available at: <https://www.polygon.com/2017/12/8/16752674/game-awards-PlayerUnknown's Battlegrounds-best-multiplayer-game> [Accessed 8 April 2021].

[8] Gamesradar.com. 2021. The Legend of Zelda: Breath of the Wild scores big at the 35th Golden Joystick Awards presented with OMEN by HP | GamesRadar+. [online] Available at: <https://www.gamesradar.com/the-legend-of-zelda-breath-of-the-wild-scores-big-at-the-35th-golden-joystick-awards-presented-with-omen-by-hp/> [Accessed 8 April 2021].

[9] Kaggle.com. 2021. PlayerUnknown's Battlegrounds Finish Placement Prediction (Kernels Only) | Kaggle. [online] Available at: <https://www.kaggle.com/c/PlayerUnknown's Battlegrounds-finish-placement-prediction/overview> [Accessed 11 April 2021].

[10] Gaming, B., 2018. The History and Evolution of Esports. [online] Bountie Gaming. Available at: <https://bountiegaming.medium.com/the-history-and-evolution-of-Esports-8ab6c1cf3257#:~:text=First%20Esports%20Tournament,on%20the%20video%20game%20Spacewar.> [Accessed 14 April 2021].

[11] Kerr, C., 2017. Playerunknown's Battlegrounds pulls in over $11 million in three days. [online] Gamasutra.com. Available at: <https://www.gamasutra.com/view/news/294563/Playerunknowns_Battlegrounds_pulls_in_over_11_million_in_three_days.php> [Accessed 14 April 2021].

[12] Pcgamer.com. 2017. PlayerUnknown's Battlegrounds has sold 1 million copies | PC Gamer. [online] Available at: <https://www.pcgamer.com/playerunknowns-battlegrounds-has-sold-1-million-copies/> [Accessed 14 April 2021].

[13] Bailey, D., 2017. PUBG has now sold over 70 million copies in three years. [online] PCGamesN. Available at: <https://www.pcgamesn.com/playerunknowns-battlegrounds/pubg-sales> [Accessed 14 April 2021].

[14] Business Insider. 2017. The crazy new game that pits 100 people against each other on a deserted island will come to consoles. [online] Available at: <https://www.businessinsider.com/playerunknowns-battlegrounds-coming-to-ps4-and-xbox-2017-5?r=US&IR=T> [Accessed 14 April 2021].

[15] IGN. 2017. PlayerUnknown's Battlegrounds Hits 2 Million Copies Sold - IGN. [online] Available at: <https://www.ign.com/articles/2017/05/02/playerunknowns-battlegrounds-hits-2-million-copies-sold> [Accessed 14 April 2021].

[16] VG247. 2018. PUBG has over 5 million players on Xbox One, players gifted with celebratory jacket - VG247. [online] Available at: <https://www.vg247.com/2018/03/15/pubg-5-million-players-xbox-one-free-

jacket/> [Accessed 14 April 2021].

[17] Grubb, J., 2017. PlayerUnknown's Battlegrounds surpasses $100 million in revenue. [online] VentureBeat. Available at: <https://venturebeat.com/2017/06/23/playerunknowns-battlegrounds-surpasses-100-million-in-revenue/> [Accessed 14 April 2021].

[18] Sims, D., 2017. The Deathmatch Game That's Selling Faster Than Minecraft. [online] The Atlantic. Available at: <https://www.theatlantic.com/entertainment/archive/2017/07/the-apocalyptic-appeal-of-playerunknowns-battlegrounds/533289/> [Accessed 14 April 2021].

[19] Statista. 2021. PlayerUnknown's Battlegrounds number of players on Steam 2021 | Statista. [online] Available at: <https://www.statista.com/statistics/755111/PlayerUnknown's Battlegrounds-number-players/#:~:text=First%20released%20at%20the%20start,January%202018%20at%203.24%20million.> [Accessed 14 April 2021].

[20] Wikipedia contributors. (2021, May 1). PlayerUnknown's Battlegrounds. In Wikipedia, The Free Encyclopedia. Retrieved 08:14, May 2, 2021, from https://en.wikipedia.org/w/index.php?title=PlayerUnknown%27s_Battlegrounds&oldid=1020834773

[21] Wiki, P., 2021. *PLAYERUNKNOWN'S BATTLEGROUNDS Wiki*. [online] Pubg.fandom.com. Available at: <https://pubg.fandom.com/wiki/PLAYERUNKNOWNS_BATTLEGROUNDS_Wiki> [Accessed 2 May 2021].

[22] Team Liquid Extends SAP Analytics Partnership Into League of Legends [Internet]. The Esports Observerhome of essential Esports business news and insights. 2020 [cited 23 April 2021]. Available from: https://Esportsobserver.com/sap-team-liquid-lol-data/

[23] Mindshareworld.com. 2021. *KFC: Colonel KI*. [online] Available at: <https://www.mindshareworld.com/switzerland-de/unsere-arbeiten/kfc-colonel-ki> [Accessed 29 April 2021].

[24] Medium. 2021. *Predicting Player Performance, Using Neural Networks*. [online] Available at: <https://christopherzita.medium.com/predicting-player-performance-using-neural-networks-f6142784b681> [Accessed 29 April 2021].

[25] Silva AL, Pappa GL, Chaimowicz L. Continuous Outcome Prediction of League of Legends Competitive Matches Using Recurrent Neural Networks. InSBC-Proceedings of SBCGames 2018 (pp. 2179-2259).

[26] Edwards, T., 2020. Continuous Outcome Prediction of League of Legends Matches. Undergraduate. University College London.

[27] Unger, C., Huang, B., Gupta, A., Chang, A. and Suduwa Dewage, A., 2021. *PUBG Placement Prediction* - Kaggle *Competition*. [online] PUBG Placement Prediction - Kaggle Competition. Available at: <https://cliveunger.github.io/EE460J_Final_Project_PUBG/> [Accessed 28 April 2021].

[28] Andrea, T., 2016. Introduction to K-means Clustering. [online] Oracle AI & Data Science Blog. Available from <https://blogs.oracle.com/ai-and-datascience/post/introduction-to-k-means-clustering> [Accessed 14 April 2021].

[29] GeeksforGeeks. 2021. ML | K-means++ Algorithm - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/ml-k-means-algorithm/> [Accessed 14 April 2021].

[30] Brownlee, J., 2020. Linear Regression for Machine Learning. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/linear-regression-for-machine-learning/#:~:text=Linear%20regression%20is%20a%20linear,single%20output%20variable%20(y).&text=When%20there%20is%20a%20single,to%20as%20simple%20linear%20regression.> [Accessed 14 April 2021].

[31] Hackernoon.com. 2019. An Introduction to Ridge, Lasso, and Elastic Net Regression | Hacker Noon. [online] Available at: <https://hackernoon.com/an-introduction-to-ridge-lasso-and-elastic-net-regression-cca60b4b934f> [Accessed 14 April 2021].

[32] Medium. 2018. Lasso Versus Ridge Versus Elastic Net. [online] Available at:

<https://medium.com/@vijay.swamy1/lasso-versus-ridge-versus-elastic-net-1d57cfc64b58> [Accessed 14 April 2021].

[33] Wikipedia contributors. (2021, April 9). Gradient boosting. In Wikipedia, The Free Encyclopedia. Retrieved 13:01, April 14, 2021, from https://en.wikipedia.org/w/index.php?title=Gradient_boosting&oldid=1016946224

[34] Brownlee, J., 2021. A Gentle Introduction to XGBoost for Applied Machine Learning. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/> [Accessed 14 April 2021].

[35] Xgboost.readthedocs.io. 2021. XGBoost Documentation — xgboost 1.4.0-SNAPSHOT documentation. [online] Available at: <https://xgboost.readthedocs.io/en/latest/> [Accessed 18 April 2021].

[36] Kaggle.com. 2021. *PUBG Finish Placement Prediction (Kernels Only)* | Kaggle. [online] Available at: <https://www.kaggle.com/c/pubg-finish-placement-prediction/data> [Accessed 20 April 2021].

[37] Julienbeaulieu.gitbook.io. 2021. *Feature Scaling and Normalization*. [online] Available at: <https://julienbeaulieu.gitbook.io/wiki/sciences/machine-learning/linear-regression/feature-scaling-and-normalization> [Accessed 21 April 2021].

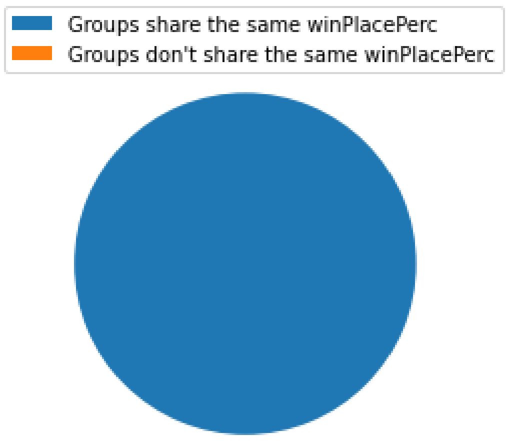
[38] stage?, D., Asthana, S., QUIT--Anony-Mousse, H. and Bernini, N., 2021. *Do Clustering algorithms need feature scaling in the pre-processing stage?*. [online] Data Science Stack Exchange. Available at: <https://datascience.stackexchange.com/questions/22795/do-clustering-algorithms-need-feature-scaling-in-the-pre-processing-stage#:~:text=Yes.,KGs%20before%20calculating%20the%20distance.> [Accessed 21 April 2021].

[39] DeepAI. 2021. *One Hot Encoding*. [online] Available at: <https://deepai.org/machine-learning-glossary-and-terms/one-hot-encoding> [Accessed 21 April 2021].

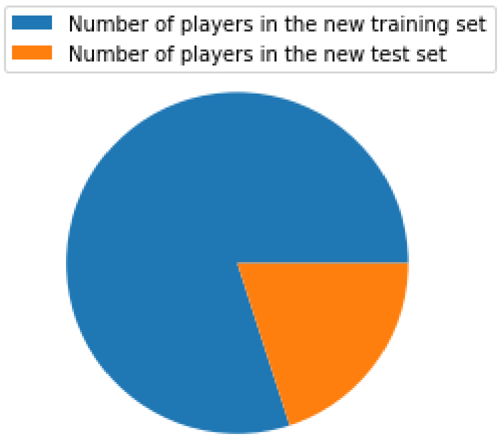
[40] Scikit-learn.org. 2021. *3.1. Cross-validation: evaluating estimator performance — scikit-learn 0.24.1 documentation*. [online] Available at: <https://scikit-learn.org/stable/modules/cross_validation.html> [Accessed 21 April 2021].

Appendix A

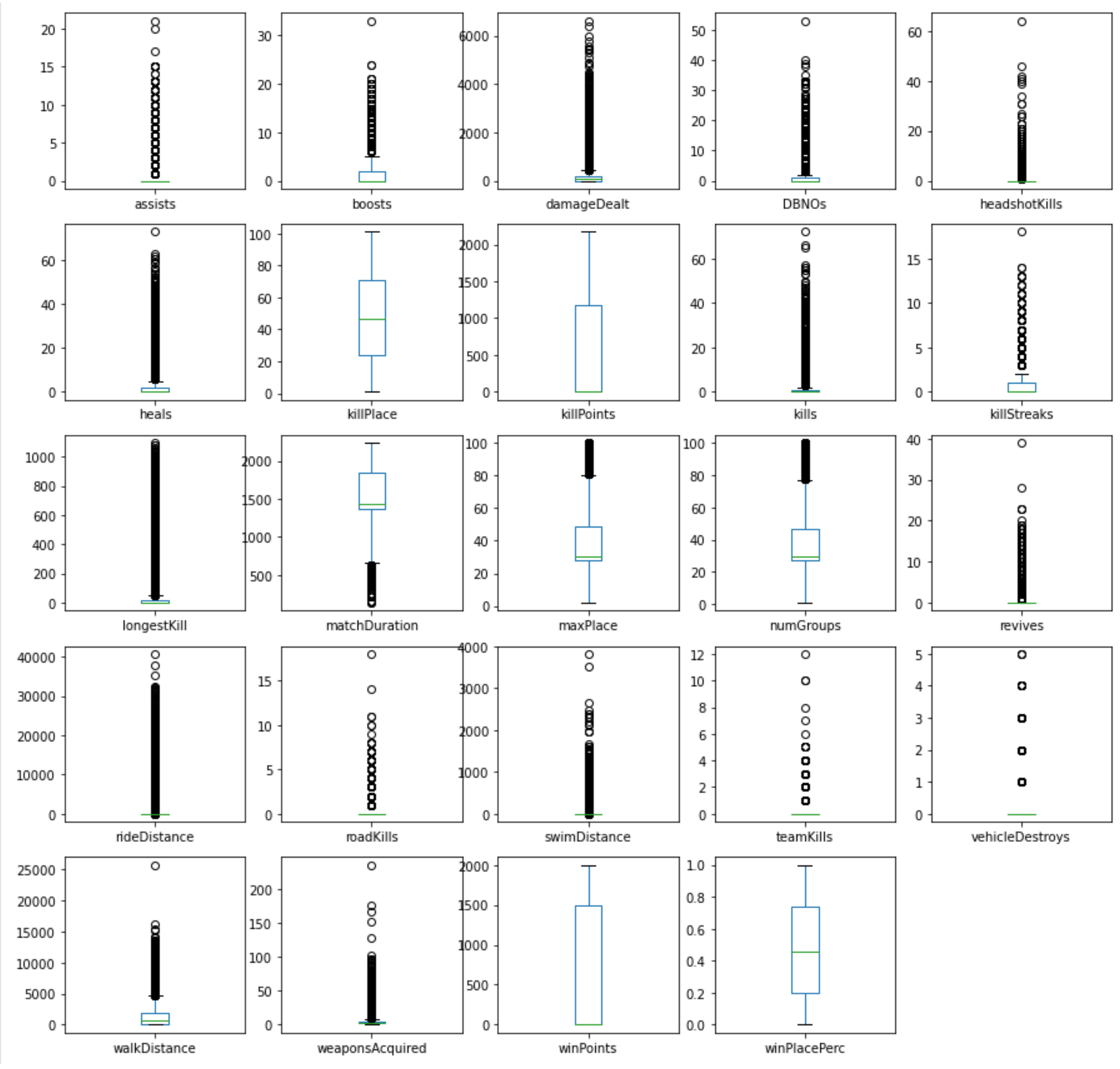
Additional Data Visualisations



Additional Figure 1: Check if each member of a group share the same *winPlacePerc*

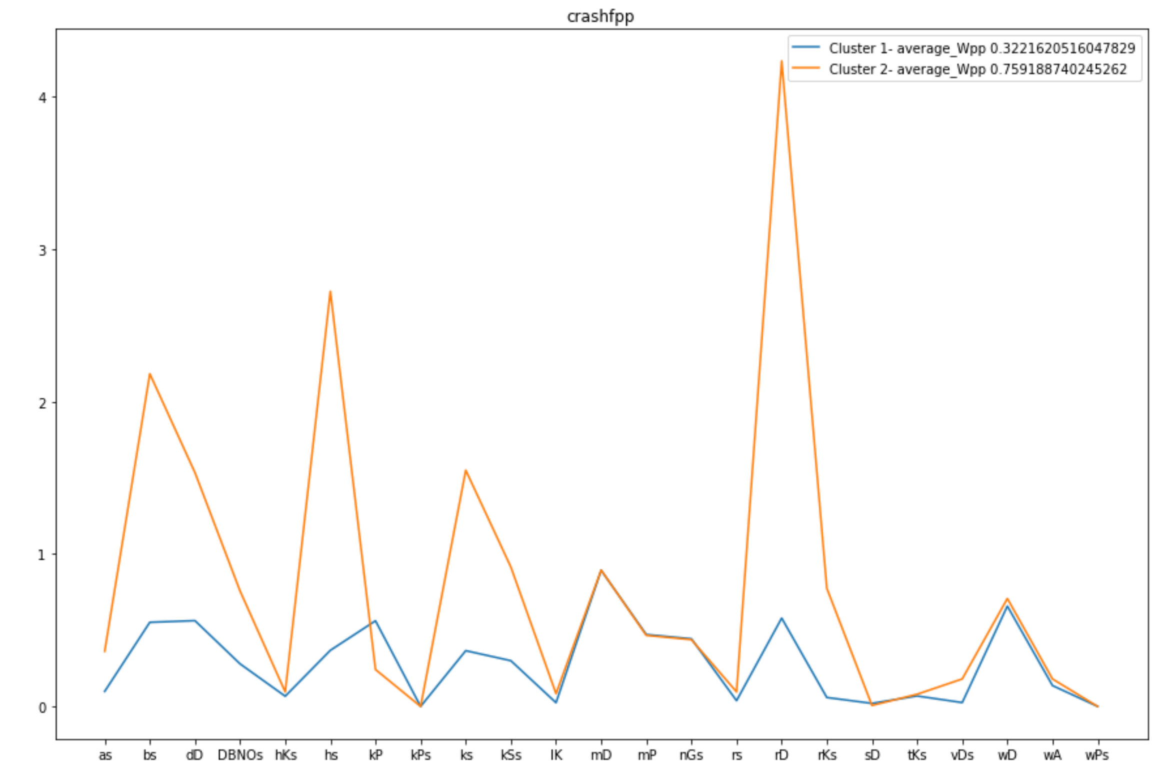


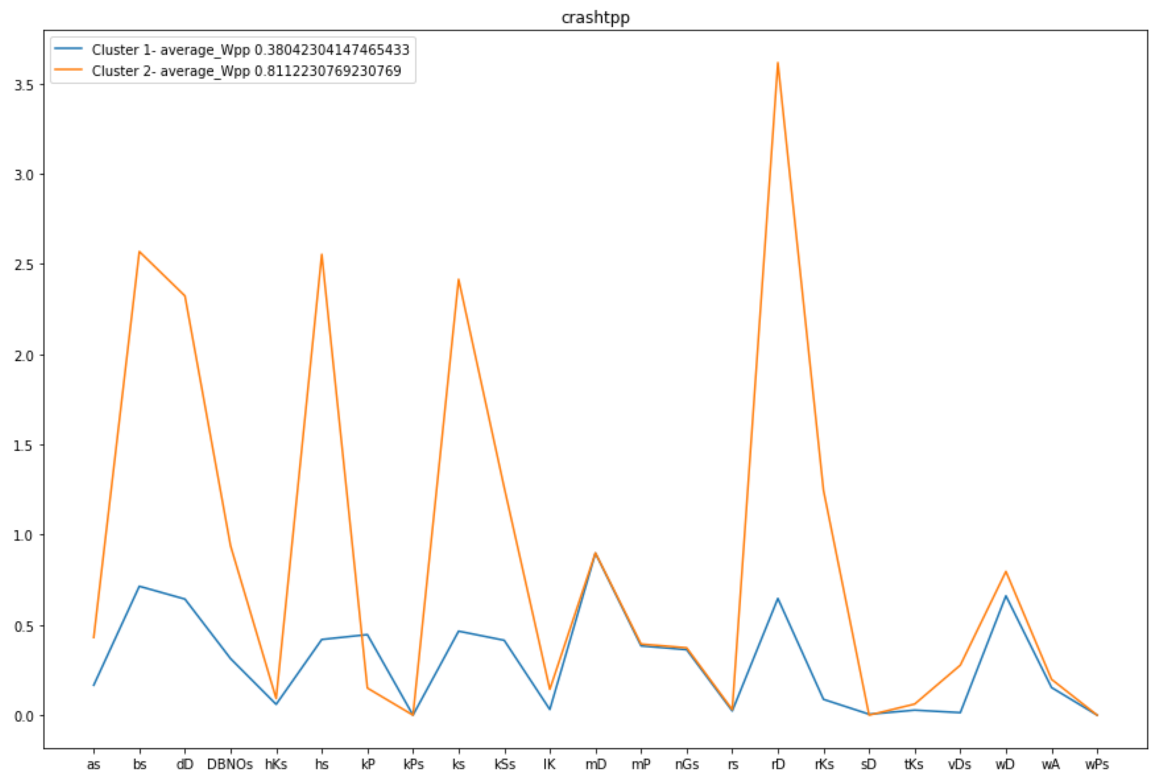
Additional Figure 2: Proportion of data after train-test splitting

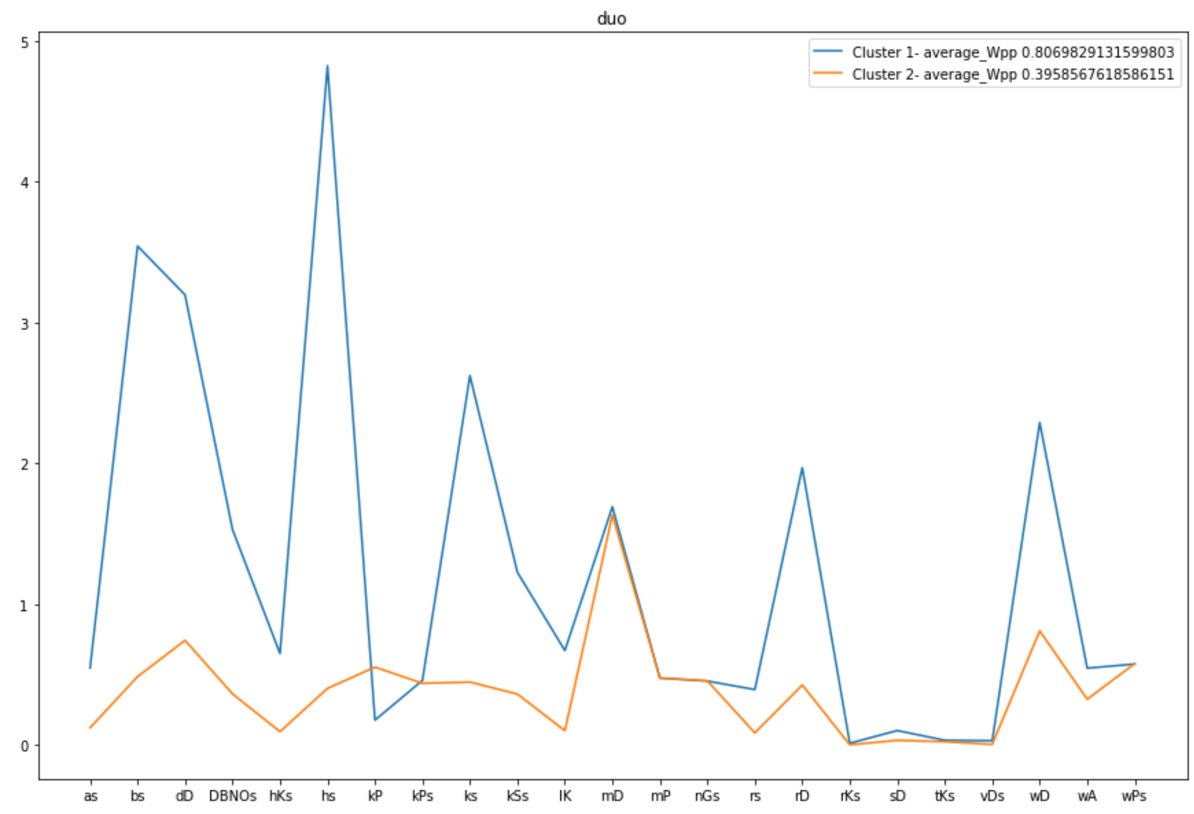


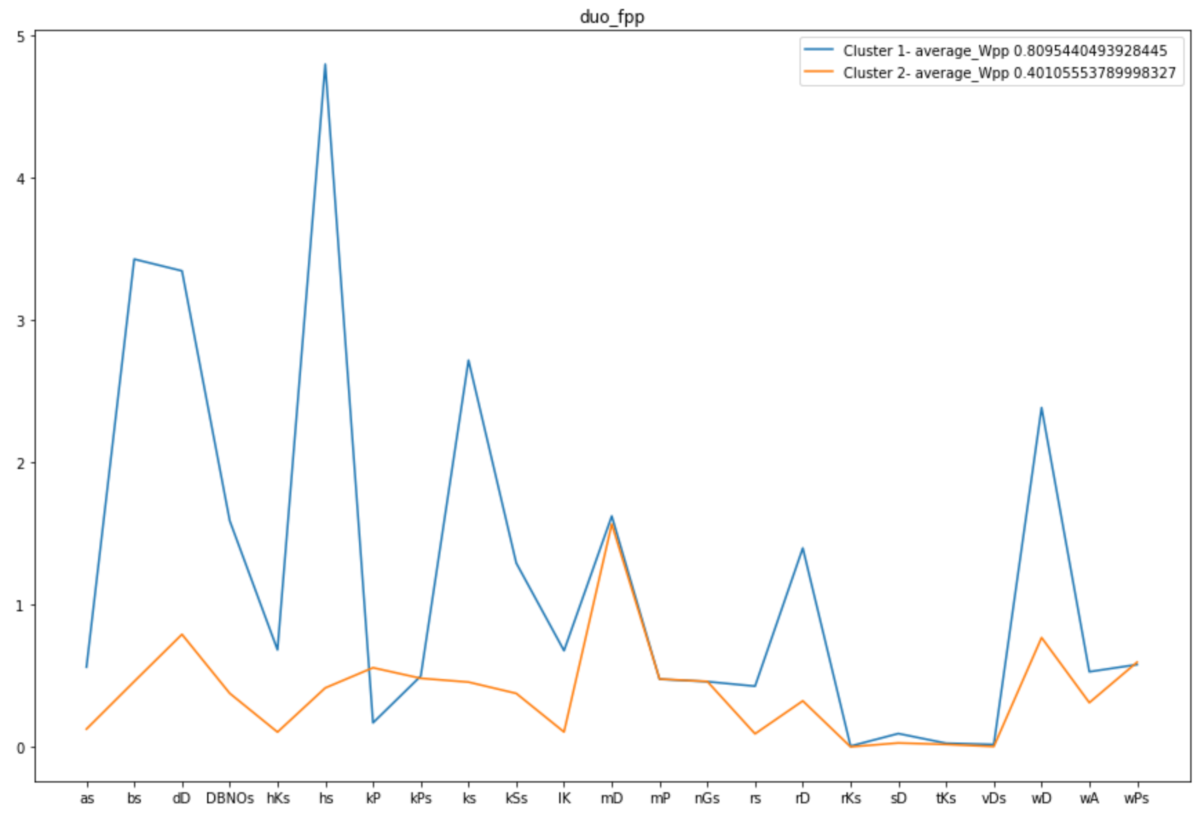
Additional Figure 3: Box plot of each feature in the post-splitting training set

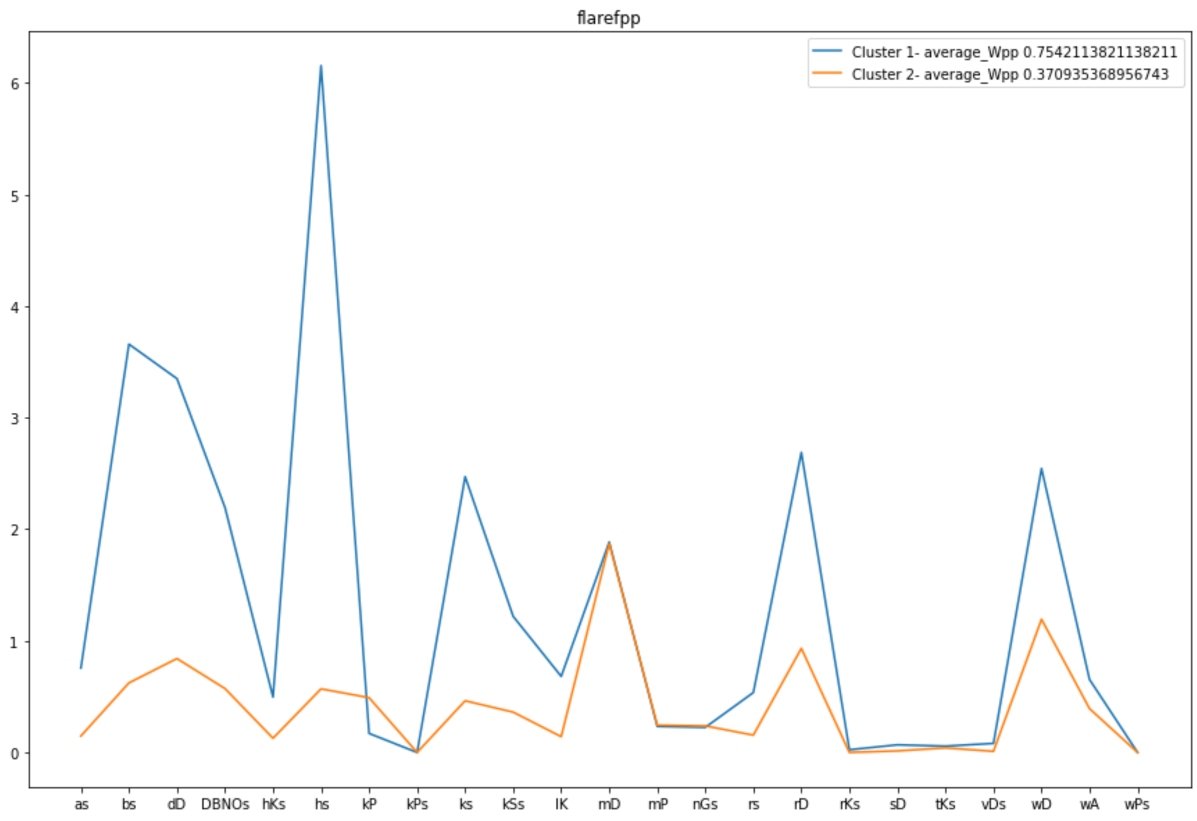
The following 16 figures show the visualisation of all *matchType*s (shown on the following page)

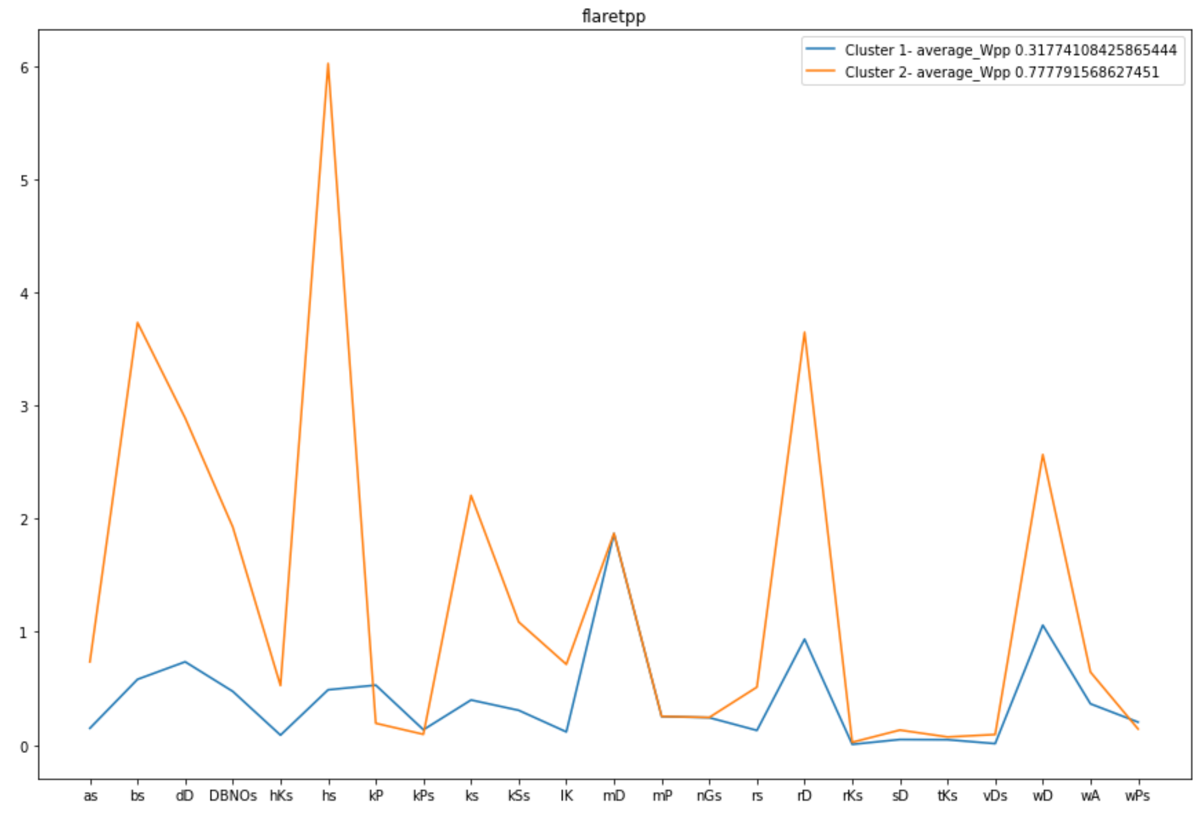

crashfpp
Cluster 1- average_Wpp 0.3221620516047829
Cluster 2- average_Wpp 0.759188740245262
4
3
2
1
0
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

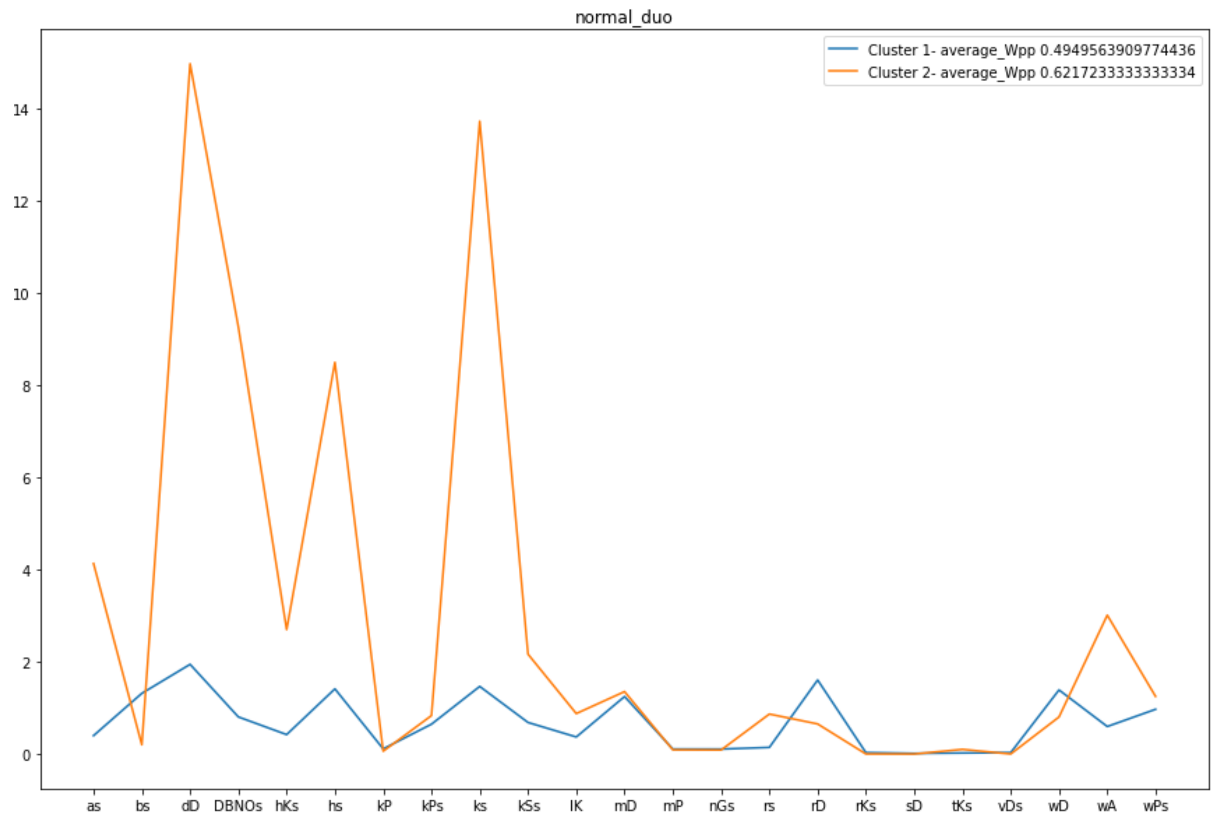

crashtpp
Cluster 1- average_Wpp 0.38042304147465433
Cluster 2- average_Wpp 0.8112230769230769
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

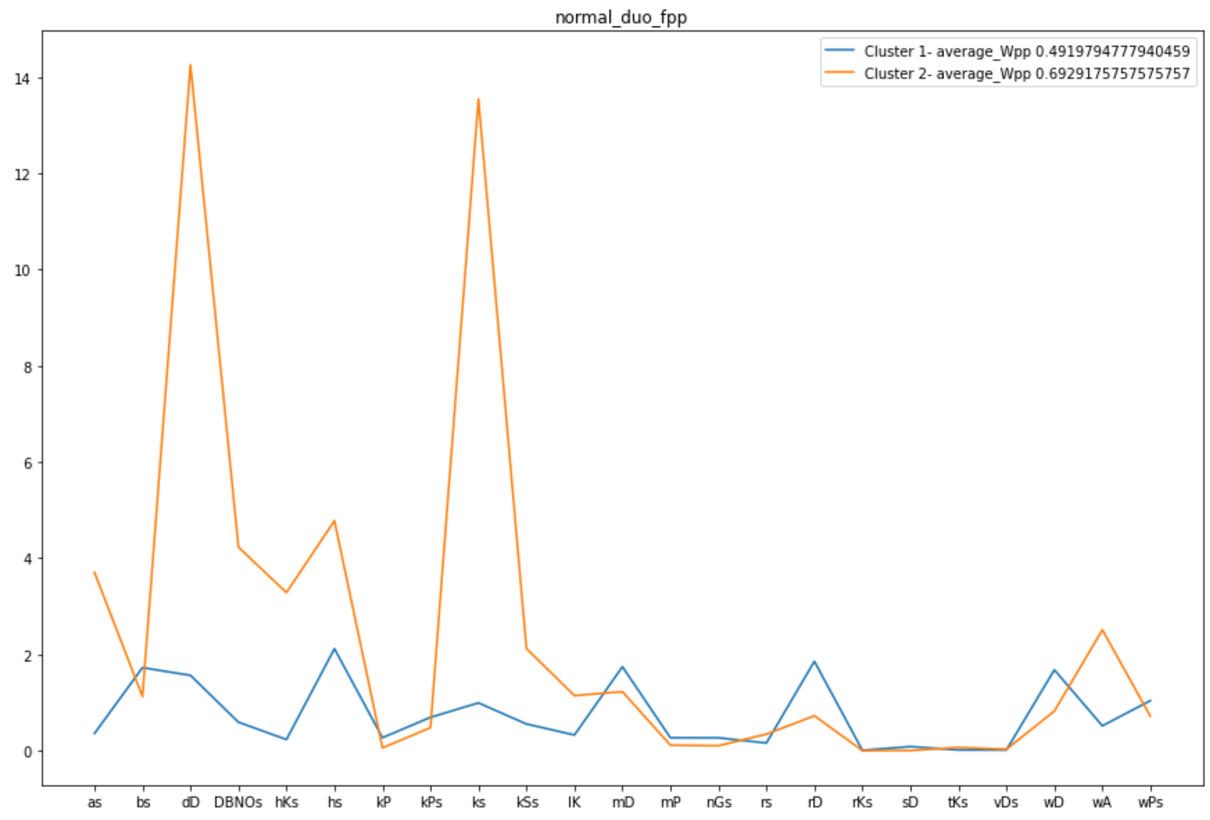

duo
Cluster 1- average_Wpp 0.8069829131599803
Cluster 2- average_Wpp 0.3958567618586151
as bs dD DBNOs hKs hs kP kPs ks kSs lK mD mP nGs rs rD rKs sD tKs vDs wD wA wPs

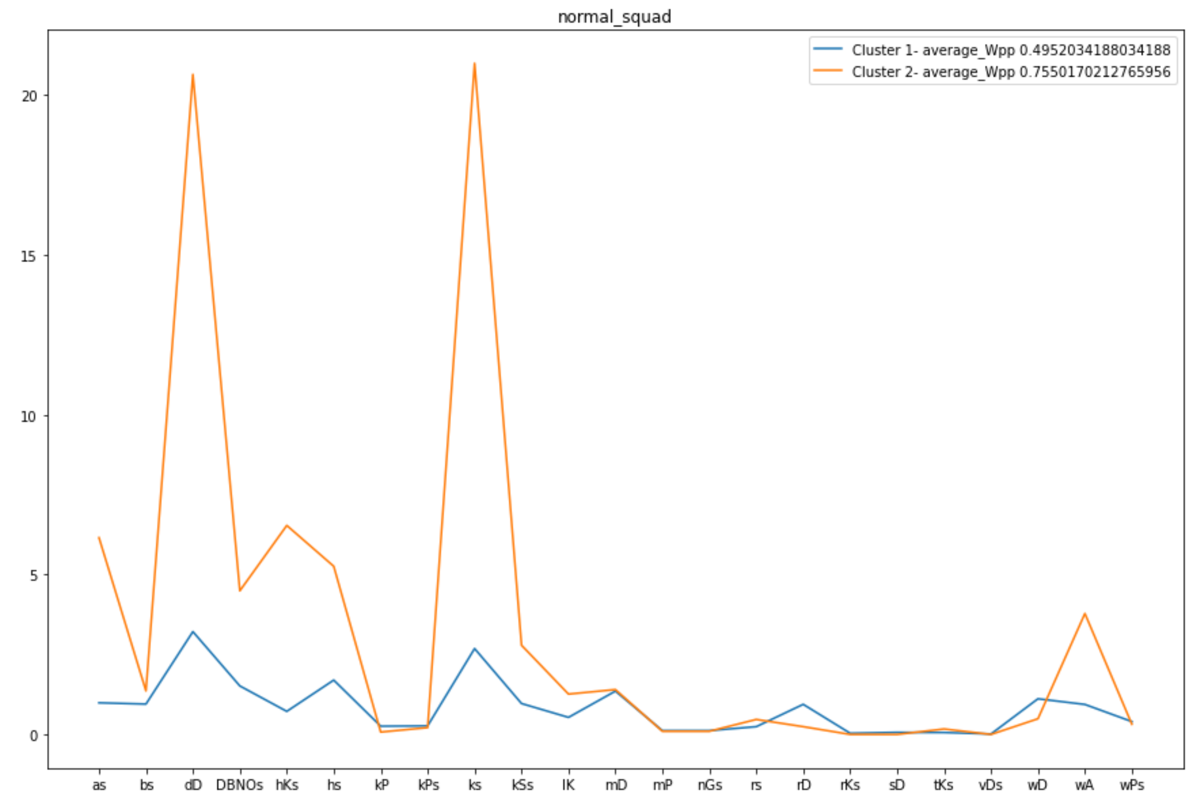

duo_fpp
Cluster 1- average_Wpp 0.8095440493928445
Cluster 2- average_Wpp 0.40105553789998327
as bs dD DBNOs hKs hs kP kPs ks kSs lK mD mP nGs rs rD rKs sD tKs vDs wD wA wPs

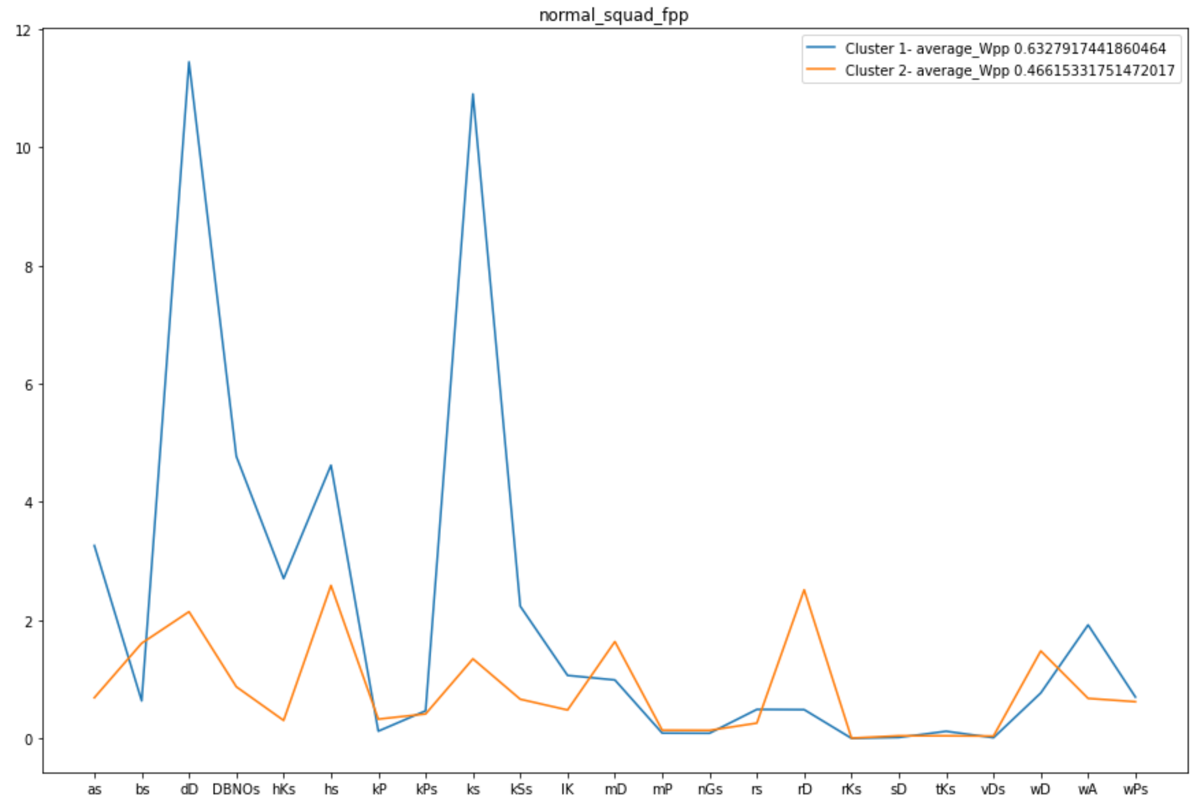

flarefpp
Cluster 1- average_Wpp 0.7542113821138211
Cluster 2- average_Wpp 0.370935368956743
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

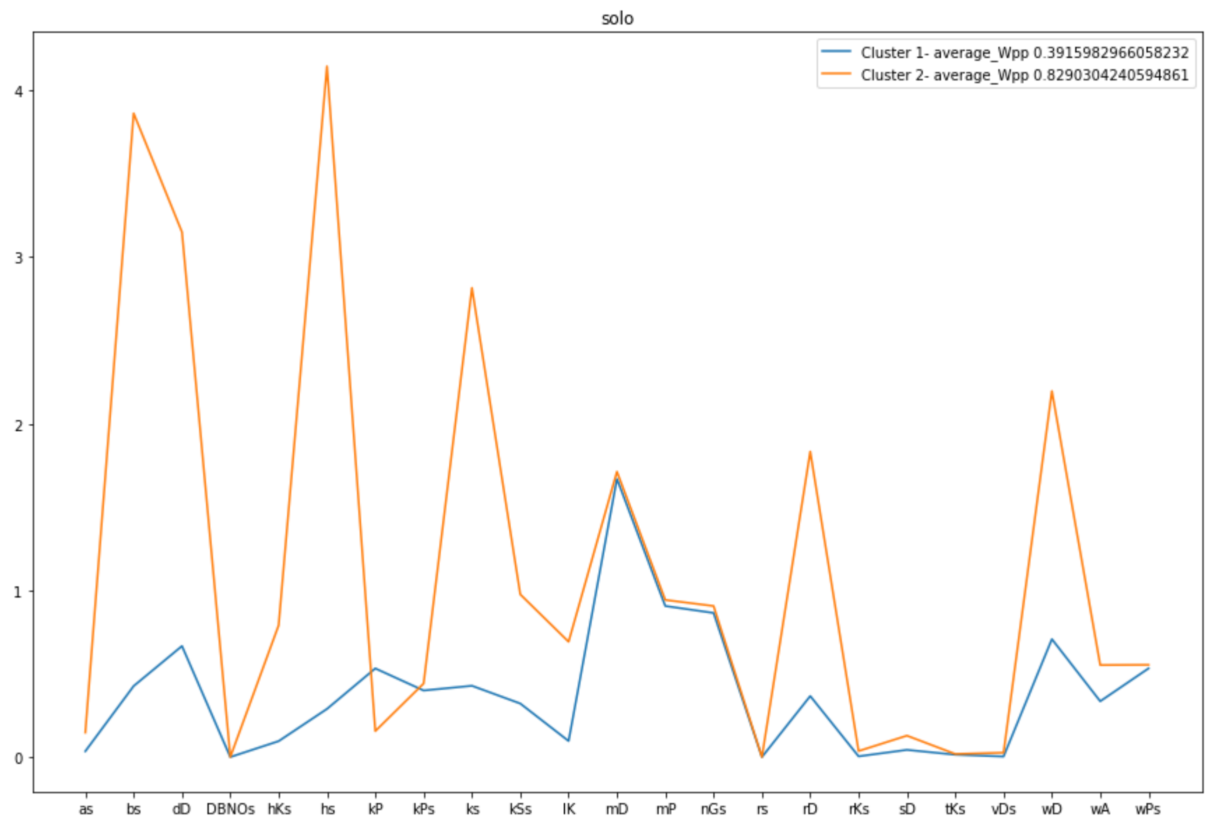

flaretpp
Cluster 1- average_Wpp 0.31774108425865444
Cluster 2- average_Wpp 0.777791568627451
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

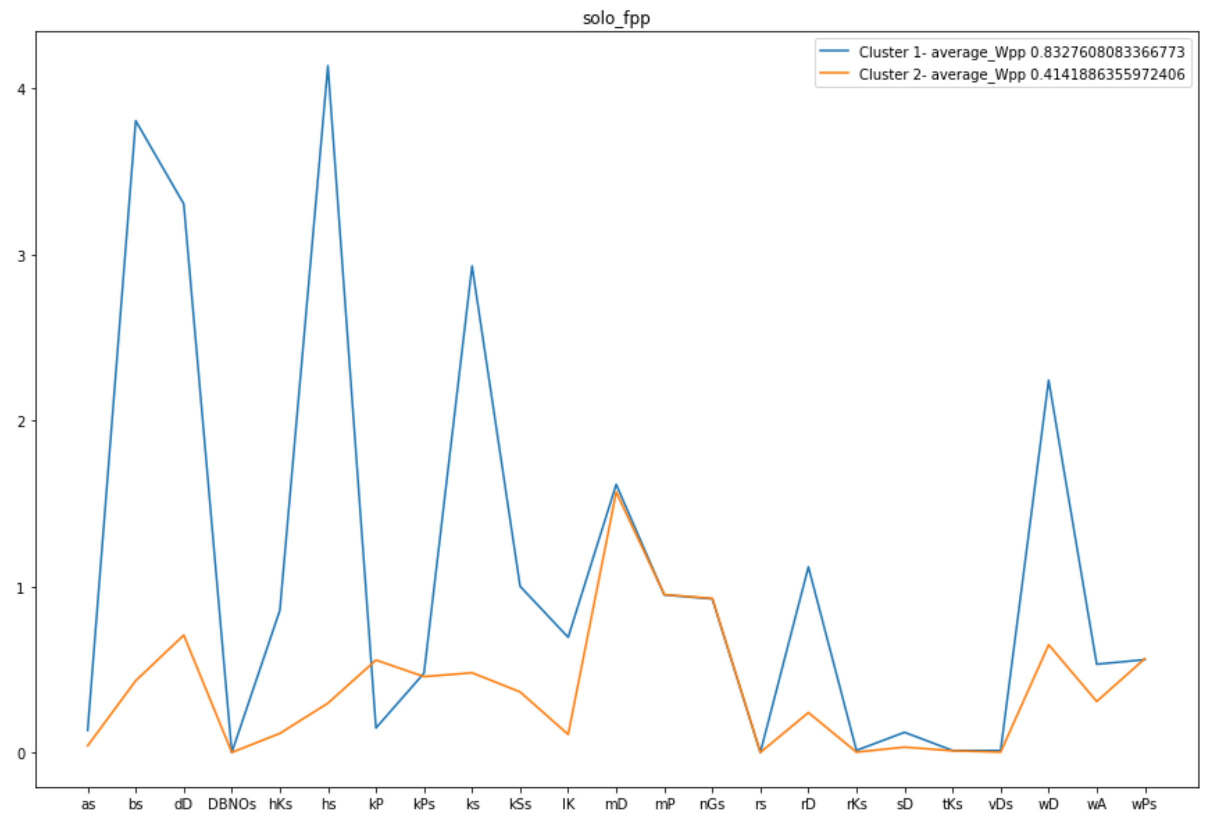

normal_duo
Cluster 1- average_Wpp 0.4949563909774436
Cluster 2- average_Wpp 0.6217233333333334
0
2
4
6
8
10
12
14
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

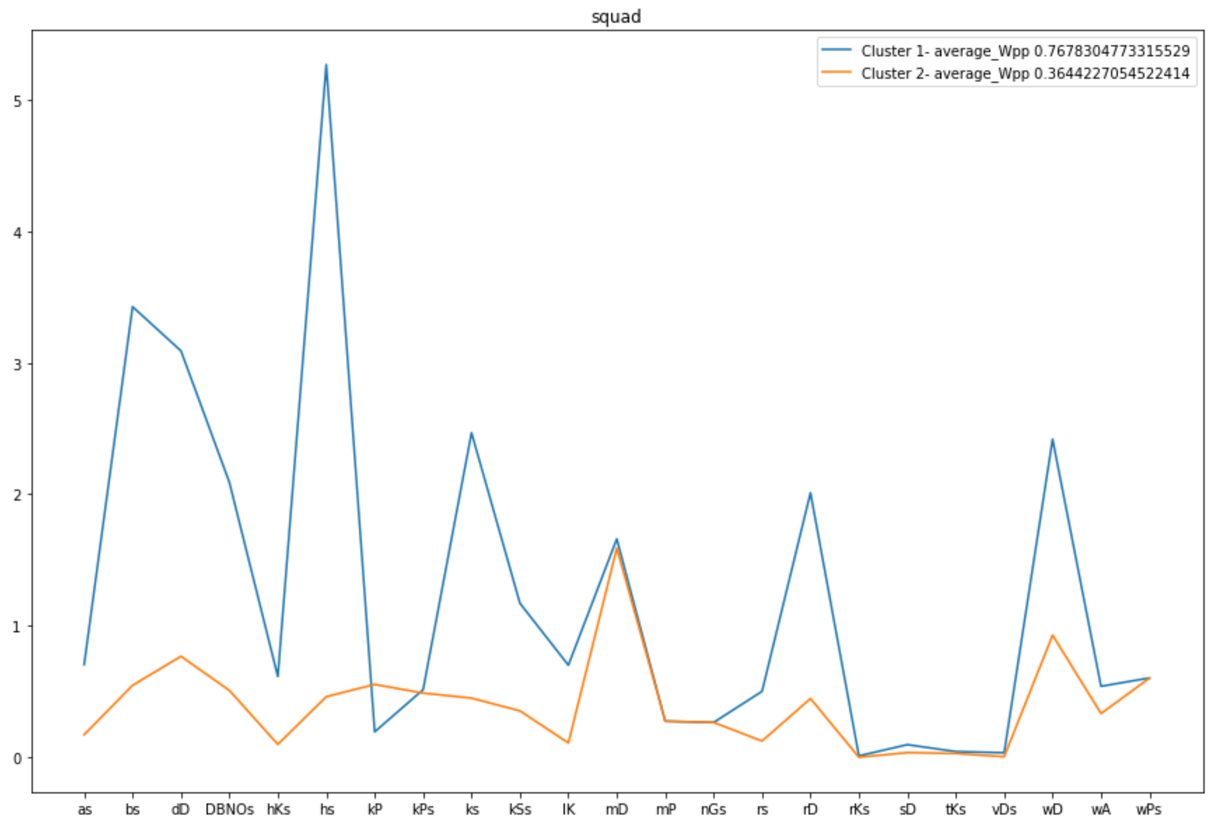

normal_duo_fpp
Cluster 1- average_Wpp 0.4919794777940459
Cluster 2- average_Wpp 0.6929175757575757
0
2
4
6
8
10
12
14
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

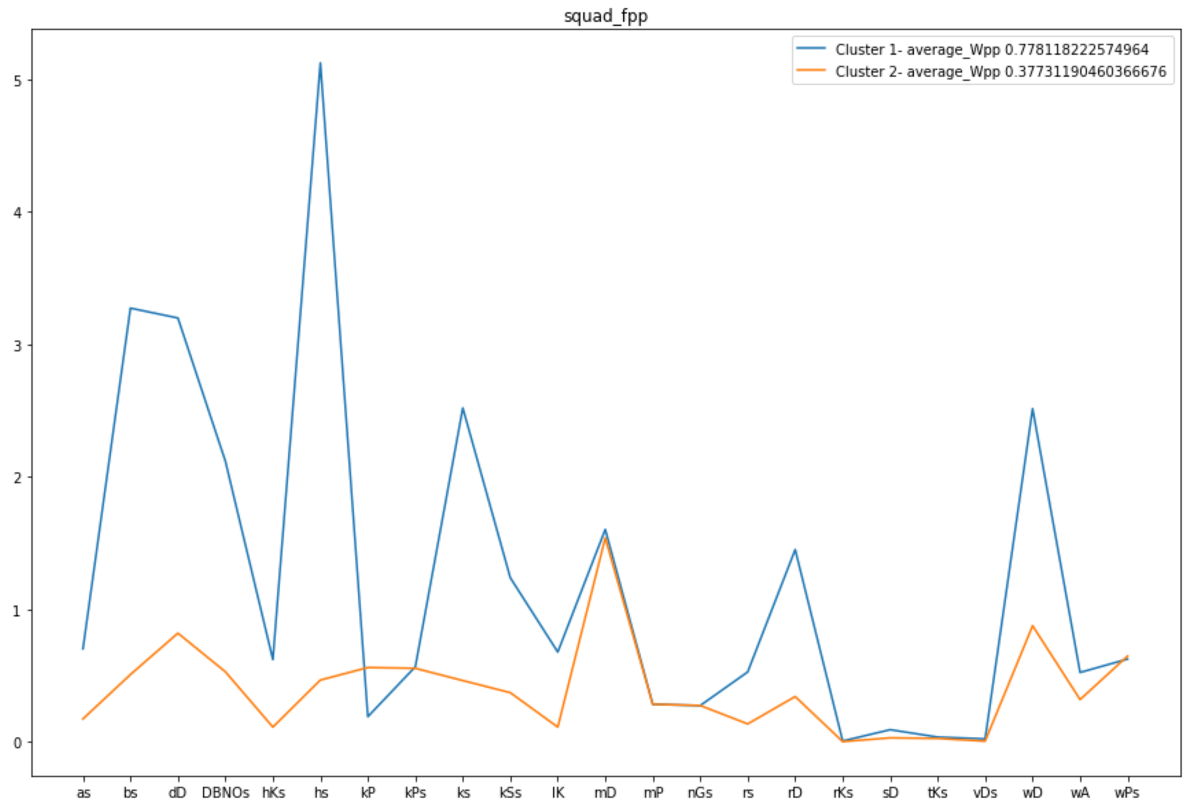

normal_squad
Cluster 1- average_Wpp 0.4952034188034188
Cluster 2- average_Wpp 0.7550170212765956
0
5
10
15
20
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

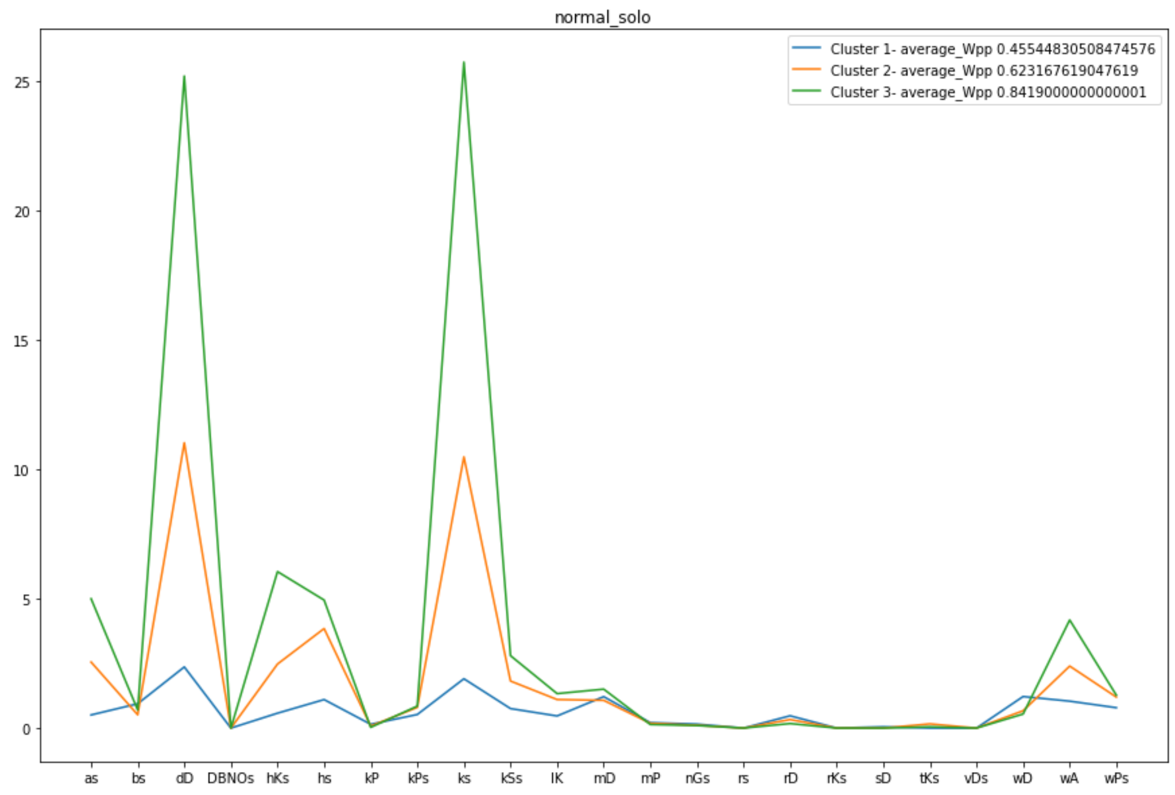

normal_squad_fpp
Cluster 1- average_Wpp 0.6327917441860464
Cluster 2- average_Wpp 0.46615331751472017
0
2
4
6
8
10
12
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

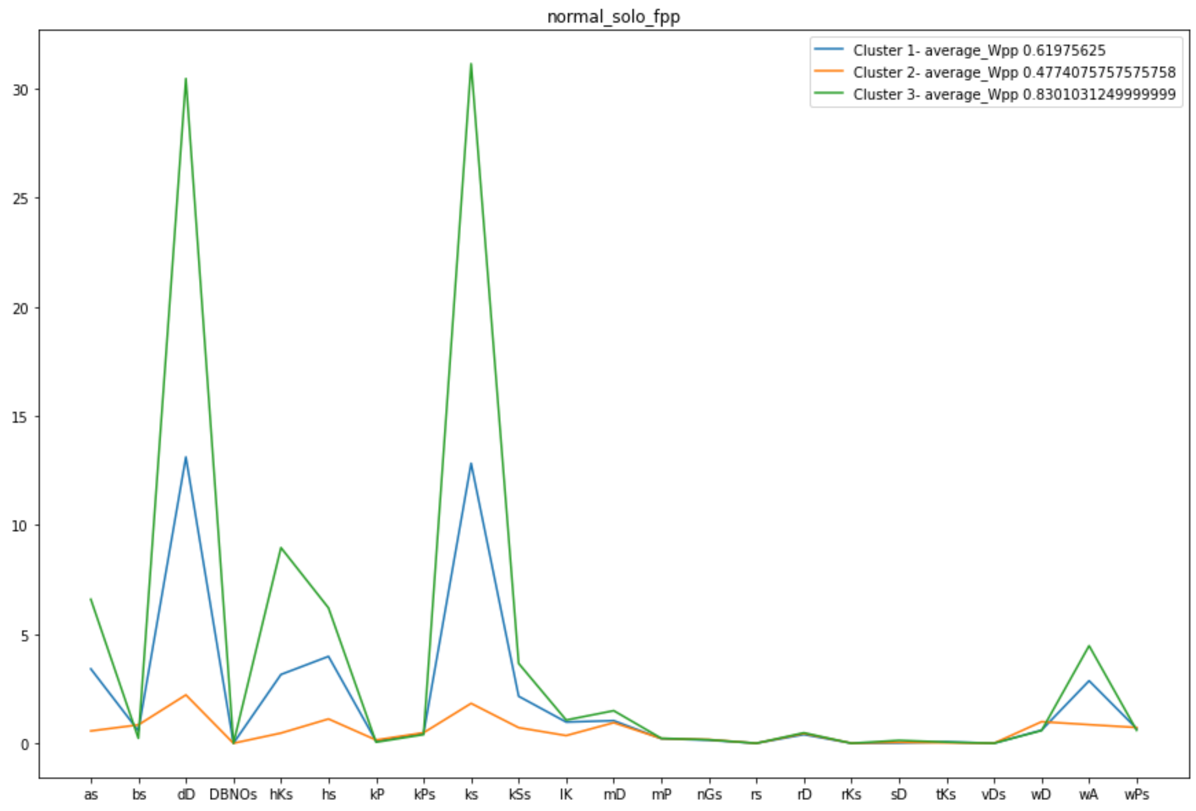

solo
Cluster 1- average_Wpp 0.3915982966058232
Cluster 2- average_Wpp 0.8290304240594861
0
1
2
3
4
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs


solo_fpp
Cluster 1- average_Wpp 0.8327608083366773
Cluster 2- average_Wpp 0.4141886355972406
0
1
2
3
4
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs


squad
Cluster 1- average_Wpp 0.7678304773315529
Cluster 2- average_Wpp 0.3644227054522414
0
1
2
3
4
5
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs


squad_fpp
Cluster 1- average_Wpp 0.778118222574964
Cluster 2- average_Wpp 0.37731190460366676
0
1
2
3
4
5
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs


normal_solo
Cluster 1- average_Wpp 0.45544830508474576
Cluster 2- average_Wpp 0.623167619047619
Cluster 3- average_Wpp 0.8419000000000001
0
5
10
15
20
25
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs


normal_solo_fpp
Cluster 1- average_Wpp 0.61975625
Cluster 2- average_Wpp 0.4774075757575758
Cluster 3- average_Wpp 0.8301031249999999
0
5
10
15
20
25
30
as
bs
dD
DBNOs
hKs
hs
kP
kPs
ks
kSs
lK
mD
mP
nGs
rs
rD
rKs
sD
tKs
vDs
wD
wA
wPs

Appendix B

Example Source Code

This section of the appendix gives an example of the source code created for this research. Significant code segments are presented, including methods for scaling, visualising, and analysing, as well as evidence for all model training. Dependencies have been excluded for brevity. All the code was developed using a Python Jupyter notebook.

[1]

```
train_inds, test_inds = next(GroupShuffleSplit(test_size=.20, n_splits=2, random_state = 7).split(train_data, groups=train_data['groupId']))

train = train_data.iloc[train_inds]
test = train_data.iloc[test_inds]

train = train.reset_index(drop=True)
test = test.reset_index(drop=True)
```

[2]

```
def scale(data set):

    #Maintain the original data
    data set = data set.copy()



    data set['damageDealt'] = data set['damageDealt'] * 1/100
    data set['killPlace'] = data set['killPlace'] * 1/100
    data set['killPoints'] = data set['killPoints'] * 1/1000
    data set['longestKill'] = data set['longestKill'] * 1/100
    data set['matchDuration'] = data set['matchDuration'] * 1/1000
    data set['maxPlace'] = data set['maxPlace'] * 1/100
    data set['numGroups'] = data set['numGroups'] * 1/100
    data set['rideDistance'] = data set['rideDistance'] * 1/1000
    data set['swimDistance'] = data set['swimDistance'] * 1/100
    data set['walkDistance'] = data set['walkDistance'] * 1/1000
    data set['weaponsAcquired'] = data set['weaponsAcquired'] * 1/10
    data set['winPoints'] = data set['winPoints'] * 1/1000
```

```
        return data set
```
[3]
```
    def getAverageWPP(y,n_clusters):

        #Initialize a list for storing the means with n_clusters 0s
        means = [0]*n_clusters

        #Get the mean in each cluster
        for i in range(n_clusters):
            means[i] = y[y["cluster"] == i]["winPlacePerc"].mean()

        return means
```
[4]
```
  #Used to do score VS n_clustet plotting
  scores_1 = []
  Xs_1 = []

  #Used to store best number of clusters, best score and best average WPP
  best_n_clusters = 0
  best_score = 0
  best_average_Wpp = 0

  for i in range(2,11):

      #Assign number of clusters
      kmeans_1.n_clusters = i

      #Train the model
      kmeans_1.fit(cluster_train_X)

      #Get the score
      score = metrics.calinski_harabasz_score(cluster_train_X,kmeans_1.labels_)

      #Add a cluster column to store labels
      cluster_train_X["cluster"] = kmeans_1.labels_

      #Add a cluster column to store labels
      cluster_train_y["cluster"] = kmeans_1.labels_

      #Get the average WPP
      average_Wpp = getAverageWPP(cluster_train_y,i)

      #If the score for current n_cluster is higher, then replace the best values
      if(score>best_score):
            best_score = score
            best_n_clusters = i
            best_average_Wpp = average_Wpp

        #Print the values
        print("n_clusters: ",i,",Score: ",score, ",Average WPP: ",average_Wpp)

        #Append the n_cluster
        Xs_1.append(i)

        #Append the score
        scores_1.append(score)

        #Delete the cluster column
        cluster_train_X = cluster_train_X.drop(["cluster"],axis=1)
        cluster_train_y = cluster_train_y.drop(["cluster"],axis=1)

    #Print the best values
    print("best_n_clusters:    ",best_n_clusters,",    best_score:    ",    best_score,",    best_average_Wpp:
    ",best_average_Wpp)
```
[5]
```
    def distinguish_clusters(kmeans,X_train_temp,y_train_temp,n_clusters,title):

        X_train_temp = X_train_temp.copy()
        y_train_temp = y_train_temp.copy()

        #Set the n_clusters
        kmeans.n_clusters = n_clusters

        #Get the clusters
        kmeans.fit(X_train_temp)

        #Get average wpp
        y_train_temp["cluster"] = kmeans.labels_
        average_Wpp = getAverageWPP(y_train_temp,n_clusters)

        #Get the centroids
        centroids = kmeans.cluster_centers_

        #Extract the centroids
        average_data = pd.DataFrame(centroids)

        xticks = []
        for i in range(23):
            xticks.append(i)

        #Plot the features
        average_data.T.plot.line(figsize = (15,10),title = title,xticks=xticks)
        locs, labels=plt.xticks()

        plt.xticks(locs,['as', 'bs', 'dD', 'DBNOs', 'hKs', 'hs','kP', 'kPs', 'ks', 'kSs', 'lK','mD', 'mP', 'nGs', 'rs','rD',
    'rKs', 'sD', 'tKs','vDs', 'wD', 'wA', 'wPs'])

        #Here are the full names
#           plt.xticks(locs,['assists', 'boosts', 'damageDealt', 'DBNOs', 'headshotKills', 'heals',
#               'killPlace', 'killPoints', 'kills', 'killStreaks', 'longestKill',
#               'matchDuration', 'maxPlace', 'numGroups', 'revives',
#               'rideDistance', 'roadKills', 'swimDistance', 'teamKills',
#               'vehicleDestroys', 'walkDistance', 'weaponsAcquired', 'winPoints'])

        legends = []
        for i in range(n_clusters):
            legends.append("Cluster "+str(i+1)+"- average_Wpp "+str(average_Wpp[i]))
        plt.legend(legends)
        plt.show()
```
[6]
```
    def get_centroids(X_out,X_with,kmeans,n_clusters,y_train_temp):

        #Maintain the original data
        X = X_out.copy()
        X_with = X_with.copy()

        #Train kmeans
        kmeans.n_clusters = n_clusters
        kmeans.fit(X)

        #Get the centroids
        centroids = kmeans.cluster_centers_

        #Get the most common matchtype and dominance
        most_common_matchTypes = []
        dominance = []
        n_members = []
        X_with["cluster"] = kmeans.labels_

        for i in range(n_clusters):
            temp_cluster = X_with[X_with["cluster"] == i]
            most_common_type = temp_cluster['matchType'].mode()[0]
```

```
                most_common_type_count                                                  =
temp_cluster['matchType'].value_counts()[most_common_type]
                most_common_matchTypes.append(most_common_type)
                filtered_temp_cluster = temp_cluster[temp_cluster['matchType'] != most_common_type]
                filtered_common_type = filtered_temp_cluster['matchType'].mode()[0]
                filtered_type_count                                                     =
filtered_temp_cluster['matchType'].value_counts()[filtered_common_type]
                dominance.append((most_common_type_count-
filtered_type_count)/most_common_type_count)
                n_members.append(temp_cluster.shape[0])

            #Extract the centroids
            average_data = pd.DataFrame(centroids)
            average_data.columns = X_out.columns

            #Get the average WPP
            y_train_temp["cluster"] = kmeans.labels_
            average_Wpp = getAverageWPP(y_train_temp,n_clusters)

            #Add columns for most common matchtype, dominance and average WPP
            average_data["most_common"] = most_common_matchTypes
            average_data["dominance"] = dominance
            average_data["averageWpp"] = average_Wpp
            average_data["n_member"] = n_members

            average_data = average_data.sort_values('averageWpp',ascending=False)

            index = []
            for i in range(n_clusters):
                index.append("cluster_"+str(i+1))
            average_data.index = index

            return average_data
```
[7]
```
        plt.figure(figsize = (20,20))
        plt.subplots_adjust(hspace=.5,wspace=.5)

        #Used to allocate position for sub plots
        p = 1

        #Used to store the best number of clusters for each type
        n_clusters_collection = []

        for index in range(len(allTypes)):
```

```
#Get the name
name = names[index]

#Get the dataframe with specific type, without changing the original data
single_type = allTypes[index].copy()

#Extract X and y
single_y = single_type["winPlacePerc"]
single_X = single_type.drop(["winPlacePerc"],axis=1)

#Delete matchType column
single_X_temp = single_X.drop(['Id','groupId','matchId','matchType'],axis=1)

#Convert numpy array to pandas dataframe
single_y_temp = pd.DataFrame({"winPlacePerc": single_y})

scores = []
Xs = []

best_n_clusters = 0
best_score = 0
best_average_Wpp = 0
print(name+":")
for i in range(2,11):

    #Set the number of clusters
    kmeans_mt.n_clusters = i

    #Train the kmeans_mt
    kmeans_mt.fit(single_X_temp)

    #Get the score
    score = metrics.calinski_harabasz_score(single_X_temp,kmeans_mt.labels_)

    #Add a cluster column
    single_X_temp["cluster"] = kmeans_mt.labels_
    single_y_temp["cluster"] = kmeans_mt.labels_

    #Calculate the average WPP
    average_Wpp = getAverageWPP(single_y_temp,i)

    #If current score is larger than the best score, then replace the values
    if(score>best_score):
```

```
                best_score = score
                best_n_clusters = i
                best_average_Wpp = average_Wpp

            print("n_clusters: ",i,",Score: ",score, ",Average WPP: ",average_Wpp)

            #Append X and scores with current value
            Xs.append(i)
            scores.append(score)

            #Delete the cluster column
            single_X_temp = single_X_temp.drop(["cluster"],axis=1)
            single_y_temp = single_y_temp.drop(["cluster"],axis=1)

        print("best_n_clusters: ",best_n_clusters,", best_score: ", best_score,", best_average_Wpp:
",best_average_Wpp,"\n")

        #Append the best n_clusters
        n_clusters_collection.append(best_n_clusters)

        #Plot the line
        ax = plt.subplot(4,4,p)
        plt.xticks(Xs)
        ax.plot(Xs,scores)
        ax.set_title(name)
        p+=1

    plt.savefig("Individual_analysis.png")
    plt.show()
```
[8]
```
    for i in range(len(types_with_two)):
        #Get the name
        name = names_with_two[i]

        #Get the dataframe with specific type, without changing the original data
        single_type = types_with_two[i].copy()

        #Extract X and y
        single_y = single_type["winPlacePerc"]
        single_X = single_type.drop(['Id','groupId','matchId',"winPlacePerc"],axis=1)

        #Delete matchType column
        single_X_temp = single_X.drop(["matchType"],axis=1)
        #Convert numpy array to pandas dataframe
        single_y_temp = pd.DataFrame({"winPlacePerc": single_y})

        distinguish_clusters(kmeans_mt,single_X_temp,single_y_temp,2,name)
```
[9]

```
    def custom_XGBhyperParameterTuning(X_train, y_train, has_groupId, groupId = None):

        X_train = X_train.copy()
        y_train = y_train.copy()
        xgb_model = XGBRegressor()
        param_tuning = {
            'learning_rate': [0.01, 0.1, 0.5],
            'max_depth': [3, 5, 7, 10],
            'min_child_weight': [1, 3, 5, 7],
            'colsample_bytree': [0.5, 0.7],
            'n_estimators' : [100, 200, 500],
            'objective': ['reg:squarederror'],
            'tree_method':['gpu_hist'],
            'gpu_id':[0]
        }

        start_time = timer(None)

        #Extract the validation set

        if has_groupId:

            temp_train = pd.concat([X_train,y_train],axis=1)
            train_inds, validate_inds = next(GroupShuffleSplit(test_size=.20, n_splits=2,
random_state = 7).split(temp_train, groups=groupId))

            train = temp_train.iloc[train_inds]
            validate = temp_train.iloc[validate_inds]
            train = train.reset_index(drop=True)
            validate = validate.reset_index(drop=True)
            y_train = train['winPlacePerc']
            X_train = train.drop(['winPlacePerc'],axis=1)
            y_validate = validate['winPlacePerc']
            X_validate = validate.drop(['winPlacePerc'],axis=1)

        else:

            X_train,X_validate,y_train,y_validate = train_test_split(X_train, y_train, test_size = 0.2,
random_state = 7)
```

```
        X_train = X_train.reset_index(drop=True)
        X_validate = X_validate.reset_index(drop=True)
        y_train = y_train.reset_index(drop=True)
        y_validate = y_validate.reset_index(drop=True)

    #Do the hyperparameter tuning
    best_feature = None
    best_mae = 100

    for i in range(10):

        random_feature = random_sample(param_tuning)
        xgb_model.set_params(**random_feature)
        xgb_model.fit(X_train,y_train)

        preds = xgb_model.predict(X_validate)
        mae = mean_absolute_error(preds,y_validate)

        print("Fit ",i+1," features: ", random_feature," MAE:", mae)

        if mae < best_mae:
            best_mae = mae
            best_feature = random_feature


    timer(start_time)

    return best_feature
```
[10]
```
def random_sample(dictionary):

    new_dict = dictionary.copy()
    for key in new_dict:
        new_dict[key] = random.choice(new_dict[key])
    return new_dict
```
[11]
```
num_test = scaled_test.shape[0]
err_xgb = 0
err_sgd = 0
err_svr = 0

#XGB
for index in range(len(train_xgb_allTypes)):
    temp_train = train_xgb_allTypes[index]
```

```
        temp_test = test_xgb_allTypes[index]

        temp_train = temp_train.drop(['Id','matchId','matchType'],axis=1)
        temp_train = temp_train.groupby("groupId").mean()
        temp_train_y = temp_train["winPlacePerc"]
        temp_train_X = temp_train.drop(["winPlacePerc"],axis=1)

        temp_test = temp_test.drop(['Id','matchId','matchType'],axis=1)
        temp_test = temp_test.groupby("groupId").transform('mean')
        temp_test_y = temp_test["winPlacePerc"]
        temp_test_X = temp_test.drop(["winPlacePerc"],axis=1)

        temp_xgb = XGBRegressor(objective="reg:squarederror")
        temp_xgb.fit(temp_train_X,temp_train_y)
        err_xgb                                                          +=
mean_absolute_error(temp_xgb.predict(temp_test_X),temp_test_y)*temp_test.shape[0]

    #SGD
    temp_train = train_sgd_allTypes[0]
    temp_test = test_sgd_allTypes[0]

    temp_train = temp_train.drop(['Id','matchId','matchType'],axis=1)
    temp_train = temp_train.groupby("groupId").mean()
    temp_train_y = temp_train["winPlacePerc"]
    temp_train_X = temp_train.drop(["winPlacePerc"],axis=1)

    temp_test = temp_test.drop(['Id','matchId','matchType'],axis=1)
    temp_test = temp_test.groupby("groupId").transform('mean')
    temp_test_y = temp_test["winPlacePerc"]
    temp_test_X = temp_test.drop(["winPlacePerc"],axis=1)

    temp_sgd = SGDRegressor()
    temp_sgd.fit(temp_train_X,temp_train_y)
    err_sgd                                                              +=
mean_absolute_error(temp_sgd.predict(temp_test_X),temp_test_y)*temp_test.shape[0]

    #SVR
    for index in range(len(train_svr_allTypes)):
        temp_train = train_svr_allTypes[index]
        temp_test = test_svr_allTypes[index]

        temp_train = temp_train.drop(['Id','matchId','matchType'],axis=1)
        temp_train = temp_train.groupby("groupId").mean()
        temp_train_y = temp_train["winPlacePerc"]
```

```
            temp_train_X = temp_train.drop(["winPlacePerc"],axis=1)

            temp_test = temp_test.drop(['Id','matchId','matchType'],axis=1)
            temp_test = temp_test.groupby("groupId").transform('mean')
            temp_test_y = temp_test["winPlacePerc"]
            temp_test_X = temp_test.drop(["winPlacePerc"],axis=1)

            temp_svr = LinearSVR(loss = 'epsilon_insensitive') #L1 loss
            temp_svr.fit(temp_train_X,temp_train_y)
            err_svr                                                                    +=
mean_absolute_error(temp_svr.predict(temp_test_X),temp_test_y)*temp_test.shape[0]
    (err_xgb+err_sgd+err_svr)/num_test
```
[12]
```
    def approach_3(train,test,n_clusters,kmeans,first_kt):
        train = train.copy()
        test = test.copy()

        if first_kt:
            train_d = train.drop(['Id','matchId',"matchType"],axis=1)
            train_d = train_d.groupby("groupId").transform('mean')
            train_y = train_d["winPlacePerc"]
            train_X = train_d.drop(["winPlacePerc"],axis=1)

            test_d = test.drop(['Id','matchId',"matchType"],axis=1)
            test_d = test_d.groupby("groupId").transform('mean')
            test_y = test_d["winPlacePerc"]
            test_X = test_d.drop(["winPlacePerc"],axis=1)
        else:
            train_d = train.drop(['Id','matchId','groupId',"matchType"],axis=1)
            train_y = train_d["winPlacePerc"]
            train_X = train_d.drop(["winPlacePerc"],axis=1)

            test_d = test.drop(['Id','matchId','groupId',"matchType"],axis=1)
            test_y = test_d["winPlacePerc"]
            test_X = test_d.drop(["winPlacePerc"],axis=1)

        kmeans.n_clusters = n_clusters
        kmeans.fit(train_X)

        train["cluster"] = kmeans.labels_
        test["cluster"] = kmeans.predict(test_X)

        train_n = train.drop(['Id','matchId','matchType'],axis=1)
        test_n = test.drop(['Id','matchId','matchType'],axis=1)
        train_n["matchType"] = train["matchType"]
        test_n["matchType"] = test["matchType"]
        train_n = pd.get_dummies(data = train_n,columns=["matchType"])
        test_n = pd.get_dummies(data = test_n,columns=["matchType"])

        num_test = test.shape[0]
        sum_error = 0
        temp_set = np.array([])
        for i in range(n_clusters):

            temp_train_n = train_n[train_n["cluster"] == i]
            temp_test_n = test_n[test_n["cluster"] == i]

            temp_train_n.drop(["cluster"],axis=1)
            temp_test_n.drop(["cluster"],axis=1)

            temp_train_n = temp_train_n.groupby("groupId").mean()
            temp_train_n_y = temp_train_n["winPlacePerc"]
            temp_train_n_X = temp_train_n.drop(["winPlacePerc"],axis=1)

            temp_test_n = temp_test_n.groupby("groupId").transform('mean')
            temp_test_n_y = temp_test_n["winPlacePerc"]
            temp_test_n_X = temp_test_n.drop(["winPlacePerc"],axis=1)

            temp_parameter =
custom_XGBhyperParameterTuning(temp_train_n_X,temp_train_n_y,False)
            temp_xgb = XGBRegressor().set_params(**temp_parameter)
            temp_xgb.fit(temp_train_n_X,temp_train_n_y)
            sum_error +=
mean_absolute_error(temp_xgb.predict(temp_test_n_X),temp_test_n_y)*temp_test_n.shape[0]
            temp_set = np.append(temp_set,np.absolute(temp_xgb.predict(temp_test_n_X)-
temp_test_n_y))

        return sum_error/num_test,temp_set
```
[13]
```
    def custom_SGDhyperParameterTuning(X_train, y_train, has_groupId, groupId = None):

        X_train = X_train.copy()
        y_train = y_train.copy()
        sgd_model = SGDRegressor()
        param_tuning = {
            'loss':['squared_loss'],
            'penalty' : ['elasticnet'],
            'alpha' : [0.0001, 0.001, 0.01],
            'l1_ratio' : [0.15,0.35,0.55],
            'max_iter':[1000,2000,5000],
            'eta0' : [0.1,0.01,0.001]
        }

        start_time = timer(None)

        #Extract the validation set
        print("Start Splitting...")
        if has_groupId:

            temp_train = pd.concat([X_train,y_train],axis=1)
            train_inds, validate_inds = next(GroupShuffleSplit(test_size=.20, n_splits=2,
    random_state = 7).split(temp_train, groups=groupId))

            train = temp_train.iloc[train_inds]
            validate = temp_train.iloc[validate_inds]
            train = train.reset_index(drop=True)
            validate = validate.reset_index(drop=True)
            y_train = train['winPlacePerc']
            X_train = train.drop(['winPlacePerc'],axis=1)
            y_validate = validate['winPlacePerc']
            X_validate = validate.drop(['winPlacePerc'],axis=1)

        else:

            X_train,X_validate,y_train,y_validate = train_test_split(X_train, y_train, test_size = 0.2,
    random_state = 7)
            X_train = X_train.reset_index(drop=True)
            X_validate = X_validate.reset_index(drop=True)
            y_train = y_train.reset_index(drop=True)
            y_validate = y_validate.reset_index(drop=True)

        #Do the hyperparameter tuning
        best_feature = None
        best_mae = 100

        print("Start validation...")
        for i in range(10):

            random_feature = random_sample(param_tuning)
            sgd_model.set_params(**random_feature)
            print("Start training...",random_feature)
            sgd_model.fit(X_train,y_train)

            preds = sgd_model.predict(X_validate)
            mae = mean_absolute_error(preds,y_validate)

            print(" MAE:", mae)

            if mae < best_mae:
                best_mae = mae
                best_feature = random_feature


        timer(start_time)

        return best_feature
```