27 November 2018

Alexandre ALLANI
Adrien CLOTTEAU
Axel DURAND
Alexis MICHEL

Projet de fouille de données

PUBG - Finish Placement Prediction

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Contents

Vous pouvez retrouver une version plus détaillé de l'analyse des résultats (avec le code) en cliquant sur le lien suivant : `https://github.com/Syndorik/PUBG-Kaggle-Challenge`.

Vous pouvez télécharger :

- La version notebook (fait sous jupyter notebook) : Challenge PUBG.ipynb
- La version HTML : Challenge PUBG.html

1 Contexte

PUBG (PlayerUnknown's BattleGrounds) est un jeu vidéo en ligne de type "Battle Royale" actuellement très actif avec ses millions de joueurs mensuels. Comptabilisant plus de 50 millions de ventes à l'unité, ce jeu vidéo se classe comme le cinquième jeu vidéo le plus vendu de tous les temps.

Une partie du jeu se déroule de la manière suivante : 100 joueurs sans équipements ni armes sont largués à différents endroits d'une grande carte par avion. Chaque joueur doit récupérer des armes et munitions, tuer ses adversaires et rester le seul survivant sur le terrain. La zone de jeu se réduit au fur et à mesure du temps afin de forcer les derniers joueurs encore en vie à s'affronter. Les parties peuvent se dérouler en solo (chacun pour soi) ou par équipes de 2, 3 ou 4 membres.

PUBG a récemment rendu public des données de plus de 65 000 parties.

2 Compréhension du problème

Le but de ce challenge Kaggle (`https://www.kaggle.com/c/pubg-finish-placement-prediction`) est de trouver un modèle prédictif précis pour déterminer le classement final des différents joueurs d'une partie, en fonction des données collectées lors de la partie et de certaines statistiques générales de chaque joueur. Nous nous sommes intéressés à ce problème et souhaitons y apporter notre contribution.

Le but de ce problème est de prédire les valeurs $winPlacePerc_{pred,i}$ de tous les joueurs du fichier test, comprises entre 0 et 1, qui représentent les indices de classements des joueur dans une des partie tels que, pour un joueur i :

$$winPlacePerc_{pred,i} = \frac{(Nombredejoueurs - Classementdujoueuri)}{(Nombredejoueurs - 1)}$$

Ainsi, un joueur classé premier à la fin de la partie obtient un score de 1, un joueur classé dernier obtient un score de 0.

À partir de ces $winPlacePerc_{pred,i}$, des $winPlacePerc_{real,i}$ correspondant aux valeurs réelles de l'indice de classement et au nombre n de joueurs sur toutes les parties, la "Mean Absolute Error" (MAE) est la mesure utilisée par Kaggle pour évaluer la performance de notre modèle. Plus la MAE est faible, plus le modèle est précis :

$$MAE = \frac{\sum_{i=0}^{n} |winPlacePerc_{pred,i} - winPlacePerc_{real,i}|}{n}$$

Actuellement, le meilleur score obtenu par un participant au challenge est de 1,67

3 Compréhension des données

3.1 Collecte des données initiales

Le challenge comprend les données suivantes :

- **train_V2.csv** (données avec les indices réels de classements / 29 variables)
- **test_V2.csv** (données de différentes parties sans les indices réels de classements / 28 variables)
- **sample_submission_V2.csv** (exemple de données au format désiré par l'équipe PUBG pour évaluer les résultats / 2 variables - ID et indices de classement)

Ces données sont disponibles en téléchargement sur la page du challenge kaggle après inscription au challenge.

3.2 Description des données

train_V2.csv et test_V2.csv contiennent de nombreuses variables pour chaque joueur, comme le nombre de joueurs tués ou la distance parcourue en kilomètres (Voir Annexe 1 pour la liste complète des variables). train_V2 contient, en plus des mêmes variable que test_V2, l'indice de classement réel noté "winPlacePerc". C'est notre target.

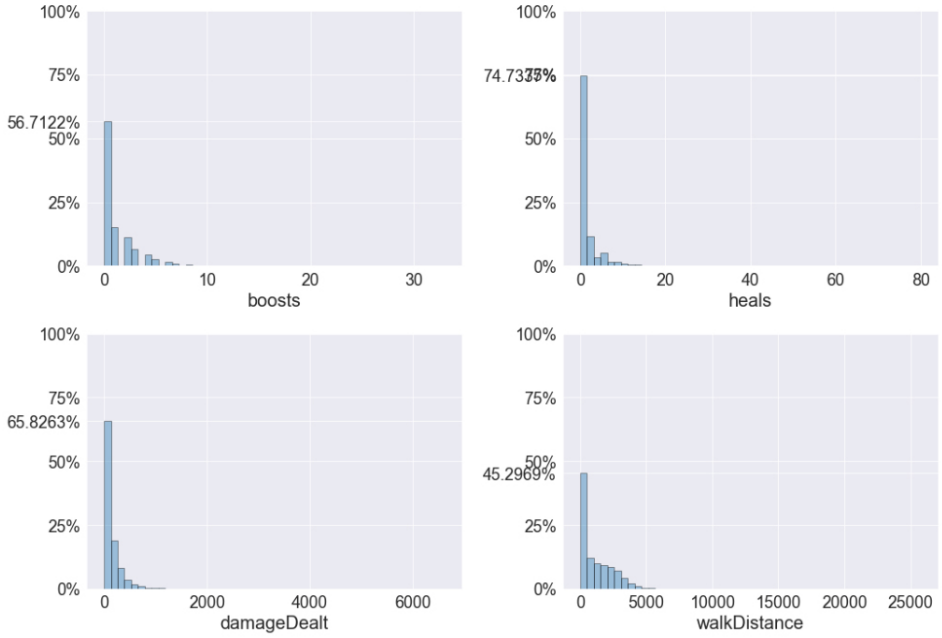

Figure 1: Répartition de quatre variables de test_V2.csv et train_V2.csv

Parmi les variables communes à train_V2 et test_V2, l'évolution des variables boosts et heals indique que les joueurs utilisent très peu d'éléments (boosts et soins dans ce cas) dans le jeu. De plus, la plupart des joueurs n'infligent que peu de dégâts ("damageDealt"). La distance parcourue à pied indique également qu'une grande majorité des joueurs ne marchent presque pas. Le jeu obligeant les joueurs à bouger régulièrement, on peut en déduire que la plupart des joueurs meurent dès les premières minutes du jeu.

Les variables rankPoints, killPoints et winPoints ont posé quelques problèmes à la compréhension. killPoints représente le classement général d'un joueur suivant le nombre de joueurs qu'il a tué au cours des parties précédentes. winPoints correspond au nombre de parties gagnées. rankPoints est un classement fait par PUBG dont la formule en fonction d'autres variables nous est inconnue.

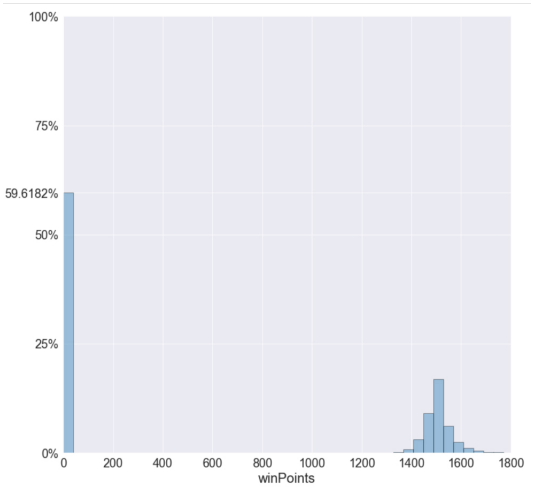

Figure 2: Répartition des valeurs de winPoints

Les problèmes rencontrés se sont manifestés au cours d'une mise à jour des données au début du mois de Novembre, où certaines données comprenaient un winPoints et killPoints à 0, les autres avaient systématiquement la valeur de rankPoints à 0. Les données sont donc séparées en deux catégories : Les anciennes données avec deux classements (winPoints et killPoints) pour lesquelles le classement rankPoints n'existe pas et les données où rankPoints existe et pour lesquelles winPoints et killPoints n'existe plus. Lorsque la valeur d'une variable n'existe plus dans ce cas, elle est automatiquement mise à 0, comme on peut le voir sur la figure 2 pour "winPoints". Ces trois variables sont donc à manipuler avec prudence.

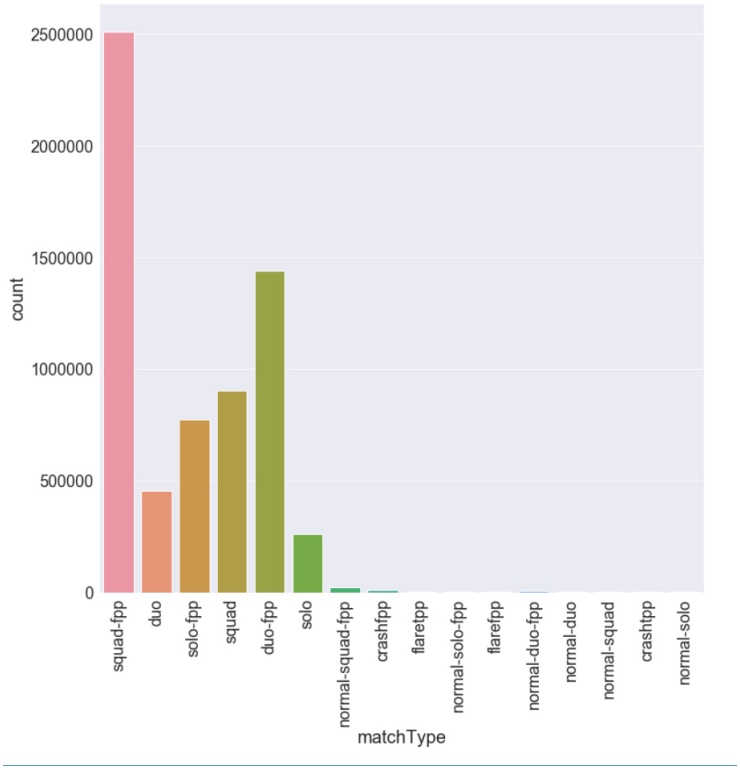

Figure 3: Répartition des types de parties de jeu

Le mode de jeu (duo, solo, en équipe plus nombreuses ?) peut également fortement impacter les résultats et est donc une variable importante à prendre en compte. Les données montrent que plus de 95% des parties se réalisent en "squad-fpp", "duo", "squad", "duo-fpp" et "solo" dans train_V2 et test_V2. Le matchType "fpp" signifie "first person player", soit un match joué en vision à la première personne. Les "squad" sont des parties à un, deux, trois ou quatres joueurs dans une même équipe.

- train_V2.csv contient 4 446 966 données et test_V2.csv en contient 1 934 174.

3.3 Exploration des données

Il est intéressant de regarder, dans un premier temps, les corrélations directes entre les variables et "winPlacePerc".

Comme prévu, on peut remarquer sur la figure 4 que "walkDistance" et "killplace" (Rang du joueur dans la partie en fonction du nombre d'ennemi tués) sont très corrélées à l'indice de classement et inversement pour "killPoints", "rankPoints", "winPoints" et "matchDuration". On a pu remarquer aussi que la relation entre "winPlacePerc" et ses variables corrélées est non linéaire.

Il est également intéressant de regarder la matrice de corrélation des 29 variables (figure 5). "killPoints", "rankPoints" et "winPoints" sont fortement corrélés entre eux. "maxPlace" et "numGroups" sont également très corrélés. Pour éviter toute redondance dans le modèle, nous allons considérer une fusion des variables au lieu de chaque variable dans le modèle pour les variables très corrélées entre elles.

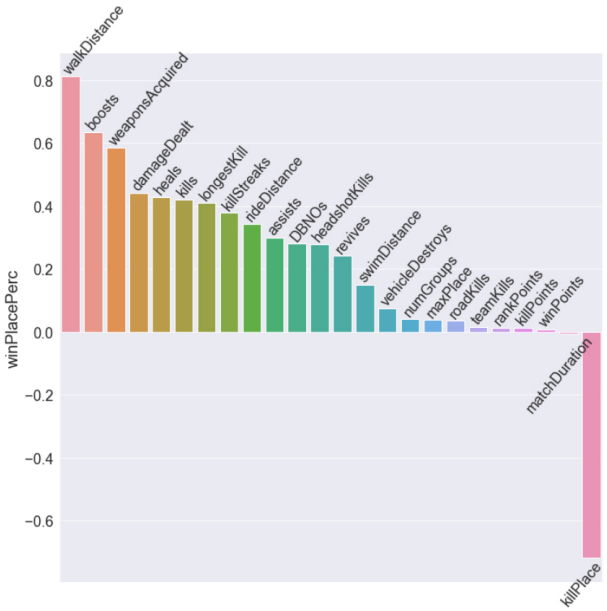

Figure 4: Correllation entre winPlacePerc et les différentes variables

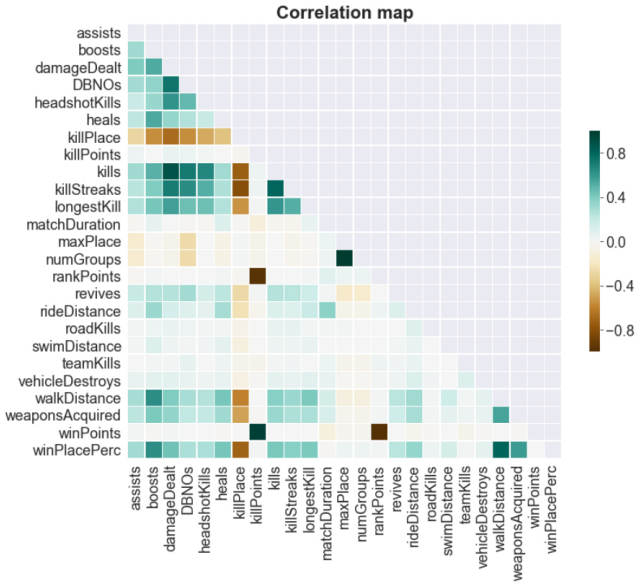

Figure 5: Correllation entre les variables de train_V2 et test_V2

De plus, on peut noter une tendance linéaire entre plusieurs variables (“boosts” et “killPlace” par exemple).

3.4 Vérification de la qualité des données

- Les données sont globalement au bon format. Seule une donnée contient un NaN sur la variable “winPlacePerc”. Le joueur en question a sûrement dû être contraint de quitter le jeu.
- Nous n’avons pas nous-mêmes collecté les données, nous ne pouvons donc pas prouver la fiabilité des données.

4 Préparation des données

4.1 Nettoyage des données

Certaines valeurs de variables seules ou à plusieurs peuvent décrire des situations impossibles ou très peu probables dans une partie d’après notre analyse du jeu. Le but du nettoyage de données est donc de retirer les données vraisemblablement illégitimes qui correspondent aux critères suivants :

- 1294 joueurs tuent d’autres joueurs sans avoir bougé de la partie. En effet, pour avoir une arme, il faut bouger sur la carte.
  (“walkDistance” = 0 & “kills” > 1)
- 658 joueurs tuent plus de 10 joueurs sans avoir bougé de plus de 50 mètres. La carte se rétrécissant au bout d’un moment, le joueur est obligé de bouger. Il est très peu probable qu’un joueur tue 10 personnes avant de devoir bouger.
  (“walkDistance” < 50 & “kills” > 10)
- 19 joueurs tuent plus de 7 personnes en roulant dans un véhicule. Les véhicules étant peu nombreux et difficiles à manoeuvrer, il est peu probable que les joueurs tuents beaucoup à leur bord.
  (“roadKills” > 7)
- 17 joueurs ont nagé plus de 2000 mètres. La carte comprend très peu d’eau et les joueurs ont des barres de fatigue qui diminuent lorsqu’ils nagent.
  (“swimDistance” > 2000)

- 1604 joueurs ont effectué plus de 80Les headshots sont relativement rares.
  (“headshotKills”/”kills” > 80
- 118 joueurs ont tué plus de 8 joueurs sur un court instant. Cela revient à tuer un joueur toutes les dix secondes, et ce, 8 fois.
  (“killStreaks” > 8)
- 22 joueurs ont tué une personne à plus de 1000 mètres. À 1000 mètres, il est presque impossible de viser, le zoom de lunette maximal étant de 16.
  (“longestKill” > 1000)
- 297 joueurs ont marché plus de 10 kilomètres sachant que la carte fait 5 kilomètre de long et de large.
  (“walkdistance” > 10000)
- 214 joueurs ont roulé plus de 20 kilomètres. Comme expliqué précédemment, la carte est un carré de 5 kilomètres de côté.
  (“rideDistance” > 20000)
- 94 joueurs récupèrent plus de 60 armes. Ces joueurs peuvent être considérés comme tricheurs, cas extrêmement rares ou joueurs “trolls” (qui veulent juste explorer la carte par exemple). Pour l’application des modèles, il vaut mieux retirer ces joueurs.
  (“weaponsAcquired” > 60)

4.2 Transformation des données

Nous avons réalisé un one-hot encoding de la variable “matchType”. En effet, comme pour toute variable sous forme de chaîne de caractère parmi un vocabulaire précis, il vaut mieux utiliser à la place un vecteur. L’algorithme arrive mieux à prendre en compte ce type de variable qu’un string.

Création de nouvelles variables joueurs :

- Variable “teamMates” : nombre de coéquipiers d’un joueur.
- Ratio de tirs dans la tête par nombre de personnes tuées.
- Ratio de “killStreak” (série rapide d’adversaires tués) par nombre de tués.
- Ratio de la distance parcourue par nombre de tués. Le classement joueur en nombre de tués divisé par le nombre d’équipes
- Nombre de boosts et de heals utilisés par le joueur.
- La somme de toutes les distances parcourues (marche + natation + conduite).
- La somme de toutes les distances divisée par le nombre d’armes obtenues.

Et création de nouvelles variables d’équipe (moyenne, maximum et minimum pour chaque) :

- Dommages infligés
- Nombre d’armes
- Nombre de réanimations
- Nombre de heals+boosts utilisés

De plus, le dataset que nous utilisions au début du projet a été actualisé en cours de projet. Ainsi, la partie du dataset pré-mise-à-jour ne contenait uniquement que les colonnes “winPlace” et “killPlace” (classement général en nombre de victoires et en nombre de tués). Suite à la publication de la deuxième version, les nouvelles données ne contenaient plus qu’une variable : “rankplace”, remplaçant les deux précédentes. Ainsi, une partie de dataset n’a que “rankplace”, “winPlace” ou “killPlace” de renseigné et un autre ne l’a pas. Nous avons remplacé toutes les valeurs manquantes par les moyennes concernées.

5 Modèles utilisés

Nous avons décidé de comparer plusieurs algorithmes permettant de réaliser un classement à partir des données. Tout d’abord, nous avons pensé à utiliser une régression linéaire, qui permettrait de donner une ‘note’ à

chaque joueur en fonction de tous les paramètres à disposition, et de s'en servir pour réaliser un classement des joueurs en fin de partie. Cette approche est la plus simple à mettre en oeuvre, mais peut-être pas la plus précise. Nous avons essayé de calibrer ce modèle en prenant en compte les corrélations et les problèmes de multicolinéarité (VIF : variance inflation factor). C'est ainsi que nous sommes passés d'une MAE de 0.09015 à une MAE de 0.08035. Il est à noter que nous avons supprimé certaines colonnes très corrélées entre elles et provoquant des problèmes de multicolinéarité. En effet, la multicolinéarité peut avoir des conséquences dramatiques sur la robustesse du modèle.

Nous avons donc continué à chercher des modèles de régression à la fois simples et fiables et avons retenu le decisionTree algorithm. Il s'est avéré meilleur que la régression linéaire, au regard de la MAE (MAE = 0.8787).

$$MAE_{decisionTree} = 0.8787$$

Ensuite nous avons pensé utiliser un randomForest, pour introduire plus d'aléatoire, et atteindre a priori de meilleurs résultats qu'une simple régression (effet bagging, créer un strong learner à partir de weak learners, chaque échantillon réalisant un arbre de la randomForest est susceptible d'entraîner un weak learner). Cependant, il était presque impossible avec nos machines de calibrer les paramètres d'un random Forest à cause de la lenteur de l'algorithme à entraîner (en automatique avec un GridSearch, ou à la main).

$$MAE_{randomForest} = 0.0678$$

Donc, dans l'objectif d'améliorer cet algorithme, nous avons trouvé un autre modèle: XGBoost. Celui-ci consiste en un randomForest amélioré par du boosting: faire aussi un strong learner à partir de weak learners, cependant les mauvaises classifications ont plus de poids afin que les prochaines itérations corrigent le problème. Si on a une certaine amélioration avec ce dernier, on a toujours un temps d'entraînement assez conséquent. Il était donc assez difficile de tester les différents sets de variable et de tune les hyper-paramètres du modèle. La MAE obtenu était de 0.0578.

$$MAE_{XGBoost} = 0.0578$$

Après encore quelques recherches, nous sommes tombés sur le Light GBM. Cet algorithme est proche du XGBoost, et reste de type randomForest. Le gros avantage de ce dernier algorithme est qu'il est rapide, notamment pour les gros datasets comme c'est notre cas (presque 4,5M de lignes). On gagne en précision grâce au parcours d'arbre en profondeur et non en largeur comme XGBoost, et on gagne en mémoire grâce à la conversion en compartiments de valeurs discrètes des valeurs continues. Pour toutes ces raisons nous avons sélectionné l'algorithme Light GBM qui nous semble le plus adapté à notre problème. Nous avons calibré les hyper paramètres à la main, afin d'avoir le meilleur modèle possible. De plus nous avons essayé de voir quels variables rapportait le plus d'informations et lesquelles étaient utiles au modèle.

$$MAE_{LightGBM} = 0.0434$$

6 Post Precessing

Les joueurs jouant en équipe obtiennent tous le même classement en fin de partie. Ainsi, même si nos prédictions indiquaient un variance faible de l'indice de classement pour des joueurs au sein d'un même groupe, nous avons décidé de moyenner "winPlacePerc" pour tous les membres d'un même groupe. Par exemple, si un groupe de 3 joueurs a les indices : J1=0.6, J2=0.6 et J3=0.75, leurs nouvelles valeurs de "winPlacePerc" sera J1=J2=J3=0.65.

7 Évaluation des résultats

Nous étions obligés de tester nos résultats avec la plateforme Kaggle. En effet, nous disposions de datasets d'entraînement et de test bien distincts, mais notre dataset de test était incomplet : il n'y avait pas les valeurs réelles d'indices de classement. On ne peut donc pas nous même évaluer la précision des prédictions sur le dataset de test, par rapport aux valeurs attendues. Seul Kaggle dispose des véritables indices de classement correspondant au dataset de test. Pour évaluer les performances de notre modèle entraîné, il nous faut donc à chaque fois passer par la plateforme de Kaggle, ce qui représente une importante perte de temps, le fichier csv de prédiction étant tout de même de taille assez conséquente.

Nous avons donc décidé de faire nous-même nos test, en faisant de la cross validation avec le dataset d'entraînement. On aura recours aux fonctions "cross_val_score()" de scikit-learn qui réalise ce travail pour nous de façon rapide et efficace. Cependant pour les algorithmes plus lent nous utiliserons seulement un test-split, car la cross validation est trop longue à lancer. Faire nous mêmes nos test nous permet de gagner du temps. Nous avons donc pu comparer nos algorithmes et choisir le plus adapté.

Nous avons aussi divisé notre jeu de données d'entraînement, en une partie test et une partie train : il s'agit d'un test_split. Cela a été principalement utilisé pour le XGBoost et le LightGBM.

Une fois les résultats assez satisfaisants et notre modèle entraîné avec les bons hyper-paramètres, nous faisons donc notre prédiction sur le dataset de test. Nous exportons alors le fichier csv correspondant à ces prédictions, et nous interrogeons la plateforme Kaggle pour avoir notre précision de prédiction.

Les résultats finaux nous classent 278 sur 735 avec une erreur MAE de 0.0435

8 Conclusion

La méthodologie CRISP-DM nous a donc menés à des résultats concluants, validés par le site officiel de la compétition et nous offrant un classement appréciable dans celle-ci. Nous pouvons donc conclure que l'objectif est atteint. Cependant, nous remarquons également que les meilleurs modèles ont une précision bien supérieure à la nôtre et il doit exister des pistes de réflexion supplémentaires pour améliorer significativement nos performances. On peut penser à une analyse des données encore plus approfondie, avec encore d'autres variables synthétiques explicatives, ou même plus simplement un calibrage encore plus précis de nos hyper-paramètres.

9 Annexes

9.1 Annexe 1 - Listes de variables de train_V2 et test_V2

Variables de train_V2.csv et test_V2.csv :

- **DBNOs** - Nombre d'ennemi frappé.
- **assists** - Nombre d'ennemi à qui le joueur a infligé des dégâts et qui a été tué par un coéquipiers.
- **boosts** - Nombre d'item boost utilisés par le joueur.
- **damageDealt** - Dégâts totaux reçus. Les dégâts infligés à soi-même ne comptent pas dans cette variable.
- **headshotKills** - Nombre de joueurs ennemi tué par headshots.
- **heals** - Nombre d'items de soin utilisés. Id - Identité informatique du joueur.
- **killPlace** - Rang du joueur en fonction du nombre d'ennemis tués..
- **killPoints** - Classement sur le nombre de joueur tués.
- **killStreaks** - Nombre maximum d'ennemi tué par le joueur sur un court moment.
- **kills** - Nombre de joueurs ennemis tués par le joueur.
- **longestKill** - Plus grande distance d'où le joueur a tué quelqu'un.
- **matchDuration** - Durée des parties en secondes.
- **matchId** - ID pour identifier un match. Aucune partie n'est commune à train et test.
- **matchType** - Chaîne de caractère identifiant le mode de jeu. Les modes standards sont : "solo", "duo", squad", "solo-fpp", "duo-fpp" ; les autres modes viennent d'événements
- **rankPoints** - Classement Elo-like du joueur. Ce classement est assez incertain et est désapprouvé dans la prochaine version API. À utiliser avec prudence. La valeur -1 pour ce classement équivaut à "None". revives - Nombre de fois que le joueur a réanimé un coéquipier..
- **rideDistance** - Distance totale parcourue en voiture en mètres.
- **roadKills** - Nombre de joueurs tué alors que le joueur était à bord d'un véhicule.
- **swimDistance** - Distance totale parcourue en nageant en mètres.
- **teamKills** - Nombre de fois que le joueur a tué un coéquipier..
- **vehicleDestroys** - Nombre de véhicules détruits.
- **walkDistance** - Distance totale parcourue par le joueur en mètres..
- **weaponsAcquired** - Nombre d'armes enlevé aux ennemis.
- **winPoints** - Classement seulement sur les victoires au cours des matches. S'il y a une valeur différente de -1 dans la variable rankPoints, alors tous les zéro de winPoints doivent être considérés comme "None".
- **groupId** - ID pour identifier un groupe dans la partie. Si un même groupe joue dans différents matches, son groupId changera à chaque matche.
- **numGroups** - Nombre de groupes dont nous avons les données dans le matche.
- **maxPlace** - Pire classement que nous avons en données dans le matche.

Variable de train_V2.csv seulement :

- **winPlacePerc** - La prédiction finale. Le classement entre 0 et 1. Le principe est expliqué à la page 3.

9.2 Remerciements

➤ Ce rapport a été fait sous LaTeX. Nous remercions **Armand FOUCAULT** et **Benoît PORTEBOEUF** pour avoir créé ce template IMT Atlantique sous LaTeX. Il est disponible sur le lien suivant :

https://github.com/bporteboeuf/imtaLatexTemplate

10 Sources

➤ Projet Kaggle - PUBG - Finish Placement Prediction :
`https://www.kaggle.com/c/pubg-finish-placement-prediction`

➤ Aide pour limiter l'utilisation de la RAM :
`https://www.kaggle.com/gemartin`

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

IMT Atlantique Bretagne - Pays de la Loire - www.imt-atlantique.fr

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 03
T +33 (0)2 29 00 11 11
F +33 (0)2 29 00 10 00

Campus de Nantes
4, rue Alfred Kastler - La Chantrerie
CS 20722
44307 Nantes Cedex 03
T +33 (0)2 51 85 81 00
F +33 (0)2 51 85 81 99

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
T +33 (0)2 99 12 70 00
F +33 (0)2 99 12 70 08