Projet IAS - Prédiction du taux de la victoire du jeu LOL

Hongyu YAN, Liuyi CHEN, Zhenhai XU,
Shiqing HUANG, Shuangrui CHEN
Tuteur: Kirian Guiller Prof: François Landes, Kim Gerdes

Introduction

Pour un jeu eSports, la victoire est le but ultime. En raison de la profondeur et de la complexité du jeu, il peut souvent être difficile pour les gens de déterminer quelle équipe va gagner jusqu'aux dernières minutes du match.

Nous avons tenté, avec un certain succès, de déterminer quelle équipe a le plus de chances de gagner en utilisant diverses techniques d'apprentissage automatique.

Avec les données trouvées dans le kaggle et l'API de Riot Games, on a obtenue des données de match de jeu de LOL. Il y a 38 features dans le format de ces données, et en utilisant le kit Sklearn, on a trouvé des models(LR et GBR) qui adapte bien ce format et converge bien pour prédire le taux de voctoire de ce jeu.

Acquisition de données

Idées pour l'origine des données

Contrairement à d'autres groupes, nous pensons que le jeu de données que nous pouvons obtenir de kaggle est quelque peu limité. Soit ils n'ont pas assez de features, soit ils ne ciblent qu'un seul point dans le temps ou de fin de jeu. Nous avons estimé que le principal intérêt de prédire le taux de victoire d'une partie se situait pendant la partie plutôt qu'après, et nous avons donc décidé d'extraire nous-mêmes les données dont nous avions besoin. Heureusement, Riot fournit une API ouverte aux développeurs qui ont besoin de données de jeu. Nous avons donc commencé par réaliser un crawler ainsi qu'un outil de collecte de données et de statistiques. Nous avons d'abord utilisé un ensemble de données de 10 minutes trouvé sur kaggle, qui comprenait un "gameId", un "blueWins", et les features suivantes pour l'équipe *blue* et l'équipe *red* (donc 40 features en total):

- WardsPlaced
- WardsDestroyed
- FirstBlood
- Kills
- Deaths
- Assists
- EliteMonsters
- Dragons
- Heralds
- TowersDestroyed
- TotalGold
- AvgLevel
- TotalExperience
- TotalMinionsKilled
- TotalJungleMinionsKilled
- GoldDiff
- ExperienceDiff
- CSPerMin
- GoldPerMin

Procédure

Tous les données dans notre ensemble de données de base sont les données à la 10ème minute de chaque match. Les correspondances sont identifiées par gameId. D'après le gameId et le serveur dans cet ensemble de données, en utilisant l'API Riot et l'outil lolWatcher, nous avons obtenu toutes les lignes de temps correspondant à chaque match et stocké les résultats obtenus dans un fichier csv. En utilisant cet ensemble de données, nous avons enregistré des nouvelles données de match toutes les 5 minutes à partir de la 10ème minute jusqu'à la 35ème minute (le nombre d'échantillons au-delà de 35 minutes est très faible), donc 5 nouveaux ensembles de données sont généré. La structure et les features des ensembles de données nouvellement créés étaient cohérentes avec l'ensemble de données de base.

Pour valider l'exactitude de notre crawl et de notre traitement en analysant et en traitant la relation entre les données et les valeurs des features stockées dans la ligne de temps, nous avons d'abord essayé de reconstruire un ensemble de données pour la 10e minute du jeu et de le comparer à notre ensemble de données de base. Et puis nous avons généré les 5 nouveaux fichiers présentés précédemment. Leurs noms de fichiers sont "Test_XXmins.csv".

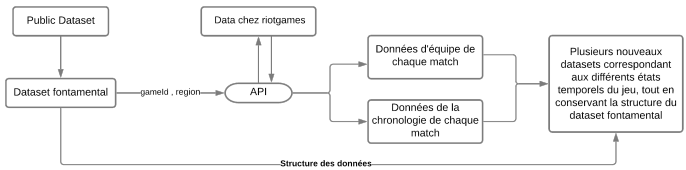

FIGURE 1. *Procédure de crawler*

Observation de donnée et Pre-Processing

Features

On peut observer que les différents features de nos données ne sont pas dans le même format, certains sont des entiers positifs, d'autres des 0 ou 1, et d'autres encore des nombres réels. Cela nous fait d'abord penser à l'arbre de régression comme modèle et à l'utilisation d'un curseur pour normaliser nos données, etc. Après avoir examiné attentivement les données, nous avons confirmé que la moitié des features de nos données ne sont pas nécessaires, et selon la matrice de corrélation, on peut voir que la moitié des features ont une symétrie négative avec l'autre moitié, donc utiliser PCA ou supprimer manuellement les features qui ont la même corrélation peut être le choix le plus favorable pour certain models.

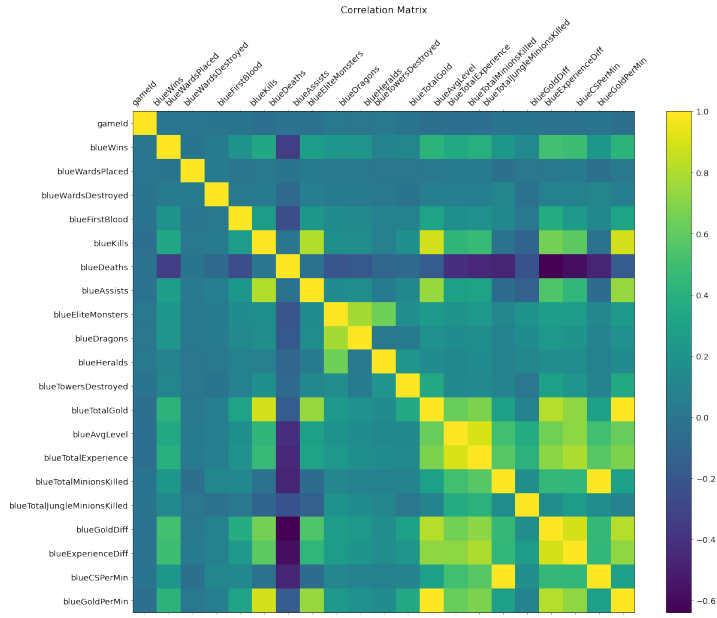

FIGURE 2. *Matrice de corrélation*

Pre-Processing

1. La signification de PCA

Nous utilisons le PCA pour la réduction de la dimensionnalité, en conservant les caractéristiques les plus importantes des données de haute dimension, en supprimant le bruit et les caractéristiques sans importance, afin d'améliorer l'efficacité du traitement des données.

2. La signification de Scaler

La distance des points d'échantillonnage dans l'espace des features sera dominée par certaines grandes valeurs des features, tandis que l'influence des autres features est relativement faible. Mais toutes les features de l'échantillon doivent avoir le même niveau d'influence sur la distance de l'échantillon dans l'espace des features. Nous devons donc normaliser les données.

Model

Séléction de model

Sur la base de l'ensemble de données de 10 minutes, après avoir comparé différents modèles qui nous semblaient avoir du potentiel, nous avons sélectionné deux modèles, la régression linéaire et la régression par boosting de Gardient (GBR). Les deux sont

considérés comme des modèles supérieurs sous l'évaluation MSE ou R2, mais comme la régression linéaire est un modèle simple, nous étions plus sceptiques quant à ses performances au début, et dans l'ensemble de données beaucoup plus tard dans le jeu, il montre qu'il est légèrement moins bon que le GBR, mais encore assez limité, et étant donné ses excellentes performances, nous pensons que c'est un choix plus économique, étant donné que nous pouvons sacrifier la précision.

Le but du modèle Gradient Boosting est d'ajouter des weak learners en utilisant la descente de gradient pour minimiser la perte du modèle. Un nouveau weak learner est ajouté à la fois et les weak learners existants dans le modèle restent inchangés. Ici, on utilise le Decision Tree comme le weak learner, il s'appelle Gradient Boosting Decision Tree.

TABLE 1. *Tests des models différents*

Model	Neg MSE	Neg MAE	R2
LinearRegression	-0.181060	-0.372796	0.274981
Lasso	-0.250062	-0.500059	-0.001320
ElasticNet	-0.250062	-0.500059	-0.001320
KNeighborsRegressor	-0.215678	-0.367813	0.136377
DecisionTreeRegressor	-0.363409	-0.365563	-0.456190
GradientBoostingRegressor	-0.181719	-0.359290	0.272326
RandomForestRegressor	-0.188095	-0.367437	0.249210

Voici son idée:

1. Nous voulons utiliser la méthode Gradient Descend pour minimiser le coût de perte: $\theta = \theta - \alpha \cdot \frac{\partial}{\partial \theta} L(\theta)$

2. Dans la m ième itération, les premiers weak learners de base m-1 sont fixés : $f_m(x) = f_{m-1}(x) + \rho_m h_m(x)$

3. Dans l'étape M nous voulons minimiser la fonction du coût $L(f) = \sum_{i=1}^{N} L(y_i, f_m(x_i))$ en utilisant Gradient Descend pour obtenir le gradient négatif(résidu) $-\frac{\partial L(y, F_{m-1}(x))}{\partial F_{m-1}(x)}$, et puis, on utilise un arbre de régression pour l'ajuster.

Ce modèle a certains avantages:

1. À chaque étape, nous ajusterons mieux les données sur la base du cycle précédent, ce qui peut assurer un faible écart. En même temps, il peut éviter l'overfitting avec de nombreux petits arbres.

2. Il sélectionne constamment des features et segmente des features, de sorte que le classement par importance des features peut être facilement obtenu.

3. L'arbre de décision peut gérer divers types de données de manière flexible, y compris des valeurs continues et discrètes. Il peut bien traiter notre dataset qui a des

données de différents types.

4. Il y a de nombreux facteurs affectant le taux de victoire. Chaque facteur est important, mais pas décisif. D'après ce que nous avons appris en cours par rapport à l'arbre de décision, il peut bien refléter que si on pousse beaucoup de tours, mais que l'économie est bien pire, on ne gagnera pas nécessairement.

Le modèle Linear Regression marche aussi bien. Par rapport au modèle Gradient Boosting, il est plus simple et fonctionne plus vite. Il s'adapte aussi bien à nos données mais nous n'avons pas d'hyper-paramètres à optimiser pour lui, car le principe du modèle Linear Regression est très simple. Pour chaque exemple $(x^{(i)}, y^{(i)})$ on a la fonction $y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}$ (y prédit), et l'objectif est de minimiser l'écart entre y prédit et y vrai. En général, on utilise la méthode des moindres carrés (Least Squared Error) pour calculer l'écart et la descente de gradient pour l'optimisation.

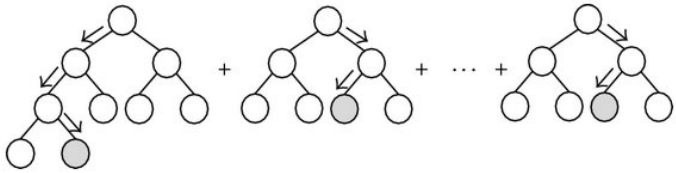

FIGURE 3. *Gradient Boosting*

Fonction Score

Nous avons essayé trois métriques d'évaluation du modèle MSE, MAE et R2. Et nous avons décidé d'utiliser R2 pour calculer l'erreur. Tout d'abord, nous ne pouvons pas observer l'écart entre les différents modèles avec MAE. MSE et R2 sont bien. Mais lorsque les dimensions sont différentes, il est difficile de mesurer l'efficacité du modèle. R2 nous aide à régler ce problème en comparant notre modèle avec le modèle de base (baseline model).

Le meilleur score possible est de 1,0 et il peut être négatif (car le modèle peut être arbitrairement pire). Si le score est de 0, alors notre modèle égale au modèle de base.

La formule de R2 : $R^2 = 1 - \frac{\sum_i (\hat{y}^{(i)} - y^{(i)})^2}{\sum_i (\bar{y}^{(i)} - y^{(i)})^2}$

Nous avons fait des modèles sur différents délais nous permettant d'obtenir le taux de victoire en temps réel. Nous pouvons observer que les résultats obtenus s'améliorent de plus en plus au fil du temps. Mais à partir de 30 minutes, en raison de l'augmentation de l'importance des opérations et la composition de l'équipe, la corrélation entre le taux de victoir avec les features a diminué.

Hyper-paramètre optimization

Nous utilisons la méthode GridSearchCV fournie dans sklearn qui automatise la recherche d'un optimum parmi les hyperparamètres. Il fait essayer chaque combinaison de paramètres via un parcours de boucle et renvoie la combinaison de paramètres avec la meilleure valeur de score.

Le paramètre de GridSearchCV 'param_grid' doit entrer un dictionnaire ou une liste, c'est-à-dire les valeurs des paramètres à optimiser. Le dictionnaire suivant est entré dans param_grid:

```
'pca__n_components': range(20, 25),
'gbr__n_estimators': range(30, 50),
'gbr__learning_rate': [0.1, 0.01, 0.3],
'gbr__loss': ['ls', 'lad', 'huber', 'quantile']
```

La signification et la plage de sélection des paramètres de PCA:

n_components: Nous essayons de réduire le nombre de features de 20 à 25 pour optimiser l'efficacité d'apprentissage du modèle

La signification et la plage de sélection des paramètres de BGR:

n_estimators: le nombre maximum d'itérations d'apprenants faibles, ou le nombre maximum d'apprenants faibles.Si n_estimators trop petits peuvent entraîner un Underfitting, et si trop grands peuvent entraîner un Overfitting. Nous avons limité la plage de 30 à 50.

learning_rate: le taux d'apprentissage est un paramètre de réglage dans un algorithme d'optimisation qui détermine la taille de pas à chaque itération tout en se déplaçant vers un minimum d'une fonction de perte. Une trop faible valeur de learning_rate semble retenir le modèle de converger, une grande valeur produit des effets imprévisibles. Nous essayons d'ajuster les paramètres avec 0,1, 0,01, 0,3.

loss: fonction de perte à optimiser.

Par rapport à la méthode de LAD, LS présente l'avantage que la solution optimale est unique et peut généralement converger plus rapidement que LAD. Bien qu'elle soit grandement affectée par la perturbation des outlier (données anormales), mais parce que notre ensemble de données ne le fait pas avoir cette interférence, donc LS est un meilleur choix.

Least squares: $S = \sum_{i=1}^{n} |y_i - f(x_i)|$

Least absolute deviations: $\sum_{i=1}^{n} (f(x_i, \vec{\alpha}) - y_i)^2 = \left\| \vec{f} - \vec{y} \right\|_2^2$

Le résultat de l'optimisation des hyperparamètres:

n_components = 23; n_estimators = 32; learning_rate = 0.1; loss = "ls"

Résultat

En utilisant les paramètres et les modèles optimaux, nous avons entraîné six modèles avec l'état du jeu à différents moments (de 10 à 35 minutes). Ceci a été enregistré via la sérialisation en python. Grâce à ces modèles et à notre robot d'exploration, la saisie de l'identifiant d'un jeu ou d'un joueur génère instantanément un graphique du taux de victoire en temps réel. Pour ce faire, nous générons un graphique des features qui ont le plus d'impact sur le taux de gain et la variation du taux de gain.

TABLE 2. *R2 Scores*

Data Time	R2 - Linear	R2 - GBR
10 mins	0.275249 +/- (0.021682)	0.279042 +/- (0.028355)
15 mins	0.394802 +/- (0.025301)	0.419105 +/- (0.035374)
20 mins	0.461979 +/- (0.017610)	0.483372 +/- (0.019836)
25 mins	0.507631 +/- (0.025282)	0.527720 +/- (0.033660)
30 mins	0.482030 +/- (0.038144)	0.494009 +/- (0.055401)
35 mins	0.409463 +/- (0.058324)	0.382040 +/- (0.071225)

Conclusion

Comme nous pouvons le voir dans les résultats, le taux de victoire de League of Legends n'est pas affecté par un seul facteur et varie de temps en temps sur le taux de victoire du jeu, par exemple, le manque de cohérence entre les premiers stades et la victoire finale. Et au-delà de 25 minutes, le lien entre le jeu et les données statistiques disponibles commence à diminuer en raison de la maturité des joueurs et de facteurs liés à la composition des équipes. Il est donc possible d'appliquer des modèles différents à des moments différents, les prédictions avant les victoires pouvant éventuellement intégrer des facteurs plus détaillés, et les prédictions en fin de match pouvant éventuellement intégrer des considérations relatives à l'alignement.

Mais étant donné que nos projections de taux de gain peuvent atteindre un R2 de 0,5 points dans la période la plus haute et un bas de 0,27, nous pensons que c'est un modèle acceptable et tout à fait souhaitable.

À l'avenir, nous espérons poursuivre ce projet, par exemple en créant un petit site web (lors de la saisie d'un nom d'utilisateur, prédisant directement le taux de gain du jeu actuel de l'utilisateur en temps réel) et en essayant de quantifier l'impact des alignements.

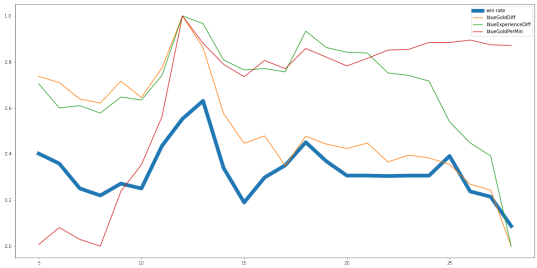

FIGURE 4. *Gradient Boosting*

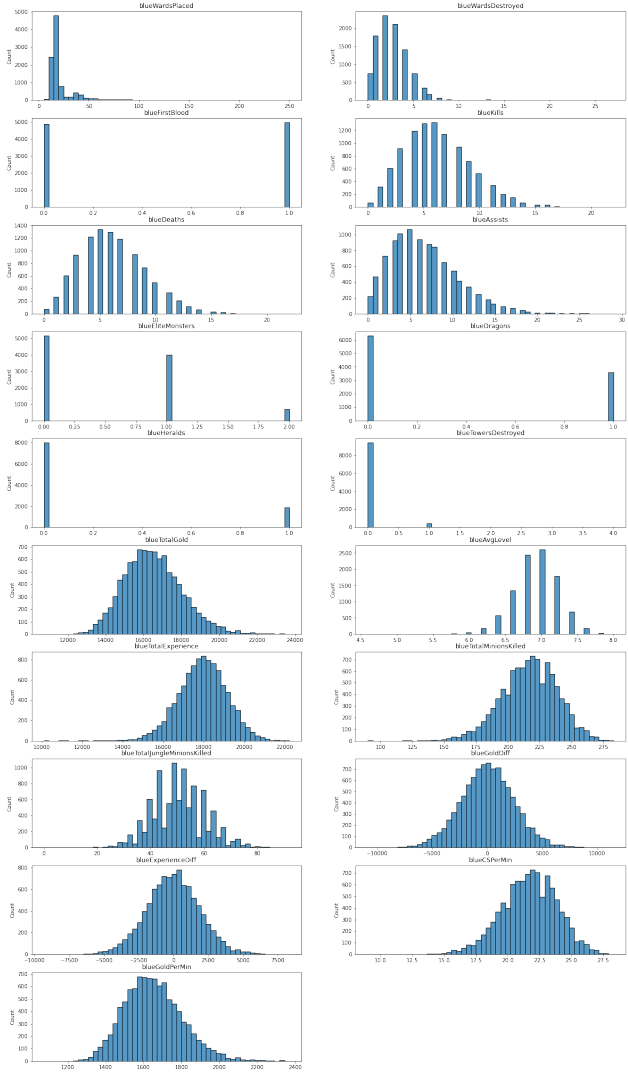

FIGURE 5. *Distribution de Features*