Gradient Boosting

Badr GHAZLANE

06/12/2016

Gradient Boosting

Introduction

Suite au Random Forest, afin d'améliorer les résultats de ma prédiction, j'ai décidé d'utiliser le gradient boosting extreme. Il s'utilise sur R via le package XGB. Cet algorithme est un dérivé du gradient boosting, qui a l'avantage par rapport au gradient boosting de pouvoir contrôler le sur-apprentissage. D'ailleurs, dans cette partie je vais détailler mon travail en considérant que j'ai utilisé le gradient boosting afin de simplifier l'approche. De plus, toujours dans l'optique d'améliorer mon score de prédiction j'ai décidé de me concentrer sur la partie de cross-validation, réglage des hyper-paramètres, entrainement et prédiction. En contre-partie, j'ai repris une partie de la préparation des données d'un autre compétiteur. Son travail a consisté à mettre en valeur les dates minimales auxquelles les objets sont traités par la machine.

Packages utilisés

Les packages utilisés dans cette partie permettent l'utilisation de l'algorithme du gradient boosting: xgboost ou l'utilisation de format spéciaux tels que la matrice "sparse" ou encore l'affichage de graphique avec dplyr et tidyr.

Préparation des données

Comme dans la partie précédente l'importation des données se fait par la fonction fread() qui les stocke dans une data.table puis comme expliqué dans la première partie les valeurs manquantes sont dans un premier temps remplacées par -2 et -10 respectivement dans les données numériques et temporelles.

Choix des variables

Tout d'abord, notons que au vu de la performance de ma machine et au vu des données, je ne pouvais garder la totalité des variables les 2300 variables. Dans un premier temps, j'ai choisi les 30 variables les plus corrélées avec la target Response tout en étant conscient que la plupart de ces variables étaient corrélées entre elles. L'apport d'informations était donc très faible. Cela se montrait dans le score de la prédiction qui était dans les 90% premiers pour-cent.

Par la suite, j'ai utilisé le XGB lui même pour choisir les variables. Cette partie sera détaillée dans l'explication du gradient boosting. Pour cela, j'ai entraîne mon model sur mes données «faiblement» préparées. Puis j'ai récupéré les variables de plus grandes importances: - 138 variables dans les données numériques - 58 variables dans les données temporelles

Featuring

La taille des données empêche, d'appliquer des opérations sur un bloc entier. Par exemple, le calcul du min ou max sur toutes les données prend plus de 20min. Mais en appliquant ces mêmes opérations sur des morceaux de données réduit le temps de traitement. Ainsi, j'ai reparti mes données en 20 morceaux (chunks).

Sur chacun des morceaux, je récupère le min et max des données temporelles dans un premier temps sur les données d'entraînement puis sur les données tests.

Mais la partie technique du featuring repose sur la date minimale auquel l'objet se trouve dans une machine. Cette partie complexe et spécifique à ce problème a volontairement été mise de côté. J'ai donc repris le travail d'un Kaggler (Faron), il a pris la date minimale pour chaque objet et a trié par ordre croissant d'Id. Puis il a affecté des valeurs selon l'ordre.

Mise en place du critère d'évaluation MCC

Le Matthew Correlation Coefficient est utilisé pour évaluer la qualité d'une classification binaire. Il retourne une valeur comprise entre -1 et $+1$. Un coefficient de $+1$ représente une prédiction parfaite, 0 une prédiction aléatoire et -1 indique une prédiction en total opposition avec l'observation.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN * FN)}}$$

Les éléments du MCC sont définis tels que:

TP est donné par: $\sum_{i=1}^{N} \mathbb{I}\left\{\widehat{y}_i = 1 \cap y_i = 1\right\}$
TN est donné par: $\sum_{i=1}^{N} \mathbb{I}\left\{\widehat{y}_i = 0 \cap y_i = 0\right\}$
FP est donné par: $\sum_{i=1}^{N} \mathbb{I}\left\{\widehat{y}_i = 1 \cap y_i = 0\right\}$
FN est donné par: $\sum_{i=1}^{N} \mathbb{I}\left\{\widehat{y}_i = 0 \cap y_i = 1\right\}$

La fonction qui calcule le MCC prend en paramètre les prédictions qui sont les probabilités que $\widehat{y}_i$ appartienne à une classe. Donc, un seuil est défini: conf. La valeur standard 0.5 lui est attribuée afin de pouvoir définir TP, TN, FP et FN selon les formules ci-dessus.

```
conf = .5
mcc_eval <- function(preds, dtrain) {
    labels <- getinfo(dtrain, "label")
    tp <- as.numeric(sum((labels == 1) & (preds >= conf)))
    tn <- as.numeric(sum((labels == 0) & (preds < conf)))
    fp <- as.numeric(sum((labels == 0) & (preds >= conf)))
    fn <- as.numeric(sum((labels == 1) & (preds < conf)))

    if(tp+fp == 0) {

        mcc <- 0
    } else {
        mcc <- (tp*tn-fp*fn)/(sqrt((tp+fp)*(tp+fn)*(tn+fp)*(tn+fn)))
    }
    return(list(metric="mcc", value=mcc))
}
```

Cross-Validation

Avant de débuter la cross-validation, il est nécessaire d'assembler les données en prenant soin de mettre de côté les Response et Id dans des vecteurs car le training n'exige que les données pures. Rappelons que l'intérêt de la cross-validation est double. Elle permet dans un premier temps d'avoir une idée de la performance du modèle via le score. De plus, elle permet de contrôler le sur-apprentissage. Le sur-apprentissage caractérise un modèle qui fait des prédictions très précises sur les données qu'il a apprises, mais dès qu'il est confronté

à de nouvelles données, le modèle est trop complexe pour s'adapter et faire des prédictions précises. Même si le gradient boosting extreme contient une amélioration qui permet le sur-apprentissage, les nombreux paramètres peuvent facilement nous faire basculer dans le sur-apprentissage.

Ensuite, on découpe les données afin de faire la cross-validation. On divise l'échantillon original en k échantillons, puis on sélectionne un des k échantillons comme ensemble de validation et les (k-1) autres échantillons constitueront l'ensemble d'apprentissage. On calcule le MCC. Puis on répète l'opération en sélectionnant un autre échantillon de validation parmi les (k-1) échantillons qui n'ont pas encore été utilisés pour la validation du modèle. L'opération se répète ainsi k fois pour qu'en fin de compte chaque sous-échantillon ait été utilisé exactement une fois comme ensemble de validation. La moyenne des k MCC est enfin calculée pour estimer l'erreur de prédiction.

Instinctivement, plus k est grand et plus l'erreur sera précise, mais il est évident que le temps de calcul augmente. Au vue de mes données j'ai d'abord commencé par k=10, mais le temps de calcul est beaucoup trop long, d'autant plus que la précision n'est pas significative. J'ai donc choisi k=5. Les données sont présentées dans un ordre précis (en fonction de l'ordre dans lequel les objets passent dans les machines) il était donc impératif de les randomiser. Et, étant donné du grand déséquilibre entre les classes, il a aussi fallu stratifier les k-folds. Ce qui donne k strates où la répartition des 2 classes est similaire.

Notons l'estimateur de Response dans chaque fold $\widehat{y_i^k}$ pour $k \in \{1, 2, 3, 4, 5\}$ et les données dans chaque fold $\vec{x_i^k}$ pour $k \in \{1, 2, 3, 4, 5\}$.

Entrainement

Dans cette partie, nous allons entraîner le modèle. Pour cela, il est nécessaire de faire la distinction entre 2 fonctions: xgb.cv() et xgb.train(). La première entraîne le modèle en faisant la cross validation, et restitue le MCC moyen sur les 5 folds à chaque itération. L'avantage est qu'elle enregistre le MCC moyen et qu'il est par conséquent possible de l'afficher sous forme de graphique mais son inconvénient est qu'elle est beaucoup plus lente que la 2ème. En effet, la deuxième ne prend qu'un seul fold et entraîne le modèle dessus. Elle est donc plus rapide, mais son inconvénient et qu'elle ne stocke pas les valeurs du MCC moyen. J'utilise donc la première dans un cadre pédagogique et la deuxième dans le cadre de la performance.

Choix des hyper-paramètres

Par la suite, j'ai défini les paramètres et hyper-paramètres du gradient boosting afin de les faire varier. J'ai réalisé un grid-search mais seulement pour une quarantaine d'estimateurs du fait du temps de traitement.

Le grid search a été réalisé de la façon suivante. J'ai classé les paramètres dans l'ordre ci-dessous. Je fais varier le premier paramètre dans les valeurs que j'ai défini. Puis je garde celle qui aboutit à la meilleure prédiction. Je fixe cette valeur définitivement puis je fais varier le paramètre suivant dans les valeurs définies puis je garde la meilleure valeur, et ainsi de suite.

Les différents paramètres sur lesquels j'ai joué sont répartis en 2 classes. Les premiers d'entre-eux sont relatifs au gradient boosting de façon intrinsèque. Les deuxièmes d'entre-eux caractérisent le classifieur simple, c'est à dire l'arbre.

Paramètres propres au gradient boosting

- le learning rate (eta): C'est le coefficient qui est attribué à chaque nouveau super-classifieur. Ainsi, il permet de contrôler l'apprentissage. En effet, chaque modèle se voit attribuer un coefficient, puis tous les modèles pondérés sont sommés itérativement entre-eux. Ce paramètre est fondamental: Une bonne valeur du learning rate permet de trouver un bon minimum de la fonction de perte, mais il faut faire attention à ne pas introduire des valeurs trop faibles, qui certes minimiseront la fonction de perte efficacement, mais le sur-apprentissage donnera une mauvaise prédiction. Les valeurs que j'ai testé sont: [0.0001, 0.001, 0.01, 0.1, 0.2, 0.3]. La valeur optimale est 0.01

- base_score: Il correspond au biais initial introduit. Sa valeur par défaut est 0.5, mais étant donnée que le score des meilleurs prédictions est 0.52, il est incensé d'avoir un tel score initial. D'autant plus que, les prédictions les plus courantes sont évaluées par des critères d'erreurs qu'il faut minimiser, d'où la valeur par défaut élevée. À l'inverse, mon base_score doit être petit ce qui traduit une grande erreur et ce qui permet au gradient boosting de compenser cette erreur avec le classifieur simple pondéré. J'ai donc fait varier cette valeur de 0 à 0.01 avec un pas de 0.001. La valeur optimale est 0.005
- n_rounds: Il correspond au nombre de classifieurs simples utilisées et donc au nombre d'itérations. Plus il y'a de classifieurs simples et plus le modèle est précis. En revanche un modèle trop complexe est synonyme de sur-apprentissage. J'ai testé les valeurs suivantes: [50,200,700, 800,20000] La valeur optimale est 800.

Paramètres propres au classifieur simple

- max_depth: Il s'agit de la profondeur de l'abre, plus l'abre est profond et plus il y'a de splits, donc plus le modèle est complexe. L'idée centrale du gradient boosting est d'agréger des classifieurs simples. Ce paramètre contrôle donc le sur-apprentissage. J'ai testé 3,4,5 et 7. J'ai choisi des valeures relativement faibles étant donné que j'avais choisis n_rounds=800
- min_child_weight: Il s'agit de la somme des poids des classifieurs simples minimale. Plus il est élevé et moins il y'a de sur-apprentissage. Sa valeur par défaut étant 1 j'ai testé 2,3 et 4.
- subsample: Proportion d'observations considérées dans chaque arbre. J'ai testé des valeurs entre 0.6 et 1 avec un pas de 0.1
- col_sample_bytree: Proportion de colonnes choisies pour chaque arbre. J'ai testé des valeurs entre 0.6 et 1 avec un pas de 0.1

```
param <- list(objective = "binary:logistic",
              eta = 0.01,
              max_depth = 4,
              min_child_weight = 3,
              subsample = 0.8,
              colsample_bytree = 0.8,
              base_score = 0.005,
              eval_metric = mcc_eval,
              maximize = TRUE)

#SANS XGB.CV ENLEVER LES COMMENTAIRES EN DESSOUS
#dmodel <- xgb.DMatrix(X[model,], label = Y[model])
#dvalid <- xgb.DMatrix(X[valid,], label = Y[valid])
```

Théorie du gradient boosting

Puis on lance la fonction xgb.cv() qui va entraîner le model sur chacun des folds puis va calculer le MCC sur chacun des folds à chaque itération du gradient boosting.

Comme dans tout problème d'apprentissage supervisé, on a les données $\{y_i, \vec{x_i}\}_1^N$ et on veut trouver la fonction F^* qui relie $\vec{x_i}$ à y_i et telle que la fonction de perte $\psi(y, F(x))$ est minimisée.

On définit ensuite F^* telle que: $F^* = argmin_{F(\vec{x})}\ E_{y,\vec{x}}[\psi(y, F(\vec{x}))]$ En réalité, F^* est donnée par la stratégie bayésienne car c'est la stratégie pour laquelle la perte moyenne est minimale.

Les pertes sont définies par la fonction de perte qui est dans ce cas à l'entropie croisée, car dans les paramètres du gradient boosting, j'ai choisi d'utiliser une régression logistique binaire afin de pourvoir définir un threshold qui maximise le MCC. Donc on a:

$$\psi(y_i, F(\vec{x})) = -y_i * \ln(p(\widehat{y_i}/\vec{x_i})) - (1 - y_i) * \ln(1 - p(\widehat{y_i}/\vec{x_i}))$$

L'objectif de l'algorithme de gradient boosting est d'approcher la fonction F^* par une fonction F qui est définie telle que: $F(\vec{x}) = \sum_{m=0}^{M} \beta_m * h(\vec{x}, \vec{a}_m)$

L'idée principale est d'agréger plusieurs classifieurs ensembles mais en les créant itérativement. Ces « mini-classifieurs » sont généralement des fonctions simples et paramétrées: des arbres de décision dont chaque paramètre est le critère de split des branches. Le super-classifieur final F est une pondération par les coefficients β_m de ces mini-classifieurs. Notons que M est le nombre de classifieurs dont est composé le super-classifieur final et $\vec{a}_m$ sont les paramètres des classifieurs simples autrement dit le poids des variables.

On peut distinguer 4 étapes dans la construction du super classifieur final F:

1. Prendre une pondération quelconque (poids β_m) de mini-classifieurs (paramètres $\vec{a}_m$) et former son super-classifieur
2. Calculer l'erreur induite par ce super-classifieur, et chercher le mini-classifieur qui s'approche le plus de cette erreur (ce qui revient à le chercher dans l'espace des paramètres)
3. Retrancher le mini-classifieur au super-classifieur tout en optimisant son poids par rapport à la fonction de perte définie
4. Répéter le procédé itérativement M fois

Remarque 1: Pour éviter la surcharge j'ai volontairement omis l'indice k qui définit dans quel fold se trouve la donnée. Mais, pour être précis, cet algorithme s'applique à chacun des 5 folds.

Remarque 2: Dans la boucle for ci-dessus ρ est introduit pour éviter une confusion qui sera expliquée ci-dessous. Il désigne aussi les coefficients qui pondèrent les classifieurs simples.

Initialisation de F_0:

$$F_0(\vec{x}) = argmin_\rho \sum_{i=1}^{N} \psi(y_i, \rho)$$

.

Puis POUR m=1 à M:

On calcule l'erreur que commet le super classifieur, elle est parfois appelée "résidus":

$$\tilde{y}_i = \left[-\frac{\partial \psi(y_i, F(\vec{x}))}{\partial F(\vec{x_i})} \right]_{F(\vec{x}) = F_{m-1}(\vec{x})}$$

On actualise $\vec{a}_m$ de telle sorte à ce qu'on essaie de corriger l'erreur du super-classifieur à l'aide des classifieurs simples pondérés. On calcule alors l'erreur quadratique moyenne que l'on minimise par une première descente de gradient.

$$\vec{a}_m = argmin_{\vec{a},\beta} \sum_{i=1}^{N} [\tilde{y}_i - \beta * h(\vec{x}, \vec{a}_m)]^2$$

β désigne la valeur le coefficient du classifieur simple qui a servi à minimiser $\vec{a}_m$ donc pour éviter toute ambiguïté, il était nécessaire d'introduire ρ_m qui désigne donc le coefficient du classifieur simple à l'itération m sans minimisation. On actualise ρ_m de telle sorte à minimiser la fonction de perte. Cette minimisation impose une descente de gradient. Le coefficient de la descente de gradient constitue le learning rate.

$$\rho_m = argmin_\rho \sum_{i=1}^{N} \psi(y_i, F_{m-1}(\vec{x_i}) + \rho * h(\vec{x_i}, \vec{a}_m))$$

On actualise enfin le super-classifieur:

$$F_m(\vec{x}) = F_{m-1}(\vec{x}) + \rho_m * h(\vec{x}, \vec{a}_m)$$

Appel de mcc_eval() qui calcule $MCC^{(k)}$, le score dans chaque fold. Ensuite le score total est la moyenne des 5 folds, d'où:

$$MCC_m = \frac{1}{5}\sum_{k=1}^{5} MCC_m^{(k)}$$

FinDePOUR

Revenons au chapitre Choix des variables. Nous avions utilisé l'algorithme de gradient boosting afin de sélectionner les variables les plus pertinentes. La boucle pour précédente s'applique donc et au fur et à mesure des itérations nous obtenons le vecteur $\vec{a}_m$, et ce dernier est obtenu par minimisation de l'écart entre l'erreur du super-classifieur final et du classifieur simple. Or, rappelons que $\vec{a}_m$ sont les paramètres du classifieur simple, autrement dit le poids de chaque variable dans le classifieur simple. Choisir les variables consiste donc à choisir les celles de plus grands poids.

faire le lien avec les hyper-paramètres.

```
#LANCER AVEC 300 ESTIMATEURS LA NUIT

xgb_cv_1 = xgb.cv(params = param,
                  data = X,
                  label = Y,
                  nrounds = 200,
                  nfold = 5,
                  prediction = TRUE,
                  showsd = TRUE,
                  stratified = TRUE,
                  verbose = TRUE,
                  print.every.n = 1
)
```

Une fois le model entraîne, on vérifie le MCC à chaque itération (ici itération désigne le nombre d'estimateurs). Puis on évalue la qualité de l'entrainement du model.

Les courbes ci-dessous montrent l'évolution du MCC train et test au cours des itérations.

Graphiques sur l'évolution du MCC au cours des itérations

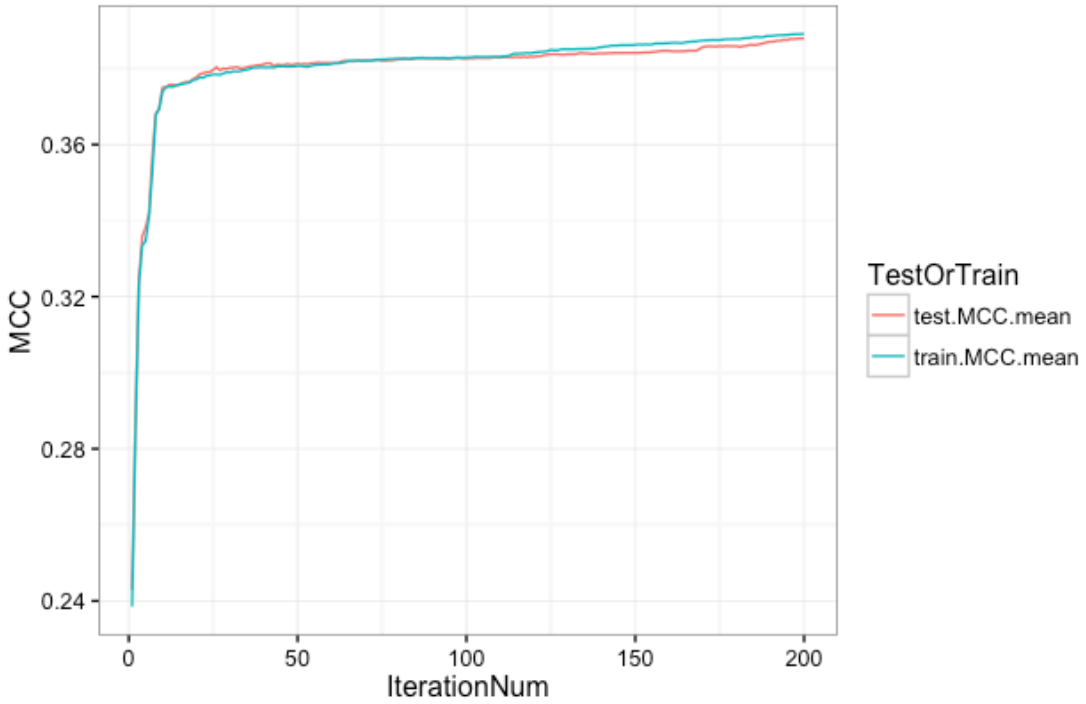

Fig.2 : Évolution la moyenne du MCC au cours des itérations

Ces résultats de l'apprentissage du modèle montrent l'intérêt de la cross-validation. En effet, le sur-apprentissage intervient lorsque l'erreur train est faible tandis que l'erreur de test est élevée. Dans notre problème, le MCC n'est pas une erreur mais un coefficient de performance, plus l'erreur est grande et plus le MCC est faible.

- On constate que la moyenne du MCC test augmente ce qui traduit que l'erreur diminue au cours de itérations. Un palier est atteint au bout d'une quinzaine d'itérations.
- L'écart-type du MCC fluctue "aléatoirement" du fait d'un super-classifieur trop simple qui généralise la prédiction. Si l'on se réfère au du gradient boosting, le super-classifieur est une pondération de classifieurs simples. Il n'y a donc pas assez d'estimateurs sur les 15 premières prédictions. Mais à partir de 15/20 itérations le modèle devient robuste jusqu'à se stabiliser à un écart type du MCC de 0.01. Cela concorde avec le palier atteint par la moyenne du MCC qui se stabilise.

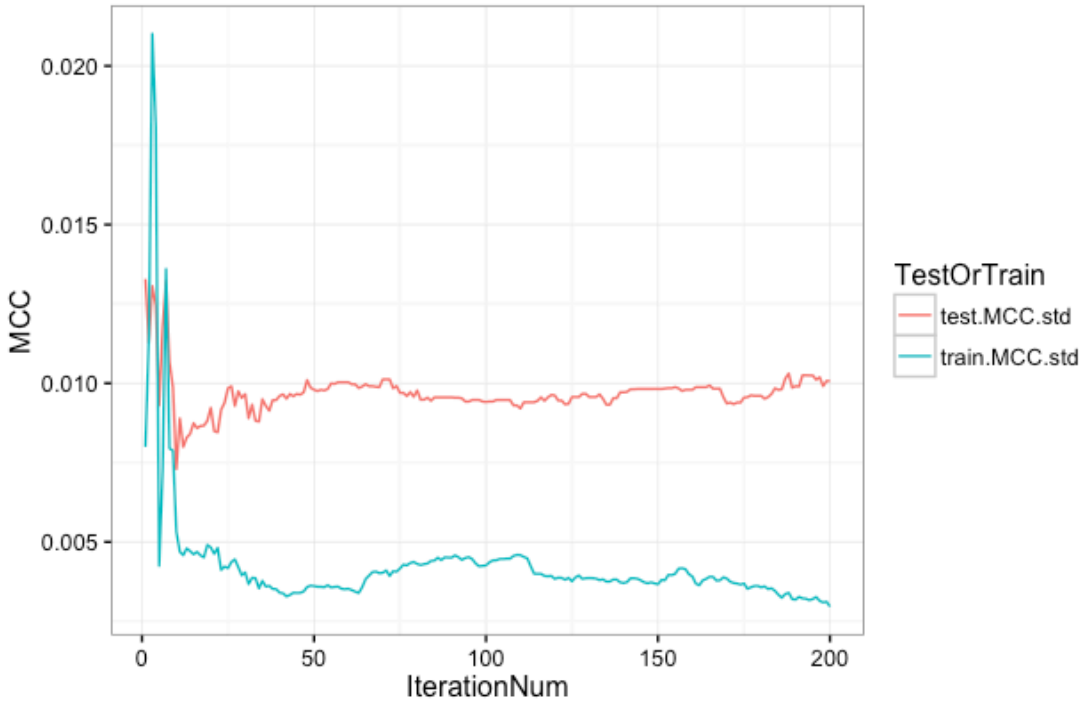

Fig.1 : Évolution de l'écart-type du MCC au cours des itérations

Prediction et treshold

Ce choix permet d'optimiser la prédiction, surtout dans ce cas où les classes sont très déséquilibrées. L'objectif est donc de trouver le seuil ou « treshold » optimal qui permet de maximiser le MCC.

Ça renvoie des booléens donc on multiplie par 1 pour avoir la classe: 0 ou 1.

```
pred <- predict(xgb_model, dvalid)
matt <- data.table(thresh = seq(0.99, 0.9999, by = 0.0001))
matt$scores <- sapply(matt$thresh, FUN = function(x) mcc(Y[valid], (pred > quantile(pred, x)) * 1))
threshold <- matt$thresh[which(matt$scores == max(matt$scores))]

#rm(X)
```

Test

Préparation des données

On prépare de nouveau les données tel que précédemment.

Prédiction

On fait appel à la fonction predict() qui calcule à l'aide de la fonction F définie lors de l'algorithme du gradient boosting:

$$F(\vec{x_i^{test}}) = p(\hat{y_i} = 1/\vec{x_i^{test}})$$

Ecriture du fichier

On met sous forme demandé par Kaggle afin d'évaluer la prédiction sur la plateforme.

Résultats

- Le XGB appliqué aux données avec le featuring mixte (ma sélection de variable, mes choix sur les NA valeurs et l'affectation de score sur la date min de Faron) et avec mon réglage du modèle donne un score de 0.41. Ce qui m'a classé les 15% mais à la fin de l'épreuve je me trouvais dans les 29%.

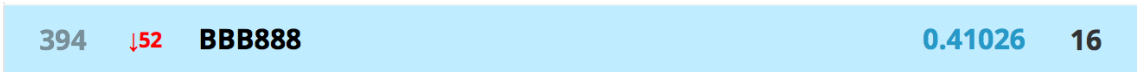

Fig.3 : Imprimé écran du classement

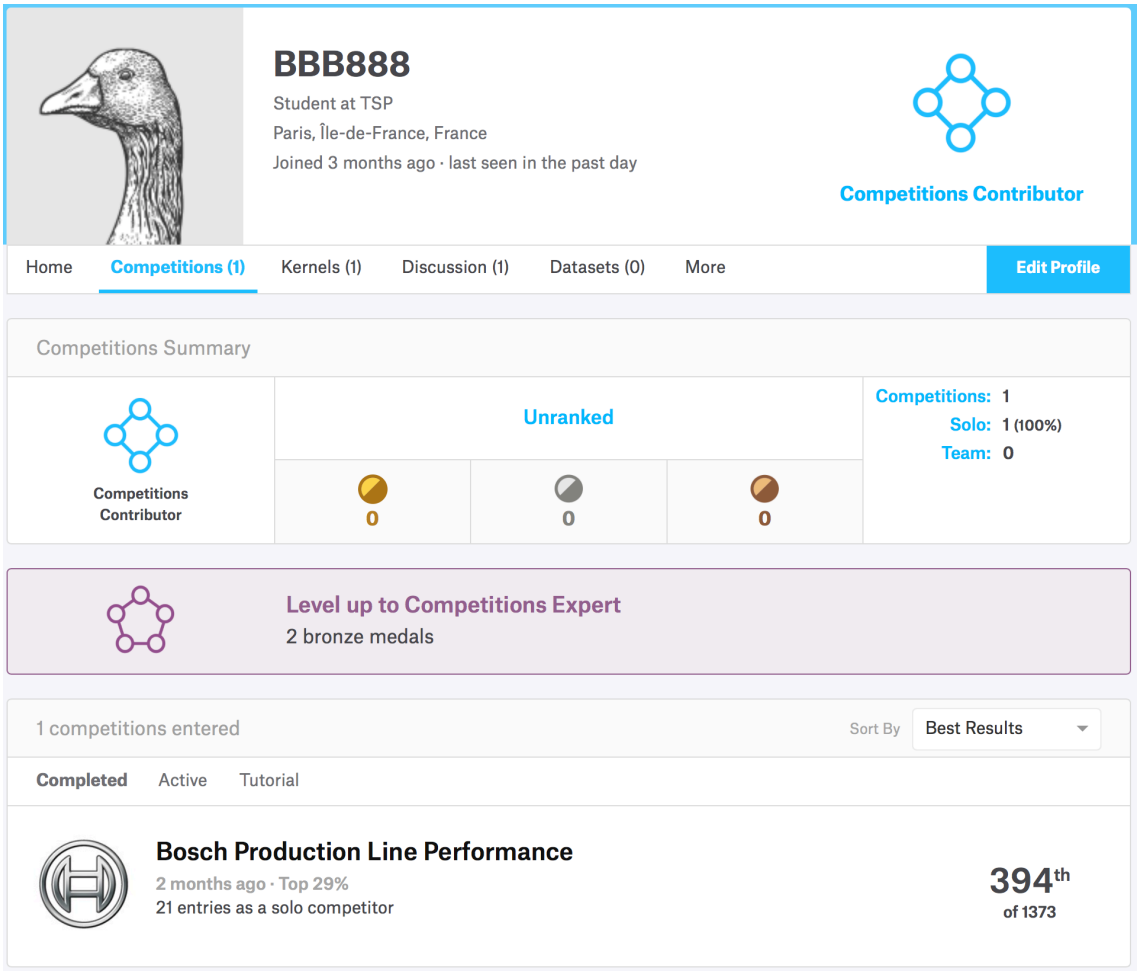

Fig.4 : Page d'accueil Kaggle montrant le classement et le score

Conclusion:

Finalement, l'utilisation du gradient boosting s'est montrée très concluante du fait des résultats obtenus. Ce travail est "classique" dans tous les problèmes de machine-learning. Je pourrai donc réutiliser cette approche sur d'autres compétitions de Data Science. Ce modèle étant performant, il peut donner suite à une agrégation de modèles tel que le Random Forest mixé au XGB.