GLO-4027 - Bike Sharing Demand

Premier traitement des données

17 avril 2019

Étudiants en programme d'échange international:
*Bastien **CHABAL***
*Corentin **GIRAUD***

Projet Kaggle – Repository GitHub

Sommaire

1. Introduction

Le projet que nous avons choisi est intitulé Bike Sharing Demand (plateforme Kaggle). On se propose d'analyser les données du système de partage de vélos de la ville de Washington, D.C.

Le système de partage de vélos au sein d'une ville est très simple : plusieurs kiosques sont répartis dans la ville et des personnes (abonnées ou non) peuvent louer un vélo et faire le trajet qu'elles souhaitent jusqu'à un autre kiosque.

2. Rappel général sur les données

Les données sont réparties en deux fichiers *.csv* distincts:

- Le **set d'entraînement** contient deux ans de données (année 2011 et 2012), où sont relevées toutes les heures différentes informations, dont le nombre total de locations. Ce set ne contient que les données des **19 premiers jours de chaque mois**.
- Le **set de test** contient des données exactement similaires au set d'entraînement, mais pour **tous les jours après le 20 du mois inclus**.

Après une première visualisation des données proposées par le set d'entraînement, on constate qu'il contient 10 886 entrées, réparties sur **12 attributs**. Chaque tuple représente les données pour 1 heure. Tous les attributs ont des valeurs, ce qui signifie qu'il n'y aura pas de prétraitement des données afin de corriger **l'intégralité**.

3. Rappels divers

3.1. Principale transformation des données: l'attribut date

Nous avons découpées l'attribut en plusieurs attributs correspondants chacuns à une unité temporelle. Nos données sont donc maintenant munies des attributs *date*, *hour*, *month*, *year* et *weekday*. Nous avons donc trois types de valeurs différentes :

- Les attributs que l'on nomme *categoricalAttributeNames* : ce sont des entiers qui représentent des catégories (par exemple, season=1 représente le premier quart de l'année).
- les attributs numériques que l'on nomme *numericalAttributeNames* : ce sont des float qui représente une mesure numérique.
- les attributs que l'on nomme *dropAttributes*. On les supprime, on ne souhaite pas conserver dans l'entraînement du modèle car :
    - ce sont les attributs que l'on souhaite prédire (*count, registered, casual*)
    - ce sont des attributs redondants (*datetime*)

3.2. Division des données d'entraînement

On souhaite mettre en place un apprentissage supervisé. Ainsi, on divise *dataTrain* de cette façon :

- 70% pour l'entraînement
- 30% pour la validation

La démarche est la même indépendamment du modèle que l'on souhaite mettre en place :

1. On entraîne le modèle sur les 70% des données
2. On effectue des prédiction sur les 30% restant
3. On compare nos prédictions avec les valeurs réelles (notamment en calculant un score (cf. partie suivante)

3.3. Score RMSLE

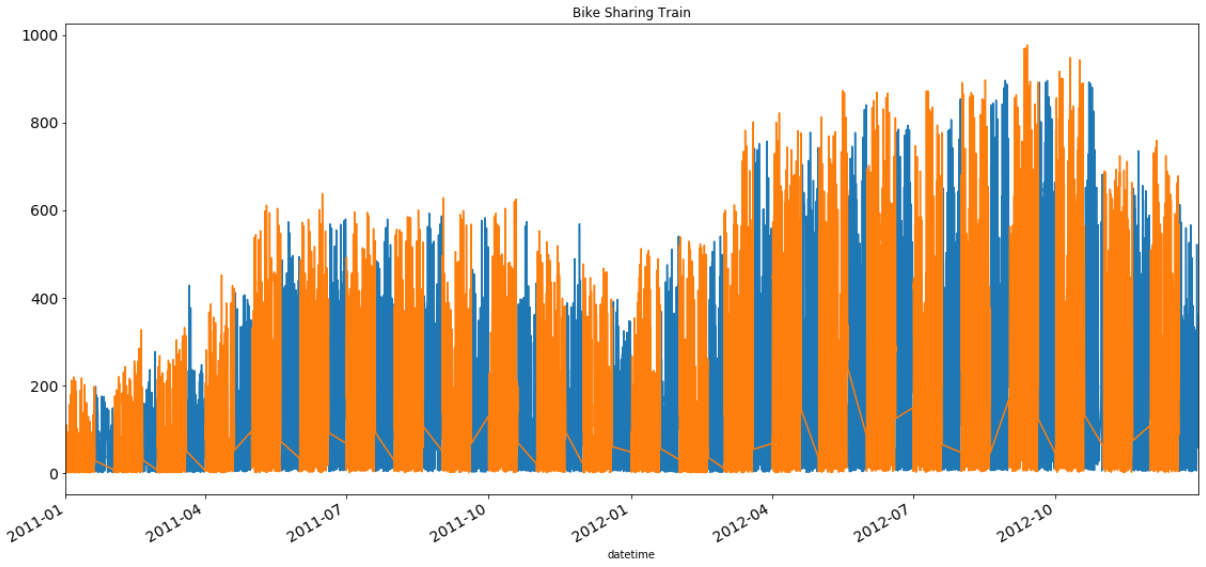
Pour tester nos résultats nous avons défini un *RMSLE scorer* au sein du code qui mesure le ratio entre la valeur prédite est la valeur réelle. D'après nos recherches sur cet indicateur, il est utilisé quand on ne veut pas pénaliser les différences entre deux valeurs (réelle et prédite) élevées.

On constate également que la plateforme Kaggle utilise ce scorer pour calculer le scoreboard de la compétition. Le leader du projet Kaggle obtient un RMSLE de **0.3375**, ce qui peut être un bon référentiel.

3.4. Nos précédentes prédictions

Pour le rendu précédent, après avoir traiter nos données, nous avons décidé appliquer un algorithme *Random Forest* (ou prédiction par forêt aléatoire) pour effectuer des prédictions. Nous avons justifié le choix de ce modèle par le fait que nous disposons d'un large éventail d'attributs qui semblent chacun avoir une importance. Nous pouvons donc dégrossir le problème en une série de questions comme dans un arbre de décision.

Ci dessous, nous pouvons visualiser le graphe complet: en orange les données connues (du 1 au 19 du mois), en bleu les données prédites (du 20 à la fin du mois).



Notre score Kaggle à la fin du rendu précédent est de **0.48108**, ce qui est améliorable.

4. Application d'un nouvel algorithme

4.1. L'approche

Le sujet du TP nous demande clairement d'utiliser une nouvelle approche de résolution du problème avec l'utilisation d'un nouvel algorithme. Nous avons choisi de pousser l'approche des algorithmes qui utilisent une méthode d'agrégation de modèles. L'idée est donc d'entraîner plusieurs modèle et de les agréger en un nouveau qui apprend de chaque modèle. Nous sommes donc passé d'un modèle de RandomForest à un modèle de Gradient Boosting.

Sans plus attendre, nous avons utilisé le modèle grâce à la librairie scikit-learn dans sa forme brute, sans modifier les paramètres par défaut. Le score RMSLE fut moins bons que pour RandomForest.

Nous avons donc creuser le potentiel de cet algorithme en comprenant la signification des différents paramètres qui peuvent l'influencer. En effet, une paramétrisation de l'algorithme a permis d'améliorer les résultats.

Dans cette partie, nous allons expliquer l'algorithme gradient boosting tout en expliquant à l'aide d'exemple associés à notre cas d'étude. Nous terminerons par la présentation des résultats de l'algorithme.

4.2. Les algorithmes ensemblistes

Lorsque nous essayons de faire une prédiction à l'aide d'un algorithme de d'apprentissage, les principales causes de différence entre les valeurs réelles et les valeurs prédites sont le bruit, la variance et les biais. A l'exception du bruit, les algorithmes ensembliste viennent résoudre ces causes de différence.

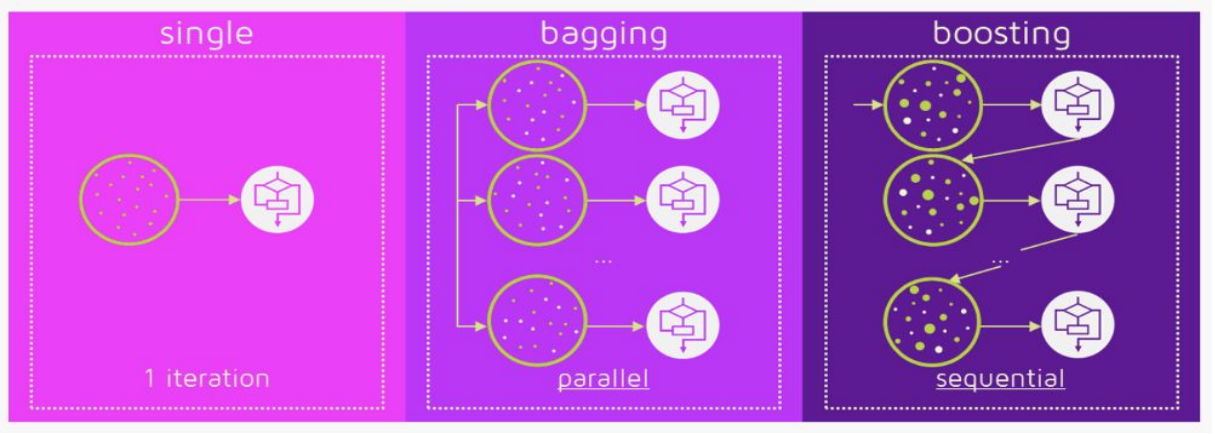
L'idée est très simple: élaborer plusieurs modèles de prédictions différents, en calculer une "moyenne" pour élaborer un modèle unique. Logiquement, plusieurs prédicteurs ensemble seront meilleurs qu'un seul.

4.3. La différence entre Random Forest et Gradient Boosting

Les techniques ensemblistes que les algorithmes du même nom utilisent sont classés en deux catégories:

- *Bagging*: qui consiste à construire plusieurs modèles de prédictions **indépendants** et qui les combine en une sorte de "moyenne" de modèle (moyenne pondérée, moyenne simple …). Un exemple d'implémentation des techniques de bagging est l'algorithme Random Forest que nous avons utilisé pour nos précédentes prédictions.
- *Boosting*: qui consiste à construire plusieurs modèles **séquentiellement** et les agréger. Cela signifie que le modèle calculé à l'étape N-1 pourra influencer le modèle calculé à l'étape N. L'algorithme Gradient Boosting en est une implémentation.

L'illustration suivante résume les différences entre les différentes approches que nous venons de présenter:



Reference: https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/

En plus de venir réduire la variance des prédictions (tout comme les techniques de Bagging), les techniques de Boosting viennent réduire les biais sur nos données. Mais cela peut entraîner un sur-apprentissage qui n'existe pas dans les technique de bagging.

Dans le rapport précédent, nous avions estimé que RandomForest était adaptée au problème car nous disposons d'un large éventail d'attributs qui semblent chacun avoir une importance, et nous pouvons dégrossir le problème en une série de questions comme dans un arbre de décision.

Ainsi, l'utilisation de Gradient Boosting s'inscrit dans la continuité de cette réflexion : il utilise les arbres de décision et se "corrige" avec chacun d'eux.

Enfin, il existe une différence majeure entre ces deux modèles : leur simplicité d'utilisation. Grâce à scikit-learn, mettre en place une RandomForest est assez aisée. Cependant, le modèle de Gradient Boosting est doté de nombreux paramètres. Bien sûr, on peut automatiser la recherche des paramètres optimaux, mais il faut se souvenir que la construction des arbres est séquentielle : cette recherche peut donc prendre plusieurs heures !

4.4. L'algorithme Gradient Boosting

L'algorithme Gradient Boosting est principalement utilisé avec les arbres de décision (et plus spécifiquement des arbres CART, que nous ne détaillerons pas dans le présent rapport). Il s'agit d'un **algorithme supervisé**. Cela signifie qu'il est basée sur une fonction de perte (*loss function*) et qu'il essaye à travers plusieurs itérations de la réduire.

Imaginons que nous avons une fonction MSE (*mean squared error*) définie comme notre fonction de perte:

$$Fonction\ de\ perte\ =\ MSE\ =\ \sum(valeursRéelles\ -\ valeursPrédites)^2$$

Nous pouvons alors calculer les valeurs pour lesquels notre fonction de perte est minimale:

$$valeursAjustées\ =\ valeusrPrédites\ +\alpha\ *\ f(valeursRéelles,\ valeursPrédites)$$

La fonction f étant environ égal au gradient notre $Fonction\ de\ perte$:

$$f(valeursRéelles,\ valeursPrédites)\ \approx\ gradient(Fonction\ de\ perte)$$

$f(valeursRéelles,\ valeursPrédites)\ \approx (valeursRéelles\ -\ valeursPrédites)$ dans le cas de notre fonction de perte MSE.

On voit ainsi que les valeurs finales utilisées pour la prédictions sont le résultats de plusieurs itérations de calcul de modèle d'arbres de décisions sur les données. Les valeurs finales ajustées utilisées pour la prédiction suivent une **descente graduelle** de la fonction de perte multipliée par un taux d'apprentissage α.

Pour éviter le problème de sur apprentissage, on peut utiliser les techniques suivantes:

- On peut fixer une taille limite pour les arbres
- On peut aussi utiliser des échantillons des données pour construire les modèles (on parle alors de stochastic gradient boosting)

En conclusion, on voit que l'algorithme ensembliste Gradient Boosting est très paramétrable et que chaque paramètre peut influencer grandement les résultats qu'il produit. La partie suivante se concentre sur l'application de l'algorithme Gradient Boosting à notre problème et les résultats que nous obtenons.

4.5. Résultats et paramètres que nous avons utilisés

La librairie *scikit-learn* écrite pour le langage Python nous a beaucoup aidé pour l'implémentation de l'algorithme Gradient Boosting.

Dans un premier temps, la classe *GradientBoostingRegressor* implémente une variante très proche de l'algorithme Gradient Boosting. Dans un second temps, la fonction *GridSearchCV* nous a permis d'automatiser nos tests sur les paramètres de l'algorithme *GradientBoostingRegressor* et de choisir le meilleur.

Mais pour évaluer quels sont meilleurs paramètres de l'algorithme, la fonction *GridSearchCV* a besoin d'un estimateur d'erreur et d'une stratégie de réarrangement des données afin d'assurer le fait que chaque portion de nos données soit le plus représentatives de la globalité. Nous avons opté pour un division en 5 portions en utilisant une validation stratifiée (de base dans la librairie). En augmentant le nombre de portions, on applique plus de validations à notre modèle et donc un meilleur calcul de ces erreurs. On augmente encore le temps de calcul mais on améliore les résultats!

Dans nos premiers tests, nous avons utiliser un nombre d'itération de boosting (avec la fonction de perte *GradientBoostingRegressor)* de 100. Nous avons choisi d'augmenter ce nombre à 1000. L'algorithme étant peu sensible au problème de sur apprentissage, les résultats sont généralement meilleurs. Nous avons ensuite poussé cette valeur plus loin et nous nous sommes rendu compte que les résultats étaient moins bons. Dans notre notebook Jupyter, notre paramètre se nomme *n_estimators*.

Nous avons aussi faire des tests sur les paramètres de la profondeur des arbres (*max_depth*) des modèles, du taux d'apprentissage α (*learning_rate)*.

La fonction *GridSearchCV* permet d'afficher les meilleurs paramètres à appliquer à l'algorithme Gradient Boosting (via l'attribut *best_params_* du retour de la fonction).

Malgré la multitude de paramètre modifiables, nous n'avons modifié que l'attribut *n_estimators* : il s'agit du nombre d'arbre que va créer le modèle. Par défaut à 100, la recherche des paramètres optimaux à révéler que ce chiffre devrait plutôt s'approcher de 1000.

Nous nous sommes focalisés sur cet attribut car sa simple modification donnait un résultat meilleur que RandomForest. Mais comme dit précédemment, une recherche globale des paramètres optimaux peut être automatisée facilement, bien qu'elle prendrait quelques heures de travail au CPU de notre ordinateur.

5. Une autre idée de solution: la division en deux sous problèmes

Nous pouvons avoir une autre approche sur le problème. En effet, lors de l'étape d'analyse des données, nous nous sommes rendus compte que le problème pouvait se séparer en deux sous problèmes. Nous pouvons faire une prédiction sur l'attribut du nombre de locations enregistrés (*registered*) et une autre prédiction sur l'attribut du nombre de location non-enregistré (*casual*) puisque ces variables ont un comportement très différents. La prédiction finale (le nombre de locations) n'est juste que l'addition de ces deux propriétés.

Notre démarche était de découvrir si Random Forest avait un meilleur RMSLE sur registered ou casual que Gradient Boosting et inversement. S'ils étaient chacun meilleur pour un des attributs, on aurait entraîné deux modèles différents puis fait leur addition pour avoir le *count* final.

Cette étape nous a permis de découvrir que c'est l'attribut *casual* qui fait augmenter le RMSLE final :

- Une RandomForest au RMSLE de **0.356** sur *count* a un RMSLE de **0.335** sur *registered* et **0.550** sur *casual*.
- Un GradientBoosting au RMSLE de **0.300** sur *count* a un RMSLE de **0.296** sur *registered* et **0.524** sur *casual*.

On constate également que GradientBoosting est meilleur sur les deux attributs. Cependant, cela ne veut pas dire qu'il ne faut pas diviser le problème. Comme nous l'avons déjà dit, GradientBoosting étant très paramétrable, il peut s'adapter à la prédiction de ces deux attributs. On peut alors imaginer une prédiction finale composée de deux GradientBoosting qui travaillent sur ces deux attributs et dont on fait la somme.

6. Conclusion

À travers ce projet, nous avons pu nous familiariser un problème de classification de données que nous avons souhaité étudier avec les arbres de décision. À partir d'une simple itération pour définir un modèle, nous avons poussé l'analyse plus loin en explorant les algorithmes d'aggrégations de modèles (algorithmes ensembliste).

Grâce au langage Python et à ses très riches libraires de *machine learning*, nous avons suivi les principales étapes pour l'étude d'un problème concret de prédictions: la compétition Bike Sharing Demand sur la plateforme Kaggle. Nous avons implémenté les algorithmes Random Forest et Gradient Boosting pour améliorer progressivement nos prédictions et notre score Kaggle.

Bien que ces librairies facilitent la mise en place des modèles, le pré-traitement des données et l'ajustement des paramètres dépendent du programmeur. Cela nous a fait prendre conscience de la difficulté pour un analyste de produire des algorithmes compréhensibles, par lui-même et par d'autres. En effet, les algorithmes ensembliste et le *machine learning* en général ont un effet boîte noire qui est intimidant.

Enfin, ce projet est une découverte des outils d'analyse et de prédiction des données et de nombreux éléments restent à améliorer. Un travail plus approfondi sur casual et registered, l'utilisation des technologies du cloud pour résoudre les problèmes de temps de calcul… Il reste beaucoup à explorer et nous avons hâte de le faire après ce projet.

Références

https://medium.com/@viveksrinivasan/how-to-finish-top-10-percentile-in-bike-sharing-demand-competition-in-kaggle-part-1-c816ea9c51e1

https://medium.com/@viveksrinivasan/how-to-finish-top-10-percentile-in-bike-sharing-demand-competition-in-kaggle-part-2-29e854aaab7d

https://www.kaggle.com/general/21582

https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d

https://en.wikipedia.org/wiki/Random_forest

https://www.analyticsvidhya.com/blog/2015/06/solution-kaggle-competition-bike-sharing-demand/

https://scikit-learn.org/stable/

https://lovelyanalytics.com/2016/09/12/gradient-boosting-comment-ca-marche/

https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d

https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/

https://medium.com/all-things-ai/in-depth-parameter-tuning-for-gradient-boosting-3363992e9bae