XGBoost
Origines et applications

Rapport de veille technologique

Damien DOUTEAUX — @DDouteaux

Mars
25
2017

MOS 4.4 – Nouvelles Technologiques de l'IC

ÉCOLE CENTRALE LYON

1. XGBoost en deux mots

L'objectif de cet article est de présenter XGBoost. Mais qu'est-ce que XGBoost ? Il s'agit d'une méthode de machine learning apparue il y a trois ans et dérivant des méthodes dites de boosting. Cette méthode est présentée par ses créateurs comme étant :

- Flexible Prise en compte de plusieurs thématiques de machine learning.
- Portable Utilisable sous toutes les plateformes (Windows, Linux, MAC).
- Multi-langages L'algorithme a été porté en Python, JAVA (Spark), C++,...
- Distribuée Depuis deux ans, l'algorithme est utilisable avec Hadoop et Spark.
- Perfomante Cet algorithme est donné pour être plus rapide que les algorithmes de sa famille et fournir de même meilleurs résultats.

Mais avant toute chose, que veut exactement dire l'acronyme « XGBoost » ?

E**X**treme **G**radiant **Boost**ing

Ainsi, la méthode XGBoost s'organise autours de trois points essentiels :

- *Boosting* Il s'agit d'une famille d'algorithme utilisés initiallement en apprentissage supervisé. Le principe du boosting sera détaillé en Section 2.2.
- *Gradient Boosting* Il s'agit d'une version du boosting dans laquelle l'objectif sera d'optimiser une fonctione faisant apparaître des gradients. Les détails de cette idée seront détaillées en Section 2.5.
- *« Extreme »* Ce qualificatif signifie que la recherche de performances est poussée au maximum pour cette méthode, comme nous l'étudierons en Section 3.2.

Ainsi, nous allons dans un premier temps présenter les grandes lignes théoriques derrière cet algorithme avant de partir dans l'étude des implémentations et des applications de XGBoost.

Nous nous intéresserons également aux aspects plus techniques de XGBoost, en regardant en détails les principaux paramètres de cette méthode d'apprentissage.

Il sera également intéressant de regarder quels sont les usages réels de cette méthode, afin de nous forger un avis sur sa pérennité.

Tous les supports présentés en séance ainsi que le code du site web associé à ce rendu peuvent être trouvés sur ma page Github : `https://github.com/DDouteaux/XGBoost_Veille_Douteaux/`*. N'hésitez pas non plus à retrouver le compte Twitter de cette veille, @DDouteaux.*

En bref

2. Le boosting

2.1 • Qu'est-ce que le boosting ?

Le boosting est une méthode de Machine Learning apparue à la fin des années 1980 et ayant évoluée au fil du temps en plusieurs version, les principales étant AdaBoost ou encore les GBM (*Gradient Boosted Models*)[1].

Le succès de ces méthodes provient de leur manière originale d'utiliser des algorithmes déjà existant au sein d'une stratégie adaptative. Cette stratégie leur permet alors de convertir un ensemble de règles et modèles peu performant en les combinant pour obtenir de (très) bonnes prédictions. L'idée principale est en effet d'ajouter de nouveaux modèles au fur et à mesure, mais de réaliser ces ajouts en accord avec un critère donné. En ce sens, cette famille de méthodes se différencie des Random Forest qui vont elles miser sur l'aléatoire pour moyenner l'erreur.

Cet aspect fondamental du boosting permet ainsi une forte réduction du biais et de la variance de l'estimation, mais surtout garanti une convergence rapide. La contrepartie se fait au niveau de la sensibilité au bruit comme nous le verrons dans la description des algorithmes.

[1] Historique du Boosting
1989
Proposition de la méthode et premier algorithme par R. SCHAPIRE
1996
Première implémentation d'AdaBoost par Y. FREUND et R. SCHAPIRE
1999
Apparition du Boosting de gradient (GBM) par L. BREIMAN et J. FREIDMAN
2014
Apparition et implémentation de XGBoost par T. CHEN

2.2 • Premier algorithme

Pour commencer, nous allons présenter l'idée originale de l'algorithme de Boosting présenté par R. SCHAPIRE en 1989. Comme nous l'avons précisé, l'idée est de construire de « mauvais » classifieurs au fur et à mesure qui combiné fourniront un excellent classifieur.

Pour nos exemples, nous allons considérer la situation de la Figure 1. Le problème a traiter ici est un problème de classification supervisé, c'est-à-dire que l'on désire entraîner un classifieur pour pouvoir séparer les carrés des cercles.

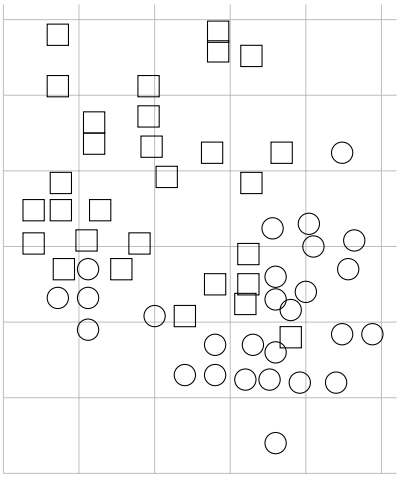
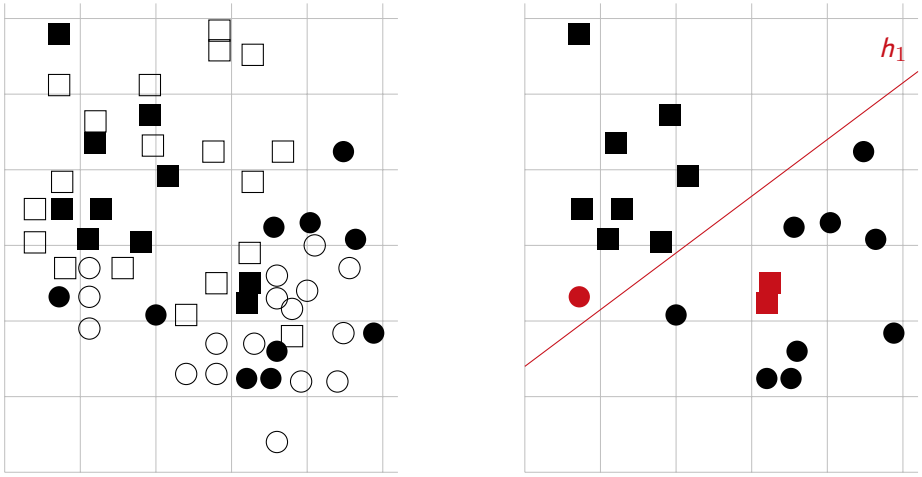


FIGURE 1 • Deux ensembles à séparer par le modèle que l'on entraîne (d'après [1])

Pour commencer, l'idée est d'entraîner un premier classifieur. Cependant, pour retirer une partie du biais des données, ce dernier ne sera entraînée que sur une partie des données (choisie arbitrairement). Nous retrouvons ainsi en Figure 2a l'ensemble d'apprentissage qui est en rouge, et en Figure 2b le classifieur appris sur ces données, notés h_1.

Sans surprise, ce classifieur n'est pas parfait et réalise un certain nombre d'erreurs (indiquées en rouge sur la Figure 2b). Toute l'idée du Boosting



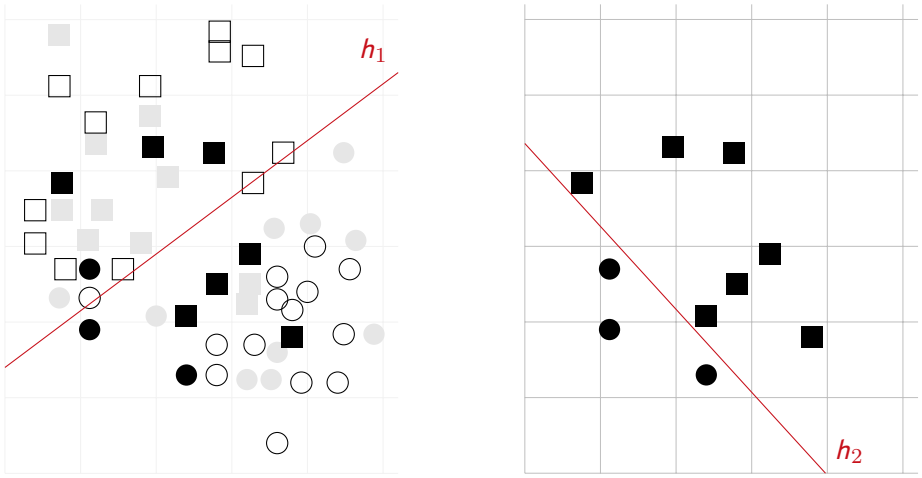
(a) Données d'apprentissage choisies aléatoirement (■ ●)

(b) Classifieur h_1 obtenu par entraînement sur l'ensemble ■ ●

FIGURE 2 • Premier modèle appris sur les données (d'après [1])

est alors de créer de nouveaux modèles en changeant l'ensemble d'apprentissage de manière « intelligente ».

Dans notre cas, nous allons choisir un nouvel ensemble d'apprentissage qui regroupe des éléments n'étant pas dans l'ensemble initial et pour lesquels les prévisions de h_1 sont équiprobablement correctes et incorrectes. Cet ensemble est représenté sur la Figure 3a. On apprend alors à partir de cet ensemble un nouveau classifieur h_2 présenté à la Figure 3b.



(a) Données d'apprentissage pour le deuxième classifieur (■ ●)

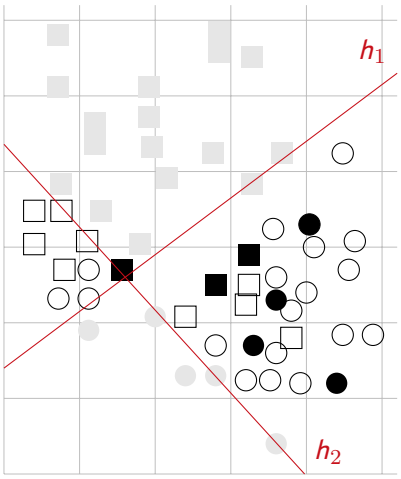
(b) Classifieur h_2 obtenu par entraînement sur l'ensemble ■ ●

FIGURE 3 • Deuxième modèle appris sur les données (d'après [1])

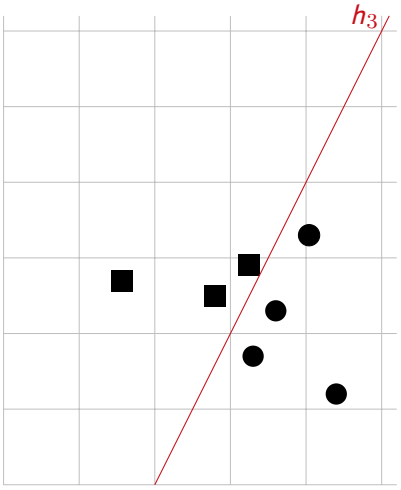
On remarque qu'ici le modèle h_2 ne fait pas d'erreurs sur son ensemble d'apprentissage (qui est cependant peu fourni), ceci est un cas particulier et n'est pas automatique. À cette étape, nous disposons donc de deux modèles, le second permettant de « rattraper » des erreurs du premier.

Nous continuons alors sur le même principe, en entraînant un troisième classifieur. Son ensemble d'apprentissage sera choisit dans les points n'ayant pas encore servis à l'apprentissage, et pour lesquels les deux autres classifieurs sont en désaccord, voir la Figure 4a. Le troisième classifieur, noté h_3 est représenté à la Figure 4b.

À la fin de ce processus (qui aurait pu durer plus longtemps si nous



(a) Données d'apprentissage pour le deuxième classifieur (■ ●)



(b) Classifieur h_2 obtenu par entraînement sur l'ensemble ■ ●

Figure 4 • Troisième modèle appris sur les données (d'après [1])

l'avions désiré), nous obtenons trois classifieurs. Comme cela a été vu, ces derniers ont été créés de manière à ce qu'ils corrigent leurs erreurs entre eux, ce paramètre étant pris en compte par la construction des ensembles d'apprentissage. Ainsi, l'utilisation de tous ces classifieurs revient pour chaque point à réaliser une prédiction avec tous les classifieurs et d'attribuer la classe ayant le plus de vote. On moyenne ainsi les erreurs des classifieurs entre eux et on réduit ainsi le biais de la prédiction. Cette décision finale est illustrée à la Figure 5, où les erreurs sont précisées en rouge.

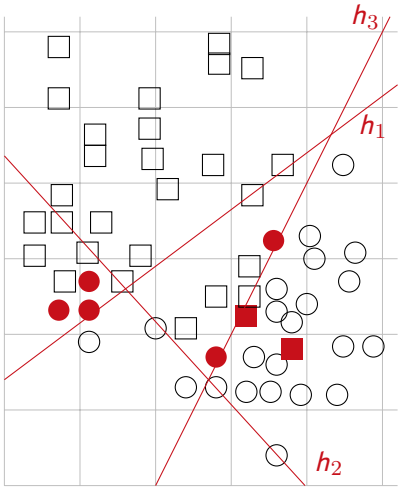


Figure 5 • Prévision finale conjointe des trois classifieurs entraînés par Boosting (d'après [1])

Cette première version de l'algorithme de Boosting a subit de nombreuses améliorations jusqu'à aujourd'hui, les plus notables étant :

- *Adaboost* Cette version de l'algorithme change la manière de construire l'ensemble d'apprentissage. Au lieu de prendre des sous-ensembles d'apprentissage, cette méthode propose de pondérer très fortement les éléments mal classés, et dans le même temps de pondérer faiblement les éléments bien classés. L'idée est alors qu'à chaque étape on va principalement apprendre sur les données en erreurs, et contrebalancer les votes précédents. Cette méthode va donc se focaliser sur les cas « difficiles » de l'ensemble d'apprentissage. Si cette solution permet de converger très rapidement et avec grande précision, on voit également qu'elle va s'ef-

forcer au maximum de bien classer tous les éléments, y compris ceux étant abhérents. Ainsi, cette méthode va augmenter l'improtance des exemples mal-classés, isolés (outliers) ou encore mal-enregistrés.

- *Gradient Boosting Machine (GBM)* Une version un peu remaniée de ces derniers sera abordée à la Section 2.5.

2.3 • Perte et complexité

Avant d'aborder la question des GBM, nous allons nous intéresser à deux grands concepts importants en machine learning et qui sont tout particulièrement gardés à l'œil dans le cas de XGBoost et du boosting de manière générale.

Nous avons vu que l'idée du Boosting est d'adapter l'ensemble d'apprentissage à chaque étape pour chercher des classifieurs qui vont se compléter, ie. moyenner leurs erreurs. Il s'agit en fait de « coller aux données » du mieux que possible (mais sans sur-apprendre !). Nous verrons que cet aspect est lié à la notion de perte.

De même, nous avons vu que l'idée n'est pas d'apprendre des modèles intermédiaires très puissants, mais que l'ensemble soit performant. Ainsi, nous verrons que le fait de prendre des modèles simples est lié à la notion de complexité.

2.4 • Fonction de perte

Avant d'aller plus loin, regardons formellement quelle est l'idée théorique derrière le Boosting. Ce dernier cherche en fait à optimiser (minimiser) une fonction objectif. Cette fonction peut s'écrire en deux termes :

$$\text{objectif}(\Theta) = \mathcal{L}(\Theta) + \Omega(\Theta) \tag{1}$$

La relation (1) fait apparaître deux termes [2] :

- $\mathcal{L}(\Theta)$ La fonction de perte
- $\Omega(\Theta)$ La fonction de complexité

Ce sont ces deux fonctions (et leur intérêt !) qui vons nous intéresser par la suite. Pour cela, nous allons considérer la situation cobaye de la Figure 6.

[2] Dans la relation (1), la variable Θ représente juste le modèle pour lequel on fait les calculs.

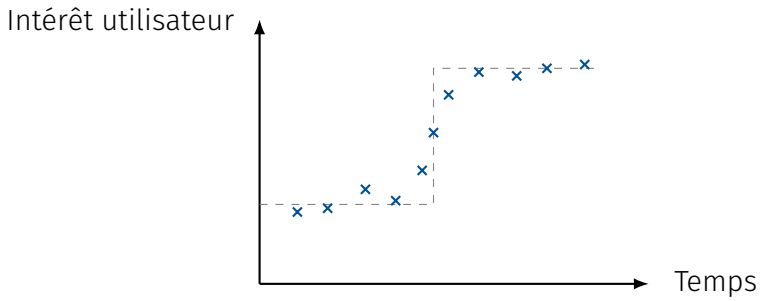


FIGURE 6 • Situation de test pour comprendre les notions de perte et complexité (d'après [2])

Sur la Figure 6, nous avons également représenté en - - - - un modèle qui représenterait un bon compromis.

2.4.1 *Notion de perte*

Cette fonction (notée $\mathcal{L}$ dans la relation (1)), représente la qualité prédictive du modèle sur l'ensemble d'apprentissage. Attention, il est important de voir qu'on s'intéresse ici à l'adéquation à l'ensemble d'apprentissage,

mais pas sur l'ensemble de test (puisqu'on ne le connaît pas à priori). Les fonctions communément utilisées sont par exemple l'erreur MSE (liée à la norme $\mathcal{L}_2$) ou une expression de perte logistique.

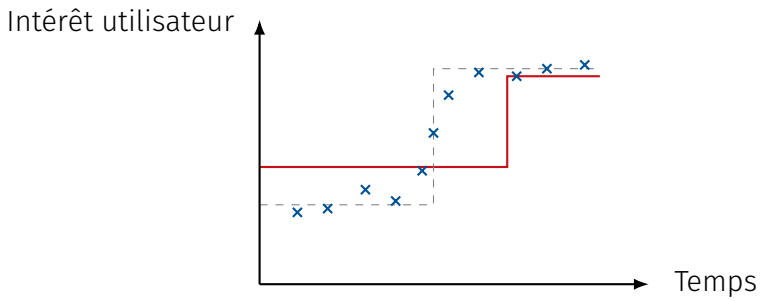


FIGURE 7 • Une situation où la perte n'est pas optimisée (d'après [2])

La Figure 7 propose une situation où la perte induite pas le modèle est plus importante que ce qu'il est possible de dans le cas optimal. L'idée de cette fonction $\mathcal{L}$ est donc de pouvoir détecter ces situations et d'estimer si un nouveau modèle améliore ou non cette situation.

2.4.2 *Notion de complexité*

Cette fonction (notée Ω dans la relation (1)), est aussi appelée terme de régularisation. Elle permet de représenter la complexité du modèle. L'idée est ici qu'un modèle trop précis (*overfitting*) sera généralement trop complexe. Ainsi, ce terme permettra de contrôler la complexité du modèle pour éviter les phénomènes d'*overfitting* sur les données d'apprentissage et éviter une chute importante de la qualité prédictive du modèle sur l'ensemble d'apprentissage.

Un exemple de fonction de complexité vous sera proposé lors de la Section 2.5.

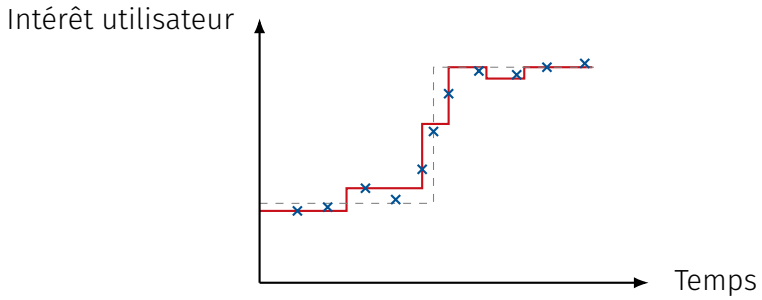


FIGURE 8 • Une situation où la complexité n'est pas optimisée (d'après [2])

Le modèle qui était représenté en gris sur les Figures 6 à 8 correspondait en fait à un bon compromis entre $\mathcal{L}$ et Ω.

2.5 • Gradient Boosting et arbres

Désormais que nous venons de voire l'importance de la perte et de la complexité engendrées par les modèles, nous allons voir comment les utiliser dans le cas du gradient boosting appliqué aux arbres.

2.5.1 *Modèles à base d'arbres*

Pour cette partie, nous considérerons un exemple proposé par les concepteurs de XGBoost et repris dans la documentation [2] et [3]. On considère donc la situation où on essaie de créer un modèle pour savoir si les membres d'une famille apprécient les jeux vidéos.

Les modèles qui seront pris en exemple sont des modèles à base d'arbre, car ils seront plus simple à visualiser et expliquer.

Un modèle pouvant convenir et obtenu par exemple avec la méthode CART est représenté à la Figure 9.

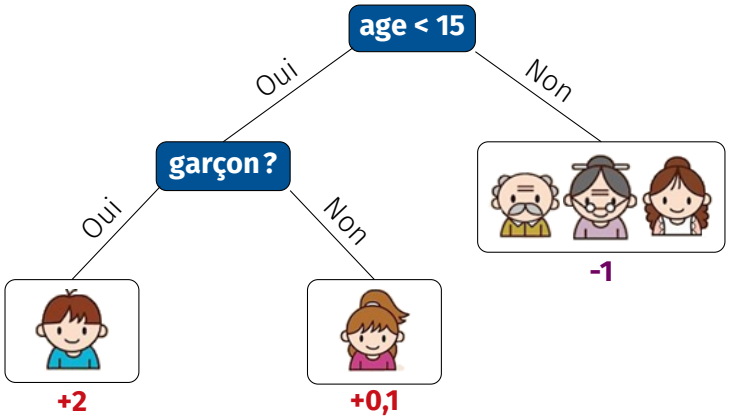


FIGURE 9 • Arbre pouvant convenir pour expliquer la goût des jeux vidéos dans la famille (d'après [3])

Ce modèle nous apprend par exemple que des garçons de moins de 15 ans devraient avoir un fort attrait pour les jeux vidéos, ce qui n'est par exemple pas le cas des individus de plus de 15 ans. Pour réduire le taux d'erreur et obtenir de meilleurs résultats, on peut comme nous l'avons vu à la Section 2.2 utiliser plusieurs de ces modèles simples et les combiner.

Par exemple, à la Figure 10, nous avons entraîné deux arbres, et les résultats en prédiction consistent à sommer les prédictions (scores) obtenus avec chaque arbre. Ceci permet d'obtenir des résultats plus sûrs et de combiner plus d'informations qu'un seul modèle ne le permettrait.

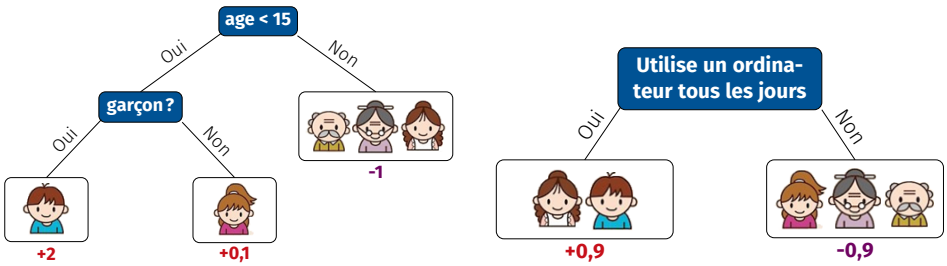


FIGURE 10 • Deux arbres utilisés conjointement pour obtenir le résultat (d'après [3])

Avec les deux arbres de la Figure 10, on peut alors calculer une nouvelle prédiction[3] :

$$f\left(\text{  }\right) = 2 + 0,9 = 2,9 \qquad f\left(\text{  }\right) = -1 - 0,9 = -1,9$$

Pour construire tous les arbres nécessaires, on pourrait fonctionner comme dans la Section 2.2. Cependant, ceci permettrait de bien optimier la partie coût des erreurs (fonction $\mathcal{L}$ de la fonction objectif), mais en aucun cas de limiter la complexité finale. Nous allons donc voir dans la suite un algorithme permettant de prendre en compte ses deux aspects simultanément.

2.5.2 *Ajouter des arbres en optimisant le coût*

L'objectif est de construire K arbres. Pour cela, la méthode retenue est dite additive, dans la mesure où nous allons ajouter un arbre par itération de l'algorithme. L'objectif est alors d'optimiser l'arbre ajouté à chaque

[3] La formule théorique donnant la prédiction $\hat{y}_i$ est $\hat{y}_i = \sum_{k=1}^{K} f_k(x_i)$, où on utilise K arbres notés chacun f_k. L'individu pour lequel on cherche une prédiction est noté x_i.

étape. Quand on passe de l'étape $t-1$ (ie. avec $t-1$ arbres) à l'étape t, l'objectif devient :

$$\text{objectif}^{(t)} = \sum_{i=1}^{n} \ell\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + \text{constante} \qquad (2)$$

La fonction ℓ représente la perte entre la théorie (y_i) et la prédiction à l'étape t qui est obtenue en ajoutant la prédiction de l'arbre f_t. Une solution courrante est de prendre l'erreur MSE pour ℓ.

Dans le cas de l'erreur MSE, la relation (2) peut s'écrire « simplement », ce qui n'est pas toujours le cas. Ainsi, l'usage veut que l'on utiliser un développement de Taylor pour remanier la relation (2), on obtient alors la relation (3)[4].

$$\text{objectif}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \qquad (3)$$

Quelle est la particularité de la relation (3) ? Il s'agit de sa seule dépendance en les dérivée de la fonction de perte, il est donc simple d'utiliser des fonctions de perte particulières sans devoir changer tout le code. Ces dérivées sont donc tout simplement des paramètres du modèle final. Un exemple d'utilisation de ces fonctions de coûts personnalisées sera proposé à la Section 7.3.

[4] Dans la relation (3), la fonction g_i représente la dérivée première de ℓ et h_i sa dérivée seconde selon $\hat{y}_i^{(t-1)}$.

2.5.3 *Optimiser la complexité*

Dans la relation (3), nous avons déjà traité la question de la fonction de coût, reste à traiter le cas de la complexité du modèle (aussi appelée régularisation). Si l'on note ω_j le score prévu par la feuille j de l'arbre, une expression de la complexité Ω courrament utilisée est alors[5] :

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2 \qquad T \text{ est le nombre de feuilles de } f_t \qquad (4)$$

On remarque que la relation (4) a la particularité de proposer deux nouveaux paramètres :

- γ Ce paramètre permet de porter plus ou moins l'accent sur le nombre de feuilles de l'arbre.
- λ Ce paramètre permet de porter plus ou moins l'accent sur les poids des feuilles, pour éviter qu'ils prennent des valeurs irréalistes.

[5] Cette formule, assez simple à analyser tire sa légitimité de ses performances jugées (très) bonnes en pratique. Bien entendu, d'autres expressions seraient possibles !

2.5.4 *Choix de l'arbre*

Nous savons désormais quelle fonction optimiser. Reste à savoir comment filtrer les arbres en fonction de cette formule.

Solution naïve La solution naïve consisterait à envisager *toutes* les structures d'arbres possibles. Une fois cette énumération réalisée, on calculer la fonction objectif pour chacune de ces structures et on trouve ensuite celle fournissant la meilleure valeur pour l'objectif. Il resterait alors à optimiser les poids des feuilles pour cette structure. Des formules adaptées à ce cas existent et sont détaillées dans [3].

Construction au coup par coup Cependant, on se rend compte qu'en pratique cette solution est coûteuse en temps de calcul et n'est donc

pas réaliste. La démarche proposée à la place est alors de construire les arbres au coût par coût.

L'idée est alors de commencer à la racine. À partir de cette dernière, on énumère tous les découpages possibles, et on garde celui de meilleur gain. On fait ensuite de même pour le fils gauche et le fils droit et ainsi de suite. Ce algorithme nécessite donc de calculer un gain pour les découpages proposés. Dans le nouvel arbre, chaque individu va être associé (comme auparavant) à un g_i et un h_i. Le gain s'écrit alors[6] :

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma \tag{5}$$

Les termes de l'expression (5) sont dans l'ordre :

- Le score de la nouvelle feuille gauche.
- Le score de la nouvelle feuille droite.
- Le score de la feuille initiale (celle que l'on découpe).
- Un terme de régularisation sur le nombre de feuilles, voir (4).

Une première remarque est que le gain peut être négatif. Là où des méthodes d'élagage arrêteraient l'algorithme, l'idée pour XGBoost est de continuer autant que possible (jusqu'à épuisement des instances ou que l'on ait atteint une profondeur/nombre de feuilles demandés). Procéder ainsi, peut par exemple permettre de retrouver des découpage très intéressants que nous aurions éventuellement pu oublier si on coupait au premier gain négatif.

Un exemple de découpage et une partie des calculs associés vous est présenté à la Figure 11.

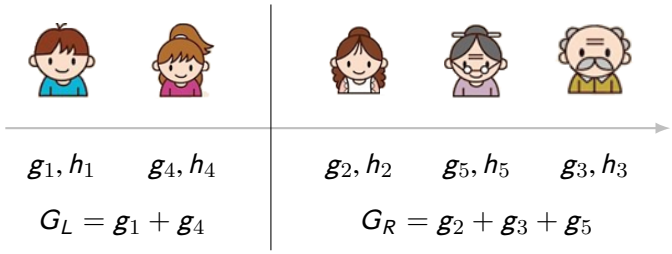


FIGURE 11 • Comment déterminer le meilleur découpage à une étape donné (d'après [3])

En pratique, on ne teste même pas tous les découpages possibles. À la place, on procède comme à la Figure 11, ie. on ordonne les instances par valeur attendue (le y_i théorique). On regarde alors juste les découpages obtenus en coupant entre les éléments de la gauche vers la droite[7].

En résumé, l'algorithme revient à construire les n arbres un à un. Ces derniers sont alors construits au coup par coup en calculant les gains des différents découpages possibles. Les arbres sont ensuite élagués après avoir été totalement construits. On optimise ainsi à chaque étape de l'algorithme une fonction de objectif influant sur la qualité du modèle (fonction de coût/perte) aussi bien que sur sa complexité.

[6] Les termes indexés par L représentent des sommes sur les g_i ou les h_i pour la feuille de gauche. Ceux indexés par R représentent des sommes pour la feuille de droite. Les cœfficients λ et γ correspondent à ceux présentés Section 2.5.3.

[7] On regarde ainsi uniquement n_l découpages possibles, où n_l est le nombre d'éléments de la feuille, ce qui est bien mieux que de regarder tous les découpages possibles, c'est-à-dire $\approx \sum_{i=1}^{\left\lfloor \frac{n_l}{2} \right\rfloor} \binom{i}{n_l}$ solutions.

3. Plus que du boosting

Nous venons de voir les idées générales et les algorithmes de base derrière XGBoost. En particulier, nous avons vu que cette méthode porte une attention toute particulière à la régularisation, ie. la complexité du modèle. Nous allons désormais voir d'autres éléments annexes donnant toute sa puissance à cette méthode.

3.1 • Importance des variables

Dans un premier temps, XGBoost permet à l'instar des autres méthodes de Boosting ou de type Random Forest[8] de calculer une importance relative pour les variables.

La solution est alors d'entraîner tous les arbres prévus. Pour chaque variable, il s'agit alors de procéder en trois temps :

- *1.* On compte le nombre de fois où la variable a été sélectionnée pour créer deux arbres fils dans tous nos arbres.
- *2.* Pour chaque cas où la variable a été sélectionnée, on calcule la diminution d'erreur qu'elle a engendrée dans l'arbre.
- *3.* On moyenne tous les résultats obtenus *sur le nombre d'arbres* pour obtenir l'indication final d'importance.

Ce calcul fournit ainsi une idée de l'importance de la variable, mais ce calcule n'a de valeur qu'en comparaison avec d'autres valeurs calculées, mais aucunement de manière indépendante.

[8] Attention, si le résultat est le même, la manière de l'obtenir sera différente. En effet, pour ces calculs, les méthodes de type Random Forest vont s'appuyer sur des individus dis « *out of bag* », ce qui ne sera pas le cas ici puisque tous les individus ont été utilisés lors de l'apprentissage.

3.2 • Performances

3.2.1 *Principales améliorations*

Outre son aspect algorithmique qui permet des paramétrages particuliers, XGBoost a également été conçue pour être une méthode d'apprentissage performante du point de vue de l'utilisation des ressources et de la parallélisation.

Parallélisation Comme nous l'avons expliqué en Section 2.5, XGBoost va entraîner des arbres au fur et à mesure pour améliorer une métrique. Ainsi, il n'est pas possible de paralléliser l'entraînement des arbres, puisque l'arbre n va dépendre des arbres entraînés précédemment (que ce soit sur l'échantillon ou le poids des données).

Cependant, XGBoost se démarque en proposant à la place une solution pour paralléliser la création d'arbre en calculant des branches de manière indépendante[9].

À noter que cette parallélisation peut être réalisée par une utilisation multi-threads d'une machine ou bien par la répartition du calcul dans des clusters.

[9] Plus de détails sur les algorithmes qu'il est possible d'utiliser à cette fin peuvent être trouvés au lien suivant [4].

Utilisation mémoire Pour le cas de données trop importantes[10], XGBoost peut en le couplant à un stockage de données SSD réaliser une version dégradée de son algorithme qui va consister en des apprentissages partiels du modèles sur des bouts de données. Il est toujours possible d'utiliser l'algorithme en multi-threads pour chaque bout de données.

L'idée est ainsi de conserver les données sur le disque et de ne les monter que partiellement tour à tour en mémoire, on utiliser ainsi une

[10] Et si l'on ne dispose pas d'autres machines pour paralléliser les calculs et donc l'utilisation de la mémoire.

mémoire dite « externe », pour faire tourner l'algorithme[11]. On remarquera aussi que cette stratégie permet d'optimiser les accès au cache de mémoire.

Rapidité Les différents éléments précédents (et en particulier la parallélisation), permettent à XGBoost d'être un algorithme rapide. Ceci est également accentué par le fait qu'une grande partie des modules sont écrits en C++, et que l'interfaçage avec les autres langages se fait principalement par des modules créés par dessus ces briques élémentaires. Ceci peut être observé sur le répertoire Github de XGBoost, où les volumes relatifs des langages sont les suivants :

C++ (41,9%) Scala (16,3%) R (14,4%) Python (11,5%)
JAVA (7,4%) Cuda (4,9%) Autres (3,6%)

On remarque donc que quasiment 50% de l'*ensemble* du code est directement écrit en C++, qui reste parmis les langages utilisés le moins gourmands en ressources et le plus rapide si bien utilisé.

3.2.2 *Benchmarking des solutions*

Plusieurs études ont été menée pour comparer les performances de XGBoost vis-à-vis d'autres solutions.

Benchmarking par Tong He Le créateur de XGBoost a réalisé un comparatif de temps d'exécution entre son algorithme et d'autres algorithme courants sur le challenge Kaggle du boson de Higgs [5]. Les résultats de ce benchmarking sont repris à la Figure 12.

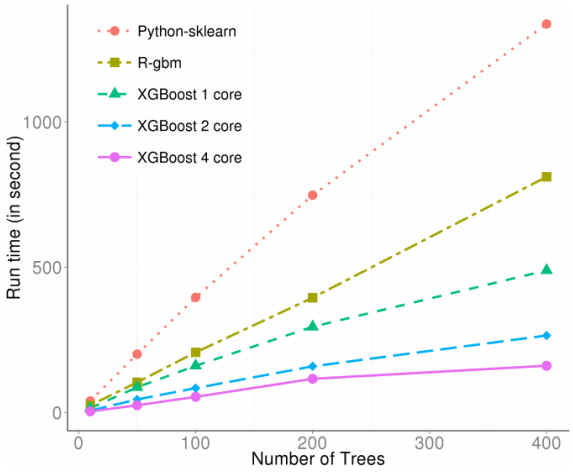


FIGURE 12 • Comparaison des temps d'apprentissage entre XGBoost et d'autres algorithmes importants de Machine Learning sur le challenge Kaggle du boson de Higgs [6]

Sur la Figure 12, les courbes *Python-sklearn* et *R-gbm* correspondent à des algorithmes GBM déjà existants sur ces plateformes, alors que XGBoost correspond à des essais avec l'algorithme nouvellement développé. L'influence du nombre de cœurs utilisés est également testé ici (tous les apprentissages se font à même paramétrage). On tire de ce graphe les conclusions suivantes :

- *XGBoost et autres GBM* Mis à part pour un faible nombre d'arbres appris, XGBoost obtient des performances de bien meilleures (deux fois plus rapide) que les autres solutions de GBM.
- *Influence du nombre de cœur* Plus le nombre de cœurs pour paralléliser XGBoost est important, plus ce dernier est rapide, même si on observe un tassement passé quatre cœurs[12].

[11] Dans un article présentant les améliorations sur les plans théoriques et des performances de XGBoost [3], les auteurs de la méthode précisent que pour arriver à ce résultat, une structure de données « en blocs » est utilisée

[12] Ceci est logique, car comme nous l'avons vu, il n'est pas possible de paralléliser tout le processus d'apprentissage, mais seulement une partie de ce dernier. Ainsi, il restera une partie que l'on ne peut réduire, d'où ce constat.

3.2.3 *Benchmarking sur des implémentations de Random Forest*

Un autre Benchmarking a également visé à comparer les performances de XGBoost avec celles de différentes implémentations de Random Forest. Les résultats sont regroupés à la Figure 13.

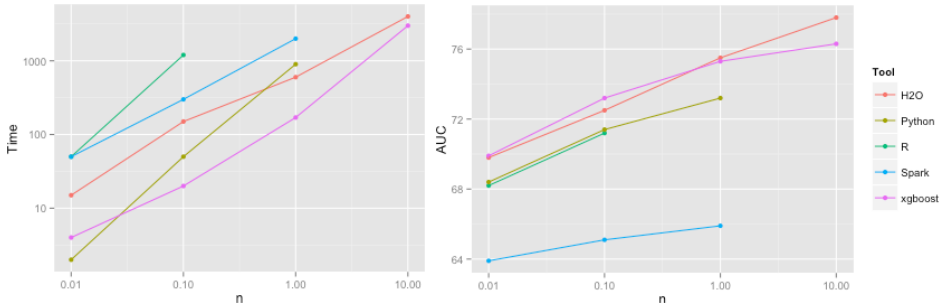


FIGURE 13 • Comparaison des temps d'apprentissage et des résultats entre XGBoost et différentes implémentations de Random Forest [7]

On retrouve également sur ces figures des conclusions similaires à celles du premier benchmarking, à savoir :

- *Temps d'exécution* Sur tous les cas testés (sauf un cas limite où les temps sont tous faibles), XGBoost était déjà (2015) l'implémentation la plus rapide lors de cette étude.
- *AUC* L'étude s'intéressait également aux performances via une mesure d'AUC. Ici, XGBoost au coude à coude avec l'implémentation de Random Forest de H2Oai, cette dernière étant même un peu meilleure sur des taux d'apprentissage plus élevés.

Quoiqu'il en soit, XGBoost apparaît dans ces différentes situtations comme un solution performante tant sur le point de vue des performances que sur celui des temps d'exécution.

3.3 • Valeurs manquantes

Les valeurs manquantes sont un problème courant en analyse de données. La plupart des algorithmes proposent aujourd'hui deux solutions, soit l'utilisateur pré-traite ses données avant de les utiliser et retire ou modifie au besoin les données manquantes, soit il « laisse faire » l'algorithme, au risque que la solution ne lui convienne pas.

XGBoost se démarque ici en proposant un traitement non binaire (ie. on n'assigne pas par défaut `null` ou `0`,...) pour traiter les données manquantes. À la place, l'algorithme propose d'assigner une direction aux valeurs manquantes plutôt qu'une valeur numérique particulière.

Ceci signifie que lorsqu'une découpe est réalisée selon une variable avec des valeurs manquantes, l'algorithme va les classer données « à gauche ou à droite » dans les fils du nœuds. A posteriori de ce classement, XGBoost va regarder quel classement fournirait le meilleur gain et re-répartir ces instances de manière à maximiser ce dernier[13].

Ce comportement est activé par défaut pour XGBoost, et permet à l'algorithme d'apprendre une direction optimale au lieu de simplement considérer toutes les valeurs inconnues comme identiques ou les rejeter.

[13] Pour les personnes intéressées, l'algorithme explicite est précisé dans l'article de T. CHEN (Algorithme 3) [3]

3.4 • Cross-validation native

Pour simplifier l'utilisation de XGBoost, et uniformiser l'utilisation de cette dernière avec XGBoost, les auteurs des packages ont inclus une fonction de cross-validation directement dans XGBoost.

Cette dernière permet alors de reprendre tous les paramètres disponibles par XGBoost en apprentissage pour la cross-validation et d'avoir ainsi un fonctionnement continu. On retrouve en particulier la possibilité d'utiliser une fonction de perte personnalisée[14].

[14] Plus de détails sur les paramètres de XGBoost seront proposés en Section 4.2, on peut également trouver un descriptif de tous les paramètres en cross-validation à [8].

4. Mise en œuvre

4.1 • Grandes étapes de développement de XGBoost

Comme tout projet informatique, XGBoost a été codée de manière itérative afin de s'adapter aux besoins des utilisateurs. Nous allons donc voir ici quelles sont les grandes étapes de développement de cette méthode et quels sont les objectifs futurs[15].

4.1.1 *Premières implémentations*

L'origine de la méthode provient de recherches de T. CHEN sur le boosting d'arbres. L'auteur n'ayant pas trouvé de soution lui convenant, il a décidé d'implémenter sa propre solution et d'en faire un package « maison ». Afin d'optimiser les performances, ce premier code a été réalisé en C++ en utilisant la librairie OpenMP pour avoir une parallélisation automatique sur les CPU multi-threads [5].

Ce package a alors été utilisé sur le challenge Kagle *Higgs Boson Challenge* et ses résultats ont été parmis les meilleurs et ont permis à d'autres concurrents d'améliorer leurs résultats.[16,17].

Face à ce succès, un wrapper Python a alors été mis en place ainsi qu'une API pour l'utilisation. On retrouve alors la première version sur le répertoire Git, qui date de mars 2014. La première version du module Python est elle fournie dès mais 2014.

Suite à cela, le code continue a être développé pour aboutir en septembre 2014 à un module R et un début de parallélisation pour le booster linéaire. De même, l'algorithme de calcul d'arbres est accéléré.

On se retrouve donc à cette étape avec les éléments de la Figure 14.

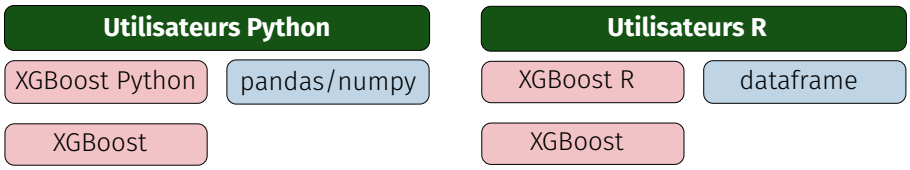


FIGURE 14 • Première étape de développement de XGBoost (d'après [10])

Ainsi, fin 2014, XGBoost était accessible dans les deux principaux langages de Machine Learning, Python et R.

4.1.2 *Mise en place de la version distribuée*

Suite aux bons résultats dans de nouveaux challenges Kaggle et en pratique, la méthode continu à se développer. Ainsi, une release de mai 2015 permet d'instaurer les points suivants :

- *YARN* Une version distribuée qui fonctionne avec YARN et permet de traiter des volumes de données directement liés au Big Data.
- *HDFS* Enregistrement et chargement de données depuis HDFS.
- *Utilisation mémoire* Une première version expérimentale de la gestion de mémoire externe est mise en place.
- *Améliorations* De plus, des améliorations continues aux packages R et Python sont mises en place, notament sur la possibilité d'enregistrer et charger des modèles via ces langages. De plus, le wrapper pour SKLearn (plateforme Python) est terminé.

[15] L'historique qui vous sera proposé ici est issue de données récoltées dans les *Release Note* de la page Git Hub du projet [9], mais aussi d'un retour sur expérience du fondateur de XGBoost sur sa page personnelle [10].

[16] En plus du développement accéléré de XGBoost suite à ce succès, son auteur a également publié un article de recherche précisant ses motivations, les modèles utilisés, et introduisant son code et ses algorithmes à la communauté [5].

[17] Les excellents résultats de XGBoost sur cette compétition ont également eu pour conséquence que cette méthode soit vue comme une avancée majeure en termes d'outils utilisés dans la recherche physique. Ainsi, XGBoost s'est vue attribuée la *High Energy Physics meets Machine Learning Award* pour cette contribution.

Ainsi, XGBoost s'est ouvert à l'univers du Big Data via YARN et HDFS, mais s'est aussi concentré sur la création d'une interface uniforme entre les principaux langages (Pyhton et R). Ceci a ensuite permis aux développeurs de se concentrer plus spécifiquement sur les performances. Dans ces principaux langages, les principaux éléments accesibles sont présentés de la Figure 15[18].

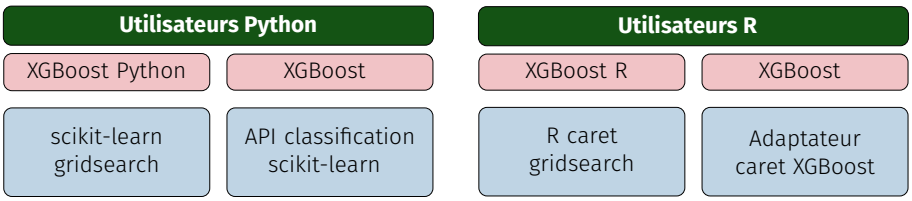


FIGURE 15 • Deuxième étape de développement de XGBoost, uniformisation des offres entre langages (d'après [10])

[18] Dans cette figure, apparaît le terme de « *gridsearch* ». Il s'agit en fait d'une méthode permettant de rechercher des paramètres optimaux pour les algorithmes (pour les paramètres de XGBoost, voir la Section 4.2). Ceci montre donc qu'en optimisant l'interface avec Scikit Learn et R, on augmente ainsi les capacité de XGBoost en l'interfaçant convenablement avec des technologies efficaces et déjà en place.

4.1.3 *Refactoring, autres compatibilité et parallélisme*

Après avoir réalisé les interfaces avec R et Python et mis un premier pied dans le monde du Big Data et du parallélisme, les développements se sont axés pour compléter l'offre logiciel et améliorer la qualité de l'existant. Ainsi, depuis mai 2015, seulement deux nouvelles versions majeures ont été distribuées, mais étendant le pannel de possibilités.

Mise à jour de janvier 2016 Cette mise à jour a terminé les travaux sur les libraires R et Python en corrigeant les principaux bugs et en ajoutant plus de possibilités en paramétrage. En particulier pour Python, l'installation est simplifiée via un support pour `pip`. De même, des compatibilités avec les *Data Frames* de Panda ont été ajoutées.

Outre ces premiers usages, une API JAVA est également proposée et prête à l'emploi.

Enfin, cette mise à jour marque un point important du point de vue de la maintenance future en ajoutant des sécurté supplémentaires et des solutions d'intégration continue pour rendre plus robuste les futures étapes.

Mise à jour de juillet 2016 Cette dernière mise à jour majeure se caractérise par un refactoring important de la librairie. En particulier, le code C++ est entièrement remanié pour utiliser la version C++11, ceci implique en particulier des changements dans la gestion de l'aléatoire ou encore la gestion des pointeurs (et les questions de sécurité liées).

Cette mise à jour contient ensuite un nombre important de changements ou d'amélioration pour divers langages :

- *R* Possibilité d'utilisation des capacités de gestion de mémoire externe, ceci reste cependant toujours bloqué sous Windows pour cause de problèmes techniques.
- *Multi-threading* Correctif pour rendre les librairies XGBoost et Rabit *thread-safe*.
- *JAVA* Un package spécifique pour JAVA et Scala est fourni, nommé XGBoost4j. La version JAVA distribuée fonctionne également sur les frameworks JAVA Flink et Spark.

Ces mises à jour montrent donc la réelle volonté actuelle pour XGBoost, qui est de pouvoir fonctionner sur diverses machines de manière distribuée, de manière transparente. Pour y arriver, un composant d'exécution pour faire le pont entre XGBoost et les différentes plateforme a été développé, du nom de Rabit. Son intérêt est illustré à la Figure 16.

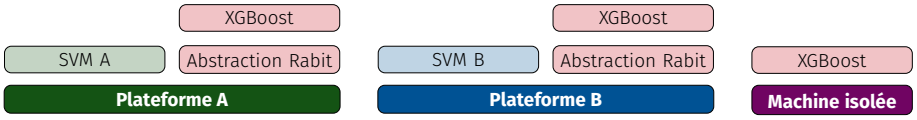


FIGURE 16 • Extension de XGBoost vers un système distribué transparent par le développement de Rabit (d'après [10])

4.1.4 *Intégration dans des plateformes de machine learning*

En parallèle des développements « officiels » de XGBoost, un certains nombre de plateformes de machine learning ont également annoncé intégrer XGBoost parmis les méthodes qu'elles proposaient à leurs utilisateurs. Ceci est le cas pour :

- *Data Science Studio* Il s'agit d'une plateforme de machine learning éditée par la société Dataiku. Cela permet ainsi à ses utilisateurs de pouvoir utiliser XGBoost sans avoir à rédiger (beaucoup) de code. Des exemples d'utilisation sont également fournis sur leur site [11]. Cet ajout a été réalisé en août 2015.
- *H2O.ai* Le 28 janvier 2017, le CTO de la société H2O.ai annonce sur les réseaux sociaux que l'algorithme XGBoost sera utilisable dans leur plateforme lors de la prochaine release majeure [12].

Ces deux exemples tendent à montrer que la méthode requiert l'intérêt de la communauté, et ce très tôt (mi-2015), ce qui peut être un bon indicateur de pérénité.

4.1.5 *Résumé*

En résumé de ces avancées, T. CHEN estime que le développement de XGBoost s'est fait sur le même modèle que celui d'Unix, à savoir être « *ouvert et s'intégrant bien aux autres systèmes par une interface commune*. Son idée est résumée de la manière suivante [10] :

> *XGBoost was designed to be closed package that takes input and produces models in the beginning. The XGBoost package today becomes fully designed to be embeded into any languages and existing platforms. It is like a Lego brick, that can be combined with other bricks to create things that is much more fun than one toy.*

4.2 • Les paramètres

L'implémentation de XGBoost laisse comme nous l'avons déjà laissé supposer propose un nombre important de paramètres laissés libres à l'utilisation. Ces paramètres peuvent être regroupés en trois familles que nous détaillerons tour à tour dans cette section.

4.2.1 *Paramètres génériques*

Ces paramètres permettent de définir les grandes lignes de l'algorithme, comme la version de boosting utilisée et des aspects plus « administratifs ». Trois paramètres principaux sont proposés.

`Booster` Il définit le type de boosting à employer pour l'entraînement du modèle. Nous avons présenté en Section 2.5 la version de boosting associé aux arbres.

Silent Ce mode permet de préciser si l'on demande lors de l'entraînement l'affichage d'informations dans la console (il s'agit en fait d'un mode *verbose*).

Nthread Ce paramètre permet de limiter le nombre de threads que l'algorithme va utiliser. En effet, comme nous l'avons évoqué à la Section 3.2 cette méthode est optimisée pour utiliser au mieux les capacités de calcul de la machine.

Valeurs usuelles Les valeurs communes et celles par défaut associées à ces paramètres sont listées à la Table 1.

TABLE 1 • Valeurs pour les paramètres génériques de XGBoost

Paramètres	Valeurs	Par défaut
`booster`	`gbtree` (arbres) `gblinear` (linéaires)	`gbtree`
`silent`	`1` (activé) `0` (non activé)	`0`
`nthread`	Valeur numérique	Rien, ie. le nombre maximum de threads disponibles.

D'autres paramètres existent pour rendre la liste exhaustive. Pour plus de détails sur ces paramètres restants on peut se reporter à la documentation [2].

4.2.2 *Paramètres de Boosting*

Comme nous l'avons vu à la Table 1, il existe différents types de Boosting que peut utiliser XGBoost. Dans la continuité des points abordés, nous nous attarderons plus en détail sur le paramétrage du booster à base d'arbres[19]. Ce choix est également motivé dans la mesure où ce dernier est le plus utilisé. On retrouve ici la partie la plus importante des paramètres pour XGBoost, et en particulier ceux permettant de contrôler le sur-apprentissage par le modèle.

Eta Ce paramètre permet de contrôler le taux d'apprentissage (ie. un « pas d'apprentissage »). Il permet en particulier d'augmenter la robustesse du modèle en diminuant les poids à chaque itérations.

Min_child_weight Il définit une somme minimale pour les scores des observations d'une feuille après apprentissage[20]. Ce paramètre étant fortement lié aux questions de sur-apprentissage, il est conseille de l'entraîner en utiliser la cross-validation proposée avec XGBoost.

Max_depth La profondeur maximale des arbres qui peuvent être appris. Comme pour `min_child_weight`, ce paramètre est lié au sur-apprentissage, il est donc conseiller de le choisir en utilisant la cross-validation.

Max_leaf_nodes Il s'agit d'une version « duale » du paramètre `max_depth`, qui au lieu de contrôler la profondeur de l'arbre va contrôler le nombre de feuille terminale autorisées[21]. Si jamais ce paramètre est défini, l'algorithme ne prendra pas en compte de valeurs pour max_depth. La même

[19] La liste de paramètres ici proposée n'est pas exhaustive, nous avons cependant retenu les plus importants et ceux les plus souvent mis en avant lors des exemples d'utilisation de XGBoost. Pour une liste plus complète de ces paramètres, vous pouvez vous référer à [13].

[20] Attention, il ne s'agit pas de compter le nombre d'observations par feuilles, mais les poids de ces observations.

[21] Les arbres créés sont binaires, la correspondance est donc parfaite. Ainsi, un arbre de profondeur p produira au maximum 2^p feuilles.

remarque s'applique quant à la manière de fixer sa valeur.

Gamma L'intérêt est ici de fixer une valeur seuil pour autoriser la découpe d'un nœuds en deux sous-nœuds basé sur un critère de gain. Ce paramètre est le γ observé dans la relation (5), et permet donc de définir le taux de laxisme pour la division des nœuds. De même, sa valeur dépendra fortement de la fonction de coût retenue, dans la mesure où le gain n'a pas d'unité absolue mais va dépendre de l'échelle imposée par la fonction de coût.

On rappelle que ce paramètre entre aussi en jeu dans la formule calculant la complexité d'un nouveau modèle, et est le paramètre lié au nombre de feuille de l'arbre (voir la relation (4)).

Lambda Il s'agit d'un terme de régularisation pour la norme imposée sur les cœfficients des feuilles dans le calcul de la complexité d'un arbre, comme observé avec la relation (4). Ce terme intervient aussi en régularisation dans le calcul du gain (5). Ce paramètre peut être utilisé en réduction du sur-apprentissage.

Alpha Ce terme n'avait pas été abordé dans la partie théorique. Ce paramètre est analogue à `lambda`, il est cependant lié à norme un des poids[22]. Comme pour une méthode de Lasso traditionnelle, utiliser ce paramètre permet de réduire le temps de calcul et est particulièrement bien adapté au contexte de hautes dimensions.

[22] Il s'agit alors d'écrira la complexité sous la forme

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2 + \alpha \sum_{j=1}^{T} |\omega_j|$$

Le fonctionnement est alors analogue à celui d'une régression Lasso, avec une solution analytique qui n'existe pas, mais des solutions numériques possibles qui vont en plus faire de la sélection dans les paramètres.

Subsample Dans la mesure où l'algorithme va apprendre plusieurs modèles, l'idée est ici de sélectionner une partie (ou la totalité) de l'ensemble d'apprentissage et de renouveler ce pannel pour chaque modèle. On évite ainsi de trop coler à ces données.

Valeurs usuelles La Table 2 propose alors les valeurs possibles pour ces différents paramètres ainsi que les valeurs par défaut. Un paramètre numérique avec une valeur de `0` indique que ce dernier n'est pas utilisé par défaut.

TABLE 2 • Valeurs pour les paramètres liés au boosting d'arbre

Paramètres	Valeurs	Par défaut
`eta`	Conseillée entre `0.01` et `0.2`.	`0.3`
`min_child_weight`	Valeur numérique	1
`max_depth`	Conseillé entre 3 et 10	6
`max_leaf_nodes`	Conseillé entre 8 et 1024	0
`gamma`	Valeur numérique	0
`lambda`	Valeur numérique	1
`alpha`	Valeur numérique	0
`subsample`	Valeur <1	1

4.2.3 *Paramètres d'apprentissage*

Pour terminer sur le détail des principaux paramètres de XGBoost, nous allons nous intéresser aux paramètres liés à l'apprentissage, c'est-à-dire

pour la fonction objectif et les métriques utilisées. On retrouve trois grands axes.

Objective Il s'agit de définir quel est la fonction de coût (la fonction ℓ dans la relation (2)) à minimiser. Les calculs que nous avions proposés en Section 2.5 supposait de prendre l'erreur MSE. Les trois solutions les plus communes sont :

- *linear* Utilisation d'une simple fonction associée à la régression linéaire (RMSE).
- *logistic* Pour des problèmes de classification binaire, il s'agit d'utiliser la fonction de régression logistique qui renvoie une probabilité d'appartenance à une classe.
- *softmax* Pour des problèmes de classification multiclasse, elle renvoie elle la classe prédite (et non une probabilité d'appartenance).
- *softprob* De même que pour `softmax` si ce n'est que l'on renvoie ici la probabilité.

Eval_metric Cette fonction peut être utilisée pour évaluer le modèle sur l'ensemble de test. À noter que si une fonction est précisée pour `objective` cette dernière sera nécessairement réutilisée. Des fonctions communes sont :

- *rmse* Fonction d'erreur quadratique, utilisée couramment en régression.
- *mae* Fonction d'erreur absolue.
- *logloss* Utilisation de l'opposé de la log-vraissemblance.
- *merror* Une mesure de taux d'erreur pour le cas multiclasse.
- *mlogloss* Une utilisation de la log-vraissemblance pour le cas multiclasse.
- *auc* L'air sous la courbe ROC.

Seed Un paramètre classique pour pouvoir « figer » l'aléatoire entre deux implémentations et utilisations de l'algorithme. Cela permet donc de fournir des modèles reproductibles si nécessaire.

Valeurs usuelles Pour conclure sur ce dernier ensemble de paramètre, nous précisons à la Table 3 les valeurs usuelles et par défaut pour ces paramètres.

Paramètre	Valeurs	Par défaut
`objective`	`logistic`; `softmax`; `softprob`; `linear`; … Possibilité de fonction personnalisée	`linear`
`eval_metric`	`rmse`; `mae`; `logloss`; `error`; `merror`; `mlogloss`; `auc`; …	Variable[23]
`seed`	Valeur entière	0

TABLE 3 • Valeurs usuelles et par défaut des paramètres pour l'apprentissage

4.2.4 *Prévenir le sur-apprentissage*

Comme nous l'avons mentionné, un certain nombre de paramètres sont à surveiller plus particulièrement. Nous regroupons ici les recommanda-

tions sur ces paramètres pour éviter les cas de sur-apprentissage.

Avant tout, il est bon de rappeler que le principe d'utiliser la régularisation pour XGBoost est déjà en soi un moyen de limiter ce sur-apprentissage. Cependant, il est nécessaire de biens configurer les paramètres associés comme nous allons le détailler ici.

Comment réduire le sur-apprentissage ? Dans le détail, l'algorithme XGBoost offre deux méthodes pour réduire le sur-apprentissage :

- *Contrôle de la complexité* Cette dernière peut être contrôlée en contraignant les arbres qui seront générés. Les paramètres concernés sont ici `max_depth`, `min_child_weight`, `max_leaf_node` et `gamma`.
- *Ajout d'aléatoire* Afin de rendre le résultat moins sensible au bruit. Deux solutions sont préconisées par les développeurs de XGBoost :
  - Utiliser les paramètres influant sur l'ensemble d'apprentissage (`subsample` et `colsample_bytree`[24]).
  - Réduire le pas d'apprentissage `eta` mais augmenter `num_round`[25] dans ce cas.

Les différents paramètres et la manière de déterminer leurs valeurs sera précisé dans les paragraphes qui suivent.

`min_child_weight` En évitant d'apprendre des arbres avec des feuilles peu représentatives (ie. de poids faible), on se concentre sur l'apprentissage d'arbres plus représentatifs. Ainsi, des valeurs élevées permettent de prévenir l'apprentissage de relations trop spécifiques à l'ensemble utilisé pour les apprentissages.

`max_depth` De même, utiliser des arbres trop développés va entraîner que ces dernier soient trop spécifiques aux données des ensembles d'apprentissages, il est donc nécessaire d'utiliser des valeurs plus faibles.

`max_leaf_node` Ce paramètre étant directement lié à `max_depth`, la manière de l'appréhender est la même.

`subsample` Pour éviter d'être trop spécifique vis-à-vis des données d'apprentissage, prendre des fractions différentes entre chaque arbre permet de moyenner les biais. Ainsi, prendre une fraction « faible » de ces données est une option intéressante, il faut cependant faire attention à ce que la valeur ne soit pas trop basse, le risque étant alors d'être en sous-apprentissage en ne considérant pas assez de données.

`lambda` Régulariser convenablement les scores des différentes feuilles peut permettre d'améliorer les problèmes de sur-apprentissage. Ainsi, bien que ce paramètre ne soit pas souvent utilisé, il est une option à considérer en cas de problème persistant.

[24] Ce paramètre n'avait pas été présenté auparavant. Son intérêt est de sélectionner aléatoirement des ensembles de colonnes en construisant les arbres afin d'introduire de l'aléatoire dans l'utilisation de ces dernières et éviter que tous arbres ne se ressemblent (on est proche de l'idée des Random Forest).

[25] Ce paramètre non présenté auparavant correspond au nombre d'étapes de Boosting a réaliser pour construire le modèle. L'idée est donc ici de l'augmenter pour contrebalancer la baise du pas d'apprentissage.

5. Applications

5.1 • Challenges Kaggle

5.1.1 *Apparition de XGBoost*

Comme cela a été expliqué, XGBoost est initiallement apparue lors d'un challenge Kaggle. Le challenge en question date de septembre 2014 et avait pour objectif d'explorer les apports possibles du Machine Learning pour la découverte de l'importance des expériences. En particulier, l'idée était la recherche possible de signal provenant de ces bosons à écarter d'un bruit important.

Le code utilisé était encore une version simple, mais qui a fourni un résultat capable de se classer parmis les 10% les meilleurs[26].

[26] On pourra retrouver le code utilisé sur le répertoire Git de XGBoost [14].

De plus, ces bons résultats ont conduits un nombre important de compétiteurs à tester cette solution qui a fini par être une des plus utilisées dans cette compétition.

Une présentation proposée post-challenge précise qu'il était ainsi possible se classer vers le vingt-cinquième rang de la compétition en utilisant un simple modèle XGBoost [15].

5.1.2 *Utilisations suivantes*

Par la suite, XGBoost a acquis une notoriété grandissante parmis les challengers de Kaggle et est aujourd'hui très largement diffusé dans cette communauté. Il entre ainsi dans la plupart des solutions obtenant de bons classements au côté d'autres méthodes comme les Random Forest. Des exemples marquants ont été regroupés dans la Table 4.

Classement	Année	Challenge	Concurrent
2nd	2017	Allstate Claims Severity Competition	A. NOSKOV
1er	2016	Knowledge Discovery and Data Mining Cup	V. SANDULESCU
1et et 3ème	2015	CERN LHCb experiment Flavour of Physics competition	V. MIRONOV
1er	2015	Caterpillar Tube Pricing competition	M. FILHO
2ème	2016	AirBNB New User Bookings	N. KUROYANAGI
2ème	2016	Allstate Claims Severity	A. NOSKOV
10%	2014	Higgs Boson Competition	T. CHEN
...			

TABLE 4 • Quelques résultats utilisant XGBoost lors de challenges Kaggle

L'influence de XGBoost dans le milieu des challenges Kaggle peut se résumer ainsi :

> *Sur les challenges Kaggle de 2015, 17 solutions gagnantes sur 29 utilisaient XGBoost.*

5.2 • Utilisation par des entreprises

De manière générale, les entreprises ne vont préciser en détail quels sont les modèles qu'elles utilisent pour obtenir leurs performances. Il est ainsi assez difficile de savoir qui utilise XGBoost à partir de simples témoignages. Cependant, deux sources nous ont permis d'avoir quelques informations sur la diffusion de cette méthode au sein de ces structures :

- *La page officielle de XGBoost* Sur cette dernière, les auteurs ont recensés quelques cas d'application de XGBoost en entreprise, selon des échos ou des discussions avec des responsables.
- *Les sites d'annonces d'emplois* Si la plupart des entreprise ne donnent pas leurs modèles, elles vont cependant demander aux candidats de maîtriser certaines techniques, et on peut ainsi avoir une idée des secteurs utilisant cet ensemble de méthodes.

5.2.1 *Cas d'utilisation mis en avant par XGBoost*

Les quelques cas de la Table 5 sont mis en avant sur la page Github de XGBoost

TABLE 5 • Cas d'utilisation de XGBoost en entreprise, mis en avant par l'équipe de XGBoost

Entreprise	Utilisation
Alibaba	Utilisation dans son produit ODPS Cloud Service.
Tencent	Utilisation pour faire un choix de prédiction dans le publicités à proposer aux internautes. En particulier, l'équipe technique motive son choix par la facilité d'intégration et la clareté du design de XGBoost.
Auto Home	L'utilisation est ici pour la publicité en ligne et son optimisation.

En plus de ces entreprises, on retrouve des intégrations de XGBoost parmis les plateformes de Machine Learning comme Anaconda, Dataiku ou encore H2Oai.

5.2.2 *Entreprise détectée via les offres d'emploi*

En regardant les offres sur les sites LinkedIn ainsi que sur Indeed, nous avons pu mettre en avant les entreprises de la Table 6, classées par secteurs d'activité.

On retrouve donc la plupart des « grands » secteurs d'activité, et on remarque la présence de quelques grandes entreprises comme AXA ou Expedia. De plus, la répartition géographique est assez homogène, avec des résultats en Asie, en Europe ou en Amérique.

5.3 • Applications au domaine médical

Nous venons de voir avec la Table 6 que certains entreprises situées dans l'assurance s'intéressaient à XGBoost. Ceci nous a conduit à nous intéresser à l'utilisation qui peut être faite de XGBoost en médecine.

Secteur	Entreprises
Banque, crédit et assurances	AXA, Younited Crédit, Chubb, American Family Insurance
Éditeurs de jeux	Pretty Simple, Kinect
Vente	Expedia, Magnetic, Nativo, Seldon Technologies
Consultants et services	Sonsoft Inc, Service Titan, EDJ Analytics, Comma Soft, Intermont
Réseaux	Sesco

TABLE 6 • Entreprises demandant des compétences dans l'utilisation de XGBoost pour certaines de leurs annonces

Si encore une fois aucun exemple professionnel n'est ressorti, nous avons pu mettre en avant deux travaux de personnes indépendantes pour réaliser de l'analyse de données médicales avec XGBoost.

Amélioration du taux de non-présentation des patients Ce cas d'application est issu d'un travail personnel qui fut présenté devant des gérants d'agences de santé, et n'a donc pas été réalisé qu'à des fins « d'études ».

L'idée est ici de détecter des patients qui auraient dû se rendre dans des centres de santé mais ne l'ont pas fait et n'ont ainsi pas été diagnostiqués avant que des symptômes graves ne se déclenchent.

Dans l'article détaillé [16], l'auteur décrit l'intégralité de son processus de traitement de données. Nous retiendrons pour notre part ici que le modèle utilisé est un simple modèle XGBoost, pour lequel une phase de préparation des données et de recherche des paramètres optimaux a été menée, notamment en suivant les bonnes pratiques présentées en Section 6.

Le modèle obtenu fourni ici des taux de vrais positifs et vrais négatifs compris entre 90 et 92%, ce qui est considéré comme bon dans le domaine par l'auteur de l'étude. De plus, ce modèle a permis à son auteur de dresser une carte des quartiers à risque concernant ces non-présentations, que nous fournissons en Figure 17.

FIGURE 17 • Carte de risque pour que les patients ne se présente pas au centre de soin établie à partir d'un modèle XGBoost.

5.3.1 *Prévision de foyer de grippe*

Ce cas d'application (bien que beaucoup relayé sur Twitter) provient d'un membre de Github réalisant des essais de différentes méthodes de Machine Learning avec R. Dans le cadre de cet essai, l'idée était devoir s'il était possible de créer un modèle pour prévoir les morts dûs à la grippe H7N9 en Chine en 2013.

Après avoir fait des essais avec la plupart des méthodes classiques (Random Forest, Elastic Net et KNN entre autres), l'auteur s'est intéressée aux apports possibles de XGBoost à cette étude.

La conclusion est ici que XGBoot a permis d'obtenir des résultats avec moins d'imprécisions que l'ensemble des autres méthodes, et ce en utilisant les pratiques proposée à la Section 6[27].

[27] Le code fournit est assez complet, mais le conclusions peu détaillées, en particulier les notations utilisées pour la représentation des résultats. Le lecteur intéressé pourra cependant se rendre à la page en question [17] pour de plus amples détails.

6. Bonnes pratiques

En regardant les principaux tutoriels dédiés à l'utilisation de XGBoost, ainsi que les recommandations des auteurs de la méthode, il est possible d'en dégager un ensemble de bonnes pratiques. Nous allons présenter ces dernières dans cette partie.

Ces bonnes pratiques seront à avoir en tête lors de toute utilisation de XGBoost.

6.1 • Fixer un niveau d'apprentissage élevé

Pour mémoire, ce niveau d'apprentissage correspond au paramètre `eta`. Il est recommandé initialement de le fixer vers 0,1. Ce dernier sera ensuite affiné pour des valeurs entre 0,05 et 0,3 en fonction des données et des problèmes concernés.

L'idée est ici de prendre un niveau élevé (mais pas trop non plus !) afin que les apprentissages soient rapides et permettent de trouver les autres paramètres optimaux sans passer un temps (trop) considérable dans les phases d'apprentissage. Ainsi, une fois les autres paramètres correctement fixés, on pourra réduire le taux d'apprentissage pour affiner une dernière fois le modèle. On optimise ainsi le temps de configuration du modèle mais aussi ses performances.

6.2 • Trouver le nombre optimal d'arbres

Le nombre d'arbres est un des paramètres qui va le plus influer sur le temps d'apprentissage pour XGBoost, en effet, leur apprentissage est une tâche coûteuse et qui peut devenir inutile passé un certain stade (on observera des plateaux sur les performances au bout d'un certain moment).

À ce compte, il est conseillé d'utiliser les options de validation croisée automatiquement incluses dans XGBoost[28]. Utiliser cette méthode va alors permettre de retourner le nombre optimal d'arbres *in fine*.

[28] Pour rappel, cette option va permettre de réaliser une validation croisée à chaque étape de boosting.

6.3 • Gérer les paramètres des arbres

Ces paramètres[29] sont ceux qui auront le plus d'impact sur le modèle de sortie, comme pour le taux d'apprentissage, on partira de valeur élevée que l'on réduira au cours des itérations.

Il est aussi possible d'utiliser des solutions de *grid search*[30] afin de trouver ces valeurs. Le temps de calcul peut alors courament avoisinner les 15 à 30 minutes (ou plus), ces opérations étant coûteuses.

[29] Pour mémoire, il s'agit de `max_depth`, `gamma`, `subsample`, `min_child_weight`,...Pour revoir la signification de ces paramètres, voir la Section 4.2.2.

[30] Par exemple en utilisant `GridSearchCV` en Python.

6.4 • Gérer les paramètres de régularisation

Ces paramètres[31] vont permettre de simplifier le modèle tout en améliorant ses performances. On peut encore une fois utiliser des solutions de *grid search* pour réaliser ce paramétrage.

[31] Pour mémoire, il s'agit de `lambda` ou `alpha`, pour revoir leur signification, voir la Section 4.2.2.

6.5 • Réduire le niveau d'apprentissage

Une fois les choix optimums réalisés pour les paramètres de régularisation ou d'arbres, on peut finir par affiner le taux d'apprentissage, comme

nous l'avions expliqué initialement, afin d'optimiser une dernière fois notre apprentissage.

6.6 • Utiliser l'AUC pour estimer les modèles

Cette dernière recommandation ne vient pas réellement d'un paramètre, mais est un conseil générique, qui est de prendre l'aire sous la courbe ROC comme mesure par défaut pour les modèles, dans la mesure où cette dernière est souvent plus parlante que ne peut l'être la précision, le rappel,...

Ainsi, il est recommandé de l'utiliser, sauf en cas de besoins spécifiques[32].

[32] En particulier, nous avons vu comme XGBoost pouvait utiliser des fonctions de pertes personnalisées pour des cas plus exotiques à la Section 7.3.

6.7 • Remarques générales

Enfin, un point comparable dans toutes les solutions proposées est que la recherche des paramètres optimaux va augmenter les performances, mais ne se suffit pas à elle-même. Ainsi, il est recommandé de l'utiliser en même temps que des solutions d'extraction de descripteurs pertinents, que l'utilisation de méthode dites *Ensemble*[33] ou encore de *Stacking*[34].

[33] L'idée est ici d'avoir plusieurs modèles experts qui vont décider par vote majoritaire ou par moyenne.

[34] Il s'agit également d'une méthode permettant d'utiliser conjointement des modèles, un tutoriel basé uniquement sur ce point est disponible à [18]. De même, A. Noskov, classé seconde au « *Allstate Claims Severity Competition* » (février 2017) a proposé une description de son modèle utilisant du stacking [19].

7. Exemples

7.1 • Utilisation basique

Avant de regarder plus en détail des utilisations des fonctionnalités originales et avancées de XGBoost, nous fournirons ici quelques exemples d'utilisations de XGBoost avec les principaux langages sur lesquels il a été porté[35].

[35] Les différents exemples présentés proviennent directement de la documentation de XGBoost et présentent l'avantage de concerner le même jeu de données.

7.1.1 *En Python*

Tout le code nécessaire à l'utilisation de XGBoost est regroupé au sein du package développé mentionné dans l'historique de la Section 4.1[36].

[36] Pour l'installation des packages sur les différentes plateforme, vous pouvez vous référer à [20].

```python
import xgboost as xgb

# Lecture des données
dtrain = xgb.DMatrix('demo/data/agaricus.txt.train')
dtest = xgb.DMatrix('demo/data/agaricus.txt.test')

# Paramètres spécifiés dans un dictionnaire
param = {'max_depth':2, 'eta':1, 'silent':1, 'objective':'binary:
    logistic'}
num_round = 2

# Création du modèle
bst = xgb.train(param, dtrain, num_round)

# Application du modèle en prédiction
preds = bst.predict(dtest)
```

7.1.2 *En R*

Tout le code nécessaire pour l'utilisation de XGBoost est ici aussi contenu dans un package nommé `xgboost`[37].

[37] Pour l'installation du package sous R, vous pouvez vous référer à [20], il est à noter que cette installation est automatique en utilisant le package disponible sur le CRAN.

```r
# Lecture des données
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test

# Entrainement du modèle
bst <- xgboost(data = train$data, label = train$label, max.depth =
    2, eta = 1, nround = 2, nthread = 2, objective = "binary:
    logistic")

# Application du modèle en prédiction
pred <- predict(bst, test$data)
```

7.1.3 *En Julia*

Julia est un langage plus récent, mais pouvant être utilisé à même titre que le Python, notamment dans des contextes de calculs numériques, de statistiques ou même de programmation générale et de serveurs webs[38].

[38] Julia est aussi censé être plus rapide que Python d'un point de vue exécution, d'où son intérêt aujourd'hui.

À ce titre, XGBoost a rapidement été porté pour ce langage, on remarquera dans le code ci-dessous la proximité avec les syntaxes de R et Python pour l'utilisation de XGBoost.

```julia
using XGBoost

# Lecture des données
train_X, train_Y = readlibsvm("demo/data/agaricus.txt.train", (6513, 126))
test_X, test_Y = readlibsvm("demo/data/agaricus.txt.test", (1611, 126))

# Entrainement du modèle
num_round = 2
bst = xgboost(train_X, num_round, label=train_Y, eta=1, max_depth=2)

# Application du modèle en prédiction
pred = predict(bst, test_X)
```

7.1.4 *En Scala*

Pour terminer sur les exemples d'utilisation basique de XGBoost, nous fournissons un exemple dans un langage un peu plus exotique qu'est Scala. L'intérêt est ici de voir que XGBoost a également été porté sur des langages à mi-chemin entre l'objet et le fonctionnel.

On remarquera également que le portage de XGBoost sur JAVA s'est principalement fait par Scala[39]. La syntaxe devient cependant plus lourde qu'elle ne peut l'être en R ou Python, comme en témoigne le code suivant.

[39] On remarquera en particulier qu'il existe tout un pan de la communauté Spark, qui semble passer directement par Scala pour utiliser XGBoost (et considérer l'interfaçage avec JAVA comme secondaire), comme en témoigne [21] ou encore [22].

```scala
import ml.dmlc.xgboost4j.scala.DMatrix
import ml.dmlc.xgboost4j.scala.XGBoost

object XGBoostScalaExample {
  def main(args: Array[String]) {
    // Lecture des données
    val trainData =
      new DMatrix("/path/to/agaricus.txt.train")

    // Paramètres spécifiés dans une liste
    val paramMap = List(
      "eta" -> 0.1,
      "max_depth" -> 2,
      "objective" -> "binary:logistic").toMap
    val round = 2

    // Entrainement du modèle
    val model = XGBoost.train(trainData, paramMap, round)

    // Application du modèle en prédiction
    val predTrain = model.predict(trainData)
    model.saveModel("/local/path/to/model")
  }
}
```

7.1.5 *Résumé sur les utilisations basiques*

En définitive, nous pouvons remarquer ici que quel que soit le langage, l'utilisation de XGBoost reste assez transparente, et axée en quatre grandes parties :

- *Chargement des données* Plus particulièrement, on remarque que les données sont chargées et enregistrées au format « `DMatrix` », qui correspond à un format spécialement développé pour XGBoost et ses optimisations de performances. Cette couche est parfois cachée (Python, R) et parfois visible (Scala).

- *Spécification des paramètres* Nous passons sous silence ici la manière de chercher ces valeurs (voir la Section 6 pour des directions à suivre), mais quoiqu'il en soit, ces dernières sont toujours fournies à l'algorithme sous forme de structures de données telles des `map`, des `hash` ou des dictionnaires.
- *Entraînement du modèle* Quel que soit le langage, il suffit d'appeler une méthode XGBoost dont les paramètres reprennent ceux déterminées précédemment, et à laquelle on fournit également l'ensemble d'apprentissage et le nombre d'itérations.
- *Prédiction avec le modèle* Pour cela, on utilise les méthodes de prédictions natives des langages (`predict`,...).

Ainsi, le fonctionnement reste fortement transparent et transposable entre les langages, dans les limites syntaxiques de ces derniers.

7.2 • Utilisation avec Spark

Dans un contexte de calcul distribué ou adapté au Big Data, nous avons vu que l'utilisation de Scala pouvait s'avérer intéressante. Il est cependant possible d'utiliser directement XGBoost via le langage Spark (et par là même l'intégrer dans un écosystème Hadoop).

En utilisant Spark, on peut ainsi gérer une parallélisation de calcul plus importante, qu'elle soit au sein d'une machine ou entre plusieurs machines.

Un exemple de code vous est fourni ci-dessous.

```
// Initialisation du contexte Spark (paramètres de parallélisation).
val spark = SparkSession.builder().appName("SimpleXGBoost
    Application").config("spark.executor.memory", "2G").config("
    spark.executor.cores", "4").config("spark.default.parallelism",
     "4").master("local[*]").getOrCreate()

// Paramètres pour XGBoost
val numRound = 10 // Nombre d'itérations
val numWorkers = 4 // Nombre de processus parallèles
val paramMap = List(
     "eta" -> 0.023f,
     "max_depth" -> 10,
     "min_child_weight" -> 3.0,
     "subsample" -> 1.0,
     "colsample_bytree" -> 0.82,
     "colsample_bylevel" -> 0.9,
     "base_score" -> 0.005,
     "eval_metric" -> "auc",
     "seed" -> 49,
     "silent" -> 1,
     "objective" -> "binary:logistic").toMap

// Entrainement du modèle
println("Starting Xgboost ")
val xgBoostModelWithDF = XGBoost.trainWithDataFrame(trainingData,
    paramMap,round = numRound, nWorkers = numWorkers,
    useExternalMemory = true)

// Application du modèle en prédiction
val predictions = xgBoostModelWithDF.setExternalMemory(true).
    transform(testData).select("label", "probabilities")
```

Ce code est parlant à plusieurs titres :

- *Paramètres* On retrouve dans cette application les principaux paramètres de XGBoost détaillés à la Section 4.2.
- *Parallélisation* On peut remarquer la syntaxe pour utiliser la parallélisation tant avec XGBoost qu'avec Spark. En particulier, le nombre

de *workers* pour XGBoost est identique au nombre d'unités de calcul parallèles spécifiées pour Spark.

- *Syntaxe* On remarque encore une fois une fort rapprochement syntaxique par rapport aux autres langages courants comme Python, R ou bien Julia.

Si ce code peut sembler anodin, son « existence » montre l'impact de XGBoost et la volonté de pouvoir inclure cette méthode dans les frameworks de Big Data, signe de l'intérêt de cette méthode.

7.3 • Fonction de coût personnalisée

Nous avons à plusieurs reprise mentionné la possibilité d'utiliser les fonctions de pertes personnalisées avec XGBoost. Un exemple est fournit ci-dessous où l'on définit manuellement la fonction de log-vraissemblance pour la perte[40].

Cet exemple est réalisé en R, mais le fonctionnement sera le même pour d'autres langages.

```r
loglossobj <- function(preds, dtrain) {
  # On extrait les labels de l'ensemble d'apprentissage
  labels <- getinfo(dtrain, "label")
  # Calcul du gradient et de la partie de la hessienne utiles
  # dans les relations de boosting.
  preds <- 1/(1 + exp(-preds))
  grad <- preds - labels
  hess <- preds * (1 - preds)
  # Renvoie des résultats sous forme de liste.
  return(list(grad = grad, hess = hess))
}

# Entrainement du modèle avec notre méthode
model <- xgboost(data = train$data, label = train$label, nrounds =
    2, objective = loglossobj, eval_metric = "error")
```

[40] La mesure de log-vraissemblance fait en réalité partie des options possibles par défaut dans XGBoost, elle est cependant présentée sous cette forme « manuelle » ici à titre d'exemple. Pour mémoire, la manière de l'appeler serait de faire `objective = "binary:logistic"`.

On remarque donc que la définition est des plus simples[41], dans la mesure où les seuls éléments demandés sont de pouvoir exprimer le gradient et la hessienne de la fonction de perte.

[41] Tant que l'on peut trouver des expressions algébriques...

7.4 • Sélection de variables

Le dernier exemple d'application que nous proposerons ici est celui de la recherche d'importance des variables. En effet, certains algorithmes de Machine Learning (comme les Random Forest), peuvent permettre de comparer l'importance des variables entre elles.

Dans la mesure où XGBoost va à l'instar des Random Forest entraîner plusieurs arbres et faire des choix sur les variables les plus pertinentes pour créer de nouvelles branches, il est possible de mettre en place un calcul d'importance des variables via XGBoost. Ce calcul peut être réalisé en utilisant par exemple le code R suivant.

```r
bst <- xgboost(data = train$data, label = train$label, max.depth =
    2, eta = 1, nthread = 2, nround = 2,objective = "binary:
    logistic")
importance_matrix <- xgb.importance(agaricus.train$data@Dimnames
    [[2]], model = bst)
xgb.plot.importance(importance_matrix)
```

Ce code permet alors d'obtenir les évaluations de la Figure 18.

Dans ce cas particulier on remarque que la plupart de l'information est portée par le premier paramètre, les autres ne retenant qu'une part

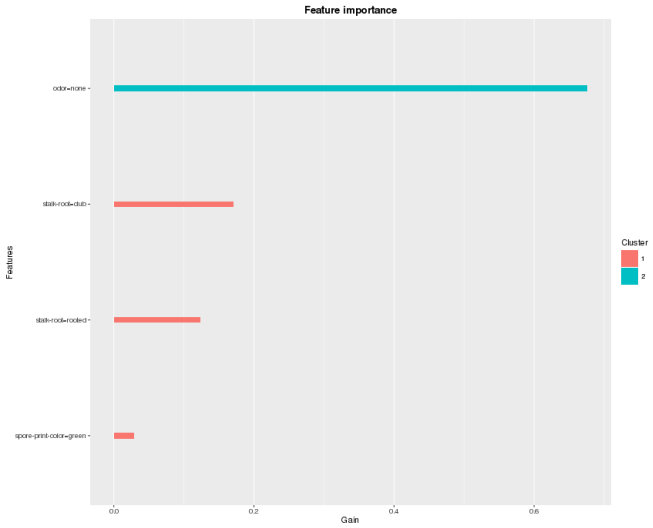


Figure 18 • Exemple d'affichage de l'importance des variables via XGBoost

plus faible de cette dernière. En particulier, le dernier pourraît être supprimé de l'apprentissage, dans la mesure où il semble peu révélateur.

8. Une solution d'avenir ?

Pour terminer sur ce panorama concernant XGBoost, nous allons tenter de répondre à la question suivante : « XGBoost est-elle une solution vouée à perdurer ? ».

Tout d'abord, comme nous l'avons vu, XGBoost est une méthode récente, dont la première implémentation remonte à trois ans. Nous avons cependant mis en avant le fait que cet algorithme a très vite été porté sur les principales plateformes et langages de Machine Learning. Il s'agit ainsi d'un premier point en faveur d'une perrenité de XGBoost, la méthode a été unanimement reconnue comme utile dans la communauté des *data scientists* qui ont investi du temps pour l'intégrer à leurs écosystème. La finalisation de point étant en particulier un interfaçage complet avec le framework Haddop via une API unifiée JAVA/Scala.

Dans un second temps, nous avons également vu que même si les entreprises restent discrètes sur le sujet, des personnes ayant connaissance (et maîtrisant) XGBoost commencent à être recherchées par ces dernières. Ainsi, L'ancrage de XGBoost dans l'industrie est en train de s'opérer, ce qui est facteur très encourageant pour sa survie.

Enfin, les performances permises tant du point de vue des résultats en prédiction que des temps de calculs font de cette solution une méthode compétitive, y compris vis-à-vis des méthodes déjà bien implantées comme les Random Forest.

Tous ces éléments tendent à penser que XGBoost n'est pas qu'un « effet de mode », mais une méthode qui devrait s'installer durablement dans le domaine du Machine Learning, comme l'ont fait auparavant les Random Forest, les SVM,...Cependant, cette richesse a un coût, avec un algorithme demandant une certaine expertise dans sa configuration.

Pour terminer, nous pouvons également avancer que les développements « bruts » de XGBoost arrivent à un degré de maturité importants, et se font donc moins importants. Ceci peut se voir dans le graphe de commits de la page Github de XGBoost, où on remarque que le volume de commits reste faible depuis un peu plus six mois, signe de la maturité du projet, qui est désormais de plus en plus intégré dans les plateformes, mais avec un code initial stable. Ce graphe vous est proposé à la Figure 19.

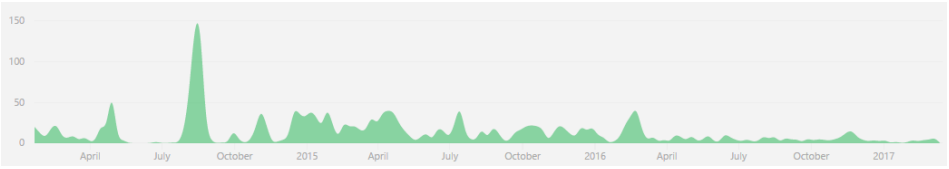


Figure 19 • Évolution du nombre de commits sur la page Github de XGBoostn établi en mars 2017

9. Références

[1] Haytham ELGHAZEL. Méthodes ensemblistes pour l'apprentissage supervisé. *Support de cours à l'UCBL*, 2016.

[2] T. CHEN et al. Scalable and flexible gradient boosting. `https://xgboost.readthedocs.io/en/latest/`, Accès le 25/03/2017.

[3] T. CHEN et C. GUESTRIN. Xgboost : A scalable tree boosting system. *ArXiV*, 2016.

[4] Z. FANG. Parallel gradient boosting decision trees. `http://zhanpengfang.github.io/418home.html`, Accès le 25/03/2017.

[5] T. CHEN et T. HE. Higgs boson discovery with boosted trees. *JMLR : Workshop and Conference Proceedings*, 2014.

[6] T. HE. Parallel computation with r and xgboost. `http://www.parallelr.com/parallel-computation-with-r-and-xgboost/`, 24 janvier 2017.

[7] Szilard PAFKA. Benchmarking random forest implementations. `http://datascience.la/benchmarking-random-forest-implementations/`, 19 mai 2015.

[8] CRAN. xgb.cv : Cross validation. `https://rdrr.io/cran/xgboost/man/xgb.cv.html`, Accès le 25/03/2017.

[9] T. CHEN et Moutai. Xgboost change log. `https://github.com/dmlc/xgboost/blob/master/NEWS.md`, Accès le 12/03/2017.

[10] T. CHEN. Story and lessons behind the evolution of xgboost. `http://homes.cs.washington.edu/~tqchen/2016/03/10/story-and-lessons-behind-the-evolution-of-xgboost.html`, Accès le 12/03/2017.

[11] M. SCORDIA pour Dataiku. Xgboost in data science studio. `https://blog.dataiku.com/2015/08/24/xgboost_and_dss`, Accès le 13/03/2017.

[12] A. CANDEL via Twitter. Xgboost will soon be part of h2o.ai. `https://twitter.com/ArnoCandel/status/825198541958045696/photo/1`, Accès le 13/03/2017.

[13] J. AARSHAY. Complete guide to parameter tuning in xgboost (with codes in python). `https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/`, 1er mars 2016.

[14] Tong HE. Xgboost, higgs trains for r. `https://github.com/dmlc/xgboost/blob/master/demo/kaggle-higgs/higgs-train.R`, Accès le 25/03/2017.

[15] T. CHEN et T HE. Xgboost, extreme gradient boosting. *Slideshare*, 2015.

[16] James MARQUEZ. Walk-throught of patient no-show supervised machine learning classification with xgboost in r. `http://jamesmarquezportfolio.com`, Accès le 25/03/2017.

[17] ShirinG (Github). Extreme gradient boosting and preprocessing in machine learning - addendum to predicting flu outcome with r. `https://shiring.github.io/machine_learning/2016/12/02/flu_outcome_ML_2_post`, Accès le 25/03/2017.

[18] B. GORMAN. A kaggler's guide to model stacking in practice. `http://blog.kaggle.com/2016/12/27/a-kagglers-guide-to-model-stacking-in-practice/`, 27 décembre 2016.

[19] A. NOSKOV. Allstate claims severity competition, 2nd place winners's interview. `http://blog.kaggle.com/2017/02/27/allstate-claims-severity-competition-2nd-place-winners-interview-alexey-noskov/`, 27 février 2017.

[20] T. CHEN et al. Installation guide. `http://xgboost.readthedocs.io/en/latest/build.html`, Accès le 25/03/2017.

[21] E. CUOCO. Spark and xgboost using scala. `https://www.elenacuoco.com/2016/10/10/scala-spark-xgboost-classification/`, 10 octobre 2016.

[22] T. CHEN N. ZHU. Xgboost : Implementing the winningest kaggle algorithm in spark and flink. `http://www.kdnuggets.com/2016/03/xgboost-implementing-winningest-kaggle-algorithm-spark-flink.html`, Accès le 25/03/2017.

[23] Tong HE. An introduction to xgboost r package. `http://dmlc.ml/rstats/2016/03/10/xgboost.html`, 10 mars 2016.

[24] Tong HE. Xgboost, extreme gradient boosting. `http://www.saedsayad.com/docs/xgboost.pdf`, Accès le 25/03/2017.