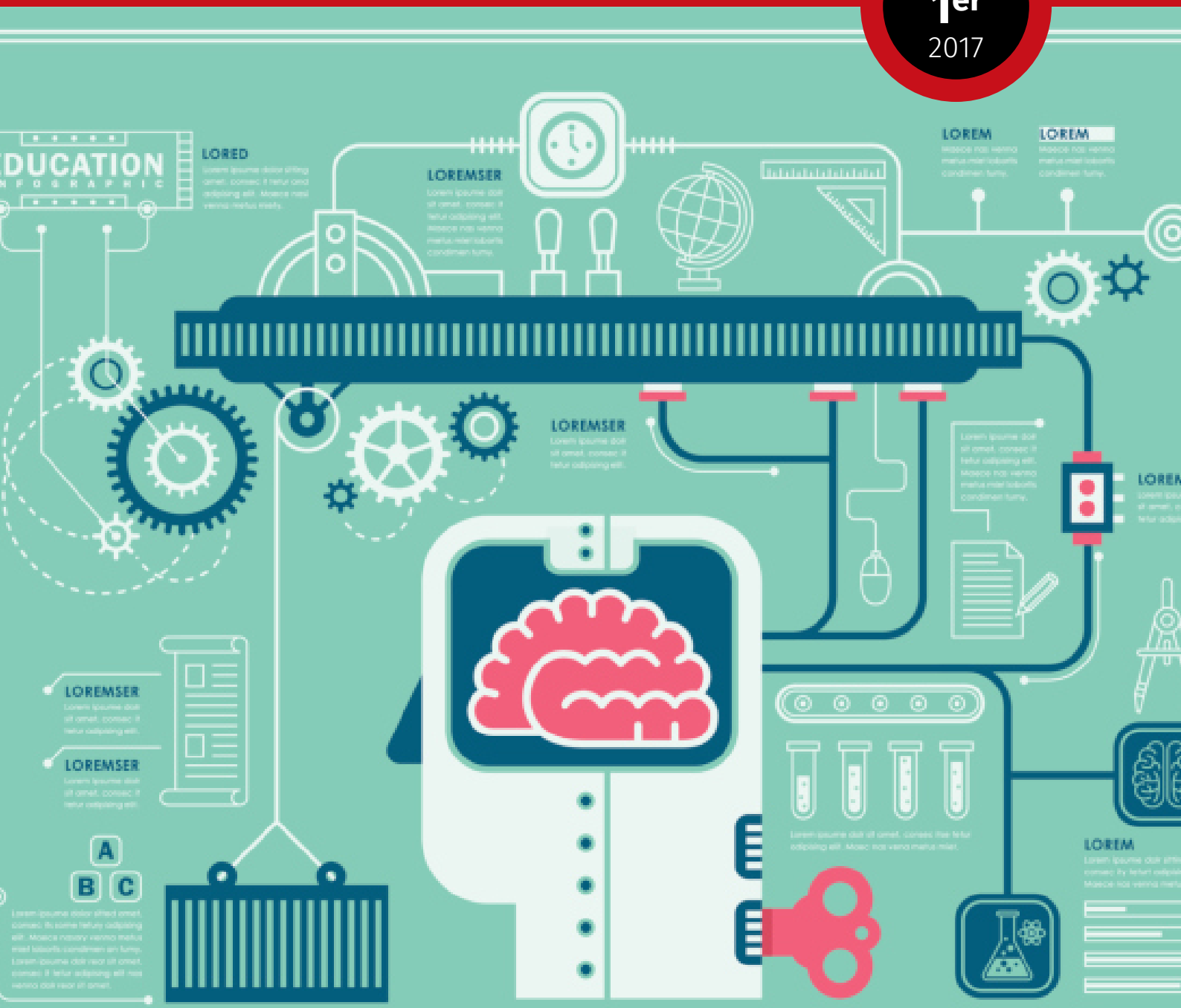
XGBoost
Origine et applications

Principe de la méthode

Mars
1er
2017



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.

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.

[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

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.

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.

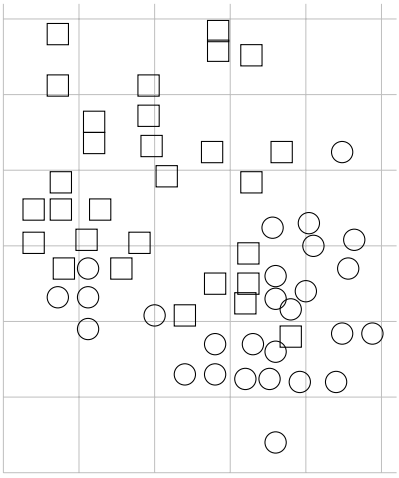
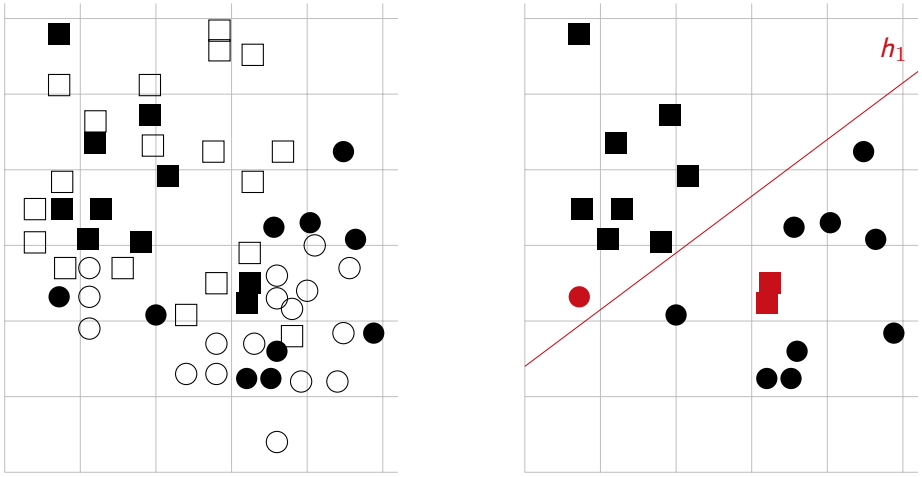


FIGURE 1 • Deux ensembles à séparer par le modèle que l'on entraîne (d'après [**?**])

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 **??** l'ensemble d'apprentissage qui est en rouge, et en Figure **??** le classifieur appris sur ces données, notés h_1.



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

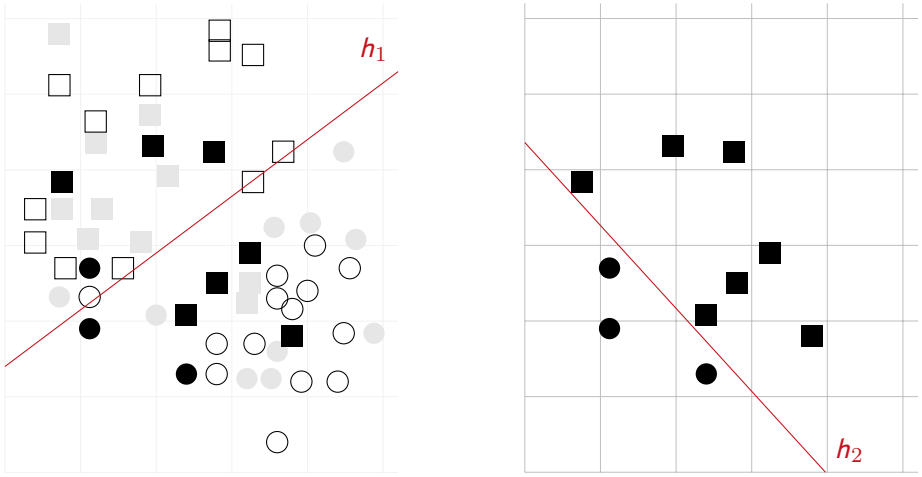
(b) Classifieur h_1 obtenu par entraînement sur l'ensemble ■ ●

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

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

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 **??**. On apprend alors à partir de cet ensemble un nouveau classifieur h_2 présenté à la Figure **??**.



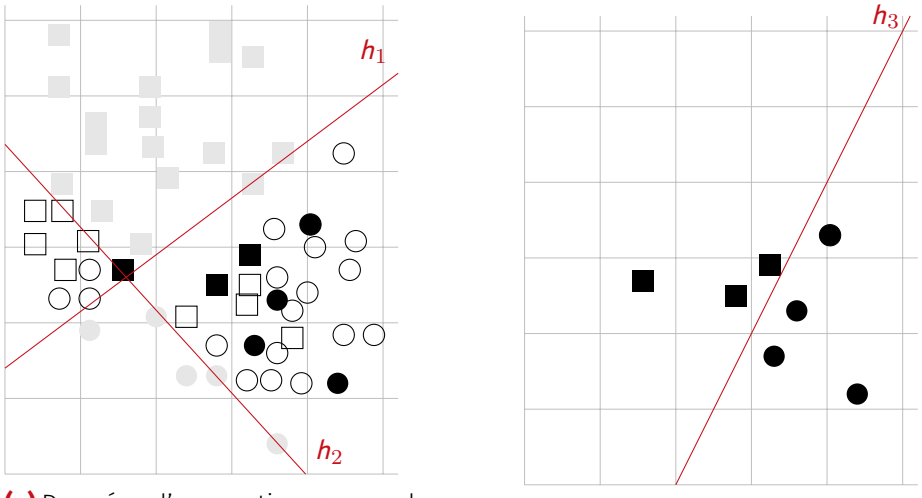
(a) Données d'apprentissage pour le deuxième classifieur (■ ●)

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

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

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 **??**. Le troisième classifieur, noté h_3 est représenté à la Figure **??**.



(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 [**?**])

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

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.

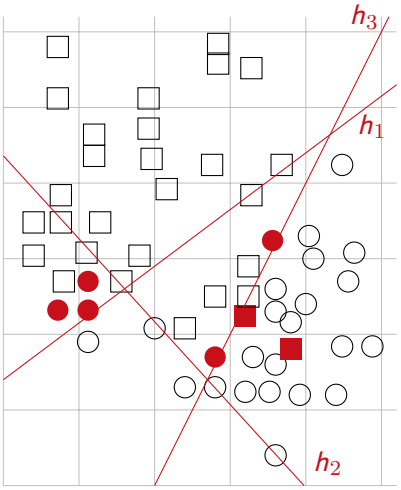


FIGURE 5 • Prévision finale conjointe des trois classifieurs entraînés par Boosting (d'après [**?**])

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'efforcer 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.

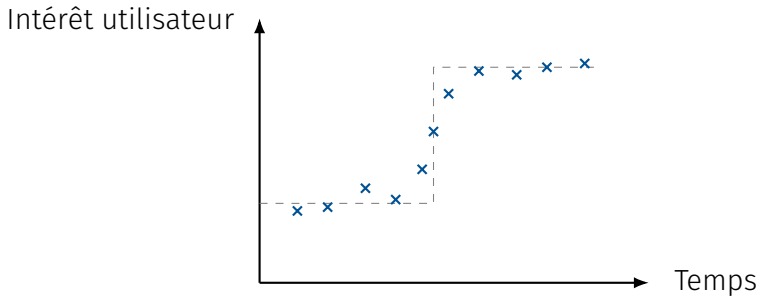


FIGURE 6 • Situation de test pour comprendre les notions de perte et complexité (d'après [**?**])

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.

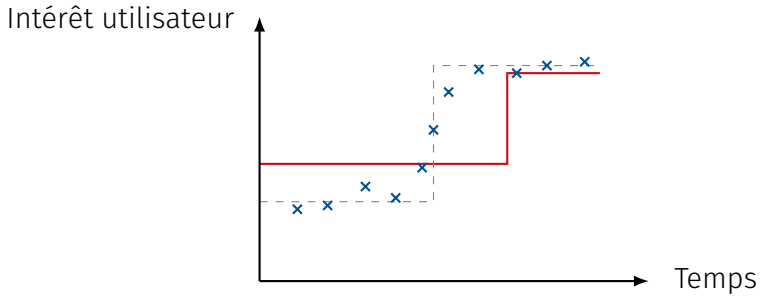


FIGURE 7 • Une situation où la perte n'est pas optimisée (d'après [**?**])

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.

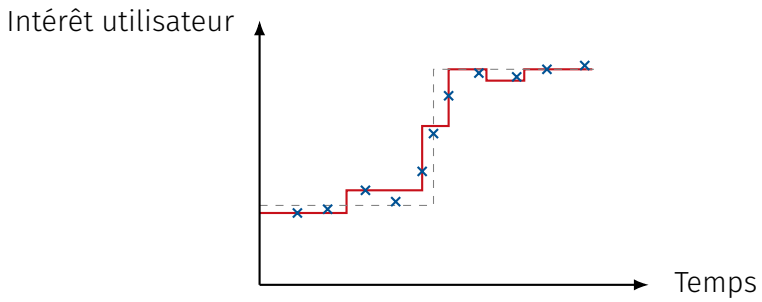


FIGURE 8 • Une situation où la complexité n'est pas optimisée (d'après [**?**])

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

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 **??** et **??**. 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.

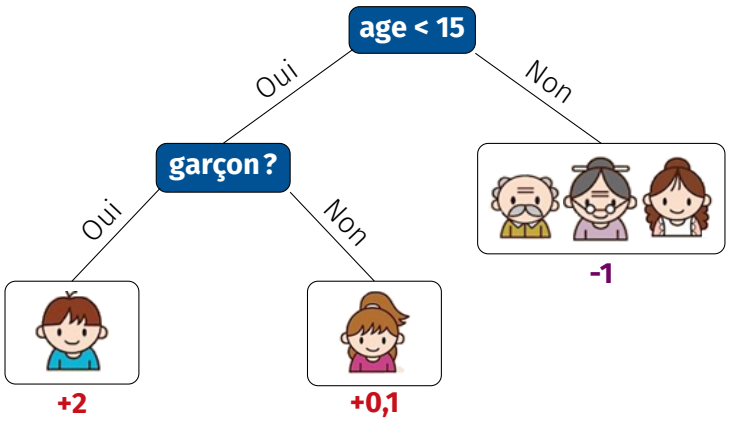


FIGURE 9 • Arbre pouvant convenir pour expliquer la goût des jeux vidéos dans la famille (d'après [**?**])

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.

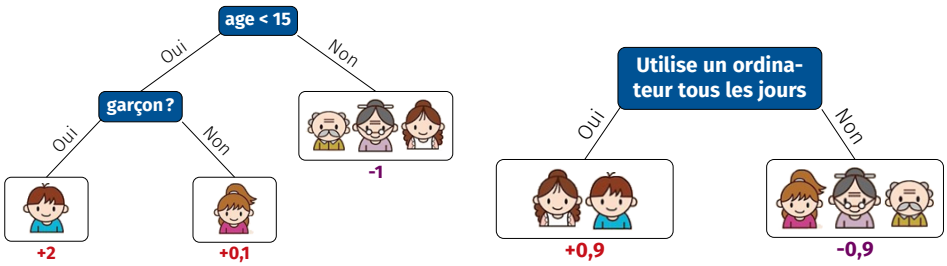


FIGURE 10 • Deux arbres utilisés conjointement pour obtenir le résultat (d'après [**?**])

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 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

[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.

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

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 **??**.

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] :

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

[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 !

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.

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 [**?**].

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] :

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

[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.

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.

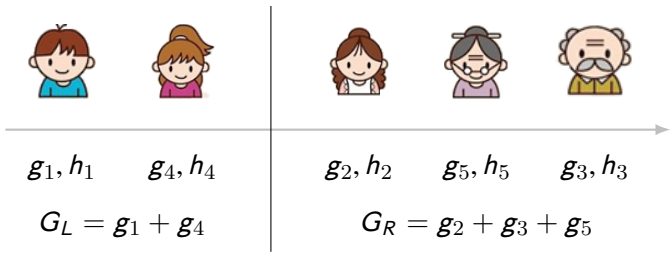


FIGURE 11 • Comment déterminer le meilleur découpage à une étape donné (d'après [**?**])

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.

[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

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.

3.2.1 *Optimisations réalisées*

Mémoire

Utilisation du cache

Conception

Externalisation des données

3.2.2 *Benchmarking des solutions*

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

3.3 • Valeurs manquantes

3.4 • Cross-validation native

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[9].

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 [3].

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.[10,11].

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 12.

[9] L'historique qui vous sera proposé ici est issue de données récoltées dans les *Release Note* de la page Git Hub du projet [1], mais aussi d'un retour sur expérience du fondateur de XGBoost sur sa page personnelle [2].

[10] 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 [3].

[11] 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.

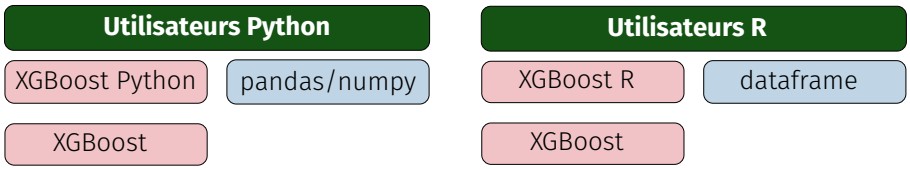


FIGURE 12 • Première étape de développement de XGBoost (d'après [2])

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, notamment sur la possibilité d'enregistrer et charger des modèles via ces langages. De plus, le wrapper pour SKLearn (plateforme Python) est terminé.

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 13[12].

[12] 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 **??**). 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.

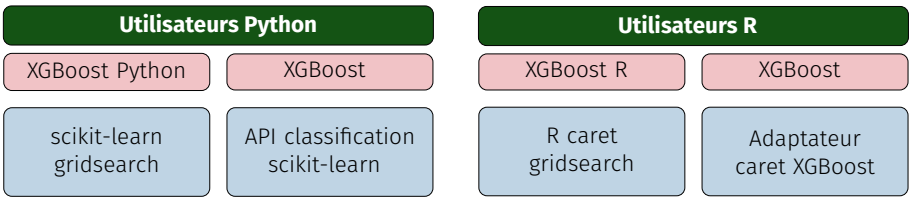


FIGURE 13 • Deuxième étape de développement de XGBoost, uniformisation des offres entre langages (d'après [2])

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

4.2 • Les paramètres

4.2.1 *Paramètres génériques*

4.2.2 *Paramètres de Boosting*

4.2.3 *Paramètres d'apprentissage*

5. Applications

5.1 • Challenges Kaggle

5.2 • Exemple médical

5.3 • En entreprise

6. Exemples

6.1 • Avec Spark

6.2 • Fonction de perte personnalisée

6.3 • Sélection de variables

6.4 • Comparaison de méthodes

7. Une solution d'avenir ?

7. Références

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

[2] 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.

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