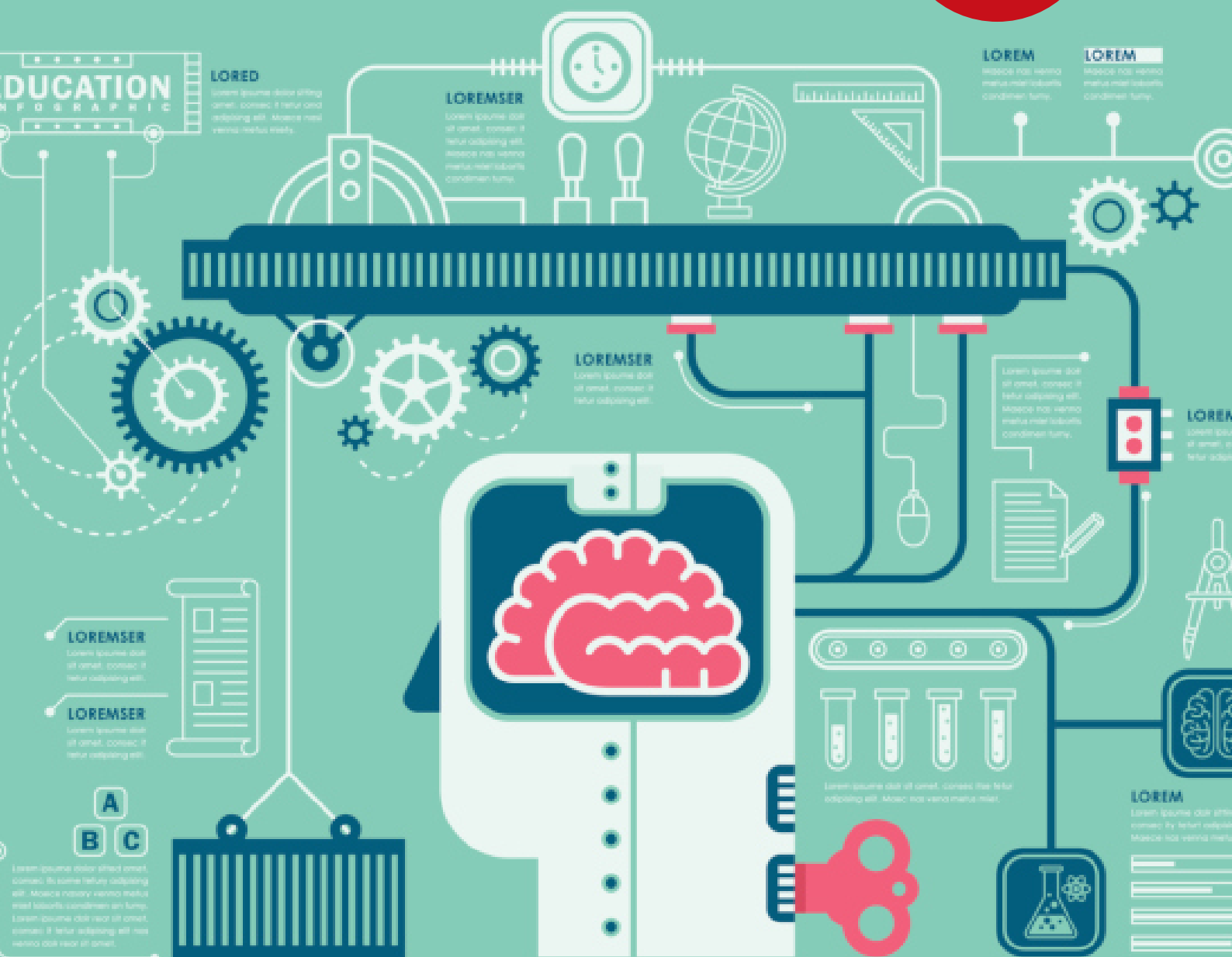


XGBoost

ORIGINE ET APPLICATIONS

Principe de la méthode

Mars
1^{er}
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.
- 🚀 **Performante** 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 » ?

EXtreme **G**radient **B**oosting

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

- ⦿ **Boosting** Il s'agit d'une famille d'algorithmes utilisés initialement 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 fonction faisant apparaître des gradients. Les détails de cette idée seront détaillés 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.

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 évolué au fil du temps en plusieurs versions, 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.

En bref

- ▶ (p. 1) XGBoost en deux mots
- ▶ (p. 1) Le boosting
- ▶ (p. 9) Plus que du boosting
- ▶ (p. 9) Mise en œuvre
- ▶ (p. 9) Applications
- ▶ (p. 10) Exemples
- ▶ (p. 10) Une solution d'avenir ?

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 à 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.

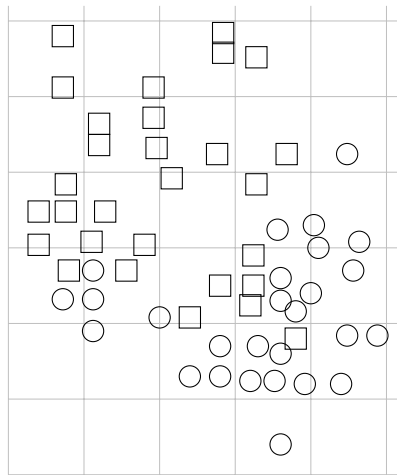
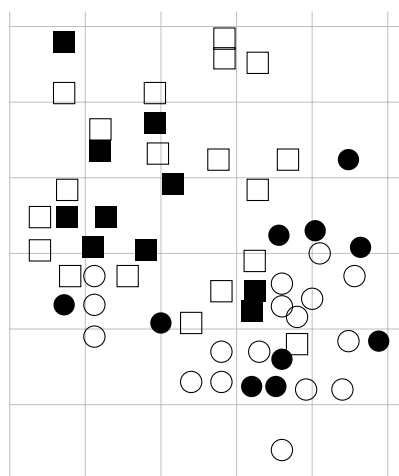
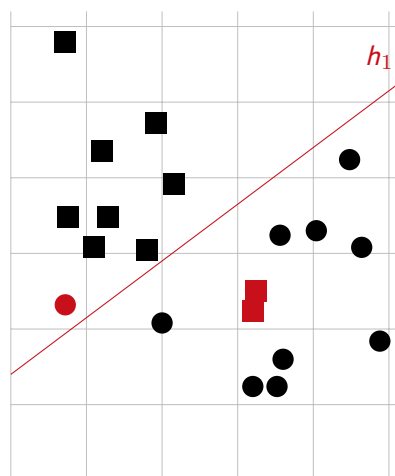


FIGURE 1 • Deux ensembles à séparer par le modèle que l'on entraîne (d'après [?])

Pour commencer, l'idée est d'entraîner un premier classifieur. Cependant, pour retirer une partie du biais des données, ce dernier ne sera entraîné 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 ??.

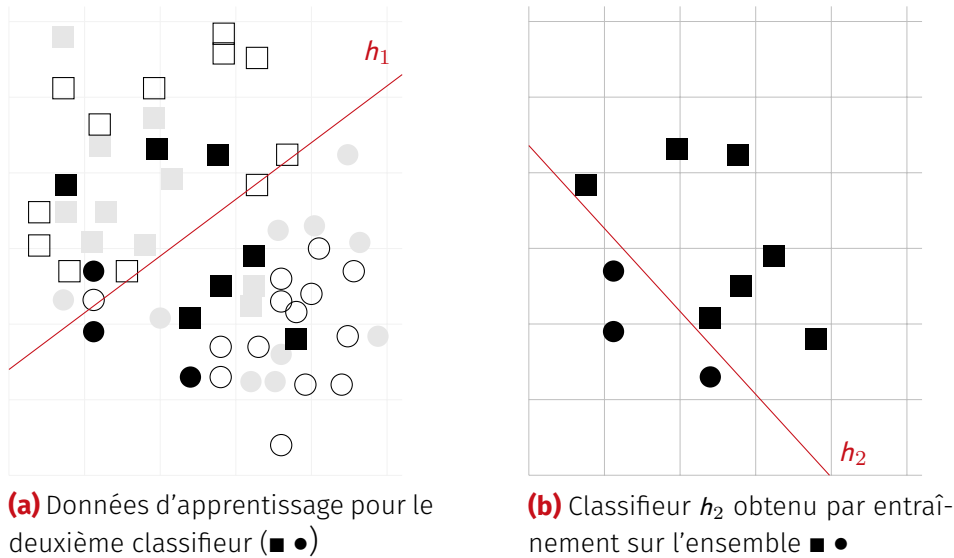


FIGURE 3 • Deuxième modèle appris sur les données (d'après [?])

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 choisi dans les points n'ayant pas encore servi 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 ??.

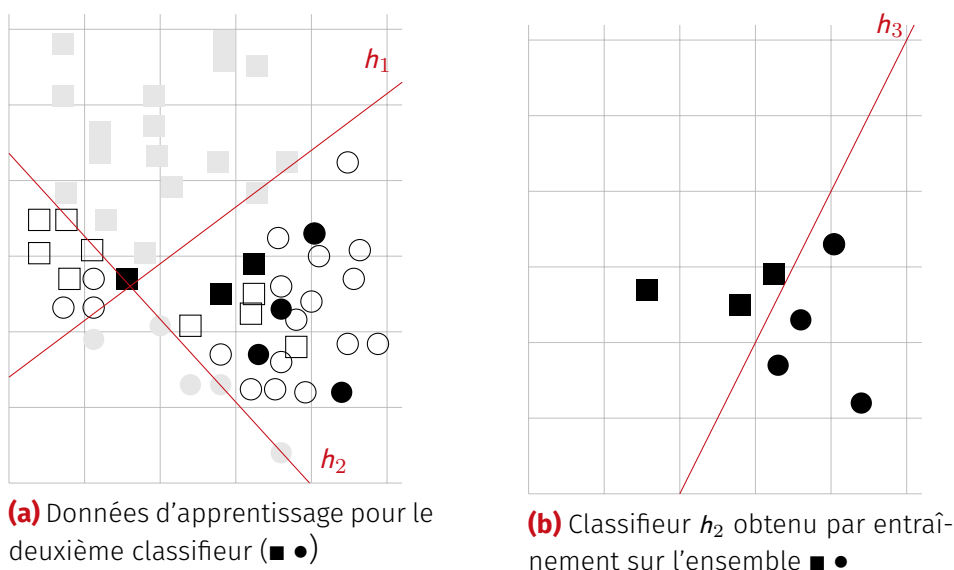


FIGURE 4 • Troisième modèle appris sur les données (d'après [?])

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.

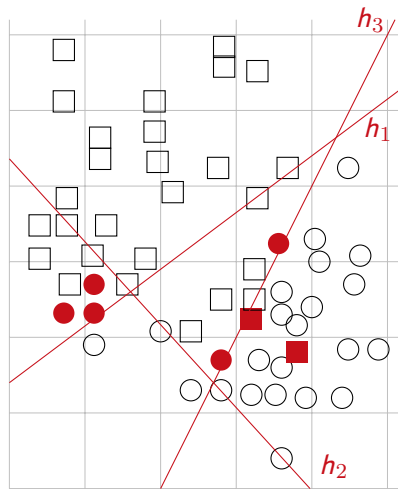


FIGURE 5 • Prédiction finale conjointe des trois classifieurs entraînés par Boosting (d'après [?])

Cette première version de l'algorithme de Boosting a subi 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'importance 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 :

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

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 vont 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.

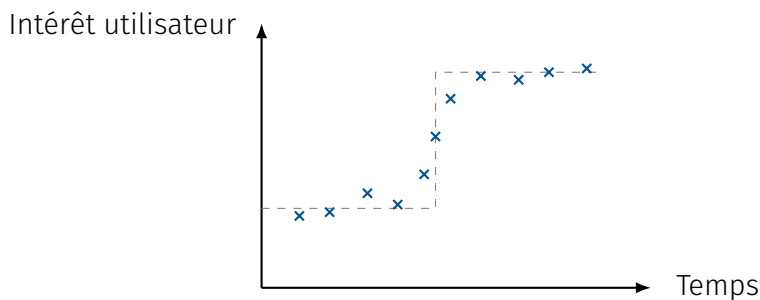


FIGURE 6 • Situation de test pour comprendre les notions de perte et complexité (d'après [?])

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 a priori). Les fonctions communément utilisées sont par exemple l'erreur MSE (liée à la norme \mathcal{L}_2) ou une expression de perte logistique.

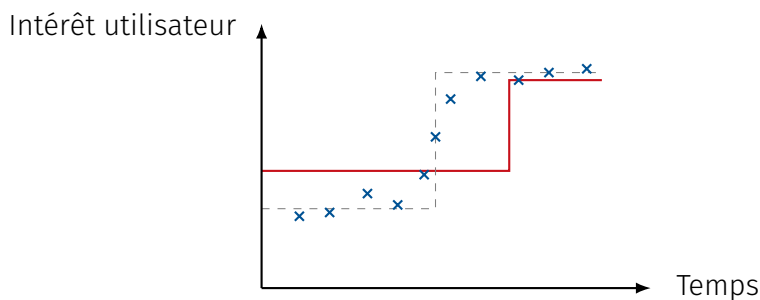


FIGURE 7 • Une situation où la perte n'est pas optimisée (d'après [?])

La Figure 7 propose une situation où la perte induite par 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.

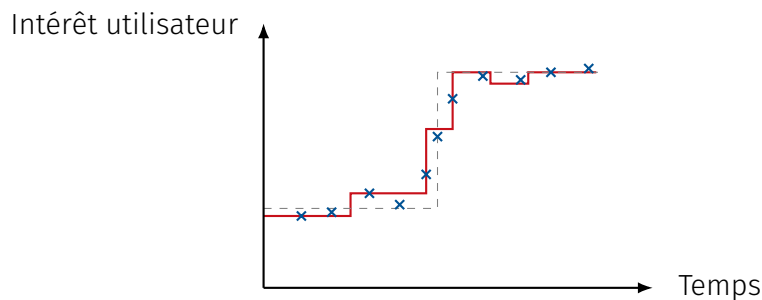


FIGURE 8 • Une situation où la complexité n'est pas optimisée (d'après [?])

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

2.5 • Gradient Boosting et arbres

Désormais que nous venons de voir 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 vidéo.

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.

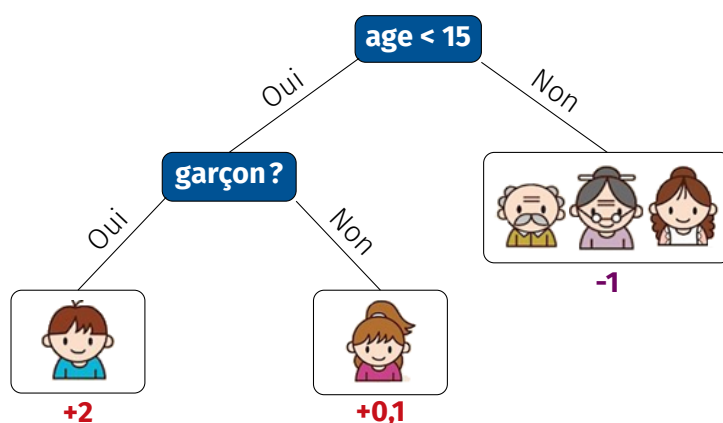


FIGURE 9 • Arbre pouvant convenir pour expliquer la goût des jeux vidéo dans la famille (d'après [?])

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.

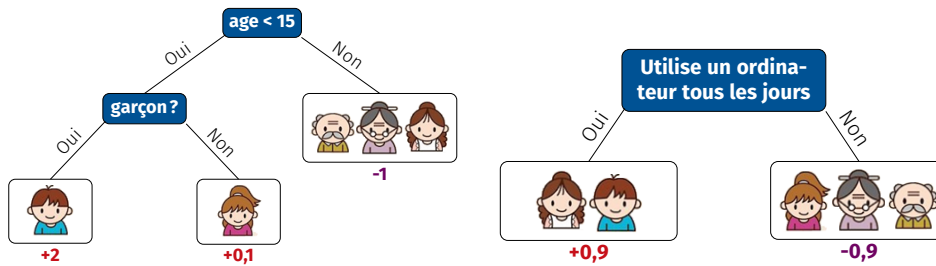


FIGURE 10 • Deux arbres utilisés conjointement pour obtenir le résultat (d'après [?])

Avec les deux arbres de la Figure 10, on peut alors calculer une nouvelle prédiction³ :

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

Pour construire tous les arbres nécessaires, on pourrait fonctionner comme dans la Section 2.2. Cependant, ceci permettrait de bien optimiser 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 :

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

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 courante 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 utilise un développement de Taylor pour remanier la relation (2), on obtient alors la relation (3)⁴.

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

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 couramment utilisée est alors⁵ :

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

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 calcule 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 alors⁶ :

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

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 coefficients 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.

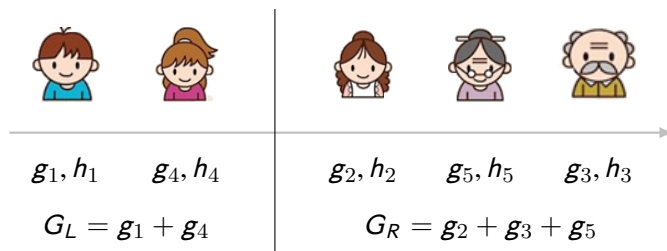


FIGURE 11 • Comment déterminer le meilleur découpage à une étape donné (d'après [?])

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 droite⁷.

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.

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 Forest⁸ 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 1. On compte le nombre de fois où la variable a été sélectionnée pour créer deux arbres fils dans tous nos arbres.
- ② 2. Pour chaque cas où la variable a été sélectionnée, on calcule la diminution d'erreur qu'elle a engendrée dans l'arbre.
- ③ 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 calcul n'a de valeur qu'en comparaison avec d'autres valeurs calculées, mais aucunement de manière indépendante.

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 $\sum_{i=1}^{\lfloor \frac{n_l}{2} \rfloor} \binom{n_l}{i}$

solutions.

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

4. Mise en œuvre

4.1 • XGBoost

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?