Le boosting en apprentissage statistique et XGBoost

Cristian BEREGOI, Mohamed CHAIB

Sommaire

1 Introduction

Dans le domaine en constante évolution de l'apprentissage automatique, l'une des techniques les plus puissantes et influentes est le boosting. Introduit dans les années 1990, le boosting, révolutionne la manière dont les modèles d'apprentissage supervisés sont construits et optimisés. Cette méthode se repose sur l'agrégation de plusieurs modèles faibles pour créer un modèle fort, qui montre des performances remarquables dans divers domaines, allant de la reconnaissance d'image à la prévision financière.

L'objectif de ce mémoire est de fournir une analyse approfondie du boosting, en explorant ses principes fondamentaux, ses différentes variantes et ses applications pratiques. Dans notre recherche, on va se concentrer sur le boosting des arbres de décision. Les arbres de décision sont des structures arborescentes utilisées pour prendre des décisions basées sur des observations de données. Chaque nœud interne de l'arbre représente un test sur une caractéristique (attribut), chaque branche représente le résultat du test, et chaque nœud terminal (feuille) représente une prédiction de la cible (classe ou valeur). Les arbres de décision jouent un rôle important dans notre recherche car ce sont des modèles qui permettent un traitement naturel des données mixtes, des valeurs manquantes. Comme avantage est aussi l'insensibilité aux transformations monotones, la capacité de traiter les données non pertinentes. Ensuite nous abordons des aspects théoriques et mathématique. Une attention particulière va être consacré aux algorithmes de boosting les plus populaires, tels que AdaBoost, Gradient Boosting et XGBoost. Pour chaque algorithme nous discutons de son fonctionnement, de ses avantages et ses inconvénients, ainsi que les contexte dans lesquels ils sont plus efficaces.

En somme, ce mémoire vise à fournir une compréhension complète et nuancée du boosting, en démontrant non seulement ses capacités impressionnantes, mais aussi les considérations pratiques nécessaires pour son implémentation efficace. Par cette exploration, nous espérons contribuer à une meilleure maîtrise de cette technique essentielle et encourager son utilisation judicieuse dans des applications variées de l'apprentissage automatique.

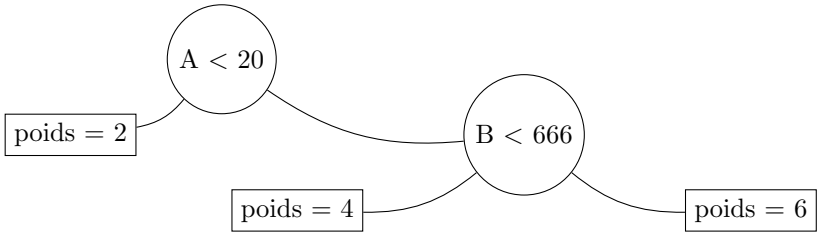
2 Arbres de décision

2.1 Définition

Les arbres de décision sont les prédicteurs faibles utilisés par les modèles de type *Gradient Boosted Tree* présentés ici. C'est en les combinant qu'ils constituent un prédicteur fort.

Les arbres de décision sont une structure de données qui permet d'accéder à une valeur sur la base d'une série de décisions.

Le schéma suivant illustre ce principe pour un arbre permettant de comparer des données constituées de deux colonnes : A et B. Suivant la valeur de ces deux colonnes, les valeurs prédites seront 2,4 ou 6



légende :
- cercles = noeuds;
- rectangles = feuilles, qui contiennent les poids, c'est-à-dire les valeurs prédites;
- Les noeuds contiennent un test binaire. Si le test est positif, le parcours de l'arbre se continue sur la branche de gauche. S'il est négatif, le parcours se continue sur la droite.

Le tableau ci-dessous éclaircit le fonctionnement de cette structure dans le cadre de l'arbre présenté

A	**B**	**Décision (poids)**
18	5	**2**
18	155	**2**
23	555	**4**
23	777	**6**

2.2 Construction de l'arbre

Contexte

Les méthodes basées sur les arbres partitionnent l'espace des caractéristiques en un ensemble de rectangles, puis ajustent un modèle simple dans chacun d'eux. Elles sont simples mais puissantes. Nous décrivons une méthode populaire pour la régression et la classification basées sur les arbres appelée CART.

Arbres de Régression

Soit un jeu de données constitué de p entrées et une réponse, pour chacune des N observations

: c'est-à-dire, (x_i, y_i) pour $i = 1, 2, \ldots, N$, avec $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$. L'algorithme doit décider automatiquement des variables de division et des points de coupure, ainsi que de la forme que l'arbre doit avoir. Supposons d'abord que nous ayons une partition en M régions $R_1, R_2, \ldots, R_T$, et que nous modélisions la réponse comme une constante c_j dans chaque région :

$$f(x) = \sum_{j=1}^{T} c_j I(x \in R_j).$$

Pour chaque variable de division, la détermination du point de coupure s peut être faite très rapidement et donc, en scannant toutes les entrées, la détermination du meilleur couple (j, s) est faisable.

Après avoir trouvé la meilleure division, nous partitionnons les données en deux régions résultantes et répétons le processus de division sur chacune des deux régions. Ensuite, ce processus est répété sur toutes les régions résultantes. Quelle taille doit avoir l'arbre? Un très grand arbre pourrait sur-ajuster les données, tandis qu'un petit arbre pourrait sous-ajuster les données.

La taille de l'arbre est un paramètre de réglage gouvernant la complexité du modèle, et la taille optimale de l'arbre doit être choisie de manière adaptative à partir des données.

Arbres de Classification

Si la cible est un résultat de classification à K modalités, les seuls changements nécessaires dans l'algorithme de l'arbre concernent les critères de division des nœuds et d'élagage de l'arbre. Pour la régression, nous utilisons la mesure d'impureté des nœuds par erreur quadratique définie , mais cela n'est pas approprié pour la classification. Dans un nœud m, représentant une région R_m avec N_m observations, nous définissons

$$p_{jk} = \frac{1}{N_j} \sum_{x_i \in R_j} I(y_i = k),$$

la proportion des observations de classe k dans le nœud j. Nous classifions les observations dans le nœud j dans la classe $k(j) = \text{argmax}_k p_{jk}$, la classe majoritaire dans le nœud m. Différentes mesures $Q_j(T)$ d'impureté des nœuds incluent les suivantes :

Erreur de classification : $\frac{1}{N_j} \sum_{i \in R_j} I(y_i \neq k(j)) = 1 - p_{jk(j)}$.

Indice Gini : $\sum_{k \neq k'} p_{jk} p_{jk'} = \sum_{k=1}^{T} p_{jk}(1 - p_{jk})$.

Entropie croisée ou déviance : $-\sum_{k=1}^{K} p_{jk} \log p_{jk}$.

3 Fonctionnement du Boosting et AdaBoost

3.1 Boosting: le fonctionnement, le but et les enjeux

Le boosting est une méthode d'apprentissage automatique principalement utilisée pour augmenter les performances des modèles de classification et de régression. Il s'agit d'une méthode d'ensembles qui combine plusieurs modèles faibles pour créer un modèle puissant et efficace. Le boosting est réalisé en entraînant successivement plusieurs modèles de base. Chaque nouveau modèle est dédié à la correction des erreurs commises par les modèles précédents. Comment cela marche exactement ? Le modèle simple se déduit à partir de l'ensemble d'entraînement initial. Les erreurs de cette approche sont examinées, ainsi les prédictions inexactes sont déterminés. Par la suite le nouveau modèle de base porte une attention particulière aux erreurs commises par le modèle précèdent, qui sont caractérisés par un poids élevé pour les observations. Finalement les prédictions de tous les modèles sont combinées par une méthode de vote pondéré ou par une moyenne pondérée des prédictions des modèles individuels.

Le boosting cherche à améliorer la précision et la robustesse du modèle final par rapport à un modèle de base unique. Il se concentre sur les erreurs des modèles précédents ainsi, il vise de réduire les erreurs de prédictions de manière significative. Également il peut gérer des données bruyantes ou imparfaites en apprenant progressivement à partir des erreurs.

Cependant, comme le boosting se concentre sur les erreurs spécifiques des modèles précédentes, cela peut mener à un surapprentissage, c'est-à-dire que le modèle est trop adapté aux données d'entrainements et ne généralise pas bien aux nouvelles données. De plus le processus de réentrainement séquentiel peut être coûteux en termes de calcul pour un grand nombre de données ou pour un nombre d'itérations important. Comme mentionné malgré le fait que boosting peut être robuste, cela peut amplifier le bruit dans les données, cela est dû à des modèles complexes ou un nombre important d'itérations.

3.2 AdaBoost: le modèle essentiel

3.2.1 Fonctionnement

L'algorithme le plus fameux due à Freund And Schapire (1997) appellé AdaBoost.M1. Considère un problème à deux classes, avec la variable output codé comme $Y \in \{-1, 1\}$. Un vecteur prédicteur X, un classificateur $G(X)$ qui produit une prédiction en prenant une des deux variables $\{-1, 1\}$. Le taux d'erreur sur l'échantillon d'entrainement est

$$err = \frac{1}{n}\sum_{i=1}^{n} 1(y_i \neq G(x_i))$$

et le taux d'erreur attendu sur les prédictions futures est de

$$E_{XY} I\{Y \neq G(X)\}.$$

Un classificateur faible est un classificateur dont le taux d'erreur n'est que légèrement supérieur à celui d'une supposition aléatoire. Le but de boosting est d'appliquer de manière séquentielle l'algorithme de classification faible à des versions modifiées à plusieurs reprises des données, produisant ainsi une séquence des classificateurs faibles $Gm(x)$, m=1,. . . ,M. Les prédictions de chacun d'entre eux sont ensuite combinées par un vote à la majorité pondérée pour produire la prédiction finale :

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

où $\alpha_1, \ldots, \alpha_M$ sont des coefficients calculés par l'algorithme de boosting, et mesure la contribution de chaque $G_m(x)$. Le but est de donner l'influence la plus importante au classificateur les plus précis dans la séquence.

La modification de données à chaque pas de boosting consiste à appliquer des poids $w_1, \ldots, w_M$ à chaque observation d'entrainement (x_i, y_i) i=1,. . . ,N. Au départ, tous les poids sont fixés à $w_i = 1/\text{N}$, de sorte que la première étape consiste simplement à entraîner le classificateur sur les données de la manière habituelle. Pour chaque itération successive m = 2,3,. . . ,M les poids des observation sont individuellement modifié et l'algorithme de classification est réappliqué aux observations pondérées. À l'itération m, les observations qui ont été mal classées par le classificateur $G_{m-1}(x)$, induites à l'étape précédente voient leur poids augmenter, tandis que les poids sont diminués pour ceux qui ont été classés correctement. Ainsi, au fur et à mesure des itérations, les observations difficiles à classer correctement reçoivent une influence toujours plus grande. Chaque classificateur successif est ainsi contraint de se concentrer sur les observations d'apprentissage mal classés par les classificateurs précédents dans la séquence.

Algorithme 1.1 AdaBoost.M1

1. Initialiser les poids des observations $w_i = 1/N$, $i = 1, 2, ..., N$.
2. Pour m=1 à M:

(a) Ajuster un classificateur $G_m(x)$ aux données d'apprentissage en utilisant les poids w_i.

(b) Calculer

$$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$

(c) Calculer $\alpha_m = log(\frac{1-err_m}{err_m})$.

(d) Modifier w_i

$$w_i = w_i \cdot \exp(\alpha_m \cdot I(y_i \neq G_m(x_i))),$$

i=1,2,...,N.

3. Output

$$G(x) = sign(\sum_{i=1}^{N} \alpha_m G_m(x)).$$

Le classificateur current $G_m(x)$ est déterminé à l'aide des observations pondérées ligne (2a) de l'algorithme. Le taux d'erreur pondéré qui en résulte est calculé à la ligne (2b). La ligne (2c) calcule les poids α_m donné à $G_m(x)$ dans la production du classificateur final $G(x)$ (ligne 3). Les poids individuels de chacune des observations sont mis à jour pour l'itération suivante à la ligne (2d.) Les observations mal classées par $G_m(x)$ sont pondérées par un facteur $\exp(\alpha_m)$, ce qui augmente leur influence relative sur l'induction du classificateur suivant $G_{m+1}(x)$ dans la séquence.

L'aptitude d'Adaboost d'augmenter considérablement les performances même d'un classificateur faible est illustré en figure 1. Les caractéristiques $X_1, \cdots, X_{10}$ sont gaussiens et la cible déterministe Y, definie ainsi:

$$Y = \begin{cases} 1 & \text{si } \sum_{j=1}^{10} X_j^2 > \chi_{10}^2(0.5) \\ -1 & \text{sinon} \end{cases} \tag{1.2bis}$$

Dans ce cas $\chi^2(0.5) = 9.34$ est la médiane d'une variable aléatoire avec 10 degrés de liberté. Ici, le classificateur faible n'est qu'un "moignon" : un arbre de classification à deux nœuds terminaux. En appliquant ce classificateur seul à l'ensemble des données d'apprentissage, on obtient un taux d'erreur très faible de 45,8 % pour l'ensemble de test de formation, contre 50 % pour le random guessing. Cependant, au fur et à mesure des itérations de boosting, le taux d'erreur diminue régulièrement, atteignant 5,8 % après 400 itérations. Ainsi, le renforcement de ce classificateur simple et très faible réduit son

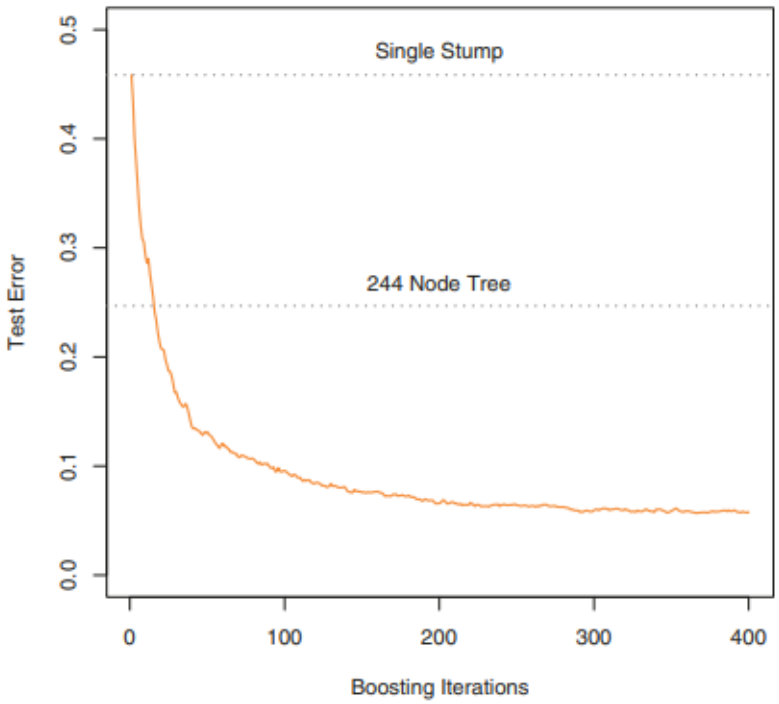


Figure 1: Test taux d'erreur pour boosting avec stumps.

taux d'erreur de prédiction de près d'un facteur quatre. Il est également plus performant qu'un grand arbre de classification (taux d'erreur de 24,7 %).

Les concepts fondamentaux

Le boosting s'adapte à un modèle additif

Le boosting est un moyen d'adapter une expansion additive à un ensemble de fonctions élémentaires "de base".

En mathématiques, une expansion additive est une expression qui consiste à décomposer une fonction ou une série en une somme de termes plus simples. Les fonctions de base sont les classificateurs individuels $G_m(x) \in \{-1, 1\}$. Plus généralement, les expansions des fonctions de base prend la forme

$$f(x) = \sum_{m=1}^{M} \beta_m \cdot b(x, \gamma_m) \tag{1.3}$$

où β_m, m=1,2...M sont les coefficients d'expansions, et $b(x, \gamma_m) \in \mathbf{R}$ sont de manière générale des

simple fonction de l'argument multi varié x, caractérise par un ensemble des paramètre γ.

Algorithme 1.2

1.Initialiser $f_0(x) = 0$

2. Pour m=1 à M:

(a) Calculé

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)\right)$$

b) Modifier

$$f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m).$$

Pour les arbres, γ paramtrisent les variables de séparation et les points de séparation aux nœuds internes , et les prédictions au noueds terminaux.

Typiquement, ces modèles sont ajusté en minimisant une fonction de perte dont la moyenne est calculée sur les données d'apprentissage, tel que l'erreur quadratique ou fonction de perte basée sur la vraisemblance,

$$\min_{\{\beta_m, \gamma_m\}_{m=1}^{M}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m)\right). \tag{1.4}$$

Pour de nombreuses fonctions de perte $L(y, f(x))$ et, ou fonctions de base $b(x; \gamma)$, cela nécessite des techniques d'optimisation numérique à forte intensité de calcul. Cependant, une alternative simple peut souvent être trouvée lorsqu'il est possible de résoudre rapidement le sous-problème de l'ajustement d'une seule fonction de base,

$$\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, \beta b(x_i; \gamma)\right) \tag{1.5}$$

Modélisation additive par étapes successives en avant

Modélisation additive par étapes successives en avant approximé la solution à (1.4) en ajoutant de manière séquentielle des nouvelles fonctions de base à l'expansion sans ajuster les paramètres et les coefficients de ceux qui ont déjà été ajoutés. Cette procédure est décrite dans l'Algorithme (1.2). A chaque itération m, on recherche la fonction de base optimale $b(x; \gamma_m)$ et le coefficient β_m correspondant à ajouter à l'expansion actuelle $f_{m-1}(x)$. On obtient alors $f_m(x)$, et le processus est répété. Les termes ajoutés précédemment ne sont pas modifiés. Pour l'erreur quadratique

$$L(y, f(x)) = (y - f(x))^2, \tag{1.6}$$

on a,

$$L(y_i, f_{m-1}(x_i) + \beta b(x_i;\gamma)) = (y_i - f_{m-1}(x_i) - \beta b(x_i;\gamma))^2 \quad (1)$$
$$= (r_{im} - \beta b(x_i;\gamma))^2, \quad (1.7)$$

où $r_{im} = y_i - f_{m-1}(x_i)$ est simplement le résidu du modelé actuel à l'observation i. Ainsi, pour l'erreur quadratique le terme $\beta_m b(x;\gamma_m)$ qui correspond le mieux aux résidus actuels est ajouté à l'expansion à chaque étape. Cependant, nous allons montrer que la perte d'erreur quadratique n'est généralement pas un bon choix pour la classification.

La perte exponentielle et AdaBoost

Dans cette partie on va montrer que AdaBoost.M1 (Algorithme 1.1) est équivalent à la modélisation additive par étapes successives en avant (Algorithme 1.2) en utilisant la fonction de perte exponentielle.

$$L(y, f(x)) = \exp(-yf(x)) \quad (1.8)$$

Pour AdaBoost, les fonctions de base sont les classificateurs individuels $G_m(x) \in \{-1, 1\}$. En utilisant la fonction de perte exponentielle, il faut résoudre

$$(\beta_m, G_m) = \arg\min_{\beta,G} \sum_{i=1}^{N} \exp\left[-y_i \left(f_{m-1}(x_i) + \beta G(x_i)\right)\right]$$

pour ce classificateur G_m et le coefficient correspondant β_m à ajouter à chaque étape. Cela peut être exprimé ainsi

$$(\beta_m, G_m) = \arg\min_{\beta,G} \sum_{i=1}^{N} w_i^{(m)} \exp(-\beta y_i G(x_i)) \quad (1.9)$$

avec $w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$. Car chaque $w_i^{(m)}$ ne dépend pas ni de β ni $G(x)$, cela peut être vu comme un poids qui est appliqué à chaque observation. Ce poids dépend de $f_{m-1}(x)$, ainsi les valeurs de poids individuelles changent à chaque itération m.

La solution de (1.9), on peut l'obtenir en deux étapes. Tout d'abord, pour chaque valeur de $\beta > 0$, la solution à (1.9) pour $G_m(x)$ est

$$G_m = \arg\min_{G} \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)), \quad (1.10)$$

qui est le classificateur qui minimise le taux d'erreur pondéré dans la prédiction de y. Cette idée peut être exprimé en (1.9) ainsi

$$e^{-\beta} \cdot \sum_{y_i = G_(x_i)} w_i^{(m)} + e^{-\beta} \cdot \sum_{y_i \neq G(x_i)} w_i^{(m)},$$

qui par la suite peut être écrit

$$(e^{\beta} - e^{-\beta}) \cdot \sum_{i=1}^{N} w_i^{(m)} \cdot I(y_i \neq G(x_i)) + e^{-\beta} \cdot \sum_{i=1}^{N} w_i^{(m)} \quad (1.11)$$

En introduisant ce G_m dans (1.9) et en résolvant pour β, on obtient

$$\beta_m = \frac{1}{2} log \frac{1 - err_m}{err_m} \tag{1.12}$$

où err_m est le taux d'erreur pondéré minimisé

$$err_m = \frac{\sum_{i=1}^{N} w_i^{(m)} \cdot I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i^{(m)}}. \tag{1.13}$$

Cette aproximation est actualisé

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x),$$

qui a un impact sur les poids des itérations suivantes

$$w_i^{(m+1)} = w_i^{(m)} \cdot e^{-\beta_m y_i G_m(x_i)}. \tag{1.14}$$

En utilisant le fait que $-y_i G_m(x_i) = 2 \cdot I(y_i \neq G_m(x_i)) - 1$, (1.14) devienne

$$w_i^{(m+1)} = w_i^{(m)} \cdot e^{\alpha_m I(y_i \neq G_m(x_i)} \cdot e^{-\beta_m}, \tag{1.15}$$

où $\alpha_m = 2\beta_m$ est la quantitée définie à la ligne 2.c de l'AdaBoost.M1 (Algorithme 1.1). Le facteur $e^{-\beta_m}$ en (1.15) est équivalent à la ligne 2.d de l'Algorithme 1.1.

On peut considérer la ligne 2(a) de l'algorithme Adaboost.M1 comme une méthode pour résoudre approximativement la minimisation en (1.10). D'où la conslusion que AdaBoost.M1 minimise le critère de perte exponentielle (1.8) par le biais d'une approche de la modélisation additive par étapes successives en avant.

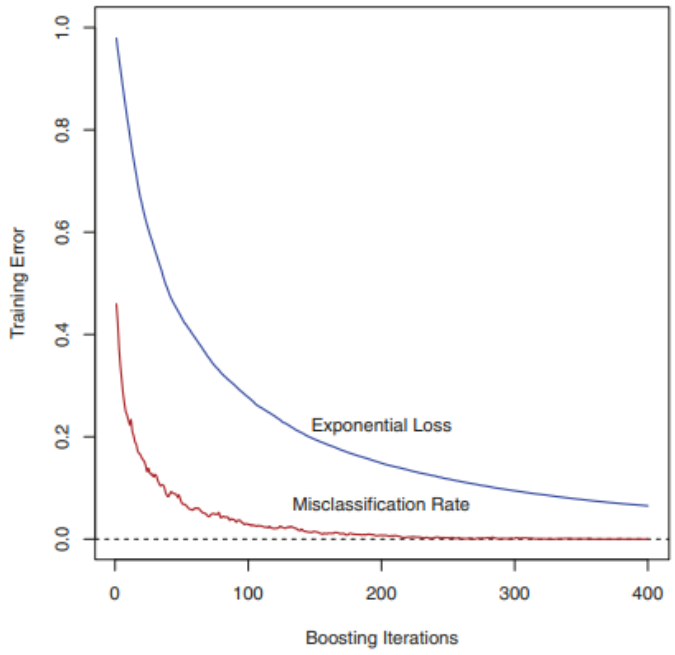


Figure 2: Data Simulé, boosté data des arbres de decision simple.

La figure 2 montre l'ensemble d'entrainement et son taux d'erreur de classification erronée et la perte exponentielle moyenne pour le problème des données simulés de la figure 2. L'erreur de classification diminue jusqu'à zéro approximativement à partir de 250 itérations, mais la perte exponentielle continue de décroitre. On remarque que dans le graphique 2 l'erreur de classification de l'ensemble test continue de s'améliorer après les 250 itérations. On peut affirmer que AdaBoost n'optimise pas l'erreur de mauvaise classification de l'ensemble d'apprentissage ; la perte exponentielle est plus sensible aux changements des probabilités des classes estimées.

Propriétés du critère perte exponentielle

AdaBoost a été conçu d'une perspective complètement différente de la section précédente. Son équivalence à la modélisation additive par étapes successives en avant sur une fonction de perte exponentielle a été découverte cinq ans après son début. En étudiant les propriétés du critère de perte exponentielle, il est toujours possible de mieux comprendre la procédure et de trouver des moyens de l'améliorer.

L'atout principal de la fonction de perte exponentielle est dans le contexte de la modélisation additive est le calcul. Cela mène à l'algorithme AdaBoost de repondération modulaire simple. Cependant, on peut s'interroger sur ses propriétés statistiques. Qu'est ce qu'elle estime et quelle est la qualité de l'estimation ? On répond à la première question en cherchant son minimiseur de

population. Il est simple de montrer que

$$f^*(x) = \arg\min_f E_{Y\,|\,x}\left(e^{-Yf(x)}\right) = \frac{1}{2}\log\frac{\Pr(Y=1|x)}{\Pr(Y=-1|x}, \tag{1.16}$$

or cela equivalent

$$\Pr(Y=1|x) = \frac{1}{1+e^{-2f^*(x)}}.$$

Donc, l'expansion additive produite par AdaBoost estime la moitié des log-odds de $P(Y=1|x)$. Cela justifie l'utilisation de son signe comme règle de classification dans (1.1).

Un autre critère de perte avec le même minimiseur de population est la log-vraisemblance négative binomiale ou la déviance (cross-entropy), qui interprète f comme un logit transformé. Soit

$$p(x) = \Pr(Y=1|x) = \frac{e^{f(x)}}{e^{-f(x)}+e^{f(x)}} = \frac{1}{1+e^{-2f(x)}} \tag{1.17}$$

et on définie $Y^{'} = (Y+1)/2 \in \{0,1\}$. Alors la fonction de perte log-vraisemblance est

$$l(y,p(x)) = Y^{'}\log p(x) + (1+Y^{'})\log(1-p(x))$$

de manière équivalente

$$-l(Y,f(x)) = \log(1+e^{-2Yf(x)}). \tag{1.18}$$

Puisque le minimiseur de population de log-vraisemblance est une vraie probabilité, $p(x) = \Pr(Y = 1|x)$, on constate de (1.17) que le minimiseur de population de la déviance $E_{Y|x}[-l(Y,f(x)]$ et $E_{Y|x}[e^{-Yf(x)}]$ sont les mêmes. On note que e^{-Yf} elle-même n'est pas une log-vraisemblance correcte, car ce n'est pas le logarithme d'aucune fonction de masse de probabilité pour une variable aléatoire binaire.

3.2.2 Les fonctions de pertes et la robustesse

Dans cette section on va examiner les différentes fonctions de perte pour la classification et la régression linéaire, et on les caractérise en terme de leur robustesse à des données extrêmes.

Fonction de perte robuste pour la classification.

Bien que les deux déviances exponentielles (1.8) et la déviance binomiale (1.18) donnent la même solution quand elles sont appliqués à la distribution commune de la population, mais il n'en va de même pour les ensemble de données finis. Les deux critères sont des fonctions décroissantes de la marge $yf(x)$. En classification (avec une réponse (-1,1)) les marges jouent le rôle analogue des résidus $y - f(x)$ dans la régression. La règle de classification $G(x) = sign[f(x)]$ implique que les observations avec la marge positive $y_i f(x_i) > 0$ sont classifiées correctement alors que ceux avec une marge négative $y_i f(x_i) > 0$ sont mal classées.

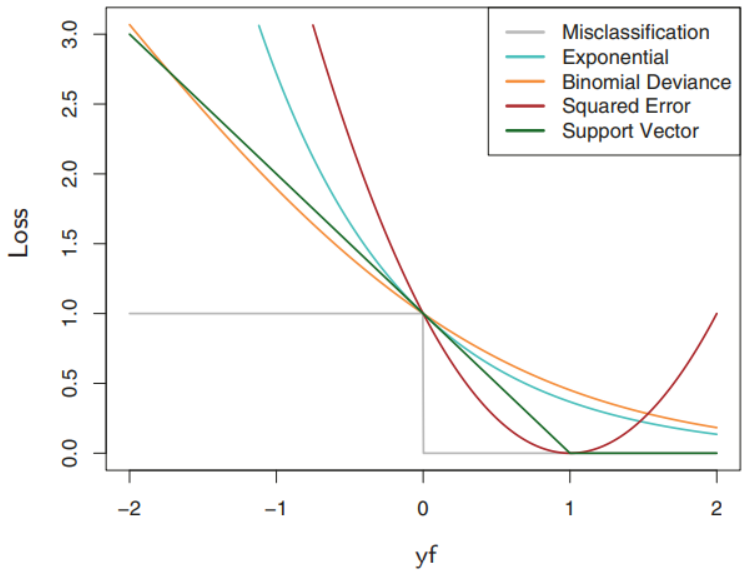


Figure 3: Fonctions de perte pour la classification à deux classes.

La frontière de décision est définie par $f(x) = 0$. Le but de l'algorithme de classification est de produire des marges positives aussi souvent que possible. Tout critère de perte utilisé pour la classification doit pénaliser les marges négatives plus lourdement que les marges positives car les marges positives sont déjà correctement classifiées.

La figure 3 montre que les deux critères exponentielle (1.8) et binomiale (1.18) comme une fonction des marges $yf(x)$. La perte de classification erronée est également indiquée $L(y, f(x)) = I(yf(x)) < 0$, qui prévoit une pénalité unitaire pour les marges négatives, et aucune pénalité pour celles positives. La perte exponentielle et la perte de déviance peuvent être considérées comme des approximations continues monotones de la perte de classification erronée. Ils pénalisent continuellement les valeurs de marge de plus en plus négatives, plus fortement qu'ils ne récompensent les valeurs de marge de plus en plus positives. La différence entre les deux se situe au niveau du degré. La pénalité associée à la déviance binomiale augmente linéairement pour une marge importante de plus en plus négative, tandis que le critère exponentiel augmente l'influence de ces observations de manière exponentielle. À tout moment du processus de formation, le critère exponentiel concentre beaucoup plus d'influence sur les observations présentant des marges négatives importantes. La déviance binomiale concentre relativement moins d'influence sur ces observations, répartissant plus uniformément l'influence sur l'ensemble des données. Elle est donc beaucoup plus robuste dans les environnements bruyants où le taux d'erreur de Bayes n'est pas proche de zéro, et en particulier dans les situations où il y a une mauvaise spécification des étiquettes de classe dans les données d'apprentissage. Il a été observé empiriquement que les performances d'AdaBoost se dégradent considérablement dans de telles situations. La figure montre également la perte par erreur quadratique.

Le minimiseur du risque correspondant sur la population est

$$f^*(x) = \arg\min_f(x) E_{Y|x}(Y - f(x))^2 = E(Y|x) = 2 \cdot \Pr(Y = 1|x) - 1. \tag{1.19}$$

Comme avant la règle de classification est $G(x) = sign[f(x)]$. La perte par erreur quadratique n'est pas un bon substitut de l'erreur de classification. Comme remarqué dans la figure 3 il ne s'agit pas d'une fonction monotone décroissante de marge croissante yf(x). Pour les valeurs de marge $y_i f(x_i) > 1$, elle augmente de façon quadratique, ce qui a pour effet d'accroître l'influence (l'erreur) sur les observations qui sont classées correctement avec une certitude croissante, réduisant ainsi l'influence relative des personnes incorrectement classées $y_i f(x_i) < 0$. Par conséquent, si l'affectation à une classe est l'objectif, un critère décroissant monotone constitue une meilleure fonction de perte de substitution.

Avec la classification à K classes, la réponse Y prend des valeurs dans l'ensemble non ordonné $G = \Gamma_1, ..., \Gamma_k$. Il suffit de connaître les probabilités conditionnelles de la classe $p_k(x) = \Pr(Y = \Gamma_k|x)$, $k = 1, 2, \cdots, K$, alors le classificateur de Bayes est

$$G(x) = \Gamma_k \text{ où } k = \arg\max_l p_l(x). \tag{1.20}$$

En principe, nous n'avons pas besoin d'apprendre les $p_k(x)$, mais simplement de savoir lequel est le plus grand. Cependant, dans les applications d'exploration de données, l'intérêt se porte souvent davantage sur les probabilités de classe $p_l(x), l = 1, ..., K$ elles-mêmes, plutôt que d'effectuer une affectation de classe. Le modèle logistique se généralise naturellement à K classes,

$$p_k(x) = \frac{e^{f_k(x)}}{\sum_{l=1}^K e^{f_l(x)}}, \tag{1.21}$$

ce qui garantit que $0 \leq p_k(x) \leq 1$ et que leur somme est égale à un. Notons que nous avons ici K fonctions différentes, une par classe. Les fonctions $f_k(x)$ sont redondantes, puisque l'ajout d'un $h(x)$ arbitraire à chacune d'entre elles laisse le modèle inchangé. Traditionnellement, l'un d'entre eux est fixé à zéro : par exemple, $f_K(x) = 0$. Nous préférons ici conserver la symétrie, et imposer la contrainte $\sum_{k=1}^K f_k(x) = 0$. La déviance binomiale s'étend naturellement à la fonction de perte de déviance multinomiale de classe K :

$$\begin{aligned} L(y, p(x)) &= -\sum_{k=1}^K I(y = \Gamma_k) \log p_k(x) \\ &= -\sum_{k=1}^K I(y = \Gamma_k) f_k(x) + \log\left(\sum_{l=1}^K e^{f_l(x)}\right). \end{aligned} \tag{1.22}$$

Comme dans le cas à deux classes, le critère (1.22) ne pénalise les prédictions incorrectes que de façon linéaire en fonction de leur degré d'incorrection.

Fonctions de perte robustes pour la régression

Dans le cadre de la régression, analogue à la relation entre la log vraisemblance exponentielle et binomiale est la relation entre la perte erreur quadratique $L(y, f(x)) = (y - f(x)^2)$ et perte absolue $L(y, f(x)) = |y - f(x)|$. Les solutions 'population' sont $f(x) = E(Y|x)$ pour la perte erreur quadratique , et $f(x) = median(Y|x)$ pour la perte absolue; pour les distributions d'erreurs symétriques, elles sont les mêmes. Cependant, sur les échantillons finis la perte erreur quadratique met beaucoup plus l'accent sur les observations qui possèdent des résidus absolues importants $|y_i - f(x_i)|$ pendant le processus d'adaptation (fitting). Il est beaucoup moins robuste, et ses performances se dégradent pour les distributions d'erreur a queue longue et surtout, d'autres critères plus robustes, tels que la perte absolue, donnent de bien meilleurs résultats dans ces situations. Dans la littérature sur la robustesse statistique, une variété de critères de perte de régression a été proposés pour offrir une forte résistance (si n'est pas l'immunité absolue) aux valeurs aberrantes, tout en étant presque aussi efficace que les moindres carrés pour les erreurs gaussiennes. Elles sont souvent meilleures que l'une ou l'autre pour les distributions d'erreurs avec des queues modérément lourdes. L'un de ces critères est le critère de perte de Huber utilisé pour la régression M :

$$L(y, f(x)) = \begin{cases} [y - f(x)]^2 & \text{pour } |y - f(x)| \leq \delta, \\ 2\delta|y - f(x)| - \delta^2 & \text{sinon.} \end{cases} \tag{1.23}$$

La figure compare ces trois fonctions de perte. Ces considérations suggèrent que lorsque la robustesse est une préoccupation, comme c'est particulièrement le cas dans les applications d'exploration de données (data mining). La perte par erreur quadratique pour la régression et la perte exponentielle pour la classification ne sont pas les meilleurs critères d'un point de vue statistique. Cependant, ils conduisent tous deux à d'élégants algorithmes de stimulation modulaire dans le contexte de la modélisation additive par étapes. Pour la perte par erreur quadratique, il suffit d'ajuster l'apprenant de base aux résidus du modèle actuel $y_i - f_{m-1}(x_i)$ à chaque étape.

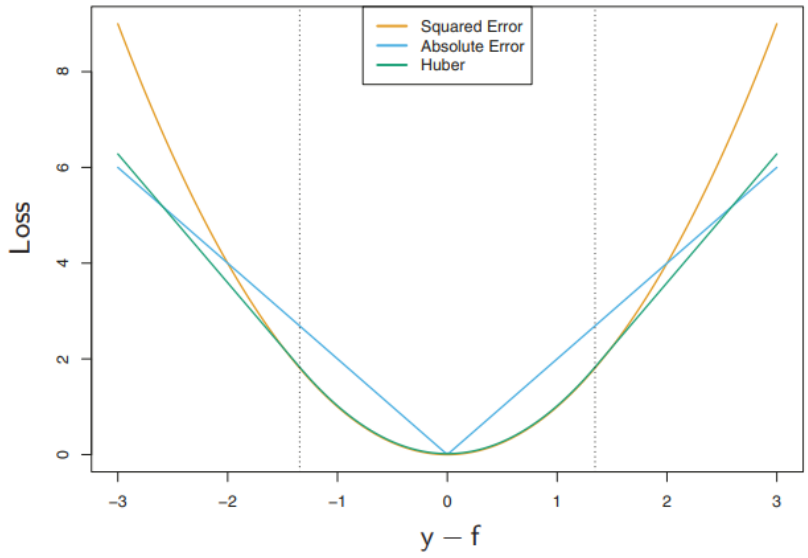


Figure 4: Comparaison de trois fonctions de perte pour la régression, en fonction de fonction de la marge $y - f$.

Pour les pertes exponentielles, on procède à un ajustement pondéré de l'apprenant de base aux valeurs de sortie y_i, avec des poids $w_i = \exp(-y_i f_{m-1}(x_i))$. L'utilisation d'autres critères plus robustes directement à leur place ne donne pas lieu à des algorithmes de boosting aussi simples et réalisables.

4 Gradient Boosting

4.1 introduction

Comparé à AdaBoost, le Gradient Boosting se distingue par son approche plus flexible et puissante. AdaBoost fonctionne en ajustant les poids des observations en fonction des erreurs des modèles précédents, augmentant ainsi l'importance des observations mal prédites. Mathématiquement, AdaBoost met à jour les poids w_i des observations x_i en utilisant l'erreur du modèle précédent ϵ_t :

$$w_i \leftarrow w_i \exp(\alpha_t \cdot \mathbb{1}(y_i \neq h_t(x_i)))$$

où α_t est un coefficient basé sur l'erreur ϵ_t.

En revanche, le Gradient Boosting optimise directement les résidus des prédictions en ajustant les modèles successifs pour corriger les erreurs des modèles précédents. Il construit chaque nouveau modèle h_t pour minimiser une fonction de perte $L(y, F_{t-1}(x) + \rho_t h_t(x))$ par descente de gradient, où ρ_t est un facteur de réduction et F_{t-1} est le modèle accumulé jusqu'à l'itération $t-1$. Ce processus est formalisé comme suit :

$$\mathbf{F}_t(x) = \mathbf{F}_{t-1}(x) + \rho \cdot h_t(x)$$

Cette approche permet au Gradient Boosting de mieux capturer les relations complexes et de réduire l'overfitting grâce à des techniques de régularisation telles que le shrinkage (réduction du taux d'apprentissage) et la sous-sélection des échantillons.

Avant d'entrer plus dans le détail, il faudrait tout d'abord définir les outils nécessaires à l'utilisation du Gradient Boosting

4.2 Rappels mathématiques : Optimisation

Problème (Minimisation) Soit $U \subset \mathbb{R}^N$ un ouvert non vide de $\mathbb{R}^N$ et $f : U \to \mathbb{R}$ une fonction définie sur U (qui peut être plus ou moins régulière). On s'intéresse au problème de minimisation

$$\inf_{x \in K} f(x), \quad (P)$$

où $K \subset U$ est un sous-ensemble non vide de U.

On s'intéresse aux questions suivantes :

- Existent-il des minimiseurs de f sur K ?
- Dans quelles conditions les minimiseurs sont uniques ?
- Comment peut-on retrouver les minimiseurs de f sur K ?
- Si on ne peut pas les calculer, comment peut-on les retrouver numériquement et quels sont les enjeux des algorithmes ?

4.2.1 Convexité et Unicité des Minimiseurs

4.2.2 Convexité

Définition Soit C un ensemble convexe non vide de $\mathbb{R}^N$. C est convexe si $\forall x, y \in C, \forall t \in [0,1], (1-t)x + ty \in C$.

Fonction Convexe Une fonction $f : U \to \mathbb{R}$ est convexe sur C si $\forall x, y \in C, \forall t \in [0,1], f((1-t)x + ty) \leq (1-t)f(x) + tf(y)$.

Fonction Strictement Convexe f est strictement convexe si $\forall t \in]0,1[, \forall x, y \in K, x \neq y, f((1-t)x + ty) < (1-t)f(x) + tf(y)$.

Fonction Fortement Convexe f est fortement convexe s'il existe $\lambda > 0$ tel que $\forall x, y \in C, \forall t \in [0,1], f((1-t)x + ty) \leq (1-t)f(x) + tf(y) - \frac{\lambda}{2}t(1-t)\|y - x\|^2$.

4.2.3 Quadratiques et Convexité

Fonction Quadratique Soit $A \in \mathcal{M}_N(\mathbb{R})$ une matrice carrée, $b \in \mathbb{R}^N$ un vecteur et $c \in \mathbb{R}$. Une fonction quadratique $f : \mathbb{R}^N \to \mathbb{R}$ est définie par :

$$f(x) = \frac{1}{2}\langle Ax, x\rangle - \langle b, x\rangle + c.$$

Gradient et Hessienne Le gradient et la hessienne de f sont :

$$\nabla f(x) = (A + A^T)x - b, \quad Hf(x) = A + A^T.$$

Si A est symétrique, alors $\nabla f(x) = Ax - b$ et $Hf(x) = A$.

Critère de Convexité

- f est convexe si A est positive.
- f est strictement convexe si A est définie positive.
- f est fortement convexe si A est définie positive avec $\lambda_{\min}(A) > 0$.

Existence et Unicité de Minimiseurs

- Si $A \succeq 0$, f admet un minimiseur unique car elle est coercive et strictement convexe.
- Sur $K = \{x \in \mathbb{R}^N : \sum_{i=1}^N x_i \leq 1\}$, f admet un minimiseur unique sous les mêmes conditions.
- Sur $K = \{x \in \mathbb{R}^N : x_i \geq 0, \sum_{i=1}^N x_i \leq 1\}$, f admet un min/max unique si elle est strictement convexe/concave.

Critères d'Optimalité Pour A symétrique, un point critique x^* est un minimiseur si :

$$\begin{cases} \nabla f(x^*) = Ax^* - b = 0 \\ Hf(x^*) \succeq 0 \end{cases}$$

4.2.4 Unicité

Théorème Soit $K \subset \mathbb{R}^N$ convexe non vide et $f : K \to \mathbb{R}$ une fonction strictement convexe. Si f admet un minimiseur, alors il est unique.

Exemple $f(x, y) = x^2$, $g(x, y) = e^x + e^y$, $h(x, y) = \|(x, y)\|^2$. f admet une infinité de minimiseurs, g n'admet pas de minimiseur, h admet un unique minimiseur.

Critère de Convexité du 1er Ordre f est convexe sur C si et seulement si $\forall x_0, x \in C, f(x) \geq f(x_0) + \langle \nabla f(x_0), x - x_0\rangle$.

Critère de Convexité du 2ème Ordre f est convexe sur C si et seulement si $\forall x \in C, Hf(x) \succeq 0$.

4.2.5 Algorithmes de descente

On considère une fonction (coût) $f : \mathbb{R}^N \to \mathbb{R}$ une fonction différentiable que l'on souhaite minimiser.

$$I = \inf_{x \in \mathbb{R}^N} f(x)$$

On supposera la fonction **fortement convexe** et on aura donc existence et unicité d'un minimiseur :

$$\exists! x^* \in \mathbb{R}^N \text{ t.q. } I = f(x^*)$$

Si de plus, f est **différentiable**, ce minimiseur est caractérisé par les *conditions d'optimalité du premier ordre* (connues aussi sous le nom d'equations d'Euler)

$$\nabla f(x^*) = 0 \tag{2}$$

L'equation ci-dessus est en général non-linéaire et donc il n'est pas possible d'avoir une formule explicite de x^* . C'est pourquoi on est amenené à chercher une valeur approchéee de x^*. Or, une méthode classique pour obtenir des valeurs approchées se présente sous la forme d'algorithme de descente (de valeurs de f).

Définition (Gradient à pas fixe):

$$x_{k+1} = x_k - \rho \nabla f(x_k), \quad k \geq 0$$

où $x_0 \in \mathbb{R}^N$.

Théorème: Soit $f : \mathbb{R}^N \to \mathbb{R}$ une application de classe C^1 (différentiable suffit) telle que :

- f est μ-convexe pour un $\mu > 0$,
- ∇f est une application L-lipschitzienne.

Si $0 < \rho < \frac{2\mu}{L^2}$, alors l'algorithme de (GPF) converge linéairement vers l'unique point de minimum x^* de f : il existe une constante $0 < C < 1$, dépendant uniquement de μ, L t.q.

$$\forall k \in \mathbb{N}, \|x_{k+1} - x^*\| \leq C \|x_k - x^*\|.$$

L'algorithme de gradient à pas optimal

Définition (Gradient à pas optimal):

$$x_{k+1} = x_k - \rho_k^* \nabla f(x_k), \quad k \geq 0$$

où $x_0 \in \mathbb{R}^N$ et

$$\rho_k^* = \arg\min_{\rho} f(x_k - \rho \nabla f(x_k)).$$

Théorème: Soit $f : \mathbb{R}^N \to \mathbb{R}$ une application de classe C^1 (différentiable suffit) telle que :

- f est μ-convexe pour un $\mu > 0$,

- ∇f est une application L-lipschitzienne.

Alors l'algorithme de (GPO) converge linéairement vers l'unique point de minimum x^* de f : il existe une constante $0 < C < 1$, dépendant uniquement de μ, L, a et b, t.q.

$$\forall k \in \mathbb{N}, \|x_{k+1} - x^*\| \leq C\|x_k - x^*\|.$$

Appliquons directement ces concepts pour définir le Gradient Boosting

4.3 Application : Gradient Boosting

4.3.1 Cadre

Les arbres de régression et de classification partitionnent l'espace de toutes les valeurs des variables prédictives conjointes en régions disjointes R_j, $j = 1, 2, \ldots, J$, comme représenté par les feuilles de l'arbre. Une constante γ_j est attribuée à chaque région et la règle prédictive est

$$x \in R_j \Rightarrow f(x) = \gamma_j.$$

Ainsi, un arbre peut être formellement exprimé comme

$$T(x; \Theta) = \sum_{j=1}^{J} \gamma_j I(x \in R_j),$$

avec des paramètres $\Theta = \{R_j, \gamma_j\}_{j=1}^{J}$. J est généralement traité comme un hyperparamètre (que l'on détermine à l'avance). Les paramètres sont trouvés en minimisant le risque empirique

$$\hat{\Theta} = \arg\min_{\Theta} \sum_{j=1}^{J} \sum_{x_i \in R_j} L(y_i, \gamma_j). \tag{E1}$$

Cela constitue un problème d'optimisation combinatoire redoutable, et on se contente habituellement de solutions sous-optimales approximatives (cf. Programmation Dynamique). Il est utile de diviser le problème d'optimisation en deux parties :

Trouver γ_j donné R_j : Étant donné les R_j, estimer les γ_j est trivial, et souvent $\hat{\gamma}_j = \bar{y}_j$, la moyenne des y_i dans la région R_j. Pour la perte de classification, $\hat{\gamma}_j$ est la classe modale des observations dans la région R_j (vote à la majorité).

Trouver R_j : C'est la partie où seules des solutions approximatives sont trouvées. Trouver les R_j implique d'estimer aussi les γ_j. Une stratégie typique consiste à utiliser un algorithme de partitionnement récursif glouton descendant pour trouver les R_j. De plus, il est parfois nécessaire d'approximer (E1) par un critère plus lisse et plus pratique pour l'optimisation des R_j :

$$\tilde{\Theta} = \arg\min_{\Theta} \sum_{i=1}^{N} \tilde{L}(y_i, T(x_i, \Theta)). \tag{E2}$$

Ensuite, étant donné les $\hat{R}_j = \tilde{R}_j$, les γ_j peuvent être estimés plus précisément en utilisant le critère original.

On avons décrit une telle stratégie pour les arbres de classification. L'indice de Gini a remplacé la perte de classification dans la croissance de l'arbre (identifiant les R_j). Le modèle d'arbre boosté est une somme de tels arbres,

$$f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m), \tag{E3}$$

induits de manière progressive (Algorithme AdaBoost). À chaque étape de la procédure progressive, il faut résoudre

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)) \tag{E}$$

pour l'ensemble des régions et des constantes $\Theta_m = \{R_{jm}, \gamma_{jm}\}_{j=1}^{J_m}$ du prochain arbre, étant donné le modèle actuel $f_{m-1}(x)$. Étant donné les régions R_{jm}, trouver les constantes optimales γ_{jm} dans chaque région est typiquement simple :

$$\hat{\gamma}_{jm} = \arg\min_{\gamma_{jm}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma_{jm}). \tag{E4}$$

Gradient Boosting

Les algorithmes de Gradient Boosting sont des algorithmes approximatifs rapides pour résoudre (E) avec tout critère de perte différentiable peuvent être dérivés par analogie à l'optimisation numérique. La perte en utilisant $f(x)$ pour prédire y sur les données d'entraînement est

$$L(f) = \sum_{i=1}^{N} L(y_i, f(x_i)). \quad (G1)$$

L'objectif est de minimiser $L(f)$ par rapport à f, où ici $f(x)$ est une somme d'arbres (E3). Ignorant cette contrainte, minimiser (G1) peut être vu comme une optimisation numérique

$$\hat{f} = \arg\min_{f} L(f), \quad (G2)$$

où les "paramètres" $f \in \mathbb{R}^N$ sont les valeurs de la fonction d'approximation $f(x_i)$ à chacun des N points de données x_i:

$$f = \{f(x_1), f(x_2), \ldots, f(x_N)\}^T.$$

Les procédures d'optimisation numérique résolvent (G2) comme une somme de vecteurs composants

$$f_M = \sum_{m=0}^{M} h_m, \quad h_m \in \mathbb{R}^N,$$

où $f_0 = h_0$ est une estimation initiale, et chaque f_m successif est induit sur la base du vecteur de paramètres f_{m-1}, qui est la somme des mises à jour précédentes. Les méthodes d'optimisation numérique diffèrent selon la nature de la fonction pour calculer chaque vecteur d'incrément h_m ("pas").

4.3.2 Descente de Gradient appliquée à chaque étape de l'algorithme

La descente de gradient choisit $h_m = -\rho_m g_m$ où ρ_m est un scalaire et $g_m \in \mathbb{R}^N$ est le gradient de $L(f)$ évalué en $f = f_{m-1}$. Les composantes du gradient g_m sont

$$g_{im} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}. \quad (D1)$$

La longueur du pas ρ_m est la solution de

$$\rho_m = \arg\min_{\rho} L(f_{m-1} - \rho g_m). \quad (D2)$$

On applique donc une descente de gradient "à pas optimal". La solution actuelle est alors mise à jour

$$f_m = f_{m-1} - \rho_m g_m$$

et le processus est répété à l'itération suivante. La descente de gradient peut être vue comme une stratégie très gourmande, puisque $-g_m$ est la direction locale dans $\mathbb{R}^N$ pour laquelle $L(f)$ diminue le plus rapidement en $f = f_{m-1}$.

4.3.3 Gradient Boosting

Le boosting basé sur l'approche par étapes (Algorithme AdaBoost) est aussi une stratégie très gourmande. À chaque étape, l'arbre de solution est celui qui réduit au maximum (E), étant donné le modèle actuel f_{m-1} et ses ajustements $f_{m-1}(x_i)$. Ainsi, les prédictions de l'arbre $T(x_i; \Theta_m)$ sont analogues aux composantes du gradient négatif (D1). La principale différence entre eux est que les composantes de l'arbre $t_m = \{T(x_1; \Theta_m), \ldots, T(x_N; \Theta_m)\}^T$ ne sont pas indépendantes. Elles sont contraintes à être les prédictions d'un arbre de décision à J_m feuilles, tandis que le gradient négatif est la direction de descente maximale sans contrainte.

La solution à (E4) dans l'approche par étapes est analogue à la recherche linéaire (D2) dans la descente de gradient. Néanmoins, (E4) effectue une recherche linéaire séparée pour les composantes de t_m qui correspondent à chaque région terminale séparée $\{T(x_i; \Theta_m)\}_{x_i \in R_{jm}}$.

Si minimiser la perte sur les données d'entraînement (G1) était le seul objectif, la descente de gradient serait la stratégie préférée. Le gradient (D1) est trivial à calculer pour toute fonction de perte différentiable $L(y, f(x))$, tandis que résoudre (E) est difficile pour les critères robustes. Malheureusement, le gradient (D1) est défini uniquement aux points de données d'entraînement x_i, alors que l'objectif est de généraliser $f_M(x)$ à de nouvelles données non représentées dans l'ensemble d'entraînement.

Une possible résolution de ce problème est d'induire un arbre $T(x; \Theta_m)$ à la m-ème itération dont les prédictions t_m sont aussi proches que possible du gradient négatif. En utilisant l'erreur quadratique pour mesurer la proximité, cela nous mène à

$$\tilde{\Theta}_m = \arg\min_{\Theta} \sum_{i=1}^{N} (-g_{im} - T(x_i; \Theta))^2. \quad (G3)$$

En gros, on ajuste l'arbre T aux valeurs du gradient négatif (D1) par moindres carrés. Et des algorithmes rapides existent pour l'induction d'arbres de décision par moindres carrés. Bien que les régions de solution $\tilde{R}_{jm}$ de (G3) ne soient pas identiques aux régions R_{jm} qui résolvent (E), elles sont généralement assez similaires pour servir le même but. En tout cas, la procédure de boosting par étapes et l'induction d'arbres de décision de haut en bas sont elles-mêmes des procédures d'approximation. Après avoir construit l'arbre (G3), les constantes correspondantes dans chaque région sont données par (E4).

4.4 Implémentations du Gradient Boosting

L'algorithme (1) présente l'algorithme générique de Gradient Boosting pour la régression. Des algorithmes spécifiques sont obtenus en insérant différents critères de perte $L(y, f(x))$. La première ligne de l'algorithme initialise la prédiction optimale pour un arbre à un seul nœud terminal. Les composantes du gradient négatif calculées lors des itérations sont appelées pseudo-résidus, r.

Algorithm 1 Algorithme de Gradient Boosting pour la Régression

0: **Initialisation:** $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
0: **for** $m = 1$ to M **do**
0: **for** $i = 1$ to N **do**
0: Calculer $r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$
0: **end for**
0: Ajuster un arbre de régression aux cibles r_{im} donnant les régions terminales $R_{jm}, j = 1, 2, \ldots, J_m$
0: **for** $j = 1$ to J_m **do**
0: Calculer $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$
0: **end for**
0: Mettre à jour $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$
0: **end for**
0: **Sortie:** $\hat{f}(x) = f_M(x) = 0$

Pour la classification, la fonction de perte est la déviance multinomiale et K arbres sont construits à chaque itération. Chaque arbre T_{km} est ajusté à son vecteur de gradient négatif respectif g_{km},

$$-g_{ikm} = \left[\frac{\partial L(y_i, f_1(x_i), \ldots, f_K(x_i))}{\partial f_k(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)} = I(y_i = G_k) - p_k(x_i),$$

avec $p_k(x)$ donné par le modèle logistique. Bien que K arbres séparés soient construits à chaque itération, ils sont reliés par le modèle logistique. Et on le répète K fois à chaque itération m. Le résultat à la ligne 3 est K expansions d'arbres différentes (couplées) $f_{kM}(x), k = 1, 2, \ldots, K$. Elles produisent des probabilités ou font de la classification.

Les deux paramètres à définir sont le nombre d'itérations M et les tailles de chacun des arbres constituants $J_m, m = 1, 2, \ldots, M$.

Algorithm 2 Algorithme de Gradient Boosting pour la Classification

0: **Initialisation:** $f_{k0}(x) = 0$, pour $k = 1, 2, \ldots, K$.
0: **for** $m = 1$ to M **do**
0: Mettre à jour les probabilités $p_k(x) = \frac{e^{f_k(x)}}{\sum_{\ell=1}^{K} e^{f_\ell(x)}}$, pour $k = 1, 2, \ldots, K$.
0: **for** $k = 1$ to K **do**
0: Calculer les résidus $r_{ikm} = y_{ik} - p_k(x_i)$, pour $i = 1, 2, \ldots, N$.
0: Ajuster un arbre de régression aux cibles r_{ikm} pour obtenir les régions terminales R_{jkm}, $j = 1, 2, \ldots, J_m$.
0: Calculer $\gamma_{jkm} = \frac{K-1}{K} \frac{\sum_{x_i \in R_{jkm}} r_{ikm}}{\sum_{x_i \in R_{jkm}} |r_{ikm}|(1-|r_{ikm}|)}$, pour $j = 1, 2, \ldots, J_m$.
0: Mettre à jour $f_{km}(x) = f_{k,m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jkm} I(x \in R_{jkm})$.
0: **end for**
0: **end for**
0: **Sortie:** $\hat{f}_k(x) = f_{kM}(x)$, pour $k = 1, 2, \ldots, K$. =0

4.5 Arbres de Taille Appropriée pour le Boosting

Historiquement, le boosting était considéré comme une technique pour combiner des modèles. La taille optimale de chaque arbre est estimée séparément de manière habituelle lorsqu'il est construit. Un arbre (surdimensionné) est d'abord induit, puis on l'élague au nombre optimal de feuilles. Le résultat est que les arbres ont tendance à être beaucoup trop grands, surtout lors des premières itérations. Cela dégrade les performances, augmente le risque de sur-ajustement et augmente le temps de calcul.

Pour éviter ce problème, on restreint tous les arbres à la même taille, $J_m = J, \forall m$. À chaque itération, un arbre de régression avec J feuilles est induit. Ainsi, J devient un hyperparamètre à ajuster pour maximiser les performances estimées pour les données disponibles.

Les valeurs utiles pour J en considérant les propriétés de la fonction "cible" :

$$\eta = \arg\min_f \mathbb{E}_{X,Y} L(Y, f(X)).$$

La fonction cible $\eta(x)$ est celle avec le risque de prédiction minimum sur les prédiction. C'est la fonction à approximer

propriété : $\eta(X)$ est le degré auquel les variables de coordonnées $X_T = (X_1, X_2, \ldots, X_p)$ interagissent entre elles :

$$\eta(X) = \sum_j \eta_j(X_j) + \sum_{jk} \eta_{jk}(X_j, X_k) + \sum_{jkl} \eta_{jkl}(X_j, X_k, X_l) + \cdots .$$

La première somme concerne les fonctions d'une seule variable prédictive X_j. Les fonctions particulières $\eta_j(X_j)$ sont celles qui approximent au mieux $\eta(X)$ selon le critère de perte utilisé. Chaque $\eta_j(X_j)$ est appelée l'effet principal de X_j.

La deuxième somme concerne les fonctions de deux variables qui, ajoutées aux effets principaux,

ajustent au mieux $\eta(X)$. Celles-ci sont appelées les interactions de second ordre de chaque paire de variables (X_j, X_k).

La troisième somme représente les interactions de troisième ordre, et ainsi de suite (similaire à l'ANOVA à x facteurs). Les effets d'interaction de bas niveau tendent à dominer. Lorsque c'est le cas, les modèles produisant des effets d'interaction de haut niveau (Grands arbres de décision) perdent en précision.

Le niveau d'interaction des approximations basées sur les arbres est limité par la taille de l'arbre J. Aucun effet d'interaction de niveau supérieur à $J - 1$ n'est possible. Étant donné que les modèles boostés sont additifs dans les arbres (E3), cette limite s'étend à eux aussi. Fixer $J = 2$ produit des modèles boostés avec seulement des effets principaux; aucune interaction n'est permise. Avec $J = 3$, les effets d'interaction de deux variables sont également autorisés, et ainsi de suite. Cela suggère que la valeur choisie pour J devrait refléter le niveau des interactions dominantes de $\eta(x)$. La fonction est additive, donc les modèles boostés avec $J > 2$ engendrent une grande variance, donc une erreur de test plus élevée et en conséquence, un risque de sur-ajustement plus élevé.

5 XGBoost

5.1 Introduction

XGBoost est apparu en 2015 et il s'est imposé rapidement comme un des algorithmes les plus efficaces en Machine Learning. Ses principales caractéristiques :
- Utilisé pour de régression ou la classification
- Algorithme d'apprentissage supervisé, qui a besoin d'un jeu de données d'entraînement qui construira un modèle qui pourra être ensuite généralisé
- Fait partie des algorithmes "Ensemble Learning" qui impliquent l'utilisation de multiples arbres de décision pour construire la prédiction.

Il est particulièrement performant :
- Dans sa capacité à généraliser car intégrant dans sa construction des mécanismes de régularisation assez puissants et astucieux
- Sa capacité à traiter les données manquantes, sans pour autant dégrader ses performances
- Sa rapidité de calcul sur des gros volumes en faisant des approximations élégantes lors de la construction des arbres de décision.

Cet algorithme OpenSource a démontré son impact dans les compétitions d'apprentissage automatique, comme Kaggle et KDDCup, en étant le choix de nombreux gagnants.
XGBoost offre des résultats de pointe sur des problèmes diverses et son succès réside dans sa scalabilité (extensibilité). Il est plus de dix fois plus rapide que les solutions existantes sur une seule machine et s'étend à des milliards d'exemples dans des environnements distribués ou à mémoire limitée, grâce à des optimisations algorithmiques et systémiques innovantes.

Nous verrons dans ce document les principes qui gouvernent cet algorithme et qui le rendent si performant. Mais avant cela, faisons un rappel d'optimisation mathématique d'où se base l'execution de XGBoost

5.2 Optimisation : Algorithme de descente de Newton

5.2.1 La direction de Newton

Une direction qui permet d'acquérir un meilleur taux de convergence est la direction de Newton, qui approxime la fonction à l'ordre 2.

Exemple (La direction de Newton): Soit $f : \mathbb{R}^N \to \mathbb{R}$ deux fois différentiable et $x \in \mathbb{R}^N$ un point noncritique tel que $H_f(x) \neq 0$. Alors

$$d = -(H_f(x))^{-1}\nabla f(x)$$

est une direction de descente de f au point x.

En effet

$$\langle \nabla f(x), d\rangle = \langle \nabla f(x), -(Hf(x))^{-1}\nabla f(x)\rangle = \langle Hf(x)(Hf(x))^{-1}\nabla f(x), -(Hf(x))^{-1}\nabla f(x)\rangle = -\langle Hf(x)d, d\rangle \leq -\lambda_{\min}(Hf$$

Proposition : Soit $f : \mathbb{R}^N \to \mathbb{R}$ de classe C^2 et $x \in \mathbb{R}^N$ un point noncritique tel que $Hf(x) \neq 0$. Alors

$$d = -(Hf(x))^{-1}\nabla f(x)$$

minimise l'approximation de Taylor à l'ordre 2 :

$$-(Hf(x))^{-1}\nabla f(x) = \arg\min_{d\in\mathbb{R}^N}\left(f(x) + \langle \nabla f(x), d\rangle + \frac{1}{2}\langle Hf(x)d, d\rangle\right).$$

Preuve : Soit $x \in \mathbb{R}^N$ un point noncritique de f tel que $H(x) \neq 0$. On pose $\psi : \mathbb{R}^N \to \mathbb{R}$

$$\psi(d) = f(x) + \langle \nabla f(x), d\rangle + \frac{1}{2}\langle Hf(x)d, d\rangle.$$

On est en train de minimiser une fonction quadratique de matrice hessienne

$$H\psi(d) = Hf(x) \neq 0$$

donc elle est fortement convexe et admet un unique minimiseur. Ce minimiseur est caractérisé par la condition d'optimalité du premier ordre, qui est ici :

$$\nabla\psi(d) = \nabla f(x) + Hf(x)d = 0.$$

Donc le minimiseur est $d = -(Hf(x))^{-1}\nabla f(x)$. □

5.2.2 Les algorithmes de Newton

Cela conduit aux algorithmes de Newton.

Définition (Méthode de Newton) :

$$\begin{cases} x_{k+1} = x_k - (Hf(x_k))^{-1}\nabla f(x_k) & k \geq 0 \\ x_0 \in \mathbb{R}^N. \end{cases}$$

(N)

Théorème : Soit $f : \mathbb{R}^N \to \mathbb{R}$ une application de classe $C^2(\mathbb{R}^N)$ telle que

- f est μ-convexe pour un $\mu > 0$
- Hf est une application K-lipschitzienne

Si $\frac{K}{2\mu} < 1$, alors la méthode de Newton (N) converge de façon quadratique vers l'unique point de minimum x^* de f : il existe une constante $0 < C < 1$ telle que

$$\forall k \in \mathbb{N}, \|x_{k+1} - x^*\| \leq C\|x_k - x^*\|^2.$$

Preuve : Puisque x^* est un point de minimum de f, on sait que $\nabla f(x^*) = 0$. On peut donc écrire

$$\begin{aligned} x_{k+1} - x^* &= x_k - (Hf(x_k))^{-1}\nabla f(x_k) - x^* \\ &= (x_k - x^*) - (Hf(x_k))^{-1}(\nabla f(x_k) - \nabla f(x^*)) \\ &= (Hf(x_k))^{-1}[Hf(x_k)(x_k - x^*) - (\nabla f(x_k) - \nabla f(x^*))]. \end{aligned}$$

D'où

$$\|x_{k+1} - x^*\| \leq \|(Hf(x_k))^{-1}\|\|Hf(x_k)(x_k - x^*) - (\nabla f(x_k) - \nabla f(x^*))\|.$$

Lemme :

$$\|(Hf(x))^{-1}\| \leq \frac{1}{\mu}.$$

La fonction étant μ-convexe, on a, pour tout $x \in \mathbb{R}^N$

$$Hf(x) \geq \mu I_N \Leftrightarrow \lambda_{\min}(Hf(x)) \geq \mu.$$

L'inverse de la hessienne $(Hf(x))^{-1}$ aura comme valeurs propres, les inverses des valeurs propres de $Hf(x)$. En particulier, on a

$$\lambda_{\max}((Hf(x))^{-1}) = \frac{1}{\lambda_{\min}(Hf(x))} \leq \frac{1}{\mu}.$$

Or, on sait que pour une matrice symétrique la norme d'opérateur coïncide avec son rayon spectral :

$$\|(Hf(x))^{-1}\| = \rho((Hf(x))^{-1}) = \lambda_{\max}((Hf(x))^{-1}) \leq \frac{1}{\mu}.$$

Lemme :

$$\|Hf(x_k)(x_k - x^*) - (\nabla f(x_k) - \nabla f(x^*))\| \leq \frac{K}{2}\|x_k - x^*\|^2.$$

Par le théorème de Taylor à l'ordre 1 pour ∇f on a

$$\nabla f(x_k) - \nabla f(x^*) = \int_0^1 Hf(x_k + t(x_k - x^*))(x_k - x^*)\, dt.$$

En même temps, on a

$$Hf(x_k)(x_k - x^*) = \int_0^1 Hf(x_k)(x_k - x^*)\, dt.$$

On en déduit, en utilisant la condition de Lipschitz pour $Hf(x)$, que

$$\begin{aligned}
&\|Hf(x_k)(x_k - x^*) - (\nabla f(x_k) - \nabla f(x^*))\| \\
&\leq \| \int_0^1 [Hf(x_k) - Hf(x_k + t(x_k - x^*))](x_k - x^*)\, dt\| \\
&\leq \int_0^1 K\|t(x_k - x^*)\|\|x_k - x^*\|\, dt = \frac{K}{2}\|x_k - x^*\|^2.
\end{aligned}$$

Conclusion Par les deux lemmes précédents, on en déduit que

$$\begin{aligned}
\|x_{k+1} - x^*\| &\leq \|(Hf(x_k))^{-1}\|\|Hf(x_k)(x_k - x^*) - (\nabla f(x_k) - \nabla f(x^*))\| \\
&\leq \frac{1}{\mu}\frac{K}{2}\|x_k - x^*\|^2.
\end{aligned}$$

C'est enfin pour la convergence qu'intervient la condition $\frac{K}{2\mu} < 1$. □

Exemple :

Soit $f : \mathbb{R}^N \to \mathbb{R}$ une fonction quadratique

$$f(x) = \frac{1}{2}\langle Ax, x\rangle + \langle b, x\rangle + c$$

avec $A \in S^N(\mathbb{R})$ et $A \geq 0$. Alors f est de classe $C^2(\mathbb{R}^N)$ et

- f est $\lambda_{\min}(A)$-convexe.
- $Hf = A$ est une application $\lambda_{\max}(A)$-lipschitzienne :

$$\sup_{x \neq y} \frac{\|Ax - Ay\|}{\|x - y\|} = \sup_{z \neq 0} \frac{\|Az\|}{\|z\|} = \|A\| = \lambda_{\max}(A).$$

Alors, si $\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} < 2$, la méthode de Newton (N) converge. On appelle le rapport $\kappa(A)$ le conditionnement de la matrice A.

5.2.3 GPF versus Newton

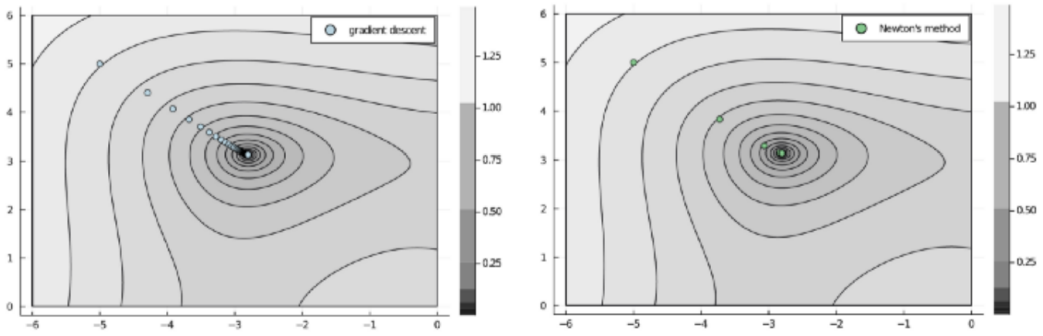


Figure 5: Descente de gradient par rapport à la méthode de Newton pour minimiser une fonction. À partir du point (5, 5), la descente de gradient converge vers le minimum en 229 étapes, alors que la méthode de Newton le fait en seulement 6.

Comparaison de la convergence

Convergence quadratique vs linéaire

L'algorithme de Newton a une convergence quadratique près de l'optimum. Cela signifie que l'erreur diminue quadratiquement à chaque itération, ce qui rend la convergence très rapide une fois que l'on est proche du minimum.

La descente de gradient, en revanche, a une convergence linéaire, ce qui signifie que l'erreur diminue proportionnellement à chaque itération. Cela peut rendre la convergence lente, surtout lorsque le taux d'apprentissage (learning-rate) est mal choisi ou que la surface de la fonction est très plate.

Utilisation de la matrice Hessienne

L'algorithme de Newton utilise la matrice Hessienne, ce qui lui permet de prendre en compte la courbure locale de la fonction. Cela permet de faire des pas plus grands et plus informés vers le minimum, en particulier dans les vallées courbées et les surfaces complexes.

La descente de gradient n'utilise que le gradient, ignorant la courbure de la fonction. Cela peut conduire à des mises à jour inefficaces, surtout dans les problèmes où les directions de courbure varient considérablement.

Adaptation automatique de la taille des pas

En utilisant l'inverse de la matrice Hessienne, l'algorithme de Newton adapte automatiquement la taille et la direction des pas en fonction de la courbure locale de la fonction, ce qui est particulièrement utile dans les fonctions où la courbure varie dans différentes directions.

La descente de gradient nécessite un ajustement manuel du taux d'apprentissage, qui doit être soigneusement choisi. Si le taux est trop grand, l'algorithme peut diverger ; s'il est trop petit, la convergence peut être extrêmement lente.

Limitations

Cependant, l'algorithme de Newton a aussi ses propres limitations :

- Le calcul et l'inversion de la matrice Hessienne peuvent être coûteux en termes de temps et de mémoire, surtout pour les fonctions de grande dimension.
- Il peut ne pas être efficace pour les fonctions qui ne sont pas bien approximées par une parabole locale ou pour les problèmes mal conditionnés.

5.3 Application

5.3.1 Eléments de l'algorithme de Newton appliqué aux Arbres de décision

Comme cité plus haut, l'algorithme de Newton-Raphson est une méthode itérative pour trouver les zéros ou les extrémums d'une fonction. En optimisation, il est utilisé pour minimiser une fonction en utilisant à la fois le gradient et la Hessienne de la fonction de coût. La mise à jour des paramètres est donnée par :

$$\theta_{t+1} = \theta_t - H^{-1}(\theta_t)\nabla f(\theta_t)$$

où:

- θ_t est le vecteur des paramètres à l'itération t,
- $\nabla f(\theta_t)$ est le gradient de la fonction de coût,
- $H(\theta_t)$ est la Hessienne de la fonction de coût.

XGBoost utilise une forme avancée de gradient boosting qui intègre des concepts de l'algorithme de Newton pour optimiser la fonction de coût additive. Voici comment ces concepts se traduisent mathématiquement dans XGBoost :

5.3.2 Fonction de Coût

La fonction de coût pour un arbre de décision à l'étape t est donnée par :

$$L^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

où:

- l est la fonction de perte,
- $\hat{y}_i^{(t-1)}$ est la prédiction à l'itération précédente,
- $f_t(x_i)$ est l'arbre ajouté à l'itération t,
- Ω est le terme de régularisation.

5.3.3 Approximation de Taylor

Pour simplifier l'optimisation, XGBoost utilise une approximation de Taylor de second ordre de la fonction de coût :

$$L^{(t)} \approx \sum_{i=1}^{n} \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t)$$

où:

- $g_i = \partial_{\hat{y}} l(y_i, \hat{y}_i^{(t-1)})$ est le gradient de la fonction de perte,
- $h_i = \partial_{\hat{y}}^2 l(y_i, \hat{y}_i^{(t-1)})$ est le Hessien de la fonction de perte.

Démontrons cette expression :

Rappels sur l'expression de Taylor

Pour toute fonction f, il est possible de l'approximer au voisinage d'une valeur a :

$$f(x) \approx f(a) + (x-a)f'(a) + \cdots + \frac{(x-a)^n}{n!} f^{(n)}(a)$$

Pour x proche de a. Cette approximation prend une forme polynomiale de degré n.

Fonction objectif à l'itération t

Pour un ensemble de données d'apprentissage avec n observations et d variables explicaives $D = \{(x_i, y_i)\}$ ($|D| = n, x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$), un modèle d'ensemble d'arbres utilise T fonctions additives pour prédire la sortie :

$$\hat{y}_i = \phi(x_i) = \sum_{j=1}^{T} f_j(x_i), f_j \in F$$

où $F = \{f(x) = w_{q(x)}\}(q : \mathbb{R}^m \to T, w \in \mathbb{R}^T)$ est l'espace des arbres de régression. Ici, q représente la structure de chaque arbre qui mappe un exemple à l'index de la feuille correspondante. T_j est le nombre de feuilles dans l'arbre j. Chaque f_j correspond à une structure d'arbre indépendante q et des poids de feuilles w. Contrairement aux arbres de décision, chaque arbre de régression contient un score continu sur chaque feuille. Pour un exemple donné, nous utilisons les règles de décision dans les arbres pour le classifier dans les feuilles et calculer la prédiction finale en additionnant les scores dans les feuilles correspondantes.

On a la forme de la fonction objectif à l'itération t (obtenue par récurrence)

$$\mathcal{L}_t = \sum_{i=1}^{n} l(\hat{y}_i^{(t-1)} + \phi_t(x_i), y_i) + \Omega(A_t)$$

Supposons que la quantité $\phi_t(x_i)$ est suffisamment petite pour pouvoir appliquer la formule de Taylor au degré 2:

$$\mathcal{L}_t \approx \sum_{i=1}^{n} \left[l(\hat{y}_i^{(t-1)}, y_i) + g_i \cdot \phi_t(x_i) + \frac{1}{2} h_i \cdot \phi_t^2(x_i) \right] + \Omega(A_t)$$

où

$$g_i = \frac{\partial}{\partial \hat{y}_i^{(t-1)}} l(\hat{y}_i^{(t-1)}, y_i) \quad \text{(gradient)}$$

et

$$h_i = \frac{\partial^2}{\partial (\hat{y}_i^{(t-1)})^2} l(\hat{y}_i^{(t-1)}, y_i) \quad \text{(hessienne)}$$

Simplification de la fonction objectif

En passant de:

$$\sum_{i=1}^{n} l(\hat{y}_i^{(t-1)} + \phi_t(x_i), y_i)$$

à

$$\sum_{i=1}^{n} l(\hat{y}_i^{(t-1)}, y_i) + g_i \cdot \phi_t(x_i) + \frac{1}{2} h_i \cdot \phi_t^2(x_i)$$

on obtient que $\mathcal{L}$ dépend uniquement de $t-1$ (de par la relation de récurrence). Cette approximation permet de simplifier les calculs.

Soit $\hat{\phi_t}$ l'expression à minimiser à l'étape t:

$$\hat{\phi_t} = \arg\min_{\phi_t} \sum_i \left[l(\hat{y}_i^{(t-1)}, y_i) + g_i \cdot \phi_t(x_i) + \frac{1}{2} h_i \cdot \phi_t^2(x_i) \right] + \Omega(A_t)$$

À l'étape t, $l(\hat{y}_i^{(t-1)}, y_i)$ est une constante car optimale a l'étape $t-1$. On peut donc l'enlever de $\hat{\phi_t}$

$$\hat{\phi_t} = \arg\min_{\phi_t} \sum_i \left[g_i \cdot \phi_t(x_i) + \frac{1}{2} h_i \cdot \phi_t^2(x_i) \right] + \Omega(\text{arbre}_t)$$

Calcul de la fonction objectif avec régularisation

On revient désormais au calcul de la fonction objectif, en précisant la régularisation Ω_t pour l'arbre t:

$$\mathcal{L}_t = \sum_{i=1}^{n} \left[g_i \cdot \phi_t(x_i) + \frac{1}{2} h_i \cdot \phi_t^2(x_i) \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

on développe les sommes:

$$\mathcal{L}_t = \sum_{i=1}^{n} g_i \cdot \phi_t(x_i) + \frac{1}{2} \sum_{i=1}^{n} h_i \cdot \phi_t^2(x_i) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

On a un problème car les sommes ne portent pas sur les mêmes objets (deux sommes sur les observations et une somme sur les arbres).

On doit alors exprimer les sommes sur toutes les observations par des sommes qui portent sur les feuilles de l'arbre, donc inverser les sommes en parcourant toutes les feuilles et en identifiant quelle observation tombe sur cette feuille. On factorise alors la valeur d'une feuille ω_j pour chacune des feuilles et on "inverse" les sommes.

Autrement dit:

$$\sum_{i=1}^{n} g_i \cdot \phi_t(x_i) = g_1 \cdot \phi_t(x_1) + g_2 \cdot \phi_t(x_2) + \cdots + g_n \cdot \phi_t(x_n)$$

avec

$$\phi_t(x_i) = w_j \quad \text{si on tombe sur la feuille } j \text{ de l'arbre de valeur } w_j$$

On peut donc factoriser par rapport à chacune des feuilles de l'arbre et sommer tous les gradients g_i qui tombent sur la même feuille j de l'arbre:

$$\sum_{i=1}^{n} g_i \cdot \phi_t(x_i) = \sum_{j=1}^{T} \left[\left(\sum_{i \in \mathcal{F}_j} g_i \right) \cdot w_j \right]$$

Car $\phi_t(x_i) = w_j$ par définition.

De la même manière:

$$\frac{1}{2} \sum_{\text{instances}} h_i \cdot \phi_t^2(x_i) = \frac{1}{2} \sum_{j=1}^{T} \left[\left(\sum_{i \in \mathcal{F}_j} h_i \right) \cdot w_j^2 \right]$$

Donc :

$$\mathcal{L}_t = \sum_{j=1}^{T} \left[\left(\sum_{i \in \mathcal{F}_j} g_i \right) \cdot w_j + \frac{1}{2} \left(\sum_{i \in \mathcal{F}_j} h_i \right) \cdot w_j^2 + \frac{1}{2} \lambda \cdot w_j^2 \right] + \gamma T$$

Avec γ et λ les hyperparamètres à définir.

Enfin :

$$\mathcal{L}_t = \sum_{j=1}^{T} \left[\left(\sum_{i \in \mathcal{F}_j} g_i \right) \cdot w_j + \frac{1}{2} \left(\sum_{i \in \mathcal{F}_j} (h_i + \lambda) \right) \cdot w_j^2 \right] + \gamma T$$

On peut maintenant trouver la valeur $\hat{w_j}$ optimale pour l'arbre T. Pour les trouver, on cherche les points stationnaires.

$$\hat{w_j} = \arg\min_{w_j} \mathcal{L}_t \implies \left. \frac{\partial \mathcal{L}_t}{\partial w_j} \right|_{w_j = \hat{w_j}} = 0$$

$$\frac{\partial \mathcal{L}_t}{\partial w_j} = 0 \implies \frac{\partial}{\partial w_j} \sum_{j=1}^{T} \left[\left(\sum_{i \in \mathcal{F}_j} g_i \right) \cdot w_j + \frac{1}{2} \left(\sum_{i \in \mathcal{F}_j} (h_i + \lambda) \right) \cdot w_j^2 \right] + \frac{\partial}{\partial w_j} \gamma T = 0$$

Or pour toutes les feuilles $\neq j$ (car indépendants de w_j)

$$\frac{\partial}{\partial w_j}\left[\left(\sum_{i\in\mathcal{F}_j} g_i\right)\cdot w_j + \frac{1}{2}\left(\sum_{i\in\mathcal{F}_j}(h_i+\lambda)\right)\cdot w_j^2 + \gamma T\right] = 0$$

Donc:

$$\frac{\partial}{\partial w_j}\left[\left(\sum_{i\in\mathcal{F}_j} g_i\right)\cdot w_j + \frac{1}{2}\left(\sum_{i\in\mathcal{F}_j}(h_i+\lambda)\right)\cdot w_j^2 + \gamma T\right] = 0$$

Soit:

$$\left(\sum_{i\in\mathcal{F}_j} g_i\right) + \left(\sum_{i\in\mathcal{F}_j}(h_i+\lambda)\right)\cdot w_j = 0$$

Car:

$$\frac{\partial}{\partial w_j}\gamma T = 0 \quad \text{Indépendant de } w_j$$

Et on obtient la valeur optimale d'une feuille quand l'arbre est construit :

$$\hat{w_j} = -\frac{\sum_{i\in\mathcal{F}_j} g_i}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)}$$

Calcul de la valeur de la fonction de perte pour un arbre

On a:

$$\mathcal{L}_t = \sum_{j=1}^{T}\left[\left(\sum_{i\in\mathcal{F}_j} g_i\right)\cdot w_j + \frac{1}{2}\left(\sum_{i\in\mathcal{F}_j}(h_i+\lambda)\right)\cdot w_j^2\right] + \gamma T$$

Remplaçons w_j par sa valeur:

$$\mathcal{L}_t = \sum_{j=1}^{T}\left[\left(\sum_{i\in\mathcal{F}_j} g_i\right)\cdot -\frac{\sum_{i\in\mathcal{F}_j} g_i}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)} + \frac{1}{2}\left(\sum_{i\in\mathcal{F}_j}(h_i+\lambda)\right)\cdot\left(-\frac{\sum_{i\in\mathcal{F}_j} g_i}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)}\right)^2\right] + \gamma T$$

$$\mathcal{L}_t = \sum_{j=1}^{T}\left[-\frac{(\sum_{i\in\mathcal{F}_j} g_i)^2}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)} + \frac{1}{2}\cdot\frac{(\sum_{i\in\mathcal{F}_j} g_i)^2}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)}\right] + \gamma T$$

$$\mathcal{L}_t = \sum_{j=1}^{T}\left[-\frac{1}{2}\cdot\frac{(\sum_{i\in\mathcal{F}_j} g_i)^2}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)}\right] + \gamma T$$

Soit:

$$\mathcal{L}_t = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in\mathcal{F}_j} g_i)^2}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)} + \gamma T$$

Cette formule constitue la fonction de perte pour l'arbre construit à l'étape t. Plus elle est petite, meilleure sera la contribution de cet arbre. Cette formule nous servira le split optimal pour l'arbre, c'est-à-dire le critère à utiliser pour construire une nouvelle branche décisionnelle.

5.3.4 Détermination des splits pour la construction de l'arbre

Avec la formule de perte pour un arbre donné, il sera simple de déterminer les critères permettant de construire une nouvelle branche de l'arbre. Pour expliquer cet algorithme, on prend l'hypothèse que l'ensemble des features et des valeurs sont testées, pour évaluer la pertinence du nouveau split.

Construisons l'arbre de manière iterative. À l'étape 0, on a un arbre avec une seule feuille et une seule valeur. On veut savoir s'il est nécessaire ou non d'effectuer un nouveau split, et si oui, comment. Puis on répète le processus à chaque étape de la construction de l'arbre.

Devons nous faire un nouveau split ?

Pour le savoir, on se base sur la fonction de perte. Notre but est de diminuer la perte donc les deux nouveaux noeuds doivent diminuer la perte globale. Pour cela, on calcule :

- Perte de la feuille de Gauche ($\mathcal{F}_{jg}$) + Perte de la feuille de droite ($\mathcal{F}_{jd}$)
- Perte de la feuille initiale ($\mathcal{F}_j$)

On rappelle que la régularisation est intégrée dans les fonctions de perte de chacune des feuilles pour empêcher l'arbre de trop se complexifier.

Pour que le split soit intéressant, on vérifie que la somme pénalisée des fonctions de perte des deux nouvelles feuilles est inférieure à celle de la feuille initiale. En effet, si on a plus de perte avec le nouveau split, celui-ci est alors inutile et on peut alors l'élaguer.

C'est-à-dire :

$$\mathcal{L}_{\mathcal{F}_{jg}} + \mathcal{L}_{\mathcal{F}_{jd}} < \mathcal{L}_{\mathcal{F}_j}$$

On connaît l'expression de la perte pour un arbre (somme de la perte sur toutes ses feuilles) :

$$\mathcal{L}_t = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i + \lambda} + \gamma T$$

On en déduit la perte calculée sur une seule feuille :

$$-\frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i + \lambda} + \gamma$$

Les termes de pénalisation λ et γ sont pris en compte.
L'équation devient alors :

$$-\frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_{jg}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{jg}} h_i + \lambda} + \gamma - \frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_{jd}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{jd}} h_i + \lambda} + \gamma < -\frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i + \lambda} + \gamma$$

Alors :

$$-\frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_{jg}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{jg}} h_i + \lambda} - \frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_{jd}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{jd}} h_i + \lambda} + 2\gamma < -\frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i + \lambda}$$

Et :

$$-\frac{\left(\sum_{i\in\mathcal{F}_{j\,g}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{j\,g}} h_i+\lambda}-\frac{\left(\sum_{i\in\mathcal{F}_{j\,d}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{j\,d}} h_i+\lambda}+2\gamma<-\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i+\lambda}$$

Donc :

$$\frac{\left(\sum_{i\in\mathcal{F}_{j\,g}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{j\,g}} h_i+\lambda}+\frac{\left(\sum_{i\in\mathcal{F}_{j\,d}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{j\,d}} h_i+\lambda}-\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i+\lambda}-2\gamma>0$$

Un split, transformant une feuille de l'arbre en deux nouvelles feuilles, peut donc être réalisé si cette relation est satisfaite. Dans ce cas, le nouveau split va faire converger la prédiction vers la solution, tout en respectant les contraintes de régularisation !

Application pour la régression

Rappelons les résultats obtenus jusqu'à présent. Pour un arbre donné

- La valeur optimale des feuilles est la suivante :

$$\hat{w_j}=-\frac{\sum_{i\in\mathcal{F}_j} g_i}{\sum_{i\in\mathcal{F}_j}(h_i+\lambda)}$$

- La perte sur une seule feuille est :

$$-\frac{1}{2}\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i+\lambda}+\gamma$$

- La condition de split est la suivante :

$$\frac{\left(\sum_{i\in\mathcal{F}_{j\,g}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{j\,g}} h_i+\lambda}+\frac{\left(\sum_{i\in\mathcal{F}_{j\,d}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{j\,d}} h_i+\lambda}-\frac{\left(\sum_{i\in\mathcal{F}_j} g_i\right)^2}{\sum_{i\in\mathcal{F}_j} h_i+\lambda}-2\gamma>0$$

  avec g_i gradient de la fonction de perte et h_i la hessienne de la fonction de perte $\mathcal{L}$.

Définissons $\mathcal{L}$ pour la régression comme :

$$\mathcal{L}(\hat{y}_i,y_i)=\frac{1}{2}\times\sum_{i=1}^{n}(\hat{y}_i-y_i)^2$$

Calculons le gradient pour cette fonction :

$$g_i=\frac{\partial L(\hat{y}_i,y_i)}{\partial\hat{y}_i}=\sum_{i=1}^{n}(\hat{y}_i-y_i)\quad(\text{ - somme des résidus} = y_i-\hat{y}_i)$$

Calculons la hessienne pour cette fonction

$$h_i = \frac{\partial^2 L(\hat{y}_i, y_i)}{\partial \hat{y}_i^2} = \sum_{i=1}^{n} 1 = |\mathcal{F}_j|$$

On obtient :

- La valeur optimale des feuilles :

$$w_j = -\frac{\sum_{i \in \mathcal{F}_j} (\hat{y}_i - y_i)}{h_j + \lambda}$$

- La perte sur une seule feuille :

$$-\frac{1}{2} \cdot \frac{\sum_{i \in \mathcal{F}_j} (\hat{y}_i - y_i)^2}{|\mathcal{F}_j| + \lambda} + \gamma$$

- La notion de score permettant de trouver le split optimal :

$$\text{Score} = \frac{(\sum_{i \in \mathcal{F}_j} y_i - \hat{y}_i)^2}{|\mathcal{F}_j| + \lambda}\Big|_{\mathcal{F}_{j\,g}} + \frac{(\sum_{i \in \mathcal{F}_j} y_i - \hat{y}_i)^2}{|\mathcal{F}_j| + \lambda}\Big|_{\mathcal{F}_{j\,d}} - \frac{(\sum_{i \in \mathcal{F}_j} y_i - \hat{y}_i)^2}{|\mathcal{F}_j| + \lambda}\Big|_{\mathcal{F}_j} - 2 \cdot \gamma$$

Application pour la classification

Définissons la fonction de perte pour la classification, comme étant la fonction logistique

$$L = -\sum_{i \in \mathcal{F}_j} (y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)) \quad \text{avec } p_i = \frac{1}{1 + e^{-\hat{y}_i}}$$

Calculons le gradient et le hessien pour cette fonction,

- Gradient:

$$g_i = \frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} = \frac{\partial L(\hat{y}_i, y_i)}{\partial p_i} \cdot \frac{\partial p_i}{\partial \hat{y}_i}$$

- Premier terme du produit:

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial p_i} = -\sum_{i \in \mathcal{F}_j} \left(\frac{y_i}{p_i} - \frac{1 - y_i}{1 - p_i} \right) = -\sum_{i \in \mathcal{F}_j} \left(\frac{y_i(1 - p_i) - (1 - y_i)p_i}{p_i(1 - p_i)} \right) = -\sum_{i \in \mathcal{F}_j} \left(\frac{y_i - p_i}{p_i(1 - p_i)} \right)$$

- Deuxième terme du produit:

$$\frac{\partial p_i}{\partial \hat{y}_i} = p_i(1 - p_i)$$

- Gradient:

$$g_i = -\sum_{i \in \mathcal{F}_j} \left(\frac{y_i - p_i}{p_i(1 - p_i)} \right) \cdot p_i(1 - p_i) = -\sum_{i \in \mathcal{F}_j} (y_i - p_i)$$

- Hessienne:

$$h_i = \frac{\partial^2 L(\hat{y}_i, y_i)}{\partial \hat{y}_i^2} = \sum_{i \in \mathcal{F}_j} p_i(1 - p_i)$$

Avec ces valeurs, on obtient :

- La valeur optimale des feuilles:

$$w_j = -\frac{\sum_{i \in \mathcal{F}_j} (y_i - p_i)}{\sum_{i \in \mathcal{F}_j} p_i(1 - p_i) + \lambda}$$

- La perte sur une seule feuille:

$$-\frac{1}{2} \cdot \frac{\sum_{i \in \mathcal{F}_j} (y_i - p_i)^2}{\sum_{i \in \mathcal{F}_j} p_i(1 - p_i) + \lambda} + \gamma$$

5.4 Exemple Pratique en Python avec XGBoost

Dans cette section, nous allons appliquer ces concepts en utilisant le jeu de données California Housing avec XGBoost en utilisant la métrique de régression MSE *(Mean Squared Error)*.

```python
import xgboost as xgb
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Charger les donnees
california_housing = fetch_california_housing()
X, y = california_housing.data, california_housing.target

# Diviser les donnees en ensembles d'entrainement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Creer un DMatrix pour XGBoost
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Definir les parametres du modele
params = {
    'objective': 'reg:squarederror',
    'booster': 'gbtree',
    'eta': 0.1,
    'max_depth': 3,
    'lambda': 1,
    'alpha': 0
}

# Entrainer le modele
```

```
num_round = 100
bst = xgb.train(params, dtrain, num_round)

# Faire des predictions
y_pred = bst.predict(dtest)

# Evaluer le modele
mse_xgb = mean_squared_error(y_test, y_pred)

# Evaluer la vitesse de convergence
start_time = time.time()
bst = xgb.train(params, dtrain, num_round)
xgb_time = time.time() - start_time
```

Listing 1: Implémentation des données California Housing avec XGboost

On peut en profiter pour comparer les performances de ce modèle avec ceux générés avec AdaBoost et Gradient Boosting :

```
    # Gradient Boosting
gbr = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3)

start_time = time.time()
gbr.fit(X_train, y_train)
gbr_time = time.time() - start_time

y_pred_gbr = gbr.predict(X_test)
mse_gbr = mean_squared_error(y_test, y_pred_gbr)

    # AdaBoost
abr = AdaBoostRegressor(n_estimators=100, learning_rate=0.1)

start_time = time.time()
abr.fit(X_train, y_train)
abr_time = time.time() - start_time

y_pred_abr = abr.predict(X_test)
mse_abr = mean_squared_error(y_test, y_pred_abr)

print(f"XGB - MSE: {mse_xgb}, Time: {xgb_time} secondes")
print(f"GBM - MSE: {mse_gbr}, Time: {gbr_time} secondes")
print(f"ADB - MSE: {mse_abr}, Time: {abr_time} secondes")

XGB - MSE: 0.29522676196268116, Time: 0.110097169876099 sec
GBM - MSE: 0.29399732486438634, Time: 8.156917095184326 sec
ADB - MSE: 0.5662084586613196,  Time: 5.215624094009399 sec
```

Listing 2: Comparaison des performances du modèle généré avec AdaBoost et Gradient Boosting

En obtient les résultats suivants :

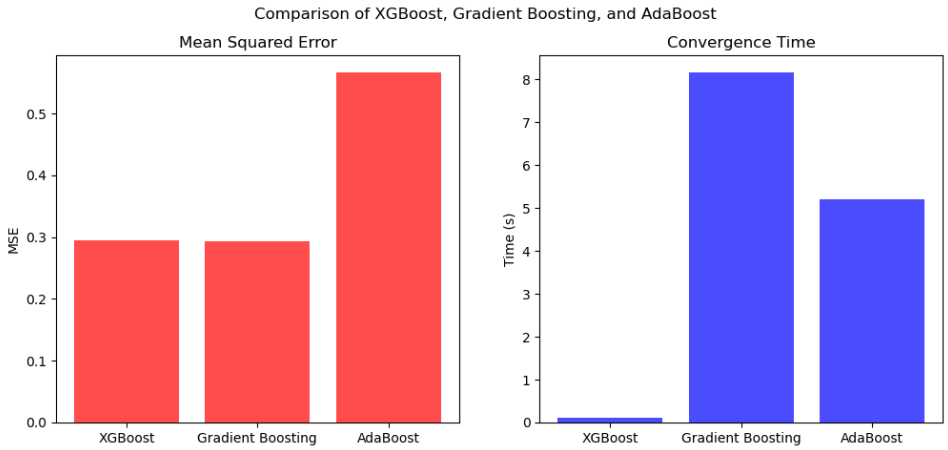


Figure 6: Graphe des MSE obtenues et des vitesses de convergences de chaque algorithme

On remarque que la MSE obtenue est divisée par 2 avec Gradient Boosting et XGBoost par-rapport à AdaBoost, et que la vitesse de convergence de XGBoost est environ 73 fois plus rapide que Gradient Boosting et environ 47 fois plus rapide que AdaBoost

Mais la question suivante demeure : Pourquoi une telle vitesse de convergence ?

La réponse pourraît être celle qui fut formulée pour l'algorithme de Newton et par ce qu'on a de-veloppé jusqu'à présent, mais essayons de développer davantage la question en évoquant quelques algorithmes utilisés par XGBoost pour fournir ces excellents résultats

5.5 Extension : algorithmes de recherche de divisions

5.5.1 Cadre de base

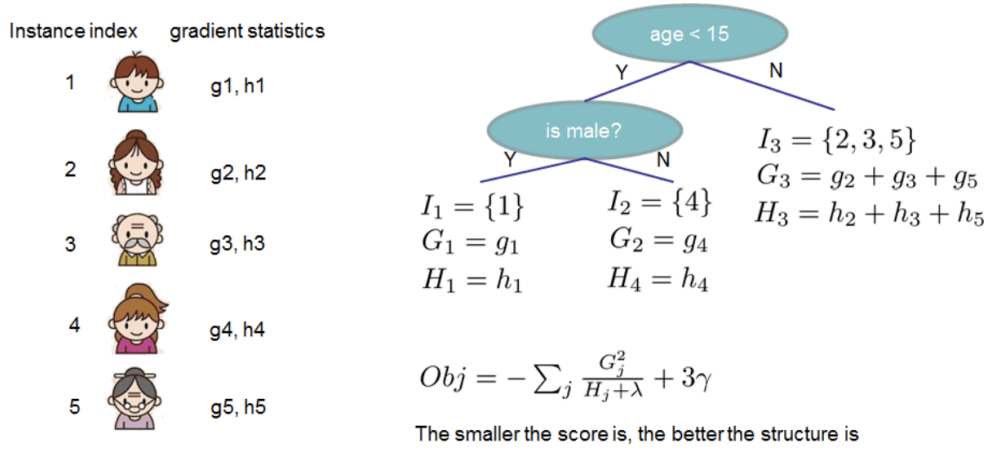


Figure 7: Calcul du Score de Structure. (Chen et Guestrin. 2016)

Redéfinissons le cadre établi jusqu'à présent :

On pose $\mathcal{F}_j = \{i|q(x_i) = j\}$ comme l'ensemble des instances de la feuille j. On redéfinit l'équation de perte muni du terme de régularisation Ω comme suit :

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 = \sum_{j=1}^{T} \left[\left(\sum_{i \in \mathcal{F}_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in \mathcal{F}_j} h_i + \lambda \right) w_j^2 \right] + \gamma T$$

Et le poids optimal w_j^* de la feuille j pour une structure fixe $q(x)$, par

$$w_j^* = -\frac{\sum_{i \in \mathcal{F}_j} g_i}{\sum_{i \in \mathcal{F}_j} h_i + \lambda}$$

Ce qui donne

$$\mathcal{L}^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{\left(\sum_{i \in \mathcal{F}_j} g_i \right)^2}{\sum_{i \in \mathcal{F}_j} h_i + \lambda} + \gamma T.$$

L'équation peut être utilisée comme une fonction de scoring pour mesurer la qualité d'une structure d'arbre q, similaire au score d'impureté pour évaluer les arbres de décision, sauf qu'il est dérivé pour une gamme plus large de fonctions objectives.

Normalement, il est impossible d'énumérer toutes les structures d'arbre possibles q. Un algorithme glouton qui commence par une seule feuille et ajoute itérativement des branches à l'arbre est utilisé à la place. Avec $\mathcal{F}_{jg}$ et $\mathcal{F}_{jd}$ les ensembles d'instances des nœuds gauche et droit après la scission. En posan $\mathcal{F}_j = \mathcal{F}_{jg} \cup \mathcal{F}_{jg}$, alors la réduction de la perte après la scission est donnée par

$$L_{\text{split}} = \frac{1}{2}\left[\frac{\left(\sum_{i\in\mathcal{F}_{jg}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{jg}} h_i + \lambda} + \frac{\left(\sum_{i\in\mathcal{F}_{jd}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{jd}} h_i + \lambda} - \frac{\left(\sum_{i\in\mathcal{F}_{j}} g_i\right)^2}{\sum_{i\in\mathcal{F}_{j}} h_i + \lambda}\right] - \gamma$$

Cette formule est généralement utilisée en pratique pour évaluer les candidats de scission.

On peut modéliser ces processus par deux algorithmes connus, l'algorithme glouton exact et l'algorithme approximatif

5.5.2 Algorithme glouton exact

La première technique est la réduction qui réduit les poids nouvellement ajoutés par un facteur η après chaque étape de boosting tree, elle évite l'influence de chaque arbre individuel et laisse de l'espace pour que les arbres futurs améliorent le modèle.

Algorithm 3 Algorithme glouton exact pour la recherche de scission

Require: I, ensemble d'instances du nœud courant
Require: d, dimension des colonnes
  gain $\leftarrow 0$
  $G \leftarrow \sum_{i\in I} g_i$, $H \leftarrow \sum_{i\in I} h_i$
  for $k = 1$ to m **do**
    $G_L \leftarrow 0$, $H_L \leftarrow 0$
    for j in sorted(I, by x_{jk}) **do**
      $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
      $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
      score $\leftarrow \max(\text{score}, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
    end for
  end for
Ensure: Scission avec score max =0

5.5.3 Algorithme approximatif

La deuxième technique est le sous-échantillonnage des colonnes (caractéristiques). Cette technique est couramment utilisée dans RandomForest, mais non appliquée au Boosting Trees auparavant. L'utilisation des sous-échantillons de colonnes accélère également les calculs de l'algorithme parallèle décrit dans la suite.

Definition : Un percentile est une mesure statistique qui indique la valeur en dessous de laquelle se trouve un certain pourcentage des observations dans un ensemble de données.

En entrée, La proposition peut être faite par arbre (globale) ou par scission (locale).

Algorithm 4 Algorithme approximatif pour la recherche de scission

for $k = 1$ to m **do**
  Proposer $S_k = \{s_{k1}, s_{k2}, \ldots, s_{kl}\}$ par percentiles sur la colonne k.
end for
for $k = 1$ to m **do**
  $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} h_j$
end for
Suivre les mêmes étapes que dans la section précédente pour trouver le score max uniquement parmi les scissions proposées. =0

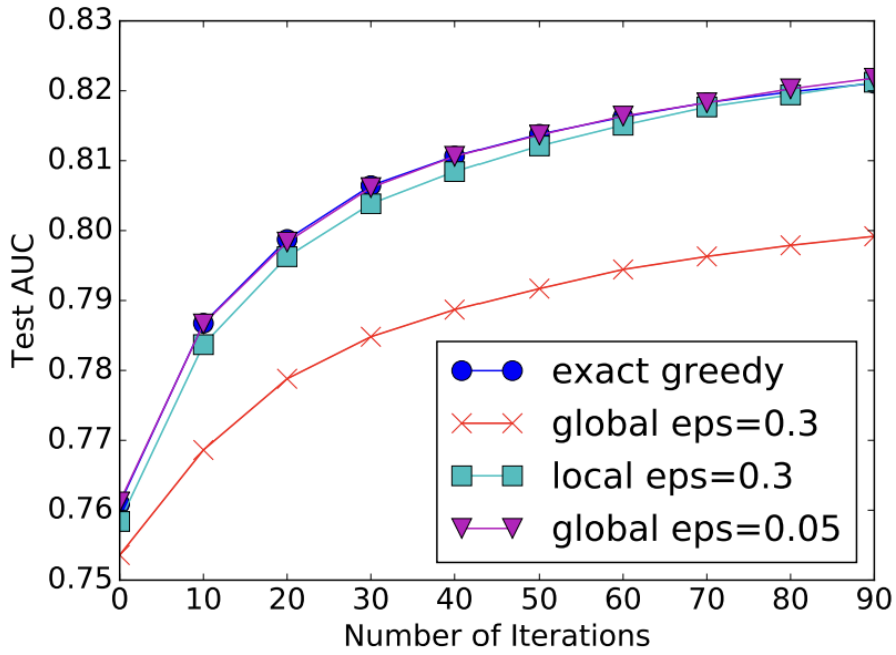


Figure 8: Comparaison de la convergence de l'AUC de test sur le jeu de données Higgs 10M. Le paramètre eps correspond à la précision de l'esquisse approximative. Cela se traduit approximativement par 1/eps compartiments dans la proposition. Nous constatons que les propositions locales nécessitent moins de compartiments, car elles affinent les candidats de division.

L'algorithme approximatif propose d'abord des points de scission candidats basés sur les percentiles de la distribution des caractéristiques. Ensuite, il associe les valeurs continues des caractéristiques dans des compartiments définis par ces points candidats. Pour chaque compartiment, l'algorithme applique les gradients et hessiennes des fonctions de perte. Enfin, parmi les splits candidats, l'algorithme sélectionne celui qui maximise la réduction de la perte, optimisant ainsi la division des données pour améliorer la performance du modèle.

Il comporte deux variantes selon le moment où les propositions de splits sont faites.
La variante globale propose tous les splits candidats dès la phase initiale de construction de l'arbre et réutilise ces propositions à chaque niveau, nécessitant ainsi moins d'étapes de proposition mais plus de candidats initiaux pour compenser le manque d'affinement après chaque division.
En revanche, la variante locale repropose des splits après chaque division, affinant les candidats et étant potentiellement plus adaptée pour des arbres plus profonds. Comparativement, la proposition locale requiert généralement moins de candidats que la globale, bien qu'elle puisse atteindre une précision similaire avec un nombre suffisant de candidats.

5.5.4 Quantiles Pondérés

Une étape importante de l'algorithme approximatif est de proposer des points de scissions candidats. Habituellement, les percentiles d'une caractéristique sont utilisés pour que les candidats soient répartis uniformément sur les données.

Soit l'ensemble $D_k = \{(x_{1k}, h_1), (x_{2k}, h_2), \cdots, (x_{nk}, h_n)\}$ représentant les valeurs des caractéristiques de la k-ième caractéristique et les hessiennes de chaque instance d'apprentissage. On peut définir une fonction de rang $r_k : \mathbb{R} \to [0, +\infty[$ comme

$$r_k(z) = \frac{1}{\sum_{(x,h)\in D_k} h} \sum_{(x,h)\in D_k, x<z} h,$$

qui représente la proportion d'instances dont la valeur de la caractéristique k est inférieure à z. L'objectif est de trouver des splits candidats $\{s_{k1}, s_{k2}, \cdots s_{kl}\}$, tels que

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon, \quad s_{k1} = \min_i x_{ik}, \quad s_{kl} = \max_i x_{ik}.$$

Ici, ϵ est un facteur d'approximation. Intuitivement, cela signifie qu'il y a environ $1/\epsilon$ points candidats. Ici, chaque donnée est pondérée par h_i. h_i représente le poids, en effet on peut réécrire la fonction de perte comme :

$$\sum_{i=1}^{n} \frac{1}{2} h_i (f_t(x_i) - g_i/h_i)^2 + \Omega(f_t) + \text{constant},$$

qui est exactement la perte quadratique pondérée avec des étiquettes g_i/h_i et des poids h_i. Pour les grandes bases de données, il n'est pas trivial de trouver des divisions candidates qui satisfont les critères. Lorsque chaque observation a un poids égal, un algorithme existant appelé esquisse quantile résout le problème.

Intérêt pour XGBoost

L'algorithme des quantiles pondérés est essentiel pour certaines variantes avancées de XGBoost. Voici les principaux intérêts de cet algorithme :

1. Gestion des Données Pondérées

Les observations ne sont pas toutes d'importance égale. L'algorithme des quantiles pondérés permet de prendre en compte ces poids lors du calcul des quantiles, assurant que les données les plus importantes influencent davantage les résultats. De plus, c'est utile pour les ensembles de données déséquilibrés, où certaines classes ou types de données doivent être correctement représentés ou sous-représentés.

2. Calcul Efficace des Seuils de Division

Lors de la construction des arbres de décision, XGBoost utilise les quantiles pondérés pour déterminer les meilleurs seuils de division pour chaque caractéristique, améliorant ainsi la performance de l'algorithme.

3. Approximation des Quantiles pour les Données de Grand Volume

L'algorithme des quantiles pondérés permet d'approximer les quantiles avec une précision contrôlée sans nécessiter de stocker et trier toutes les données, économisant ainsi de la mémoire et du temps de calcul.

4. Maintien de la Précision dans un Contexte Distribué

Dans les implémentations distribuées de XGBoost, l'algorithme des quantiles pondérés permet de fusionner les résumés de quantiles provenant de différentes machines de manière efficace, assurant une estimation précise des quantiles après la fusion des résumés de différentes partitions de données.

5.5.5 Recherche de Scission avec Sparsité des données

Dans de nombreux problèmes du monde réel, il est assez courant que l'entrée x soit rare. Il existe plusieurs causes possibles de la rareté :

- 1) la présence de valeurs manquantes ("NA") dans les données ;
- 2) Des entrées nulles ;
- 3) Des techniques d'apprentissage automatique comme le One Hot Encoding.

Il est important de rendre l'algorithme conscient de la sparsité dans les données. Pour ce faire, nous proposons d'ajouter une direction par défaut dans chaque nœud de l'arbre (Figure ci-dessous). Lorsqu'une valeur est manquante dans la matrice sparse x, l'instance est classée dans la direction par défaut.

Il y a deux choix de direction par défaut dans chaque branche. Les directions par défaut optimales sont apprises à partir des données. L'amélioration clé est de ne visiter que les entrées

non manquantes I_k. L'algorithme ci-dessous l'explique en détail:

Algorithm 5 Recherche de Partition Sensible à la Parcimonie

Require: I {Ensemble d'instances du nœud courant}
Require: $I_k = \{i \in I \mid x_{ik} \neq \text{manquant}\}$
Require: d {Dimension des caractéristiques}
Ensure: Partition et directions par défaut avec gain maximal
0: $gain \leftarrow 0$
0: $G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
0: **for** $k \leftarrow 1$ **to** m **do**
0: $GL \leftarrow 0$, $HL \leftarrow 0$
0: **for** j **in** I_k **trié en ordre croissant par** x_{jk} **do**
0: $GL \leftarrow GL + g_j$, $HL \leftarrow HL + h_j$
0: $GR \leftarrow G - GL$, $HR \leftarrow H - HL$
0: $score \leftarrow \max\left(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right)$
0: **end for**
0: $GR \leftarrow 0$, $HR \leftarrow 0$
0: **for** j **in** I_k **trié en ordre décroissant par** x_{jk} **do**
0: $GR \leftarrow GR + g_j$, $HR \leftarrow HR + h_j$
0: $GL \leftarrow G - GR$, $HL \leftarrow H - HR$
0: $score \leftarrow \max\left(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right)$
0: **end for**
0: **end for**=0

Cet algorithme vise à optimiser la partition des données dans les arbres de décision en tenant compte des valeurs manquantes, ce qui est crucial pour XGBoost. Il commence par initialiser le gain maximal à zéro, puis calcule les sommes des gradients (G) et des hessiennes (H) pour toutes les observations du nœud. Ensuite, pour chaque paramètre (caractcéristique), il traite séparément les valeurs manquantes allant à droite et à gauche. Pour chaque direction, il trie celles non manquantes, puis, pour chaque instance triée, met à jour les sommes des gradients et des hessiennes des feuilles gauche et droite.

Le score est calculé à chaque étape en maximisant l'expression impliquant les gradients, les hessians et un terme de régularisation lambda, afin d'éviter le sur-ajustement. En parcourant les valeurs manquantes dans les deux directions, l'algorithme trouve la partition avec le gain maximal et apprend la meilleure manière de gérer les valeurs manquantes.

Contrairement à la plupart des algorithmes d'apprentissage d'arbres existants, qui sont optimisés uniquement pour les données denses ou nécessitent des procédures spécifiques pour les cas limités comme l'encodage catégoriel, XGBoost gère tous les modèles de parcimonie de manière unifiée et elle l'exploite pour rendre la complexité de calcul linéaire par rapport au nombre d'entrées non manquantes (exemple sur la figure 5). XGBoost utilise cette approche pour construire ses arbres de manière efficace et robuste, améliorant ainsi la performance et la précision des prédictions, même en présence de données incomplètes.

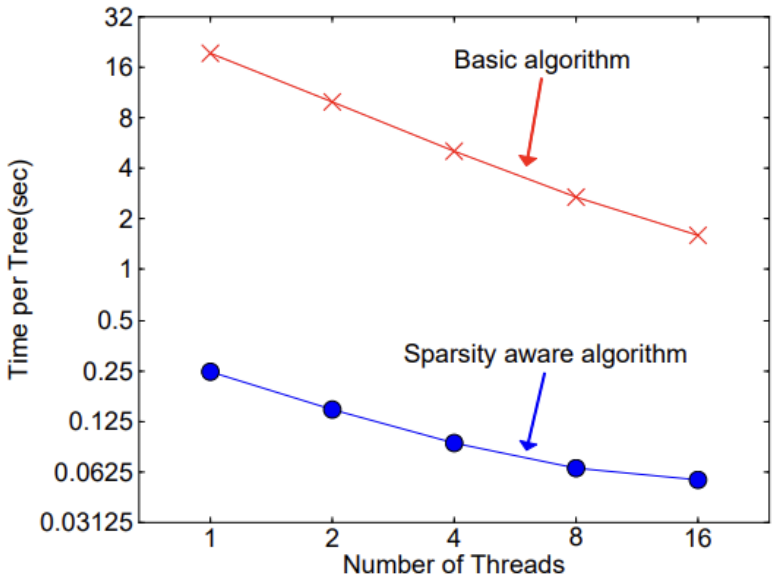


Figure 9: Résultats sur le jeu de données Allstate-50K (Chen et Guestrin, 2016)

5.5.6 Complexité des algorithmes

Soit d la profondeur maximale de l'arbre et K le nombre total d'arbres. Pour l'algorithme glouton exact, la complexité temporelle de l'algorithme original sensible à la sparité est

$$O(Kd\|x\|_0 \log n)$$

Ici, nous utilisons $\|x\|_0$ pour désigner le nombre d'entrées non manquantes dans les données d'entraînement. En revanche, le boosting tree sur la structure en blocs ne coûte que

$$O(Kd\|x\|_0 + \|x\|_0 \log n)$$

Ici, $O(\|x\|_0 \log n)$ représente le coût de prétraitement unique qui peut être amorti. Cette analyse montre que la structure en blocs permet d'économiser un facteur $\log n$ supplémentaire, ce qui est significatif lorsque n est grand.

Pour l'algorithme approximatif, la complexité temporelle de l'algorithme original avec recherche binaire est

$$O(Kd\|x\|_0 \log q)$$

En conclusion, XGBoost offre une complexité temporelle améliorée par rapport au Gradient Boosting traditionnel, qui a une complexité de $O(Kdn \log n)$. Ces améliorations permettent à XGBoost

de traiter les grands ensembles de données plus rapidement et efficacement que le Gradient Boosting traditionnel, faisant de XGBoost un choix supérieur en termes de performance et de rapidité d'exécution.

6 Conclusion générale

Les études de cas présentées ont démontré l'impact significatif du boosting dans divers secteurs, allant de la reconnaissance d'images à la finance, en passant par la détection de fraudes et la prévision de la demande. Ces exemples pratiques montrent que le boosting n'est pas seulement une technique théoriquement intéressante, mais aussi un outil indispensable pour résoudre des problèmes complexes et améliorer les performances des modèles prédictifs dans le monde réel.

Cependant, il est également crucial de reconnaître les défis associés à l'utilisation du boosting. Les questions de complexité computationnelle, de gestion des données déséquilibrées et d'interprétabilité des modèles restent des préoccupations majeures. À cet égard, des efforts continus sont nécessaires pour développer des algorithmes plus efficaces et accessibles, ainsi que pour améliorer la transparence et la compréhension des décisions prises par les modèles de boosting.

En conclusion, ce mémoire a mis en évidence non seulement la puissance et la polyvalence du boosting, mais aussi les considérations pratiques et les défis qui accompagnent son utilisation. En synthétisant les connaissances actuelles et en proposant des pistes de recherche futures, nous espérons avoir apporté une contribution significative à la compréhension et à l'application de cette technique essentielle en apprentissage automatique. Par cette exploration, nous espérons encourager une utilisation plus judicieuse et éclairée du boosting dans diverses applications, permettant ainsi de tirer pleinement parti de ses capacités pour résoudre des problèmes complexes et variés.

7 Références

- Schapire & Freund, Boosting, 1999
- Cours introduction aux methodes d'agregation, L. Rouviere, Université Rennes 2
- Cours L3 MIASHS Optimisation, Adina Ciomaga, Université Paris Cité
- XGBoost, A Scalable Tree Boosting System, Chen and Guestrin, University of Washington
- XGBoost, Notes sur cet algorithme indispensable, www.objectifdatascience.com
- Playlist XGBoost : StatQuest with Josh Starmer, YouTube
- https://xgboost.readthedocs.io/en/latest/tutorials/model.html
- ChatGPT