Université d'Angers

Master 2 Data Science
Apprentissage Statistique
Projet Annuel

Gradient Boosting Machines

Élèves :
Adrien Maïtammar

Tuteurs :
Frédéric Proïa (Université d'Angers)
Bilal Idiri (Boursorama)

17 mai 2022

Table des matières

Introduction

Notre époque est sujet à une transformation numérique et tout processus qui peut être digitalisé tend à l'être. La quantité de données générées explose. On estime qu'entre 2010 et 2020, le volume de données créés ou répliquées a été multiplié par 32 en passant de 2 à 64 zettaoctets [1]. Ces données une fois collectées, stockées et traitées forment la matière première des modèles d'intelligence artificielle.

D'autre part, dans le domaine informatique, la puissance de calcul s'est décuplée. Ces innovations ont donc permis de traiter de plus en plus de données avec des algorithmes de plus en plus complexes. Les éléments étaient réuni pour permettre l'essor de l'apprentissage automatique.

L'apprentissage automatique est une sous-catégorie de l'intelligence artificielle. Elle consiste à laisser des algorithmes découvrir les motifs récurrents dans les ensembles de données afin de réaliser une tâche. Ces données peuvent être des signaux, des images, des documents et le plus souvent des données tabulaires. En décelant les motifs dans ces données, les algorithmes apprennent et améliorent leurs performances dans l'exécution d'une tâche spécifique. La régression linéaire, les k plus proches voisins, les machine à vecteurs de support, les arbres de décision ou encore les réseaux de neurones sont les algorithmes les plus connus. En particulier, une catégorie d'algorithmes s'est imposée dans le traitement de données tabulaires pour son efficacité et sa facilité d'utilisation. Ceux sont les algorithmes de Gradient Boosting. Ils consistent à entraîner en série des modèles, souvent des arbres de décision, qui se corrigent successivement.

Ainsi, dans ce rapport, nous allons revoir des notions fondamentales de l'apprentissage automatique (chapitre 1) avant de présenter le concept de gradient boosting (chapitre 2). Ces sections présentent les éléments de base sur lesquels reposent le fonctionnement des algorithmes de boosting tels que XGBoost et LightGBM (chapitre 3). Nous verrons ensuite quels sont les hyperparamètres principaux et comment les ajuster chapitre 4. La dernière section est dédiée à la notion d'explicabilité (chapitre 5).

1. Un zettaoctet équivaut à mille milliards de gigaoctets

Chapitre 1

Bases de l'apprentissage supervisé

1.1 Définition

L'apprentissage statistique [1] se défini comme « le domaine d'étude qui donne aux ordinateurs la capacité d'apprendre sans être explicitement programmés » [2]. Comme le montre la figure 1.1, on peut distinguer trois grandes classes d'apprentissage : supervisé, non-supervisé et par renforcement. Nous nous intéresserons en particulier à l'apprentissage supervisé. Il s'agit alors de collecter un jeu de données $\mathcal{D}_n = \{(x_1, y_1), ..., (x_n, y_n)\}$ considéré comme n réalisations indépendantes du couple de variables aléatoires $(X, Y) \in \mathcal{X} \times \mathcal{Y}$. Ainsi chaque x_i correspond à une observation parfois appelée instance. Cette observation est décrite par p caractéristiques. Par exemple, si ces caractéristiques sont toutes de type numériques alors $\mathcal{X} = \mathbb{R}^p$.

Dans le cadre de la régression, on prédit une variable numérique, c'est à dire $\mathcal{Y} = \mathbb{R}$. On cherche une fonction $f(X)$ afin de prédire $Y = f(X) + \epsilon$ où ϵ est une variable aléatoire, indépendante de X, centrée et de variance finie. La solution optimale est appelée fonction de régression telle que :

$$\begin{aligned} f : \mathcal{X} &\to \mathcal{Y} \\ x &\mapsto \mathbb{E}[Y|X = x] \end{aligned}$$

Dans le cadre de la classification, on prédit une variable catégorielle, c'est à dire $\mathcal{Y} = \{1, \cdots, K\}$. La meilleure règle de classification possible est appelée classifieur de Bayes, telle que f soit définit comme suit :

$$\begin{aligned} f : \mathcal{X} &\to \mathcal{Y} \\ x &\mapsto \underset{k \in \mathcal{Y}}{\operatorname{argmax}}\, P(Y = k|X = x) \end{aligned}$$

La fonction f est inconnue, elle fait partie d'un cadre mathématique permettant d'obtenir des garanties théoriques. En pratique, l'objectif est de construire une fonction approximant f. Pour ce faire, le Data Scientist dispose de différentes méthodes. On peut citer par exemple la régression linéaire, les machines à vecteurs supports, la régression logistique, les arbres de décision ou encore les réseaux de neurones.

1. Aussi appelé "apprentissage automatique" ou encore "machine learning" en anglais.
2. Citation d'Arthur Samuel en 1959.

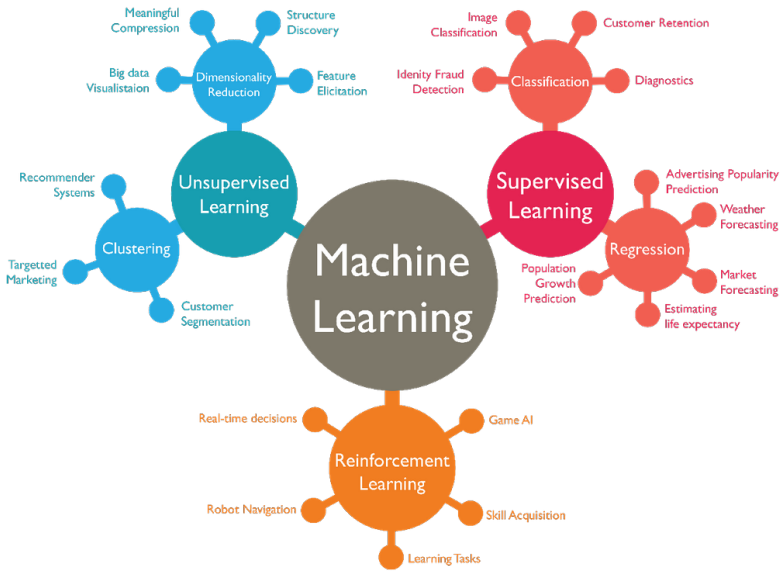

FIGURE 1.1 – Vue globale des applications en apprentissage automatique. Source : KDnuggets.

Chaque méthode utilise un algorithme qui prend en entrée les observations afin de construire une fonction reliant X à y où dorénavant X est une matrice de taille $n \times p$ désignant n observations ayant p caractéristiques et y est un vecteur de taille $n \times 1$ représentant les réponses associées aux observations. On note $\mathcal{F}$ l'espace des fonctions que peut produire l'algorithme. Les prédictions sont notées $\hat{y} = \hat{f}(X)$. Le problème est alors de trouver une fonction déterministe $\hat{f}$ qui minimise la fonction coût :

$$\mathcal{L}(y, \hat{f}(X)) = \mathcal{L}(y, \hat{y}) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) \tag{1.1}$$

où l est une fonction de perte telle que pour tout $\hat{y}_i \neq y_i$ on ait $l(y_i, \hat{y}_i) > 0$.

Par ailleurs, à performance similaire, il est toujours préférable d'opter pour un modèle plus simple. C'est le principe de parcimonie. Pour ce faire il est possible d'ajouter à la fonction coût un terme de régularisation $\mathcal{P}(\theta)$ augmentant avec la complexité du modèle. On souhaite alors minimiser la fonction objectif suivante :

$$obj(y, \hat{f}(X)) = \mathcal{L}(y, \hat{f}(X)) + \mathcal{P}(\hat{f}) \tag{1.2}$$

La fonction objectif est un moyen mathématique précis, rigoureux et souple permettant d'évaluer la qualité d'apprentissage d'un modèle.

Pour résumer, un modèle de machine learning supervisé repose sur un algorithme prenant en entrée des données. Les données sont constituées d'observations auxquelles est associé un vecteur de réponse. L'algorithme construit alors une fonction reliant les observations au vecteur de réponse. Ce dernier doit apprendre cette relation. Autrement dit, la complexité de l'algorithme doit être en adéquation avec la complexité du lien entre X et Y. La section suivante présente à ce sujet une notion fondamentale de l'apprentissage automatique, le compromis biais-variance.

1.2 Compromis biais-variance

En pratique, il ne faut pas attendre d'un modèle de machine learning de donner des prédictions justes dans 100% des cas. En effet, les données peuvent contenir plus ou moins de bruit. Ceci étant dit, il est possible d'orienter notre prise de décision en fonction du type d'erreur commis par le modèle. Il existe trois types d'erreur. Pour les mettre en évidence, plaçons nous dans le cas d'une régression tel que présenté en section 1.1. On choisit l'erreur quadratique en tant que fonction coût :

$$\mathcal{L}(y, y') = (y - y')^2$$

Calculons dans ce cas l'espérance de la fonction coût :

$$\begin{aligned}
\mathbb{E}[\mathcal{L}(Y, \hat{f}(X))] &= \mathbb{E}[(Y - \hat{f}(X))^2] \\
&= \mathbb{E}[((f - \hat{f})(X) + \epsilon)^2] \\
&= \mathbb{E}[\epsilon^2] + \mathbb{E}[((f - \hat{f})(X))^2] + 0 \\
&= \mathbb{V}[\epsilon^2] + \mathbb{V}[(f - \hat{f})(X)] + \mathbb{E}[(f - \hat{f})(X)]^2 \\
&= \sigma^2 + variance + biais^2
\end{aligned}$$

On obtient une décomposition avec trois termes. Le premier est la variance du bruit, c'est un paramètre intrinsèque aux données et hors de notre contrôle. Le deuxième et le troisième terme sont sous notre contrôle. Ils représentent respectivement la variance de l'erreur obtenue par le modèle et la moyenne au carré (*ie.* le biais au carré) de l'erreur obtenue par le modèle.

Le biais est la différence entre les prédictions du modèle $\hat{f}(X)$ et les vraies valeurs y. Cet écart est dû au fait que le modèle n'est pas assez complexe. Par exemple, lorsque qu'on utilise un modèle de régression linéaire, on fait l'hypothèse que la variable cible Y suit une loi normale et qu'elle correspond globalement à une combinaison linéaire des variables explicatives $X^1, \cdots, X^p$. Ainsi, l'ensemble $\mathcal{F}$ des fonctions que peut produire notre modèle est relativement restreint. Si la relation entre X et y n'est pas linéaire aucune fonction ne peut s'ajuster correctement sur les données.

La variance du modèle augmente par définition lorsque les erreurs sont fluctuantes entre différents jeux de données. En particulier, un modèle avec une forte variance est symptomatique d'une incapacité à généraliser. Autrement dit, bien que le modèle obtienne de bonnes performances sur un jeu d'entraînement, il est possible que ses performances décroissent fortement sur de nouvelles données. Cela peut être dû au fait que le modèle s'ajuste trop bien aux données d'apprentissage au point de prendre en compte le bruit de la données.

L'objectif de tout modèle d'apprentissage automatique supervisé est d'obtenir un faible biais et une faible variance. Cependant, pour un jeu de données et une fonction de prédiction données, on remarque que la variance augmente si le biais baisse et inversement. C'est pourquoi, on parle de compromis biais-variance. Il est donc nécessaire de choisir un modèle avec une flexibilité en adéquation avec la complexité du problème.

1.3 Apprentissage machine

Les notions de biais et de variance permettent d'évaluer la qualité d'apprentissage d'un modèle de machine learning. Pour ce faire, il est nécessaire de diviser au minimum le jeu de donnée en deux parties. La première partie est utilisée pour entraîner le modèle, c'est le *train set*. La seconde partie des données, appelée *test set*. Elle est utilisée pour effectuer la phase d'inférence. L'objectif est d'évaluer le modèle sur des données nouvelles. Ainsi, en choisissant une mesure de performance adaptée au problème, on calcule l'erreur sur le jeu d'entraînement et sur le jeu de test. Suite à ces opérations, de manière schématique, il est possible de se trouver dans trois configurations : le sous-apprentissage, le sur-apprentissage et un apprentissage correct.

Premièrement, une erreur d'entraînement élevée et une erreur de test proche le celle d'entraînement est le signe d'un fort biais et d'une forte variance. Le modèle n'arrive pas à apprendre les motifs sur les données d'entraînement et pas conséquent il atteint également une faible performance sur les observations du jeu de test. Cette configuration est appelée sous-apprentissage.

Pour remédier à cette situation, on doit rendre le modèle plus complexe. Cela peut passer par l'ajout de nouvelles caractéristiques sur les observations, par un temps d'entraînement plus long. Il est aussi possible que l'algorithme choisit ne soit pas adapté au problème.

Deuxièmement, une erreur d'entraînement faible avec une erreur de test beaucoup plus élevée est signe d'une variance élevée. Cette différence de performance est due au fait que le modèle apprenne trop bien les données d'entraînement mais sans pouvoir généraliser. Cette configuration est appelée sur-apprentissage.

Pour remédier à cette situation, on doit rendre le modèle moins complexe. Cela peut passer par l'utilisation de technique de régularisation ou bien ou récoltant davantage de données.

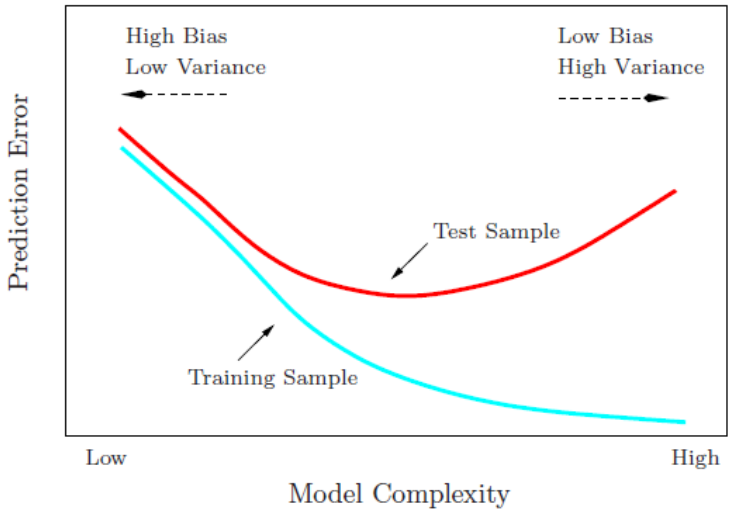

FIGURE 1.2 – Erreur d'entrainement et de test en fonction de la complexité du modèle. Source : [1]

Enfin, entre ces deux situations se trouve un juste milieu où l'erreur d'entraînement est convenable et l'erreur de test est légèrement supérieur. Dans ce cas, le modèle atteint les performances souhaitées et il parvient à les conserver sur des données qu'il n'a jamais vu.

La figure 1.3 illustre ces trois configurations pour différents modèles. La ligne du haut et du milieu représentent respectivement une tâche de régression et de classification. La ligne du bas représente l'évolution de l'erreur d'entraînement et de test pour des modèles apprenant par itérations. C'est le cas des modèles de boosting et des réseaux de neurones.

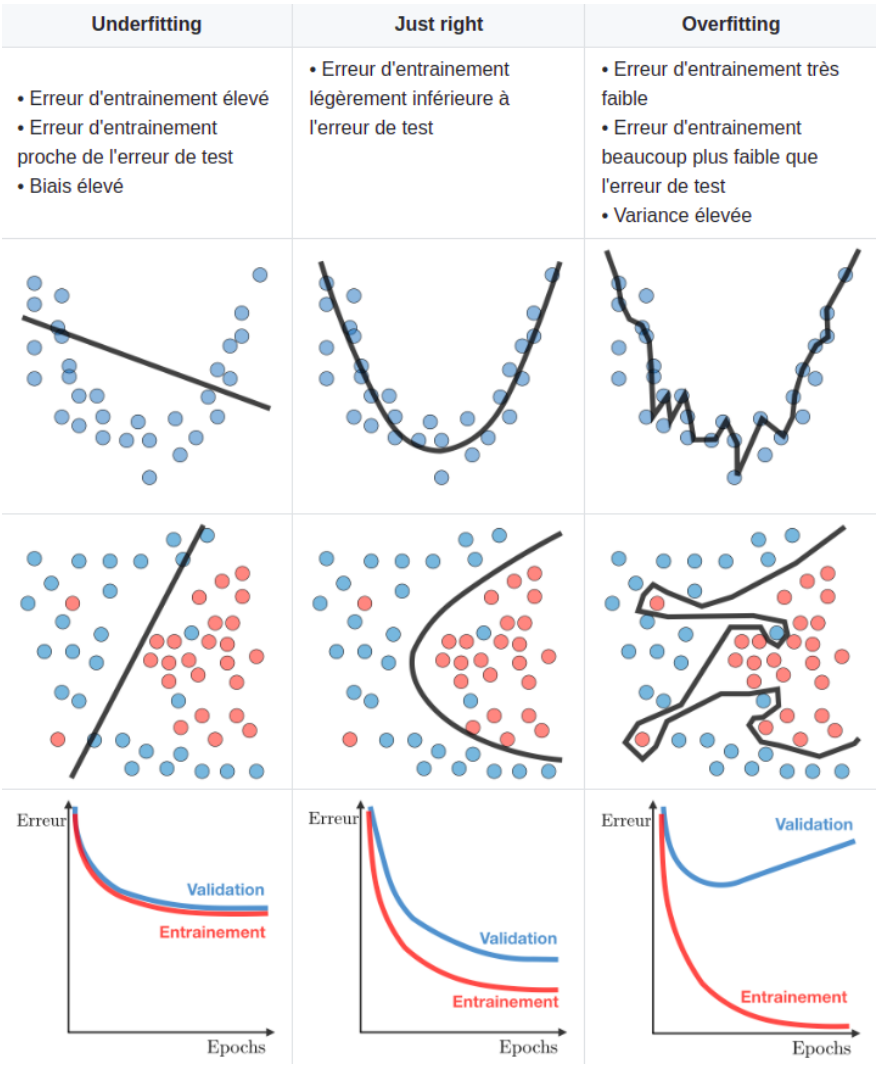

FIGURE 1.3 – Sous-apprentissage et sur-apprentissage. Source : Shervine Amidi

Chapitre 2

Gradient Boosting

Les algorithmes de Gradient Boosting sont incontournables dans le monde de l'apprentissage machine. Ils sont polyvalents et efficaces sur les tâches de régression, de classification et de classement sur les données structurées. De plus, leur interprétabilité et leur capacité à s'ajuster sur des données volumineuses font de ce type d'algorithme un choix de premier plan pour la mise en production de modèle ML.

Le Gradient Boosting est une méthode d'ensemble. Les méthodes d'ensemble combinent les prédictions de plusieurs modèles faibles afin de construire un modèle fort. On distingue généralement deux familles de méthodes d'ensemble.

D'un côté, il y a les méthodes de bagging dont le principe est de construire plusieurs estimateurs indépendamment, puis de faire la moyenne de leurs prédictions. L'estimateur final tend a être plus robuste que les estimateurs de base car sa variance est réduite. L'algorithme de bagging le plus connu est le Random Forest.

D'autre part, il y a les méthodes de boosting qui combinent séquentiellement des modèles faibles en les entraînant sur une version modifiée des données d'entraînement. Les algorithmes les plus connus sont AdaBoost, GBDT, XGBoost, LightGBM et CatBoost.

Les présentations générales étant faites, nous présenterons dans les sections suivantes des éléments théoriques sur le Gradient Boosting ainsi que son lien avec la descente de gradients. Puis, nous mettrons en évidence les spécificités liées aux tâches de régression et classification. Enfin, nous présenterons les avantages en faveur des arbres de décision en tant que sous-modèle.

2.1 Théorie

Le boosting est une stratégie qui consiste à combiner plusieurs modèles simples en un seul modèle composite. L'idée est que, à mesure que des modèles simples sont introduits, le modèle global devient de plus en plus fort. Dans la terminologie du boosting, les modèles simples sont appelés modèles faibles ou apprenants faibles. La prédiction du modèle final $\hat{y}$ est obtenue par addition des M apprenants faible Δ_m :

$$\hat{y} = F_M(x) = \sum_{m=1}^{M} \Delta_m(x)$$

On peut aussi formuler cela de manière récursive tel que :

$$\begin{aligned} F_0(x) &= f_0(x) \\ F_m(x) &= F_{m-1}(x) + \Delta_m(x) \end{aligned}$$

Si on utilise une métaphore, le gradient boosting peut être vue comme une partie de golf. Placer la balle dans le trou est équivalent à effectuer une prédiction correcte. Le premier modèle représente le premier coup, frappé fort et sans trop de précision afin de se rapprocher de l'objectif y. Les modèles suivants sont plus précis et permettent petit à petit d'atteindre la cible. Ils tentent de réduire l'écart entre les prédictions du modèle composite F_m et la variable cible y.

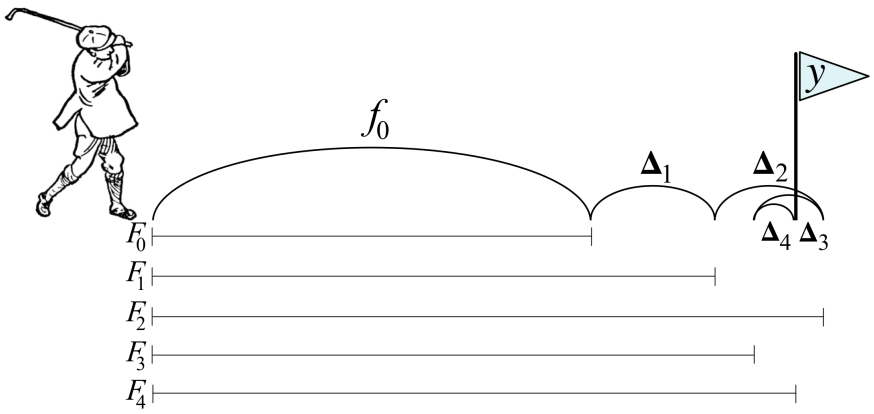

FIGURE 2.1 – Métaphore du golfeur. Source : explained.ai.

Dans le but de réduire l'écart entre les prédictions $\hat{y}_m = F_m(x)$ et la variable cible y, il faut pouvoir mesurer la qualité de la prédiction. Or, dans le chapitre 1 nous avons introduit les notions de fonction objectif et de fonction de perte qui remplissent ce rôle :

$$\mathcal{L}(y, \hat{y}) = \sum_{i=1}^{N} l(y_i, \hat{y}_i)$$

Si la fonction de perte l est convexe alors il est possible de minimiser cette fonction objectif en effectuant une descente de gradient :

$$\hat{y}^{(m)} = \hat{y}^{(m-1)} + \eta(-\nabla\mathcal{L}(y, \hat{y}^{(m-1)}))$$

Cependant, ce qui importe c'est de construire une suite de modèles permettant de prédire y. C'est pourquoi, on ajuste l'apprenant faible Δ_m sur l'opposé du gradient de la fonction objectif, c'est à dire :

$$\Delta_m \approx -\nabla\mathcal{L}(y, \hat{y}^{(m-1)}) = -\nabla\mathcal{L}(y, F_{(m-1)}(x))$$

et donc :

$$F_m(X) = F_{(m-1)}(X) + \eta\Delta_m(X)$$

Nous allons à présent préciser le lien en entre la descente de gradient et le gradient boosting. Car bien que l'objectif de ces deux méthodes soit de trouver le minimum d'une fonction convexe, la finalité n'est pas la même. En effet, dans le cas de la descente de gradient on veut trouver le point de minimum x^* pour la fonction f. Pour ce faire, on va se déplacer dans l'espace où vit x dans la direction opposé au gradient avec un pas η :

$$x^m = x^{m-1} - \eta\nabla f(x^{m-1})$$

Si η est bien choisit alors $\lim_{m\to+\infty} f(x^m) = \min_x f(x) = f(x^*)$.

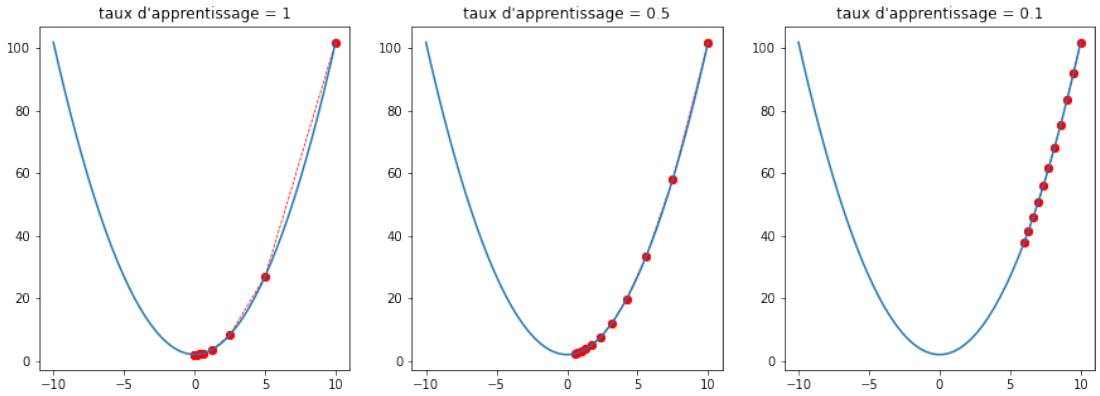

FIGURE 2.2 – Descente de gradient

Par exemple, les réseaux de neurones utilisent la descente de gradient dans la phase de rétro-propagation pour optimiser les poids de chaque neurone, c'est à dire les paramètres du modèle.

Pour ce qui est du gradient boosting, on ajoute une approximation de l'opposé du gradient par un modèle faible à la sortie actuelle, $\hat{y}^{(m-1)}$. Ainsi, on ne modifie pas les paramètres d'un modèle mais on ajoute un petit modèle au modèle composite afin de tendre vers le minimum de la fonction coût $\mathcal{L}$.

Pour obtenir un modèle performant sur une tâche complexe, on peut d'une part, associer un grand nombre d'apprenants faibles afin de capter une grande quantité de motifs tout évitant d'apprendre le bruit de la donnée. D'autre part, réduire η, appelé taux d'apprentissage, permet de laisser la place à davantage de sous-modèles et à plus de complexité. La convergence vers le minimum de la fonction coût est néanmoins ralentie.

Dans les sections suivantes nous allons présenter les algorithmes de gradient boosting appliqués aux tâches de régression et de classification.

2.2 Régression

Avec les notations utilisées précédemment, l'algorithme de gradient boosting tree pour une tâche de régression est le suivant :

Algorithme 1 : Gradient Boosting Regressor

Données : $(X, y) = \{(x_i, y_i)\}_{i=1}^{n}$

Étape 1 : Initialiser

$$F_0(X) = f_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} l(y_i, \gamma)$$

Étape 2 : Pour $m = 1$ à M :

a) Calculer

$$r_{im} = -\frac{\partial l(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)} \quad \text{pour } i = 1, \ldots, n$$

b) Entraîner un apprenant faible Δ_m sur $r_{im},\ i = 1, 2, \ldots, N$

c) Mettre à jour $F_m(X) = F_{m-1}(x) + \Delta_m(X)$

Étape 3 : Retourner $\hat{y} = F_M(X)$

On initialise l'algorithme en construisant un premier sous modèle très simple en cherchant la constante qui minimise la fonction coût.

Suite à cette première étape, on calcule les pseudo-résidus r_{i1} égaux au gradient négatif comme expliqué précédemment. On appelle cette quantité ainsi car si on utilise la fonction de perte erreur quadratique, on obtient :

$$l(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$$

et

$$-\frac{\partial l(y_i, f(x_i))}{\partial f(x_i)} = f(x_i) - y_i$$

Puis, un apprenant faible est entraîné sur ces résidus. Cela permet d'effectuer une prédiction pour chaque élément de notre jeu de données.

Enfin, on additionne les prédictions de ce modèle aux précédentes. On réitère M fois les étapes 2.a à 2.c avant de retourner les prédictions du modèle final $F_M(x)$.

Le choix de la fonction de perte a un très fort impact dans la construction de notre modèle car c'est ce qui détermine les quantités que les sous-modèles apprennent. L'erreur quadratique, la Pseudo-Huber loss et la fonction logcosh sont les fonctions de perte les plus communes.

2.3 Classification

La version du gradient boosting pour une tâche de classification est similaire à l'algorithme pour la régression. On adapte en choisissant une fonction de perte adéquate, l'entropie croisée :

$$l(y_i, f(x_i)) = -\sum_{k=1}^{K} y_{ik} log(p_k(x_i))$$

où y_{ik} est l'indicatrice égale à 1 si l'observation x_i appartient à la classe k, autrement dit :

$$y_{ik} = \mathbb{1}_{y_i=k}$$

et $p_k(x_i)$ est la probabilité conditionnelle d'appartenance à la classe k calculée grâce à la fonction softmax :

$$p_k(x_i) = \frac{e^{f^k(x_i)}}{\sum_{l=1}^{K} e^{f^l(x_i)}}$$

tel que $f^k(x_i)$ soit la prédiction du modèle de la classe k pour l'observation x_i. Ainsi, la construction, l'ajustement, le calcul des prédictions des arbres et la mise à jour du modèle se fait pour chaque classe soit K fois à chaque étape.

Algorithme 2 : Gradient Boosting Classifier

Données : $(X, y) = \{(x_i, y_i)\}_{i=1}^{n}$

Étape 1 : Initialiser $F_{k0}(X) = f_{k0}(X) = 0,\ k = 1, \ldots, K$

Étape 2 : Pour $m = 1$ à M :
Pour $k = 1$ à K :

a) Calculer

$$p_{km}(x_i) = \frac{e^{F_{km}(x_i)}}{\sum_{l=1}^{K} e^{F_{lm}(x_i)}},\ k = 1, 2, \ldots, K$$

b) Calculer $r_{ikm} = y_{ik} - p_{km}(x_i),\ i = 1, 2, \ldots, N$

c) Entraîner un apprenant faible Δ_{km} sur $r_{ikm},\ i = 1, 2, \ldots, N$

d) Mettre à jour $F_{km}(X) = F_{k,m-1}(X) + \Delta_m(X)$

Étape 3 : Retourner $\hat{y}_k = F_{kM}(X),\ k = 1, 2, \ldots, K.$

On initialise le premier modèle F_{k0} à l'étape 1. On remarque que la probabilité associée à chaque classe est uniforme, elle vaut $\frac{1}{K}$.

A l'étape 2, on calcule la probabilité associée à chaque classe k en fonction du modèle F_{km} construit précédemment puis on calcule les résidus r_{ikm}, correspondent toujours au gradient négatif de la fonction de perte, sur lesquels on ajuste l'apprenant faible Δ_{km} avant de mettre à jour le modèle composite F_{km}. On note qu'un seul apprenant faible est nécessaire pour une classification binaire.

Le principe des algorithmes gradient boosting a été exposé. Les sections suivantes présentent des techniques permettant d'optimiser l'apprentissage.

2.4 Régularisation

De part leur grande flexibilité, les algorithmes de Gradient Boosting sont sujets au sur-apprentissage. C'est pourquoi utiliser des techniques de régularisation peut significativement améliorer les performances du modèle.

Premièrement, choisir les sous-modèles plus simple va mécaniquement réduire la complexité du modèle final.

Deuxièmement, des sous-modèles choisis sont optimaux à chaque itération. Bien que cette stratégie génère une solution optimale à l'étape actuelle, elle présente l'inconvénient de ne pas construire le modèle global optimal et de sur-apprendre les données d'apprentissage. Pour contrer cela, on fixe un taux d'apprentissage η permettant de réguler la vitesse d'apprentissage du modèle.

$$F_m(x) = F_{m-1}(x) + \eta\Delta_m(x)$$

Plus η est petit plus la contribution des nouveaux apprenants est faible, plus l'apprentissage est ralenti. Les prédictions du modèle final sont alors :

$$\hat{y}_i = F_M(x_i) = \sum_{m=1}^{M} \eta\Delta_m(x_i)$$

Troisièmement, pour améliorer l'apprentissage, il est possible d'ajouter de la diversité entre les sous-modèles en sélectionnant aléatoirement un sous-ensemble des données. Cette méthode est appelée Gradient Boosting stochastique. De la même manière, on peut aussi entraîner les apprenants faibles sur un sous échantillon aléatoire des variables descriptives. Ces méthodes permettent de réduire la variance du modèle composite ainsi que le temps d'entraînement.

2.5 Gradient Boosting Decision Tree

Les arbres de décision sont les estimateurs de base les plus populaires. En effet, ils constituent une méthode "prête à l'emploi" dans le sens où un grand nombre de traitements préalables fastidieux ou un réglage minutieux de la procédure d'apprentissage peuvent être évités. On parle alors de Gradient Boosting Decision Tree.

Les avantage des arbres de décision sont les suivants :
- la simplicité de compréhension et d'interprétation du modèle avec la hiérarchisation explicite de l'information, c'est un modèle *white box*.
- la robustesse en présence de valeurs aberrantes.
- la gestion de variables quantitatives et qualitatives.
- la non prise en compte des variables constantes.
- la possible gestion des valeurs manquantes (cf. 3.1.3).
- une complexité logarithmique en fonction du nombre d'individus lors de la phase de prédiction.

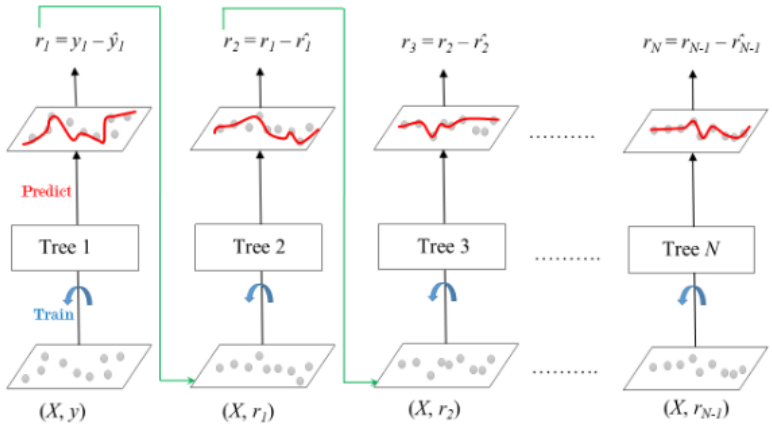

FIGURE 2.3 – Gradient Boosting Decision Tree. Source :GeeksforGeeks.

2.6 Résumé

Les algorithmes de Gradient Boosting sont basés sur trois principaux éléments :
- une fonction coût à optimiser
- un sous modèle appelé apprenant faible pour effectuer des prédictions
- un modèle additif pour concaténer les prédictions des apprenants faibles s'ajustant sur l'opposé du gradient de la fonction coût calculée à chaque itération.

La complexité du modèle dépend du nombre d'itérations effectuées, du taux d'apprentissage et de la complexité des apprenants faibles.

L'apprenant faible peut être divers. Le plus populaire est l'arbre de décision. Il présente l'avantage d'être non paramétrique, d'avoir une bonne interprétabilité et explicabilité ainsi que de supporter les données manquantes et redondantes.

Les modèles de Gradient Boosting et en particulier les modèles GBDT sont sujet au sur-apprentissage. Pour contrer cela, il est possible de régler les hyperparamètres tels que :
- le nombre d'itération.
- le taux d'apprentissage.
- la profondeur des arbres et/ou le nombre maximum de nœuds terminaux.
- le taux d'échantillonnage des individus.
- le taux d'échantillonnage des variables.

Chapitre 3

Librairie de Gradient Boosting

3.1 XGBoost

XGBoost est une librairie de machine learning très populaire permettant de construire des modèles de gradient boosting de manière optimisé et parallélisable. Étant nativement implémentée en langage C pour des questions de rapidité d'exécution, la librairie propose également une interface dans de nombreux langages dont Python. Nous allons présenter les principales fonctionnalités qui permettent à XGBoost d'être efficace, flexible et portable.

Nous avons évoqué dans la section 2.5 de nombreux avantages en faveur des arbres de décisions en tant qu'apprenant faible. C'est en effet le sous-modèle privilégié par XGBoost. Nous allons donc voir dans un premier temps comment est-il possible d'introduire la notion de régularisation au cours de la construction des arbres de décision. Puis nous présenterons l'algorithme qui en découle ainsi qu'une seconde version pour les données très volumineuses. Enfin, nous verrons comment XGBoost traite les données manquantes.

3.1.1 Tree booster et régularisation

Pour rappel, on dispose d'un ensemble de données avec n observations et p caractéristiques tel que $D = (x_i, y_i)_{i=1}^{n}$ où $x_i \in \mathbb{R}^p$ et $y_i \in R$. Un modèle XGBoost utilise M sous-modèles appelés *booster* pour prédire la sortie tel que :

$$\hat{y}_i = \sum_{m=1}^{M} \Delta_m(x_i), \quad \Delta_m \in \mathcal{F}$$

Si Δ est un arbre de décision alors l'ensemble des sous-modèles pouvant être construits est noté :

$$\mathcal{F} = \{\Delta(x) = w_{q(x)}\}$$

où $q : \mathcal{X} \to T$ et $w \in R^T$. Ici, q représente la structure de chaque arbre de décision reliant une observation à l'indice de la feuille correspondante. T est le nombre de feuilles dans l'arbre. Chaque feuille contient un score continu tel que w_i représente le score de la i-ème feuille. Pour une observation donnée, nous utiliserons les règles de décision des arbres (données par la structure q) pour classer les données. La prédiction finale pour une observation donnée est la somme des prédictions de chaque arbre. La minimisation d'une

fonction objectif régularisée guide la construction de l'ensemble des apprenants faibles du modèle.

A l'itération m, la fonction objectif s'écrit :

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

où

$$\Omega(\Delta) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

La complexité du modèle est pénalisée lorsque le nombre de nœuds terminaux T est grand. De plus, on applique une pénalisation L^2 sur les poids associés à ces derniers, autrement dit, on favorise le modèle avec de faibles poids. L'approximation par un développement de Taylor donne :

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

où

$$g_i = \partial_{\hat{y}_i^{(m-1)}} l(y_i, \hat{y}_i^{(m-1)}) \quad \text{et} \quad h_i = \partial^2_{\hat{y}_i^{(m-1)}} l(y_i, \hat{y}_i^{(m-1)})$$

sont les dérivées d'ordre 1 et 2 de la fonction de perte, autrement dit le gradient et la hessienne.
En supprimant le terme constant, la fonction objectif à l'étape m s'écrit :

$$\tilde{\mathcal{L}}^{(m)} = \sum_{i=1}^{n} \left[g_i \Delta_m(x_i) + \frac{1}{2} h_i \Delta_m^2(x_i) \right] + \Omega(\Delta_m)$$

Puis, en développant le terme Ω et en regroupant par feuille tel que $I_j = \{i, q(x_i) = j\}$ soit l'ensemble des éléments du nœud j, on obtient :

$$\begin{aligned} \tilde{\mathcal{L}}^{(m)} &= \sum_{i=1}^{n} [g_i \Delta_m(x_i) + \frac{1}{2} h_i \Delta_m^2(x_i)] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned}$$

La fonction de perte l est convexe et deux fois différentiables. Donc, pour une structure d'arbre $q(x)$ fixée, on peut calculer le poids optimal pour la feuille j :

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} = -\frac{G_j}{H_j + \lambda}$$

en notant $G_j = \sum_{i \in I_j} g_i$ la somme des gradients et $H_j = \sum_{i \in I_j} h_i$ la somme des hesiennes de la feuille j. On calcule ensuite la valeur optimale correspondante :

$$\tilde{\mathcal{L}}^{(m)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

Cette équation permet de mesurer la qualité d'une structure d'arbre q. Le score obtenue est semblable au score d'impureté pour l'évaluation des arbres de décision CART. Néanmoins, on dispose d'une gamme plus large de fonctions objectives car toute fonction convexe deux fois dérivable peut être utilisée comme fonction de perte.

En pratique, il est impossible d'énumérer toutes les structures d'arbres possibles. C'est pourquoi, l'algorithme part d'un seul nœud (la racine de l'arbre) et ajoute itérativement des branches à l'arbre. Pour effectuer la division d'un nœud, on itère sur les observations présentes dans le nœud en calculant le gain :

$$gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma \tag{3.1}$$

ou G_L, H_L et G_R, H_R sont les sommes des gradients et hessiennes des éléments du nœud gauche et du nœud droit. Si le gain est négatif, la division n'a pas lieu et le nœud devient un nœud terminale.

Il existe plusieurs algorithmes qui effectuent la recherche du partitionnement optimal. La section suivante les présente brièvement.

3.1.2 Algorithme de partitionnement

Trouver la meilleur division à chaque nœud est la tâche la plus complexe dans un arbre de décision. On effectue donc une approximation en restreignant les partitions possibles à l'ensemble des valeurs prises par chaque variables dans le jeu de données :

$$\{\{X^j \leq x_{ij}\}, \quad j = 1, \ldots, p, \quad i = 1, \ldots, n\}$$

Afin d'optimiser le temps de calcul, les données sont triées selon les valeurs des variables. Puis l'algorithme les parcourt en additionnant leurs gradients et hessiennes afin de calculer le gain avec l'équation 3.1.

Algorithme 3 : Exact Greedy Algorithm for Split Finding

Données : I ensemble des instances du nœud, p nombre de variables

$gain = 0$

$G = \sum_{i \in I} g_i, \quad H = \sum_{i \in I} h_i$

pour $j = 1$ *à* p **faire**

  $G_L = 0$, $H_L = 0$

  pour i *dans* $(I, par\, asc(x_{ij}))$ **faire**

    $G_L = G_L + g_i$, $H_L = H_L + h_i$

    $G_R = G - G_L$, $H_R = H - H_L$

    $gain = \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

  fin

fin

Sorties : Division avec le gain maximal

Cette algorithme propose une approximation d'autant plus précise qu'il y a de données. Néanmoins, lorsque le volume de ces dernières est trop important, elles ne peuvent être chargée en mémoire. La méthode exacte n'est alors plus opérationnelle.

Une seconde approche consiste à utiliser une représentation des données par histogramme. La recherche des meilleurs divisions s'effectue alors à l'aide des déciles. Les histogrammes peuvent être calculés pour chaque arbre ou pour chaque nœud. Le papier [2] présente en détail dans son annexe cet algorithme appelé Weighted Quantile Sketch permettant de construire déciles sur un jeu de données avec des poids associés aux observations. Un fois les déciles calculés, on peut appliquer l'algorithme 4.

Algorithme 4 : Approximate Algorithm for Split Finding

Données : $S = \{\{s_{j1}, s_{j2}, \ldots, s_{jl}\}\}_{j=1}^{l}$: ensemble des bornes des histogrammes des variables, p : nombre de variables

$gain = 0$

$G = \sum_{i \in I} g_i, \quad H = \sum_{i \in I} h_i$

pour $j = 1$ *à* p **faire**

  $G_L = 0,\ H_L = 0$

  pour $v = 1$ *à* l **faire**

    $g_v = \sum_{i \in \{i | s_{j,v-1} < x_{ij} \leq s_{j,v}\}} g_i, \quad h_v = \sum_{i \in \{i | s_{j,v-1} < x_{ij} \leq s_{j,v}\}} g_i$

    $G_L = G_L + g_v,\ H_L = H_L + h_v$

    $G_R = G - G_L,\ H_R = H - H_L$

    $gain = \max(gain,\ \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

  fin

fin

Sorties : Division avec le gain maximal

3.1.3 Données manquantes

Dans la pratique, les données sont fréquemment parsemées de valeurs manquantes. Si on souhaite utiliser par exemple un modèle linéaire, il n'est pas possible d'utiliser les données en l'état car l'algorithme utilise le produit matriciel. Plusieurs solutions sont possibles pour traiter ce problème.

Premièrement, on peut choisir de supprimer les valeurs manquantes. Si leur nombre est trop important alors l'apprentissage va en pâtir à cause de la perte d'information en terme d'exemple ou de caractéristique.

Deuxièmement, on peut choisir d'imputer les valeurs nulles. Soit par une statistique soit par les prédictions d'un modèle tiers. Dans les deux cas, ces valeurs seront sujet à un biais.

Troisièmement, il est possible d'utiliser les arbres de décision qui ont la capacité de traiter les données creuses. Lors de l'entraînement du modèle pour chaque division, les valeurs nulles seront affectées au nœud fils ayant le plus gros gain.

Algorithme 5 : Division avec données manquantes

Données : I l'ensemble des instances du nœud, $I_k = \{i \in I,\ x_{ik} \neq nulle\}$, m le nombre de variables

$gain = 0$
$G = \sum_{i \in I} g_i$, $H = \sum_{i \in I} h_i$
pour $k = 1$ *à* m **faire**
  Attribuer les valeurs manquantes à droite.
  $G_L = 0$, $H_L = 0$
  pour j *dans* $(I, asc(x_{jk}))$ **faire**
    $G_L = G_L + g_j$, $H_L = H_L + h_j$
    $G_R = G - G_L$, $H_R = H - H_L$
    $gain = \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  fin
  Attribuer les valeurs manquantes à gauche.
  $G_R = 0$, $H_R = 0$
  pour j *dans* $(I, desc(x_{jk}))$ **faire**
    $G_R = G_R + g_j$, $H_R = H_R + h_j$
    $G_L = G - G_R$, $H_L = H - H_R$
    $gain = \max(gain, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  fin
fin

Sorties : Division et direction par défaut des valeurs manquantes avec le gain maximal

En procédant ainsi, on considère en quelque sorte que le fait de ne pas avoir d'information est une information en soit. C'est aussi un gain de temps dans le pré-traitement des données.

3.1.4 Résumé

La librairie XGBoost permet de construire des modèles de machines learning portables grâce à l'implémentation des algorithmes dans de nombreux langages. De plus, les modèles XGBoost se montrent efficaces de part l'utilisation de la méthode de gradient boosting et d'autre part en y intégrant des régularisations au cours de l'apprentissage. Nous développerons ce point lors de l'optimisation des hyperparamètres dans le chapitre 4. Enfin, la construction de modèle est accélérée grâce au traitement automatique des valeurs nulles.

La section suivante est dédiée à la librairie de gradient boosting, également très populaire, LightGBM.

3.2 LightGBM

LightGBM est une librairie implémentant des algorithmes de gradient boosting avec arbres de décision et proposant des optimisations afin de facilité le traitement des données (très) volumineuses. En effet, pour les GBTD, le goulot d'étranglement en terme de complexité de calcul se trouve dans la cherche du meilleur point de partitionnement. Ainsi dans le but de réduire le temps de calcul LightGBM utilise deux algorithmes : *Gradient-based One-Side Sampling* (GOSS) et *Exclusive variable Bundling* (EFB). Le premier algorithme permet de basé les calculs sur un sous-ensemble des instances en privilégiant les instances avec un fort gradient qui tendent à jouer un rôle plus important dans le gain d'information. Le second algorithme, réduit le nombre de variable en se basant sur l'exclusivité mutuelle des variables et en approximant la solution de ce problème.

3.2.1 Gradient-based One-Side Sampling

L'algorithme GOSS part du constat suivant : les observations avec différents gradients jouent des rôles différents dans le calcul du gain d'information. En particulier, selon la définition du gain d'information, les instances avec des gradients plus grands, c'est-à-dire les instances sous-entraînées, contribueront davantage au gain d'information. Par conséquent, lors de l'échantillonnage descendant des instances de données, afin de conserver la précision de l'estimation du gain d'information, il est préférable de conserver les instances avec des gradients importants, et d'éliminer de manière aléatoire les instances avec des gradients faibles.

Ainsi, il est possible de reprendre l'algorithme 1 est d'ajouter entre les étapes 2.a) et 2.b) la procédure suivante :

1. Trier les instances selon la valeur absolue des gradients par ordre décroissant.
2. Sélectionner les instances les plus élevées a * 100%.
3. Échantillonner aléatoirement b * 100% des instances restantes. Cela réduira la contribution des exemples bien entraînés par un facteur $b < 1$.
4. Sans le point 3, le nombre d'échantillons ayant de petits gradients serait de $1 - a$ (actuellement il est de b). Afin de maintenir la distribution originale, LightGBM amplifie la contribution des échantillons ayant de petits gradients par une constante $\frac{1-a}{b}$ pour mettre davantage l'accent sur les instances sous-entraînées sans modifier excessivement la distribution des données.

Algorithme 6 : GOSS

Données : $(X, y) = \{(x_i, y_i)\}_{i=1}^{n}$

Entrées : a : ratio de sous-échantillonnage des données avec un fort gradient,
b : ratio de sous-échantillonnage des données avec un faible gradient

Étape 0 : Initialiser $fact = \frac{1-a}{b}, \quad top\_n = a \times n, \quad rand\_n = b \times n$

Étape 1 : Initialiser $F_0(X) = f_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} l(y_i, \gamma)$

Étape 2 : Pour $m = 1$ à M :

a) Calculer les gradients $r_m = -\left[\frac{\partial l(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}\right]_{i=1,\ldots,n}$

b) Trier les gradient $sorted = GetSortedIndices(asb(r_m))$

c) $top\_set = sorted[1 : top\_n]$

d) $rand\_set = RandomPick(sorted[top\_n : n], rand\_n)$

e) Sous-échantillonner $new\_set = top\_set + random\_set$

f) Initialiser les poids $w_m = \{1, 1, \ldots\}$

g) Sur-pondérer l'échantillon aléatoire $w_m[rand\_set] = w_m[rand\_set] \times fact$

h) Entraîner un apprenant faible Δ_m sur r_m pondéré par w_m

i) Mettre à jour $F_m(X) = F_{m-1}(x) + \Delta_m(X)$

Sorties : $\hat{y} = F_M(X)$

Le papier original LightGBM [3] propose une analyse théorique de l'algorithme GOSS qui aboutit à deux résultats.

Premièrement, on obtient une borne supérieur sur l'erreur d'approximation du gain par rapport à la méthode exacte 3. Cette erreur tend vers 0 en $\mathcal{O}(\sqrt{n})$ lorsque le nombre d'observations augmente. De plus, GOSS peut être meilleur qu'un sous-échantillonage aléatoire sous la condition :

$$\frac{\alpha_a}{\sqrt{b}} > \frac{1-a}{\sqrt{b-a}} \quad \text{où} \quad \alpha_a = \frac{\max_{x_i \in A \cup A^c} |g_i|}{\max_{x_i \in A^c} |g_i|}$$

tel que A soit le top $a * 100\%$ des instances avec un fort gradient.

Deuxièmement, l'erreur de généralisation avec GOSS sera proche de celle calculée en utilisant toutes les observations si l'approximation est précise, c'est à dire si il y a assez d'observations. D'autre part, l'échantillonnage augmentera la diversité des apprenants de base, cela peut aider à améliorer la performance de la généralisation.

3.2.2 Exclusive variable Bundling

En grande dimension, les données sont généralement très éparses. On appelle cela le fléau de la grande dimension. Néanmoins, dans un tel espace de nombreuses caractéristiques sont mutuellement exclusives, c'est-à-dire qu'elles ne prennent jamais de valeurs non nulles simultanément. Il est donc possible de regrouper ces variables exclusives en une seule variable appelée *exclusive variable bundle* que l'on peut traduire par faisceau ou parquet. Ainsi, grâce à un algorithme de balayage des caractéristiques, il est possible de construire les mêmes histogrammes de variables à partir des *bundles*. De cette façon, la complexité de la construction des histogrammes passe de $\mathcal{O}(\#data \times \#variable)$ à $\mathcal{O}(\#data \times \#bundle)$ avec $\#bundle << \#variable$. L'apprentissage est alors considérablement accéléré sans nuire à la précision.

Il y a deux problèmes à résoudre. Le premier est de déterminer quelles caractéristiques doivent être regroupées. Le second est de savoir comment transformer chacun des parquets en une variable, un *bundle*.

Regroupement

Regrouper les variables mutuellement exclusives par parquets correspond à un problème coloration de graphe. Les variables constituent les sommets du graphes et les arrêtes relient les variables non mutuellement exclusives. Cependant, les problèmes de coloration de graphe sont NP-difficiles, c'est à dire qu'ils ont une grande complexité et qu'ils ne peuvent pas être résolus en un temps polynomial. C'est pourquoi, un algorithme d'approximation est utilisé afin de produire les regroupements.

En effet, si on tolère un petit taux de conflits γ au sein d'un même paquet alors la précision de l'entraînement sera affectée [1]. Il y a donc un juste milieu à trouver entre la précision et l'efficacité. Afin de prendre en compte le nombre de conflits entre chaque pair de variable, on pondère l'arrête qui les relie par cette quantité. L'algorithme 7 présente la procédure de regroupement des variables (presque) exclusives. Il est appliqué une seule fois avant entraînement et sa complexité vaut $\mathcal{O}(\#variable^2)$.

Détaillons l'algorithme de groupement des variables. Dans un premier temps, le graphe est construit comme évoqué précédemment. Puis on récupère les index des variables triées en fonction de leur nombre de valeurs nulles par ordre décroissant. Enfin, on itère sur la liste des variables triées afin de l'ajouter à un *bundle* lorsque le nombre de conflits est assez faible sinon un nouveau *bundle* est créé.

1. Cette perte est d'au plus $\mathcal{O}([(1-\gamma)n]^{-2/3})$, voir [3]

Algorithme 7 : EFB - Regroupement

Données : $X = (X^j)_{j=1}^n$: variables

Entrées : K : nombre de conflits maximum par *bundle*

$G = \text{ComputeGaph}((X^j)_{j=1}^n)$

$searchOrder = G.sortNumNull()$

$bundles = \{\}$, $bundlesConflic = \{\}$

pour i *dans* $searchOrder$ **faire**

  $needNew = True$

  pour $j = 1$ *à* $len(bundles)$ **faire**

    $cnt = \text{ConflictCnt}(bundles[j], X^i)$

    si $cnt + bundlesConflict[i] \leq K$ **alors**

      $bundles[j].add(X^i)$

      $needNew = False$

      break

    fin

    si $needNew$ **alors**

      Add X^i as new bundle to *bundles*

    fin

  fin

fin

Sorties : *bundles*

Fusion

La seconde partie de algorithme EFB propose moyen de fusionner les variables d'un même *bundle* en une seule variable dans le but de réduire le temps d'entraînement. L'essentiel est de s'assurer que les valeurs des variables originales puissent être identifiées à partir des nouvelles.

Pour ce faire, on va construire un histogramme à partir des variables (continue ou non) en prenant soin de mettre dans des cases différentes les variables exclusives. Ceci peut être fait en ajoutant des décalages aux valeurs originales des variables.

Par exemple, supposons que nous ayons deux caractéristiques dans un *bundle*. À l'origine, la variable A prend la valeur $[0; 10)$ et la variable B prend la valeur $[0; 20)$. Nous ajoutons ensuite un décalage de 10 aux valeurs de la caractéristique B afin que la variable finale prenne la valeur $[10, 30)$. Après cela, il est possible de fusionner les caractéristiques A et B et d'utiliser une variable agrégée avec la plage $[0\,;\,30]$ afin de remplacer les caractéristiques originales A et B.

Algorithme 8 : EFB - Fusion

Entrées : n : nombre d'observation, B : un *bundles*

$binRanges = 0$, $totalBin = 0$

pour b *in* B **faire**
  $totalBin+ = b.numBin$
  $binRanges$.append($totalBin$)
fin

$newBin = \text{new}Bin(numData)$

pour $i = 1$ *à* n **faire**
  $newBin[i] = 0$
  pour $j = 1$ *à* $len(B)$ **faire**
    si $B[j].bin[i] \neq 0$ **alors**
      $newBin[i] = B[j].bin[i] + binRanges[j]$
    fin
  fin
fin

Sorties : $newBins$, $binRanges$

Chapitre 4

Optimisation des hyperparamètres

De manière générale, un modèle d'apprentissage automatique dispose de deux types de paramètre.

Premièrement, il y a les paramètres qui s'ajustent durant l'apprentissage. Pour les modèles de gradient boosting à base d'arbres de décision, les paramètres appris au cours de l'apprentissage correspondent à l'ensemble des paires $\{(variable, observation)\}$ utilisées pour partitionner les données à chaque nœud et aux poids w associés. Nous avons vu que chaque algorithme dispose de ses propres méthodes pour effectuer la recherche des paramètres optimaux (*cf.* algorithmes 3, 4, 6, 7, 8). Ainsi, une fois le type d'algorithme choisit, ces paramètres sont ajustés automatiquement.

Deuxièmement, il y a les paramètres fixés en amont de l'apprentissage. On les appelle hyperparamètres. C'est en modifiant la valeur de ces derniers qu'il est possible d'orienter la structure des arbres, le temps de calcul et surtout les performances du modèle.

La première section de ce chapitre détaille les principaux hyperparamètres des méthodes de gradient boosting. La seconde section présente des approches afin de trouver la combinaison idéale.

4.1 Hyperparamètre

4.1.1 Hyperparamètres structurels

Nombre d'estimateurs

Le premier hyperparamètre à considérer lors de la configuration de l'entraînement d'une forêt d'arbres de décision est le nombre d'estimateurs. Ce nombre indique combien d'arbres vont être entraînés, de manière séquentielle et incrémentale, pour construire le prédicteur final.

Le raisonnement qui doit sous-tendre le choix de la valeur de ce paramètre s'appuie sur la notion de compromis biais-variance, évoquée au chapitre 1. Pour rappel, ce compromis est au cœur de toute approche de modélisation, et concerne l'endroit où est positionné le curseur entre un modèle simple mais systématiquement biaisé et un modèle complexe mais probablement sur-entraîné pour un ensemble de données.

Appliqué à la détermination du nombre d'estimateurs, ce principe revient à choisir un nombre suffisamment grand pour capturer la variabilité des données tout en évitant de sur-apprendre.

Profondeur maximale

L'autre paramètre à prendre en compte pour influer sur la structure des arbres appris est la profondeur maximale. Elle indique tout simplement la profondeur maximale que peut atteindre un arbre. C'est bien un maximum, qui peut ne pas être atteint, si le nombre d'échantillons n'est pas suffisant pour ajouter un étage de plus ou si le gain apporté n'est pas suffisant. La profondeur maximale pilote donc indirectement le nombre de nœuds de l'arbre et, plus important encore, le nombre de feuilles.

Au-delà d'être un paramètre dimensionnant pour la structure des arbres, il est à mettre en relation avec le nombre de données utilisées pour l'entraînement. Les arbres de décision étant quasi systématiquement des arbres binaires, une profondeur de n étages correspond à un nombre de feuilles de 2^n.

Le calcul des poids associés à une feuille se faisant sur la base des échantillons de données présentes dans cette feuille, il faut donc un minimum de 2^{n-1} données pour entraîner un arbre de profondeur n.

De plus, pour estimer quelle profondeur d'arbre est nécessaire on peut également remarquer que ce choix à un impact sur l'ordre d'interactions des variables. Avec un arbre de profondeur 1, les données sont partitionnées en deux selon une seule variables. L'ordre d'interaction vaut 1. Autrement dit, les prédictions d'un arbre reposent sur une seule variable. Avec un arbre de profondeur 2, on sépare les données selon une première variable puis selon une deuxième. On capte alors les interactions d'ordre 2. Et ainsi de suite. Donc pour un arbre binaire de profondeur n, il est possible de capter les interactions d'ordre n.

Dans la pratique l'ordre des interactions n'est pas connu. C'est pourquoi, on teste un ensemble de valeurs pertinentes et on choisit la valeur donnant le meilleur résultat sur le jeu de validation.

4.1.2 Hyperparamètres d'apprentissage

En plus des hyperparamètres gouvernant la construction des arbres de décision, il existe une série de paramètres qui influent sur l'apprentissage.

Lorsqu'est considéré un paramètre d'entraînement, il est important d'identifier à quelle étape de la construction de l'arbre de décision il intervient. Deux possibilités sont à envisager :

- Le paramètre intervient lors du calcul du gain. Pour rappel, le gain quantifie l'intérêt de l'ajout d'un nouvel étage à l'arbre de décision. Un gain important va motiver cet ajout, tandis qu'un gain trop faible va stopper l'expansion de l'arbre. L'impact dans ce cas est donc structurel.
- Le paramètre intervient dans le calcul du poids optimal attaché à chaque feuille de l'arbre. L'impact se situe ainsi directement au niveau de la correction qui va être apportée. C'est donc directement la prédiction qui va être affectée.

À noter que ces deux possibilités ne sont pas mutuellement exclusives. Certains paramètres peuvent influer sur ces deux étapes.

Taux d'apprentissage

Le taux d'apprentissage, ou learning rate en anglais, déjà évoque en section 2.4, est une valeur comprise entre 0 et 1. Elle permet de quantifier quelle fraction de la correction va être prise en compte. En conséquence, il s'agit d'un paramètre correspondant à la seconde possibilité listée ci-dessus, et qui va influer sur la valeur prédite. C'est-à-dire que le poids calculé à l'aide de la formule

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} = -\frac{G_j}{H_j + \lambda}$$

va être multiplié par ce taux d'apprentissage. S'il vaut zéro, aucune correction n'est appliquée et le modèle n'apprend rien. A contrario, si ce taux prend comme valeur 1, la correction est intégralement appliquée.

La vocation de ce taux d'apprentissage est d'éviter un sur-apprentissage, en ne transférant qu'une partie du poids optimal calculé. Il faut donc trouver le compromis entre un learning rate trop petit, qui implique un plus grand nombre d'estimateurs, et un learning rate proche de 1.0, qui risque d'entraîner un sur-apprentissage.

Paramètre de régularisation gamma

Gamma, noté γ, comme il a été vu dans le chapitre sur XGBoost, est un paramètre de régularisation des modèles générés. C'est-à-dire qu'il gouverne l'apprentissage des arbres de décision et pilote en particulier l'ajout de nouveaux nœuds sur la base du gain apporté. La raison en est donnée par la formule 3.1, qui détaille le calcul du gain optimal :

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

Ce gain fait apparaître le gradient G, la hessienne H, T le nombre de feuilles de l'arbre, le paramètre lambda et enfin gamma, et ce pour les nouveaux nœuds de droite (R) et gauche (L). Ce gain est calculé pour chaque possibilité de critère de séparation des données associées au nœud parent. Le meilleur critère est celui qui apporte le plus de gain.

Le rôle de gamma se révèle clairement à travers ce calcul du gain. Si le terme de gauche est inférieur à gamma, alors il est négatif et ne sera donc pas retenu. Ce paramètre est donc de nature à modifier la structure de l'arbre généré. Jouer sur gamma revient à contrôler la facilité avec laquelle un nœud est découpé en deux. Si gamma est nul, le découpage se fait automatiquement alors qu'avec une valeur de gamma strictement supérieure à zéro, le découpage n'a lieu que si le gain généré dépasse ce seuil.

Paramètre de régularisation de type L2 : lambda

Le second paramètre qui apparaît dans la formule du gain est lambda. Comme le rappelle la formule ci-dessus, l'impact de lambda se situe au niveau du dénominateur du

gain. Mais lambda se retrouve aussi dans l'expression calculant le poids optimal d'une feuille :

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

C'est donc un paramètre qui va jouer sur les deux tableaux : le pilotage de la structure de l'arbre de décision, à travers son intervention dans le calcul du gain et la valeur prédite à travers son intervention dans le calcul du poids d'une feuille.

Lorsque la fonction de perte est l'erreur au carré, la hessienne vaut 2. La somme des hessiennes H est donc le double du nombre de lignes appartenant au nœud courant. Ajouter lambda à cette valeur, se trouvant au dénominateur, revient donc à réduire le gain, et ce d'autant plus que les échantillons associés à ce nœud sont peu nombreux. En effet, plus il y a de points de données attachés à ce nœud, plus G et H augmente et plus l'impact de lambda devient négligeable.

Parallèlement, lambda va aussi avoir un effet sur le poids attaché à la feuille concernée, en tendant à réduire cette valeur d'autant plus que le nombre d'échantillons s'y trouvant est réduit. C'est un résultat intéressant, dans la mesure où une feuille contenant peu de données issue de l'entraînement aura en définitive un effet modéré sur la prédiction.

Paramètre de régularisation de type L1 : alpha

Dans le même ordre d'idée que lambda, il existe un autre paramètre défini pour influer nativement sur la valeur des poids. Il est généralement noté alpha et fait intervenir la somme de la valeur absolue des poids dans la fonction de régularisation.

Parallèlement à lambda qui réalise une régularisation de type L^2, puisque s'appliquant au carré des poids, alpha réalise une régularisation de type L^1, puisque s'appliquant sur les valeurs absolues des poids :

$$\Omega(\Delta) = \gamma T + \frac{1}{2}\lambda\|w\|^2 + \alpha|w|$$

À ce titre, alpha intervient conjointement sur la structure de l'arbre, en pesant sur le gain, mais aussi sur le poids des feuilles.

$$gain = \frac{1}{2}\left[\frac{reg_\alpha(G_L)^2}{H_L + \lambda} + \frac{reg_\alpha(G_R)^2}{H_R + \lambda} - \frac{reg_\alpha(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma$$

avec

$$reg_\alpha(x) = sign(x) \times \max(0, |x| - \alpha)$$

Les régularisations de type L^1 ont pour conséquence d'ajouter alpha au poids quand ce dernier est négatif, et à lui retirer alpha lorsqu'il est positif. Cela revient donc à le faire tendre vers zéro. Ce comportement tend à générer des modèles creux, sparse models en anglais, c'est-à-dire des modèles où un maximum de poids tendent vers zéro.

Les régularisations de type L^1 réalisent donc une forme de sélection des caractéristiques réellement discriminantes : c'est la *variable selection* en anglais. Cette étape est parfois réalisée en amont de l'entraînement du modèle, mais en sélectionnant ce type de régularisation, il est possible de s'en passer, ce qui simplifie la tâche.

4.1.3 Sous-échantillonage

Le sous-échantillonnage consiste à tirer au sort un pourcentage des variables ou des observations. Cela rend les arbres moins corrélés et plus robuste au bruit. De plus, cette technique empêche certains effets de masquage des caractéristiques. Le sous-échantillonage des observation s'effectue avant chaque itération de boosting. Le sous-échantillonage des variables peut s'effectuer par arbre, par niveau et par noeud.

4.2 Méthode d'optimisation

Identifier la bonne combinaison d'hyperparamètres n'est pas chose facile de prime abord. Cela dépend de la complexité du phénomène à modéliser. La manière la plus fiable de fixer ces paramètres est de procéder à plusieurs entraînements, en évaluant méthodiquement les performances sur le jeu d'entraînement et ceux de tests. Ces multiples entraînements peuvent se faire séquentiellement, à la main, ou être automatisés.

Ainsi, le principe sous-jacent aux méthodes d'optimisation des hyperparamètres est le suivant : il faut explorer l'espace de configuration et converger le plus rapidement possible vers la combinaison optimale selon un critère particulier.

Ce critère n'est pas nécessairement la même fonction que la méthode objectif utilisée pour optimiser les poids grâce au Gradient Boosting. Il peut s'agir de tout autre indicateur, et notamment il n'est pas requis que cet indicateur soit dérivable. Toute fonction retournant un scalaire peut fonctionner.

La difficulté réside dans la combinatoire à explorer qui est très grande. Il est fréquent de rencontrer des entraînements où l'espace à explorer s'étend sur :

— une profondeur allant de 3 à 10 niveaux,
— un nombre d'arbres allant de 5 à 500,
— gamma variant de 0.1 à 10,
— un taux d'apprentissage allant de 0.1 à 1,
— etc.

Le produit cartésien de toutes ces possibilités dépasse allègrement les 100 000 possibilités. Une exploration par brute force, en testant toutes les combinaisons rencontrées, est dans de rares cas possible, mais il faut dans la plupart des situations avoir recours à des méthodes plus intelligentes. C'est d'autant plus vrai que le jeu d'entraînement est large et que le temps d'apprentissage est long.

4.2.1 Force brute

La première méthode envisageable, évoquée ci-dessus, est celle de la brute force. Elle consiste simplement à tester toutes les combinaisons possibles et à retenir celle donnant le critère optimal.

Limitée à des cas où l'espace de configuration est peu étendu, ce type de méthode est appelé en anglais Grid Search. Scikit-learn offre une méthode générique qui implémente cette stratégie : GridSearchCV.

4.2.2 HalvingGridSearch

Une variante de la méthode par brute force consiste à effectuer la recherche en travaillant sur un nombre d'observations de plus en plus important.

Dans une première itération, toute les combinaisons sont évaluées sur un petit sous-ensemble des données. À chaque itération suivante, seule la moitié des combinaisons présentant les meilleures performances est conservée, tandis que parallèlement le nombre de lignes de données est doublé. Les itérations se poursuivent ainsi, jusqu'à ce que toutes les données aient été utilisées ou qu'il ne reste plus qu'une combinaison.

Cette stratégie se retrouve dans la librairie scikit-learn sous la classe HalvingGridSearchCV. Son principal avantage réside dans le gain de temps qu'elle apporte, en écartant rapidement les combinaisons les moins prometteuses en travaillant sur peu de données.

4.2.3 Le hasard

Lorsque chaque entraînement coûte cher en temps de calcul, l'exploration systématique de toutes les combinaisons peut s'avérer impraticable, ou imposer une restriction sur l'ensemble des paramètres à explorer et donc réduire le potentiel du modèle. Dans ce cas, il est possible de s'appuyer sur le hasard pour explorer la combinatoire. Cela ne garantit pas que la meilleure combinaison va être retenue, mais cela permet de garder le contrôle sur le nombre d'itérations et le temps de calcul.

La classe RandomizedSearchCV de scikit-learn implémente cette stratégie.

4.2.4 Approche de type substitut

S'en remettre au hasard peut sembler étonnant, mais en pratique, cela permet généralement de trouver à moindre coût un ensemble d'hyperparamètres satisfaisant. Il reste néanmoins possible de passer à côté d'une configuration particulièrement bénéfique pour le modèle considéré.

L'idéal serait de procéder de manière non exhaustive, comme le fait l'approche aléatoire, tout en guidant la recherche vers les combinaisons les plus prometteuses.

C'est ce que font les méthodes d'optimisation basée sur l'utilisation d'un substitut. Le principe consiste à construire un modèle capable de prédire le score associé à une configuration d'hyperparamètres. Différents types de modèles sous-jacents sont envisageables. Il peut s'agir de combinaisons de gaussiennes, ou Gaussian Mixture en anglais, ou bien de modèles de type RandomForest, ou encore d'arbres de décision entraînés avec un Gradient Boosting.

Une implémentation de ce type de méthode, basée sur l'algorithme Tree Parzen Estimator de la librairie Hyperopt, est donnée dans la section suivante.

4.3 Exemple

Tout d'abord, nous importons des modules.

```
# Load data
from sklearn.datasets import load_iris
# Split data
from sklearn.model_selection import train_test_split
# Metrics
from sklearn.metrics import precision_score, recall_score, fbeta_score
# Model
import xgboost as xgb
# Hyperparameters optimization
from hyperopt import Trials, fmin, tpe, hp, STATUS_OK
from hyperopt.pyll.base import scope
# Save results
import mlflow

# Fixed seed to ensure reproducibility
SEED = 42
```

Puis nous chargeons les données et les divisons en trois : *train*, *valid*, *test*. La partie *train* sert à entraîner le modèle. La partie *valid* sert à sélectionner le modèle avec l'erreur de validation la plus faible. Et la partie *test* permet d'estimer l'erreur que commettrait le modèle sur des données nouvelles.

```
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25
                                    , stratify=y, random_state=SEED)
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train,
                                    test_size=0.25, stratify=y_train,
                                    random_state=SEED)
```

Nous définissons une fonction permettant de calculer des mesures de performance : la précision, le rappelle, et le score F0.5.

```
def classification_metrics(y_true, y_pred):
    return {
        "precision": precision_score(y_true, y_pred, average="micro"),
        "recall": recall_score(y_true, y_pred, average="micro"),
        "fbeta_score": fbeta_score(y_true, y_pred, average="micro", beta
                                    =0.5),
    }
```

La fonction *fit_model* permet d'entraîner le modèle, ainsi que de calculer les erreurs d'entraînement et de validation. De plus, la librairie MLFlow permet de sauvegarder très simplement le résultat des expériences y comprit les modèles entraînés.

```
def fit_model(model, X_train, X_valid, y_train, y_valid, params, metric)
                                        :
    with mlflow.start_run(nested=True) as sub_run:
        model = model(**params)
        model.fit(X_train, y_train)
        y_pred_train = model.predict(X_train)
        y_pred_valid = model.predict(X_valid)
        metrics_train = {f"train_{metric}": value for metric, value in
                                            classification_metrics(
                                            y_train, y_pred_train).items
                                            ()}
        metrics_valid = {f"valid_{metric}": value for metric, value in
                                            classification_metrics(
                                            y_valid, y_pred_valid).items
                                            ()}
        metrics = {**metrics_train, **metrics_valid}

        mlflow.log_metrics(metrics)
        mlflow.log_params(model.get_params())
        mlflow.xgboost.log_model(model, "model")

    return {
        "status": STATUS_OK,
        "loss": metrics[metric],
    }
```

La fonction *build_train_objective* initialise la fonction *train_func* avec les arguments adéquats.

```
def build_train_objective(model, X_train, X_valid, y_train, y_valid,
                                        metric):
    def train_func(params: dict):
        return fit_model(model, X_train, X_valid, y_train, y_valid,
                                            params, metric)
    return train_func
```

Enfin la fonction *evaluate* combine l'ensemble en initialisant l'expérience MLFlow, les *Trials* d'Hyperopt et la fonction *train_ objective*.

```
def evaluate(model,
             X_train,
             X_valid,
             y_train,
             y_valid,
             space,
             max_evals,
             metric,
             experiment_name):

    mlflow.set_experiment(experiment_name)

    trials = Trials()

    train_objective = build_train_objective(model, X_train, X_valid,
                                            y_train, y_valid, metric)

    with mlflow.start_run() as run:
        argmin = fmin(
            fn = train_objective,
            space = space,
            algo=tpe.suggest,
            max_evals=max_evals,
            trials=trials)
```

Il ne reste plus qu'à définir l'espace de recherche des hyperparamètres et à lancer les entraînements.

```
space = {
    "nthread": 1,
    "use_label_encoder": False,
    "objective": "multi:softmax",
    "eval_metric":"mlogloss",
    "n_estimators": scope.int(hp.uniform("n_estimators", 5, 50)),
    "learning_rate": hp.uniform("learning_rate", 0.01, 0.5),
}

evaluate(xgb.XGBClassifier, X_train, X_valid,  y_train, y_valid, space,
                                    max_evals=10, metric="
                                    valid_fbeta_score", experiment_name=
                                    "Iris_XGBoost")
```

Les résultats sont accessibles via l'interface utilisateur de MLFlow en tapant "*mlflow ui*" dans un terminal et en cliquant sur le lien https.

Avec ce code il donc possible d'effectuer une recherche des meilleurs hyperparamètres et de sauvegarder les résultats des expériences simplement en définissant un espace de recherche et en lançant la dernière ligne de code.

Chapitre 5

Explicabilité

L'explicabilité d'un modèle d'apprentissage automatique est le degré auquel un humain peut comprendre la cause d'une prédiction. Avoir une bonne explicabilité permet :

— d'avoir confiance dans les prédictions du modèle.
— de développer une meilleure compréhension et intuition du problème.
— de détecter les biais et les cas extrêmes apprit par le modèle.
— satisfaire des obligations réglementaires telles que le droit à l'explication formulé dans le texte européen nommé RGPD.

Nous avons vu précédemment qu'un modèle de machine learning est le résultat du traitement d'un ensemble de données par un algorithme. C'est pourquoi, comprendre les données et maîtriser l'algorithme est un pré-requis.

Les sections précédentes avaient pour but de mettre en lumière le fonctionnement des algorithmes de boosting. Nous avons aussi vu qu'un modèle de boosting est la combinaison de dizaines voir de centaine d'arbre de décision. On perd de ce fait l'interprétabilité qu'avait l'estimateur de base. On appelle ce type de modèle une boîte noire. C'est pourquoi, nous allons présenter des méthodes permettant de mieux comprendre les prédictions de ces modèles.

5.1 Importance des variables

Le calcul de l'importance des variables, *feature importances* en anglais, est la méthode d'explicabilité la plus commune. Elle est propre aux modèles reposant sur les arbres de décision. Elle attribue à chaque variable un poids donnant une idée de son importance au global. Plusieurs manières de calculer ces poids sont envisageables.

5.1.1 Calcul basé sur le niveau d'utilisation

Selon ce mode de calcul, l'importance est donné pour chaque variable par le nombre de fois où cette dernière se trouve impliquée au niveau d'un nœud, pour séparer les données en deux sous-ensembles. Ce mode est nommé mode *weight*. Cela donne une idée d'à quel point la variable est utilisée pour garantir un bon niveau de prédiction.

Une importance de ce type élevée implique généralement une plage de valeurs étendue pour cette variable, puisque la méthode de Gradient Boosting a trouvé pertinent de la découper en de nombreuses plages de valeurs différentes. Cette dépendance à la cardinalité de la variable peut introduire un biais dans l'explicabilité, dans la mesure où une variable

avec une large cardinalité a de fortes chances de se retrouver utilisée dans de nombreux nœuds.

5.1.2 Calcul basé sur les gains

Dans ce mode de calcul, nommé *gain*, le poids est calculé en moyennant les gains obtenus lorsque la variable considérée a été utilisée comme critère de séparation de l'ensemble de données.

Cela donne donc pour chaque variable une idée moyenne de l'impact de cette variable sur la réduction de l'erreur. L'avantage de ce mode de calcul est qu'il donne une idée précise de l'apport de la variable en termes de gain apporté au modèle, indépendamment du nombre de fois où elle est utilisée.

5.1.3 Calcul basé sur la couverture

Le dernier mode de calcul de la variable importance se base sur la couverture, c'est-à-dire sur le nombre d'échantillons concernés par la décision prise relativement à la variable considérée. Ce calcul se nomme *cover*. Dans ce cas de mesure, l'importance donne une idée du volume d'observations qui ont été impactées par cette variable, et donc une idée de l'impact de cette variable sur les prédictions.

Deux modes de calculs sont possibles : soit le nombre d'échantillons associés est moyenné, soit il est sommé.

5.1.4 Exemple

L'implémentation du calcul de l'importance de chaque variable, selon une vue globale, se fait donc de manière immédiate, et avec un surcoût négligeable aussi bien en termes de temps de calcul que de stockage. C'est pour cela qu'il est implémenté par défaut par les librairies XGBoost et LightGBM.

Ces trois types de graphiques apportent des informations différentes. Par exemple, une variable avec un gain élevé mais une couverture faible peut nous induire à penser qu'il existe des sous-ensembles au sein des données.

Voici un exemple de code permettant de calculer chaque type de *feature importance* pour un model XGBoost sur le jeu de données Boston :

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from xgboost import XGBRegressor, plot_importance

boston = load_boston()

X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = boston.target

xgb = XGBRegressor().fit(X, y)
```

```
fig, ax = plt.subplots(1,3,figsize=(14,5))
plot_importance(xgb.get_booster(), ax=ax[0], title='weight',
                importance_type='weight', show_values=False, height=0.5)
plot_importance(xgb.get_booster(), ax=ax[1], title='gain',
                importance_type='gain', show_values=False, height=0.5)
plot_importance(xgb.get_booster(), ax=ax[2], title='cover',
                importance_type='cover', show_values=False, height=0.5)
```

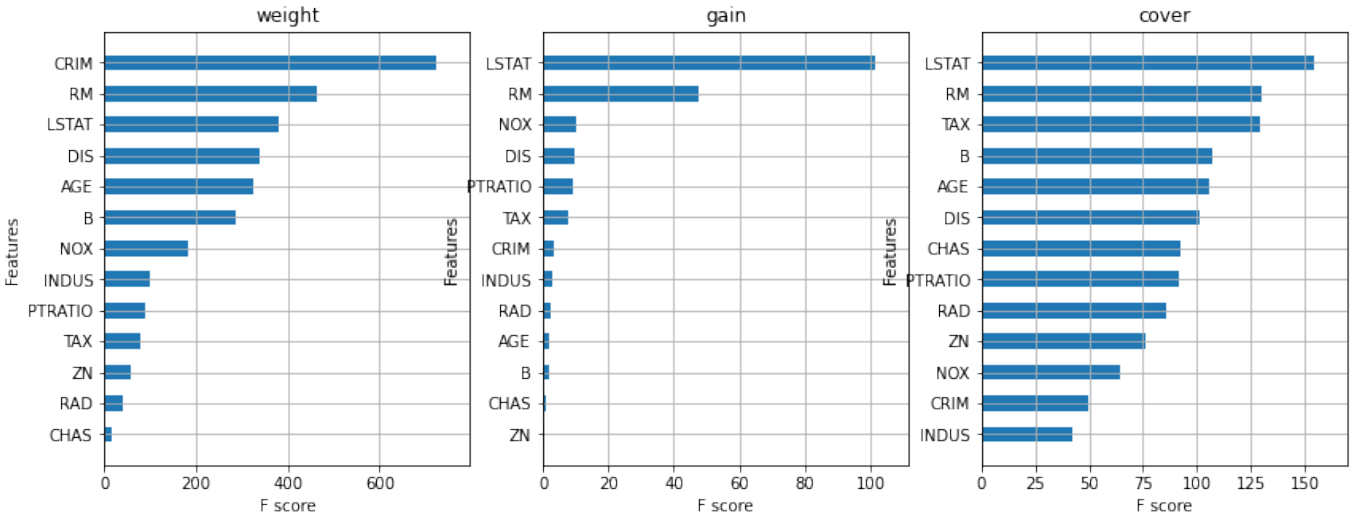

FIGURE 5.1 – Importance des variables

5.2 SHAP

La section précédente a présenté les *feature importances*, qui apportent un éclairage quantitatif sur le rôle de chaque variable, aussi bien dans la construction du modèle que dans la prédiction. Leur valeur explicative reste néanmoins cantonnée à une vue globale.

Dans cette section, la méthode SHAP, Shapley Additive exPlanation, va être présentée. Elle permet une compréhension plus fine d'un modèle, en détaillant de manière quantitative et signée l'impact de chaque variable sur chacune des prédiction. Cela signifie qu'elle informe sur la direction dans laquelle la prédiction a été tirée, à la hausse ou à la baisse, pour chaque variable.

Enfin, c'est une méthode générique qui s'applique à n'importe quel type de modèle, du réseau de neurones profond aux SVM, en passant bien sûr par les arbres de décision. On dit alors qu'elle est agnostique.

5.2.1 Objectif

Dans le cas où un modèle linéaire est utilisé pour réaliser une prédiction, cette dernière se calcule avec la formule suivante :

$$\hat{y}_i = \alpha_0 + \alpha_1 x_{i1} + \alpha_2 x_{i2} + \ldots + \alpha_p x_{ip}$$

La prédiction est alors une combinaison linéaire de chacune des variables X_j. Le poids d'une variable j dans la prédiction i est donc le produit du ième coefficient alpha avec la valeur de la ième variable.

Un tel modèle embarque directement un modèle explicatif de type additif, puisque la prédiction est la somme de chaque poids multiplié par chaque variable. Si le coefficient est positif, alors la variable influe sur la prédiction à la hausse. S'il est négatif, la prédiction est tirée vers le bas. La valeur absolue du produit de la variable par le coefficient donne l'importance de cette variable au sens de Shapley.

L'objectif de la méthode SHAP est de proposer une procédure pour construire automatiquement un tel modèle et qui soit localement additif. Il s'agit d'une certaine façon de linéariser le modèle, aussi complexe qu'il soit, pour en extraire un modèle explicatif linéaire. Sous forme mathématique, ce modèle explicatif g s'exprime comme suit :

$$g(z') = \phi_0 + \sum_{j=1}^{p} \phi_i z'_i$$

où le vecteur z' ne contient que des 0 et des 1. La présence d'un 1 en z'_j indique que la variable X_j est utilisée, tandis qu'un 0 indique que ce n'est pas le cas.

En appliquant ce principe au modèle linéaire calculant $\hat{y}_i$ donné ci-dessus, il est possible de le reformuler comme suit :

$$\hat{y}_i = \phi_0 + \sum_{j=1}^{p} (\alpha_j x_{ij}) z'_i$$

Cette reformulation met immédiatement en évidence le fait que dans le cas d'un modèle linéaire, pour une prédiction donnée, les valeurs de Shapley associées sont obtenues en multipliant les coefficients par leurs variables associées.

5.2.2 Valeurs de Shapley

Le type de modèle décrit ci-dessus est additif par construction, or il existe une infinité de possibilités pour décomposer un nombre en une somme de p nombres. Fort heureusement, en exigeant les propriétés suivantes pour les valeurs des variable importances :

- Exactitude locale : la somme des variable importances doit être égale à la prédiction.
- Absence : si une variable ne participe pas au modèle, alors l'importance associée doit être nulle.
- Consistence : si deux modèles sont comparés et que la contribution d'un modèle pour une variable est supérieure à l'autre, alors la variable importance doit aussi être supérieure à celle de l'autre modèle.

Il n'existe plus qu'une possibilité : les valeurs de Shapley. La formule pour les calculée est la suivante :

$$\phi_j(f, x_i) = \frac{1}{p!} \sum_R [f_{P_j^R \cup j}(x_i) - f_{P_j^R}(x_i)]$$

où p précise le nombre de variables présentes dans le modèle, R est l'ensemble des permutations possibles pour ces variables, P_j^R la liste des variables d'indice strictement inférieur à j de la permutation considérée et f le modèle dont il faut calculer les valeurs de Shapley pour l'observation i.

Le principe de cette méthode est applicable à tout type de modèle : il s'agit de construire un modèle sans la variable j pour chaque sous-modèle possible. Pour cela, toutes les permutations possibles sont balayées. La différence entre la prédiction obtenue pour chaque modèle et le même modèle avec la variable considérée est alors calculée. La moyenne de cette différence donne alors la variable importance selon Shapley. Autrement dit, une valeurs de Shapley $\phi_j(f, x_i)$ donne la contribution marginale moyenne de la variable j pour l'observation i.

Bien que très simple, cette formule est extrêmement coûteuse en temps de calcul dans le cas général, le nombre de modèles à entraîner augmentant de manière factorielle en fonction du nombre de variables. L'opération est par ailleurs à réitérer pour chaque prédiction.

5.2.3 TreeSHAP

En pratique, la nécessité de construire $n!$ modèles est rédhibitoire. Pour ne serait-ce que cinq variables, il faut entraîner pas moins de $5! = 120$ modèles, et ce autant de fois qu'il y a de prédictions à analyser.

Fort heureusement, il existe une solution, pour tirer parti de la structure des arbres de décision et réduire drastiquement le temps de calcul. Il n'est alors plus nécessaire que d'entraîner un seul modèle.

Lors de la construction des arbres de décision, les quantités *gain*, *weight* et *cover* peuvent être stockées pour chaque nœud et être mises à profit pour calculer à moindre coût une bonne estimation des valeurs de Shapley.

L'idée est de se baser sur un seul modèle et d'éviter ainsi d'avoir à entraîner un nombre exponentiel de modèles. Pour cela, ils réutilisent les poids associés aux feuilles et le cover. Le but est d'obtenir, à partir de cet unique modèle, les prédictions pour toutes les combinaisons de variables possibles.

La méthode est la suivante : pour une observation donnée, et pour la variable pour laquelle il faut calculer la valeur de Shapley, il suffit de parcourir les arbres de décision du modèle. À chaque nœud, si la décision implique l'une des variables du sous-ensemble, tout se passe comme un parcours standard. Cependant, si la décision du nœud se fait à partir d'une variable qui n'a pas été retenue par le sous-ensemble, il n'est pas possible de choisir quelle branche de l'arbre suivre. Dans ce cas, les deux branches sont explorées, et les poids en résultant sont pondérés par la couverture, c'est-à-dire par le nombre d'observations concernées par le test. Il ne reste plus qu'à calculer la différence entre sous-modèle sans et sous-modèle avec la variable et en faire la moyenne.

5.2.4 Interprétation et visualisation

Afin de bien interpréter les valeurs de Shapley pour une prédiction donnée, le mieux est de garder en tête la formulation mathématique sous-jacente : il s'agit d'un modèle linéaire additif local à la prédiction.

Les valeurs ne sont donc valables qu'à proximité des valeurs d'une observation donnée.

Explicabilité locale

Il existe plusieurs façons de représenter les valeurs de Shapley, généralement sous la forme d'un diagramme en barres comme il a été fait pour la variable importance classique dans la seconde section. Seulement, cette fois-ci, les données sont signées.

D'autres représentations sont possibles. Les quelques lignes suivantes montrent le graphique en cascade que fournit l'implémentation phare de SHAP sur le boston dataset :

```
import xgboost
import shap

X, y = shap.datasets.boston()
model = xgboost.XGBRegressor().fit(X, y)

explainer = shap.Explainer(model)
shap_values = explainer(X)

shap.plots.waterfall(shap_values[0])
```

Ce qui donne le graphique suivant :

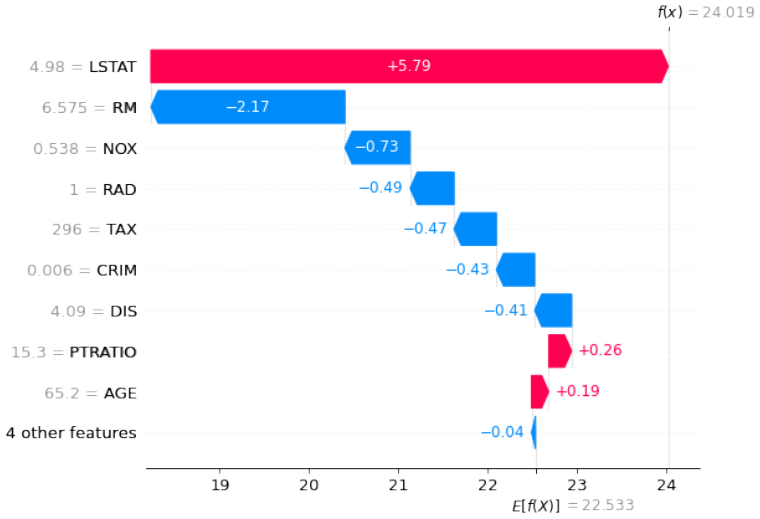

FIGURE 5.2 – Valeurs de Shapley locales

La lecture se fait à partir du bas du graphique, en partant de la valeur moyenne des valeurs à prédire. En remontant les variables en ordre inverse de leur valeur de Shapley, cette valeur de base est modifiée progressivement, jusqu'à arriver à la valeur prédite.

Le graphique confirme les intuitions fournies par le calcul du gain 5.1, tout en ajoutant l'information cruciale du signe de la modification.

Explicabilité globale

L'explicabilité globale se fait en agrégeant les valeurs de Shapley pour chaque observation du jeu d'entraînement.

Là encore, plusieurs représentations sont envisageables. Ce listing utilisant la librairie SHAP montre une visualisation compacte :

```
import xgboost
import shap
from matplotlib import pyplot as plt

X, y = shap.datasets.boston()
model = xgboost.XGBRegressor().fit(X, y)

explainer = shap.Explainer(model)
shap_values = explainer(X)

shap.plots.beeswarm(shap_values)
plt.show()
```

Il permet d'un coup d'œil d'observer le modèle globalement :

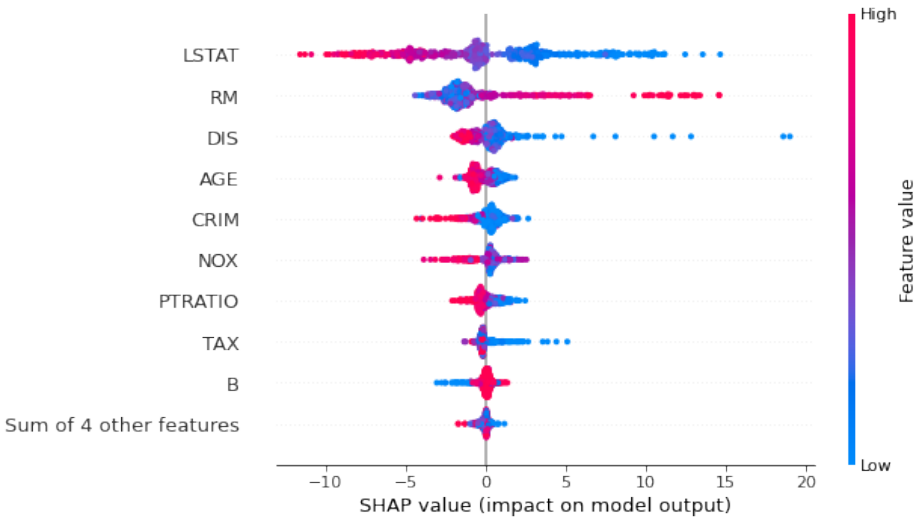

FIGURE 5.3 – Valeurs de Shapley globales

Dans le cas présent, cela montre que la variable LSTAT a un effet relativement équilibré sur la prédiction, l'augmentant globalement autant qu'elle l'a réduit. Son impact est par ailleurs assez variable en amplitude.

L'effet de RM, par contre, se concentre autour d'une valeur négative au sein d'un cluster assez compact.

Conclusion

Les méthodes de gradient boosting constituent actuellement l'état de l'art de l'apprentissage automatique lorsqu'il s'agit de données structurés. L'objectif de ce rapport est de décrire le cadre théorique dans lequel s'inscrivent ces méthodes et de présenter leur fonctionnement.

Sans être complètement exhaustif sur ce vaste sujet, nombre de points ont été traités, de la théorie de l'apprentissage à celle du Gradient Boosting, en passant par la présentation des algorithmes XGBoost et LightGBM, sans oublier les aspects optimisation des hyperparamètres et explicabilité.

Les points important sont listés ci-dessous :

— Les modèles de Gradient Boosting entraînent séquentiellement des apprenants faibles s'ajustant sur l'opposé du gradient de la fonction objectif calculée à chaque itération.
— Les fonctions objectifs pilotent la construction des arbres. Jouer avec ces dernières est un levier puissant pour construire des modèles performants.
— L'affinage des hyperparamètres est une étape cruciale, qui nécessite une compréhension fine de leur rôle et la construction méthodique de jeu d'entraînement, de validation et de test.
— Être capable d'expliquer un modèle est la clé non seulement de son adoption auprès du destinataire, mais aussi un moyen indispensable pour l'améliorer.
— Il ne faut pas oublier que l'enrichissement des données est essentiel. Sans données, il est impossible de construire un modèle. La méthode de Gradient Boosting ne fait qu'en révéler le potentiel.

Enfin, il est crucial de réaliser que le cadre du Gradient Boosting s'applique aux arbres de décision mais qu'il est possible de l'étendre à d'autres types de modèles sous-jacents. C'est au final une approche très générique et très puissante.

Bibliographie

[1] Trevor Hastie, Robert Tibshirani et Jerome H. Friedman. *The Elements of Statistical Learning : Data Mining, Inference, and Prediction, 2nd Edition.* 2009.

[2] Tianqi Chen et Carlos Guestrin. « XGBoost : A Scalable Tree Boosting System ». In : *CoRR* abs/1603.02754 (2016). arXiv : 1603.02754. URL : http://arxiv.org/abs/1603.02754.

[3] Guolin Ke et al. « LightGBM : A Highly Efficient Gradient Boosting Decision Tree ». In : *Advances in Neural Information Processing Systems 30 (NIP 2017)*. 2017. URL : https://www.microsoft.com/en-us/research/publication/lightgbm-a-highly-efficient-gradient-boosting-decision-tree/.

[4] Jerome H. Friedman. « Greedy Function Approximation : A Gradient Boosting Machine ». In : *The Annals of Statistics* 29.5 (2001), p. 1189-1232. ISSN : 00905364. URL : http://www.jstor.org/stable/2699986.

[5] L. S. Shapley. « 17. A Value for n-Person Games ». In : *Contributions to the Theory of Games (AM-28), Volume II*. Sous la dir. d'Harold William Kuhn et Albert William Tucker. Princeton University Press, 2016, p. 307-318. URL : https://doi.org/10.1515/9781400881970-018.

[6] Christoph Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2019.

[7] Scott Lundberg et Su-In Lee. « A Unified Approach to Interpreting Model Predictions ». In : (2017). arXiv : 1705.07874 [cs.AI].

[8] Scott M. Lundberg, Gabriel G. Erion et Su-In Lee. *Consistent Individualized Feature Attribution for Tree Ensembles*. 2019. arXiv : 1802.03888 [cs.LG].

[9] Terence Parr et Jeremy Howard. « Gradient boosting performs gradient descent ». In : (2018). URL : https://explained.ai/gradient-boosting/descent.html.

Table des figures