ECOLE SUPERIEURE DE LA STATISTIQUE ET DE L ANALYSE DE L INFORMATION

RAPPORT DE RECHERCHE

Une exploration approfondie de XGBoost

Réalisé par :
Yarbane Cheikh Mohamed Vadel

Avril 2025

Table des matières

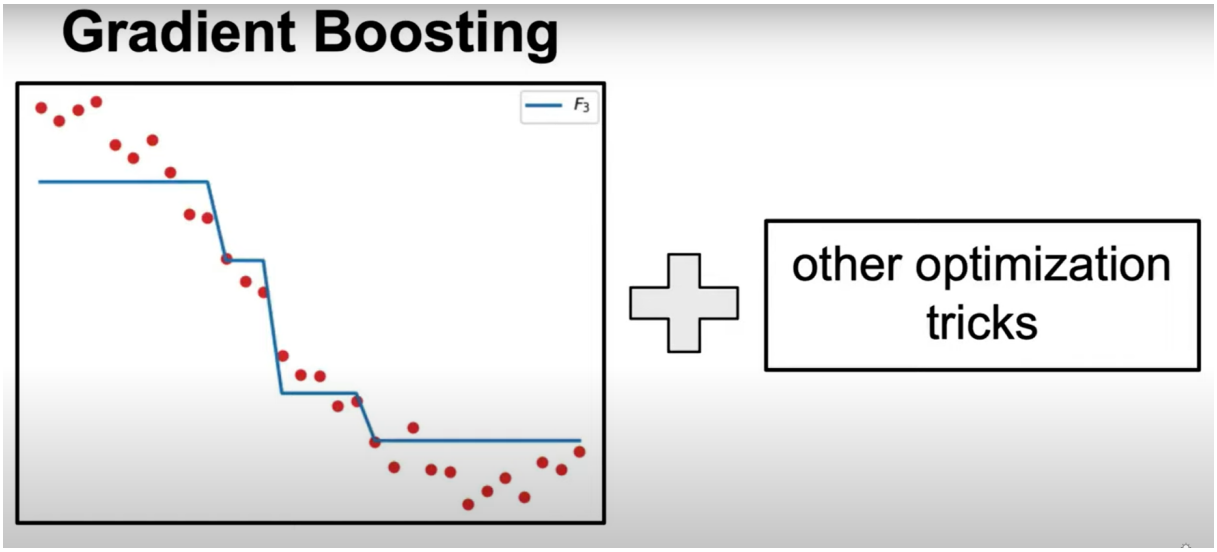
1 Introduction

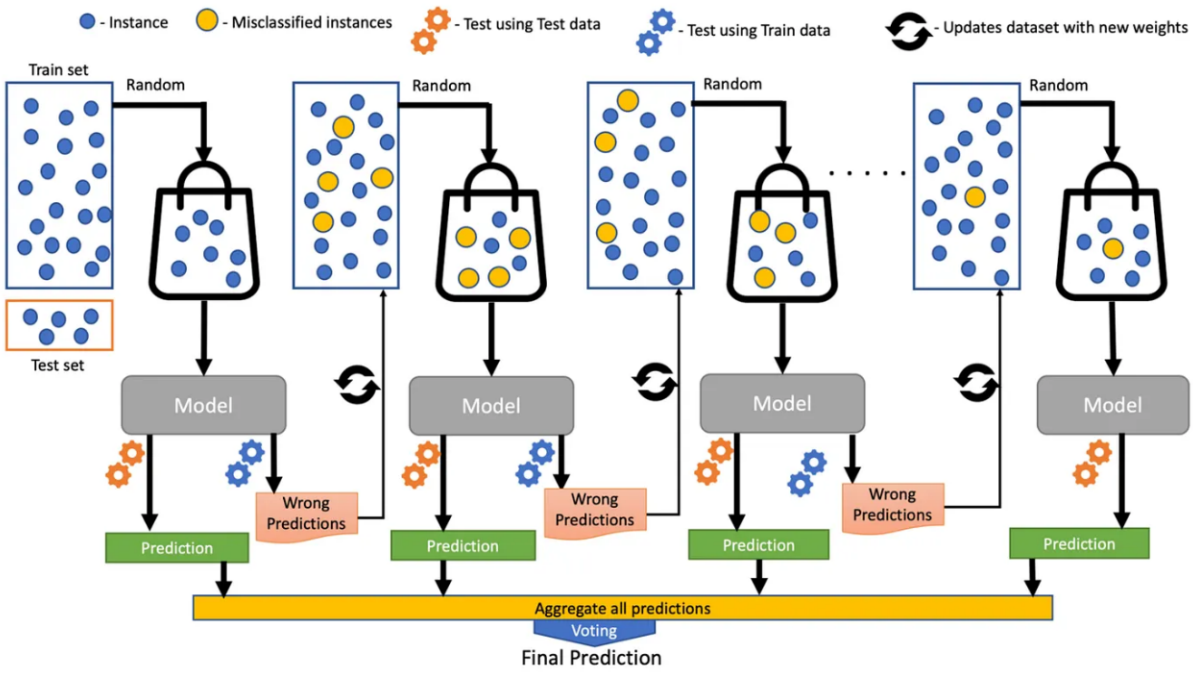
XGBoost, ou *Extreme Gradient Boosting*, est un algorithme d'apprentissage automatique à la pointe de la technologie, largement reconnu pour ses performances exceptionnelles dans les tâches de classification et de régression. Il s'agit d'une version optimisée et étendue du Gradient Boosting, une méthode qui repose sur la combinaison séquentielle de modèles faibles — généralement des arbres de décision — pour construire un modèle prédictif robuste.

Contrairement au Gradient Boosting classique, XGBoost introduit de nombreuses améliorations algorithmiques et systémiques : traitement parallèle, régularisation explicite pour limiter le surapprentissage, gestion efficace de la mémoire, et prise en charge native des valeurs manquantes. Ces ajouts font de XGBoost un outil à la fois rapide, précis et capable de gérer des ensembles de données complexes à grande échelle.

Le fonctionnement de XGBoost repose sur un processus itératif où chaque nouvel arbre est entraîné pour corriger les erreurs des arbres précédents. Ce raffinement progressif est guidé par la descente de gradient, qui permet d'ajuster les paramètres du modèle afin de minimiser la fonction de perte. Grâce à l'intégration de techniques de régularisation, XGBoost parvient également à améliorer la capacité de généralisation du modèle sur des données non vues.

Utilisé dans des domaines variés comme la finance, la santé ou le commerce électronique, XGBoost s'est imposé comme un algorithme de référence, tant pour sa précision que pour sa flexibilité. De plus, son caractère relativement interprétable permet aux praticiens de mieux comprendre les variables qui influencent les prédictions, ce qui le rend particulièrement adapté aux contextes où la transparence est cruciale.





2 Fondements du Gradient Boosting

Dans cette partie, nous allons présenter les idées principales qui sous-tendent le fonctionnement du gradient boosting. Cette méthode d'apprentissage supervisé repose sur l'idée d'assembler plusieurs modèles faibles, généralement des arbres de décision peu profonds, pour créer un modèle fort et performant. Le gradient boosting construit ces modèles de manière séquentielle, chaque nouveau modèle cherchant à corriger les erreurs commises par les précédents.

2.1 les etapes du gradient boosting

"Les étapes constitutives du gradient boosting suivent un processus algorithmique précis, que nous exposons ci-dessous.

2.1.1 premiere etape

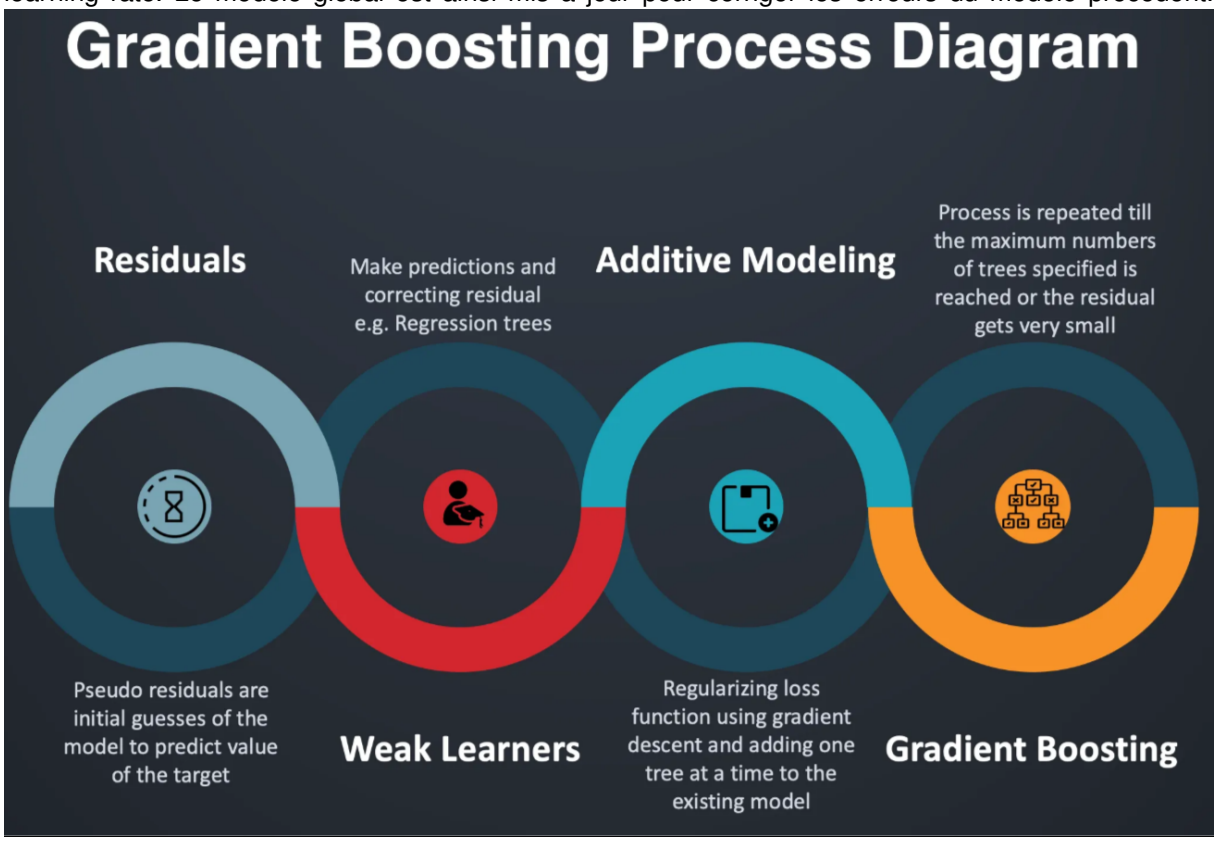
La première étape du gradient boosting consiste à initialiser le modèle avec une prédiction constante, souvent choisie comme la moyenne des valeurs cibles pour les problèmes de régression, ou la proportion des classes dans le cas d'une classification

2.1.2 Deuxième étape : Calcul des résidus

Dans cette étape, les résidus sont calculés en soustrayant les prédictions actuelles du modèle des valeurs réelles. Ces résidus représentent l'erreur du modèle et servent de base pour la correction des erreurs dans les itérations suivantes. Un nouveau modèle faible, généralement un arbre de décision, est ensuite entraîné pour prédire ces résidus.

2.1.3 Troisième étape : Mise à jour des prédictions

Une fois que le modèle a appris à prédire les résidus, ces prédictions sont ajoutées aux prédictions actuelles du modèle. L'ajustement est effectué en pondérant les nouvelles prédictions par le learning rate. Le modèle global est ainsi mis à jour pour corriger les erreurs du modèle précédent.



3 Du concept à l'algorithme

Entrée :

— Un ensemble d'entraînement $\{(x_i, y_i)\}_{i=1}^{n}$

— Une fonction de perte différentiable $L(y, F(x))$

— Un nombre d'itérations M

Étapes de l'algorithme :

1. **Initialisation du modèle** avec une constante :

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

   Cette étape cherche une constante qui minimise la perte sur toutes les données. Par exemple, si la perte est l'erreur quadratique, cela revient à la moyenne des y_i.

2. **Pour** $m = 1$ **à** M :

   (a) **Calcul des pseudo-résidus :**

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{pour } i = 1, \ldots, n$$

   Ces résidus indiquent dans quelle direction et quelle amplitude ajuster le modèle.

   (b) **Entraîner un apprenant faible** $h_m(x)$ **sur le jeu de données** $\{(x_i, r_{im})\}_{i=1}^{n}$

   (c) **Trouver un coefficient multiplicateur optimal** γ_m :

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right)$$

   Cette étape détermine de combien on doit corriger le modèle avec le nouvel apprenant.

   (d) **Mettre à jour le modèle** :

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

   Le modèle est amélioré itérativement en ajoutant des corrections pondérées.

3. **Retourner le modèle final** :

$$F_M(x)$$

4 Optimisations apportées par XGBoost au Gradient Boosting

XGBoost (eXtreme Gradient Boosting) est une amélioration du Gradient Boosting classique, intégrant plusieurs astuces d'optimisation visant à augmenter la performance, la précision et la vitesse d'exécution. Voici les principales optimisations :

1. **Régularisation (L1 et L2)**
   Contrairement au Gradient Boosting traditionnel, XGBoost introduit une régularisation explicite sur la complexité du modèle, afin de prévenir le surapprentissage. La fonction de coût globale à minimiser devient :

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

   avec le terme de régularisation :

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

   — $l(y_i, \hat{y}_i)$: fonction de perte (par exemple, erreur quadratique, log loss, etc.)
   — f_t : arbre de décision à l'étape t
   — T : nombre total de feuilles dans l'arbre f_t
   — w_j : score (poids) associé à la feuille j
   — γ : pénalité appliquée par feuille ajoutée (contrôle la complexité structurelle)
   — λ : coefficient de régularisation L2 (contrôle la magnitude des poids)

   Cette régularisation permet de réduire le risque de surapprentissage en pénalisant les modèles trop complexes.

2. **Réduction de l'apprentissage (Shrinkage)**
   Chaque nouvel arbre est pondéré par un facteur η (taux d'apprentissage) :

$$\hat{y}^{(t)} = \hat{y}^{(t-1)} + \eta f_t(x)$$

   Cela ralentit l'apprentissage, rendant le modèle plus robuste.

3. **Sous-échantillonnage des colonnes (features)**
   À la manière des forêts aléatoires, XGBoost sélectionne aléatoirement un sous-ensemble de variables pour construire chaque arbre, réduisant le surapprentissage et accélérant l'entraînement.

4. **Algorithmes gloutons exacts et approximatifs**
   Deux méthodes sont disponibles :
   — *Exact greedy* : calcule la meilleure séparation exacte.
   — *Approximate greedy* : utilise des histogrammes ou sketches quantiles pour des séparations quasi-optimales.
   Cela permet d'équilibrer la précision et la vitesse.

5. **Gestion efficace de la parcimonie**
   XGBoost gère efficacement les données creuses ou avec valeurs manquantes en utilisant des directions par défaut dans les nœuds d'arbre.

6. **Sketch quantile pondéré**
   Permet de traiter efficacement les données pondérées lors de la séparation approximative, en conservant une distribution fidèle des valeurs.

7. **Structure mémoire optimisée (DMatrix)**
   L'accès mémoire est optimisé avec des structures comme `DMatrix`, ce qui accélère les calculs sur de grands ensembles de données.
8. **Parallélisation de la construction des arbres**
   Le processus de recherche de séparation est parallélisé entre les colonnes et les blocs de données, exploitant les processeurs multicœurs pour un apprentissage plus rapide.

5 Hyperparamètres de XGBoost

Les performances de XGBoost dépendent fortement du choix de ses hyperparamètres. Ces derniers sont fixés avant l'entraînement du modèle et contrôlent son comportement, sa complexité, sa régularisation et le processus d'optimisation. On distingue plusieurs catégories d'hyperparamètres :

1. Hyperparamètres de structure des arbres

— `max_depth` : profondeur maximale de chaque arbre. Une grande profondeur augmente la capacité d'apprentissage mais accroît le risque de surapprentissage.
— `min_child_weight` : poids minimal (somme des hessiennes) requis dans un nœud fils. Un seuil élevé conduit à des arbres plus simples.
— `gamma` : réduction minimale de la perte requise pour effectuer une division. Plus cette valeur est élevée, plus le modèle est conservateur.
— `subsample` : fraction des exemples d'apprentissage utilisée pour construire chaque arbre. Valeur typique : entre 0.5 et 1.
— `colsample_bytree` : fraction des caractéristiques (features) utilisée pour chaque arbre.
— `colsample_bylevel` : fraction de variables utilisée à chaque niveau de l'arbre.
— `colsample_bynode` : fraction de variables utilisée à chaque nœud.

2. Hyperparamètres de régularisation

— `lambda` : terme de régularisation L2 (ridge) appliqué aux scores des feuilles.
— `alpha` : terme de régularisation L1 (lasso).

3. Hyperparamètres du processus d'apprentissage

— `learning_rate` (`eta`) : taux d'apprentissage. Diminue l'impact de chaque nouvel arbre. Valeur faible = apprentissage plus lent mais plus robuste.
— `n_estimators` : nombre d'arbres (ou itérations de boosting).
— `early_stopping_rounds` : nombre d'itérations sans amélioration sur un jeu de validation avant arrêt automatique.

4. Hyperparamètres liés au type de booster

— `booster` : type de booster à utiliser : `gbtree` (arbres), `gblinear` (modèle linéaire), ou `dart` (dropout d'arbres).
— `tree_method` : méthode de construction des arbres : `auto`, `exact`, `approx`, `hist`, `gpu_hist`.

5. Hyperparamètres spécifiques à DART (Dropout)

- `rate_drop` : probabilité de dropout pour un arbre donné.
- `skip_drop` : probabilité de ne pas effectuer de dropout à une itération.
- `sample_type` : type d'échantillonnage : `uniform` ou `weighted`.
- `normalize_type` : méthode de normalisation : `tree` ou `forest`.

6. Autres hyperparamètres utiles

- `objective` : fonction objectif (classification, régression, etc.) : `binary:logistic`, `reg:squarederror`, etc.
- `eval_metric` : métrique(s) d'évaluation : `rmse`, `auc`, `error`, etc.
- `seed` : graine aléatoire pour la reproductibilité.
- `verbosity` : niveau d'affichage des logs : `0` (silencieux) à `3` (debug).

Un bon réglage de ces hyperparamètres, souvent via validation croisée ou recherche par grille, est essentiel pour obtenir un modèle performant et généralisable.

6 Méthodes d'estimation des hyperparamètres

L'estimation des hyperparamètres est une étape cruciale dans la construction d'un modèle performant avec XGBoost. Plusieurs techniques permettent d'explorer efficacement l'espace des hyperparamètres. On distingue notamment :

1. Recherche par grille (*Grid Search*)

Cette méthode consiste à définir une grille de valeurs possibles pour chaque hyperparamètre, puis à entraîner un modèle pour chaque combinaison. Bien qu'exhaustive, cette approche est coûteuse en temps de calcul, surtout lorsque le nombre de paramètres est élevé.

2. Recherche aléatoire (*Random Search*)

Contrairement à la recherche par grille, cette méthode explore l'espace des hyperparamètres de manière aléatoire. Elle est souvent plus efficace que la grille lorsque certains hyperparamètres ont peu d'impact sur la performance.

3. Optimisation bayésienne

Elle repose sur une modélisation probabiliste de la fonction objectif. Des bibliothèques comme `Hyperopt`, `Optuna` ou `Scikit-Optimize` (Skopt) utilisent cette approche pour proposer des combinaisons prometteuses d'hyperparamètres tout en minimisant le nombre d'essais.

4. Validation croisée

Toutes les méthodes précédentes utilisent généralement la validation croisée pour évaluer la qualité des combinaisons d'hyperparamètres. La technique la plus courante est la validation croisée *k-fold*, qui permet d'obtenir une estimation robuste de la performance.

5. Algorithmes de tuning automatisés

Des outils comme `Optuna`, `Ray Tune` ou `AutoML` automatisent totalement le processus de tuning, en intégrant des stratégies intelligentes d'exploration et d'évaluation.

Le choix de la méthode dépend du budget computationnel, de la taille du dataset, et de la criticité du modèle. Dans la pratique, la recherche aléatoire ou bayésienne combinée à la validation croisée donne un bon compromis entre performance et coût.

7 Méthodes d'interprétation de XGBoost

Bien que XGBoost soit un modèle de type *boîte noire*, il existe plusieurs techniques pour interpréter ses prédictions, que ce soit globalement (comportement général du modèle) ou localement (explication d'une prédiction individuelle). Ces méthodes sont essentielles dans des contextes où l'explicabilité est requise (finance, santé, etc.).

1. Importance des variables (*Feature Importance*)

XGBoost fournit des mesures intégrées d'importance des variables :

— **Gain** : amélioration moyenne de la fonction de perte lorsqu'une variable est utilisée dans une division.

— **Cover** : mesure la couverture, c'est-à-dire le nombre d'échantillons impactés par une feature.

— **Frequency** : nombre de fois qu'une variable est utilisée dans les arbres.

Ces indicateurs sont disponibles via la fonction `plot_importance()` de la bibliothèque `xgboost`.

2. SHAP values (SHapley Additive exPlanations)

Les valeurs SHAP sont basées sur la théorie des jeux coopératifs (valeurs de Shapley). Elles permettent :

— d'attribuer à chaque feature sa contribution exacte à la prédiction,

— d'interpréter chaque prédiction individuellement (interprétation locale),

— d'analyser l'importance globale des variables.

Cette méthode est très rigoureuse mathématiquement et bien adaptée aux modèles en arbre comme XGBoost.

3. Courbes de dépendance partielle (PDP)

Les *Partial Dependence Plots* représentent la relation entre une ou deux variables et la prédiction moyenne du modèle. Ils montrent l'effet marginal d'une variable sur la sortie du modèle, en gardant les autres constantes.

4. Courbes ICE (Individual Conditional Expectation)

Les courbes ICE sont une version individualisée des PDP. Elles permettent d'observer, pour chaque individu, comment la prédiction évolue lorsque l'on fait varier une variable spécifique. Elles sont utiles pour détecter des effets non linéaires ou des interactions spécifiques.

5. LIME (Local Interpretable Model-agnostic Explanations)

LIME est une méthode agnostique qui approxime localement le comportement du modèle par un modèle simple et interprétable (souvent linéaire). Elle permet d'expliquer une prédiction individuelle, indépendamment du type de modèle utilisé (même les réseaux de neurones).

6. Résumé comparatif

Méthode	**Locale/Globale**	**Précision**	**Utilisation recommandée**
Importance des variables	Globale	Moyenne	Vue d'ensemble rapide
SHAP	Les deux	Très haute	Analyse approfondie
PDP	Globale	Moyenne	Intuition globale
ICE	Locale	Haute	Étude individuelle
LIME	Locale	Moyenne	Explication agnostique

L'interprétation d'un modèle comme XGBoost est une étape clé pour renforcer la confiance des utilisateurs, détecter des biais éventuels, et respecter les exigences réglementaires.

Références

[1] Wikipedia contributors. *Gradient Boosting*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Gradient_boosting

[2] StatQuest with Josh Starmer. *YouTube Channel - StatQuest*. https://www.youtube.com/user/joshstarmer

[3] Rohan Harode, Shubham Malik, and Akash Singh Kunwar. *XGBoost : A Deep Dive into Boosting*. SFU Professional Computer Science, Medium, February 4, 2020. https://medium.com/sfu-cspmp/xgboost-a-deep-dive-into-boosting-f06c9c41349

[4] XGBoost Developers. *XGBoost Documentation*. https://xgboost.readthedocs.io/en/stable/