Projet de semestre

Gradient Boosting et Boosted Trees : Introduction et Applications Expérimentales

Étudiant : Ayemen Assadi

Abstract:

Gradient boosting est une technique d'apprentissage automatique particulièrement efficace lorsqu'elle est combinée à des arbres de décision. Dans ce rapport, nous présentons le principe général de cette approche, son implémentation avec des arbres de décision, ainsi que plusieurs algorithmes représentatifs (tels que XGBoost et AdaBoost). Enfin, nous évaluons ces algorithmes sur des données réelles, et nous discutons leurs performances respectives.

Université de Genève
Printemps 2026

Chapter 1

Decision Trees - Random forests

Arbres de décision

Dans cette section, nous présentons les arbres de décision (Decision trees), et nous discutons leurs principes et propriétés les plus importantes.

1.1.1 Introduction et définitions

Tout comme un être humain qui considère plusieurs options avant de se décider, les arbres de décision modélisent chaque prédiction comme une ligne de choix qu'on appelle une branche (d'où le nom arbre) parmi plusieurs, chacune dépend des facteurs définis à l'avance. Les arbres de décision appartiennent typiquement à l'apprentissage supervisé : le modèle apprend sa structure à partir d'exemples étiquetés dont la réponse correcte est connue. Ils sont très utilisés car ils transforment des problèmes complexes en une succession d'étapes simples, ce qui rend leurs résultats aisément interprétables.

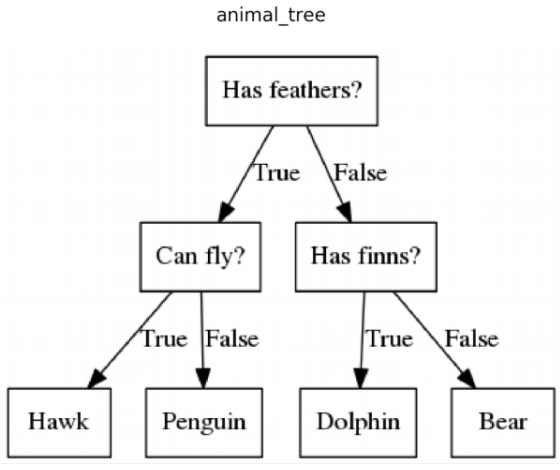

Figure 1.1: Arbres de décision typique

Donc, à partir de l'illustration on voit clairement qu'on a commencé d'une question initiale unique — **nœud racine** — qui représente la totalité des données. À partir de ce nœud, l'arbre se ramifie en différentes directions selon les attributs (ou caractéristiques) des données en réponse à la question correspondant au noeud parent.

Les éléments principaux d'un arbre de décision sont les suivants:

- **Noeud racine :** point de départ représentant la totalité des données.
- **Branches :** lignes reliant les nœuds, montrant le cheminement d'une décision à une autre.
- **Noeuds internes :** points où les décisions sont prises en fonction des caractéristiques des données.
- **Noeuds feuilles :** points terminaux de l'arbre où la décision finale est rendue.

Les arbres de décision sont utilisés principalement pour résoudre deux grands types de problèmes:

- **Classification :** ils servent à prédire des résultats catégoriels (par exemple : « courriel spam » ou « non spam »). Ces arbres partitionnent les données selon leurs attributs afin de les ranger dans des classes prédéfinies qui se situent dans les noeuds feuilles.
- **Regression :** ils servent à prédire des résultats continus (par exemple : le prix d'une maison). Contrairement aux arbres de classification, ils ne produisent pas de catégories mais des valeurs numériques estimées à partir des caractéristiques d'entrée.

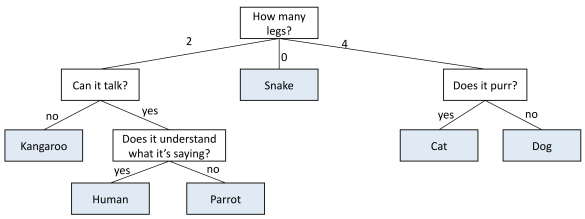

Figure 1.2: Arbre de classification

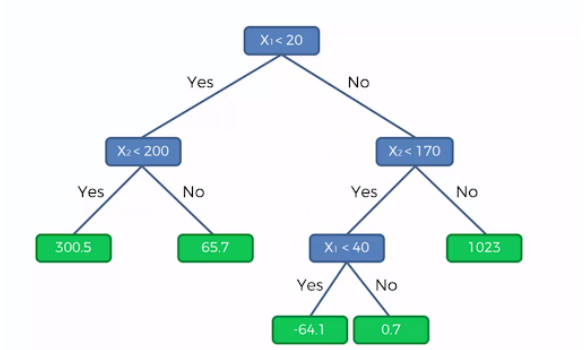

Figure 1.3: Arbre de regression

La tâche principale des arbres de décision consiste à apprendre comment partitionner au mieux l'ensemble des données afin de classer correctement les observations. Dans le cas de la régression on peut prédire une valeur numérique avec la meilleure précision possible. Une notion centrale mérite d'être définie : **le critère de division** (splitting criterion).

Remarque : Les noeuds d'un arbre de décision pour la régression sont construits en prenant la moyenne de chaque partition générée par la dernière division.

Dans le domaine du Machine learning, on distingue généralement deux critères de séparation :

1.Gini impurity:

Il s'agit d'un indice qui mesure la pertinence d'un attribut pour partitionner les données en groupes homogènes. Mathématiquement, il correspond à la probabilité qu'un élément tiré au hasard soit incorrectement classé si on lui attribue l'étiquette majoritaire de son groupe. L'algorithme CART utilise cette mesure pour évaluer différentes divisions possibles et retient celle qui produit la séparation la plus pure.

Par exemple, demander à des personnes si elles sont fatiguées permet de former deux groupes : d'un côté, celles susceptibles de boire du café, de l'autre, celles qui ne le sont pas.

Soit p_i la probabilité qu'un élément soit dans la classe i et D le dataset,

alors:

$$Gini(D) = \sum_i p_i(1 - p_i) = \sum_i p_i - \sum_i p_i^2 = 1 - \sum_i p_i^2$$

L'arbre de classification est construit de manière à obtenir des feuilles contenant des données dont les labels sont identiques. On doit donc mesurer l'hétérogénéité des labels dans les noeuds pour gérer leur division.
Exemple de calcul:

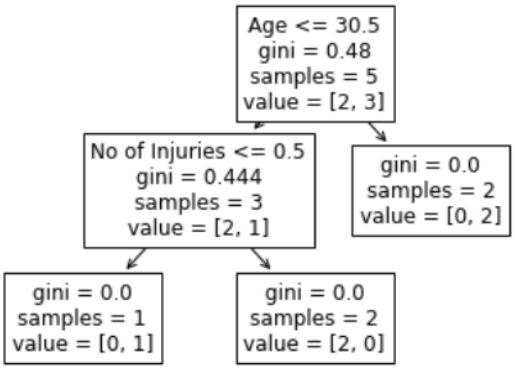

Figure 1.4: Exemple de calcul de l'indice de Gini

Comme on peut voir, pour chaque noeud interne on a une condition tout en haut de la case. Cette condition divise le dataset à l'étape courante, mais les partitions ne sont pas parfaites, c'est-à-dire qu'on trouve des groupes qui ne sont pas complètement purs. L'indice de Gini quantifie cette impureté.
Par exemple, pour le noeud racine : on a $p_1 = \frac{groupe1}{samples} = \frac{2}{5}$ et $p_2 = \frac{groupe2}{samples} = \frac{3}{5}$.
Donc : $Gini(D) = 1 - \frac{4}{25} - \frac{9}{25} = 0.48$ ce qui est exactement indiqué comme indice Gini pour cette division.

Le deuxième indice qu'on va voir est :

2.Entropy:
Il s'agit d'un indice très utilisé dans de nombreux domaines en Physique, Théorie de l'information,etc.
L'entropie quantifie l'incertitude présente dans un ensemble de données. Des algorithmes tels qu'ID3 et C4.5 exploitent l'entropie pour calculer le gain d'information, c'est-à-dire la diminution de l'incertitude apportée par une division. L'arbre choisit alors la division qui maximise cette réduction d'incertitude, obtenant ainsi la séparation la plus nette entre les classes. Dans l'analogie du café, demander s'il est matin ou après-midi réduit l'incertitude car cela répartit les personnes en groupes plus distincts, facilitant ainsi la décision.
Soit p_i la probabilité qu'un élément soit dans la classe i et D le dataset,
alors:

$$\boldsymbol{H(D) = -\sum_i p_i \ln(p_i)}$$

Exemple de calcul: On reste sur le même exemple, cette fois-ci on choisit le noeud "No of injuries <= 0.5". De cette division, on détermine $p_1 = \frac{group1}{sample} = \frac{2}{3}$ et $p_2 = \frac{group2}{sample} = \frac{1}{3}$.
Donc : $H(D) = -\frac{2}{3}\ln(\frac{2}{3}) - \frac{1}{3}\ln(\frac{1}{3}) = 0.636$

1.1.2 Algorithmes d'apprentissage:

Dans la section précédente, on a introduit les concepts nécessaires pour comprendre comment les arbres de décision fonctionnent et comment leur approche s'avère très utile en décomposant un problème ou une question complexe en une hiérarchie de simple questions (divisions). Si on remarque bien, on constate que la question qui se pose est comment réaliser les divisions les plus optimales. Malheureusement, il n'y a pas un algorithme qui donne les meilleures divisions, sauf si on veut tester toutes les combinaisons possibles (ce qui est totalement impossible dans un temps qui est raisonnable). Pourtant, il existe des algorithmes qui donnent des divisions acceptables.
Dans cette section, on verra deux algorithmes courants pour trouver de bonnes divisions.
Une notion importante à comprendre avant de discuter ces algorithmes est :

Pruning (Elagage):
Un arbre de décision trop profond risque de mémoriser le bruit présent dans les données d'entraînement, ce qui nuit à sa capacité de généralisation. Pour limiter ce phénomène de sur-apprentissage, on utilise une technique appelée élagage (pruning). Celle-ci consiste à supprimer certaines branches (ou sous-arbres) qui apportent peu de gain prédictif. On distingue deux approches : l'élagage précoce (pre-pruning), qui arrête la construction de l'arbre avant qu'il ne devienne trop complexe, et l'élagage postérieur (post-pruning), qui élague les nœuds après la construction complète.

Avec cette notion en place, on peut voir le premier algorithme :

C4.5:
C4.5 (Iterative Dichotomiser 4.5) est un algorithme d'apprentissage supervisé qui construit des arbres de décision pour des tâches de classification. Il adopte une stratégie gloutonne (greedy) et descendante (du haut vers le bas) : à chaque étape, il sélectionne l'attribut qui maximise le taux de gain d'information, une mesure calculée à partir de l'entropie. Cet algorithme gère aussi bien les variables catégorielles que continues, ce qui le rend plus flexible.

Fonctionnement de l'algorithme :
C4.5 construit l'arbre de décision en sélectionnant le meilleur attribut de segmentation à l'aide du Gain Ratio (taux de gain). Cette mesure favorise des divisions équilibrées et réduit le biais envers les attributs possédant de nombreuses valeurs distinctes.
Etape 1: Initialiser le noeud racine.
L'intégralité de l'ensemble de données est placée au noeud racine.
Etape 2: Calculer le gain d'information (Information mutuelle).
On calcule le gain d'information pour chaque attribut afin de mesurer la réduction de l'entropie obtenue après la division.

$$\text{Gain d'information} = H(D) - \sum_{v=1}^{V} \frac{|D_v|}{|D|} H(D)$$

v : L'ensemble de divisions possibles, D_v : l'ensemble des données après une division , $|D_v|$: le cardinal de D_v.
Etape 3: Calculer l'information du split (Split information).
L'information de split mesure la manière dont l'ensemble des données est réparti entre les différentes branches après une division. Elle est donnée par la formule :

$$\text{Split Information} = -\sum_{v=1}^{V} \frac{|D_v|}{|D|} \ln(\frac{|D_v|}{|D|})$$

Etape 4: Calculer le taux de gain.

Le taux de gain normalise le gain d'information pour éviter de favoriser les attributs avec de nombreuses valeurs distinctes :

$$GainRatio = \frac{Gain\,d'information}{Split\,d'information}$$

Etape 5: Sélectionner le meilleur attribut et diviser.
L'attribut ayant le taux de gain le plus élevé est sélectionné, puis l'ensemble des données est partitionné en sous-ensembles selon ses valeurs.
Etape 6: Répéter récursivement.
Le même processus se répète récursivement sur chaque sous-ensemble jusqu'à ce que les nœuds deviennent purs (c'est-à-dire qu'ils ne contiennent qu'une seule classe) ou qu'il n'y ait plus d'attributs disponibles.

Remarques:

- L'algorithme C4.5 gère les variables continues en les discrétisant; on choisit un ou plusieurs seuils (thresholds) qui donnent des divisions, ensuite on choisit le seuil maximisant le taux de gain. Comment choisir ces seuils est une question d'implémentation (il n'y a pas une méthode universelle).
- Pour les variables catégorielles, on peut les encoder. Aussi, l'encodage des variables catégorielles est une question d'implémentation car cela dépendra du contexte de l'implémentation de l'algorithme.

Limites de l'algorithme C4.5:

- Même avec la régularisation (pruning), C4.5 peut encore souffrir de sur-apprentissage (overfitting) sur des ensembles de données bruités.
- L'algorithme peut devenir coûteux en calcul lorsqu'il est confronté à de très grands ensembles de données ou à un grand nombre d'attributs
- Les arbres de décision générés par C4.5 peuvent devenir complexes et difficiles à interpréter lorsqu'ils sont trop profonds.

Le deuxième algorithme d'apprentissage des arbres décisionnels :

CART:
CART est un algorithme très répandu qui se différencie des autres méthodes d'arbres de décision par une particularité : il produit systématiquement des divisions binaires (oui/non) pour chaque attribut. À chaque nœud, il sélectionne la coupure qui sépare au mieux les données.

- CART peut traiter aussi bien des problèmes de classification que de régression.
- Pour la classification, il utilise l'indice de Gini ; pour la régression, il utilise la réduction de variance.
- Il génère exclusivement des arbres binaires : chaque nœud interne donne lieu à exactement deux branches (deux nœuds enfants)

Fonctionnement de l'algorithme:

Etape 1: Initialiser le noeud racine.
Le processus commence par le placement de l'intégralité de l'ensemble de données au nœud racine et calculer son indice Gini (L'indice Gini initial). Ce nœud contient tous les échantillons d'apprentissage avant toute division.

Etape 2: Calculer l'indice (critère) de division.

Pour la classification, on calcule l'indice Gini des deux groupes qui émergent, l'indice **Gini pondéré** et le Gain porté par cette division

$$Gini_{pondere} = \frac{N_g}{N_p} Gini_g + \frac{N_d}{N_p} Gini_d$$

N_g : nombre de données dans l'enfant gauche , N_d : nombre de données dans l'enfant droit ,
N_p : nombre d'enfant dans le noeud parent , $Gini_d$: indice Gini de l'enfant droit ,
$Gini_g$: indice Gini de l'enfant gauche

$$Gain = Gini_{parent} - Gini_{pondere}$$

Pour la régression, nous calculons la variance des deux groupes qui émergent de la division.

Etape 3: Choisir la meilleure division.

L'algorithme examine différentes caractéristiques (attributs) et points de division possibles pour déterminer dans quelle mesure ils permettent de partitionner les données en groupes plus homogènes.
Pour la classification, on choisit l'attribut qui a mené au meilleur gain. Pour la régression, le meilleur attribut est celui qui mène à une division avec des groupes de variances minimales.

Etape 4: Créer des branches binaires.

CART crée systématiquement des divisions binaires, ce qui signifie que chaque nœud est divisé en exactement deux nœuds enfants (gauche et droite). Cela simplifie la structure de l'arbre.

Etape 5: Répéter récursivement.

Le même processus se répète pour chaque sous-ensemble, en divisant les données jusqu'à ce que les critères d'arrêt soient satisfaits, comme l'obtention de nœuds purs (contenant une seule classe) ou l'atteinte d'un nombre minimum d'échantillons.

Limites de CART :

- CART peut souffrir de sur-apprentissage (overfitting) si l'arbre devient trop profond.
- L'algorithme peut être sensible à de petites variations dans l'ensemble de données, ce qui peut conduire à des structures d'arbre différentes.
- Les arbres de grande taille peuvent devenir coûteux en calcul et plus difficiles à interpréter.

1.1.3 Avantages et inconvénients :

Comme le fameux statisticien britannique **George E.P.Box** a dit : "**All models are wrong; but some of them are useful**" (Tous les modèles sont faux, mais certains sont utiles). Les abres de décision sont aussi des modèles qui présentent bien des inconvénients mais ils ont prouvé leur utilité.

Avantages:

- En finance et en assurance, les arbres de décision sont utilisés pour évaluer les risques, qu'il s'agisse de défauts de remboursement, de sinistres ou d'autres pertes. En explorant les différentes branches issues des données clients — comme l'historique de crédit, le niveau de revenu ou les profils de sinistres — ils permettent aux actuaires, souscripteurs et analystes financiers d'affiner leurs estimations de risque.

- Dans le domaine du Business, les modèles de machine learning basés sur des arbres de décision sont également précieux pour la prévision : croissance des ventes, évolution des prix, désabonnement des clients, ou encore demande et niveaux de stock dans les chaînes d'approvisionnement.
- Dans le secteur de la santé, les modèles de ML s'appuient fréquemment sur les arbres de décision pour analyser les données patient. Ainsi, un modèle peut croiser des symptômes, des résultats d'examens et des antécédents familiaux afin de rassembler les informations nécessaires à l'orientation des diagnostics et des traitements.

Inconvénients:

- Sur-apprentissage : les arbres de décision ont tendance à devenir trop complexes en apprenant des détails spécifiques aux données d'entraînement plutôt que des schémas généralisables. Il en résulte un modèle performant pendant l'apprentissage, mais qui se révèle inefficace face à des données nouvelles jamais rencontrées.
- Sensibilité au bruit (noisy data) : Un arbre de décision peut également être trompé par des variations aléatoires ou sans pertinence dans un jeu de données, qui ne correspondent à aucune tendance réelle. Même un faible niveau de bruit peut induire des divisions erronées, rendant les prédictions instables.
- Un arbre unique prend ses décisions de façon indépendante, ce qui le rend vulnérable aux erreurs et au sur-apprentissage. Les méthodes d'ensemble, quant à elles, combinent les prédictions de plusieurs arbres. Cette approche collective offre généralement des résultats plus précis, plus fiables et plus cohérents.

Random Forests (Forêts aléatoires)

Dans cette section, nous présentons les Forêts aléatoires (Random Forests), et nous discutons leurs principes et propriétés les plus importantes.

1.2.1 Introduction et Définitions:

Les arbres de décision imposent un problème structurel. Un arbre trop profond, avec de nombreuses feuilles, souffre de **overfitting** : chaque prédiction s'appuie sur un nombre très restreint d'exemples historiques (ceux de sa feuille). À l'inverse, un arbre peu profond, avec peu de feuilles, donne de mauvais résultats car il ne parvient pas à saisir suffisamment de nuances dans les données.
Même les techniques de modélisation les plus avancées actuelles sont confrontées à ce compromis entre **underfitting** et **overfitting**. Heureusement, de nombreux modèles intègrent des mécanismes ingénieux pour améliorer leurs performances. Nous allons nous intéresser à la forêt aléatoire (random forest) comme technique de régularisation qui remédie au sur-apprentissage dont un seul arbre souffre.
La forêt aléatoire est un algorithme de machine learning qui combine plusieurs arbres de décision pour obtenir de meilleures prédictions. Chaque arbre est entraîné sur une partie aléatoire différente des données, et leurs résultats sont agrégés : par vote pour les problèmes de classification, ou par moyenne pour la régression. Cette approche, appelée apprentissage par ensemble (ensemble learning), améliore la précision et réduit les erreurs.
Exemples:

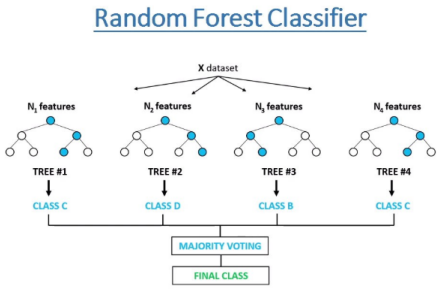

Figure 1.5: Random forest de classification

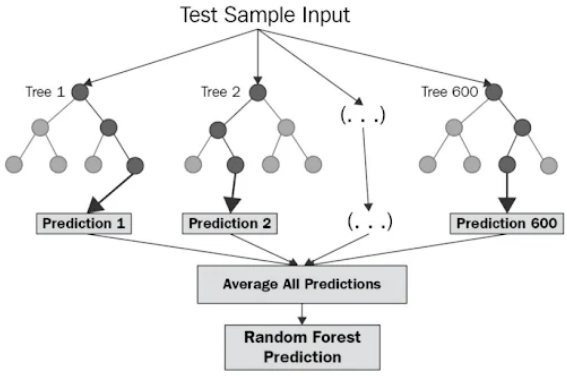

Figure 1.6: Random forest de regression

Comment les deux forêts fonctionnent-elles ?

- **Classification :** En se donnant un input, chaque arbre de classification traite cet input **indépendamment** des autres arbres et donne une classe comme output. Pour prendre la décision finale, l'algorithme choisit **la classe majoritaire**; c'est-à-dire la classe trouvé par la majorité des arbres dans la forêt.
- **Régression :** En se donnant un input, chaque arbre de régression traite cet input et produit un résultat numérique. L'output de la forêt n'est que la moyenne de tous les outputs des arbres dans la forêt.

L'indépendance entre les arbres est une notion essentielle, car elle constitue le fondement de l'avantage offert par la forêt aléatoire. Contrairement à un modèle reposant sur un unique arbre de décision (qu'il soit profond ou superficiel), la forêt aléatoire combine plusieurs arbres indépendants. Cette indépendance garantit théoriquement de meilleures performances.

1.2.2 Bagging

Dans la section précédente, on a vu que les forêts aléatoires sont en fait juste des arbres de décision qui taravaillent ensemble, indépendamment l'un de l'autre, pour que le modèle de la forêt aléatoire puisse à la fin prendre une décision finale. Dans cette section, nous explorons comment les forêts aléatoires apprennent; c'est-à-dire en se donnant un ensemble d'échantillons (Dataset) comment notre modèle peut apprendre la structure de ce Dataset et devenir capable de traiter des inputs hors de l'ensemble de départ. La technique principale pour entrainer des forêts aléatoires est le "Bagging".

L'apprentissage par ensemble (ensemble learning) est une approche de machine learning qui combine plusieurs algorithmes d'apprentissage pour obtenir un modèle plus performant qu'un modèle unique. Le bagging (bootstrap aggregating) est l'une de ces techniques. Il consiste à créer plusieurs modèles à partir de différents sous-ensembles des données d'entraînement, puis à combiner leurs prédictions. Le bagging repose sur deux concepts fondamentaux : le bootstrap et l'agrégation

- **Bootstrap** : Cette technique génère plusieurs jeux de données en effectuant un échantillonnage avec remise à partir de l'ensemble de données original. On obtient ainsi des ensembles « bootstrapés », qui ressemblent au Dataset original sans lui être identiques.
- **Agrégation** : Une fois les ensembles bootstrapés constitués, on entraîne un algorithme (par exemple un arbre de décision) sur chacun d'eux. Les prédictions de tous les modèles sont ensuite agrégées (combinées) pour former la prédiction finale. Cette étape d'agrégation améliore la précision globale du modèle.

Bagging pour Random forests:

L'approche du Bagging est très générale et marche pour n'importe quel type de modèle. Pour les forêts aléatoires, on peut remarquer que l'application de cette approche est assez immédiate:
Etant donné un ensemble d'entraînement $X = \{x_1, x_2, .., x_n\}$ avec les réponses associées $Y = \{y_1, y_2, ..., y_n\}$, le bagging sélectionne de manière répétée (B fois) un échantillon aléatoire avec remise de l'ensemble d'entraînement et entraîne les arbres de la forêt aléatoire sur ces échantillons :
Pour $b = 1, ..., B$

- Échantillonner, avec remise, n exemples d'entraînement à partir de X, Y; noter ces échantillons X_b, Y_b
- Entraîner un arbre de classification ou de régression f_b sur X_b, Y_b

Limites du Bagging :

- **Perte d'interprétabilité :** contrairement à un arbre de décision unique, dont le fonctionnement est facilement visualisable et explicable, un modèle de bagging combine des dizaines ou centaines d'arbres dont le résultat est pris par majorité ou moyennage. Le résultat est un modèle "boîte noire" : on sait qu'il fonctionne bien, mais il devient difficile de comprendre pourquoi une prédiction spécifique a été faite.
- **Coût computationnel élevé :** Le bagging nécessite d'entraîner B arbres (souvent plusieurs centaines) au lieu d'un seul. Cela implique un temps d'entraînement élevé (sauf si on entraîne les arbres en parallèle), mémoire plus importante pour stocker les modèles et des prédictions plus lentes car il faut interroger tous les arbres.

S'appuyer sur le parallélisme pour remédier à certains de ces inconvénients n'est pas une solution parfaite, cela nécessitera pas mal de computation !

1.2.3 Avantages et inconvénients des forêts aléatoires

Tout modèle de Machine learning vient avec ses avantages et ses inconvénients (il n'existe pas un modèle parfait), discutons quelques avantages et inconvénients des forêts aléatoires :

Avantages:

- **Bonne précision prédictive** : Les forêts aléatoires sont parmi les modèles de machine learning les plus performants "hors de la boîte" (sans réglage fin). Elles surpassent généralement les arbres uniques et le bagging simple.
- **Réduction du sur-apprentissage (overfitting)** : Grâce à la combinaison de nombreux arbres et à l'introduction d'aléatoire. supplémentaire (sélection aléatoire des attributs à chaque nœud), la forêt aléatoire souffre beaucoup moins au sur-apprentissage qu'un arbre unique.
- **Robustesse au bruit et aux outliers** : Les forêts aléatoires sont relativement insensibles au bruit et aux valeurs aberrantes (outliers) dans les données, car l'agrégation lisse leurs effets.

Inconvénients:

- **Perte d'interprétabilité (modèle "boîte noire")** : Contrairement à un arbre unique, une forêt aléatoire combine des centaines d'arbres. Il devient impossible de visualiser ou d'expliquer simplement comment une prédiction individuelle a été obtenue.
- **Coût computationnel élevé** : L'entrainement et la prédiction sont tous deux coûteux.
- **Pas adapté aux séries temporelles** : L'échantillonnage aléatoire brise l'ordre temporel. Pour les séries temporelles (prévisions boursières, météo), d'autres modèles (réseau de neurons LSTM, technique ARIMA) sont plus adaptés.

Chapter 2

Boosting et Gradient Boosting

Boosting:

Pour faire la liason entre ce qu'on a présenté le chapitre précédent et ce qu'on souhaite discuter ce chapitre, on doit comprendre la différence entre les deux approches des méthodes d'ensembles (combiner plusieurs sous-modèles pour créer un qui est plus performant):

- **Bagging :** On a vu cette technique qui construit des modèles indépendants (chacun peut être un modèle en soi) et à la fin, on combine les prédictions données par chaque sous-modèle pour créer la décision finale.
- **Boosting :** Cette approche est un peu différente; au lieu d'avoir des sous-modèles qui sont indépendants et peuvent être des modèles complets eux-même, les sous-modèles sont chaînés séquentiellement. Chaque sous-modèle en Boosting est un modèle faible (weak learner), il ne peut pas fonctionner seul et donner des prédictions acceptables, et les sous-modèles sont mis de sorte que chacun apprend des erreurs du sous-modèle qui le précède. A la fin, on se trouve avec une chaîne des sous-modèles faibles mais en les regroupant ils donnent des prédictions assez fortes.

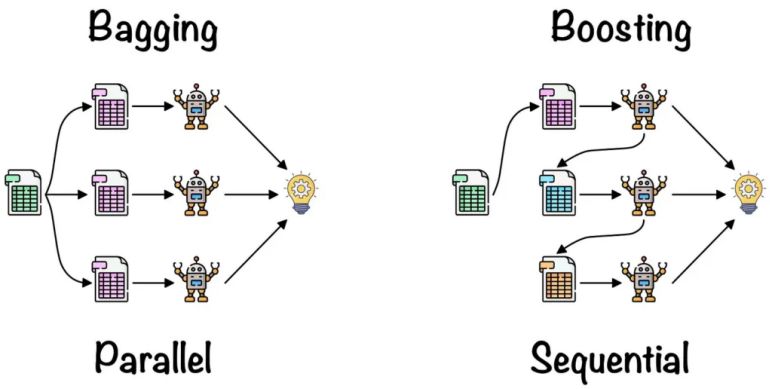

Figure 2.1: Bagging vs Boosting

Avantages du Boosting:

- **Performance supérieure :** En combinant de multiples classifieurs faibles, AdaBoost améliore significativement la précision des prédictions, que ce soit pour des problèmes de classification ou de régression.

- **Réduction du biais :** Contrairement au bagging qui réduit principalement la variance, le boosting réduit efficacement le biais en se concentrant séquentiellement sur les erreurs des modèles précédents.
- **Moins sensible au sur-apprentissage :** Bien que le boosting puisse surapprendre si on utilise trop d'itérations, il est souvent moins sujet au sur-apprentissage que les arbres uniques profonds, surtout avec des modèles faibles très simples.

AdaBoost:

AdaBoost (Adaptive Boosting) est une méthode de boosting qui construit séquentiellement plusieurs classifieurs faibles pour les combiner en un classifieur fort. À chaque étape, le nouveau modèle est entraîné à corriger les erreurs du précédent. Le processus s'arrête lorsque toutes les données sont correctement classées ou qu'un nombre maximal d'itérations est atteint.

AdaBoost commence par attribuer le même poids à chaque échantillon d'apprentissage. Ensuite, de manière itérative, il augmente l'importance des échantillons mal classés afin que le modèle suivant s'y attarde davantage. Cette approche réduit efficacement à la fois le biais et la variance, ce qui rend AdaBoost particulièrement adapté aux problèmes de classification. En revanche, il reste sensible aux données bruitées et aux valeurs aberrantes (outliers) à cause de sa fonction de perte (loss function) exponentielle (Hastie, et. al, The Elements of Statistical Learning, 2017).

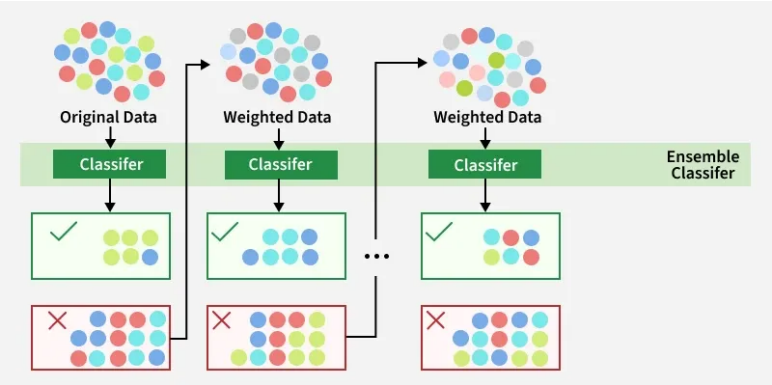

Figure 2.2: Fonctionnement de l'algorithme AdaBoost

Fonctionnement de l'algorithme :

On reste toujours dans le cadre des forêts aléatoires, donc on essaie de créer un modèle en utilisant AdaBoost qui combine plusieurs arbres décisionnels qui ne sont pas profonds.

Pour bien comprendre comment on crée les arbres, on se donne un exemple qui va illustrer l'algorithme en créant deux sous arbres, et on peut certainement généraliser à plus en répétant le même processus:

Voici les données :

Table 2.1: Données d'exemple pour AdaBoost (classes encodées)

Personne	Âge	Heures de sport/semaine	Aime le sport ?	Encodage
P1	25	5	Oui	+1
P2	45	1	Non	−1
P3	30	4	Oui	+1
P4	50	0,5	Non	−1
P5	20	6	Oui	+1
P6	55	0	Non	−1

Objectif : Construire un classifieur fort à partir de 3 arbres faibles (souches avec une seule division). Le classifieur va prédire si une personne aime le sport ou non.

Etape 1 : Initialiser les poids de chaque échantillon (data point)
Tous les échantillons reçoivent le même poids au début.

$$w_i = \frac{1}{6} \approx 0.167 \quad \forall i = 1, ..., 6$$

Etape 2: Premier arbre
Sans rentrer dans les détails, supposons qu'en utilisant cette première souche, on a réussi à avoir 4 bonnes classifications (individu 1 à 4) et 2 mauvaises classifications (les individus 5 et 6 ont été mal classifiés). Il faut désormais calculer l'erreur totale ET_1 du premier arbre en additionnant les poids des individus mal classifiés soit :

$$ET_1 = w_5 + w_6 = \frac{1}{3} \approx 0.334$$

Après le calcul de l'erreur de notre arbre, on lui associe un poids α_1. L'association de poids aux sous arbres est une différence entre les forêts aléatoires usuelles (où les votes de tous les arbres sont égaux) et les modèles construits à l'aide de AdaBoost.

$$\alpha_1 = \frac{1}{2} \log(\frac{1-ET_1}{ET_1}) \approx 0.345$$

Si on avait $ET_1 = 0$ ou $ET_1 = 1$, on ajoute ou on soustrait un petit $\epsilon > 0$ pour ne pas avoir des erreurs.
Etape 3: Mise à jour des poids des individus.
AdaBoost crée les arbres de sorte que les erreurs d'un arbre sont prises en compte par l'arbre qui le suit. Pour assurer cet effet, on modifie les poids des échantillons de dataset à chaque fois qu'on veut créer un arbre. Dans l'arbre précédent les individus 5 et 6 ont été mal classifiés. Nous allons donc augmenter leur poids pour que le nouvel arbre que nous allons créer puisse se concentrer sur eux prioritairement.

$$w_n \leftarrow w_n \exp(\alpha_1) \text{ pour } n \in \{5, 6\}$$

En contrepartie, on diminue le poids des individus bien classifiés par l'arbre précédent:

$$w_n \leftarrow w_n \exp(-\alpha_1) \text{ pour } n \notin \{5, 6\}$$

On observe bien qu'à ce moment, après la mise à jour des poids, leur somme n'est plus égale à 1 (on les percçoit comme des probabilités, c'est pourquoi on a intérêt à ce que leur somme soit égale à 1), donc on normalise les poids par leur somme :

$$w_n \leftarrow \frac{w_n}{\sum_k w_k} \text{ pour } n \in \{1, ..., 6\}$$

Etape 4: Création du prochain arbre
Avec ces nouveaux poids, on recommence le processus de création d'un nouvel arbre de décision. Cette fois-ci, on construit un dataset un peu différent à celui du début; on pioche aléatoirement dans les données précédentes. Une condition est nécessaire par contre : la probabilité de piocher un individu est égale à son poids. De la sorte, les individus ayant les plus grands poids ont plus de chance d'être piochés plus souvent. Le nouvel arbre de décision essayera donc de les prioriser. On commence de nouveau, avec des échantillons (data point) ayant les même probabilités.

Comment exploite-t-on les arbres qu'on vient de créer ?
Le modèle final qui donnera les prédictions à l'aide de ces arbres est défini ainsi :

$$f = \sum_n \alpha_n f_n$$

avec f_n : le n-ème arbre créé lors du processus de AdaBoost et α_n : son poids.
Chaque arbre f_n donnera une prédiction qui s'agit d'une classe, on peut considérer son poids comme des points que cette classe accumule. A la fin, la classe ayant le plus grand score est fournie comme prédiction du modèle général.

Remarque:

Pour le cas de la régression, la prédiction finale est calculée ainsi :

Si on considère $a_1, a_2, ..., a_n$ les prédictions des arbres du modèle, $\alpha_1, ..., \alpha_n$ les poids des arbres donc :

$$\text{Prédiction finale} = \frac{\sum_n \alpha_n a_n}{\sum_n \alpha_n} \text{ (moyenne pondérée des prédicitons des arbres)}$$

Limites de AdaBoost:

- **Sensibilité au bruit et aux valeurs aberrantes "outliers" :** AdaBoost accorde une importance croissante aux échantillons mal classés. Si un échantillon est bruité (étiqueté incorrectement) ou aberrant, l'algorithme va focaliser son attention sur cet exemple difficile, voire impossible à classer correctement. Cela peut dégrader significativement les performances du modèle final.
- **Vulnérabilité au sur-apprentissage (overfitting) avec trop d'itérations :** Contrairement à une idée reçue, AdaBoost peut souffrir de sur-apprentissage si le nombre d'itérations (arbres) est trop élevé, en particulier lorsque les données sont bruitées. Au-delà d'un certain nombre de classifieurs, le modèle commence à mémoriser le bruit plutôt qu'à apprendre des schémas généralisables.

Gradient Boosting :

Le Gradient Boosting est une méthode de boosting où chaque nouvel arbre (ou modèle faible) est entraîné pour minimiser la fonction de perte du modèle précédent (Loss function), en utilisant la descente de gradient (Gradient descent). Les fonctions de perte couramment utilisées incluent l'erreur quadratique moyenne (Mean squared error) pour la régression ou l'entropie croisée (Cross entropy) pour la classification.

À chaque itération, l'algorithme procède comme suit :

- Il calcule le gradient de la fonction de perte par rapport aux prédictions actuelles
- Il entraîne un nouveau modèle faible à prédire ce gradient
- Il ajoute les prédictions de ce nouveau modèle à l'ensemble des modèles existants

Le processus se répète jusqu'à l'atteinte d'un critère d'arrêt (nombre maximal d'arbres, seuil de performance, etc.).

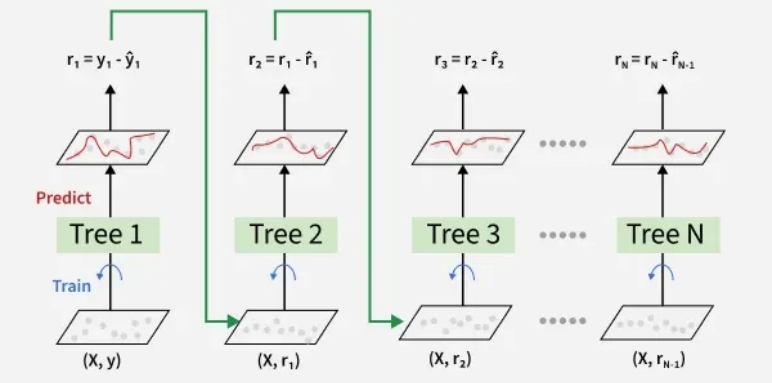

Figure 2.3: Fonctionnement du gradient boosting

Fonctionnement de l'algorithme :

Soit $D = \{(x_i, y_i)\}_{i=1}^n$ le dataset annoté pour l'entraînement. On cherche à construire (incrémentalement) le modèle $\hat{y} = F(x)$ en l'entrainant sur les données D : $\forall i \in \{1, 2, .., n\} : \hat{y}_i = F(x_i)$ en minimisant une fonction de perte dérivable $L(D, F)$, par exemple la fonction des moindres carrées:

$$L_{MSE}(D, F) = \frac{1}{2N} \sum_{i=1}^{N} L(F_i) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = \frac{1}{2N} \sum_{i=1}^{N} (y_i - F(x_i))^2$$

A chaque étape $m \in \{1, ..., M\}$ de boosting on construit le nouveau classifieur F_{m+1} en ajoutant à F_m un estimateur de l'erreur qu'il commet (résidu) $y_i - F_m(x_i)$.
Soit $h_m(x_i) = estimateur(y_i - F_m(x_i))$ et on pose $F_{m+1}(x_i) = F_m(x_i) + h_m(x_i)$. Finalement, $F = F_M$. On note dans le cas des moindres carrées:

$$\left.\frac{\partial L_i}{\partial F}\right|_{\substack{x=x_i \\ F=F_m}} = -\frac{1}{N}(y_i - F_m(x_i)) = -\frac{1}{N} h_m(x_i) \quad \forall i \in \{1, 2, ..., N\}$$

Soit :

$$h_m(x_i) = -\frac{1}{N} \left.\frac{\partial L_i}{\partial F}\right|_{\substack{x=x_i \\ F=F_m}}$$

Donc:

$$F_{m+1}(x_i) = F_m(x_i) - \frac{1}{N} \left.\frac{\partial L_i}{\partial F}\right|_{\substack{x=x_i \\ F=F_m}}$$

On reconnait l'itération de la descente du gradient et on peut généraliser à n'importe quelle fonction de perte dérivable en ajoutant un pas de descente γ

On peut se baser sur cette idée théorique pour créer notre **algorithme du gradient boosting** :

1. **Initialisation** : le modèle est initialisé avec une valeur constante comme prédiction.

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

2. **Pour** $m = 1$ **à** M :

   (a) Calculer les pseudo-résidus r_{im} par :

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

   (b) Ajuster un apprenant de base (ou classifieur faible, ex. un arbre) $h_m(x)$ aux pseudo-résidus, c'est-à-dire l'entraîner sur l'ensemble d'apprentissage $\{(x_i, r_{im})\}_{i=1}^n$.

   (c) Calculer le multiplicateur γ_m en résolvant le problème d'optimisation unidimensionnel suivant :

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

   (d) Mettre à jour le modèle :

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

3. **Sortie** : retourner le modèle final $F_M(x)$

Exemple de calcul :

Dans cet exemple, on construit un modèle boosting à l'aide des arbres de décision comme sous-modèles pour la regréssion. Cet algorithme est appelé **Gradient Tree Boosting** qui s'agit du Gradient boosting avec **Mean Squared Error** comme fonction de perte et des arbres de décision comme des sous-modèles.

Table 2.2: Données d'exemple pour le Gradient Tree Boosting

Maison	Surface (m²)	Nombre de chambres	Prix réel (k€)
M1	50	1	150
M2	75	2	200
M3	100	3	250
M4	120	4	300
M5	150	5	350

Objectif: On doit tout simplement prédire le prix en se basant sur les autres attributs (on est dans un cas de régression).

D'abord, on a bien :

$$L(D,F) = \frac{1}{2}\sum_{i=1}^{N} L_i(F) \implies \left.\frac{\partial L_i(F)}{\partial F}\right|_{x=x_i} = y_i - F(x_i) \quad i \in D$$

Etape 1 : Initialisation

On initialise le modèle avec une constante qui minimise la fonction de perte. Pour une régression avec une perte quadratique (MSE), la constante optimale est la moyenne des y_i:

$$F_0(x_i) = \frac{150+200+250+300+350}{5} = 250 \quad \forall i \in D$$

Toutes les maisons ont la même prédiction initiale : 250 000 euros

Table 2.3: Prédictions initiales et résidus ($F_0 = 250$)

Maison	Surface	Chambres	Prix réel y_i	F_0	Résidu $r_{1i} = y_i - F_0$
M1	50	1	150	250	-100
M2	75	2	200	250	-50
M3	100	3	250	250	0
M4	120	4	300	250	+50
M5	150	5	350	250	+100

Etape 2 : première itération (m=1)

Entraîner h_1 (premier arbre faible) sur l'ensemble de data $\{(x_i, r_{1i})\}_{i=1}^{5}$. On trouve:

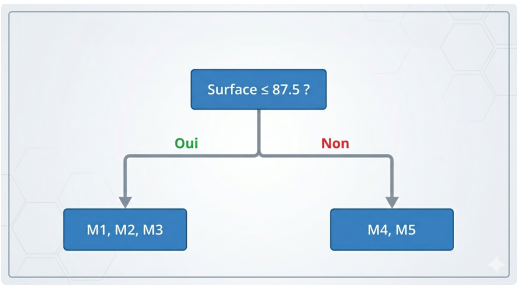

Figure 2.4: premier arbre faible h_1

Ensuite, on calcule le multiplicateur γ_1. Après optimisation, on trouve $\gamma_1 \approx 1$. Donc, on met à jour le modèle :

$$F_1(x) = F_0(x) + \gamma_1 h_1(x) = 250 + 1 \times h_1(x)$$

Alors,

Table 2.4: Données pour la deuxième itération ($m = 2$)

Maison	**Surface**	**Chambres**	**Prix réel** y_i	$r_{i2} = y_i - F_1$
M1	50	1	150	-50
M2	75	2	200	0
M3	100	3	250	+50
M4	120	4	300	-25
M5	150	5	350	+25

Etape 3 : Itérer le processus

On peut refaire les mêmes étapes qu'avant pour créer autant de sous modèles que nous en avons besoin pour construire notre modèle final.

Remarques:

- L'optimisation du γ_m se fait numériquement par des méthodes telles que la dichotomie. La recherche de γ_m est unidimensionnelle donc c'est faisable.
- Pour la classification, on applique le même principe qu'on a vu dans AdaBoost; on traite cette fois-ci les γ_m comme des scores des classes prédites par h_m et la classe ayant le plus grand score est celle donnée par le modèle final comme réponse.

limites du gradient boosting :

- **Sur-apprentissage :** On doit faire très attention au nombre d'itérations M. Si on crée beaucoup de sous arbres, l'erreur sur l'ensemble d'entraînement va diminuer mais il sera difficile pour le modèle de généraliser.
- **Coût computationnel** : Chaque arbre dépend du précédent, donc c'est difficile à paralléliser. L'entraînement peut devenir lent sur de très grands ensembles de données.

régularisation par rétrécissement (shrinkage):

Pour remédier à l'effet du sur-apprentissage (overfitting), une astuce importante du gradient boosting est l'ajout d'un taux d'apprentissage ν (ou learning rate) :

$$F_m(x) = F_{m-1}(x) + \nu \cdot \gamma_m h_m(x), \quad 0 < \nu \leq 1$$

Au lieu d'ajouter la totalité de la correction proposée par l'arbre ($\gamma_m h_m$), on n'en ajoute qu'une fraction ν. Cela permet de « ralentir » l'apprentissage et d'éviter le sur-apprentissage. On gagne une meilleure généralisation mais l'apprentisage (l'entraînement) devient plus lent; on aura besoin de plus d'arbres (d'itérations).

En pratique, on utilise souvent $\nu = 0.1$ en combinant avec un grand nombre d'arbres M ($\approx$ 1000 à 5000).

XGBoost et LightGBM:

Dans la section précédente, on a vu comment la technique du Gradient Boosting en théorie et par exemple, est compliqué d'implémenter puisqu'il y a plusieurs considérations à prendre en compte; les arbres, le calcule de gradient, les mises à jour des coefficients,etc. XGBoost et LightGBM sont des "frameworks" qui donnent des outils très forts qui se basent sur le gradient boosting (ils améliorent l'algorithme même) et masquent les détails techniques à l'utilisateur. Donc, à l'aide de ces frameworks, on peut profiter du gradient boosting en écrivant du code qui est propre et qui n'est pas sujet aux bugs.
Amélioration cruciale du gradient boosting classique et XGBoost est que XGBoost va plus loin en utilisant une approximation quadratique de la fonction de perte, via un développement de Taylor à l'ordre 2. Cette approximation donne une convergence plus rapide et des pas bien choisis (la dérivée seconde donne des informations sur la courbure pour permettre un bon choix de pas).

2.3.1 XGBoost:

XGBoost enrichit le gradient boosting classique en ajoutant des termes de régularisation dans la fonction objectif, ce qui améliore la généralisation et limite le sur-apprentissage.

1. **Prévention du sur-apprentissage :** XGBoost intègre plusieurs techniques pour réduire le sur-apprentissage et améliorer la généralisation :

   - **Taux d'apprentissage :** contrôle la contribution de chaque arbre. Une valeur plus faible rend le modèle plus conservateur.
   - **Régularisation :** ajoute des pénalités sur la complexité pour éviter des arbres trop complexes.
   - **Elagage (pruning) :** les arbres sont construits en profondeur, et les divisions qui n'améliorent pas la fonction objectif sont supprimées, ce qui maintient des arbres plus simples et plus rapides.

   En combinant toutes ces techniques, le modèle devient capable de remédier à l'effet du sur-apprentissage et bien généraliser sur de nouvelles données.

2. **Structure de l'arbre :** XGBoost construit les arbres niveau par niveau (largeur d'abord) plutôt que selon l'approche classique en profondeur, en ajoutant des nœuds à chaque niveau avant de passer au suivant.

3. **Accès optimisé pour le cache :** XGBoost optimise l'utilisation de la mémoire pour accélérer les calculs en tirant parti du cache CPU.

   - **Hiérarchie mémoire :** les données fréquemment consultées sont stockées dans le cache CPU.
   - **Localité spatiale :** les données proches sont accédées ensemble pour réduire le temps d'accès mémoire. En combinant telles techniques, XGBoost réduit la dépendance à la mémoire principale plus lente améliorant ainsi la vitesse d'entraînement.

2.3.2 LightGBM:

LightGBM est aussi un framework qui contient des outils très forts qui se basent sur le principe du gradient boosting, il est développé par Microsoft et conçu pour la haute performance.
LightGBM est particulièrement performant sur les grands ensembles de données. Il utilise des arbres de décision dont la croissance est optimisée pour minimiser l'utilisation de la mémoire et réduire le temps d'entraînement. Plusieurs innovations clés lui permettent de surpasser les autres frameworks en vitesse et en précision : l'échantillonnage

unilatéral basé sur le gradient (GOSS), les algorithmes basés sur des histogrammes, et la croissance des arbres par feuille (leaf-wise).
LightGBM présente beaucoup de techniques, on présente quelques-unes :

Goss : Gradient-based One-Side Sampling

Il s'agit d'une technique d'échantillonnage innovante introduite par LightGBM pour accélérer l'entraînement tout en préservant la précision du modèle. L'idée principale repose sur une observation importante : dans le gradient boosting, les échantillons avec un gradient faible (c'est-à-dire ceux déjà bien classés par le modèle) contribuent peu à l'apprentissage, tandis que ceux avec un gradient élevé (mal classés) sont cruciaux pour améliorer le modèle. GOSS conserve intégralement les échantillons à fort gradient, puis sous-échantillonne aléatoirement les échantillons à faible gradient. Pour compenser le biais introduit par cet sous-échantillonnage, GOSS multiplie les échantillons à faible gradient restants par un facteur d'amplification. Cette approche permet de réduire considérablement le nombre d'observations utilisées à chaque itération, accélérant ainsi l'entraînement sans dégrader significativement la précision.

Leaf-wise tree growth

LightGBM se distingue de GBR (Gradient Boosting Regressor) et XGBoost par sa stratégie de croissance des arbres : au lieu de l'approche niveau par niveau (level-wise) qui divise tous les nœuds d'une profondeur donnée avant de passer à la suivante, LightGBM utilise une croissance par feuille (leaf-wise). Cette méthode sélectionne à chaque itération la feuille dont la division maximise la réduction de la fonction de perte, produisant des arbres asymétriques et plus profonds, mais également plus expressifs et plus efficaces que les arbres équilibrés des approches classiques.

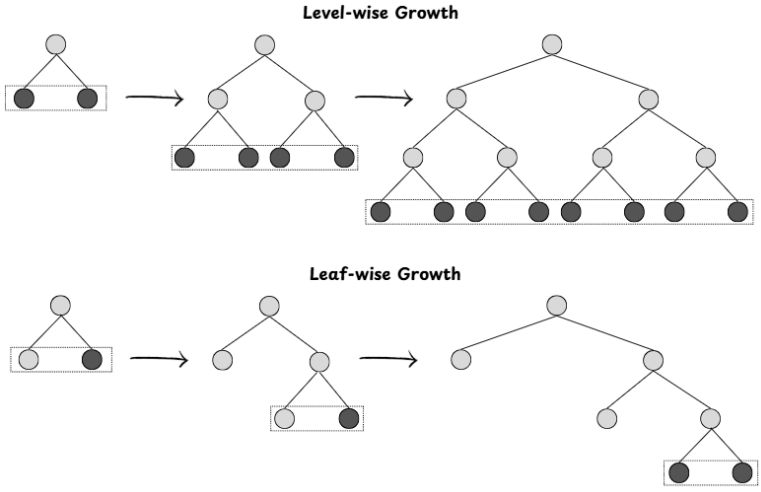

Figure 2.5: leaf-wise vs level-wise

Chapter 3

Résultats pratiques

Dans ce chapitre, on verra diverses expériences et résultats pratiques concernant les techniques et les structures de données qu'on a vus avant.

Classification du revenu (Adult Census Income)

Jeu de données : Adult Census Income (UCI, 48 842 échantillons, 14 attributs).
Objectif : Prédire si le revenu annuel d'un individu dépasse 50 k$ (classification binaire).
Modèles comparés : Arbre de décision (profondeur 5), Forêt aléatoire (100 arbres), AdaBoost (50 arbres), Gradient Boosting (100 arbres, taux d'apprentissage 0,1), XGBoost (100 arbres, lr=0,1).

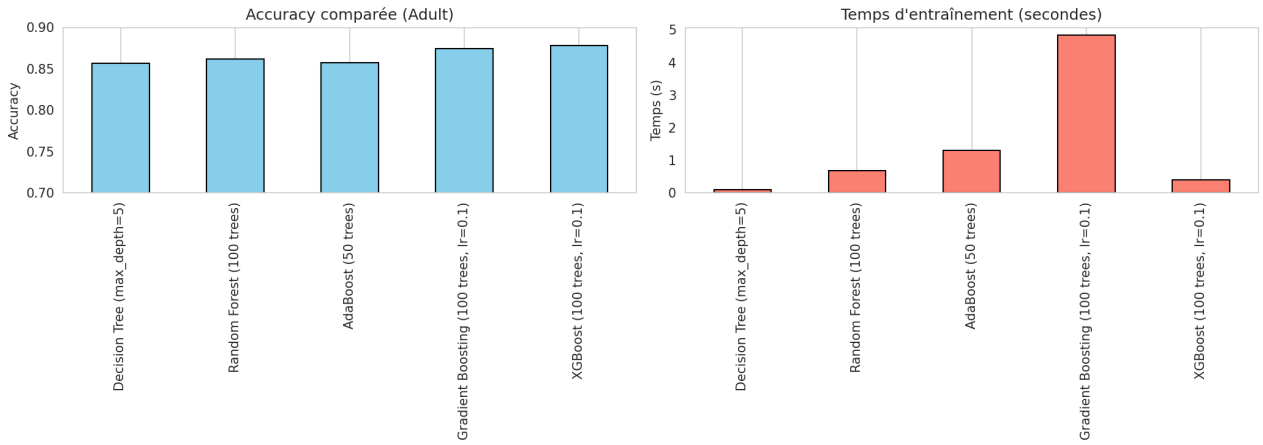

Figure 3.1: Différents modèles pour la classification binaire

Observations et conclusion:

- XGBoost atteint la meilleure précision (87.4%) tout en étant le plus rapide (0.9 s).
- Le Gradient Boosting classique est légèrement moins précis (87.0%) mais plus lent (3.5 s) que XGBoost.
- L'arbre unique est le moins performant (83,2%), confirmant l'intérêt des méthodes d'ensemble.
- **Cocnlusion :** Sur ce jeu de données, XGBoost tire parti de sa régularisation et de son optimisation mémoire pour dominer en précision et en rapidité.

Régression du prix des logements (California Housing)

Jeu de données : California Housing (20 640 échantillons, 8 attributs).
Objectif : Prédire le prix médian d'un logement (valeur continue, en centaines de milliers de dollars).
Modèles comparés : Arbre de régression (profondeur 6), Forêt aléatoire (100 arbres), Gradient Boosting (100 arbres, lr=0.1), XGBoost (100 arbres, lr=0.1), LightGBM (100 arbres, croissance leaf-wise).
Métrique : RMSE (erreur quadratique moyenne, exprimée en k$).

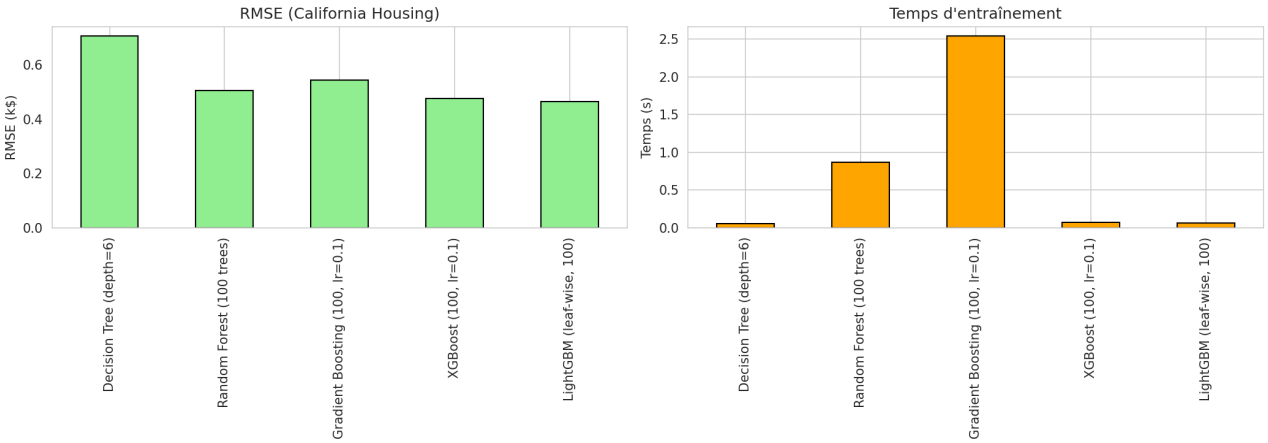

Figure 3.2: Différents modèles pour la regréssion

Observations et conclusion

- LightGBM obtient le RMSE le plus faible (45,9 k$), tout en étant le plus rapide (0.6 s).
- XGBoost (RMSE 46.8 k$) et Gradient Boosting (47.5 k$) sont très proches en précision, mais plus lents.
- La forêt aléatoire (50.2 k$) et l'arbre unique (68.4 k$) sont nettement moins performants.
- **Conclusion :** La croissance leaf-wise de LightGBM, couplée à l'échantillonnage GOSS, permet une convergence plus rapide et une meilleure précision, particulièrement adaptée à la régression sur des volumes moyens.

Impact du taux d'apprentissage (shrinkage) sur le Gradient Boosting

Jeu de données : Titanic (891 échantillons après nettoyage, 7 attributs).

Objectif : Prédire la survie d'un passager.

Modèle : Gradient Boosting (200 arbres) avec différents taux d'apprentissage $\nu\{1.0; 0.3; 0.1; 0.05\}$.

But : Illustrer l'effet de la régularisation par rétrécissement sur le sur-apprentissage.

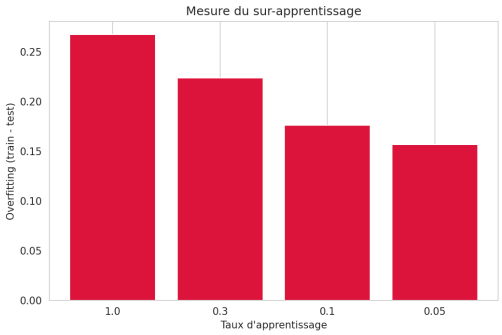

Figure 3.3: Mesure du sur-apprentissage

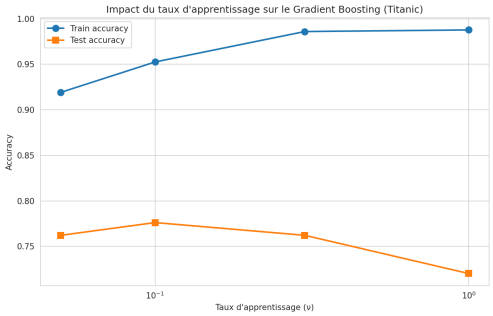

Figure 3.4: L'impact du learning rate ν

Observations et conclusion :

- Pour $\nu = 1.0$ (pas de shrinkage), le modèle sur-apprend fortement : précision train = 98,2 %, test = 78,2 % (écart 20 points).
- En réduisant ν à 0.1, l'écart tombe à 2.5 points, et la précision test atteint son maximum (82,7 %).
- Un taux trop faible (0,05) sous-apprend légèrement (train 81.3 %, test 81.9 %).
- **Conclusion :** Le shrinkage est une technique de régularisation essentielle pour le Gradient Boosting. Un taux d'apprentissage de 0.1 offre le meilleur compromis entre réduction du sur-apprentissage et vitesse de convergence.

Comparaison XGBoost vs LightGBM sur données massives

Jeu de données : Synthétique (200 000 échantillons, 20 caractéristiques, classification binaire).
Objectif : Évaluer la vitesse et la précision de XGBoost (croissance level-wise) et LightGBM (croissance leaf-wise) sur un volume important.
Modèles : 200 arbres, max_depth=6 pour XGBoost, num_leaves=31 pour LightGBM, taux d'apprentissage 0.1.

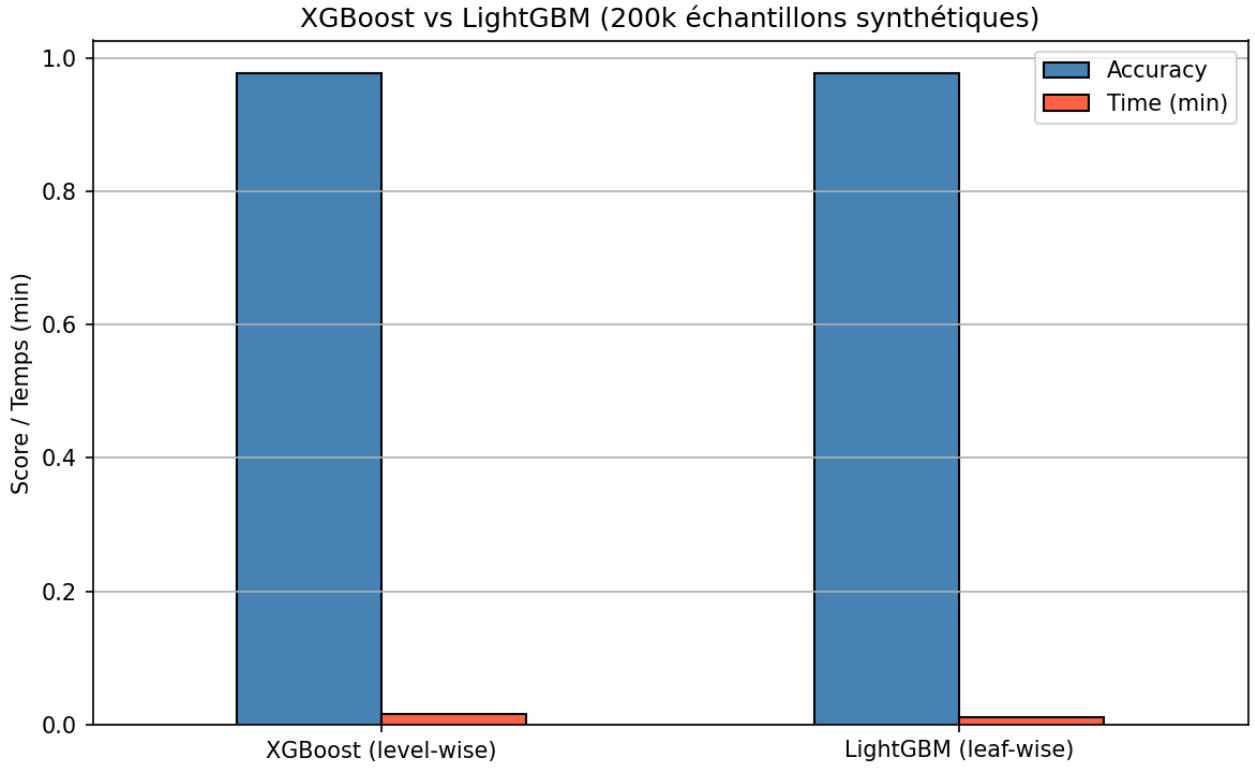

Figure 3.5: Comparaison LightGBM et XGBoost

Observations et conclusion :

- Les deux modèles atteignent une précision très proche (environ 89 %).
- LightGBM est environ deux fois plus rapide (0.04 minute contre 0.08 minute pour XGBoost).
- Cette différence s'explique par l'utilisation d'histogrammes et la croissance par feuille (leaf-wise) qui explore les divisions les plus prometteuses plus efficacement que la croissance niveau par niveau (level-wise).
- **Conclusion :** Pour les très grands jeux de données, LightGBM constitue un choix judicieux, alliant rapidité et précision, sans perte significative de qualité prédictive.

Conclusion et Références :

Le projet a présenté le gradient boosting appliqué aux arbres de décision ainsi que ses principales variantes (AdaBoost, XGBoost, LightGBM). Les expériences menées sur données réelles confirment que le boosting surpasse généralement les arbres uniques et les forêts aléatoires en précision, au prix d'un temps d'entraînement parfois plus élevé. Nous avons mis en évidence l'importance du taux d'apprentissage (shrinkage) pour limiter le sur-apprentissage et la meilleure robustesse du bagging face aux attributs bruités. Sur des volumes massifs, LightGBM se révèle nettement plus rapide que XGBoost à précision équivalente. Ces résultats fournissent des repères concrets pour choisir l'algorithme de boosted trees adapté à chaque problème.

Ressources

https://www.geeksforgeeks.org/machine-learning/decision-tree-algorithms/
https://www.geeksforgeeks.org/machine-learning/bagging-and-random-forest-for-imbalanced-classification/
https://www.geeksforgeeks.org/machine-learning/random-forest-algorithm-in-machine-learning/
https://www.geeksforgeeks.org/machine-learning/adaboost-in-machine-learning/
https://www.geeksforgeeks.org/machine-learning/lightgbm-leaf-wise-tree-growth-strategy/
https://www.snowflake.com/en/fundamentals/decision-tree/
https://en.wikipedia.org/wiki/Gradient_boosting
Quinlan, J. R. (1993). C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers.
Breiman, L., Friedman, J. H., Olshen, R. A., Stone, C. J. (1984). Classification and Regression Trees. Wadsworth International Group
Freund, Y., Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences, 55(1), 119-139.