Introduction aux méthodes ensemblistes*

Mélina Hillion
Unité SSP-Lab
Insee
melina.hillion@insee.fr

Olivier Meslin
Unité SSP-Lab
Insee
olivier.meslin@insee.fr

July 3, 2025

Abstract

A compléter

Keywords: machine learning • méthodes ensemblistes • formation

*Nous remercions Daffy Duck et Mickey Mouse pour leur contribution.

Sommaire

1. Introduction ····· 4

2. Aperçu des méthodes ensemblistes ····· 5

2.1. Que sont les méthodes ensemblistes? ····· 5

2.2. Pourquoi utiliser des méthodes ensemblistes? ····· 5

2.3. Comment fonctionnent les méthodes ensemblistes? ····· 7

2.3.1. Le modèle de base: l'arbre de classification et de régression ····· 7

2.3.2. Le *bagging* (Bootstrap Aggregating) et les forêts aléatoires ····· 9

2.3.3. Le *gradient boosting* ····· 13

2.4. Comparaison entre forêts aléatoires et *gradient boosting* ····· 16

2.4.1. Quelle approche choisir? ····· 17

3. Les arbres de décision ····· 18

3.1. Le principe fondamental: partitionner pour prédire ····· 18

3.1.1. Les défis du partitionnement optimal ····· 18

3.1.2. Les solutions apportées par les arbres de décision ····· 19

3.1.3. Terminologie et structure d'un arbre de décision ····· 19

3.1.4. Propriétés des arbres de décision ····· 21

3.2. La construction d'un arbre de décision par l'algorithme CART ····· 23

3.2.1. Définir une mesure d'impureté adaptée au problème ····· 24

3.2.2. Construire l'arbre de décision par un partitionnement séquentiel ····· 26

3.2.3. Contrôler la complexité de l'arbre ····· 27

3.2.4. Utiliser l'arbre pour prédire ····· 27

3.3. Avantages et limites des arbres de décision ····· 28

3.3.1. Avantages ····· 28

3.3.2. Limites ····· 28

4. Le *bagging* ····· 30

4.1. Principe du *bagging* ····· 30

4.2. Pourquoi (et dans quelles situations) le *bagging* fonctionne ····· 30

4.2.1. La régression: réduction de l'erreur quadratique moyenne par agrégation ···· 31

4.2.2. La classification: vers un classificateur presque optimal par agrégation ····· 32

4.3. Quand utiliser le *bagging* en pratique 34
5. La forêt aléatoire 36
5.1. Principe de la forêt aléatoire 36
5.2. Comment construit-on une forêt aléatoire? 36
5.3. Pourquoi les forêts aléatoires sont-elles performantes? 38
5.3.1. Réduction de la variance par agrégation 38
5.3.2. Convergence et limite théorique au surajustement 38
5.3.3. Facteurs influençant l'erreur de généralisation 38
5.4. Evaluation des performances par l'erreur *Out-of-Bag* (OOB) 39
5.5. Interprétation et importance des variables 40
5.5.1. Mesures d'importance classiques (et leurs biais) 41
5.5.2. Méthodes d'importance avancées 42
5.6. Le *gradient boosting* 43
5.6.1. Introduction 43
5.6.2. Les premières approches du *boosting* 43
5.6.3. Comment fonctionne le *gradient boosting*? 46
5.6.4. Un exemple simple: la régression avec perte quadratique 52
5.6.5. Le grand ennemi du *gradient boosting*: le surajustement 52
5.7. Sujets avancés: traitement des données pendant l'entraînement 53
5.7.1. Le traitement des variables continues: l'utilisation des histogrammes 53
5.7.2. Le traitement des variables catégorielles 54
5.7.3. Le traitement des valeurs manquantes 59
5.8. Sujets avancés: fonctionnement des algorithmes 60
5.8.1. Leaf-wise versus level-wise 60
5.8.2. La classification multiclasse non-ordonnée 60
5.8.3. La classification multiclasse ordonnée 63
5.8.4. Les fonctions de perte custom 63
6. Une bien belle section 64
7. Préparation des données 65
7.1. Préparation des variables explicatives 65

7.1.1. Quels sont les traitements inutiles? ····· 65
7.1.2. Comment traiter les variables catégorielles? ····· 65
7.1.3. Comment traiter les valeurs manquantes? ····· 66
7.1.4. Est-il utile de créer des variables additionnelles? ····· 66
7.2. Préparation de la variable-cible ····· 67
7.2.1. Train-test ····· 68
7.3. Evaluation des performances du modèle et optimisation des hyper-paramètres ···· 68
7.3.1. Estimation de l'erreur par validation croisée ····· 68
7.3.2. Choix des hyper-paramètres du modèle ····· 69
8. Guide d'usage des forêts aléatoires ····· 71
8.1. Quelles implémentations utiliser? ····· 71
8.2. Les hyperparamètres clés des forêts aléatoires ····· 71
8.3. Comment entraîner une forêt aléatoire? ····· 74
8.3.1. Approche simple ····· 74
8.3.2. Approches plus avancées ····· 75
8.4. Mesurer l'importance des variables ····· 76
9. Guide d'usage du *gradient boosting* ····· 78
9.1. Quelle implémentation utiliser? ····· 78
9.2. Les hyperparamètres clés du *gradient boosting* ····· 79
9.3. Comment entraîner un algorithme de *gradient boosting*? ····· 85
9.3.1. Préparer l'optimisation des hyperparamètres ····· 85
9.3.2. Optimiser les hyperparamètres ····· 87
9.4. Utiliser un modèle de *gradient boosting* en prédiction ····· 89
References ····· 90

1. Introduction

Deux constats:

- Puissance prédictive, diversité des cas d'usage, pertinence pour les travaux de la statistique publique.
- Pourtant, méthodes peu connues et peu utilisées dans la statistique publique, probablement en raison de l'éclatement des ressources consacrées aux méthodes ensemblistes: articles fondateurs parfois difficiles à lire, nombreux tutoriels souvent peu développés.

Ce document répond à plusieurs ambitions:

- proposer deux niveaux de lecture

L'existence de ces deux niveaux de lecture implique nécessairement quelques répétitions, qu'on espère toutefois peu nombreuses.

- Proposer une introduction raisonnée aux méthodes ensemblistes qui rassemble l'ensemble des connaissances pertinentes: fondements théoriques et propriétés des algorithmes, description des implémentations existantes et guide d'usage détaillé.

Bien insister sur la

Détailler le plan

- La première partie propose une vue d'ensemble des méthodes ensemblistes

2. Aperçu des méthodes ensemblistes

Principe: Cette section propose une introduction intuitive aux méthodes ensemblistes. Elle s'adresse aux lecteurs qui souhaitent acquérir une compréhension générale du fonctionnement de ces techniques et identifier rapidement les situations concrètes dans lesquelles elles peuvent être utiles. L'objectif est d'en expliciter les principes-clés sans recourir au formalisme mathématique, afin de rendre le contenu accessible sans prérequis.

2.1. Que sont les méthodes ensemblistes?

Les méthodes ensemblistes sont des techniques d'apprentissage supervisé en *machine learning* développées depuis le début des années 1990. Leur objectif est de prédire une variable-cible y (appelée *target*) à partir d'un ensemble de variables prédictives $\mathbf{X}$ (appelées *features*), que ce soit pour des tâches de classification (prédire une catégorie) ou de régression (prédire une valeur numérique). Elles peuvent par exemple être utilisées pour prédire le salaire d'un salarié, la probabilité de réponse dans une enquête, le niveau de diplôme…

Plutôt que de s'appuyer sur un seul modèle complexe, les méthodes ensemblistes se caractérisent par la combinaison des prédictions de plusieurs modèles plus simples, appelés "apprenants faibles" (*weak learner* ou *base learner*), pour créer un modèle performant, dit "apprenant fort" (*strong learner*).

Le choix de ces modèles de base, ainsi que la manière dont leurs prédictions sont combinées, sont des facteurs déterminants de la performance finale. Le présent document se concentre sur les méthodes à base d'**arbres de décisions**, qui sont parmi les plus utilisées en pratique. Nous allons examiner les fondements de ces méthodes, leurs avantages et inconvénients, ainsi que les algorithmes les plus populaires.

2.2. Pourquoi utiliser des méthodes ensemblistes?

Les méthodes ensemblistes sont particulièrement bien adaptées à de nombreux cas d'usage de la statistique publique, pour deux raisons. D'une part, elles sont conçues pour s'appliquer à des *données tabulaires* (enregistrements en lignes, variables en colonnes), structure de données omniprésente dans la statistique publique. D'autre part, elles peuvent être mobilisées dans toutes les situations où le statisticien mobilise une régression linéaire ou une régression logistisque (imputation, repondération…).

Les méthodes ensemblistes présentent trois avantages par rapport aux méthodes économétriques traditionnelles (régression linéaire et régression logistique):

- Elles ont une **puissance prédictive supérieure**: alors que les méthodes traditionnelles supposent fréquemment l'existence d'une relation linéaire ou log-linéaire entre y

et $\mathbf{X}$, les méthodes ensemblistes ne font quasiment aucune hypothèse sur la relation entre y et $\mathbf{X}$, et se contentent d'approximer le mieux possible cette relation à partir des données disponibles. En particulier, les modèles ensemblistes peuvent facilement modéliser des **non-linéarités** de la relation entre y et $\mathbf{X}$ et des **interactions** entre variables explicatives *sans avoir à les spécifier explicitement* au préalable, alors que les méthodes traditionnelles supposent fréquemment l'existence d'une relation linéaire ou log-linéaire entre y et $\mathbf{X}$.

- Elles nécessitent **moins de préparation des données**: elles ne requièrent pas de normalisation des variables explicatives et peuvent s'accommoder des valeurs manquantes (selon des techniques variables selon les algorithmes).
- Elles sont généralement **moins sensibles aux valeurs extrêmes et à l'hétéroscédasticité** des variables explicatives que les approches traditionnelles.

Elles présentent par ailleurs deux inconvénients rapport aux méthodes économétriques traditionnelles. Premièrement, bien qu'il existe désormais de multiples approches permettent d'interpétrer partiellement les modèles ensemblistes, leur interprétabilité reste moindre que celle d'une régression linéaire ou logistique. Deuxièmement, les modèles ensemblistes sont plus complexes que les approches traditionnelles, et leurs hyperparamètres doivent faire l'objet d'une optimisation, par exemple au travers d'une validation croisée. Ce processus d'optimisation est généralement plus complexe et plus long que l'estimation d'une régression linéaire ou logistique. En revanche, les méthodes ensemblistes sont relativement simples à prendre en main, et ne requièrent pas nécessairement une puissance de calcul importante.

i Et par rapport au *deep learning*?

Si les approches de *deep learning* sont sans conteste très performantes pour le traitement du langage naturel, des images et du son, leur supériorité n'est pas établie pour les applications reposant sur des données tabulaires. Les comparaisons disponibles dans la littérature concluent en effet que les méthodes ensemblistes à base d'arbres sont soit plus performantes que les approches de *deep learning* (L. Grinsztajn, E. Oyallon, and G. Varoquaux [1], R. Shwartz-Ziv and A. Armon [2]), soit font jeu égal avec elles (D. McElfresh *et al.* [3]). Ces études ont identifié trois avantages des méthodes ensemblistes: elles sont peu sensibles aux variables explicatives non pertinentes, robustes aux valeurs extrêmes des variables explicatives, et capables d'approximer des fonctions très irrégulières. De plus, dans la pratique les méthodes ensemblistes sont souvent plus rapides à entraîner et moins gourmandes en ressources informatiques, et l'optimisation des hyperparamètres s'avère souvent moins complexe (R. Shwartz-Ziv and A. Armon [2]).

2.3. Comment fonctionnent les méthodes ensemblistes?

Ce paragraphe présente d'abord le modèle de base sur lesquelles sont construites les méthodes ensemblistes à base d'arbres: l'arbre de classification et de régression (CART) (Section 2.3.1). Bien que simples et intuitifs, les arbres CART sont souvent insuffisants en termes de performance lorsqu'ils sont utilisés isolément.

Elle introduit ensuite les **deux grandes familles de méthodes ensemblistes** décrites dans ce document: le *bagging* et les forêts aléatoires (Section 2.3.2), et le *gradient boosting* (Section 2.3.3).

2.3.1. Le modèle de base: l'arbre de classification et de régression

2.3.1.1. Qu'est-ce qu'un arbre CART?

Le modèle de base des méthodes ensemblistes est souvent un arbre de classification et de régression (CART, L. Breiman, J. Friedman, R. Olshen, and C. Stone [4]). Un arbre CART est un algorithme prédictif qui traite un problème de prédiction complexe en le décomposant en une série de décisions simples, organisées de manière hiérarchique. Ces décisions permettent de segmenter progressivement les données en régions homogènes au sein desquelles il est plus simple de faire des prédictions. Il s'agit d'un outil puissant pour explorer les relations entre les variables explicatives et la variable cible, sans recourir à des hypothèses *a priori* sur la forme de cette relation.

Trois caractéristiques essentielles définissent un arbre CART:

- L'arbre partitionne l'espace des variables explicatives X en régions (appelées feuilles ou *leaves*) les plus homogènes possible, au sens d'une mesure de l'hétérogénéité (par exemple, l'entropie ou l'erreur quadratique moyenne). Ces divisions vont permettre de regrouper des observations similaires pour faciliter la prédiction;
- Chaque région est définie par un ensemble de conditions, appelées règles de décision (*splitting rules* ou *decision rules*), appliquées successivement sur les variables explicatives. Par exemple, une première règle pourrait poser la question : "L'individu est-il en emploi ?", et subdiviser les données en deux groupes (oui/non). Une deuxième règle pourrait alors affiner la segmentation en posant la question : "L'individu est-il diplômé du supérieur ?". Une région spécifique serait ainsi définie par la condition combinée : "l'individu est en emploi et est diplômé du supérieur".
- Une fois l'arbre construit, chaque feuille produit une prédiction en se basant sur les données de la région correspondante. En classification, la prédiction est généralement la classe la plus fréquente parmi les observations de la région. En régression, la prédiction est souvent la moyenne des valeurs observées dans la région.

Deux conséquences importantes découlent de cette construction:

- L'algorithme CART ne fait **aucune hypothèse *a priori*** sur la relation entre les variables explicatives $\mathbf{X}$ et la variable cible y. C'est une différence majeure avec les modèles économétriques standards, tels que la régression linéaire qui suppose une relation linéaire de la forme $E(y) = \mathbf{X}\boldsymbol{\beta}$.
- **L'arbre final est une fonction constante par morceaux**: la prédiction est **identique** pour toutes les observations situées dans la même région; elle ne peut varier qu'entre régions.

Illustration, et représentation graphique (sous forme d'arbre et de graphique).

2.3.1.2. Avantages et limites des arbres CART

Les arbres CART présentent plusieurs avantages: leur principe est simple, ils sont aisément interprétables et peuvent faire l'objet de représentations graphiques intuitives. Par ailleurs, la flexibilité offerte par le partitionnement récursif assure que les arbres obtenus reflètent les corrélations observées dans les données d'entraînement.

Ils souffrent néanmoins de deux limites. D'une part, les arbres CART ont souvent un **pouvoir prédictif faible** qui en limite l'usage. D'autre part, ils sont **peu robustes et instables**: on dit qu'ils présentent une **variance élevée**. Ainsi, un léger changement dans les données (par exemple l'ajout ou la suppression de quelques observations) peut entraîner des modifications significatives dans la structure de l'arbre et dans la définition des régions utilisées pour la prédiction (feuilles). Les arbres CART sont notamment sensibles aux valeurs extrêmes, aux points aberrants et au bruit statistique. De plus, les prédictions des arbres CART sont sensibles à de petites fluctuations des données d'échantillonnage: celles-ci peuvent aboutir à ce qu'une partie des observations change brutalement de feuille et donc de valeur prédite.

Ces limites motivent l'utilisation des deux familles de méthodes ensemblistes présentées dans la suite (le *bagging*, dont la *random forests*, et le *gradient boosting*), qui s'appuient sur un grand nombre d'arbres pour accroître à la fois la précision et la stabilité des prédictions. La différence essentielle entre ces deux familles portent sur la façon dont les arbres sont entraînés.

i Les familles de méthodes ensemblistes

Les méthodes ensemblistes basées sur des arbres de décision se répartissent en **deux grandes familles**, qui se distinguent selon la manière dont les modèles de base sont construits. Lorsque les modèles de base sont entraînés en parallèle et indépendamment les uns des autres, on parle de *bagging* (*Bootstrap Aggregating*). La *forêt aléatoire* (*random forest*) est une variante particulièrement performante du *bagging*. Lorsque les modèles de base sont *entraînés de manière séquentielle*, chaque modèle visant à corriger les erreurs des modèles précédents, on parle de *boosting*. Ce document aborde essentiellement le *gradient boosting*, qui est l'approche de *boosting* la plus utilisée actuellement.

2.3.2. Le *bagging* (Bootstrap Aggregating) et les forêts aléatoires

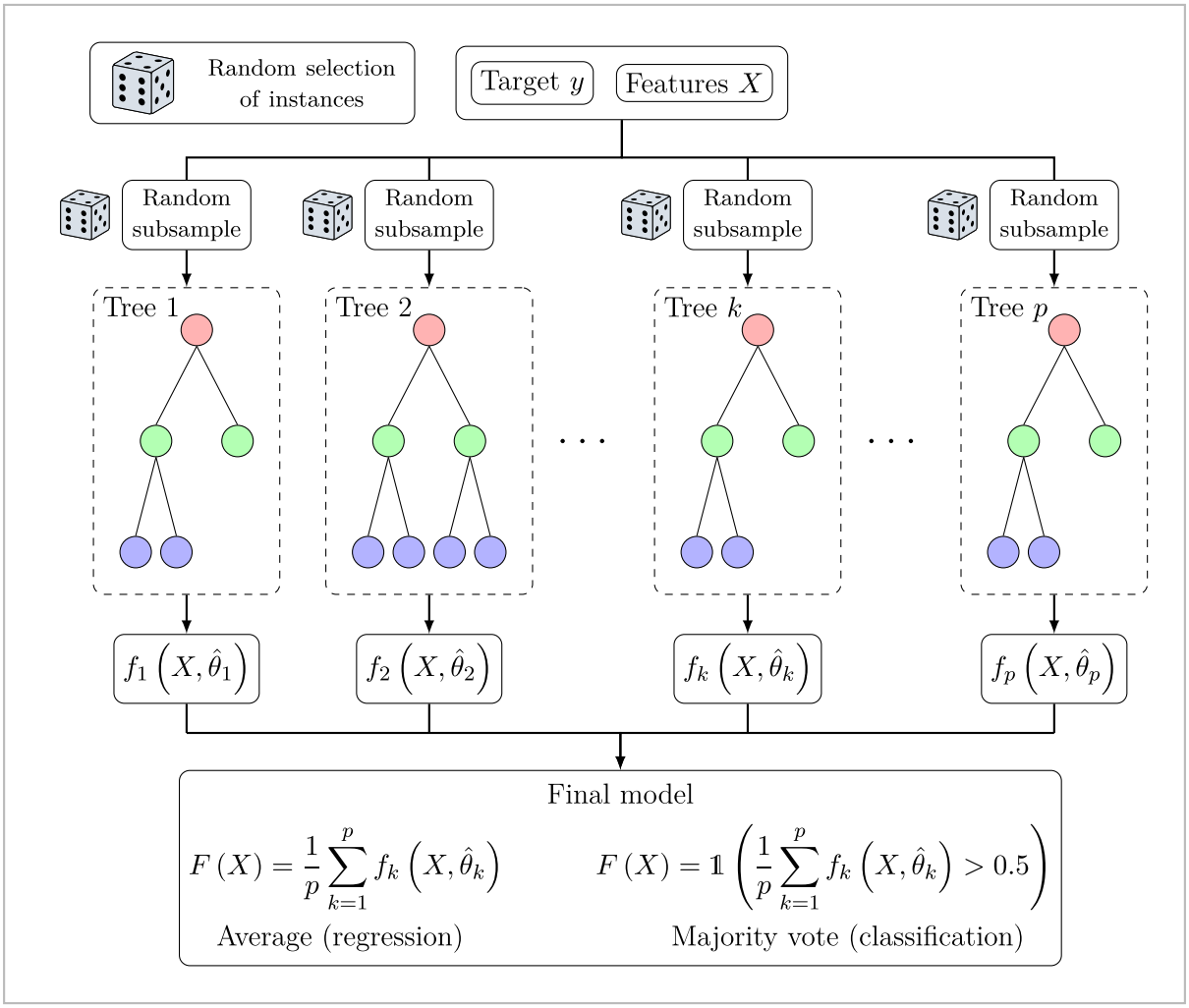
2.3.2.1. Le *bagging*

Le *bagging* (Bootstrap Aggregating) est une méthode ensembliste qui repose sur l'agrégation des prédictions de plusieurs modèles individuels, entraînés indépendamment les uns des autres, pour construire un modèle global plus performant (L. Breiman [5]). Cette approche constitue également le socle des forêts aléatoires, qui en sont une version améliorée.

Le *bagging* offre deux avantages majeurs par rapport aux arbres de décision CART : une meilleure capacité prédictive et une plus grande stabilité des prédictions. Cette amélioration découle de la stratégie d'entraînement. Au lieu d'entraîner un seul modèle sur l'ensemble des données, le bagging procède en trois étapes principales:

- **Tirage de sous-échantillons aléatoires**: À partir du jeu de données initial, plusieurs sous-échantillons sont générés par échantillonnage aléatoire avec remise (*bootstrapping*). Chaque sous-échantillon a la même taille que le jeu de données original, mais peut contenir des observations répétées, tandis que d'autres peuvent être omises.
- **Entraînement parallèle**: Un arbre est entraîné sur chaque sous-échantillon de manière indépendante. Ces arbres sont habituellement assez complexes et profonds.
- **Agrégation des prédictions**: Les prédictions des modèles sont combinées pour produire le résultat final. En classification, la prédiction finale est souvent déterminée par un vote majoritaire, tandis qu'en régression, elle correspond généralement à la moyenne des prédictions.

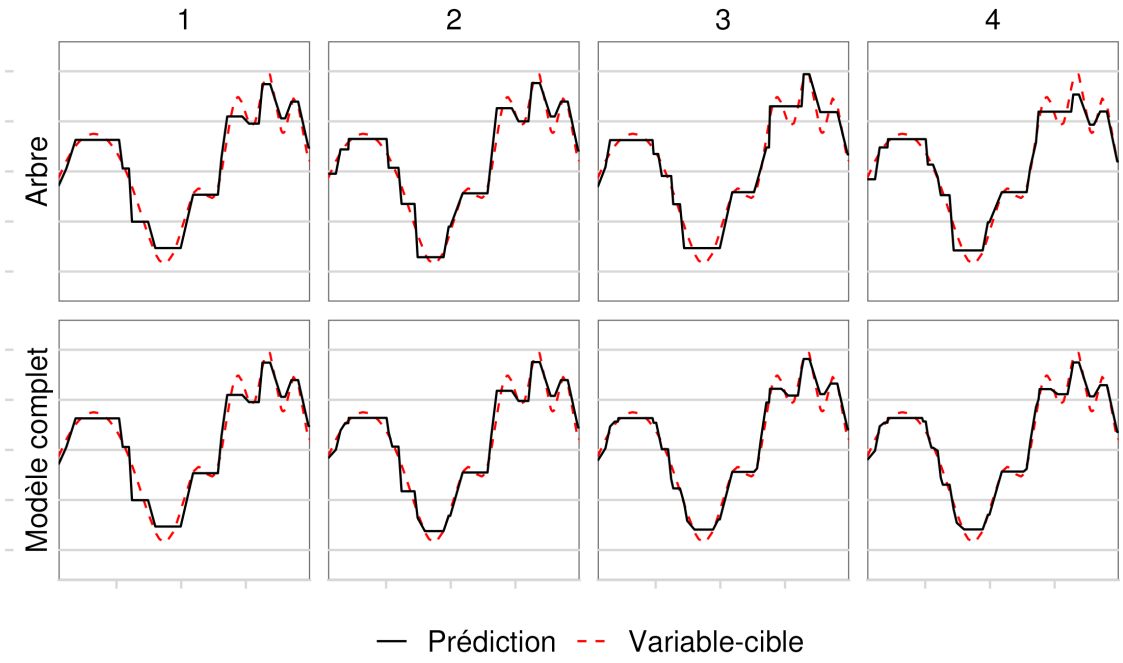
Figure 1 : Représentation schématique d'un algorithme de *bagging*



La Figure 1 propose une représentation schématique du *bagging*: d'abord, des sous-échantillons sont générés aléatoires avec remise à partir du jeu de données d'entraînement. Ensuite, des arbres de décision sont entraînés indépendamment sur ces sous-échantillons. Enfin, leurs prédictions sont agrégées pour obtenir les prédictions finales. On procède généralement au vote majoritaire (la classe prédite majoritairement par les arbres) dans un problème de classification, et à la moyenne dans un problème de régression.

L'efficacité du *bagging* provient de la réduction de la variance qui est permise par l'agrégation des prédictions. Chaque arbre est entraîné sur un sous-échantillon légèrement différent, sujet à des fluctuations aléatoires. L'agrégation des prédictions (par moyenne ou vote majoritaire) de tous les arbres réduit la sensibilité du modèle final aux fluctuations des données d'entraînement. Le modèle final est ainsi plus robuste et plus précis que chacun des arbres pris individuellement.

Figure 2 : Illustration d'un algorithme de *bagging* en une dimension



La figure Figure 2 illustre le fonctionnement du *bagging* sur un exemple simple en une dimension. On peut lire sur le panneau du haut la prédiction de chacun des quatre premiers arbres du modèle. Sur le panneau du bas est représentée la prédiction du modèle complet, qui comprend un seul arbre pour la figure la plus à gauche, et quatre arbres pour la figure la plus à droite. Cette figure permet de constater trois caractéristiques du *bagging*: chaque arbre est un modèle à part entière, chaque arbre produit une prédiction proche mais différente de celles des autres arbres, et l'agrégation de ces différentes prédictions permet d'améliorer les performances prédictives du modèle d'ensemble.

Malgré ses avantages, le *bagging* souffre d'une limite importante qui provient de la **corrélation entre les arbres**. En effet, malgré le tirage aléatoire des sous-échantillons, les arbres présentent souvent des structures similaires, car les règles de décision sous-jacentes restent généralement assez proches. Cette corrélation réduit l'efficacité de l'agrégation et limite les gains en performance.

Pour réduire cette corrélation entre arbres, les forêts aléatoires introduisent une étape supplémentaire de randomisation. Leur supériorité prédictive explique pourquoi le *bagging* seul est rarement utilisé en pratique. Néanmoins, les forêts aléatoires tirent leur efficacité des principes fondamentaux du *bagging*.

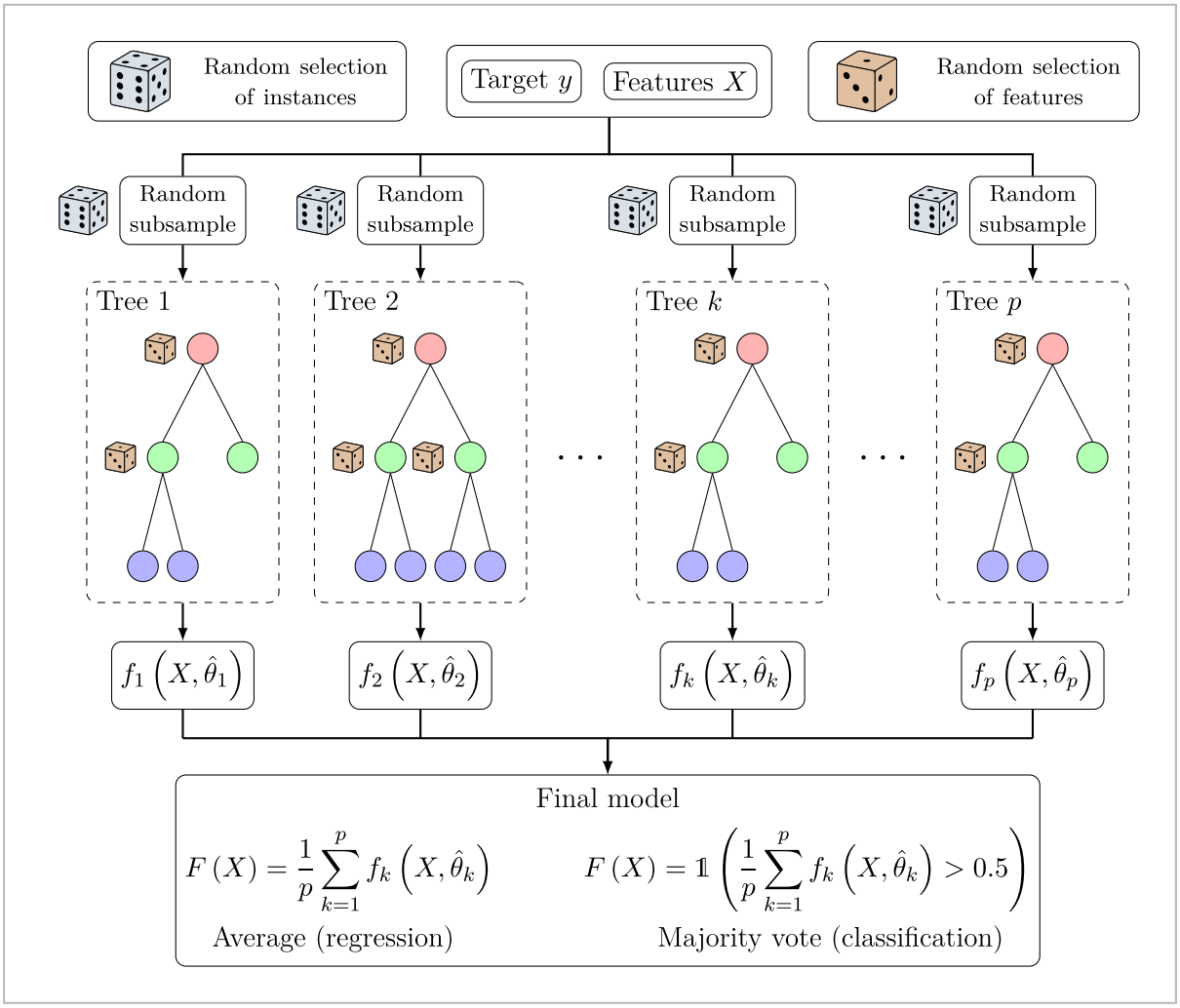
2.3.2.2. Les forêts aléatoires (*random forests*)

Les forêts aléatoires (*random forests*, L. Breiman [6]) sont une variante du *bagging* qui vise à produire des modèles très performants en conciliant deux objectifs: maximiser le pouvoir

prédictif des arbres pris isolément, et minimiser la corrélation entre ces arbres (le problème inhérent au *bagging*).

Pour atteindre ce second objectif, la forêt aléatoire introduit une nouvelle source de randomisation: la **sélection aléatoire de variables**. Lors de la construction de chaque arbre, au lieu d'utiliser toutes les variables disponibles pour déterminer la meilleure séparation à chaque nœud, un sous-ensemble aléatoire de variables est sélectionné. En limitant la quantité d'information à laquelle chaque arbre a accès au moment de chaque nouvelle division, cette étape supplémentaire contraint mécaniquement les arbres à être plus diversifiés (car deux arbres ne pourront plus nécessairement choisir les mêmes variables pour les mêmes séparations). Cela réduit significativement la corrélation entre les arbres, améliorant ainsi l'efficacité de l'agrégation. L'ensemble des prédictions devient ainsi plus précis et moins sujet aux fluctuations aléatoires.

Figure 3 : Représentation schématique d'un algorithme de forêt aléatoire



La figure Figure 3 propose une représentation schématique d'une forêt aléatoire. La logique d'ensemble reste la même que celle du *bagging*. L'échantillonnage *bootstrap* est inchangé, mais

l'étape de construction de chaque arbre est modifiée pour n'utiliser, à chaque nouvelle division, qu'un sous-ensemble aléatoire de variables. L'agrégation des prédictions se fait ensuite de la même manière que pour le *bagging*.

Le principal enjeu de l'entraînement d'une forêt aléatoire est de trouver le bon arbitrage entre puissance prédictive des arbres individuels (que l'on souhaite maximiser) et corrélation entre les arbres (que l'on souhaite minimiser). L'optimisation des hyper-paramètres des forêts aléatoires (dont le plus important est le nombre de variables sélectionnées à chaque noeud) vise précisément à choisir le meilleur compromis possible entre pouvoir prédictif invividuel et diversité des arbres.

Les forêts aléatoires sont très populaires car elles sont faciles à implémenter, peu sensibles aux hyperparamètres (elles fonctionnent bien avec les valeurs par défaut de la plupart des implémentations proposées en R ou en Python), et offrent de très bonnes performances dans de nombreux cas. Cependant, comme toute méthode d'apprentissage automatique, elles restent sujettes au surapprentissage (voir encadré), bien que dans une moindre mesure par rapport à d'autres techniques comme le *gradient boosting*.

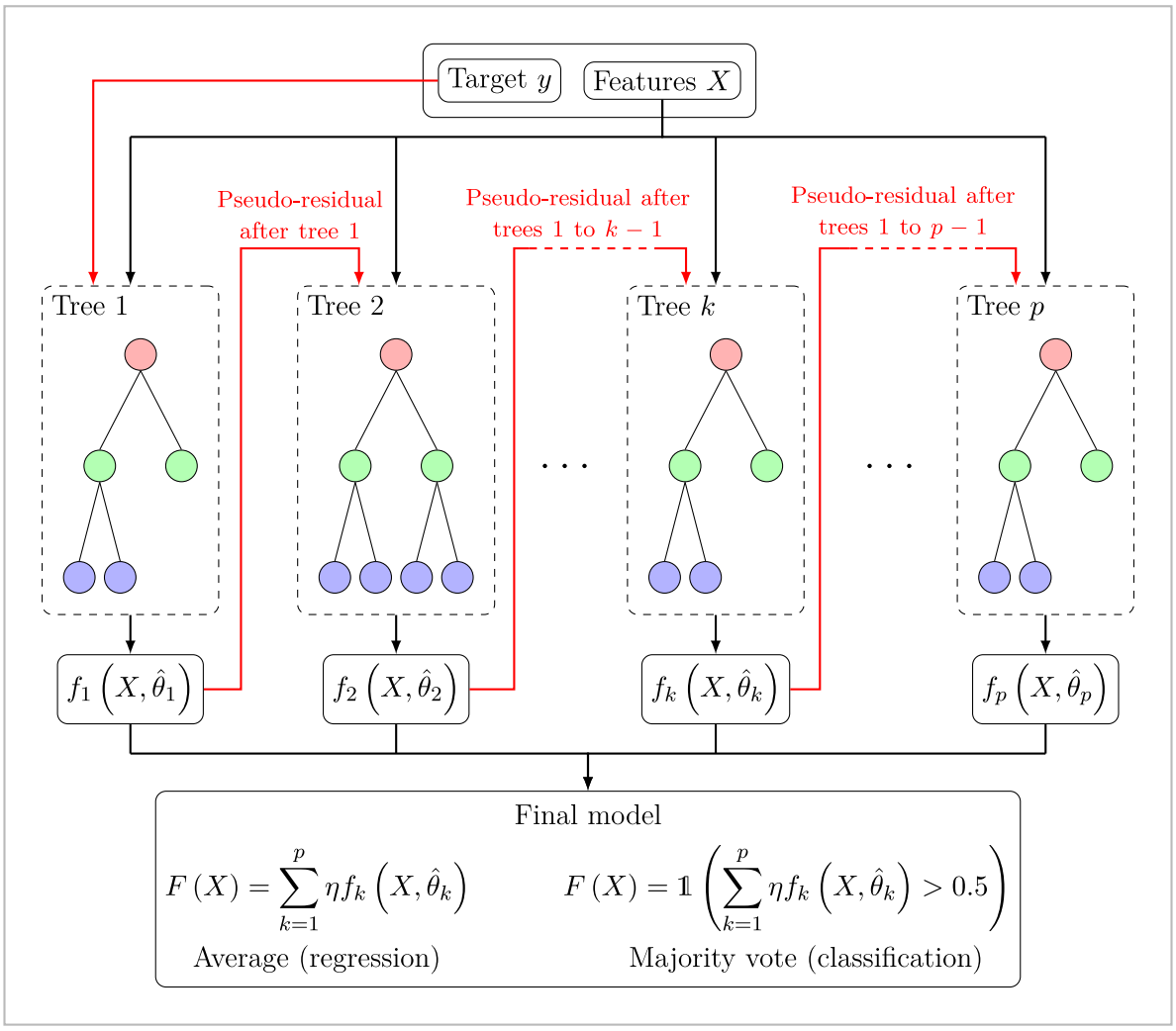
i Qu'est-ce que le surapprentissage?

Le surapprentissage (*overfitting*) est un phénomène fréquent en *machine learning* où un modèle apprend non seulement les relations sous-jacentes entre la variable cible et les variables explicatives, mais également le bruit présent dans les données d'entraînement. En capturant ces fluctuations aléatoires plutôt que les tendances générales, le modèle affiche une performance excellente mais trompeuse sur les données d'entraînement, et s'avère médiocre sur des données nouvelles ou de test, car il ne parvient pas à généraliser efficacement.

2.3.3. Le *gradient boosting*

Contrairement aux forêts aléatoires qui combinent des arbres de décision complexes et indépendants, le *gradient boosting* construit un ensemble d'arbres plus simples et entraînés de manière séquentielle. Chaque arbre vise à corriger les erreurs commises par les arbres précédents, améliorant progressivement la précision du modèle global. Cette approche repose sur des fondements théoriques très différents de ceux du *bagging*.

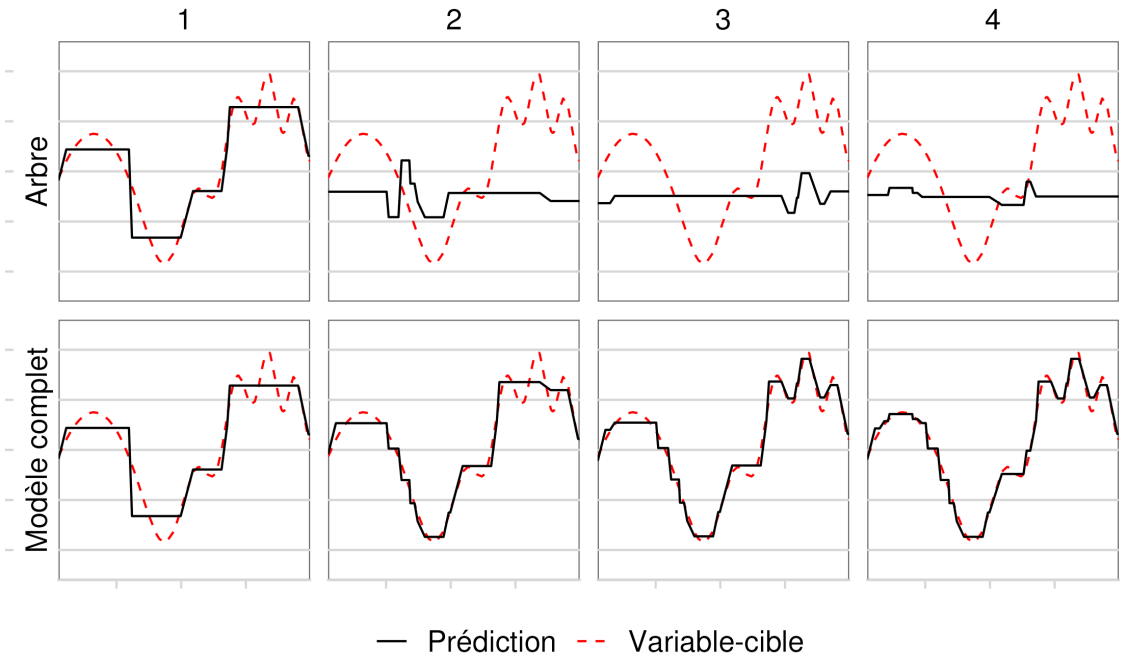
Figure 4 : Représentation schématique d'un algorithme de *gradient boosting*



La logique du *gradient boosting* est illustrée par la figure Figure 4:

- Un premier modèle simple et peu performant est entraîné sur les données.
- Un deuxième modèle est entraîné de façon à corriger les erreurs du premier modèle (par exemple en pondérant davantage les observations mal prédites);
- Ce processus est répété en ajoutant des modèles simples, chaque modèle corrigeant les erreurs commises par l'ensemble des modèles précédents;
- Tous ces modèles sont finalement combinés (souvent par une somme pondérée) pour obtenir un modèle complexe et performant.

Figure 5 : Illustration d'un algorithme de *boosting* en une dimension



La figure Figure 5 illustre le fonctionnement du *boosting* sur un exemple simple en une dimension. On peut lire sur le panneau du haut la prédiction de chacun des quatre premiers arbres du modèle. Sur le panneau du bas est représentée la prédiction du modèle complet, qui comprend un seul arbre pour la figure la plus à gauche, et quatre arbres pour la figure la plus à droite. Cette figure permet de constater trois caractéristiques du *boosting*: chaque arbre n'est pas interprétable en lui-même car il dépend des arbres précédents, chaque arbre produit une prédiction qui tâche de corriger les erreurs de l'ensemble des arbres précédents, et l'agrégation de ces différentes prédictions permet d'obtenir un modèle très performant, qui reflète précisément les données sur lesquelles il est entraîné.

Le *gradient boosting* offre des performances élevées mais exige une attention particulière portée sur la configuration des hyperparamètres et sur la prévention du surapprentissage. En particulier, les hyperparamètres sont nombreux et, contrairement aux forêts aléatoires, nécessitent un ajustement minutieux pour obtenir des résultats optimaux. Une mauvaise configuration peut conduire à des performances médiocres ou à un surapprentissage. L'utilisation du *gradient boosting* nécessite donc une bonne connaissance du fonctionnement des algorithmes. En outre, les algorithmes de *gradient boosting* peuvent être sensibles au bruit dans les données et aux erreurs dans la variable cible. Un prétraitement rigoureux des données est donc essentiel. Enfin, une validation rigoureuse sur un jeu de données de test indépendant (non utilisé pendant l'entraînement) est indispensable pour évaluer la qualité du modèle obtenu par *gradient boosting*.

2.4. Comparaison entre forêts aléatoires et *gradient boosting*

Les forêts aléatoires et le *gradient boosting* paraissent très similaires au premier abord: il s'agit de deux approches ensemblistes, qui construisent des modèles très prédictifs performants en combinant un grand nombre d'arbres de décision. Mais en réalité, ces deux approches présentent plusieurs différences fondamentales:

- Les deux approches reposent sur des **fondements théoriques différents**: la loi des grands nombres pour les forêts aléatoires, la théorie de l'apprentissage statistique pour le *boosting*.
- **Les arbres n'ont pas le même statut dans les deux approches**. Dans une forêt aléatoire, les arbres sont entraînés indépendamment les uns des autres et constituent chacun un modèle à part entière, qui peut être utilisé, représenté et interprété isolément. Dans un modèle de *boosting*, les arbres sont entraînés séquentiellement, ce qui implique que chaque arbre n'a pas de sens indépendamment de l'ensemble des arbres qui l'ont précédé dans l'entraînement. Par ailleurs, les arbres d'une forêt aléatoire sont relativement complexes et profonds (car ce sont des modèles à part entière), alors que dans le *boosting* les arbres sont plus souvent simples et peu profonds.
- Les **points d'attention lors de l'entraînement** des algorithmes sont différents: l'enjeu principal de l'entraînement d'une forêt aléatoire est trouver le bon arbitrage entre puissance prédictive des arbres et corrélation entre arbres, tandis que l'entraînement d'un algorithme de *gradient boosting* porte davantage sur la lutte contre le surapprentissage.
- **Complexité d'usage**: les forêts aléatoires s'avèrent plus faciles à prendre en main que le *gradient boosting*, car elles comprennent moins d'hyperparamètres et leur optimisation est moins complexe.
- **Conditions d'utilisation**: il est possible d'évaluer la qualité d'une forêt aléatoire en utilisant les données sur lesquelles elle a été entraînée grâce à l'approche *out-of-bag*, alors que c'est impossible avec le *gradient boosting*, pour lequel il faut impérativement conserver un ensemble de test. Cette différence peut sembler purement technique en apparence, mais elle s'avère importante en pratique dans de nombreuses situations, par exemple lorsque les données disponibles sont de taille restreinte, lorsque le modèle doit être utilisé sur les données sur lesquelles il est entraîné (pour repondérer une enquête par exemple) ou lorsque les ressources informatiques disponibles ne sont pas suffisantes pour mener un exercice de validation croisée.

2.4.1. Quelle approche choisir?

De façon générale, le point de départ recommandé est d'entraîner une forêt aléatoire avec les hyperparamètres par défaut, puis d'optimiser ces hyperparamètres, et enfin de se tourner vers le *gradient boosting* lorsque les performances de la forêt aléatoire ne sont pas suffisantes, ou lorsqu'elle est inadaptée au cas d'usage.

Principe: cette partie propose une présentation formalisée des méthodes ensemblistes, à destination des personnes souhaitant comprendre en détail le fonctionnement et les propriétés des algorithmes.

3. Les arbres de décision

Les arbres de décision désignent un éventail d'algorithmes de *machine learning*, utilisés notamment pour des tâches de classification et de régression. Ces algorithmes constituent la brique élémentaire des méthodes ensemblistes à base d'arbres (forêt aléatoire et *gradient boosting*). Cette section a pour objectif de présenter ce qu'est un arbre de décision, sa structure et la terminologie associée (Section 3.1), puis de détailler la méthode de construction des arbres par l'algorithme CART (Section 3.2).

3.1. Le principe fondamental: partitionner pour prédire

Le principe des arbres de décision consiste à **diviser l'espace des caractéristiques en sous-régions homogènes à l'aide de règles simples**, puis de former pour chaque sous-région une prédiction à partir des observations présentes dans cette sous-région. Imaginons par exemple que l'on souhaite prédire le prix d'une maison en fonction de sa superficie et de son nombre de pièces, à partir d'un ensemble de transactions pour lesquelles le prix est connu. L'espace des caractéristiques (superficie et nombre de pièces) est vaste, et les prix des maisons (la *réponse* à prédire) sont très variables. L'idée centrale des arbres de décision est de diviser cet espace en zones plus petites, au sein desquelles les maisons ayant des surfaces et un nombre de pièces similaire ont des prix proches, et d'attribuer une prédiction identique à toutes les maisons situées dans la même zone. Malgré cette apparente simplicité, les arbres de décision sont puissants et capables de modéliser des interactions complexes et non linéaires entre les variables d'un jeu de données.

3.1.1. Les défis du partitionnement optimal

L'objectif principal est de trouver la partition de l'espace des caractéristiques qui offre les meilleures prédictions possibles. Cependant, cet objectif se heurte à plusieurs difficultés, et la complexité du problème augmente rapidement avec le nombre de caractéristiques et la taille de l'échantillon:

- **Infinité des découpages possibles**: Il existe une infinité de façons de diviser l'espace des caractéristiques;
- **Complexité de la paramétrisation**: Il est difficile de représenter tous ces découpages avec un nombre limité de paramètres;
- **Optimisation complexe**: Même avec une paramétrisation, trouver le meilleur découpage nécessite une optimisation complexe, souvent irréaliste en pratique.

3.1.2. Les solutions apportées par les arbres de décision

Pour surmonter ces difficultés, les algorithmes d'arbres de décision, et notamment le plus célèbre, l'algorithme CART (Classification And Regression Tree, L. Breiman, J. Friedman, R. Olshen, and C. Stone [4]), procèdent à trois simplifications cruciales:

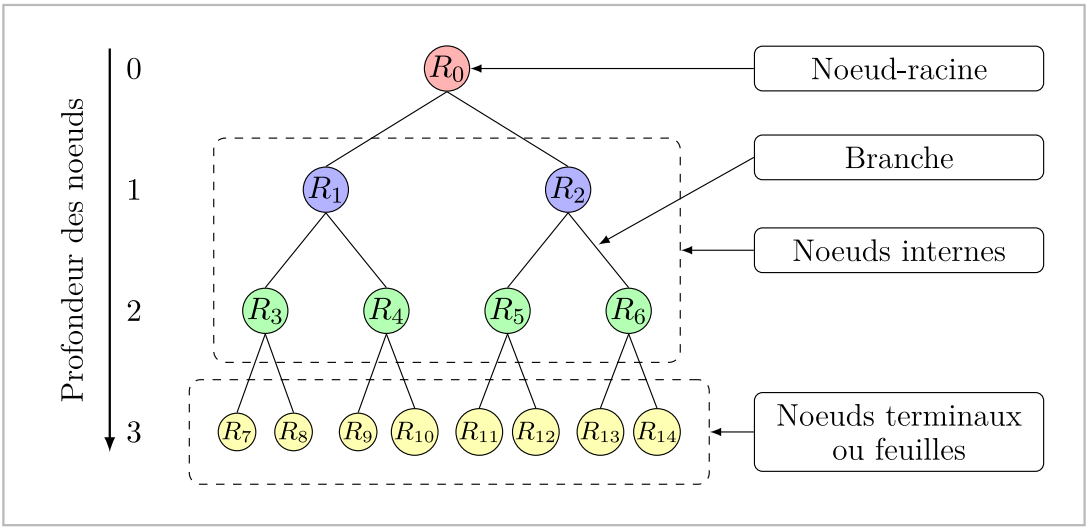
1. **Optimisation gloutonne (greedy optimization)**: plutôt que de rechercher d'emblée un partitionnement optimal, les arbres de décision partitionnent l'espace selon une approche séquentielle. A chaque étape, l'arbre choisit la meilleure division possible d'une région en deux sous-régions, *indépendamment des étapes précédentes ou suivantes*. Ce processus est répété pour chaque sous-région, ce qui permet d'affiner progressivement le partitionnement de l'espace, jusqu'à ce qu'un critère d'arrêt soit atteint. Cette méthode dite "gloutonne" (*greedy*) s'avère très efficace, car elle décompose un problème d'optimisation complexe en une succession de problèmes plus simples et plus rapides à résoudre. Le résultat obtenu n'est pas nécessairement un optimum global, mais il s'en approche raisonnablement et surtout rapidement.
2. **Simplification des règles de partitionnement**: au lieu d'explorer tous les règles de décision possibles, les arbres de décision se restreignent à des règles de décision très simples, appelés **découpages binaires** (*binary splits*): à chaque étape, l'algorithme divise chaque région de l'espace en deux sous-régions à l'aide d'une règle de décision (*decision rule*) qui ne fait appel qu'à **une seule caractéristique** (ou *variable*) et à **un seul seuil** (ou *critère*) pour cette segmentation. Cela revient à poser une question simple telle que: "La valeur de la caractéristique X dépasse-t-elle le seuil x ?" Par exemple: "La superficie de la maison est-elle supérieure à 100 m^2 ?". Les deux réponses possibles ("Oui" ou "Non") définissent deux nouvelles sous-régions distinctes de l'espace, chacune correspondant à un sous-ensemble de données plus homogènes.
3. **Simplicité des prédictions locales**: une fois le partitionnement réalisé, une prédiction est calculée pour chaque région à partir des observations des données d'entraînement présentes dans cette région. Il s'agit souvent de la moyenne des valeurs cibles dans cette région (régression) ou de la classe majoritaire (classification). Un point essentiel est que la prédiction est constante au sein de chaque région.

3.1.3. Terminologie et structure d'un arbre de décision

Cet algorithme est appelé **arbre de décision** (*decision tree*) en raison provient de la structure arborescente en forme d'arbre inversé qui apparaît lorsqu'on en fait une représentation graphique (voir figure Figure 6). Plus généralement, les principaux éléments qui composent les arbres de décision sont désignés par des termes issus du champ lexical des arbres:

- **Nœud Racine (*Root Node*)**: Le nœud-racine est le point de départ de l'arbre de décision, il est situé au sommet de l'arbre. Il contient l'ensemble des données d'entraînement avant tout partitionnement. À ce niveau, l'algorithme cherche la caractéristique la plus discriminante, c'est-à-dire celle qui permet de diviser les données en deux régions de manière à minimiser un certain critère d'hétérogénéité (comme l'indice de Gini pour la classification ou la variance pour la régression).
- **Nœuds Internes (*Internal Nodes*)**: Les nœuds internes sont les points intermédiaires où l'algorithme CART applique des règles de décision pour diviser les données en sous-régions plus petites. Chaque nœud interne se définit par une **règle de décision** basée sur une variable et un seuil, qui sert à opérer un partitionnement des données. Le plus souvent, la règle de décision est exprimée sous la forme d'une inégalité telle que `x_3 <= 7`. Chaque nœud interne a la fois un **nœud-parent** (*parent node*) dont il constitue une sous-région et deux **nœuds-enfants** (*child nodes*) qui le partitionnent. Le noeud de gauche regroupe les observations pour lesquelles la règle de décision est vérifiée (valeurs basses); le noeud de droite regroupe les observations pour lesquelles la règle de décision n'est pas vérifiée (valeurs élevées).
- **Branches (*Branches*)**: Les branches sont les connexions entre les nœuds et représentent le chemin suivies par les données. Chaque branche correspond à une décision binaire, "Oui" ou "Non", qui oriente les observations vers une nouvelle subdivision de l'espace des caractéristiques.
- **Nœuds Terminaux ou Feuilles (*Leaf Nodes*, *Terminal Nodes* ou *Leaves*)**: Les nœuds terminaux, situés à l'extrémité des branches, sont les points où le processus de division s'arrête. Ils fournissent la prédiction finale. Dans un problème de classification, la prédiction d'une feuille est soit la classe majoritaire parmi les observations de la feuille (par exemple, "Oui" ou "Non"), soit une probabilité d'appartenir à chaque classe. Dans un problème de régression, la prédiction d'une feuille est une valeur numérique, souvent la moyenne des observations de la feuille.
- **Profondeur** (*Depth*): La profondeur d'un arbre de décision correspond à la longueur du chemin le plus long entre le nœud-racine et une feuille. Le nœud-racine est situé par définition à la profondeur 0, et chaque niveau supplémentaire de l'arbre ajoute une unité à la profondeur. La profondeur totale de l'arbre est donc le nombre maximal de décisions (ou de nœuds internes) à traverser pour passer du nœud-racine à une feuille.

Figure 6 : Structure d'un arbre de décision



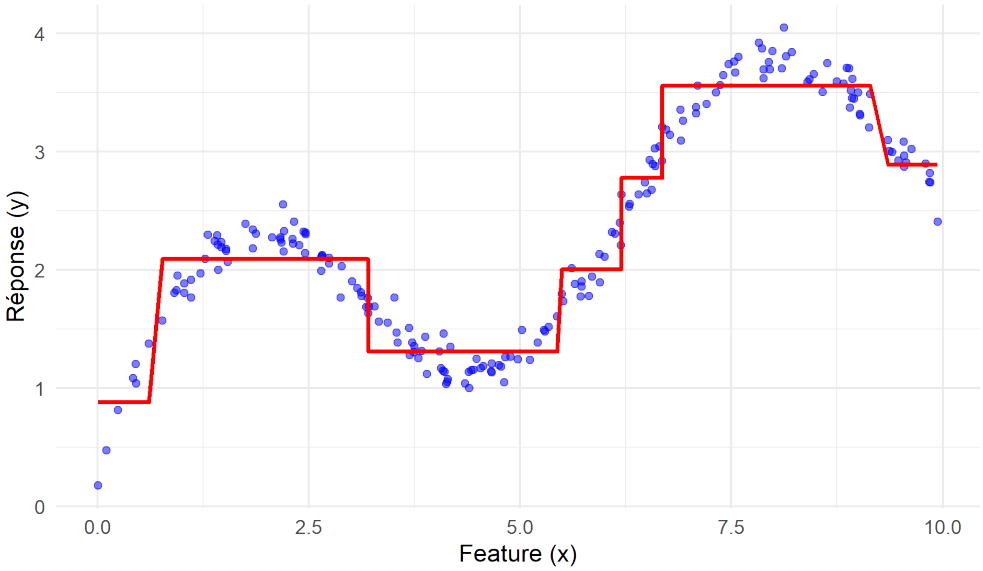
3.1.4. Propriétés des arbres de décision

Les arbres de décision ont plusieurs propriétés qui contribuent à leur puissance prédictive et facilitent leur usage en pratique:

- **Les arbres de décision ne font aucune hypothèse *a priori* sur la relation entre les variables explicatives et la variable-cible**. C'est une différence majeure avec les modèles économétriques standards, tels que la régression linéaire qui suppose une relation linéaire de la forme $E(y) = \mathbf{X}\boldsymbol{\beta}$.
- **Un arbre de décision est une fonction constante par morceaux**: la prédiction est **identique** pour toutes les observations situées dans la même région, et ne peut varier qu'entre régions[1]. Une conséquence de cette propriété est qu'**un arbre de décision peut capter sans difficultés les non-linéarités** dans la relation entre la variable-cible et les variables numériques (voir la figure Figure 7). Il est donc inutile d'inclure des variables supplémentaires telles que le carré ou le cube des variables continues.

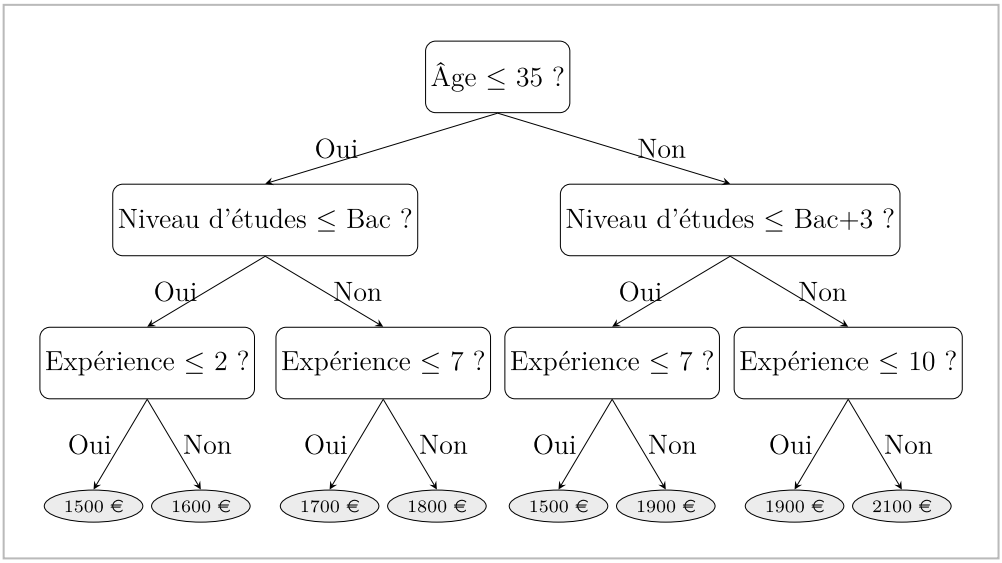
[1]Il existe néanmoins des variantes d'arbres de décision où la prédiction n'est pas constante au sein de chaque feuille, mais elles sont peu courantes en pratique et ne sont pas couvertes par le présent document.

Figure 7 : Arbre de décision et non-linéarité



- **Les arbres de décision sont par construction capables de capter des interactions entre variables explicatives sans qu'il soit nécessaire de les spécifier explicitement**. En effet, la prédiction pour une observation dépend de la combinaison des différentes variables intervenant dans les règles de décision qui mènent à la feuille terminale, ce qui traduit une interaction entre les variables. La figure Figure 8 illustre ces interactions avec un arbre de décision qui prédit le salaire en fonction de l'âge, du niveau d'étude, et de l'expérience. On voit d'une part que les feuilles terminales sont définies par la conjonction de règles de décision qui font intervenir ces trois variables, avec des seuils différents selon les branches de l'arbre, et d'autre part que l'effet d'une caractéristique sur le salaire dépend des autres caractéristiques de l'individu. Par exemple, une augmentation de l'expérience de 7 à 8 années se traduira une augmentation de salaire de 100 € si l'individu a moins de 35 ans et un niveau d'études inférieur au bac, par une augmentation de 200 € s'il a plus de 35 ans et un niveau d'étude supérieur à Bac+ 3, et sera sans effet sur le salaire dans les autres cas.

Figure 8 : Interactions dans un arbre de décision



- **Dans un arbre de décision, les valeurs prises par les variables numériques (par exemple l'âge) n'ont pas d'importance par elles-mêmes, c'est l'*ordre* de ces valeurs qui est essentiel**. Ainsi, dans la règle de décision "L'âge est-il inférieur à 30 ans?", ce n'est pas la valeur "30" qui importe par elle-même, c'est le fait qu'elle **sépare les observations** en deux groupes, selon que l'âge est inférieur ou supérieur à 30 ans. Cette propriété a pour conséquence que les arbres de décision sont insensibles aux modifications strictement monotones des variables continues. Par exemple, remplacer l'âge par l'âge au carré ne changera rien à l'arbre de décision, car les règles de décision "L'âge est-il inférieur à 30 ans?" et "L'âge au carré est-il inférieur à 900?" sont strictement équivalentes (car elles définissent les mêmes groupes).

3.2. La construction d'un arbre de décision par l'algorithme CART

Depuis les années 1980, de multiples algorithmes ont été proposés pour construire des arbres de décision, notamment CART (L. Breiman, J. Friedman, R. Olshen, and C. Stone [4]), C4.5 (J. R. Quinlan [7]) et MARS (J. H. Friedman [8]). La présente section présente la méthode de construction et l'utilisation d'un arbre de décision par l'algorithme CART. Cette méthode comprend quatre étapes:

- Choisir une mesure d'impureté adaptée au problème;
- Construire l'arbre de décision par un partitionnement séquentiel;
- Élaguer l'arbre de décision;
- Utiliser l'arbre pour prédire.

3.2.1. Définir une mesure d'impureté adaptée au problème

La **mesure d'impureté** quantifie l'hétérogénéité des observations au sein d'un nœud par rapport à la variable cible (classe pour la classification, ou valeur continue pour la régression). Plus précisément, une mesure d'impureté est conçue pour croître avec la dispersion dans un nœud: plus un nœud est homogène, plus son impureté est faible. Un nœud est dit **pur** lorsque toutes les observations qu'il contient appartiennent à la même classe (classification) ou présentent des valeurs similaires voire identiques (régression). Le choix de la mesure d'impureté dépend du type de problème (voir ci-dessous).

La mesure d'impureté est un élément essentiel de la construction des arbres de décision. En effet, c'est elle qui est utilisée pour comparer entre elles les règles de décision possibles. À chaque étape de la croissance de l'arbre (*tree growing*), l'algorithme sélectionne la règle de décision qui réduit le plus l'impureté, afin de définir des nœuds les plus homogènes possibles. L'arbre final dépend donc de la mesure d'impureté utilisée: si pour un problème donné on construit un second arbre avec une autre mesure d'impureté, on obtient généralement un arbre différent du premier (car les règles de décision retenues à chaque nœud ne sont plus les mêmes).

3.2.1.1. Mesures d'impureté pour les problèmes de classification

Dans un problème de classification où l'on souhaite classifier des observations parmi K classes, une **mesure d'impureté** $I(t)$ est une fonction qui quantifie l'hétérogénéité des classes dans un nœud donnée. Les mesures d'impureté usuelles détaillées ci-dessous partagent les deux propriétés suivantes:

- **Pureté maximale**: lorsque toutes les observations du nœud appartiennent à une seule classe, c'est-à-dire que la proportion $p_k = 1$ pour une classe k et $p_j = 0$ pour toutes les autres classes $j \neq k$, l'impureté est minimale et $I(t) = 0$. Cela indique que le nœud est **entièrement pur**, ou homogène.
- **Impureté maximale**: lorsque les observations sont réparties de manière uniforme entre toutes les classes, c'est-à-dire que la proportion $p_k = \frac{1}{K}$ pour chaque classe k, l'impureté atteint son maximum. Cette situation reflète une **impureté élevée**, car le nœud est très hétérogène et contient une forte incertitude sur la classe des observations.

Il existe trois mesures d'impureté couramment utilisées en classification:

1. L'indice de Gini

L'**indice de Gini** mesure la probabilité qu'un individu sélectionné au hasard dans un nœud soit mal classé si on lui attribue une classe au hasard, en fonction de la distribution des classes dans ce nœud. Pour un nœud t contenant K classes, l'indice de Gini $G(t)$ est donné par

$$G(t) = 1 - \sum_{k=1}^{K} p_k^2 \tag{1}$$

où p_k est la proportion d'observations appartenant à la classe k dans le nœud t.

Critère de choix: L'indice de Gini est très souvent utilisé parce qu'il est simple à calculer et capture bien l'homogénéité des classes au sein d'un nœud. Il privilégie les partitions où une classe domine fortement dans chaque sous-région.

2. L'entropie (ou entropie de Shannon)

L'**entropie** est une autre mesure de l'impureté utilisée dans les arbres de décision. Elle mesure la quantité d'incertitude ou de désordre dans un nœud, en s'appuyant sur la théorie de l'information. Pour un nœud t contenant K classes, l'entropie $E(t)$ est définie par:

$$E(t) = -\sum_{k=1}^{K} p_k \log(p_k) \tag{2}$$

où p_k est la proportion d'observations de la classe k dans le nœud t.

Critère de choix: L'entropie a tendance à être plus sensible aux changements dans les distributions des classes que l'indice de Gini, car elle attribue un poids plus élevé aux événements rares (valeurs de p_k très faibles). Elle est souvent utilisée lorsque l'erreur de classification des classes minoritaires est particulièrement importante.

3. Taux d'erreur

Le **taux d'erreur** est une autre mesure de l'impureté parfois utilisée dans les arbres de décision. Il représente la proportion d'observations mal classées dans un nœud. Pour un nœud t, le taux d'erreur TE (t) est donné par:

$$\text{TE } (t) = 1 - \max(p_k) \tag{3}$$

où $\max(p_k)$ est la proportion d'observations appartenant à la classe majoritaire dans le nœud.

Critère de choix: Bien que le taux d'erreur soit simple à comprendre, il est moins souvent utilisé dans la construction des arbres de décision parce qu'il est moins sensible que l'indice de Gini ou l'entropie aux petits changements dans la distribution des classes.

3.2.1.2. Mesures d'impureté pour les problèmes de régression

Dans les problèmes de régression, l'objectif est de partitionner les données de manière à réduire au maximum la variabilité des valeurs au sein de chaque sous-région. Pour mesurer cette variabilité, la mesure d'impureté la plus couramment employée est la somme des erreurs quadratiques (SSE). Elle évalue l'impureté d'une région en quantifiant à quel point les valeurs de cette région s'écartent de la moyenne locale. Pour un nœud t, contenant N observations avec des valeurs y_i, la SSE est donnée par:

$$\text{SSE } (t) = \sum_{i=1}^{N} (y_i - \hat{y})^2 \tag{4}$$

où $\hat{y}$ est la moyenne des valeurs y_i dans le nœud.

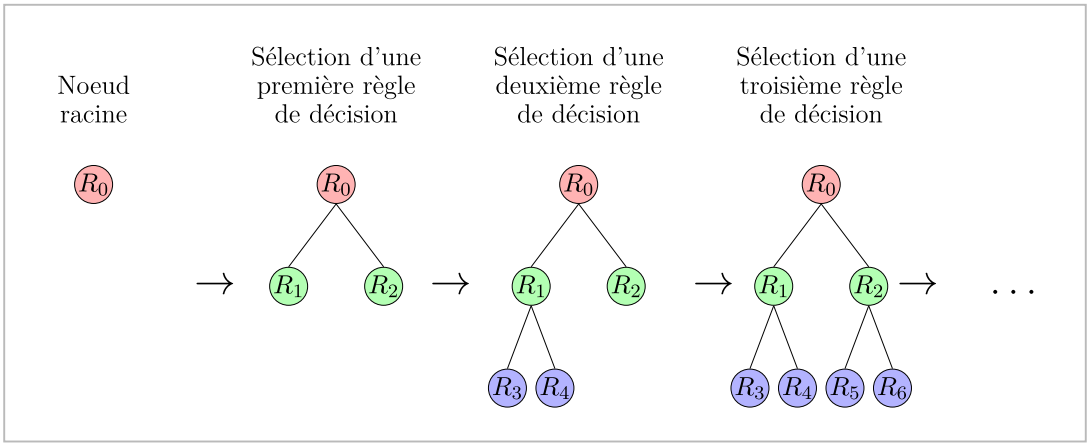
Cette mesure d'impureté a des propriétés similaires à celles présentées pour la classification: si toutes les valeurs de y_i dans un nœud sont proches de la moyenne $\hat{y}$, la SSE sera faible, indiquant une homogénéité élevée dans le nœud. Inversement, une SSE élevée indique une

grande variabilité dans les valeurs, donc un nœud impur. Une limite de cette mesure d'impureté est qu'elle est particulièrement sensible aux écarts élevés entre les valeurs observées et la moyenne prédite, et donc aux valeurs extrêmes.

3.2.2. Construire l'arbre de décision par un partitionnement séquentiel

Une fois la mesure d'impureté définie, l'algorithme CART construit séquentiellement le partitionnement de l'espace des caractéristiques en comparant les règles de décision possibles (voir figure Figure 9). La première étape de partitionnement part du nœud-racine, qui comprend l'ensemble des données d'entraînement. L'algorithme construit toutes les règles de décision candidates en parcourant toutes les valeurs de toutes les caractéristiques, les évalue en calculant la réduction de l'impureté induite par chacune d'entre elles et sélectionne la règle de décision (caractéristique et seuil) qui entraîne la réduction d'impureté maximale. Par exemple, l'algorithme évalue la règle candidate "Superficie $>$ 100 m²" en calculant la somme des impuretés au sein des deux sous-régions générées par cette règle ("Oui" et "Non"), puis calcule la différence entre cette somme et l'impureté du noeud-racine. L'algorithme évalue ensuite la règle candidate "Superficie $>$ 101 m²" de la même façon, et ainsi de suite pour toutes les valeurs de superficie, puis évalue les règles candidates construites avec le nombre de pièces, et enfin sélectionne la meilleure règle. La deuxième étape du partitionnement reproduit le même processus, cette fois au niveau d'un des deux nœuds-enfants, et ainsi de suite.

Figure 9 : Construction d'un arbre de décision



L'algorithme CART poursuit ce partitionnement récursif jusqu'à ce qu'un **critère d'arrêt** prédéfini soit atteint. Dans la plupart des implémentations de CART, les valeurs par défaut de ces critères d'arrêt sont telles que que l'algorithme construit un arbre maximal: le plus profond possible, avec une observation par feuille terminale.

3.2.3. Contrôler la complexité de l'arbre

Il est généralement préférable d'éviter les arbres trop complexes car ils sont souvent affectés par un problème de surajustement. Deux approches permettent de contrôler la complexité d'un arbre de décision:

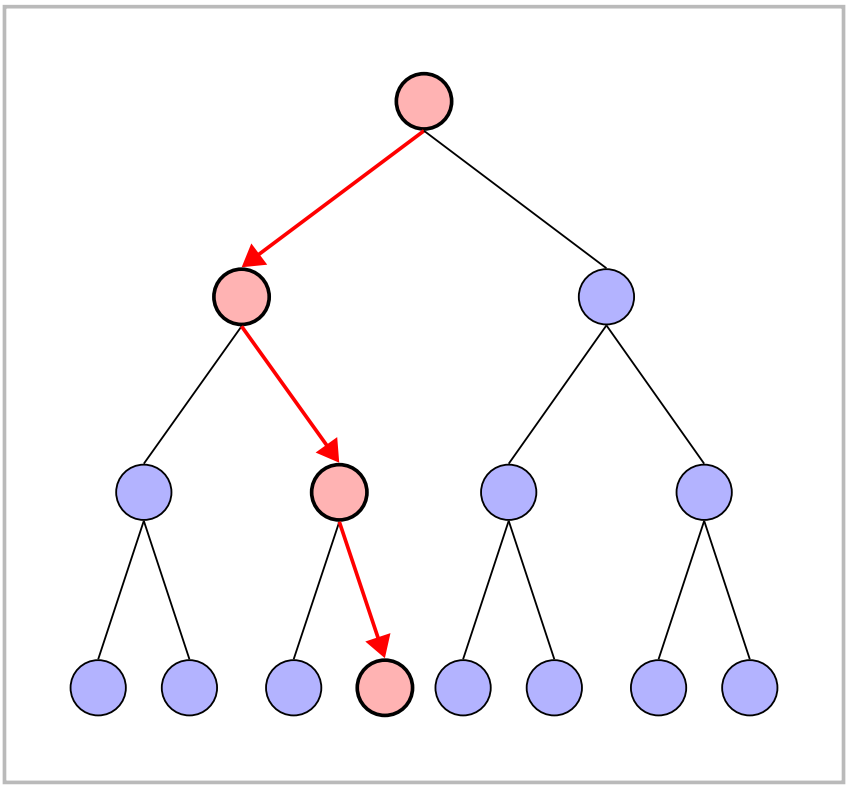
- Approche *a priori*: la complexité de l'arbre peut être plafonnée pendant sa construction à l'aide d'hyperparamètres telles que la profondeur maximale de l'arbre, le nombre minimal d'observations par feuille ou la réduction minimale de l'impureté nécessaire à chaque étape pour ajouter un noeud interne. Cette approche est simple à mettre en oeuvre, mais peut aboutir à des arbres trop simples et peu prédictifs si les hyperparamètres sont mal choisis.
- Approche *a posteriori*: l'autre approche consiste à construire un arbre maximal puis à procéder à un **élagage** (*tree pruning*) qui vise à simplifier l'arbre et à augmenter sa capacité à généraliser sur de nouvelles données en supprimant progressivement les branches les moins utiles. Il existe différents critères d'élagage, parmi lesquels le chemin de coût-complexité[2].

3.2.4. Utiliser l'arbre pour prédire

Une fois l'arbre construit, la prédiction pour une nouvelle observation s'effectue en suivant les branches de l'arbre depuis le nœud racine jusqu'à un nœud terminal (ou feuille), comme l'illustre la figure @#fig-decision-tree-prediction. À chaque nœud interne, une décision est prise en fonction des valeurs des caractéristiques de l'observation, ce qui détermine la direction à suivre vers l'une des sous-régions. Ce cheminement se poursuit jusqu'à ce que l'observation atteigne une feuille, où la prédiction finale sera simplement la valeur associée à cette feuille.

[2]Les approches d'élagage sont détaillées notamment dans ce cours et dans la documentation de `scikit-learn` .

Figure 10 : Prédire avec un arbre de décision



3.3. Avantages et limites des arbres de décision

3.3.1. Avantages

- **Simplicité et interprétabilité**: Les arbres de décision sont faciles à comprendre et à visualiser (à condition qu'ils ne soient pas trop profonds).
- **Facilité d'usage**: les arbres de décision ne demandent pas de transformations complexes des données.
- **Flexibilité**: Ils peuvent gérer des caractéristiques numériques et catégorielles, ainsi que les valeurs manquantes.
- **Gestion des interactions**: Les arbres sont des modèles non paramétriques et ne font aucune hypothèse sur la distribution des variables. Ils capturent aisément les relations non-linéaires et les interactions entre les caractéristiques.

3.3.2. Limites

- **Surapprentissage**: Les arbres de décision peuvent facilement devenir trop complexes et être surajustés d'entraînement, ce qui nuit à leur capacité prédictive sur de nouvelles données.
- **Biais envers les classes majoritaires**: En présence de données déséquilibrées, les arbres de décision peuvent privilégier la classe majoritaire, ce qui dégrade la performance sur les classes minoritaires.

- **Optimisation locale**: L'approche gloutonne peut conduire à des solutions globalement sous-optimales (optimum local).
- **Instabilité**: De petits changements dans les données peuvent entraîner des changements significatifs dans la structure de l'arbre (manque de robustesse).

4. Le *bagging*

Le *bagging*, ou "bootstrap aggregating", est une méthode ensembliste qui vise à améliorer la stabilité et la précision des algorithmes d'apprentissage automatique en agrégeant plusieurs modèles (L. Breiman [5]). Chaque modèle est entraîné sur un échantillon distinct généré par une technique de rééchantillonnage (*bootstrap*). Ces modèles sont ensuite combinés pour produire une prédiction agrégée, souvent plus robuste et généralisable que celle obtenue par un modèle unique.

Le *bagging* est peu utilisé en pratique car il a été supplanté par les forêts aléatoires (présentées dans le chapitre suivant) dont il constitue l'un des fondements méthodologiques.

4.1. Principe du *bagging*

Le *bagging* comporte trois étapes principales:

- **L'échantillonnage bootstrap** : L'échantillonnage bootstrap consiste à créer des échantillons distincts en tirant aléatoirement avec remise des observations du jeu de données initial. Chaque échantillon *bootstrap* contient le même nombre d'observations que le jeu de données initial, mais certaines observations sont répétées (car sélectionnées plusieurs fois), tandis que d'autres sont omises.
- **L'entraînement de plusieurs modèles** : Un modèle est entraîné sur chaque échantillon bootstrap. Les modèles peuvent être des arbres de décision, des régressions ou tout autre algorithme d'apprentissage. Le *bagging* est particulièrement efficace avec des modèles instables, tels que les arbres de décision non élagués.
- **L'agrégation des prédictions** : Les prédictions de tous les modèles sont ensuite agrégées, généralement en prenant la moyenne des prédictions dans le cas de la régression, et la classe majoritaire (ou la moyenne des probabilités prédites pour chaque classe) dans le cas de la classification.

4.2. Pourquoi (et dans quelles situations) le *bagging* fonctionne

L'objectif du *bagging* est de construire un prédicteur plus précis en agrégeant les prédictions de plusieurs modèles entraînés sur des échantillons (légèrement) différents les uns des autres. L. Breiman [5] a démontré que cette méthode est particulièrement efficace lorsqu'elle est appliquée à des modèles très instables, dont les performances sont particulièrement sensibles aux variations du jeu de données d'entraînement, et peu biaisés. Cette section vise à expliquer pourquoi (et sous quelles conditions) l'agrégation par *bagging* permet de construire un **prédicteur agrégé** plus performant. Dans la suite, nous notons $\varphi(x, L)$ un prédicteur (d'une valeur numérique dans le cas de la *régression* ou d'une classe dans le cas de la *classification*),

entraîné sur un ensemble d'apprentissage L, et prenant en entrée un vecteur de caractéristiques x.

4.2.1. La régression: réduction de l'erreur quadratique moyenne par agrégation

Dans le contexte de la **régression**, l'objectif est de prédire une valeur numérique Y à partir d'un vecteur de caractéristiques x. Un modèle de régression $\phi(x, L)$ est construit à partir d'un ensemble d'apprentissage L, et produit une estimation de Y pour chaque observation x.

4.2.1.1. Définition du prédicteur agrégé

Dans le cas de la régression, le **prédicteur agrégé** est défini comme suit :

$$\phi_A(x) = E_L[\phi(x, L)] \tag{5}$$

où $\phi_A(x)$ représente la prédiction agrégée, $E_L[.]$ correspond à l'espérance prise sur tous les échantillons d'apprentissage possibles L, chacun étant tiré selon la même distribution que le jeu de données initial, et $\phi(x, L)$ correspond à la prédiction du modèle construit sur l'échantillon d'apprentissage L.

4.2.1.2. La décomposition biais-variance

Pour mieux comprendre comment l'agrégation améliore la performance globale d'un modèle individuel $\phi(x, L)$, revenons à la **décomposition biais-variance** de l'erreur quadratique moyenne en x, qui est une mesure de performance très courante pour les problèmes de régression:

$$E_L[(Y - \phi(x, L))^2] = \underbrace{(E_L[\phi(x, L) - Y])^2}_{\text{Biais}^2} + \underbrace{E_L\left[(\phi(x, L) - E_L[\phi(x, L)])^2\right]}_{\text{Variance}} \tag{6}$$

L'erreur quadratique moyenne en x se décompose en deux termes:

- Le **biais** est la différence entre la valeur observée Y que l'on souhaite prédire et la prédiction moyenne $E_L[\phi(x, L)]$. Si le modèle est peu prédictif, le biais sera élevé.
- La **variance** est la variabilité des prédictions $\phi(x, L)$ autour de leur moyenne $E_L[\phi(x, L)]$. Un modèle avec une variance élevée est très sensible aux fluctuations au sein des données d'entraînement: ses prédictions varient beaucoup lorsque les données d'entraînement se modifient.

L'équation Equation 6 illustre l'**arbitrage biais-variance** qui est omniprésent en *machine learning*: plus la complexité d'un modèle s'accroît (exemple: la profondeur d'un arbre), plus son biais sera plus faible (car ses prédictions seront de plus en plus proches des données d'entraînement), et plus sa variance sera élevée (car ses prédictions, étant très proches des données d'entraînement, auront tendance à varier fortement d'un jeu d'entraînement à l'autre).

4.2.1.3. L'inégalité de Breiman (1996)

L. Breiman [5] compare l'erreur quadratique moyenne d'un modèle individuel avec celle du modèle agrégé et démontre l'inégalité suivante :

$$(Y - \phi_A(x))^2 \leq E_L\left[(Y - \phi(x, L))^2\right] \tag{7}$$

- Le terme $(Y - \phi_A(x))^2$ représente l'erreur quadratique en x du **prédicteur agrégé** $\phi_A(x)$;
- Le terme $E_L\left[(Y - \phi(x, L))^2\right]$ est l'erreur quadratique moyenne en x d'un **prédicteur individuel** $\phi(x, L)$ entraîné sur un échantillon aléatoire L. Cette erreur varie en fonction des données d'entraînement.

Cette inégalité montre que **l'erreur quadratique du prédicteur agrégé est toujours inférieure ou égale à l'erreur quadratique moyenne d'un prédicteur individuel**. Étant donné que le biais du prédicteur agrégé est égal au biais du prédicteur individuel, l'inégalité précédente implique que la **variance du modèle agrégé** $\phi_A(x)$ est **toujours inférieure ou égale** à la variance moyenne d'un modèle individuel:

$$\text{Var}\ (\phi_A(x)) = \text{Var}\ (E_L[\phi(x, L)]) \leq E_L[\text{Var}\ (\phi(x, L))] \tag{8}$$

Autrement dit, le processus d'agrégation réduit l'erreur de prédiction globale en réduisant la **variance** des prédictions, tout en conservant un biais constant.

Ce résultat ouvre la voie à des considérations pratiques immédiates. Lorsque le modèle individuel est instable et présente une variance élevée, l'inégalité $Var(\phi_A(x)) \leq E_L[Var(\phi(x, L))]$ est forte, ce qui signifie que l'agrégation peut améliorer significativement la performance globale du modèle. En revanche, si $\phi(x, L)$ varie peu d'un ensemble d'entraînement à un autre (modèle stable avec variance faible), alors $Var(\phi_A(x))$ est proche de $E_L[Var(\phi(x, L))]$, et la réduction de variance apportée par l'agrégation est faible. Ainsi, **le *bagging* est particulièrement efficace pour les modèles instables**, tels que les arbres de décision, mais moins efficace pour les modèles stables tels que les méthodes des k plus proches voisins.

4.2.2. La classification: vers un classificateur presque optimal par agrégation

Dans le cas de la classification, le mécanisme de réduction de la variance par le *bagging* permet, sous une certaine condition, d'atteindre un **classificateur presque optimal** (*nearly optimal classifier*). Ce concept a été introduit par L. Breiman [5] pour décrire un modèle qui tend à classer une observation dans la classe la plus probable, avec une performance approchant celle du classificateur Bayésien optimal (la meilleure performance théorique qu'un modèle de classification puisse atteindre).

Pour comprendre ce résultat, introduisons $Q(j \mid x) = E_L\left(1_{\varphi(x,L)=j}\right) = P(\varphi(x, L) = j)$, la probabilité qu'un modèle $\varphi(x, L)$ prédise la classe j pour l'observation x, et $P(j \mid x)$, la vraie probabilité qu'une observation de caractéristiques x appartienne à la classe j.

4.2.2.1. Définition : classificateur order-correct

Un classificateur $\varphi(x, L)$ est dit **order-correct** pour une observation x si, en espérance, il identifie **correctement la classe la plus probable**, même s'il ne prédit pas toujours avec exactitude les probabilités associées à chaque classe $Q(j \mid x)$. Formellement, un prédicteur est dit "order-correct" pour un vecteur de caractéristiques x si $argmax_j Q(j \mid x) = argmax_j P(j \mid x)$ où $P(j \mid x)$ est la vraie probabilité que l'observation de caractéristiques x appartienne à la classe j, et $Q(j \mid x)$ est la probabilité que l'observation de caractéristiques x appartienne à la classe j prédite par le modèle $\varphi(x, L)$. En termes moins techniques, la classe prédite le plus souvent par le modèle pour une observation x correspond à la classe la plus probable pour cette observation dans la vraie distribution.

En généralisant sur toutes les valeurs de x, un classificateur est **order-correct** si, pour **tout** x, la classe qu'il prédit le plus souvent correspond à celle qui a la probabilité maximale $P(j \mid x)$ dans la vraie distribution.

4.2.2.2. Prédicteur agrégé en classification: le vote majoritaire

Dans le cas de la classification, le prédicteur agrégé est défini par le **vote majoritaire**. Cela signifie que si K classificateurs sont entraînés sur K échantillons distincts, la classe prédite pour x est celle qui reçoit **le plus de votes** de la part des modèles individuels. Formellement, le classificateur agrégé $\varphi A(x)$ est défini par :

$$\varphi A(x) = \text{argmax}_j \sum_L I(\phi(x, L) = j) = argmax_j Q(j \mid x) \tag{9}$$

4.2.2.3. Performance globale: convergence vers un classificateur presque optimal

L. Breiman [5] montre que si chaque prédicteur individuel $\varphi(x, L)$ est order-correct pour une observation x, alors le prédicteur agrégé $\varphi A(x)$, obtenu par **vote majoritaire**, atteint la performance optimale pour cette observation, c'est-à-dire que lorsque le nombre de prédicteurs individuels augmente le prédicteur agrégé converge vers la classe ayant la probabilité maximale $P(j \mid x)$ pour l'observation x. Le vote majoritaire permet ainsi de **réduire les erreurs aléatoires** des classificateurs individuels. En généralisant sur toutes les valeurs de x, **si chaque prédicteur individuel $\varphi(x, L)$ est order-correct pour tout x, alors le classificateur agrégé ϕA sera optimal pour tout x**. Ce résultat d'optimalité en tout point dépend d'une hypothèse cruciale: les classificateurs individuels doivent être order-corrects en tout point. Dans les régions de l'espace où ils ne sont pas order-corrects (c'est-à-dire que la classe qu'ils prédisent le plus fréquemment n'est pas la classe la plus probable dans la vraie distribution), l'agrégation par vote majoritaire n'améliore pas les performances du prédicteur; celles-ci peuvent même se détériorer par rapport aux modèles individuels si l'agrégation conduit à amplifier des erreurs systématiques (biais).

Or, si l'on peut raisonnablement penser qu'un classificateur individuel présentant un biais faible (peu d'erreurs systématiques) sera order-correct pour la plupart des points x, il n'est pas certain qu'il le soit en tout point. En effet, chaque classificateur individuel est entraîné sur un échantillon fini de données, ce qui introduit du bruit dans son estimation de la vraie distribution conditionnelle $P(j \mid x)$. Il peut donc, pour certains x, prédire le plus souvent une classe différente de celle qui maximise la vraie probabilité $P(j \mid x)$. Par ailleurs, la variabilité due à l'échantillonnage peut induire une couverture incomplète de l'ensemble des x possibles et aboutir à des erreurs systématiques sur certains points x. C'est pour ces raisons que Breiman conclut que **le prédicteur agrégé $\varphi A(x)$ converge vers un classificateur presque optimal**, au sens d'optimal presque partout.

4.3. Quand utiliser le *bagging* en pratique

Le *bagging* est particulièrement utile lorsque les modèles individuels présentent une variance élevée et sont instables. Dans de tels cas, l'agrégation des prédictions peut réduire significativement la variance globale, améliorant ainsi la performance du modèle agrégé. Les situations où le *bagging* est recommandé incluent typiquement:

- Les modèles instables : Les modèles tels que les arbres de décision non élagués, qui sont sensibles aux variations des données d'entraînement, bénéficient grandement du *bagging*. L'agrégation atténue les fluctuations des prédictions dues aux différents échantillons.
- Les modèles avec biais faibles: En classification, si les modèles individuels sont order-corrects pour la majorité des observations, le *bagging* peut améliorer la précision en renforçant les prédictions correctes et en réduisant les erreurs aléatoires.

Inversement, le *bagging* peut être moins efficace ou même néfaste dans certaines situations :

- Les modèles stables avec variance faible : Si les modèles individuels sont déjà stables et présentent une faible variance (par exemple, la régression linéaire), le *bagging* n'apporte que peu d'amélioration, car la réduction de variance supplémentaire est minimale.
- La présence de biais élevé : Si les modèles individuels sont biaisés, entraînant des erreurs systématiques, le *bagging* peut amplifier ces erreurs plutôt que de les corriger. Dans de tels cas, il est préférable de s'attaquer d'abord au biais des modèles avant de considérer l'agrégation.
- Les échantillons de petite taille : Avec des ensembles de données limités, les échantillons bootstrap peuvent ne pas être suffisamment diversifiés ou représentatifs, ce qui réduit l'efficacité du *bagging* et peut augmenter le biais des modèles.

Ce qui qu'il faut retenir: le *bagging* peut améliorer substantiellement la performance des modèles d'apprentissage automatique lorsqu'il est appliqué dans des conditions appropriées. Il est essentiel d'évaluer la variance et le biais des modèles individuels, ainsi que la taille et

la représentativité du jeu de données, pour déterminer si le *bagging* est une stratégie adaptée. Lorsqu'il est utilisé judicieusement, le *bagging* peut conduire à des modèles plus robustes et précis, exploitant efficacement la puissance de l'agrégation pour améliorer la performance des modèles individuels.

5. La forêt aléatoire

La forêt aléatoire (*random forests*) est une méthode ensembliste puissante et largement utilisée pour les tâches de classification et de régression. Elle combine la simplicité des arbres de décision et l'échantillonnage des observations et des variables avec la puissance de l'agrégation pour améliorer les performances prédictives et réduire le risque de surajustement (*overfitting*).

5.1. Principe de la forêt aléatoire

La forêt aléatoire est une extension du *bagging*, présenté dans la section Section 4. Elle introduit un niveau supplémentaire d'aléa dans la construction des arbres, puisqu'à chaque nouvelle noeud, la règle de décision est choisie en considérant uniquement un sous-ensemble de variables **sélectionné aléatoirement**. Cette randomisation supplémentaire **réduit la corrélation** entre les arbres, ce qui permet de diminuer la variance des prédiction du modèle agrégé.

Les forêts aléatoires reposent donc sur quatre éléments essentiels:

- **Les arbres de régression et de classification**: Les modèles élémentaires sont des arbres de décision profonds.
- **L'échantillonnage *bootstrap***: Chaque arbre est construit à partir d'un échantillon aléatoire du jeu de données d'entraînement tiré avec remise (ou parfois sans remise).
- **La sélection aléatoire de variables** : Lors de la construction d'un arbre, à chaque nœud de celui-ci, un sous-ensemble aléatoire de variables est sélectionné. La meilleure règle de décision est ensuite choisie uniquement parmi ces caractéristiques.
- **L'agrégation des prédictions** : Comme pour le *bagging*, les prédictions de tous les arbres sont combinées. On procède généralement à la moyenne (ou à la médiane) des prédictions dans le cas de la régression, et au vote majoritaire (ou à la moyenne des probabilités prédites pour chaque classe) dans le cas de la classification.

5.2. Comment construit-on une forêt aléatoire?

L'entraînement d'une forêt aléatoire est très similaire à celui du *bagging* et se résume comme suit (voir figure Figure 11):

- Le nombre d'arbres à construire est défini *a priori*.
- Pour chaque arbre, on effectue les étapes suivantes:
  - Générer un échantillon *bootstrap* de taille fixe à partir des données d'entraînement.
  - Construire récursivement un arbre de décision à partir de cet échantillon:
    - À chaque nœud de l'arbre, un sous-ensemble de *features* est sélectionné aléatoirement.

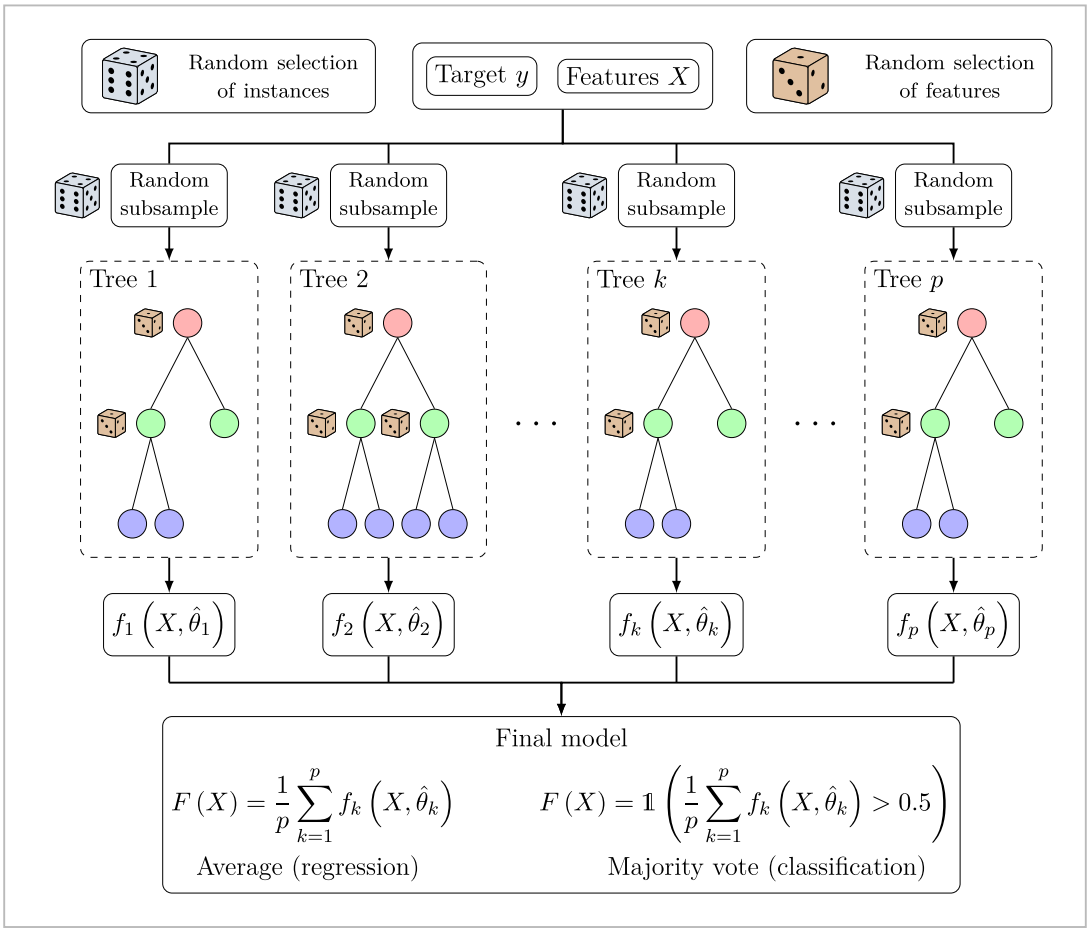
– Déterminer le couple (variable, valeur) qui définit la règle de décision divisant la population du nœud en deux sous-groupes les plus homogènes possibles.
– Créer les deux nœuds-enfants à partir de cette règle de décision.
– Arrêter la croissance de l'arbre selon des critères d'arrêt fixés *a priori*.

Pour construire la prédiction de la forêt aléatoire une fois celle-ci entraînée, on agrège les arbres selon une méthode qui dépend du problème modélisé:

- Régression: la prédiction finale est la moyenne des prédictions de tous les arbres.
- Classification: chaque arbre vote pour une classe, et la classe majoritaire est retenue.

Les principaux hyper-paramètres des forêts aléatoires (détaillés dans la section la section Section 8) sont les suivants: le nombre d'arbres, la méthode et le taux d'échantillonnage, le nombre (ou la proportion) de variables considérées à chaque nœud, le critère de division des nœuds (ou mesure d'hétérogénéité), et les critères d'arrêt (notamment la profondeur de l'arbre, le nombre minimal d'observations dans une feuille terminale, et le nombre minimal d'observations qu'un nœud doit comprendre pour être divisé en deux).

Figure 11 : Entraînement d'une forêt aléatoire



5.3. Pourquoi les forêts aléatoires sont-elles performantes?

Les propriétés théoriques des forêts aléatoires permettent de comprendre pourquoi (et dans quelles situations) elles sont particulièrement robustes et performantes.

5.3.1. Réduction de la variance par agrégation

L'agrégation de plusieurs arbres permet de réduire la variance globale du modèle, ce qui améliore la stabilité des prédictions. Lorsque les estimateurs sont (faiblement) biaisés mais caractérisés par une variance élevée, l'agrégation permet d'obtenir un estimateur avec un biais similaire mais une variance réduite. La démonstration est identique à celle présentée dans la section Section 4.

5.3.2. Convergence et limite théorique au surajustement

Bien qu'elles s'avèrent très performantes en pratique, **il n'est pas prouvé à ce stade que les forêts aléatoires convergent vers une solution optimale** lorsque la taille de l'échantillon tend vers l'infini (G. Louppe [9]). Plusieurs travaux théoriques ont toutefois fourni des preuves de convergence pour des versions simplifiées de l'algorithme (par exemple, G. Biau [10]).

Par ailleurs, une propriété importante des forêts aléatoires démontrée par L. Breiman [6] est que leur erreur de généralisation, c'est-à-dire l'écart entre les prédictions du modèle et les résultats attendus sur des données jamais vues (donc hors de l'échantillon d'entraînement), diminue à mesure que le nombre d'arbres augmente et converge vers une valeur constante. Autrement dit, **la forêt aléatoire ne souffre pas d'un surajustement croissant avec le nombre d'arbres** (contrairement aux algorithmes de *gradient boosting*). La conséquence pratique de ce résultat est qu'inclure un (trop) grand nombre d'arbres dans le modèle n'en dégrade pas la qualité, ce qui contribue à la rendre particulièrement robuste. En revanche, une forêt aléatoire peut souffrir de surajustement si ses autres hyperparamètres sont mal choisis (des arbres trop profonds par exemple).

5.3.3. Facteurs influençant l'erreur de généralisation

L'erreur de généralisation des forêts aléatoires est influencée par deux facteurs principaux :

- **La puissance prédictrice des arbres individuels** : Les arbres doivent être suffisamment prédictifs pour contribuer positivement à l'ensemble, et idéalement sans biais.
- **La corrélation entre les arbres** : Moins les arbres sont corrélés, plus la variance de l'ensemble est réduite, car leurs erreurs tendront à se compenser. Inversement, des arbres fortement corrélés auront tendance à faire des erreurs similaires, donc agréger un grand nombre d'arbres n'apportera pas grand chose.

On peut mettre en évidence ces deux facteurs dans le cas d'une forêt aléatoire utilisée pour une tâche de régression (où l'objectif est de minimiser l'erreur quadratique moyenne). Dans ce cas, la variance de la prédiction du modèle peut être décomposée de la façon suivante:

$$\text{Var}\left(\hat{f}(x)\right) = \rho(x)\sigma(x)^2 + \frac{1-\rho(x)}{M}\sigma(x)^2 \tag{10}$$

où $\rho(x)$ est le coefficient de corrélation moyen entre les arbres individuels, $\sigma(x)^2$ est la variance d'un arbre individuel, M est le nombre d'arbres dans la forêt. Cette décomposition fait apparaître l'influence de la corrélation entre les arbres sur les performance de la forêt aléatoire:

- **Si $\rho(x)$ est proche de 1** (forte corrélation entre les arbres) : la première composante $\rho\sigma^2$ domine et la réduction de variance est moindre lorsque le nombre d'arbres augmente.
- **Si $\rho(x)$ est proche de 0** (faible corrélation entre les arbres) : la seconde composante $\frac{1-\rho}{M}\sigma^2$ et la variance est davantage réduite avec l'augmentation du nombre d'arbres M.

L'enjeu de l'entraînement des forêts aléatoires est donc de **minimiser la corrélation entre les arbres tout en maximisant leur capacité à prédire correctement**, ce qui permet de réduire la variance globale sans augmenter excessivement le biais. La sélection aléatoires des caractéristiques (*features*) à chaque nœud joue un rôle majeur dans cet arbitrage entre puissance prédictive des arbres pris isolément et corrélation entre arbres.

5.4. Evaluation des performances par l'erreur *Out-of-Bag* (OOB)

La forêt aléatoire présente une particularité intéressante et très utile en pratique: **il est possible d'évaluer les performances d'une forêt aléatoire directement à partir des données d'entraînement**, grâce à l'estimation de l'erreur *Out-of-Bag* (OOB). Cette caractéristique repose sur le fait que chaque arbre est construit à partir d'un échantillon *bootstrap*, c'est-à-dire un échantillon tiré avec remise. Cela implique qu'une part des observations ne sont pas utilisées pour entraîner un arbre donné. Ces observations laissées de côté forment un **échantillon dit *out-of-bag***, que l'on peut utiliser pour évaluer la performance de chaque arbre. On peut donc construire pour chaque observation du jeu d'entraînement une prédiction qui agrège uniquement les prédictions des arbres pour lesquels cette observation est *out-of-bag*; cette prédiction n'est pas affectée par le surajustement (puisque cette observation n'a jamais été utilisée pour entraîner ces arbres). De cette façon, il est possible d'évaluer correctement la performance de la forêt aléatoire en comparant ces prédictions avec la variable-cible à l'aide d'une métrique bien choisie.

La procédure d'estimation de l'erreur OOB se déroule comme ceci:

1. **Entraînement de la forêt aléatoire**: la forêt aléatoire est entraînée sur les données d'entraînement selon la procédure détaillée ci-dessus.

2. **Calcul des prédictions *out-of-bag*** : Pour chaque observation (x_i, y_i) des données d'entraînement, on calcule la prédiction de tous les arbres pour lesquels elle fait partie de l'échantillon *out-of-bag*. La prédiction *out-of-bag* finale est obtenue en agrégeant ces prédictions selon la procédure standard détaillée ci-dessus (moyenne pour la régression, vote majoritaire pour la classification).
3. **Calcul de l'erreur OOB** : L'erreur OOB est ensuite calculée en comparant les prédictions avec la variable-cible y sur toutes les observations, à l'aide d'une métrique (précision, rappel, AUC, erreur quadratique moyenne, score de Brier…).

L'utilisation de l'erreur OOB présente de multiples avantages:

- **Approximation de l'erreur de généralisation**: L'erreur OOB est en général considérée comme une bonne approximation de l'erreur de généralisation, comparable à celle obtenue par une validation croisée.
- **Pas besoin de jeu de test séparé** : L'un des principaux avantages de l'erreur OOB est qu'elle ne nécessite pas de réserver une partie des données pour la mesure de la performance. Cela est particulièrement utile lorsque la taille du jeu de données est limitée, car toutes les données peuvent être utilisées pour l'entraînement tout en ayant une estimation fiable de la performance. Ceci dit, il est malgré tout recommandé de conserver un ensemble de test si la taille des données le permet, car il arrive que l'erreur OOB surestime la performance du modèle.
- **Gain de temps** : Contrairement à la validation croisée qui requiert de réentraîner plusieurs fois le modèle pour un jeu donné d'hyperparamètres, l'erreur OOB ne nécessite qu'un seul entraînement du modèle. Cela induit un gain de temps appréciable lors de l'optimisation des hyperparamètres.
- **Pertinence pour certains cas d'usage** : les prédictions *out-of-bag* peuvent être particulièrement utiles lorsqu'on veut utiliser le modèle _sur les données qui ont servi à l'entraîner. Bien qu'elle soit inhabituelle, cette situation se rencontre dans les travaux de la statistique publique, par exemple si l'on veut entraîner une forêt aléatoire pour prédire la probabilité de réponse dans une enquête, puis utiliser ce modèle dans le cadre d'une repondération selon la méthode des groupes de réponse homogène.

5.5. Interprétation et importance des variables

Les forêts aléatoires sont des modèles d'apprentissage performants, mais leur complexité interne les rend difficiles à interpréter, ce qui leur vaut souvent le qualificatif de "boîtes noires". Comprendre l'influence des variables explicatives sur les prédictions est crucial pour interpréter les résultats.

L'objectif des **méthodes d'interprétabilité** (ou d'importance des variables) est d'identifier les variables les plus influentes sur la variable cible, de comprendre les mécanismes prédictifs sous-jacents, et potentiellement d'extraire des règles de décision simples et transparentes. Plusieurs méthodes d'importance des variables existent, mais il est important de comprendre leurs forces et faiblesses.

5.5.1. Mesures d'importance classiques (et leurs biais)

Il existe de multiples mesures d'importance des variables. Deux d'entre elles sont fréquemment utilisées:

- **Réduction moyenne de l'impureté** (*Mean Decrease in Impurity - MDI*) : Cette méthode quantifie l'importance d'une variable par la somme des réductions d'impureté qu'elle induit dans tous les arbres de la forêt. Plus spécifiquement, pour chaque variable, on s'intéresse à la moyenne des réductions d'impureté qu'elle a engendrées dans tous les nœuds de tous les arbres où elle est impliquée. Les variables présentant la réduction moyenne d'impureté la plus élevée sont considérées comme les prédicteurs les plus importants.

La MDI présente des biais importants. Elle est notamment sensible aux variables catégorielles avec de nombreuses modalités, qui peuvent apparaître artificiellement importantes (même si leur influence réelle est faible), ainsi qu'aux variables avec une échelle de valeurs plus étendues, qui obtiennent des scores plus élevés, indépendamment de leur importance réelle. Elle est également fortement biaisée en présence de variables explicatives corrélées, ce qui conduit à surestimer l'importance de variables redondantes. Les interactions entre variables ne sont pas non plus prises en compte de manière adéquate.

- **Importance par permutation** (*Mean Decrease Accuracy - MDA*) : Cette méthode évalue l'importance d'une variable en mesurant la diminution de précision du modèle après permutation aléatoire de ses valeurs. Plus spécifiquement, pour chaque variable, les performances du modèle sont comparées avant et après la permutation de ses valeurs. La différence moyenne de performance correspond à la MDA. L'idée est que si l'on permute aléatoirement les valeurs d'une variable (cassant ainsi sa relation avec la cible), une variable importante entraînera une hausse significative de l'erreur de généralisation.

Comme la MDI, la MDA présente des biais lorsque les variables sont corrélées. En particulier, la MDA peut surévaluer l'importance de variables qui sont corrélées à d'autres variables importantes, même si elles n'ont pas d'influence directe sur la cible (C. Bénard, S. Da Veiga, and E. Scornet [11]).

Plusieurs stratégies peuvent aider à réduire les biais d'interprétation :

- Prétraitement des variables: Standardisation des variables, regroupement des modalités rares des variables catégorielles, réduction de la cardinalité des variables catégorielles.

- Analyse des corrélations: Identification et gestion des variables fortement corrélées, qui peuvent fausser les mesures d'importance.
- Choix de méthodes robustes: Privilégier les méthodes moins sensibles aux biais, comme les CIF ou la Sobol-MDA, et, le cas échéant, SHAFF pour les valeurs de Shapley. Ces méthodes sont présentées dans la sectio suivante.

5.5.2. Méthodes d'importance avancées

Pour pallier les limites des méthodes traditionnelles, des approches plus sophistiquées ont été développées.

- **Valeurs de Shapley**: Les valeurs de Shapley permettent de quantifier la contribution de chaque variable explicative à la variance expliquée de la variable cible, en tenant compte des interactions entre les variables. Elles attribuent à chaque variable une contribution marginale moyenne à la performance du modèle, en considérant toutes les combinaisons possibles de sous-ensembles de variables. Cependant, l'estimation des valeurs de Shapley est computationnellement coûteuse (complexité exponentielle avec le nombre de variables). Des méthodes approximatives existent, mais peuvent introduire des biais. L'algorithme SHAFF (C. Bénard, G. Biau, S. Da Veiga, and E. Scornet [12]) propose une solution rapide et précise à ce problème, en tirant parti des propriétés des forêts aléatoires.
- **Conditional Inference Forests** (CIF): Les CIF (C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn [13]), implémentées dans le package party de R (cforest), corrigent certains biais de la MDI en utilisant des tests statistiques conditionnels pour sélectionner les variables et les seuils de coupure dans les arbres. Elles sont particulièrement robustes face aux variables hétérogènes et aux corrélations entre variables. Couplées à un échantillonnage sans remise, les CIF fournissent des mesures d'importance plus fiables.
- **Sobol-MDA**: La Sobol-MDA combine l'idée de la MDA avec une approche basée sur les indices de Sobol, permettant de gérer efficacement les variables dépendantes. Au lieu de permuter les valeurs, elle projette la partition des arbres sur le sous-espace excluant la variable dont on souhaite mesurer l'importance, simulant ainsi son absence. Elle est plus efficace en calcul que les méthodes MDA classiques tout en fournissant une mesure d'importance cohérente, convergeant vers l'indice de Sobol total (la mesure appropriée pour identifier les covariables les plus influentes, même avec des dépendances) (C. Bénard, S. Da Veiga, and E. Scornet [11]).

5.6. Le *gradient boosting*

5.6.1. Introduction

Le fondement théorique du *boosting* est un article de 1990 (R. E. Schapire [14]) qui a démontré théoriquement que, sous certaines conditions, il est possible de transformer un modèle prédictif peu performant en un modèle prédictif très performant. Plus précisément, cet article prouve que s'il est possible de construire un modèle simple dont les prédictions ne sont que légèrement meilleures que le hasard (appelé *weak learner*), alors il est possible de construire un modèle ayant un pouvoir prédictif arbitrairement élevé (appelé *strong learner*) en améliorant progressivement ce modèle simple. Le *boosting* est donc une méthode qui combine une approche ensembliste reposant sur un grand nombre de modèles simples avec un entraînement séquentiel: chaque modèle simple tâche d'améliorer la prédiction globale en corrigeant les erreurs commises par l'ensemble des modèles précédents. Bien qu'une approche de *boosting* puisse en théorie mobiliser différentes classes de *weak learners*, en pratique les *weak learners* utilisés par les algorithmes de *boosting* sont presque toujours des arbres de décision peu profonds.

En termes plus techniques, les différentes variantes du *boosting* partagent toutes trois caractéristiques communes:

- Ils visent à **trouver une approximation** $\hat{F}$ d'une fonction inconnue $F^* : \mathbf{x} \mapsto y$ à partir d'un ensemble d'entraînement $(y_i, \mathbf{x_i})_{i=1,\dots,n}$;
- Ils supposent que la fonction F^* peut être approchée par une **somme pondérée de modèles simples** f de paramètres θ:

$$F(\mathbf{x}) = \sum_{m=1}^{M} \beta_m f(\mathbf{x}, \boldsymbol{\theta}_m) \tag{11}$$

- Ils reposent sur une **modélisation additive par étapes** (*forward stagewise additive modeling*), qui décompose l'entraînement de ce modèle complexe en une **séquence d'entraînements de petits modèles**. Chaque étape de l'entraînement cherche le modèle simple f qui améliore la puissance prédictive du modèle complet, sans modifier les modèles précédents, puis l'ajoute de façon incrémentale à ces derniers:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \hat{\beta}_m f\left(\mathbf{x}_i, \hat{\boldsymbol{\theta}}_\mathbf{m}\right) \tag{12}$$

5.6.2. Les premières approches du *boosting*

5.6.2.1. Le *boosting* par repondération: Adaboost

Dans les années 1990, de nombreux travaux ont tâché de proposer des mise en application du *boosting* (L. Breiman [15], A. J. Grove and D. Schuurmans [16]) et ont comparé les mérites des différentes approches. Deux approches ressortent particulièrement de cette littérature:

Adaboost (Adaptive Boosting, Y. Freund and R. E. Schapire [17]) et la *Gradient Boosting Machine* (J. H. Friedman [18]). Ces deux approches reposent sur des principes très différents.

Le principe d'Adaboost consiste à pondérer les erreurs commises à chaque itération en donnant plus d'importance aux observations mal prédites, de façon à obliger les modèles simples à se concentrer sur les observations les plus difficiles à prédire. Voici une esquisse du fonctionnement d'AdaBoost:

- Un premier modèle simple est entraîné sur un jeu d'entraînement dans lequel toutes les observations ont le même poids.
- A l'issue de cette première itération, les observations mal prédites reçoivent une pondération plus élevée que les observations bien prédites, et un deuxième modèle est entraîné sur ce jeu d'entraînement pondéré.
- Ce deuxième modèle est ajouté au premier, puis on repondère à nouveau les observations en fonction de la qualité de prédiction de ce nouveau modèle.
- Cette procédure est répétée en ajoutant de nouveaux modèles et en ajustant les pondérations.

L'algorithme Adaboost a été au coeur de la littérature sur le *boosting* à la fin des années 1990 et dans les années 2000, en raison de ses performances sur les problèmes de classification binaire. Il a toutefois été progressivement remplacé par les algorithmes de *gradient boosting* mis au point quelques années plus tard.

5.6.2.2. L'invention du *boosting boosting* : la *Gradient Boosting Machine*

La *Gradient Boosting Machine* (GBM) propose une approche assez différente: elle introduit le *gradient boosting* en reformulant le *boosting* sous la forme d'un problème d'optimisation qui se résout par une approche itérative de descente de gradient. Cette approche repose entièrement sur la notion de **fonction de perte**, qui mesure l'écart entre la variable-cible et la prédiction du modèle. La mécanique de la *Gradient Boosting Machine* est présentée de façon formelle dans l'encadré ci-dessous; en voici une présentation intuitive:

- Le modèle global est **initialisé** à partir des données d'entraînement. A ce stade, le modèle prédit en général la moyenne de la variable-cible pour toutes les observations.
- Première itération de *boosting*:
  - On calcule la dérivée partielle (*gradient*) de la fonction de perte par rapport à la prédiction pour chaque observation de l'ensemble d'entraînement. Cette dérivée partielle est parfois appelée **pseudo-résidu**. L'opposé de ce gradient indique à la fois dans quelle direction et dans quelle ampleur la prédiction devrait être modifiée afin de réduire la perte (ou autrement dit afin de rapprocher la prédiction de la vraie valeur).

    - Un premier arbre est entraîné à prédire l'opposé du gradient de la fonction de perte.
    - Cet arbre est ajouté au modèle global (après multiplication par un facteur d'échelle).
- Deuxième itération de *boosting*: on calcule à nouveau la dérivée partielle de la fonction de perte par rapport aux nouvelles prédictions du modèle global, puis un deuxième arbre est entraîné à prédire l'opposé du gradient de la fonction de perte, et enfin cet arbre est ajouté au modèle global.
- Cette procédure est répétée en ajoutant de nouveaux modèles et en recalculant le gradient à chaque étape.
- La qualité du modèle final est évaluée sur un ensemble de test.

L'approche de *gradient boosting* proposée par J. H. Friedman [18] présente deux grands avantages. D'une part, **toute la mécanique du *gradient boosting* est indépendante de la fonction de perte choisie et de la nature du problème modélisé**. Cette approche peut donc être utilisée avec n'importe quelle fonction de perte différentiable, ce qui permet d'appliquer le *gradient boosting* à de multiples problèmes (régression, classification binaire ou multiclasse, *learning-to-rank*…). D'autre part, **le *gradient boosting* offre souvent des performances comparables ou supérieures aux autres approches de *boosting***. Le *gradient boosting* d'arbres de décision (*Gradient boosted Decision Trees* - GBDT) est donc devenue l'approche de référence en matière de *boosting*: toutes les implémentations modernes du *gradient boosting* comme `scikit-learn`, `XGBoost`, `LightGBM`, et `CatBoost` sont des extensions et améliorations de la *Gradient Boosting Machine*.

ℹ Présentation formelle de la *Gradient Boosting Machine* (J. H. Friedman [18])

On dispose d'un jeu de données $(x_i, y_i)_{i=1}^n$ avec $x_i \in \mathbb{R}^m$ et une cible y_i. On définit une fonction de perte l qui mesure la distance entre la prédiction $\hat{y}$ et la vraie valeur y. Elle présente généralement les propriétés suivantes: elle est dérivable deux fois, atteint un minimum lorsque $\hat{y} = y$, et sa dérivée seconde par rapport à $\hat{y}$ est positive. On veut entraîner un modèle comprenant m arbres, chacun étant défini par les paramètres $\mathbf{a_m}$ (règles de décision et valeurs des feuilles terminales):

$$\hat{y}_i = F(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i) \tag{13}$$

Procédure de construction du modèle:

1. Initialiser le modèle avec $F_0(\mathbf{x}) = f_0(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^n y_i$.
2. Pour $m = 1, ..., M$:
   (a) Calculer le gradient (les pseudo-résidus) à l'issue des $m-1$ étapes précédentes: $g_{im} = \frac{\partial l(y_i, F_{m-1}(\mathbf{x}))}{\partial F_{m-1}(\mathbf{x})}$
   (b) Entraîner le m-ième *weak learner*: on cherche l'arbre f_m qui prédit le mieux l'opposé du gradient de la fonction de perte: $\hat{\mathbf{a}}_\mathbf{m} = \arg\min_\mathbf{a} \sum_{i=1}^n \left(-g_{im} - f_m(\mathbf{x}_i, \mathbf{a})\right)^2$
   (c) Mettre à jour le modèle global: $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \rho f_m(\mathbf{x}_i, \hat{\mathbf{a}}_\mathbf{m})$ avec ρ le taux d'apprentissage (*learning rate*) dont la raison d'être est présentée dans la section Section 5.6.5.

5.6.3. Comment fonctionne le *gradient boosting*?

La méthode de *gradient boosting* proposée par J. H. Friedman [18] a fait l'objet de multiples implémentations, parmi lesquelles `XGBoost` (T. Chen and C. Guestrin [19]), `LightGBM` (G. Ke *et al.* [20]), `CatBoost` (L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin [21]) et `scikit-learn`. Ces implémentations sont proches les unes des autres, et ne diffèrent que sur des points relativement mineurs. En revanche, elles s'éloignent quelque peu de la formulation initiale de la *Gradient Boosting Machine*, afin d'optimiser la construction des arbres. Bien comprendre la mécanique interne de ces implémentations s'avère important en pratique, notamment pour appréhender le rôle des multiples hyperparamètres. Cette section présente donc la mécanique d'ensemble de ces implémentations, en s'appuyant sur l'implémentation proposée par XGBoost.[3]

[3]Cette partie reprend la structure et les notations de la partie 2 de T. Chen and C. Guestrin [19].

5.6.3.1. Le modèle à entraîner

On dispose d'un jeu de données $(x_i, y_i)_{i=1}^n$ avec $x_i \in \mathbb{R}^m$ et une cible y_i. On veut entraîner un modèle global qui soit une somme de K arbres de régression ou de classification: $\hat{y}_i = F(\mathbf{x}_i) = \sum_{k=1}^K f_k(x_i)$. On rappelle que chaque arbre f est défini par trois paramètres:

- sa **structure** qui est une fonction $q : \mathbb{R}^m \to \{1, ..., T\}$ qui à un vecteur $\mathbf{x}$ de dimension m associe une feuille terminale de l'arbre;cette structure est définie par l'ensemble des règles de décision de l'arbre;
- son **nombre de feuilles terminales** T;
- les **prédictions** figurant sur ses feuilles terminales $\mathbf{w} \in \mathbb{R}^T$ (appelées poids ou *weights*).

Pour entraîner ce modèle, l'algorithme `XGBoost` minimise une fonction-objectif qui comporte à la fois une **fonction de perte**, et un **terme de régularisation**:

$$\mathcal{L} = \underbrace{\sum_{i=1}^{n} \ell(y_i, F(x_i))}_{\substack{\text{Perte sur les} \\ \text{observations}}} + \underbrace{\sum_{k} \Omega(f_k)}_{\substack{\text{Fonction de} \\ \text{régularisation}}} \text{ avec } \Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{k=1}^{K} \sum_{j=1}^{T_k} w_j^2 \tag{14}$$

Dans cette expression:

- La fonction de perte ℓ mesure la distance entre la prédiction $\hat{y}$ et la vraie valeur y (exemples: erreur quadratique moyenne, erreur absolue moyenne, perte d'entropie croisée binaire, etc.). Elle présente généralement les propriétés suivantes: elle est dérivable deux fois, atteint un minimum lorsque $\hat{y} = y$, et sa dérivée seconde par rapport à $\hat{y}$ est positive.
- Le terme de régularisation $\Omega(f)$ pénalise la complexité de chaque arbre f via deux termes: le terme γT pénalise les arbres avec un grand nombre de feuilles (T élevé) et le terme $\frac{1}{2}\lambda \sum_{j=1}^{T_t} w_j^2$ pénalise les arbres avec des poids élevés (w_j élevés en valeur absolue). Cette pénalisation privilégie les arbres plus « simples » (moins de feuilles terminales, poids de feuilles plus petits) afin d'éviter le sur-ajustement (*overfitting*). γ et λ sont des hyperparamètres de régularisation qui contrôlent la complexité de l'arbre.

5.6.3.2. Le principe d'entraînement séquentiel

On pourrait essayer d'entraîner directement ce modèle complet, en déterminant en une seule étape toutes les fonctions f telles que:

$$F = \arg \min_{f_1, \cdots, f_K} \left(\sum_{i=1}^{n} \ell \left(y_i, \sum_{k=1}^{K} f_k(x_i) \right) + \gamma T + \frac{1}{2}\lambda \sum_{k=1}^{K} \sum_{j=1}^{T_k} w_j^2 \right) \tag{15}$$

En réalité, le modèle complet est impossible à entraîner en une seule fois, car pour cela il faudrait déterminer *simultanément* K fonctions relativement complexes. Le principe du *boosting* consiste donc à construire le modèle complet de façon itérative, par **ajout successif d'arbres**. À l'itération t, on ajoute un nouvel arbre f_t pour améliorer la prédiction actuelle $\hat{y}_i^{(t-1)}$. Ainsi,

en écrivant la relation, $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$, on peut réécrire la fonction-objectif Equation 14 comme ceci:

$$L^{(t)} = \sum_{i=1}^{n} \ell\Big(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\Big) + \sum_k \Omega(f_k) \tag{16}$$

L'objectif de l'itération t devient donc de trouver le nouvel arbre f_t qui minimise cette fonction-objectif. Pour ce faire, l'algorithme doit accomplir deux choses: **déterminer la structure** de l'arbre (c'est-à-dire choisir les règles de décision qui définissent l'arbre) en tenant compte de la régularisation et **calculer la meilleure valeur w_j** pour chaque feuille terminale (c'est-à-dire la valeur qui **minimise** la fonction de perte).

La construction de l'arbre se fait selon une **approche gloutonne**, de façon très similaire à la construction d'un arbre CART (voir section Section 3): on divise les données d'entraînement en sous-régions de plus en plus petites, tant que cela génère une réduction "suffisante" de la fonction de perte. Deux différences notables séparent toutefois le *gradient boosting* des arbres CART et des forêts aléatoires:

- **les arbres utilisés dans le *gradient boosting* sont souvent relativement simples et peu profonds**. Ainsi, lors de l'entraînement de chaque arbre le partitionnement récursif des données s'arrête généralement assez tôt, soit lorsqu'est atteinte une des limites de complexité définies *a priori* (profondeur maximale de l'arbre, nombre maximal de feuilles, nombre minimal d'observations par feuille terminale), soit lorsque l'algorithme ne parvient plus à trouver de partitionnement intéressant (c'est-à-dire qui permette de réduire suffisamment la perte).
- **l'algorithme de détermination des règles de décision utilise un critère reposant sur le gradient de la fonction de perte** plutôt que sur une mesure d'impureté. Ce point est décrit plus précisément dans la section suivante.

5.6.3.3. La mécanique interne du *gradient boosting*

Les paragraphes qui suivent détaillent le processus de construction du t-ième arbre, et les équations utilisées dans ce processus.

5.6.3.3.1. Étape 1: Faire apparaître le gradient de la fonction de perte

On souhaite construire le t-ième arbre qui minimise la perte, conditionnellement aux données d'entraînement et aux $t-1$ arbres déjà construits. Formellement d'après l'équation Equation 16 on cherche un arbre f_t tel que

$$\hat{f}_t = \arg\min_f \mathcal{L}^{(t)} = \arg\min_f \sum_{i=1}^{n} \ell\Big(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\Big) + \sum_k \Omega(f_k) \tag{17}$$

Pour simplifier la construction de ce t-ième arbre, `XGBoost` reformule la fonction de perte en utilisant une **approximation de second ordre**. Plus précisément, on fait un développement limité d'ordre 2 de $l\Big(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)\Big)$ au voisinage de $\hat{y}_i^{(t-1)}$, en considérant que la prédiction

du t-ième arbre $f_t(\mathbf{x}_i)$ est un incrément de petite taille. On obtient alors la fonction de perte approchée $\mathcal{L}^{(t)}$:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[\underbrace{l\left(y_i, \hat{y}_i^{(t-1)}\right)}_{\text{(A)}} + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \underbrace{\sum_{k=1}^{t-1} \Omega(f_k)}_{\text{(B)}} + \Omega(f_t) \tag{18}$$

avec

$$g_i = \frac{\partial l\left(y_i, \hat{y}_i^{(t-1)}\right)}{\partial \hat{y}_i^{(t-1)}} \text{et} h_i = \frac{\partial^2 l\left(y_i, \hat{y}_i^{(t-1)}\right)}{\partial \hat{y}_i^{((t-1))^2}} \tag{19}$$

Les termes g_i et h_i désignent respectivement le **gradient** (dérivée première) et la **hessienne** (dérivée seconde) de la fonction de perte par rapport à la variable prédite pour l'observation i à l'issue de la $t-1$-ième étape de l'entraînement. Dans cette équation, les termes (A) et (B) sont constants car ils ne dépendent que des $t-1$ arbres précédents qui ont déjà été entraînés et qui ne sont pas modifiés par l'entraînement du t-ième arbre. On peut donc retirer ces termes pour obtenir la fonction-objectif simplifiée $\tilde{\mathcal{L}}^{(t)}$ qui sera utilisée en pratique pour l'entraînement du t-ième arbre:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i [f_t(\mathbf{x}_i)]^2 \right] + \Omega(f_t) \tag{20}$$

On cherche donc désormais un arbre f_t tel que

$$\hat{f}_t = \arg\min_f \tilde{\mathcal{L}}^{(t)} = \arg\min_f \sum_{i=1}^{n} \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i [f_t(\mathbf{x}_i)]^2 \right] + \Omega(f_t) \tag{21}$$

5.6.3.3.2. Étape 2: calculer les poids optimaux conditionnellement à une structure d'arbre

A partir de l'équation Equation 20, il est possible de faire apparaître, puis de calculer les poids w_j du t-ième arbre. En effet, l'arbre f_t peut être vu comme une fonction constante par morceaux du type $f_t(x) = w_{q(x)}$ où $q(x)$ est une fonction qui assigne un indice de feuille (un entier entre 1 et T) à chaque observation x, et w_j est le poids (valeur de sortie) de la j-ième terminale. En regroupant les observations i tombant dans la feuille j dans l'ensemble $I_j = \{i \mid q(x_i) = j\}$, la fonction de perte approchée peut être réécrite sous la forme:

$$\begin{aligned}
\tilde{L}^{(t)} &= \sum_{j=1}^{T} \sum_{i \in I_j} \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i [f_t(\mathbf{x}_i)]^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_i^2 \\
&= \sum_{j=1}^{T} \sum_{i \in I_j} \left[g_i w_j + \frac{1}{2} h_i w_j^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_i^2 \\
&= \sum_{j=1}^{T} \left[w_j \sum_{i \in I_j} g_i + \frac{1}{2} w_j^2 \left(\sum_{i \in I_j} h_i + \lambda \right) \right] + \gamma T
\end{aligned} \tag{22}$$

Dans la dernière expression, la fonction de perte simplifiée se reformule comme une combinaison quadratique des poids w_j, dans laquelle les dérivées première et seconde de la fonction de perte interviennent sous forme de pondérations ($\sum_{i \in I_j} g_i$ et $\sum_{i \in I_j} h_i$). Cette expression peut elle-même s'écrire comme la somme sur l'ensemble des feuilles de la fonction de perte relative à chaque feuille j :

$$\tilde{L}^{(t)} = \sum_{j=1}^{T} k(w_j) + \gamma T \tag{23}$$

avec $k(w_j)$ la perte relative à la feuille j :

$$k(w_j) = \sum_{i \in I_j} \left(g_i w_j + \frac{1}{2} h_i w_j^2 \right) + \frac{1}{2} \lambda w_j^2 \tag{24}$$

Pour un arbre de structure donnée, la valeur optimale w_j^* de la feuille j, c'est-à-dire la valeur qui minimise la contribution de la feuille à la fonction de perte globale se calcule facilement (en résolvant pour chaque feuille j la condition du premier ordre $\frac{\partial k}{\partial w_j} = 0$):

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{25}$$

On déduit alors, pour une structure d'arbre q donnée, la valeur optimale de la fonction-objectif pour le t-ième arbre:

$$\tilde{L}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{\left(\sum_{i \in I_j} g_i \right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \tag{26}$$

L'équation Equation 26 est utile en pratique car elle permet de comparer la qualité de deux arbres candidats, et de déterminer immédiatement lequel est le meilleur. On pourrait même penser que cette équation est à elle seule suffisante pour choisir le t-ième arbre: il suffirait d'énumérer les arbres possibles, de calculer la qualité de chacun d'entre eux, et de retenir le meilleur. En réalité, cette approche est inemployable en pratique car le nombre d'arbres possibles est extrêmement élevé. Par conséquent, cette équation n'est pas utilisée telle quelle, mais sert à comparer les règles de décision possibles à chaque étape d'une optimisation gloutonne, de façon à trouver la structure q^* du t-ième arbre.

5.6.3.3.3. Étape 3: déterminer la structure de l'arbre

La méthode de construction des arbres dans les algorithmes de *gradient boosting* est très similaire à celle décrite dans la partie sur les arbres de décision: le t-ième arbre est construit de façon gloutonne (*greedy*), par un algorithme de détermination des règles de décision (*split finding algorithm*) qui énumère, évalue et compare les règles de décision possibles par une double boucle sur les variables et les valeurs prises par ces variables. La règle de décision retenue à chaque étape sera simplement celle dont le gain est le plus élevé.

La grande différence avec les arbres CART et les forêts aléatoires est que ces algorithmes utilisent l'équation Equation 26 (et non une mesure d'impureté) pour évaluer les règles de décision candidates. Plus précisément, à partir de l'équation Equation 26 la réduction de perte associée au partitionnement de la feuille I en deux noeuds-enfants gauche (I_L) et droit (I_R) à l'aide d'une certaine règle de décision s'écrit comme:

$$\Delta\tilde{L}^{(t)} = \frac{1}{2}\Bigg[\underbrace{\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda}}_{\text{Perte du noeud-enfant gauche}} + \underbrace{\frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i + \lambda}}_{\text{Perte du noeud-enfant droit}} - \underbrace{\frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i + \lambda}}_{\text{Perte du noeud d'origine}}\Bigg] - \gamma \quad (27)$$

Cette réduction de perte est simplement la différence entre la somme des pertes après partitionnement (noeud-enfant gauche + noeud-enfant droit) et la perte avant partitionnement. Le terme γ est le terme de régularisation qui mesure le coût associé à la création d'un noeud supplémentaire. Si pour un noeud donné, la meilleure règle de décision candidate est associée à un $\Delta\tilde{L}^{(t)}$ strictement positif, alors cette règle de décision fait baisser la perte globale. On peut alors l'utiliser pour définir deux nouveaux noeuds-enfants. Sinon, on renonce à scinder ce noeud.

> ℹ Optimisations des algorithmes
>
> **L'algorithme de détermination des règles de décision est le composant le plus intense en calcul des algorithmes de *gradient boosting*.** Les différentes implémentations du *gradient boosting* proposent donc de multiples améliorations et optimisations visant à le rendre le plus efficace possible. Certaines de ces optimisations sont présentées dans la partie Section 5.7.

5.6.3.3.4. Étape 4: Ajouter des arbres jusqu'à atteindre un critère d'arrêt

Une fois que la structure du t-ième arbre a été définie et que les valeurs des feuilles terminales ont été calculées, cet arbre est ajouté au modèle global, et la prédiction est mise à jour par la formule suivante:

$$F_t(x) = F_{t-1}(x) + \eta f_t(x) \quad (28)$$

Puis les valeurs des gradients et hessiennes sont mises à jour pour chaque observation par les formules:

$$g_i = \frac{\partial l\left(y_i, \hat{y}_i^{(t)}\right)}{\partial \hat{y}_i^{(t)}} \text{et} h_i = \frac{\partial^2 l\left(y_i, \hat{y}_i^{(t)}\right)}{\partial \hat{y}_i^{((t))^2}} \quad (29)$$

Il est alors possible de commencer l'entraînement de l'arbre suivant, selon la même logique que précédemment. Le processus de construction des arbres se poursuit jusqu'à atteindre soit le nombre maximum d'arbres autorisé dans le modèle (K), soit un autre **critère d'arrêt** (par exemple, une réduction de perte minimale par arbre).

5.6.4. Un exemple simple: la régression avec perte quadratique

La présentation qui précède peut donner l'impression que la mécanique du *gradient boosting* est complexe, abstraite et difficile à comprendre. Celle-ci devient beaucoup plus facile à saisir lorsqu'on l'illustre dans un cas simple et intuitif. Prenons ainsi un problème de régression, où l'on cherche à prédire une variable continue, avec une perte quadratique: $\ell(y,\hat{y}) = \frac{1}{2}(y-\hat{y})^2$ et sans terme de régularisation. Dans ce cas particulier, le gradient de la fonction de perte est égal à l'opposé du **résidu** du modèle $\hat{y} - y$, et la hessienne de la fonction de perte est égale à 1. L'équation Equation 25 qui donne le poids optimal de la feuille j devient donc:

$$w_j^* = \frac{\sum_{i \in I_j} y - \hat{y}}{|I_j|} \tag{30}$$

où I_j désigne le nombre d'observations dans la feuille j. Ce résultat est particulièrement intuitif : la prédiction de l'arbre pour chaque feuille terminale est égale à la moyenne des résidus (erreurs) des observations qui s'y trouvent. Ainsi, à chaque étape, le t-ième arbre corrige les erreurs laissées par les modèle précédents en annulant le résidu moyen du modèle complet dans chaque feuille terminale.

Cet exemple simple illustre bien la démarche du *gradient boosting* : améliorer progressivement la qualité des prédictions en corrigeant les erreurs résiduelles, chaque nouvel arbre tâchant de capter ce que les arbres précédents n'ont pas bien appris. Par ailleurs, elle permet de comprendre pourquoi le gradient de la fonction de perte est fréquemment désigné par le terme de *pseudo-résidu*.

5.6.5. Le grand ennemi du *gradient boosting*: le surajustement

Une différence majeure entre les forêts aléatoires et les algorithmes de *boosting* est que ces derniers ne contiennent en eux-mêmes aucune limite au surajustement, bien au contraire: le *gradient boosting* est un algorithme conçu pour approximer le plus précisément possible la relation entre X et y telle qu'elle apparaît dans les données d'entraînement, qu'il s'agisse d'un signal pertinent ou d'un bruit statistique. Par conséquent, **tous les algorithmes de *gradient boosting* sont par nature très vulnérables au surajustement**. En pratique, ce risque de surajustement croît au cours de l'entraînement: au fur et mesure que le nombre d'arbres augmente, l'algorithme capture de moins en moins de relations pertinentes entre X et y, et de plus en plus le bruit et les variations aléatoires de l'échantillon d'entraînement.

La lutte contre le surajustement est donc un enjeu majeur de l'entraînement des modèles de *gradient boosting*. De multiples méthodes ont été proposées pour lutter contre le surajustement:

- la **technique de réduction** (*shrinkage technique*) consiste à réduire l'influence de chaque arbre sur le modèle global en multipliant la prédiction de cet arbre par un facteur

d'échelle compris entre 0 et 1 au moment de mettre à jour le modèle par l'équation Equation 28. Ce facteur d'échelle est appelé **taux d'apprentissage** (*learning rate*); il s'agit du paramètre η dans l'équation Equation 28. L'avantage principal de cette technique est que le modèle s'ajuste progressivement aux données, et est moins altéré par les erreurs dues à des variations aléatoires ou au bruit dans les données. Un taux d'apprentissage bas et un nombre d'itérations suffisant permettent souvent d'obtenir un modèle final plus performant sur des données de test. L'inconvénient est que réduire le taux d'apprentissage nécessite d'augmenter le nombre d'itérations pour obtenir des performances comparables, ce qui peut rallonger le temps d'entraînement.

- l'**hyperparamètre de régularisation λ** intervient dans l'équation Equation 25 et réduit la valeur absolue des poids des feuilles terminales. Cet hyperparamètre contribue à ce que chaque arbre prédise des valeurs peu élevées. L'intuition est la suivante: lorsqu'une feuille terminale contient un poids w_i élevé en valeur absolue, ce poids est probablement dû au moins en partie à des observations inhabituelles ou aberrantes (et dont le gradient g_i prend une valeur extrême); il est donc préférable de réduire légèrement ce poids pour ne pas donner trop d'importance à ces points aberrants.
- l'**hyperparamètre de régularisation γ** intervient dans l'équation Equation 27. Ce paramètre mesure la réduction minimale de la perte requise pour qu'un nœud soit divisé; une valeur plus élevée aboutit à des arbres moins profonds et contribue à limiter le surajustement en empêchant l'algorithme de créer des *splits* dont l'apport est très faible et potentiellement dû à des variations non significatives des données d'entraînement.
- la dernière approche est inspirée des forêts aléatoires et consiste à **entraîner les arbres sur un échantillon d'observations et/ou de variables**. L'échantillonnage des observations permet de réduire l'influence des éventuels points extrêmes contenus dans les données (car ils n'apparaissent pas dans les données d'entraînement de certains arbres); l'échantillonnage des variables permet de varier les variables utilisées dans les règles de décision.

5.7. Sujets avancés: traitement des données pendant l'entraînement

5.7.1. Le traitement des variables continues: l'utilisation des histogrammes

L'algorithme de détermination des critères de partition (*split-finding algorithm*) est un enjeu de performance essentiel dans les méthodes ensemblistes. En effet, l'algorithme le plus simple qui consiste à énumérer tous les critères de partition possibles (en balayant toutes les valeurs de toutes les variables) s'avère très coûteux à utiliser dès lors que les données contiennent soit un grand nombre de variables, soit des variables continues prenant un

grand nombre de valeurs. C'est pourquoi cet algorithme a fait l'objet de multiples améliorations et optimisations visant à réduire leur coût computationnel sans dégrader la qualité des critères de partition.

L'utilisation d'histogrammes (*histogram-based algorithms*) est une approche efficace qui permet de réduire de manière significative le coût computationnel lié à la recherche des *splits* optimaux en discrétisant les variables continues. Elle est proposée par toutes les implémentations courantes du *gradient boosting* (XGBoost, LightGBM, CatBoost et scikit-learn), mais pas par les implémentations des forêts aléatoires (scikit-learn et `ranger`). Elle comprend deux caractéristiques principales:

- **Discrétisation**: avant le début de l'entraînement, chaque variable continue est discrétisée en un nombre limité d'intervalles (*bins*), construits le plus souvent à partir de ses quantiles. Ce processus est appelé *binning*. Par exemple, une variable continue uniformément distribuée de 0 à 100 peut être divisée en dix intervalles ($[0, 10), [10, 20), ..., [90, 100)$). Le nombre maximal de *bins* est un hyperparamètre qui peut parfois jouer un rôle important.
- **Énumération restreinte**: l'algorithme de détermination des critères de partition ne considère que les bornes des intervalles précédemment définies (10, 20, 30, etc. dans l'exemple précédent) et non l'ensemble des valeurs prises par les variables continues. Cette modification se traduit par une nette accélération de l'entraînement, dans la mesure où le nombre de *bins* est en général beaucoup plus faible que le nombre de valeurs uniques des variables continues[4]. Elle est en revanche sans effet notable sur les performances prédictives dans la plupart des cas.

5.7.2. Le traitement des variables catégorielles

Le traitement des variables catégorielles est l'un des points les plus délicats et les plus complexes dans les méthodes ensemblistes. Cette section présente les approches possibles et récapitule quelles approches sont proposées par chaque implémentation.

5.7.2.1. Une diversité d'approches

A l'origine, les arbres CART ont été conçus pour partitionner l'espace en utilisant exclusivement des variables numériques, et ne pouvaient pas mobiliser les variables catégorielles sans un retraitement préalable. Plusieurs **méthodes d'encodage** (*one-hot-encoding*, *ordinal encoding*, *target encoding*) ont donc été développées pour surmonter ce problème en transformant les variables catégorielles en variables numériques; elles ne sont d'ailleurs pas spécifiques aux méthodes ensemblistes à base d'arbres. Depuis le milieu des années 2010, une nouvelle approche

[4]Voir cette page de la documentation de scikit-learn pour plus d'éléments.

efficace et spécifique aux méthodes ensemblistes a été introduite dans les implémentations du *gradient boosting* (*native support for categorical features*). Ce paragraphe présente ces différentes approches des variables catégorielles, et précise quelle approche est proposée par les différentes implémentations.

Les trois approches d'encodage les plus courantes sont les suivantes:

- Le ***one-hot encoding*** consiste à transformer une variable catégorielle en une série de variables binaires qui représentent chacune une modalité de la variable; pour chaque observation, seule la colonne correspondant à la modalité de la variable catégorielle aura la valeur 1, et toutes les autres auront la valeur 0. Cette approche permet de représenter des catégories de manière numérique de façon très simple et sans leur attribuer un ordre. Toutefois, le *one-hot encoding* augmente fortement la dimensionnalité des données (ce qui ralentit l'entraînement et augmente les besoins en mémoire), et est inutilisable lorsque les variables catégorielles présentent un nombre élevé de modalités[5].
- L'***ordinal encoding*** consiste à attribuer un entier unique à chaque modalité d'une variable catégorielle. Par exemple, la catégorie "Sans diplôme" sera encodée par la valeur 0, la catégorie "Baccalauréat ou moins" sera encodée par 1, etc. Simple à mettre en œuvre, cette approche permet de remplacer la variable catégorielle par une unique variable numérique et est donc utile pour traiter les variables présentant un grand nombre de modalités, pour lesquelles le *one-hot encoding* est impraticable. Elle est particulièrement adaptée aux variables catégorielles qui sont naturellement ordonnées (exemples: niveau de diplôme, catégorie d'âge, étage d'un appartement…). En revanche, cette approche est peu adaptée aux variables non ordonnées (exemples: secteur d'activité, département, pays…) car elle introduit un ordre fictif qui peut perturber les modèles qui interprètent les entiers comme des valeurs ordonnées.
- Le ***target encoding*** consiste à remplacer chaque modalité d'une variable catégorielle par la moyenne de la variable cible pour cette modalité. Cette approche est notamment proposée par CatBoost[6] et par scikit-learn[7]. Comme l'*ordinal encoding*, cette approche permet d'obtenir une unique variable numérique et est donc utile pour traiter les variables présentant un grand nombre de modalités. Par ailleurs, le *target encoding* fonctionne bien avec la méthode des histogrammes décrite précédemment, dans la mesure où les valeurs encodées sont par construction ordonnées en fonction de leur

[5]Il est bien sûr possible de n'encoder que les modalités les plus fréquentes, et de regrouper toutes les autres dans une seule variable binaire.

[6]Le *target encoding* utilisé par CatBoost est présenté en détail dans L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin [21] et sur ce billet de blog .

[7]Voir cet exemple dans la documentation de scikit-learn.

association avec la variable cible. Toutefois, il est important d'utiliser le *target encoding* en lissant la moyenne ou en recourant à une validation croisée car il est sujet au surapprentissage. De plus, les implémentations existantes réalisent l'encodage une seule fois avant l'entraînement au niveau de l'ensemble des données d'entraînement, pas au niveau de chaque *split*; l'ordre des modalités qui résulte de l'encodage peut être peu pertinent sur certaines parties des données d'entraînement si celles-ci présentent un haut degré d'hétérogénéité.

La dernière approche, appelée **support natif des variables catégorielles** (***native support for categorical features***) n'est pas un encodage et est une spécificité des implémentations du *gradient boosting*. Cette approche a été introduite par LightGBM , puis reprise par XGBoost et scikit-learn . Dans cette approche, l'objectif est de déterminer le meilleur *split* directement à partir des modalités d'une variable catégorielle, en les séparant en deux sous-ensembles (par exemple : {'A', 'B', 'C'} et {'D', 'E', 'F'} pour une variable comportant six modalités). La difficulté est qu'il est souvent impossible en pratique de trouver le partitionnement optimal par une énumération exhaustive des partitions possibles, car il existe $2^{k-1}-1$ partitions possibles pour une variable à k modalités. C'est pourquoi le support natif des variables catégorielles repose sur une méthode plus efficace dont l'optimalité a été démontrée par W. D. Fisher [22]: à chaque split, les modalités sont triées selon $-\frac{\sum_i g_i}{\sum_i h_i}$[8], puis le meilleur *split* est choisi en testant les différentes divisions possibles de cette liste triée. Par exemple, si pour un *split* donné les modalités sont triées dans l'ordre ABDFEC, l'algorithme examinera les $k-1$ *splits* A|BDFEC, AB|DFEC, ABD|FEC, etc. Cette approche peut être considérée comme une variante optimisée du *target encoding*, avec deux différences notables: l'encodage des modalités se fait à partir du gradient et de la hessienne de la fonction de perte (et non à partir de y), et cet encodage a lieu à chaque *split* et non une fois pour toutes avant l'entraînement. Par ailleurs, cette approche peut rencontrer des difficultés quand les variables catégorielles comprennent un nombre très élevé des modalités (au-delà de quelques centaines).

5.7.2.2. Comparaison des différents approches

Déterminer quelle approche des variables catégorielles est adaptée dans une situation donnée n'est pas toujours aisé. Les cas d'usage des différentes approches peuvent être résumés comme ceci:

- le *one-hot-encoding* est adapté uniquement aux variables catégorielles comprenant peu de modalités;

[8]Il s'agit de l' équation donnant le poids optimal d'une feuille terminale, avec $\lambda = 0$.

- l'*ordinal encoding* est adapté aux variables catégorielles qui sont naturellement ordonnées, et ce quel que soit le nombre de modalités (car la variable catégorielle est implicitement convertie en variable numérique);
- le *target encoding* est adapté aux variables catégorielles comprenant un grand nombre de modalités et non naturellement ordonnées;
- le support natif des variables catégorielles est adapté à tous les types de variables catégorielles, à l'exception de celles qui comprennent un nombre très élevé des modalités (au-delà de quelques centaines).

Comparé aux autres approches, le support natif des variables catégorielles comporte plusieurs avantages: il permet de réduire le nombre de *splits*, d'obtenir des arbres plus simples et d'augmenter l'efficacité computationnelle de l'entraînement des arbres. Ces avantages apparaissent clairement avec un exemple[9]. Imaginons qu'on s'intéresse à une variable catégorielle prenant les modalités ABCDEF, et que sur une feuille donnée de l'arbre, le meilleur partitionnement sur cette variable soit ACF - BDE. Une approche de *one-hot-encoding* aura besoin de trois *splits* pour approximer ce partitionnement: un premier pour séparer A et BCDEF, un deuxième pour séparer C et BDEF et un troisième pour séparer F et BCDE (l'ordre des *splits* pouvant différer). Une approche d'*ordinal encoding* aura besoin de quatre *splits*: un *split* pour isoler A, un *split* pour isoler F, et deux splits pour isoler C (car C est au milieu de l'ordre des modalités). Enfin, une approche de *target encoding* aura besoin d'un nombre variable de *splits*, qui dépend de l'ordre des modalités dans l'*encoding*: si le *target encoding* a trié les modalités dans l'ordre ACFBDE, alors un unique *split* suffira; si l'ordre est très différent, alors il faudra davantage de *splits*. Inversement, le support natif des variables catégorielles identifiera immédiatement le partitionnement optimal, et ne fera qu'un seul *split*, ce qui simplifie la structure de l'arbre et accélère l'entraînement.

5.7.2.3. Approches intégrées aux implémentations des méthodes ensemblistes

Certaines implémentations des méthodes ensemblistes prennent en charge directement certaines des quatre approches présentées ci-dessus, auquel cas il est possible d'entraîner le modèle sur des données contenant des variables catégorielles sans *preprocessing* particulier; dans les autres cas, il faut préparer les données en amont de l'entraînement et de la prédiction, par exemple en utilisant un `ColumnTransformer` de `scikit-learn` (`OneHotEncoder()` , `TargetEncoder()` , `OrdinalEncoder()`). Le tableau et les notes ci-dessous résument quelles approches sont proposées par chaque implémentations des méthodes ensemblistes.

[9]Cet exemple s'appuie sur la documentation de `scikit-learn` .

Table 1 : *Preprocessing* des variables catégorielles dans les implémentations des méthodes ensemblistes

Approche	`ranger`	`scikit-learn`	`XGBoost`	`LightGBM`	`CatBoost`
One-hot-encoding	✗	✗	✓	✓	✓
Ordinal encoding	✓	✗	✗	✗	✗
Target encoding	✓	✗	✗	✗	✓
Support natif des variables catégorielles	✗	✓	✓	✓	✗

- `ranger`: le traitement des variables catégorielles (`factors`) est contrôlé par l'argument `respect.unordered.factors`, qui peut prendre trois valeurs: `ignore` (*ordinal encoding*), `order` (*target encoding*) et `partition` (essayer toutes les combinaisons possibles). L'usage de cet argument est détaillé dans la documentation de la fonction `ranger()`. L'usage de la modalité `partition` n'est pas recommandé. En revanche, `ranger` ne prend pas en charge le *one-hot-encoding*.
- `LightGBM`: cette implémentation utilise par défaut le *one-hot-encoding* pour les variables catégorielles comprenant 4 modalités ou moins, et le support natif des variables catégorielles pour les autres. Ce seuil peut être modifié via le paramètre `max_cat_to_onehot`. En revanche, `LightGBM` ne prend pas en charge ni l'*ordinal encoding* ni le *target encoding*.
- `XGBoost`: le traitement des variables catégorielles est contrôlé par l'argument `enable_categorical`. Si `enable_categorical = True` (en Python), alors `XGBoost` applique le support natif des variables catégorielles. Par ailleurs, le paramètre `max_cat_to_onehot` permet d'utiliser le *one-hot-encoding* pour les variables catégorielles comprenant moins de `max_cat_to_onehot` modalités, et le support natif pour les autres (voir la documentation). En revanche, `XGBoost` ne prend pas en charge ni l'*ordinal encoding* ni le *target encoding*.
- `CatBoost`: cette implémentation est celle qui propose le plus d'options relatives aux variables catégorielles. Par défaut, elle utilise le *one-hot-encoding* pour les variables catégorielles comprenant peu de modalités, et une variante de *target encoding* aux autres. Le seuil par défaut varie selon le type de tâche et peut être modifié via le paramètre `one_hot_max_size`. La documentation de `CatBoost` propose un *notebook* qui détaille les différents hyperparamètres.
- `scikit-learn`: l'implémentation du *gradient boosting* (`HistGradientBoostingClassifier` , `HistGradientBoostingRegressor`) propose le support natif des variables catégorielles. Cette implémentation ne propose pas

de *one-hot-encoding* pour les variables catégorielles comprenant peu de modalités, et ne prend pas en charge ni l'*ordinal encoding* ni le *target encoding*.

5.7.3. Le traitement des valeurs manquantes

Le **traitement des valeurs manquantes dans les variables explicatives est un problème qui peut être traité à deux moments: soit de façon classique lors de la préparation des données (*preprocessing*), soit directement lors de l'entraînement des modèles**, car toutes les implémentations de référence des méthodes ensemblistes proposent une prise en charge des valeurs manquantes au moment de l'entraînement. La première approche est décrite dans la section Section 7. Le paragraphe qui suit se concentre sur la seconde approche et s'attache à décrire en détail la méthode utilisée par chaque implémentation, car les documentations sont souvent incomplètes sur ce point.

- `XGBoost`: le paramètre `missing` permet de préciser quelle valeur dans les données doit être considérée comme manquante (`np.nan` par défaut). Les valeurs manquantes sont traitées en modifiant légèrement l'algorithme de détermination des critères de partition, de façon à ce qu'il recherche à la fois le meilleur *split* et le meilleur noeud-enfant vers lequel envoyer les valeurs manquantes. Pour ce faire, le gain associé à chaque règle de décision candidate est calculé de deux façons différentes: en mettant les valeurs manquantes dans le noeud de droite, puis en les mettant dans le noeud de gauche. Le gain le plus élevé des deux indique dans quel noeud les valeurs manquantes doivent être envoyées, conditionnellement à la règle de décision candidate. Une fois que toutes les règles candidates ont été examinées, on obtient simultanément la meilleure règle de décision, et la meilleure façon de traiter les valeurs manquantes. Cette approche est systématiquement appliquée aux variables numériques, et aux variables catégorielles si le support natif des variables catégorielles est activé (`enable_categorical = True`, voir la section Section 5.7.2). Ceci dit, il arrive fréquemment que la règle de décision optimale pour un noeud repose sur une variable qui ne comprend aucune valeur manquante dans la population du noeud à partitionner; en ce cas les valeurs manquantes sont envoyées à droite par défaut[10].
- `LightGBM`: seules les valeurs `na` sont considérées comme manquantes; elles sont traitées selon une approche très similaire à celle d'XGBoost. Dans le cas des variables numériques, l'algorithme de détermination des critères de partition recherche à la fois le meilleur *split* et le meilleur noeud-enfant vers lequel envoyer les valeurs manquantes. Toutefois, lorsque la règle de décision optimale pour un noeud repose sur une variable

[10]Voir ce post

qui ne comprend aucune valeur manquante dans la population du noeud à partitionner, toute valeur manquante rencontrée dans cette variable à l'étape de prédiction est convertie en 0, ce qui peut avoir des effets imprévisibles sur les prédictions[11]. Dans le cas des variables catégorielles, les valeurs manquantes sont systématiquement envoyées vers le noeud-enfant de droite.

- `ranger` propose également une approche similaire à celle d'XGBoost, à une différence près: l'algorithme de détermination des critères de partition recherche d'abord uniquement le meilleur *split*, puis une fois celui-ci trouvé, détermine vers quel noeud-enfant il est préférable d'envoyer éventuelles les valeurs manquantes. S'il n'y en a pas, les valeurs manquantes rencontrées en prédiction seront envoyées au noeud-enfant de gauche.
- `scikit-learn` propose également une approche similaire à celle d'XGBoost, à deux différences près: seules les valeurs `np.nan` sont considérées comme manquantes, et lorsqu'il n'y a pas de valeurs manquantes au moment de construire la règle de décision, les valeurs manquantes rencontrées en prédiction seront envoyées au noeud-enfant comprenant le plus d'observations[12].
- `CatBoost`: cette implémentation utilise une approche très simple pour les valeurs manquantes. S'agissant des variables numériques, l'algorithme remplace systématiquement les valeurs manquantes par une valeur soit extrêmement faible (inférieure à toutes les valeurs observées dans les données), soit extrêmement élevée (supérieure à toutes les valeurs observées), garantissant ainsi qu'elles seront toujours dirigées vers le noeud-enfant de gauche (respectivement vers le noeud-enfant de droite). Ce choix est contrôlé par l'hyperparamètre `nan_mode`. Pour les valeurs catégorielles, CatBoost traite les valeurs manquantes comme une catégorie distincte à part entière, en leur attribuant un encodage spécifique lors de la transformation ordonnée des variables catégorielles.

5.8. Sujets avancés: fonctionnement des algorithmes

5.8.1. Leaf-wise versus level-wise

5.8.2. La classification multiclasse non-ordonnée

Points d'attention:

- Comment sont construits les modèles *exactement*? En faisant k modèles en 1-versus-all?
- Quelle est la fonction de perte exacte?

[11] Voir cette issue . Ce comportement imprévisible peut aisément être résolu par une étape de *preprocessing* des variables numériques. Par exemple, le *transformer* `MinMaxScaler` de `scikit-learn`permet de s'assurer que ces variables prennent des valeurs comprises entre 0 et 1, ce qui signifie que les valeurs manquantes seront toujours envoyées à gauche par défaut

[12] Voir la documentation

- Comment les prédictions sont-elles agrégées *exactement*?
- A quoi doit ressembler la target? Doit-elle être encodée ou pas?
- Comment gérer les déséquilibres entre classes? Il me semble qu'il y a un peu partout des class weights.
- Comment peut-on récupérer les probas de chaque classe?

5.8.2.1. XGBoost

La classification multi-classe avec XGBoost permet de prédire une classe parmi plusieurs possibles pour chaque exemple.

- **Fonction objectif** : Pour la classification multi-classe, le paramètre `objective` doit généralement être défini sur `'multi:softmax'` (pour obtenir directement les étiquettes de classe) ou `'multi:softprob'` (pour obtenir les probabilités de chaque classe). Par défaut, le classifieur XGBoost utilise automatiquement `'multi:softprob'` s'il détecte plus de deux classes dans la variable cible.
- **Nombre de classes** : Il faut spécifier le nombre de classes avec le paramètre `num_class`, qui doit correspondre au nombre de valeurs uniques dans la variable cible.
- **Entraînement** : Pendant l'entraînement, XGBoost construit un modèle qui utilise une stratégie de type "un contre tous" (ou similaire) pour chaque classe, optimisant simultanément les probabilités ou les affectations de classe à l'aide de la fonction objectif.
- **Prédiction** : Lors de la prédiction, le modèle retourne soit la classe prédite (`multi:softmax`), soit une probabilité pour chaque classe (`multi:softprob`), selon l'objectif choisi.
- **Encodage de la cible** : La variable cible doit être encodée sous forme d'entiers (0, 1, 2, …) pour chaque classe, et non sous forme de variables binaires (one-hot encoding).

Cette méthode permet à XGBoost de gérer efficacement les problèmes multi-classes en optimisant la séparation des classes via le boosting de gradient.

5.8.2.2. LightGBM

LightGBM propose deux approches de la classification multiclasse: `multiclass` et `multiclassova`:

- `multiclassova` construit un modèle en mode one-vs-all pour chaque modalité, puis applique la fonction sigmoïde aux scores pour obtenir le résultat (pas clair); un point important est que les différents classifiers binaires sont entraînés indépendamment les uns des autres.
- `multiclass`

La classification multiclasse avec LightGBM repose sur l'algorithme d'arbre de décision avec renforcement de gradient (GBDT) et des optimisations spécifiques comme GOSS et EFB pour améliorer l'efficacité et la vitesse[1][5].

- **Paramétrage** : Pour une tâche multiclasse, il faut spécifier `objective='multiclass'` et indiquer le nombre de classes avec `num_class` (par exemple, `num_class=3` pour 3 classes)[4][7].
- **Encodage de la cible** : La variable cible doit être encodée sous forme d'entiers (0, 1, 2, …) correspondant à chaque classe[2][4].
- **Apprentissage** : LightGBM entraîne un modèle qui, pour chaque itération, construit `num_class` arbres (un par classe), soit un total de `num_class * num_iterations` arbres au total[7].
- **Prédiction** : Lors de la prédiction, le modèle retourne un vecteur de scores (probabilités) pour chaque classe, et la classe prédite est celle avec le score le plus élevé[2][3].
- **Pas de normalisation requise** : Aucune normalisation ou prétraitement particulier n'est nécessaire pour la variable cible, sauf l'encodage correct[2].

En résumé, LightGBM gère la classification multiclasse en optimisant simultanément la séparation de toutes les classes via le boosting de gradient, tout en restant très efficace sur de grands volumes de données[1][5].

Lorsque vous utilisez le modèle pour la classification multiclasse, vous pouvez obtenir :

- La **classe prédite** avec la méthode `predict()` : cela retourne la classe avec la probabilité la plus élevée pour chaque échantillon.
- La **probabilité de chaque classe** avec la méthode `predict_proba()` : cela retourne un tableau de probabilités pour chaque classe, de dimension (n_échantillons, n_classes), où chaque ligne correspond à un échantillon et chaque colonne à la probabilité d'appartenir à une classe donnée.

5.8.2.3. ranger

Extraits de la documentation:

> The tree type is determined by the type of the dependent variable. For factors classification trees are grown, for numeric values regression trees and for survival objects survival trees.
>
> With the probability option and factor dependent variable a probability forest is grown. Here, the node impurity is used for splitting, as in classification forests. Predictions are class probabilities for each sample. In contrast to other implementations, each tree returns a probability estimate and these estimates are averaged for the forest probability estimate.

Je comprends que par défaut ranger construit une RF qui prédit une classe; il faut l'option `probability = TRUE` pour avoir une probabilité.

- la variable-cible doit être de type `factor`.
-

5.8.3. La classification multiclasse ordonnée

Attention, là on est franchement *advanced*.

5.8.4. Les fonctions de perte custom

6. Une bien belle section

Principe: Conseils + mise en oeuvre pratique.

7. Préparation des données

Les méthodes ensemblistes à base d'arbres ne requièrent pas le même travail préparation des données que les méthodes économétriques traditionnelles. En particulier, certaines étapes indispensables à l'économétrie cessent d'être nécessaires. En revanche, d'autres, notamment le traitement des valeurs manquantes et des variables catégorielles. Cette section gagne à être lue après ou en parallèle de la section Section 5.7.

7.1. Préparation des variables explicatives

7.1.1. Quels sont les traitements inutiles?

Deux traitements usuels en économétrie ne sont pas nécessaires pour utiliser des méthodes ensemblistes:

- Il est inutile de normaliser ou de standardiser les variables numériques car c'est l'*ordre* défini par les valeurs qui est essentiel, pas les valeurs numériques elles-mêmes (voir la section Section 3). Pour la même raison, il est inutile de modifier ou supprimer les valeurs extrêmes.
- Il n'est pas indispensable de supprimer les variables corrélées car les méthodes ensemblistes sont robustes à la multicollinéarité. Ceci dit, réduire le nombre de variables peut accélérer légèrement l'entraînement des modèles.

7.1.2. Comment traiter les variables catégorielles?

Le **traitement des variables catégorielles** est un sujet plus complexe, car plusieurs approches sont possibles et certaines implémentations peuvent prendre en charge les variables catégorielles sans préparation particulière. Il est recommandé de lire la section Section 5.7.2 qui détaille les approches possibles. L'approche à privilégier *in fine* dépendra de trois facteurs: l'implémentation utilisée, l'existence d'un ordre des catégories, et le nombre total de catégories. Voici quelques recommandations générales:

- **il est nettement préférable de convertir les variables catégorielles en variables numériques lorsqu'elles comportent un ordre naturel** (exemples: âge, niveau de revenu, niveau de diplôme, niveau de densité urbaine…). Cette approche a l'avantage de simplifier considérablement l'utilisation des variables catégorielles, en les traitant comme des variables numériques classiques. Par exemple, l'âge exprimé sous forme de tranche d'âge ($[0;9]$, $[10;19]$, $[20;29]$, etc.) peut aisément être converti en variable numérique: la valeur 0 sera associée à la modalité $[0;9]$, la valeur 1 à la modalité $[10;19]$. L'encodeur `OrdinalEncoder` de `scikit-learn` permet d'automatiser cette

tâche de façon efficace. De la même façon, il est possible de convertir une date en variable numérique exprimée en jours écoulés depuis une date de référence.

- dans le cas des variables catégorielles non ordonnées (exemple: secteur d'activité, PCS, département…), **il est préférable de tester les approches intégrées aux implémentations des méthodes ensemblistes** avant se lancer dans la construction d'une approche *ad hoc*.
- **le *one-hot-encoding* ne doit pas être utilisé pour les variables catégorielles qui présentent un nombre élevé de modalités** (par exemple au-delà de 10 modalités).

7.1.3. Comment traiter les valeurs manquantes?

Dans la mesure où toutes les implémentations de référence des méthodes ensemblistes proposent une prise en charge des valeurs manquantes (voir la section Section 5.7.3), il est à première vue inutile de traiter les valeurs manquantes avant d'entraîner un modèle. Il peut néanmoins être utile de le faire, par exemple par une imputation déterministe, pour se prémunir d'éventuels problèmes difficiles à débugguer. *A minima*, si on veut utiliser le support des valeurs manquantes fonctionne correctement, **il faut impérativement vérifier que les valeurs manquantes sont bien codées avec la modalité considérée comme valeur manquante dans l'implémentation utilisée**. Il peut par exemple arriver que les valeurs manquantes soient codées avec 0, ou -999, ou une chaîne de caractères telle que `Z` au sein d'une variable numérique. En ce cas il faut recoder la modalité manquante.

7.1.4. Est-il utile de créer des variables additionnelles?

Il n'est pas toujours simple de savoir s'il est nécessaire d'ajouter des variables additionnelles dans les données d'entraînement des algorithmes ensemblistes. Il arrive d'ailleurs qu'on affirme que créer des variables additionnelles ne présente pas d'intérêt, dans la mesure où ces algorithmes sont naturellement capables de modéliser des relations complexes, non linéaires et faisant intervenir des interactions arbitraires entre variables. Cette affirmation n'est que partiellement vraie, comme le montrent les paragraphes suivants.

Une chose est sûre: **il est inutile d'ajouter aux données des variables numériques issues d'une transformation *monotone* d'une variable existante**. En effet, si une variable additionnelle x_2 est issue d'une transformation monotone de la variable x_1 déjà présente dans les données, alors les sous-régions qui peuvent être définies par une règle de décision basée sur x_2 sont identiques à celles qui peuvent être définies avec x_1. Par conséquent, x_2 ne permet pas d'affiner le partitionnement au-delà de ce qui était déjà possible avec x_1. Cela signifie en pratique qu'il faut éviter d'inclure des variables telles que le carré ou le cube d'une variable déjà présente dans les données.

Inversement, **une variable additionnelle peut s'avérer utile si elle est issue d'une transformation *non monotone* d'une variable, ou d'une transformation quelconque de deux ou plusieurs variables.** Voici deux exemples qui en illustrent l'intérêt:

- **Transformation *non monotone* d'une variable**: imaginons qu'on veuille prédire le prix moyen des glaces à Paris uniquement avec la date, exprimée sous forme numérique en jours écoulés depuis une date de référence. Il est probable que ce prix présente une saisonnalité marquée (prix élevé en été, faible en hiver). On peut alors envisager d'ajouter une variable issue d'une transformation sinusoïdale de la date; cette variable prendra une valeur élevée à certaines périodes et plus faible à d'autres, ce qui aidera à capter la saisonnalité.
- **Transformation quelconque de deux ou plusieurs variables**: imaginons qu'on veuille prédire la probabilité de faillite d'une entreprise à partir de seulement deux variables: son chiffre d'affaires et de son excédent brut d'exploitation (EBE). Ajouter le carré de l'EBE est inutile car il s'agit d'une transformation monotone de cette variable: les entreprises ayant un EBE faible ont un EBE carré faible, et ainsi de suite. En revanche, il est probablement utile d'ajouter le taux de marge (défini comme le ratio entre chiffre d'affaires et EBE), car un taux de marge faible peut aider à repérer des sous-groupes d'entreprises en difficulté, et ce indépendamment de leur taille.

Il faut noter que dans ces deux exemples, un algorithme ensembliste parviendrait probablement à capter la saisonnalité du prix des glaces ou à repérer les entreprises à faible taux de marge même en l'absence de variables additionnelles. Ce résultat s'obtiendrait néanmoins au prix d'un grand nombre de *splits* aboutissant à des arbres profonds et complexes, autrement dit d'un modèle inutilement complexe. Si elles ne sont donc effectivement pas strictement indispensables pour que les algorithmes soient performants, des variables additionnelles bien choisies ont néanmoins pour effet de faciliter et d'accélérer la construction du modèle.

7.2. Préparation de la variable-cible

Alors qu'elles ne sont pas sensibles aux valeurs des variables explicatives, les méthodes ensemblistes sont sensibles aux valeurs prises par la variable-cible. La préparation de la variable-cible obéit donc à des règles différentes de celles applicables aux variables explicatives:

- **Choisir soigneusement les transformations appliquées à la variable-cible**: on obtient des modèles très différents selon qu'on entraîne un algorithme avec pour variable-cible le prix de vente des logements, le prix au mètre carré ou le logarithme du prix au mètre carré.
- **Repérer et traiter les valeurs extrêmes, aberrantes ou erronées prises par la variable-cible**, soit en les corrigeant, soit en supprimant les observations concernés. Ce

point est particulièrement important lorsqu'on veut entraîner un algorithme de *gradient boosting*.

- **Traiter les valeurs manquantes**, soit en imputant une valeur, soit en supprimant les observations concernés. Le choix entre les deux approches n'est pas simple: imputer la variable-cible risque d'introduire des erreurs dans les données d'entraînement, mais les supprimer risque de déséquilibrer ces mêmes données. De façon générale, l'imputation doit être privilégiée uniquement lorsqu'on dispose de solides éléments étayant cette approche.

7.2.1. Train-test

Pas indispensable pour RF, mais souhaitable. Indispensable pour GB.

7.3. Evaluation des performances du modèle et optimisation des hyper-paramètres

7.3.1. Estimation de l'erreur par validation croisée

La validation croisée est une méthode d'évaluation couramment utilisée en apprentissage automatique pour estimer la capacité d'un modèle à généraliser les prédictions à de nouvelles données. Bien que l'évaluation par l'erreur *Out-of-Bag* (OOB) soit généralement suffisante pour les forêts aléatoires, la validation croisée permet d'obtenir une évaluation plus robuste, car moins sensible à l'échantillon d'entraînement, notamment sur des jeux de données de petite taille.

Concrètement, le jeu de donné est divisé en k sous-ensembles, un modèle est entraîné sur $k-1$ sous-ensembles et testé sur le sous-ensemble restant. L'opération est répétée k fois de manière à ce que chaque observation apparaisse au moins une fois dans l'échantillon test. L'erreur est ensuite moyennée sur l'ensemble des échantillons test.

Procédure de validation croisée:

La validation croisée la plus courante est la validation croisée en k sous-échantillons (*k-fold cross-validation*):

- **Division des données** : Le jeu de données est divisé en k sous-échantillons égaux, appelés folds. Typiquement, k est choisi entre 5 et 10, mais il peut être ajusté en fonction de la taille des données.
- **Entraînement et test** : Le modèle est entraîné sur k - 1 sous-échantillons et testé sur le sous-échantillon restant. Cette opération est répétée k fois, chaque sous-échantillon jouant à tour de rôle le rôle de jeu de test.
- **Calcul de la performance** : Les k performances obtenues (par exemple, l'erreur quadratique moyenne pour une régression, ou l'accuracy (*exactitude*) pour une classification) sont moyennées pour obtenir une estimation finale de la performance du modèle.

Avantages de la validation croisée:

- **Utilisation optimale des données** : En particulier lorsque les données sont limitées, la validation croisée maximise l'utilisation de l'ensemble des données en permettant à chaque échantillon de contribuer à la fois à l'entraînement et au test.
- **Réduction de la variance** : En utilisant plusieurs divisions des données, on obtient une estimation de la performance moins sensible aux particularités d'une seule division.

Bien que plus coûteuse en termes de calcul, la validation croisée est souvent préférée lorsque les données sont limitées ou lorsque l'on souhaite évaluer différents modèles ou hyperparamètres avec précision.

Leave-One-Out Cross-Validation (LOOCV) : Il s'agit d'un cas particulier où le nombre de sous-échantillons est égal à la taille du jeu de données. En d'autres termes, chaque échantillon est utilisé une fois comme jeu de test, et tous les autres échantillons pour l'entraînement. LOOCV fournit une estimation très précise de la performance, mais est très coûteuse en temps de calcul, surtout pour de grands jeux de données.

7.3.2. Choix des hyper-paramètres du modèle

L'estimation Out-of-Bag (OOB) et la validation croisée sont deux méthodes clés pour optimiser les hyper-paramètres d'une forêt aléatoire. Les deux approches permettent de comparer les performances obtenues pour différentes combinaisons d'hyper-paramètres et de sélectionner celles qui maximisent les performances prédictives, l'OOB étant souvent plus rapide et moins coûteuse, tandis que la validation croisée est plus fiable dans des situations où le surapprentissage est un risque important (P. Probst, M. N. Wright, and A.-L. Boulesteix [23]).

Il convient de définir une stratégie d'optimisation des hyperparamètres pour ne pas perdre de temps à tester trop de jeux d'hyperparamètres. Plusieurs stratégies existent pour y parvenir, les principales sont exposées dans la section Section 8. Les implémentations des forêts aléatoires disponibles en R et en Python permettent d'optimiser aisément les principaux hyper-paramètres des forêts aléatoires.

7.3.2.1. Méthodes de recherche exhaustives

- **Recherche sur grille** (Grid Search): Cette approche simple explore toutes les combinaisons possibles d'hyperparamètres définis sur une grille. Les paramètres continus doivent être discrétisés au préalable. La méthode est exhaustive mais coûteuse en calcul, surtout pour un grand nombre d'hyperparamètres.
- **Recherche aléatoire** (Random Search): Plus efficace que la recherche sur grille, cette méthode échantillonne aléatoirement les valeurs des hyperparamètres dans un espace défini. Bergstra et Bengio (2012) ont démontré sa supériorité pour les réseaux

neuronaux, et elle est également pertinente pour les forêts aléatoires. La distribution d'échantillonnage est souvent uniforme.

7.3.2.2. Optimisation séquentielle/itérative basée sur un modèle (SMBO)

La méthode SMBO (Sequential model-based optimization) est une approche plus efficace que les précédentes car elle s'appuie sur les résultats des évaluations déjà effectuées pour guider la recherche des prochains hyper-paramètres à tester (P. Probst, M. N. Wright, and A.-L. Boulesteix [23]).

Voici les étapes clés de cette méthode:

- Définition du problème: On spécifie une mesure d'évaluation (ex: AUC pour la classification, MSE pour la régression), une stratégie d'évaluation (ex: validation croisée k-fold), et l'espace des hyperparamètres à explorer.
- Initialisation: échantillonner aléatoirement des points dans l'espace des hyperparamètres et évaluer leurs performances.
- Boucle itérative :
  - Construction d'un modèle de substitution (surrogate model): un modèle de régression (ex: krigeage ou une forêt aléatoire) est ajusté aux données déjà observées. Ce modèle prédit la performance en fonction des hyperparamètres.
  - Sélection d'un nouvel hyperparamètre: un critère basé sur le modèle de substitution sélectionne le prochain ensemble d'hyperparamètres à évaluer. Ce critère vise à explorer des régions prometteuses de l'espace des hyperparamètres qui n'ont pas encore été suffisamment explorées.
  - Évaluer les points proposés et les ajouter à l'ensemble déjà exploré: la performance du nouvel ensemble d'hyperparamètres est évaluée et ajoutée à l'ensemble des données d'apprentissage du modèle de substitution afin d'orienter les recherches vers de nouveaux hyper-paramètres prometteurs.

8. Guide d'usage des forêts aléatoires

Ce guide d'entraînement des forêts aléatoires rassemble et synthétise des recommandations sur l'entraînement des forêts aléatoires disponibles dans la littérature, en particulier dans P. Probst, M. N. Wright, and A.-L. Boulesteix [23] et G. Biau and E. Scornet [24]. Ce guide comporte un certain nombre de choix méthodologiques forts, comme les implémentations recommandées ou la procédure proposée pour l'optimisation des hyperparamètres, et d'autres choix pertinents sont évidemment possibles. C'est pourquoi les recommandations de ce guide doivent être considérées comme un point de départ raisonnable, pas comme un ensemble de règles devant être respectées à tout prix.

8.1. Quelles implémentations utiliser?

Il existe de multiples implémentations des forêts aléatoires. Le présent document présente et recommande l'usage de deux implémentations de référence: le *package* `R` `ranger` et le *package* `Python` `scikit-learn` pour leur rigueur, leur efficacité et leur simplicité d'utilisation. Il est à noter qu'il est possible d'entraîner des forêts aléatoires avec les algorithmes `XGBoost` et `LightGBM`, mais il s'agit d'un usage avancé qui n'est pas recommandé en première approche. Cette approche est présentée dans la partie **REFERENCE A LA PARTIE USAGE AVANCE**.

8.2. Les hyperparamètres clés des forêts aléatoires

Cette section décrit en détail les principaux hyperparamètres des forêts aléatoires listés dans le tableau Table 2. Les noms des hyperparamètres utilisés sont ceux figurant dans le *package* `R` `ranger`, et dans le *package* `Python` `scikit-learn`. Il arrive qu'ils portent un nom différent dans d'autres implémentations des forêts aléatoires, mais il est généralement facile de s'y retrouver en lisant attentivement la documentation.

Table 2 : Les principaux hyperparamètres des forêts aléatoires

Hyperparamètre		**Description**
`ranger`	`scikit-learn`	
`num.trees`	`n_estimators`	Le nombre d'arbres
`mtry`	`max_features`	Le nombre ou la proportion de variables candidates à chaque noeud
`sample.fraction`	`max_samples`	Le taux d'échantillonnage des données
`replacement`		L'échantillonnage des données se fait-il avec ou sans remise?
`min.node.size`	`min_samples_leaf`	Nombre minimal d'observations nécessaire pour qu'un noeud puisse être partagé
`min.bucket`	`min_samples_split`	Nombre minimal d'observations dans les noeuds terminaux
`max.depth`	`max_depth`	Profondeur maximale des arbres
`splitrule`	`criterion`	La métrique utilisée pour le choix des *splits*
`oob.error`	`oob_score`	Calculer la performance de la forêt par l'erreur OOB (et choix de la métrique pour `scikit`)

Voici une présentation des principaux hyperparamètres et de leurs effets sur les performances de la forêt aléatoire:

- Le **nombre d'arbres** par défaut varie selon les implémentations (500 dans `ranger`, 100 dans `scikit-learn`). Il s'agit d'un hyperparamètre particulier car il n'est associé à aucun arbitrage en matière de performance: la performance de la forêt aléatoire croît avec le nombre d'arbres, puis se stabilise. Le nombre optimal d'arbres est celui à partir duquel la performance de la forêt ne croît plus (ce point est détaillé plus bas) où à partir duquel l'ajout d'arbres supplémentaires génère des gains marginaux. Il est important de noter que ce nombre optimal dépend des autres hyperparamètres. Par exemple, un taux d'échantillonnage faible et un nombre faible de variables candidates à chaque noeud aboutissent à des arbres peu corrélés, mais peu performants, ce qui requiert probablement un plus grand nombre d'arbres. Dans le cas d'une classification, l'utilisation de mesures comme le score de Brier ou la fonction de perte logarithmique est recommandée pour évaluer la convergence plutôt que la précision (métrique par défaut de `ranger` et `scikit-learn`).
- Le **nombre (ou la part) de variables candidates à chaque noeud** (souvent appelé `mtry`) est un hyperparamètre essentiel qui détermine le nombre de variables prédictives sélectionnées aléatoirement à chaque nœud lors de la construction des arbres. Ce paramètre exerce la plus forte influence sur les performances du modèle, et un compromis doit être trouvé entre puissance prédictive des arbres et corrélation

entre arbres. Une faible valeur de `mtry` conduit à des arbres moins performants mais plus diversifiés et donc moins corrélés entre eux. Inversement, une valeur plus élevée améliore la précision des arbres individuels mais accroît leur corrélation (les mêmes variables ayant tendance à être sélectionnées dans tous les arbres). La valeur optimale de `mtry` dépend du nombre de variables réellement pertinentes dans les données: elle est plus faible lorsque la plupart des variables sont pertinentes, et plus élevée lorsqu'il y a peu de variables pertinentes. Par ailleurs, une valeur élevée de `mtry` est préférable si les données comprennent un grand nombre de variables binaires issues du *one-hot-encoding* des variables catégorielles (LIEN AVEC LA PARTIE PREPROCESSING). Par défaut, cette valeur est fréquemment fixée à $\sqrt{p}$ pour les problèmes de classification et à $p/3$ pour les problèmes de régression, où p représente le nombre total de variables prédictives disponibles.

- Le **taux d'échantillonnage** et le **mode de tirage** contrôlent le plan d'échantillonnage des données d'entraînement. Les valeurs par défaut varient d'une implémentation à l'autre; dans le cas de `ranger`, le taux d'échantillonnage est de 63,2% sans remise, et de 100% avec remise. L'implémentation `scikit-learn` ne propose pas le tirage sans remise. Ces hyperparamètres ont des effets sur la performance similaires à ceux du nombre de variables candidates, mais d'une moindre ampleur. Un taux d'échantillonnage plus faible aboutit à des arbres plus diversifiés et donc moins corrélés (car ils sont entraînés sur des échantillons très différents), mais ces arbres peuvent être peu performants car ils sont entraînés sur des échantillons de petite taille. Inversement, un taux d'échantillonnage élevé aboutit à des arbres plus performants mais plus corrélés. Les effets de l'échantillonnage avec ou sans remise sur la performance de la forêt aléatoire sont moins clairs et ne font pas consensus. Les travaux les plus récents semblent toutefois suggérer qu'il est préférable d'échantillonner sans remise (P. Probst, M. N. Wright, and A.-L. Boulesteix [23]).
- Le **nombre minimal d'observations dans les noeuds terminaux** contrôle la taille des noeuds terminaux. La valeur par défaut est faible dans la plupart des implémentations (entre 1 et 5). Il n'y a pas vraiment de consensus sur l'effet de cet hyperparamètre sur les performances, bien qu'une valeur plus faible augmente le risque de sur-apprentissage. En revanche, il est certain que le temps d'entraînement décroît fortement avec cet hyperparamètre: une valeur faible implique des arbres très profonds, avec un grand nombre de noeuds. Il peut donc être utile de fixer ce nombre à une valeur plus élevée pour accélérer l'entraînement, en particulier si les données sont volumineuses et si on

utilise une méthode de validation croisée pour le choix des autres hyperparamètres. Cela se fait généralement sans perte significative de performance.

- Le **critère de choix de la règle de division des noeuds intermédiaires**: la plupart des implémentations des forêts aléatoires retiennent par défaut l'impureté de Gini pour la classification et la variance pour la régression, même si d'autres critères de choix ont été proposés dans la littérature (p-value dans les forêts d'inférence conditionnelle, arbres extrêmement randomisés, etc.). Chaque règle présente des avantages et des inconvénients, notamment en termes de biais de sélection des variables et de vitesse de calcul. A ce stade, aucun critère de choix ne paraît systématiquement supérieur aux autres en matière de performance. Modifier cet hyperparamètre relève d'un usage avancé des forêts aléatoires. Le lecteur intéressé pourra se référer à la discussion détaillée dans P. Probst, M. N. Wright, and A.-L. Boulesteix [23].

8.3. Comment entraîner une forêt aléatoire?

Les forêts aléatoires nécessitent généralement moins d'optimisation que d'autres modèles de *machine learning*, car leurs performances varient relativement peu en fonction des hyperparamètres. Les valeurs par défaut fournissent souvent des résultats satisfaisants, ce qui réduit le besoin d'optimisation intensive (P. Probst and A.-L. Boulesteix [25], P. Probst, A.-L. Boulesteix, and B. Bischl [26]). Cependant, un ajustement précis des hyperparamètres peut apporter des gains de performance, notamment sur des jeux de données complexes.

Comme indiqué dans la partie Section 5.3.3, la performance prédictive d'une forêt aléatoire varie en fonction de deux critères essentiels: elle croît avec le pouvoir prédictif des arbres, et décroît avec la corrélation des arbres entre eux. L'optimisation des hyperparamètres d'une forêt aléatoire vise donc à trouver un équilibre optimal où les arbres sont suffisamment puissants pour être prédictifs, tout en étant suffisamment diversifiés pour que leurs erreurs ne soient pas trop corrélées.

La littérature propose de multiples approches pour optimiser simultanément plusieurs hyperparamètres: la recherche par grille (*grid search*), la recherche aléatoire (*random search*) et l'optimisation basée sur modèle séquentiel (SMBO), et il peut être difficile de savoir quelle approche adopter. Ce guide propose donc une première approche délibérément simple, avant de présenter les approches plus avancées.

8.3.1. Approche simple

Voici une procédure simple pour entraîner une forêt aléatoire. Elle ne garantit pas l'obtention d'un modèle optimal, mais elle est lisible et permet d'obtenir rapidement un modèle raisonnablement performant.

- **Entraîner une forêt aléatoire avec les valeurs des hyperparamètres par défaut**. Ce premier modèle servira de point de comparaison pour la suite.
- **Ajuster le nombre d'arbres**: entraîner une forêt aléatoire avec les hyperparamètres par défaut en augmentant progressivement le nombre d'arbres, puis déterminer à partir de quel nombre d'arbres la performance se stabilise (en mesurant la performance avec l'erreur OOB avec pour métrique le score de Brier). Fixer le nombre d'arbres à cette valeur par la suite.
- **Ajuster le nombre de variables candidates et le taux d'échantillonnage**: optimiser ces deux hyperparamètres grâce à une méthode de *grid search* évaluée par une approche de validation-croisée, ou par une approche reposant sur l'erreur OOB.
- **Ajuster le nombre minimal d'observations dans les noeuds terminaux**: optimiser cet hyperparamètre grâce à une méthode de *grid search* évaluée par une approche de validation-croisée, ou par une approche reposant sur l'erreur OOB. Ce n'est pas l'hyperparamètre le plus important, mais s'il est possible de le fixer à une valeur plus élevée que la valeur par défaut sans perte de performance, cela permet d'accélérer le reste de la procédure.
- **Entraîner le modèle final**: entraîner une forêt aléatoire avec les hyperparamètres optimisés déduits des étapes précédentes.
- **Évaluer le modèle final**: mesurer la performance du modèle final soit avec l'approche *out-of-bag* (OOB), soit avec un ensemble de test. Il est souvent instructif de comparer les performances du modèle final et du modèle entraîné avec les valeurs des hyperparamètres par défaut (parfois pour se rendre compte que ce dernier était déjà suffisamment performant…).

8.3.2. Approches plus avancées

Lorsque l'espace des hyperparamètres est large ou que les performances initiales sont insuffisantes, adopter des méthodes avancées comme l'ptimisation basée sur un modèle séquentiel (SMBO). En `R`, il existe plusieurs implémentations d'appuyant sur cette méthode: `tuneRF` (limité à l'optimisation de `mtry`), tuneRanger (optimise simultanément `mtry`, node size, et sample size). La méthode SMBO est généralement la plus performante, mais demande un temps de calcul plus important.

Il est également possible de remplacer les critères classiques (le taux d'erreur pour une classification par exemple) par d'autres critères de performance, comme le score de Brier ou la fonction de perte logarithmique (P. Probst and A.-L. Boulesteix [25]).

Pour gérer la contrainte computationnelle, il est possible de commencer par utiliser des échantillons réduits pour les étapes exploratoires, puis d'augmenter la taille de l'échantillon pour les tests finaux.

8.4. Mesurer l'importance des variables

Il s'avère souvent utile de savoir quelles sont les variables qui jouent le plus grand rôle dans un modèle, à la fois pour en interpréter les résultats mais aussi pour conserver uniquement les variables pertinentes de façon à construire un modèle performant et parcimonieux.

Il existe plusieurs méthodes classiques d'évaluation de l'importance des variables, telles que l'indice de Gini (*Mean Decrease in Impurity* - MDI) et l'importance par permutation (*Mean Decrease Accuracy* - MDA). Bien qu'elles soient fréquemment utilisées, il est important de garder en tête que ces mesures d'importance n'ont pas de fondement statistique précis. Par ailleurs, ces méthodes peuvent produire des résultats biaisés dans certaines situations (C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn [13], C. Bénard, S. Da Veiga, and E. Scornet [11], C. Bénard, G. Biau, S. Da Veiga, and E. Scornet [12]). En particulier, elles peuvent surestimer l'importance de certaines variables lorsque les variables prédictives sont fortement corrélées, présentent des échelles de mesure différentes ou possèdent un nombre variable de catégories. Par exemple, les variables avec un grand nombre de catégories ou des échelles continues étendues peuvent être artificiellement privilégiées, même si leur contribution réelle à la prédiction est limitée.

En pratique, il est recommandé d'utiliser des méthodes d'importance des variables moins sensibles aux biais, comme les CIF ou la Sobol-MDA. Les valeurs de Shapley, issues de la théorie des jeux, sont également une alternative intéressante. Elles attribuent à chaque variable une contribution proportionnelle à son impact sur la prédiction. Cependant, leur calcul est souvent complexe et coûteux en ressources computationnelles, surtout en présence de nombreuses variables. Des méthodes comme SHAFF (SHApley eFfects via random Forests) ont été développées pour estimer efficacement ces valeurs, même en présence de dépendances entre variables.

On conseille l'utilisation de trois implémentations pour comparer l'importances des variables d'une forêt aléatoire:

- Pour la MDI: l'algorithme CIF proposé par C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn [13] et implémenté en R
- Pour la MDA: l'algorithme Sobol-MDA proposé par C. Bénard, S. Da Veiga, and E. Scornet [11] et implémenté en R
- Pour les valeurs de Shapley : l'alogrithme SHAFF proposé par C. Bénard, G. Biau, S. Da Veiga, and E. Scornet [12] et implémenté en R

Enfin, nous recommandons de combiner plusieurs méthodes pour une analyse plus robuste et de tenir compte des prétraitements des données afin de minimiser les biais potentiels.

9. Guide d'usage du *gradient boosting*

Ce guide propose des recommandations sur l'usage des algorithmes de *gradient boosting* disponibles dans la littérature, notamment C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz [27] et P. Probst, A.-L. Boulesteix, and B. Bischl [26]. Contrairement aux forêts aléatoires, la littérature méthodologique sur l'usages des algorithmes de *gradient boosting* est assez limitée et relativement peu conclusive. Ce guide comporte un certain nombre de choix méthodologiques forts, comme les implémentations recommandées ou la procédure d'optimisation des hyperparamètres, et d'autres choix pertinents sont évidemment possibles. C'est pourquoi **les recommandations de ce guide doivent être considérées comme un point de départ raisonnable, pas comme un ensemble de règles devant être respectées à tout prix**.

9.1. Quelle implémentation utiliser?

Il existe quatre implémentations principales du *gradient boosting*: `XGBoost`, `LightGBM`, `CatBoost` et `scikit-learn`. Elles sont toutes des variantes optimisées de l'algorithme de J. H. Friedman [18] et ne diffèrent que sur des points mineurs. De multiples publications les ont comparées, à la fois en matière de pouvoir prédictif et de rapidité d'entraînement (voir notamment C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz [27], H. Alshari, A. Y. Saleh, and A. Odabaş [28] et P. Florek and A. Zagdański [29]). Cette littérature a abouti à trois conclusions. Premièrement, les différentes implémentations présentent des performances très proches (le classement exact variant d'une publication à l'autre). Deuxièmement, bien optimiser les hyperparamètres est nettement plus important que le choix de l'implémentation. Troisièmement, le temps d'entraînement varie beaucoup d'une implémentation à l'autre, et `LightGBM` est sensiblement plus rapide que les autres. Dans la mesure où l'optimisation des hyperparamètres est une étape à la fois essentielle et intense en calcul, l'efficacité computationnelle apparaît comme un critère majeur de choix de l'implémentation. C'est pourquoi **le présent document décrit et recommande l'usage de `LightGBM`**. Ceci étant, les trois autres implémentations peuvent également être utilisées, notamment si les données sont de taille limitée.

Par ailleurs, chacune de ces implémentations propose une interface de haut niveau compatible avec `scikit-learn`. **Il est vivement recommandé d'utiliser cette interface car elle minimise les risques d'erreur, facilite la construction de modèles reproductibles et permet d'utiliser l'ensemble des outils proposés par `scikit-learn`.**

9.2. Les hyperparamètres clés du *gradient boosting*

Table 3 : Les principaux hyperparamètres de `LightGBM`

Hyperparamètre	**Description**	**Valeur par défaut**
`objective`	Fonction de perte utilisée	Variable
`n_estimators` ou `num_trees`	Nombre d'arbres	100
`learning_rate` ou `eta`	Taux d'apprentissage	0.1
`max_depth`	Profondeur maximale des arbres	Pas de limite
`num_leaves`	Nombre de feuilles terminales des arbres	31
`min_child_samples`	Nombre minimal d'observations qu'une feuille terminale doit contenir	20
`min_child_weight`	Poids minimal qu'une feuille terminale doit contenir	0.001
`lambda` ou `lambda_l2`	Pénalisation quadratique sur la valeur des feuilles terminales	0
`reg_alpha` ou `lambda_l1`	Pénalisation absolue (L1) sur la valeur des feuilles terminales	0
`min_split_gain`	Gain minimal nécessaire pour diviser un noeud	0
`bagging_fraction`	Taux d'échantillonnage des données d'entraînement (utilisé uniquement si `bagging_freq` > 0)	1
`bagging_freq`	Fréquence de rééchantillonnage des données d'entraînement (utilisé uniquement si `bagging_fraction` < 1)	1
`feature_fraction`	Taux d'échantillonnage des colonnes par arbre	1
`feature_fraction_bynode`	Taux d'échantillonnage des colonnes par noeud	1
`max_bin`	Nombre de *bins* utilisés pour discrétiser les variables continues	255
`max_cat_to_onehot`	Nombre de modalités en-deça duquel `LightGBM` utilise le *one-hot-encoding*	4
`max_cat_threshold`	Nombre maximal de *splits* considérés dans le traitement des variables catégorielles	32
`sample_weight`	Pondération des observations dans les données d'entraînement	1
`scale_pos_weight`	Poids des observations de la classe positive (classification binaire uniquement)	Aucun
`class_weight`	Poids des observations de chaque classe (classification multiclasse uniquement)	Aucun

⚠ Attention aux alias!

Il arrive fréquemment que les hyperparamètres des algorithmes de *gradient boosting* portent plusieurs noms. Par exemple dans `LightGBM`, le nombre d'arbres porte les noms suivants: `num_iterations`, `num_iteration`, `n_iter`, `num_tree`, `num_trees`, `num_round`, `num_rounds`, `nrounds`, `num_boost_round`, `n_estimators` et `max_iter` (ouf!). C'est une source récurrente de confusion, mais il est facile de s'y retrouver en consultant la page de la documentation sur les hyperparamètres, qui liste les *alias*:

- hyperparamètres de `LightGBM` ;
- hyperparamètres de `XGBoost` ;
- hyperparamètres de `CatBoost` ;
- hyperparamètres de `scikit-learn` .

Voici une présentation des principaux hyperparamètres et de leurs effets sur les performances sur le modèle de *gradient boosting*:

- La **mécanique du *gradient boosting*** est contrôlée par seulement trois hyperparamètres (tous les autres hyperparamètres portant sur la construction des arbres pris isolément):
  - L'hyperparamètre `objective` définit à la fois la **nature du problème** modélisé (régression, classification…) et la **fonction de perte** utilisée lors de l'entraînement du modèle. La valeur par défaut varie selon le modèle utilisé: `regression_l2` (minimisation de l'erreur quadratique moyenne) pour la régression, `binary_log_loss` (maximisation de la log-vraisemblance d'un modèle logistique) pour la classification binaire et `softmax` (maximisation de la log-vraisemblance d'un logit multinomial) pour la classification multiclasse. Il existe de nombreuses autres possibilités, comme `regression_l1` pour la régression (minimisation de l'erreur absolue moyenne)
  - le **nombre d'arbres** contrôle la complexité générale de l'algorithme. Le point essentiel est que, contrairement aux forêts aléatoires, la performance du *gradient boosting* sur les données d'entraînement croît continûment avec le nombre d'arbres sans jamais se stabiliser. Le choix du nombre d'arbres est essentiel, et doit viser un équilibre entre amélioration du pouvoir prédictif du modèle (si les arbres supplémentaires permettent au modèle de corriger les erreurs résiduelles), et lutte contre le surajustement (si les arbres supplémentaires captent uniquement les bruits statistiques et les fluctuations spécifiques des données d'entraînement). Par ailleurs, le choix du nombre d'arbres est très lié à celui

du taux d'apprentissage, et il est nécessaire de les optimiser conjointement (P. Probst, A.-L. Boulesteix, and B. Bischl [26]).

  - le **taux d'apprentissage** (*learning rate*) contrôle l'influence de chaque arbre sur le modèle global; il s'agit de η dans l'équation REFERENCE PARTIE OVERFITTING. Cet hyperparamètre a un effet important sur la performance du modèle global (P. Probst, A.-L. Boulesteix, and B. Bischl [26]). Un taux d'apprentissage faible réduit la contribution de chaque arbre, rendant l'apprentissage plus progressif; cela évite qu'un arbre donné ait une influence trop importante sur le modèle global et contribue donc à réduire le surajustement, mais nécessite un plus grand nombre d'arbres pour converger vers une solution optimale. Inversement, un taux d'apprentissage élevé accélère l'entraînement mais peut rendre le modèle instable (car trop sensible à un arbre donné), entraîner un surajustement et/ou aboutir à un modèle sous-optimal. La règle générale est de privilégier un taux d'apprentissage faible (entre 0.01 ou 0.3). Le choix du taux d'apprentissage est très lié à celui du nombre d'arbres: plus le taux d'apprentissage sera faible, plus le nombre d'arbres nécessaires pour converger vers une solution optimale sera élevé. Ces deux hyperparamètres doivent donc être optimisés conjointement.

- La **complexité des arbres**: la profondeur maximale des arbres et le nombre de feuilles terminales contrôlent la complexité des *weak learners*: une profondeur élevée et un grand nombre de feuilles aboutissent à des arbres complexes au pouvoir prédictif plus élevé, mais induisent un risque de surajustement. Par ailleurs, de tels arbres sont plus longs à entraîner que des arbres peu profonds avec un nombre limité de feuilles. Le nombre optimal d'arbres est très corrélé au nombre de feuilles terminales: il est logiquement plus faible quand les arbres sont complexes. Il est à noter que le nombre de feuilles terminales a un effet linéaire sur la complexité des arbres, tandis que la profondeur maximale a un effet exponentiel: un arbre pleinement développé de profondeur k comprend 2^k feuilles terminales et $2^k - 1$ *splits*. Augmenter la profondeur d'une unité a donc pour effet de doubler le temps d'entraînement de chaque arbre. Ces deux hyperparamètres peuvent interagir entre eux de manière complexe (voir encadré).
- La **lutte contre le surajustement**: ces hyperparamètres de régularisation jouent un rôle important dans le contrôle de la complexité des *weak learners* et contribuent à éviter le surajustement:
  - Les pénalisations tendent à réduire le poids w_j des feuilles terminales: la pénalisation quadratique réduit la valeur absolue des poids sans les annuler (il s'agit de λ dans l' équation donnant le poids optimal d'une feuille terminale), tandis que

la pénalisation absolue élevée pousse certains poids à être nuls. La pénalisation quadratique est la plus utilisée, notamment parce qu'elle permet d'amoindrir l'influence des points aberrants.

- Le nombre minimal d'observations par feuille terminale contrôle la taille des feuilles terminales. Une valeur faible autorise le modèle à isoler de petits groupes d'observations mais induit un risque de surajustement; inversement une valeur plus élevée limite le surajustement mais peut réduire le pouvoir prédictif.
- Le gain minimal définit la réduction minimale de la perte requise pour qu'un nœud soit divisé (il s'agit du paramètre γ dans l' équation donnant le gain potentiel d'un *split*); une valeur plus élevée contribue à réduire la complexité des arbres et à limiter le surajustement en empêchant l'algorithme de créer des *splits* dont l'apport est très faible et potentiellement dû à des variations non significatives des données d'entraînement.

- Les **hyperparamètres d'échantillonnage**:
  - le taux d'échantillonnage des données d'entraînement et le taux d'échantillonnage des colonnes par noeud jouent exactement le même rôle que `sample.fraction` ou `max_samples`, et `mtry` dans une forêt aléatoire: échantillonner les données d'entraînement accélère l'entraînement, et échantillonner les colonnes au niveau de chaque noeud aboutit à des arbres plus variés. Il est à noter que l'échantillonnage des données se fait systématiquement sans remise dans les algorithmes de *gradient boosting*. Comme pour la forêt aléatoire, la valeur optimale du taux d'échantillonnage des colonnes par noeud dépend du nombre de variables réellement pertinentes dans les données, et une valeur plus élevée est préférable si les données comprennent un grand nombre de variables binaires issues du *one-hot-encoding* des variables catégorielles.
  - L'échantillonnage des colonnes par arbre sert essentiellement à accélérer l'entraînement. Si les colonnes sont échantillonnées à la fois par arbre et par noeud, alors le taux d'échantillonnage final est le produit des deux taux.
- Les **réglages relatifs au retraitement des colonnes**:
  - le nombre de *bins* utilisés pour discrétiser les variables continues (voir partie PREPROCESSING pour le détail): un faible de *bins* contribue à accélérer l'entraînement (car le nombre de *splits* potentiels est faible), mais peut dégrader le pouvoir prédictif si de faibles variations de la variable continue ont un impact notable sur la variable-cible. Inversement, une valeur élevée permet de conserver davantage d'information sur la distribution de la variable continue, mais peut ralentir l'entraînement.

  - le nombre de modalités en-deça duquel les variables catégorielles font l'objet d'un *one-hot-encoding* et le nombre maximal de *splits* considérés dans le traitement des variables catégorielles définissent la méthode utilisée pour traiter les variables catégorielles (voir partie PREPROCESSING pour le détail).
- Les **pondérations**:
  - la pondération des observations sert à pondérer les données d'entraînement (voir la section Section 5.8).
  - le poids des observations de la classe positive sert à rééquilibrer les données d'entraînement lorsque la classe positive est sous-représentée. Cet hyper-paramètre ne sert que pour la classification binaire. Par défaut les deux classes ont le même poids.
  - le poids des observations de chaque classe sert à rééquilibrer les données d'entraînement lorsque la part des différentes classes est hétérogène. Cet hyper-paramètre ne sert que pour la classification binaire multiclasse. Par défaut toutes les classes ont le même poids.

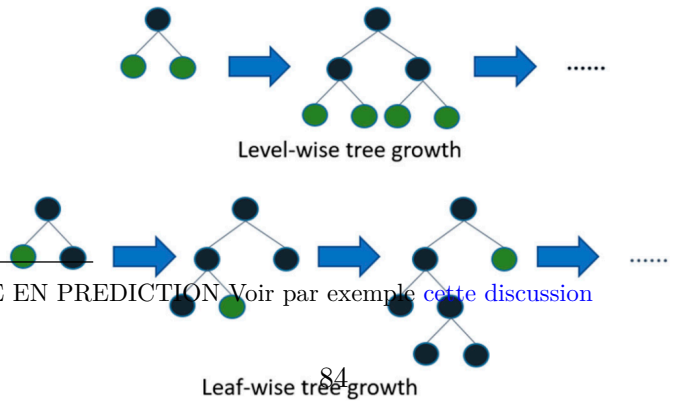
⚠ *Depth-wise versus leaf-wise*: les deux approches de la construction des arbres

Il existe deux méthodes de construction des arbres, illustrée par la figure ci-dessous:

- dans l' **approche par niveau** (dite *depth-wise*) proposée à l'origine par `XGBoost`, l'arbre est construit niveau par niveau, en divisant tous les nœuds du même niveau avant de passer au niveau suivant. L'approche *depth-wise* n'est pas optimale pour minimiser la fonction de perte, car elle ne recherche pas systématiquement le *split* le plus performant, mais elle permet d'obtenir des arbres équilibrés et de profondeur limitée. L'hyperparamètre-clé de cette approche est la profondeur maximale des arbres (`max_depth`).
- dans l' **approche par feuille** (dite *leaf-wise*) proposée à l'origine par `LightGBM`, l'arbre est construit feuille par feuille, et c'est le *split* avec le gain le plus élevé qui est retenu à chaque étape, et ce quelle que soit sa position dans l'arbre. L'approche *leaf-wise* est très efficace pour minimiser la fonction de perte, car elle privilégie les *splits* les plus porteurs de gain. L'hyperparamètre-clé de cette approche est le nombre maximal de feuilles terminales (`num_leaves`). Il se trouve que l'approche *leaf-wise* a été ajoutée par la suite à `XGBoost`, *via* l'hyperparamètre `grow_policy` qui peut prendre les valeurs `depthwise` (valeur par défaut) et `lossguide` (approche *leaf-wise*).

Si elle est en général plus performante que l'approche par niveau, l'approche par feuille peut aboutir à un modèle surajusté et difficile à utiliser car composé d'arbres complexes, déséquilibrés et très profonds[13]. Par exemple, si on fixe `num_leaves` à 256, on peut aisément obtenir un arbre avec une branche de profondeur 30. Il est donc important de spécifier à la fois `num_leaves` et `max_depth` lorsqu'on utilise l'approche par feuille, de façon à obtenir des arbres peu déséquilibrés. Une approche simple consiste à fixer conjointement les deux hyperparamètres: si on fixe `num_leaves` à 2^k, on peut fixer `max_depth` à $k + 5$. Cela signifie que dans un arbre à 256 feuilles aucune branche ne pourra pas avoir une profondeur supérieure à 13.

Figure 12



[13]REFERENCE USAGE EN PREDICTION Voir par exemple cette discussion

9.3. Comment entraîner un algorithme de *gradient boosting*?

Proposer une procédure pour l'optimisation des hyperparamètres s'avère plus délicat pour les algorithmes de *gradient boosting* que pour les forêts aléatoires, car ces algorithmes comprennent un nombre beaucoup plus élevé d'hyperparamètres, et la littérature méthodologique sur leur usage pratique reste assez limitée et peu conclusive (en-dehors des nombreux tutoriels introductifs disponibles sur internet). Trois constats sont néanmoins bien établis. Premièrement, **optimiser les hyperparamètres est essentiel pour la performance du modèle final**. Deuxièmement, **contrairement aux forêts aléatoires, les valeurs par défaut des hyperparamètres dans les différentes implémentations ne constituent le plus souvent pas un point de départ raisonnable** (C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz [27] et P. Probst, A.-L. Boulesteix, and B. Bischl [26]), en particulier pour les hyperparamètres de régularisation dont la valeur par défaut est souvent nulle. Troisièmement, **cette optimisation peut s'avérer complexe et longue**, il faut donc la mener de façon rigoureuse et organisée pour ne pas perdre de temps.

> Parfois, une forêt aléatoire suffit...
>
> Avant de se lancer dans le *gradient boosting*, il est utile d'entraîner une forêt aléatoire selon la procédure décrite dans la section Section 8. Ce modèle servira de point de comparaison pour la suite, et permettra notamment de voir si le *gradient boosting* offre des gains de performances qui justifient le temps passé à l'optimisation des hyperparamètres.

9.3.1. Préparer l'optimisation des hyperparamètres

- **Choisir les hyperparamètres à optimiser**. Le nombre élevé d'hyperparamètres fait qu'il est en pratique impossible (et inutile) des les optimiser tous, il est donc important de restreindre l'optimisation aux hyperparamètres qui ont le plus d'influence sur la performance du modèle, et d'utiliser simplement des valeurs par défaut raisonnables pour les autres hyperparamètres (voir point suivant). Sur ce point, la littérature méthodologique suggère de concentrer l'effort d'optimisation sur le nombre d'arbres, le taux d'apprentissage, la complexité des arbres (nombre de feuilles et/ou profondeur maximale) et les paramètres de régularisation. Inversement, les hyperparamètres d'échantillonnage n'ont pas réellement besoin d'être optimisés une fois qu'on a choisi une valeur par défaut raisonnable (voir P. Probst, A.-L. Boulesteix, and B. Bischl [26] et C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz [27]). La liste des hyperparamètres à optimiser peut évidemment varier en fonction du problème modélisé; en tout état de

cause, prendre le temps d'établir cette liste est essentiel pour ne pas se perdre dans les étapes d'optimisation.

- **Définir des valeurs par défaut raisonnables**. Définir les valeurs par défaut des hyperparamètres est une étape importante car elle permet de gagner du temps lors de l'optimisation des hyperparamètres. **Ce choix prend du temps** et doit reposer sur une bonne compréhension du fonctionnement de l'algorithme et sur une connaissance approfondie des données utilisées. Voici quelques suggestions de valeurs de départ issues de la littérature (voir notamment C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz [27] et P. Probst, A.-L. Boulesteix, and B. Bischl [26]); il est tout à fait possible de s'en écarter lorsqu'on pense que le problème modélisé le justifie:
  - `num_leaves`: entre 31 et 255;
  - `max_depth`: entre 8 et 12;
  - `min_child_samples`: entre 5 et 50;
  - `min_split_gain`: valeur strictement positive, commencer entre 0.1 et 1;
  - `lambda`: valeur strictement positive; commencer avec une valeur entre 0.5 et 2; choisir une valeur plus élevée s'il y a des valeurs aberrantes sur y ou de clairs signes de surajustement;
  - `bagging_fraction` : valeur strictement inférieure à 1, commencer entre 0.6 et 0.8;
  - `bagging_freq` : valeur entière strictement positive, commencer avec 1 ;
  - `feature_fraction_bynode` : valeur strictement inférieure à 1, commencer entre 0.5 et 0.7; choisir une valeur plus élevée si les données comprennent un grand nombre de variables binaires issues d'un *one-hot-encoding*;
  - `max_bin` : garder la valeur par défaut; choisir éventuellement une valeur plus élevée si la la valeur par défaut ne suffit pas à refléter la distribution des variables continues;
  - `max_cat_to_onehot` : garder la valeur par défaut;
  - `max_cat_threshold` : garder la valeur par défaut.
- **Définir la méthode d'évaluation des hyperparamètres**: indépendamment de la méthode d'optimisation des hyperparamètres (*grid search*, *random search*...), deux approches sont envisageables pour évaluer les valeurs possibles des hyperparamètres: soit une **validation croisée** (par exemple avec la fonction `GridSearchCV` de `scikit-learn`), soit une **validation simple** reposant sur un unique ensemble de validation. La validation simple peut être préférable si les données utilisées sont volumineuses (au-delà de plusieurs centaines de milliers d'observations) car elle offre un gain de temps

appréciable, sans nécessairement dégrader les résultats. La validation croisée est censée être plus robuste que l'utilisation d'un ensemble de validation, mais elle est coûteuse sur le plan computationnel (car il faut entraîner plusieurs fois un modèle pour chaque jeu d'hyperparamètres).

9.3.2. Optimiser les hyperparamètres

Cette section propose trois approches pour optimiser les hyperparamètres d'un algorithme de *gradient boosting*. Les deux premières reposent sur une validation croisée et une validation simple, associée à un *grid search*. La troisième est plus avancée et mobilise Optuna, un *framework* d'optimisation d'hyperparamètres. Aucune de ces approches ne garantit pas l'obtention d'un modèle optimal, mais elle sont relativement lisibles et permettent d'obtenir rapidement un modèle raisonnablement performant.

9.3.2.1. Approche 1: procédure itérative par validation croisée

La première approche repose sur une validation croisée. Elle peut être coûteuse sur le plan computationnel en raison du nombre de modèles à entraîner et est donc adaptée à des données peu volumineuses. Cette approche comprend quatre étapes:

- **Optimiser conjointement le nombre d'arbres et le taux d'apprentissage**. Par exemple, on évalue les performances du modèle en testant les valeurs `[100, 200, 500, 1000]` pour le nombre d'arbres, et `[0.05, 0.1, 0.15, 0.2]` pour le taux d'apprentissage, avec des valeurs raisonnables pour les autres hyperparamètres. On retient finalement le taux d'apprentissage et le nombre d'arbres du modèle le plus performant. Si l'un de ces hyperparamètres est la valeur minimale ou maximale testée (par exemple 1000 arbres, ou 0.05 pour le taux d'apprentissage), alors il est préférable de recommencer l'exercice en ajustant la liste des valeurs possibles.
- **Optimiser conjointement les hyperparamètres contrôlant la structure des arbres**. On utilise le taux d'apprentissage et le nombre d'arbres issus de l'étape précédente, puis on évalue les performances de différents couples de valeurs (`max_depth`, `min_child_samples`), et on retient enfin les valeurs des hyperparamètres du modèle le plus performant.
- **Optimiser conjointement les hyperparamètres de régularisation**. On utilise les valeurs des hyperparamètres issues des étapes précédentes, puis on évalue les performances de différents couples de valeurs (`lambda`, `min_split_gain`), et on retient les valeurs des hyperparamètres du modèle le plus performant.
- **Entraîner le modèle final** avec les hyperparamètres finaux.

Utiliser l'option `return_train_score = True` de `GridsearchCV` pour mesurer le surajustement, et durcir les hyperparamètres de régularisation quand il est important.

Cette procédure est présentée en détail dans la partie 1 du *notebook* XXXXXXXXXXXXXXXXXXXXXXXXXXXX.

9.3.2.2. Approche 2: approche itérative par validation simple

La seconde approche est moins exigeante sur le plan computationnel, et est donc plus adaptée à des données volumineuses. Elle est très similaire à la première, et s'en distingue sur deux points. D'une part, elle repose sur une validation simple, ce qui réduit le nombre de modèles à entraîner. D'autre part, le nombre optimal d'arbres n'est pas fixé à l'avance, mais est défini à chaque étape de l'optimisation des hyperparamètres par un mécanisme d'*early stopping*. Celui-ci fonctionne de la façon suivante: on définit un nombre d'arbres très élevé (par exemple 50 000) et pour chaque jeu d'hyperparamètres, on laisse l'entraînement se prolonger jusqu'à ce que les performances mesurées sur l'ensemble de validation cessent de s'améliorer. Le nombre d'arbres retenu est celui pour lequel la performance sur l'ensemble de validation est maximale.

Cette procédure comprend quatre étapes:

- **Optimiser le taux d'apprentissage**. On recherche un taux d'apprentissage optimal parmi une liste de valeurs (exemple: `[0.05, 0.1, 0.15, 0.2]`), avec des valeurs raisonnables pour les autres hyperparamètres, en entraînant le modèle sur les données d'entraînement avec un mécanisme d'*early stopping* utilisant l'ensemble de validation. On retient finalement le taux d'apprentissage du modèle le plus performant sur l'ensemble de validation. Si le taux d'apprentissage optimal est la valeur minimale ou maximale testée, il est préférable de recommencer l'exercice en ajustant la liste des valeurs possibles.
- **Optimiser conjointement les hyperparamètres contrôlant la structure des arbres**. On utilise le taux d'apprentissage issu de l'étape précédente, puis on évalue les performances de différents couples de valeurs (`max_depth`, `min_child_samples`), et on retient enfin les valeurs des hyperparamètres du modèle le plus performant sur l'ensemble de validation.
- **Optimiser conjointement les hyperparamètres de régularisation**. On utilise les valeurs des hyperparamètres issues des étapes précédentes, puis on évalue les performances de différents couples de valeurs (`lambda`, `min_split_gain`), et on retient les valeurs des hyperparamètres du modèle le plus performant sur l'ensemble de validation.
- **Entraîner le modèle final** avec les hyperparamètres finaux.

Cette approche est présentée en détail dans la partie 2 du notebook XXXXXXXXXXXXXXXXXXXXXXXXXXXX.

9.3.2.3. Approche 3: utiliser Optuna

La troisième approche est plus avancée et mobilise Optuna , une librairie d'optimisation automatique des hyperparamètres des modèles de *machine learning* qui permet d'optimiser conjointement une liste d'hyperparamètres. L'usage d'Optuna s'avère très simple: l'utilisateur définit le modèle, la mesure de performance selon laquelle qu'il souhaite optimiser les hyperparamètres ainsi que l'espace des valeurs que ceux-ci peuvent prendre (exemple: `learning_rate` entre 0.01 et 0.3, `num_leaves` entre 30 et 1000…). Optuna tire aléatoirement un jeu d'hyperparamètres, entraîne le modèle avec ces hyperparamètres, mesure la performance obtenue et utilise cette information pour guider les essais suivants. Optuna est conçu pour converger rapidement vers les sous-espaces les plus prometteurs, et pour interrompre automatiquement les essais insuffisamment performants. Optuna garde la trace des différents essais réalisés et propose des visualisations commodes.

Cette procédure est présentée en détail dans la partie 3 du notebook XXXXXXXXXXXXXXXXXXXXXXXXXXXX.

9.4. Utiliser un modèle de *gradient boosting* en prédiction

La dernière étape de l'usage d'un modèle de *gradient boosting* consiste à l'utiliser en prédiction. Cette étape peut sembler simple au premier abord: ne suffit-il pas d'utiliser les fonctions de prédiction de l'implémentation retenue? C'est effectivement le cas, mais l'usage de ces modèles en prédiction peut buter sur un problème de performance car si les implémentations du *gradient boosting* sont optimisées pour être très performantes au cours de l'entraînement des modèles, elles ne le sont pas nécessairement lorsqu'on les utilise en prédiction. Cela ne pose généralement pas de problème lorsque la prédiction porte sur des données de taille restreinte, et que les modèles restent légers (arbres peu nombreux et peu complexes). En revanche, la prédiction peut devenir très lente voire impraticable lorsqu'on entend utiliser des modèles complexes (plusieurs milliers d'arbres) avec des données volumineuses (plusieurs millions d'observations).

Heureusement, un certain nombre de librairies *open-source* proposent d'optimiser des modèles déjà entraînés pour accélérer leur usage en prédiction. Par exemple, la librairie oneDAL , disponible en Python grâce au *package* `daal4py`, permet de compiler un modèle XGBoost, LightGBM et CatBoost déjà entraîné puis de l'utiliser directement avec Python, ce qui réduit considérablement le temps de prédiction sur des processeurs Intel (entre cinq et vingt fois plus rapide selon les cas). On peut également citer les librairies suivantes dont l'utilisation est plus complexe: `tl2cgen` , Forest Inference Library développée par Nvidia, et ONNX .

References

[1] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?," *Advances in neural information processing systems*, vol. 35, pp. 507–520, 2022.

[2] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Information Fusion*, vol. 81, pp. 84–90, 2022.

[3] D. McElfresh *et al.*, "When do neural nets outperform boosted trees on tabular data?," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[4] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Cart," *Classification and regression trees*, 1984.

[5] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, pp. 123–140, 1996.

[6] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[7] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[8] J. H. Friedman, "Multivariate adaptive regression splines," *The annals of statistics*, vol. 19, no. 1, pp. 1–67, 1991.

[9] G. Louppe, "Understanding random forests: From theory to practice," *arXiv preprint arXiv:1407.7502*, 2014.

[10] G. Biau, "Analysis of a random forests model," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 1063–1095, 2012.

[11] C. Bénard, S. Da Veiga, and E. Scornet, "Mean decrease accuracy for random forests: inconsistency, and a practical solution via the Sobol-MDA," *Biometrika*, vol. 109, no. 4, pp. 881–900, 2022, doi: 10.1093/biomet/asac017 .

[12] C. Bénard, G. Biau, S. Da Veiga, and E. Scornet, "SHAFF: Fast and consistent SHApley eFfect estimates via random Forests," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2022, pp. 5563–5582.

[13] C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn, "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC bioinformatics*, vol. 8, pp. 1–21, 2007.

[14] R. E. Schapire, "The strength of weak learnability," *Machine learning*, vol. 5, pp. 197–227, 1990.

[15] L. Breiman, "Rejoinder: arcing classifiers," *The Annals of Statistics*, vol. 26, no. 3, pp. 841–849, 1998.

[16] A. J. Grove and D. Schuurmans, "Boosting in the limit: Maximizing the margin of learned ensembles," in *AAAI/IAAI*, 1998, pp. 692–699.

[17] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[18] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[19] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[20] G. Ke *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[21] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: unbiased boosting with categorical features," *Advances in neural information processing systems*, vol. 31, 2018.

[22] W. D. Fisher, "On grouping for maximum homogeneity," *Journal of the American statistical Association*, vol. 53, no. 284, pp. 789–798, 1958.

[23] P. Probst, M. N. Wright, and A.-L. Boulesteix, "Hyperparameters and tuning strategies for random forest," *Wiley Interdisciplinary Reviews: data mining and knowledge discovery*, vol. 9, no. 3, p. e1301, 2019.

[24] G. Biau and E. Scornet, "A random forest guided tour," *Test*, vol. 25, pp. 197–227, 2016.

[25] P. Probst and A.-L. Boulesteix, "To tune or not to tune the number of trees in random forest," *Journal of Machine Learning Research*, vol. 18, no. 181, pp. 1–18, 2018.

[26] P. Probst, A.-L. Boulesteix, and B. Bischl, "Tunability: Importance of hyperparameters of machine learning algorithms," *Journal of Machine Learning Research*, vol. 20, no. 53, pp. 1–32, 2019.

[27] C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz, "A comparative analysis of gradient boosting algorithms," *Artificial Intelligence Review*, vol. 54, pp. 1937–1967, 2021.

[28] H. Alshari, A. Y. Saleh, and A. Odabaş, "Comparison of gradient boosting decision tree algorithms for CPU performance," *Journal of Institue Of Science and Technology*, vol. 37, no. 1, pp. 157–168, 2021.

[29] P. Florek and A. Zagdański, "Benchmarking state-of-the-art gradient boosting algorithms for classification," *arXiv preprint arXiv:2305.17094*, 2023.