Université de Dschang - Faculté des Sciences

Projet de Fin d'Année

E - learning

Élèves :
Prénom Nom
Prénom Nom
Prénom Nom

Encadrant :
Prénom Nom

Année Universitaire 2024-2025

Remerciements

Résumé

Liste des Abréviations

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Le machine learning connaît une croissance exponentielle depuis les années 2010, portée par l'augmentation de la puissance de calcul, la disponibilité massive de données et les avancées algorithmiques. Dans ce contexte, les algorithmes de classification et de régression constituent les piliers fondamentaux de l'apprentissage supervisé. Ils sont aujourd'hui déployés dans des domaines critiques tels que la médecine prédictive, la détection de fraude, la reconnaissance d'images ou encore l'analyse de sentiment.

Parmi ces applications, la détection de fraude bancaire occupe une place particulière en raison de ses enjeux économiques majeurs et de ses contraintes techniques spécifiques. Le problème se caractérise par un déséquilibre extrême des classes, une nécessité de prise de décision en temps réel, des coûts métier fortement asymétriques entre les erreurs, et des exigences réglementaires strictes en matière d'explicabilité des décisions. Dans ce cadre, la question centrale n'est pas seulement de maximiser la performance prédictive, mais de trouver un compromis réaliste entre efficacité, interprétabilité et viabilité opérationnelle.

Ce mémoire s'inscrit dans cette problématique et vise à proposer une analyse comparative rigoureuse de trois algorithmes majeurs du machine learning supervisé appliqués à la détection de fraude par carte de crédit : les Support Vector Machines (SVM), les Random Forests et XGBoost. Bien qu'ils poursuivent le même objectif de prédiction, ces modèles reposent sur des fondements théoriques radicalement différents et présentent des comportements distincts face aux contraintes du monde bancaire réel.

L'objectif de ce travail est triple. Premièrement, il s'agit de présenter les bases théoriques de chaque algorithme de manière précise, en mettant l'accent sur les propriétés réellement pertinentes pour la détection de fraude. Deuxièmement, ce mémoire vise à fournir un cadre d'aide à la décision permettant d'orienter le choix de l'algorithme en fonction des caractéristiques du problème, en intégrant à la fois des métriques académiques classiques et des indicateurs métier concrets. Troisièmement, ce travail cherche à challenger les approches existantes en proposant des configurations algorithmiques alternatives capables d'améliorer les performances observées dans la littérature récente.

Le document est structuré en trois chapitres complémentaires. Le premier chapitre est consacré aux fondements théoriques des trois algorithmes étudiés. Il présente le principe de maximisation de la marge des SVM et le rôle du noyau dans la gestion de la non-linéarité, l'approche ensembliste des Random Forests basée sur le bagging et la réduction de la variance, ainsi que le principe du gradient boosting séquentiel implémenté par XGBoost, accompagné de ses mécanismes avancés de régularisation. Une attention particulière est portée aux stratégies de gestion du déséquilibre des classes, élément central de la détection de fraude.

Le deuxième chapitre propose une analyse comparative des trois approches selon des critères directement liés aux contraintes opérationnelles du domaine bancaire. Sont notamment étudiés la complexité algorithmique, le temps d'entraînement, la latence de prédiction, la gestion native ou artificielle du déséquilibre, l'interprétabilité des modèles, leur robustesse face à l'évolution des comportements frauduleux, ainsi que leur consommation mémoire en contexte de déploiement. Cette analyse vise à dépasser une comparaison purement académique pour adopter une perspective orientée production.

Le troisième chapitre constitue le cœur expérimental de ce mémoire. Il présente une application pratique sur un jeu de données réel de transactions bancaires anonymisées.

Plusieurs configurations algorithmiques sont implémentées et comparées, incluant des variantes de SVM avec pondération des classes, de Random Forest avec différentes techniques de rééchantillonnage, ainsi que de XGBoost exploitant ses mécanismes natifs de gestion du déséquilibre. L'évaluation ne se limite pas aux métriques classiques de classification, mais intègre un modèle de coût réaliste afin de mesurer l'impact financier réel des décisions de chaque algorithme.

Un accent particulier est mis sur l'optimisation du seuil de décision. Contrairement à l'approche standard utilisant un seuil fixe, ce travail adopte une stratégie orientée métier visant à maximiser le bénéfice net, en tenant compte du fait qu'en détection de fraude, les faux négatifs sont généralement bien plus coûteux que les faux positifs. L'impact du feature engineering sur la performance des modèles est également étudié afin d'évaluer la sensibilité de chaque algorithme à la qualité des représentations d'entrée.

La conclusion synthétise les résultats théoriques et empiriques obtenus afin de formuler des recommandations claires et pragmatiques. Elle répond notamment aux questions suivantes : dans quels contextes privilégier une Random Forest ? Quand l'investissement dans le réglage fin de XGBoost est-il justifié ? Les SVM conservent-ils une pertinence dans les systèmes modernes de détection de fraude bancaire ? À travers cette analyse, ce mémoire ambitionne de contribuer à une meilleure compréhension des compromis réels entre performance, coût et explicabilité dans les systèmes de détection de fraude en production.

Chapitre 1

Machine à vecteurs supports (MVS)

1.1 Introduction

Les *machines à vecteurs de support* (SVM, pour *Support Vector Machines*) constituent une famille d'algorithmes d'apprentissage supervisé particulièrement efficaces pour les tâches de **classification** et de **régression**. Introduites par Vladimir Vapnik et ses collègues dans les années 1990 (Boser et al., 1992), les SVM reposent sur un principe fondamental : la recherche de l'**hyperplan optimal** séparant au mieux les différentes classes de données, en maximisant la **marge** entre celles-ci.

Cette approche géométrique, combinée à l'utilisation des **fonctions noyau** (*kernel functions*) permettant de traiter des problèmes non linéaires, fait des SVM des méthodes robustes et performantes, même dans des espaces de grande dimension. Grâce à leur forte capacité de généralisation et à leur résistance au surapprentissage, les SVM sont aujourd'hui largement utilisées dans des domaines variés tels que la reconnaissance d'images, la bioinformatique, l'analyse de texte ou encore la détection de fraudes.

Leur fondement mathématique solide, reposant sur la théorie de l'**l'optimisation convexe** (Bertsekas, 2009) et la théorie de **Vapnik–Chervonenkis** (Devroye et al., 1996), leur confère une grande rigueur théorique.

Dans cette section, nous commencerons par présenter les concepts fondamentaux des SVM, en introduisant la notion d'hyperplan séparateur et de **marge maximale** dans le cas linéairement séparable. Nous aborderons ensuite le traitement des données non linéairement séparables à travers l'introduction des **marges souples** et de la **pénalisation des erreurs**. Nous examinerons également le rôle central des fonctions noyau, qui permettent une projection implicite des données dans des espaces de dimension supérieure. Enfin, nous discuterons des aspects pratiques liés à l'implémentation des SVM, notamment le choix des hyperparamètres et les stratégies d'optimisation, avant de conclure par quelques applications concrètes illustrant l'efficacité de cette méthode.

1.2 Objectifs

Trouver un hyperplan qui sépare les exemples positifs des exemples négatifs. Cependant, alors qu'un réseau de neurones trouve un optimum local, une machine à vecteurs supports (MVS) trouve un optimum global. Notons aussi que les machines à vecteurs supports ne se cantonnent pas, elles non plus, aux hyperplans, mais peuvent construire des séparateurs non linéaires de diverses formes. La simple lecture de ce qui précède pourrait laisser penser que les réseaux de neurones doivent être remisés aux oubliettes. Cependant, diverses raisons pratiques font que les deux techniques (réseau de neurones d'une part, machine à vecteurs supports d'autre part) ont chacune des qualités et des défauts et que les deux techniques doivent être connues et que les deux ont de bonnes raisons d'être utilisées.

1.3 Principe Général

Les *Support Vector Machines* (SVM) peuvent être utilisées pour résoudre des problèmes de **discrimination**, c'est-à-dire décider à quelle classe appartient un échantillon, ou des problèmes de **régression**, c'est-à-dire prédire la valeur numérique d'une variable.

La résolution de ces deux problèmes repose sur la construction d'une fonction h qui associe à un vecteur d'entrée x une sortie y :

$$y = h(x)$$

Dans ce qui suit, on se limite à un problème de discrimination à deux classes (*discrimination binaire*), où :

$$y \in \{-1, 1\}$$

Le vecteur d'entrée x appartient à un espace $\mathcal{X}$ muni d'un produit scalaire. On peut, par exemple, considérer :

$$\mathcal{X} = \mathbb{R}^N$$

1.4 Machine à vecteurs supports linéaire

1.4.1 Problème de classification, SVM : C'est quoi, au juste ?

Avant toute chose, nous allons commencer par établir le **cadre des notions** qui seront abordées par la suite. En particulier, nous cherchons à répondre à la question suivante : ***qu'est-ce qu'un problème de classification ?***

Considérons l'exemple suivant. On se place dans le **plan** et l'on dispose de **deux catégories de données** : des **ronds rouges** et des **carrés bleus**, chacune occupant une **région distincte du plan**. Toutefois, la **frontière séparant ces deux régions** n'est pas connue *a priori*. L'objectif est que, lorsqu'un **nouveau point** est présenté à l'**algorithme de classification**, et que seule sa **position dans le plan** est connue, celui-ci soit capable de **prédire la catégorie** à laquelle il appartient (rond rouge ou carré bleu).

Nous sommes ainsi face à un **problème de classification** : pour chaque **nouvelle entrée**, il s'agit de **déterminer automatiquement la catégorie** à laquelle cette entrée appartient.

Autrement dit, le **cœur du problème** consiste à identifier la **frontière séparant les différentes catégories**. Une fois cette frontière connue, il suffit de déterminer de quel côté elle se situe le point considéré afin d'en **déduire sa classe**.

Les **machines à vecteurs de support** (*Support Vector Machines*, **SVM**) constituent une **solution efficace** pour résoudre ce type de problème de **classification**. Elles appartiennent à la famille des **classificateurs linéaires**, qui reposent sur une **séparation linéaire des données**, et disposent de leur propre méthode pour **déterminer la frontière optimale** entre les catégories.

Pour que le **SVM** puisse identifier cette frontière, il est nécessaire de lui fournir des **données d'entraînement**. En pratique, on fournit au SVM un **ensemble de points étiquetés**, dont on connaît déjà la classe (par exemple, des carrés rouges et des ronds bleus, comme illustré à la Figure 1.1).

À partir de ces données, le SVM va **estimer l'emplacement le plus plausible de la frontière**. Cette phase correspond à la **période d'entraînement**, étape fondamentale de tout **algorithme d'apprentissage automatique**.

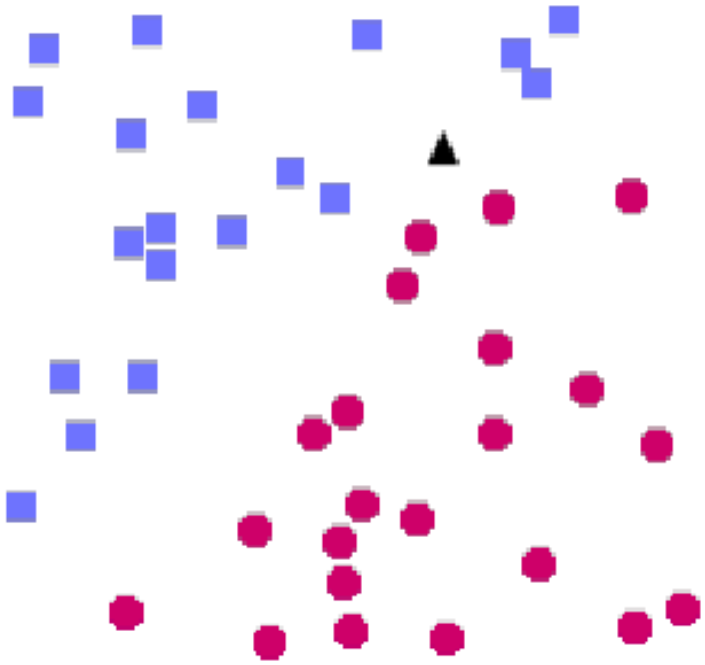

FIGURE 1.1 – L'objet à classer (le triangle noir) est-il un rond rouge ou bien un carré bleu ?

Une fois la phase d'entraînement terminée, le SVM a ainsi trouvé, à partir des données d'entraînement, l'emplacement supposé de la frontière. En quelque sorte, il a appris l'emplacement de la frontière grâce aux données d'entraînement. Qui plus est, le SVM est maintenant capable de prédire à quelle catégorie appartient une entrée qu'il n'avait jamais vue auparavant et sans intervention humaine (comme c'est le cas avec le triangle noir dans la **Figure 1.2**) : c'est là tout l'intérêt de l'apprentissage automatique.

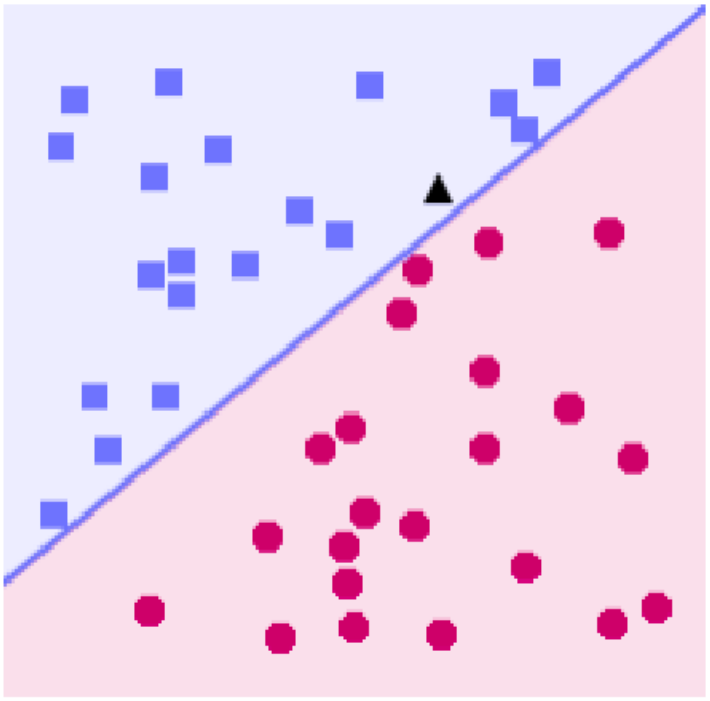

FIGURE 1.2 – Notre SVM, muni des donn'ees d'entraînement (les carr'es bleus et les ronds rouges d'ejà indiqu'es comme tels par l'utilisateur), a tranch'e : le triangle noir est en fait un carre bleu

1.5 Formalisme des SVM

De façon plus générale que dans les exemples précédents, les **machines à vecteurs de support** (SVM) ne se limitent pas à la séparation de points dans le plan. Elles permettent de séparer des données dans un **espace de dimension quelconque**.

Par exemple, dans un problème de **classification de fleurs par espèce**, si l'on dispose d'informations telles que la **taille**, le **nombre de pétales** et le **diamètre de la tige**, les données peuvent être représentées dans un espace de **dimension 3**. Un autre exemple courant est celui de la **reconnaissance d'images** : une image en niveaux de gris de 28×28 pixels contient 784 pixels, et peut ainsi être représentée comme un point dans un espace de **dimension 784**. Il est donc fréquent de travailler dans des espaces de **plusieurs milliers de dimensions**.

Fondamentalement, un SVM cherche à déterminer un **hyperplan** séparant au mieux les deux catégories du problème. Un **hyperplan vectoriel** passe nécessairement par l'**origine**. C'est pour cette raison que l'on introduit un **hyperplan affine**, qui n'est pas contraint de passer par l'origine. Ainsi, si l'on se place dans $\mathbb{R}^n$, pendant son entraînement le SVM calculera un **hyperplan vectoriel** d'équation :

$$w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n = 0$$

ainsi qu'un scalaire b. C'est ce scalaire b qui permet de travailler avec un **hyperplan affine**, comme nous allons le voir.

Le vecteur

$$\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$$

est appelé **vecteur de poids**, et le scalaire b est appelé **biais**.

Une fois l'entraînement terminé, pour classer une nouvelle entrée

$$\mathbf{x} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \in \mathbb{R}^n,$$

le SVM regardera le **signe** de :

$$h(\mathbf{x}) = w_1 a_1 + \cdots + w_n a_n + b = \sum_{i=1}^{n} w_i \cdot a_i = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b.$$

Si $h(\mathbf{x}) \geq 0$, alors $\mathbf{x}$ se situe d'un côté de l'hyperplan affine et appartient à la **catégorie 1**. Sinon, $\mathbf{x}$ se trouve de l'autre côté de l'hyperplan et appartient à la **catégorie 2**.

En résumé, pour un point $\mathbf{x}$, la fonction h permet de déterminer **de quel côté de l'hyperplan affine il se trouve** :

$$\mathbf{x} \in \begin{cases} \text{catégorie 1} & \text{si } h(\mathbf{x}) \geq 0, \\ \text{catégorie 2} & \text{si } h(\mathbf{x}) < 0. \end{cases}$$

1.6 Principe de la méthode

Considérons chaque individu $\mathbf{x}_i$ comme un point dans un espace à P dimensions. On peut également le voir comme un **vecteur** $\vec{x}_i$. Dans la suite, nous alternerons entre les deux points de vue selon le contexte.

Si le problème est **linéairement séparable**, les individus **positifs** et **négatifs** sont séparés par un **hyperplan** H. Notons H^+ l'hyperplan parallèle à H qui contient l'individu positif le plus proche de H, et H^- l'hyperplan contenant l'individu négatif le plus proche (voir **figure 1.3**).

Une **MVS linéaire** (*Machine à Vecteurs de Support*) recherche l'hyperplan qui sépare les données de manière à **maximiser la distance entre H^+ et H^-**. Cet écart est appelé la **marge**.

Équation de l'hyperplan

Un hyperplan est défini par l'équation :

$$y = \langle \mathbf{w}, \mathbf{x} \rangle + b$$

où $\langle \mathbf{w}, \mathbf{x} \rangle$ désigne le **produit scalaire** entre les vecteurs $\mathbf{w}$ et $\mathbf{x}$.

Pour une donnée $\mathbf{x}$ appartenant à la classe y, on cherche $\mathbf{w}$ tel que :

$$\begin{cases} \langle \mathbf{w}, \mathbf{x} \rangle + b \geq 1 & \text{si } y = +1, \\ \langle \mathbf{w}, \mathbf{x} \rangle + b \leq -1 & \text{si } y = -1. \end{cases}$$

Cette condition peut se réécrire de façon compacte sous la forme :

$$y\left(\langle \mathbf{w}, \mathbf{x} \rangle + b\right) - 1 \geq 0.$$

1.7 Calcul de la marge

Si l'on prend un point $\mathbf{x}_k \in \mathbb{R}^n$, on peut montrer que sa **distance à l'hyperplan** défini par le vecteur support $\mathbf{w}$ et le biais b est donnée par :

$$l_k \frac{\mathbf{w}^{\mathrm{T}} \mathbf{x}_k + b}{\|\mathbf{w}\|},$$

où $\|\mathbf{w}\|$ désigne la **norme euclidienne** de $\mathbf{w}$, et l_k est le label de la classe ($+1$ ou -1).

La **marge** d'un hyperplan de paramètres $(\mathbf{w}, b)$ par rapport à un ensemble de points $(\mathbf{x}_k)$ est donc :

$$\gamma = \min_k \; l_k \frac{\mathbf{w}^{\mathrm{T}} \mathbf{x}_k + b}{\|\mathbf{w}\|}.$$

Rappel : la marge correspond à la **distance minimale** de l'hyperplan à un des points d'entraînement.

Démonstration

On cherche à **maximiser la largeur de la marge**.

1. Le vecteur $\mathbf{w}$ est **perpendiculaire à l'hyperplan** H.
2. Soit $B \in H^+$ et $A \in H^-$ le point le plus proche de B (voir figure 8.1).
3. Pour tout point O, on a :
$$\overrightarrow{OB} = \overrightarrow{OA} + \overrightarrow{AB}.$$
4. Par définition des points A et B, $\overrightarrow{AB}$ est **parallèle à** $\mathbf{w}$. Il existe donc $\lambda \in \mathbb{R}$ tel que :
$$\overrightarrow{AB} = \lambda \mathbf{w}, \quad \text{soit} \quad \overrightarrow{OB} = \overrightarrow{OA} + \lambda \mathbf{w}.$$
5. On veut que A, B, H^- et H^+ vérifient :
$$B \in H^+ \Rightarrow \langle \mathbf{w}, \overrightarrow{OB} \rangle + b = 1, \quad A \in H^- \Rightarrow \langle \mathbf{w}, \overrightarrow{OA} \rangle + b = -1.$$
6. Donc :
$$\langle \mathbf{w}, \overrightarrow{OA} + \lambda \mathbf{w} \rangle + b = 1.$$
7. En développant :
$$\langle \mathbf{w}, \overrightarrow{OA} \rangle + b + \lambda \langle \mathbf{w}, \mathbf{w} \rangle = 1.$$
8. Or :
$$\langle \mathbf{w}, \overrightarrow{OA} \rangle + b = -1.$$
9. Donc :
$$\lambda \langle \mathbf{w}, \mathbf{w} \rangle = 2.$$

10. Ainsi :

$$\lambda = \frac{2}{\|\mathbf{w}\|^2}.$$

11. La largeur de la marge est donc :

$$|\lambda|\|\mathbf{w}\| = \frac{2}{\|\mathbf{w}\|}.$$

Conclusion : pour **maximiser la marge**, il faut **minimiser la norme de w**. En effet, une norme plus petite implique une marge plus grande, ce qui permet d'améliorer la séparation entre les classes.

Remarque : les individus positifs sont représentés par un +, et les individus négatifs par un − Les individus **positifs** sont représentés par un +, et les individus **négatifs** par un −.

On a représenté un **hyperplan** H qui sépare les positifs des négatifs. (**Remarque :** sur le schéma, il ne s'agit pas de l'hyperplan qui sépare **au mieux** les deux ensembles.)

Nous avons également représenté H^+ et H^-, tous deux **parallèles à** H.

Soit B un point de H^+ et A le point le plus proche de B appartenant à H^- (voir figure 1.3).

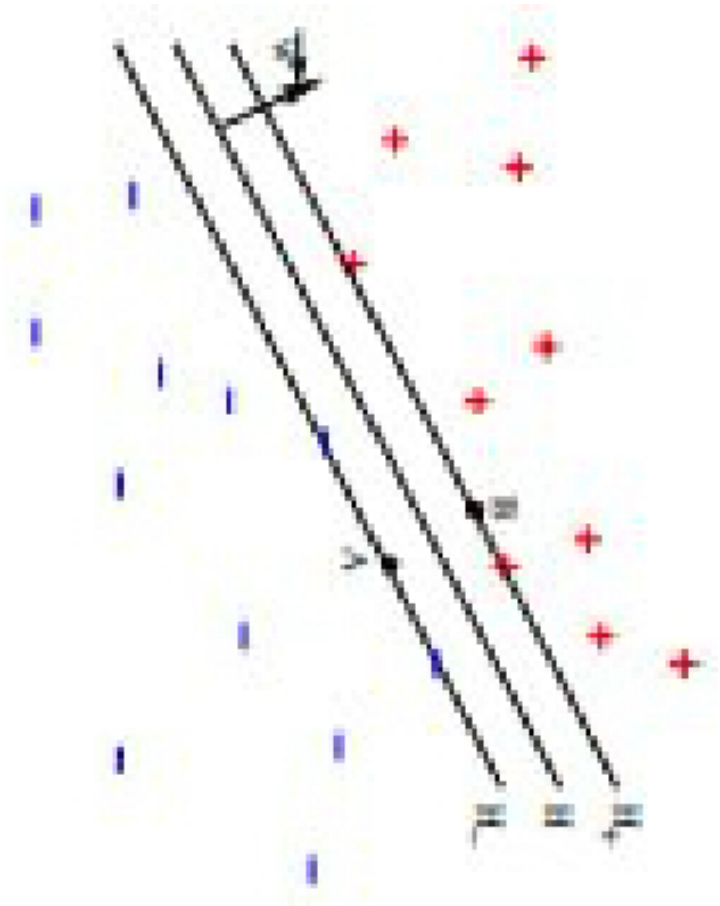

FIGURE 1.3 – Schéma illustratif : groupe H+ et H-.

1.8 Machine à vecteurs supports non linéaire

1.8.1 Introduction aux SVM non linéaires

Les Machines à Vecteurs de Support (SVM) sont des modèles d'apprentissage supervisé populaires pour la classification et la régression, introduits dans les années 1990 par Vapnik et ses collègues. Leur principe de base est de trouver un hyperplan qui maximise la marge entre les classes, défini par les vecteurs de support, qui sont les points de données les plus proches de cet hyperplan. Cependant, cette approche suppose que les données sont linéairement séparables, ce qui n'est pas toujours le cas dans des scénarios réels. Par exemple, imaginez des données où deux classes forment des cercles concentriques : aucune ligne droite ne peut les séparer efficacement.

C'est ici que les SVM non linéaires entrent en jeu. Ils étendent les capacités des SVM linéaires en traitant des données non linéairement séparables, c'est-à-dire des données où les classes ne peuvent pas être divisées par une frontière linéaire dans l'espace d'origine. Les SVM non linéaires utilisent des fonctions noyau pour transformer les données dans un espace de dimension supérieure où elles deviennent linéairement séparables, permettant ainsi de capturer des relations complexes.

1.8.2 Le problème des données non linéaires

Dans le cadre de la classification avec des algorithmes comme les Machines à Vecteurs de Support (SVM), un défi majeur survient lorsque les données ne sont pas linéairement séparables. On parle de données non linéaires lorsque les classes ne peuvent pas être divisées par une frontière linéaire simple, comme une droite en deux dimensions ou un hyperplan dans des dimensions supérieures. Ce problème est fréquent dans les applications réelles, où les relations entre les variables sont souvent complexes et ne suivent pas des motifs droits ou plats.

Prenons un exemple concret : imaginez deux classes de données disposées en cercles concentriques, l'une formant un petit cercle à l'intérieur et l'autre un anneau autour. Aucune ligne droite ne peut séparer parfaitement ces deux groupes sans inclure des points de l'autre classe. Un autre cas classique est le problème XOR, où quatre points – $(0, 0)$ et $(1, 1)$ dans une classe, $(0, 1)$ et $(1, 0)$ dans une autre – forment un motif impossible à séparer linéairement. De même, dans le dataset IRIS, bien connu en apprentissage automatique, les espèces Versicolor et Virginica se chevauchent dans l'espace des caractéristiques (comme la longueur et la largeur des pétales), rendant une séparation linéaire imparfaite.

Pour les SVM linéaires, qui reposent sur la recherche d'un hyperplan maximisant la marge entre les classes, ce type de données pose un obstacle insurmontable. Si les classes ne sont pas linéairement séparables, aucun hyperplan ne peut être trouvé sans accepter des erreurs de classification, ce qui compromet la précision du modèle. Par exemple, dans le cas des cercles concentriques, un SVM linéaire ne pourrait tracer qu'une droite coupant les deux cercles, mélangeant ainsi les points des deux classes.

Ce problème est d'autant plus critique que les données réelles – qu'il s'agisse de pixels d'images, de mots dans un texte ou de mesures biologiques – présentent souvent des structures non linéaires. Identifier cette limitation est donc essentiel pour comprendre pourquoi les SVM linéaires échouent dans ces cas et pourquoi une approche plus avancée, comme les SVM non linéaires, devient nécessaire.

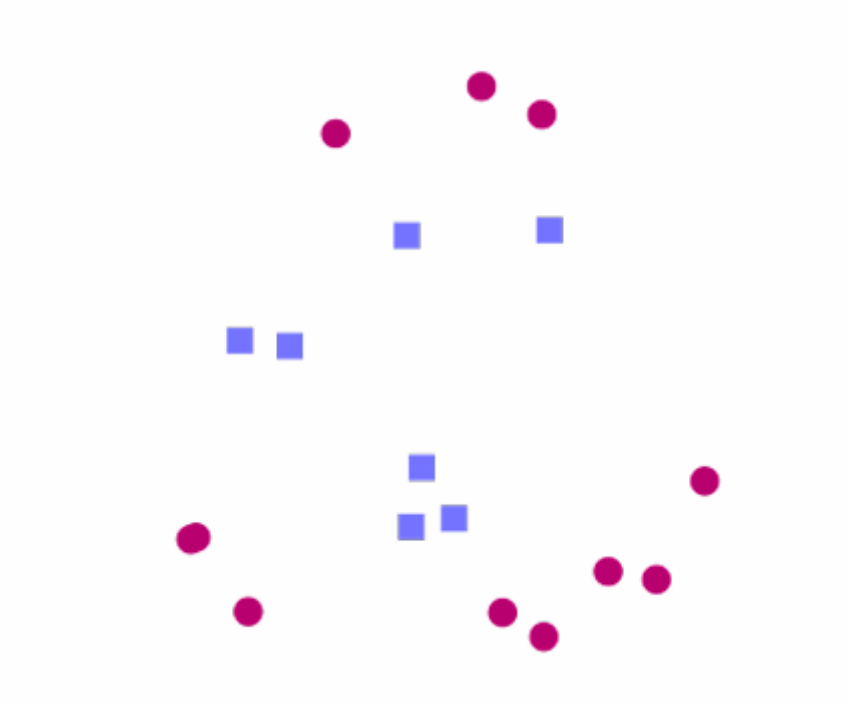

FIGURE 1.4 – Données d'entraînement non linéairement séparables

1.8.3 Principe des SVM non linéaires

Face au problème des données non linéaires, les SVM non linéaires offrent une solution ingénieuse en modifiant la manière dont les données sont traitées. Leur principe repose sur une idée fondamentale : transformer les données pour les rendre linéairement séparables, tout en conservant l'objectif de base des SVM, qui est de maximiser la marge entre les classes.

Transformation des données

Le premier concept clé est la projection des données dans un espace de dimension supérieure. Quand les données ne peuvent pas être séparées par une droite dans leur espace d'origine, l'idée est de les projeter dans un espace plus complexe où une séparation linéaire devient possible. Par exemple, prenons des données en deux dimensions (2D) formant deux cercles concentriques : une classe à l'intérieur, l'autre à l'extérieur. Aucune droite ne peut les séparer proprement en 2D. En revanche, si on transforme ces données en un espace tridimensionnel (3D) – par exemple, en ajoutant une coordonnée basée sur le carré de la distance au centre –, les points peuvent être "soulevés" différemment. Dans cet espace 3D, les cercles deviennent séparables par un plan plat. Cette transformation permet de contourner la limitation des frontières linéaires dans l'espace initial.

Hyperplan dans l'espace transformé

Une fois les données projetées dans cet espace de dimension supérieure, le SVM fonctionne comme un modèle linéaire classique : il cherche un hyperplan capable de séparer les classes. Dans notre exemple des cercles, le plan en 3D agit comme une surface qui passe entre les points "soulevés" de la classe intérieure et ceux de la classe extérieure. Ce

qui est remarquable, c'est que cet hyperplan, bien que linéaire dans l'espace transformé, correspond à une frontière non linéaire (comme un cercle) lorsqu'on le ramène à l'espace 2D d'origine. Ainsi, les SVM non linéaires exploitent cette astuce pour gérer des motifs complexes tout en restant ancrés dans une logique de séparation linéaire.

Maximisation de la marge

Enfin, comme pour les SVM linéaires, l'objectif reste de maximiser la marge. Dans l'espace transformé, le SVM identifie les vecteurs de support – les points les plus proches de l'hyperplan – et ajuste cet hyperplan pour qu'il soit le plus éloigné possible de ces points, des deux côtés. Cette maximisation garantit que le modèle reste robuste et généralise bien, même dans un espace de dimension supérieure. Ainsi, bien que la transformation change la géométrie des données, le principe fondamental des SVM demeure inchangé : trouver la frontière optimale qui sépare les classes avec la plus grande marge possible.

1.8.4 Plongée en dimension supérieure

Pour contourner le problème, l'idée est donc la suivante : il est impossible de séparer linéairement les données dans notre espace vectoriel ? Qu'importe ! Essayons dans un autre espace. De façon générale, il est courant de ne pas pouvoir séparer les données parce que l'espace est de trop petite dimension. Si l'on arrivait à transposer les données dans un espace de plus grande dimension, on arriverait peut-être à trouver un hyperplan séparateur.

Je sens que j'en ai peut-être perdu quelques-uns en cours de route. Considérons l'exemple suivant pour mieux comprendre. On souhaite construire un SVM qui, à partir de la taille d'un individu, détermine si cet individu est un jeune adolescent (entre 12 et 16 ans, carrés bleus) ou non (ronds rouges).

FIGURE 1.5 – Illustration : données sur une droite

Entre 10 et 12 ans, on mesure entre 120 et 140 cm ; entre 12 et 16 ans, entre 140 et 165 cm ; entre 16 et 18 ans, on mesure plus de 165 cm. Peut-on trouver un hyperplan qui sépare les jeunes de 12–16 ans des autres ? On constate que l'on ne peut pas trouver d'hyperplan séparateur (ici, on est en dimension 1, un hyperplan séparateur est donc un simple point). Ce qui est embêtant : si l'on ne peut pas trouver d'hyperplan séparateur, notre SVM ne sera pas capable de s'entraîner, et encore moins de classer de nouvelles entrées...

On essaie donc de trouver un nouvel espace, généralement de dimension supérieure, dans lequel on peut projeter nos valeurs d'entraînement, et dans lequel on pourra trouver un séparateur linéaire. Dans cet exemple, je vous propose d'utiliser la projection : $\phi : x \mapsto \left(\frac{x-150}{10}, \left(\frac{x-150}{10}\right)^2\right)$. On passe donc de l'espace vectoriel $\mathbb{R}$, de dimension 1, à l'espace vectoriel $\mathbb{R}^2$, de dimension 2. Les données d'entraînement obtenues après cette projection sont les suivantes :

On observe que les données deviennent alors linéairement séparables, ce qui permet à notre SVM de fonctionner correctement ! Plus formellement, l'idée de cette redescription

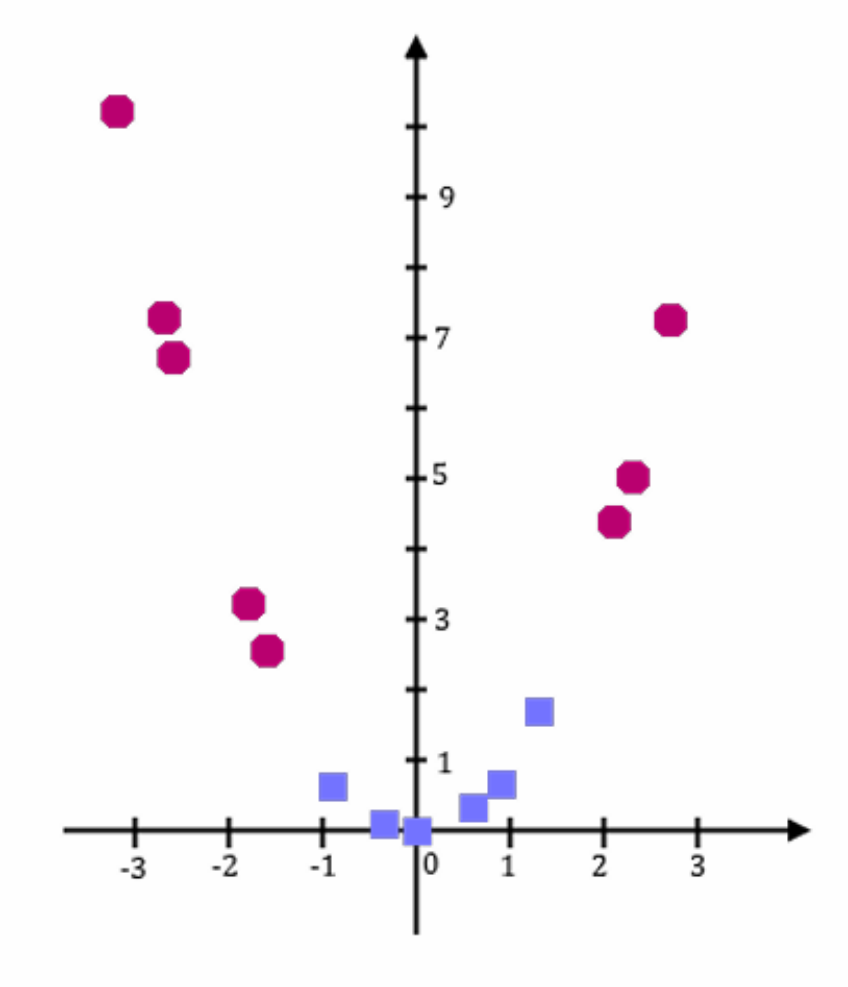

FIGURE 1.6 – Données d'entraînement après projection dans l'espace $\mathbb{R}^2$

du problème est de considérer que l'espace actuel, appelé espace de description, est de dimension trop petite ; alors que si on plongeait les données dans un espace de dimension supérieure, appelé espace de redescription, les données seraient linéairement séparables.

Évidemment, il ne suffit pas de simplement ajouter des dimensions à l'espace de description, car cela ne résoudrait pas le problème. Il faut au contraire redistribuer les points depuis l'espace de description vers l'espace de redescription, à l'aide d'une fonction, nécessairement non linéaire. On définit l'opération de redescription des points x de E vers E' par l'opération :

$$\phi : E \to E', \quad x \mapsto \phi(x) \tag{1.1}$$

Dans ce nouvel espace E', on va tenter d'entraîner le SVM comme nous l'aurions fait dans l'espace initial E. Si les données y sont linéairement séparables, c'est gagné ! Par la suite, pour classer un point x, il suffira de classer $\phi(x)$, ce qui nous permet d'obtenir un SVM fonctionnel.

1.8.5 Le « Kernel Trick » (Astuce du noyau)

Malheureusement, plus la dimension de l'espace augmente, plus les calculs deviennent complexes et longs. Que se passe-t-il alors si l'on travaille en dimension infinie ? Il est

néanmoins possible de simplifier les calculs. En posant le problème d'optimisation quadratique dans l'espace E', on s'aperçoit que les seules apparitions de ϕ sont sous la forme $\phi(x_i)^T \cdot \phi(x_j)$. Il en va de même pour l'expression de la fonction de classification h. Par conséquent, il n'est pas nécessaire de connaître explicitement E', ni même ϕ : il suffit de connaître les valeurs $\phi(x_i)^T \cdot \phi(x_j)$, qui ne dépendent que des x_i.

On définit alors une fonction noyau $K : E \times E \to \mathbb{R}$ de la manière suivante :

$$K(x, x') = \phi(x)^T \cdot \phi(x') \tag{1.2}$$

À ce stade, le calcul de l'hyperplan séparateur dans E' ne nécessite ni la connaissance de E', ni de ϕ, mais uniquement de K.

Par exemple, supposons que E soit l'espace de description, de dimension n, et que l'espace de redescription E' soit de dimension n^2. Le calcul de $K(x_i, x_j)$ se fait en $O(n)$, tandis que le calcul direct de $\phi(x_i)^T \cdot \phi(x_j)$, qui donne le même résultat, se fait en $O(n^2)$, ce qui représente un avantage immédiat en termes de temps de calcul.

C'est à partir de ce constat qu'émerge le *kernel trick*, ou astuce du noyau. Puisque seule la connaissance de K est nécessaire, pourquoi ne pas utiliser une fonction K quelconque, correspondant à une redescription dans un espace E' quelconque, et résoudre le problème de séparation linéaire sans se préoccuper de l'espace de redescription dans lequel on évolue ?

Grâce au théorème de Mercer, on sait qu'une condition suffisante pour qu'une fonction K soit une fonction noyau est qu'elle soit continue, symétrique et semi-définie positive.

Propriétés d'un noyau symétrique semi-défini positif

Une fonction ϕ est dite **symétrique** si et seulement si, pour tout $x, y \in E$, on a $\phi(x, y) = \phi(y, x)$.

Une fonction symétrique K est dite **semi-définie positive** si et seulement si, pour tout ensemble fini $\{x_1, \ldots, x_n\} \subset E$, et pour tous réels $c_1, \ldots, c_n$,

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0. \tag{1.3}$$

Par exemple, le produit scalaire usuel $(a, b) \mapsto \sum_{i=1}^{N} a_i b_i$ est symétrique et semi-défini positif, car

$$\forall \{x_1, \ldots, x_n\} \subset E, \forall c_1, \ldots, c_n \in \mathbb{R}, \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j (x_i \cdot x_j) = \left\| \sum_{i=1}^{n} c_i x_i \right\|^2 \geq 0. \tag{1.4}$$

Ainsi, il est possible d'utiliser n'importe quelle fonction noyau afin de réaliser une redescription dans un espace de dimension supérieure. La fonction noyau étant définie sur l'espace de description E (et non sur l'espace de redescription E', de plus grande dimension), les calculs sont beaucoup plus rapides.

Pour résumer, voici les quatre grandes idées à la base du *kernel trick* :

— Les données décrites dans l'espace d'entrée E sont projetées dans un espace de redescription E'.

— Des régularités linéaires sont cherchées dans cet espace E'.

— Les algorithmes de recherche n'ont pas besoin de connaître les coordonnées des projections des données dans E', mais seulement leurs produits scalaires.
— Ces produits scalaires peuvent être calculés efficacement grâce à l'utilisation de fonctions noyau.

1.8.6 Les principaux noyaux utilisés

Le "kernel trick" repose sur des fonctions noyau spécifiques qui définissent comment les données sont implicitement transformées dans un espace de dimension supérieure. Chaque noyau a ses propres caractéristiques, paramètres et cas d'utilisation. Voici les trois principaux noyaux utilisés dans les SVM non linéaires, suivis d'une comparaison rapide.

Noyau Polynomial

Formule :

$$K(x, x') = (\alpha \cdot x^\top x' + \lambda)^d \tag{1.5}$$

Description : Le noyau polynomial évalue la similarité entre deux vecteurs en élevant leur produit scalaire, ajusté par des constantes, à une puissance donnée. Ce noyau est particulièrement utile pour modéliser des relations non linéaires en transformant l'espace d'entrée en un espace de dimensions supérieures, permettant ainsi de capturer des interactions complexes entre les variables.

Paramètres :

— α : facteur d'échelle du produit scalaire.
— λ : constante ajoutée pour ajuster la souplesse du modèle.
— d : degré du polynôme, déterminant la complexité de la frontière de décision.

Exemple d'application : Reconnaissance d'écriture manuscrite

Dans la reconnaissance de chiffres manuscrits, comme ceux de la base de données MNIST, les machines à vecteurs de support (SVM) avec un noyau polynomial peuvent classifier efficacement les images en tenant compte des interactions complexes entre les pixels. Le noyau polynomial permet de modéliser des frontières de décision non linéaires, améliorant ainsi la précision de la classification.

Noyau Gaussien (RBF - Radial Basis Function)

Formule :

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) \tag{1.6}$$

Description : Le noyau gaussien mesure la similarité entre deux points en fonction de la distance euclidienne qui les sépare. Il est particulièrement efficace pour capturer des relations complexes dans les données, même lorsque la séparation linéaire n'est pas possible dans l'espace d'origine.

Paramètre :

— σ : paramètre de lissage déterminant la portée de l'influence d'un point de données sur un autre.

Exemple d'application : Détection d'e-mails indésirables (spam)

Les filtres anti-spam utilisent des SVM avec un noyau gaussien pour distinguer les e-mails légitimes des spams. Le noyau RBF capture les relations complexes entre les caractéristiques des e-mails, telles que la fréquence des mots, les structures de phrases et les métadonnées, permettant une classification précise même lorsque les spams présentent des variations subtiles.

Noyau Laplacien

Formule :

$$K(x, x') = \exp\left(-\frac{\|x - x'\|}{\sigma}\right) \tag{1.7}$$

Description : Semblable au noyau gaussien, le noyau laplacien utilise la distance euclidienne pour évaluer la similarité, mais avec une décroissance exponentielle différente. Il est souvent préféré lorsque les données présentent des variations plus abruptes ou des discontinuités.

Paramètre :

— σ : paramètre contrôlant la largeur de la fonction noyau, influençant la sensibilité aux variations locales des données.

Exemple d'application : Analyse de données biologiques

Dans la bioinformatique, le noyau laplacien est utilisé pour comparer des séquences génétiques ou protéiques. En mesurant la similarité entre les séquences, il aide à identifier des gènes homologues, à prédire des structures protéiques ou à classifier des espèces biologiques, même lorsque les séquences présentent des variations significatives.

Noyau Rationnel Quadratique

Formule :

$$K(x, x') = 1 - \frac{\|x - x'\|^2}{\|x - x'\|^2 + \sigma} \tag{1.8}$$

Description : Le noyau rationnel quadratique est une variante du noyau gaussien qui offre une mesure de similarité basée sur une fraction rationnelle de la distance entre les points. Il est utile pour modéliser des relations où la similarité diminue de manière non exponentielle avec la distance.

Paramètre :

— σ : paramètre ajustant la courbure de la fonction noyau, affectant la manière dont la similarité décroît avec la distance.

Exemple d'application : Systèmes de recommandation

Les plateformes de streaming musical ou vidéo utilisent le noyau rationnel quadratique pour modéliser la similarité entre les préférences des utilisateurs. En évaluant la distance entre les profils d'écoute ou de visionnage, ce noyau permet de recommander des contenus pertinents en tenant compte des variations subtiles dans les habitudes des utilisateurs.

Chacun de ces noyaux possède ses avantages et ses inconvénients, comme décrits dans le document. C'est donc à l'utilisateur de choisir le noyau, et les paramètres de ce noyau, qui correspond le mieux à son problème. Et comment choisir son noyau, me demanderez-vous ? Eh bien... Je n'ai pas de méthode à vous donner, ça dépend énormément de votre jeu de données.

1.9 Conclusion

Les Support Vector Machines (SVM) sont des algorithmes de classification et de régression puissants, particulièrement efficaces pour les problèmes à haute dimensionnalité et pour les jeux de données où les classes sont bien séparables. Grâce à l'utilisation des *marges maximales* et des *noyaux* (*kernels*), les SVM peuvent modéliser des frontières de décision complexes tout en minimisant le risque de surapprentissage. Bien qu'ils puissent être sensibles aux choix des hyperparamètres et coûteux en calcul pour de très grands ensembles de données, leur précision et leur robustesse font des SVM un outil essentiel dans la boîte à outils de l'apprentissage automatique.

Chapitre 2

Random Forests (RF)

2.1 Introduction

Les praticiens du machine learning sont souvent confrontés à un défi fondamental : les arbres de décision simples, bien qu'intuitifs et interprétables, ont tendance à surapprendre (*overfitting*) les données d'entraînement. Un arbre de décision qui mémorise parfaitement chaque exemple d'apprentissage aura généralement de mauvaises performances sur de nouvelles données non vues. C'est dans ce contexte que les forêts aléatoires (*Random Forests*) entrent en jeu.

Les forêts aléatoires, introduites par Leo Breiman en 2001, constituent une méthode puissante d'apprentissage en ensemble qui combine plusieurs arbres de décision afin de produire un prédicteur plus robuste et plus précis. L'idée fondamentale est remarquablement simple : si un seul expert peut commettre des erreurs, pourquoi ne pas consulter un comité d'experts et retenir l'opinion majoritaire ?

Ce principe, parfois appelé la *sagesse des foules* (*wisdom of crowds*), suggère que la prédiction moyenne issue de plusieurs modèles indépendants est généralement plus précise et plus stable que celle d'un modèle unique. Les forêts aléatoires mettent ce principe en pratique en entraînant de nombreux arbres de décision, chacun sur des versions légèrement différentes des données, puis en agrégeant leurs prédictions.

Dans le contexte de ce rapport, les forêts aléatoires occupent une position intermédiaire importante. Contrairement aux machines à vecteurs de support (*Support Vector Machines, SVM*), qui recherchent des frontières de décision optimales à l'aide de méthodes à noyaux sophistiquées, les forêts aléatoires reposent sur des structures arborescentes interprétables. À l'inverse de XGBoost, qui construit les arbres de manière séquentielle afin de corriger les erreurs précédentes, les forêts aléatoires construisent tous les arbres de façon indépendante et en parallèle.

Cette section vous guidera à travers les fondements théoriques des forêts aléatoires, leur implémentation algorithmique, les paramètres clés et les stratégies d'ajustement, leurs avantages et limitations pratiques, et enfin leur comparaison avec d'autres méthodes de la boîte à outils du machine learning. À la fin de cette section, vous comprendrez non seulement comment fonctionnent les forêts aléatoires, mais aussi quand et pourquoi les utiliser.

2.2 Fondements théoriques : l'apprentissage en ensemble

Pour comprendre pourquoi les forêts aléatoires sont si performantes, il est nécessaire de s'intéresser au principe de l'apprentissage en ensemble (*ensemble learning*). L'idée centrale est que la combinaison de plusieurs *apprenants faibles* (*weak learners*) peut produire un *apprenant fort* (*strong learner*) qui surpasse chacun des modèles pris individuellement.

Un concept fondamental du machine learning pour analyser ce phénomène est le com-

promis biais-variance (*bias–variance trade-off*). L'erreur de prédiction d'un modèle peut être décomposée en trois composantes :

$$\text{Erreur totale} = \text{Biais}^2 + \text{Variance} + \text{Erreur irréductible}$$

Le biais représente les erreurs systématiques, c'est-à-dire l'écart entre la prédiction moyenne du modèle et la valeur réelle. La variance mesure la sensibilité du modèle aux variations des données d'apprentissage, autrement dit à quel point les prédictions changent lorsque le modèle est entraîné sur des jeux de données différents. L'erreur irréductible correspond au bruit intrinsèque présent dans les données, qu'aucun modèle ne peut éliminer.

Les arbres de décision individuels, lorsqu'ils sont développés jusqu'à une profondeur maximale, présentent généralement un faible biais (car ils sont capables de modéliser des relations complexes), mais une variance élevée (car de légères variations dans les données d'entraînement peuvent produire des structures d'arbres très différentes). Ainsi, entraîner un arbre de décision profond sur des données légèrement modifiées peut conduire à un modèle totalement différent.

Les méthodes d'ensemble répondent à ce problème en exploitant une propriété mathématique clé : lorsque l'on moyenne plusieurs variables aléatoires indépendantes, la variance de la moyenne est inférieure à la variance de chacune des variables individuelles. Plus précisément, si l'on dispose de n prédictions indépendantes ayant chacune une variance σ^2, alors la variance de leur moyenne est donnée par :

$$\frac{\sigma^2}{n}$$

La technique fondamentale exploitant ce principe est le *Bootstrap Aggregating*, plus communément appelé *bagging*. Son fonctionnement est le suivant : au lieu d'entraîner un seul modèle sur l'ensemble du jeu de données, on génère plusieurs échantillons bootstrap (échantillonnage aléatoire avec remise) à partir des données originales. Chaque échantillon bootstrap a la même taille que le jeu de données initial, mais contient certaines observations répétées et en omet d'autres. Un modèle distinct est entraîné sur chaque échantillon, puis leurs prédictions sont agrégées.

D'un point de vue mathématique, si l'on considère K modèles produisant les prédictions $f_1(x), f_2(x), \ldots, f_K(x)$, la prédiction de l'ensemble est définie comme suit :

$$F(x) = \frac{1}{K} \sum_{k=1}^{K} f_k(x) \quad \text{(régression)}$$

$$F(x) = \text{mode}\{f_1(x), f_2(x), \ldots, f_K(x)\} \quad \text{(classification)}$$

Cette opération d'agrégation permet de réduire significativement la variance sans augmenter le biais, conduisant ainsi à un modèle qui généralise mieux sur de nouvelles données. Les forêts aléatoires vont encore plus loin que le bagging classique en introduisant une source supplémentaire d'aléa, qui sera abordée dans la suite.

Formule clé : la moyenne d'un ensemble de prédictions réduit la variance :

$$\text{Var}(\bar{X}) = \frac{\sigma^2}{n}$$

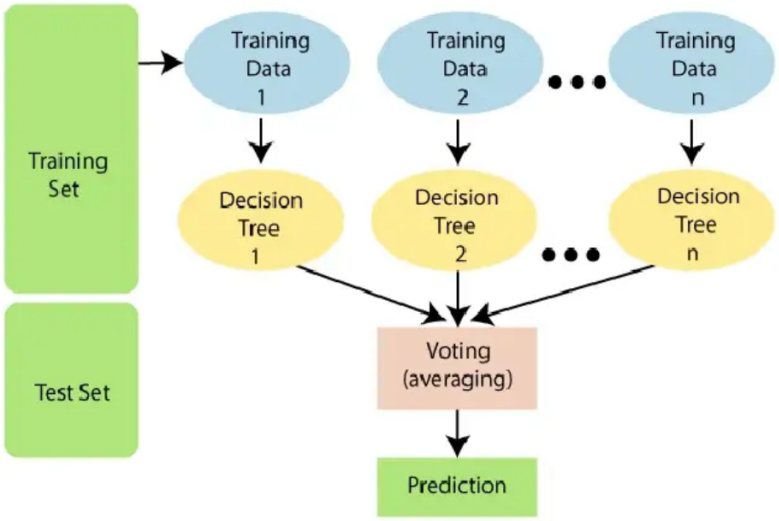

FIGURE 2.1 – Fonctionnement du Random Forest

2.3 Rappel sur les arbres de décision

Avant d'approfondir l'étude des forêts aléatoires, il est utile de rappeler le fonctionnement des arbres de décision individuels, puisqu'ils constituent les briques élémentaires de toute forêt.

Un arbre de décision est un modèle hiérarchique qui effectue des prédictions en posant une succession de questions binaires (oui/non) sur les variables d'entrée. Il est composé de nœuds internes (points de décision), de branches (réponses possibles) et de nœuds feuilles (prédictions finales). Par exemple, prédire si une personne achètera un produit peut impliquer des questions telles que « *Âge* > 30 *?* », suivies de « *Revenu* $> 50\,000$ *\$?* », et ainsi de suite.

La construction d'un arbre de décision repose sur l'algorithme CART (*Classification and Regression Trees*). En partant de l'ensemble des données d'apprentissage au niveau du nœud racine, l'algorithme divise récursivement les données en fonction des valeurs des caractéristiques qui séparent le mieux la variable cible. La question clé est alors : comment mesurer ce « meilleur » découpage ?

2.3.1 Critères de séparation

Pour les problèmes de classification, on utilise des mesures telles que l'impureté de Gini ou l'entropie.

L'impureté de Gini mesure la probabilité qu'un élément choisi aléatoirement soit mal classé :

$$\text{Gini} = 1 - \sum_{i=1}^{C} p_i^2$$

où p_i représente la proportion de la classe i dans le nœud. Un nœud pur, dans lequel toutes les observations appartiennent à une seule classe, a une impureté de Gini égale à 0. L'impureté est maximale lorsque les classes sont réparties de manière uniforme.

L'entropie mesure le contenu informationnel d'un nœud :

$$\text{Entropie} = -\sum_{i=1}^{C} p_i \log_2 p_i$$

Pour les problèmes de régression, le critère utilisé est l'erreur quadratique moyenne (*Mean Squared Error, MSE*) :

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$$

où $\bar{y}$ représente la moyenne des valeurs cibles dans le nœud.

2.3.2 Construction récursive et surapprentissage

L'algorithme évalue toutes les divisions possibles pour l'ensemble des caractéristiques et sélectionne celle qui entraîne la plus grande diminution de l'impureté ou de l'erreur. Ce processus se poursuit de manière récursive jusqu'à ce qu'un critère d'arrêt soit atteint, tel qu'une profondeur maximale, un nombre minimal d'exemples par nœud, ou l'absence de gain supplémentaire.

C'est précisément à ce niveau que se situe le problème : un arbre de décision construit sans contraintes continuera à se diviser jusqu'à ce que chaque feuille contienne un seul exemple d'apprentissage, mémorisant parfaitement les données d'entraînement. Un tel arbre présente une erreur nulle sur les données d'apprentissage, mais des performances très faibles sur les données de test, ce qui correspond à un cas classique de surapprentissage (*overfitting*).

Les arbres de décision individuels sont donc considérés comme des modèles à *forte variance et faible biais*. Ils sont suffisamment flexibles pour capturer des relations complexes (faible biais), mais cette flexibilité excessive les rend sensibles au bruit et fortement dépendants des échantillons d'apprentissage (forte variance). Cette caractéristique en fait des candidats idéaux pour les méthodes d'apprentissage en ensemble visant à réduire la variance, ce qui nous conduit naturellement aux forêts aléatoires.

Mesures clés :

$$\text{Impureté de Gini} = 1 - \sum p_i^2, \quad \text{Entropie} = -\sum p_i \log_2 p_i, \quad \text{MSE} = \frac{1}{n}\sum (y_i - \bar{y})^2$$

2.4 Algorithme des forêts aléatoires

Nous arrivons maintenant au cœur du fonctionnement des forêts aléatoires. L'algorithme introduit deux sources clés de hasard qui le rendent plus puissant qu'un simple *bagging* d'arbres de décision.

2.4.1 Étape 1 : Échantillonnage bootstrap

Pour chacun des K arbres que l'on souhaite construire (généralement entre 100 et 1000 arbres), on crée un échantillon bootstrap à partir des données d'apprentissage. Cela

consiste à sélectionner aléatoirement n observations avec remise à partir du jeu de données original de taille n.

Étant donné que l'échantillonnage se fait avec remise, certaines observations peuvent apparaître plusieurs fois, tandis que d'autres ne sont pas sélectionnées du tout. En moyenne, chaque échantillon bootstrap contient environ 63,2 % d'observations uniques provenant du jeu de données original.

2.4.2 Étape 2 : Sélection aléatoire des caractéristiques

C'est à cette étape que les forêts aléatoires se distinguent du *bagging* standard. Lors de la construction de chaque arbre, à chaque point de division, au lieu de considérer l'ensemble des p caractéristiques disponibles, on sélectionne aléatoirement un sous-ensemble de m caractéristiques, avec $m < p$. La meilleure division est alors choisie uniquement parmi ces m caractéristiques.

Ce processus est répété à chaque nœud de l'arbre, avec un nouveau sous-ensemble aléatoire de caractéristiques sélectionné à chaque fois.

Cette source de hasard est cruciale : elle permet de décorréler les arbres entre eux, évitant qu'ils ne commettent tous les mêmes erreurs. Si une caractéristique est particulièrement dominante, l'absence de sélection aléatoire conduirait la majorité des arbres à l'utiliser dès la première division, rendant les arbres fortement corrélés. En imposant l'exploration de caractéristiques différentes, on introduit de la diversité au sein de l'ensemble.

2.4.3 Étape 3 : Croissance des arbres

Chaque arbre est développé jusqu'à sa profondeur maximale, sans élagage (*pruning*). Contrairement à la construction d'un arbre de décision unique — où l'on applique souvent des critères d'arrêt anticipés ou des techniques d'élagage pour éviter le surapprentissage — les forêts aléatoires autorisent chaque arbre à surapprendre complètement son échantillon bootstrap.

Ce choix peut sembler contre-intuitif, mais il repose sur un principe fondamental : c'est l'agrégation des prédictions de nombreux arbres fortement variables qui permet de lisser le surapprentissage individuel et d'améliorer la généralisation globale.

2.4.4 Étape 4 : Agrégation des prédictions

Une fois les K arbres entraînés, le processus de prédiction est simple :

- **Classification :** chaque arbre vote pour une classe, et la classe finale est celle qui obtient le plus grand nombre de votes (vote majoritaire).
- **Régression :** la prédiction finale correspond à la moyenne des prédictions de tous les arbres.

D'un point de vue mathématique :

$$\hat{y} = \text{mode}\{T_1(x), T_2(x), \ldots, T_K(x)\} \quad \text{(classification)}$$

$$\hat{y} = \frac{1}{K} \sum_{k=1}^{K} T_k(x) \quad \text{(régression)}$$

où $T_k(x)$ représente la prédiction du k-ième arbre pour une observation x.

2.4.5 Algorithm Pseudocode

```
For k = 1 to K:
1. Draw a bootstrap sample Z_k of size n from training data
2. Grow tree T_k using Z_k:
a. At each node, randomly select m features
b. Choose best split among these m features
c. Split node and repeat until stopping criterion
3. Store tree T_k

To predict for new input x:
- Classification: return majority vote from {T_1(x), ..., T_K(x)}
- Regression: return average of {T_1(x), ..., T_K(x)}
```

FIGURE 2.2 – Random Forest

Cette combinaison élégante de l'échantillonnage bootstrap et de la sélection aléatoire des caractéristiques permet de construire un ensemble puissant qui surpasse généralement à la fois les arbres de décision individuels et les arbres issus d'un simple *bagging*.

Idée clé : les deux sources de hasard l'échantillonnage bootstrap et la sélection aléatoire des caractéristiques génèrent des arbres diversifiés dont les erreurs individuelles tendent à s'annuler lors de l'agrégation, conduisant à de meilleures performances de généralisation.

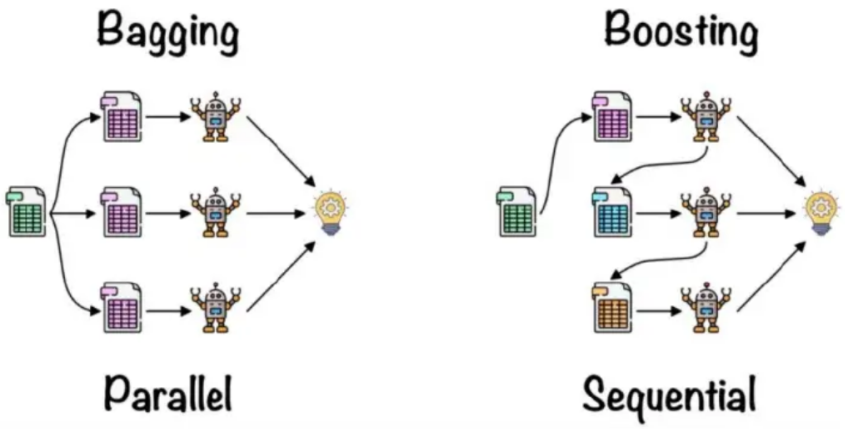

FIGURE 2.3 – Bagging vs Boosting

2.5 Hyperparamètres clés et leurs effets

La compréhension des hyperparamètres des forêts aléatoires est essentielle pour obtenir des performances optimales. Bien que les forêts aléatoires donnent généralement de bons résultats avec les paramètres par défaut, un ajustement fin de ces hyperparamètres peut améliorer significativement les performances.

2.5.1 Nombre d'arbres (`n_estimators` ou `ntree`)

Ce paramètre contrôle le nombre d'arbres à construire dans la forêt. En général, un plus grand nombre d'arbres conduit à de meilleures performances et à une plus grande stabilité,

mais avec des gains décroissants. L'erreur diminue généralement jusqu'à atteindre un plateau lorsque l'on augmente le nombre d'arbres.

- **Valeurs typiques :** 100 à 1000 arbres.
- **Effet :** un plus grand nombre d'arbres réduit la variance et stabilise les prédictions, mais augmente le temps de calcul.
- **Règle pratique :** commencer avec 100 à 500 arbres ; augmenter si les ressources de calcul le permettent et si les performances n'ont pas atteint un plateau.
- **Remarque importante :** contrairement aux méthodes de boosting, l'ajout d'arbres ne provoque généralement pas de surapprentissage ; les arbres supplémentaires deviennent simplement redondants.

2.5.2 Nombre de caractéristiques par division (`max_features` ou `mtry`)

Il s'agit sans doute du paramètre d'ajustement le plus important, car il contrôle le nombre de caractéristiques sélectionnées aléatoirement à chaque division.

- **Classification :** la valeur par défaut est $\sqrt{p}$, où p représente le nombre total de caractéristiques.
- **Régression :** la valeur par défaut est $p/3$.
- **Effet :** des valeurs plus petites produisent des arbres plus diversifiés (corrélation plus faible) mais individuellement plus faibles ; des valeurs plus grandes produisent des arbres plus performants individuellement mais plus fortement corrélés.
- **Stratégie d'ajustement :** tester des valeurs autour des paramètres par défaut ; souvent, la valeur optimale est proche de $\sqrt{p}$ en classification, mais explorer $p/3$, $p/2$ ou $2\sqrt{p}$ peut être bénéfique.

2.5.3 Contrôle de la profondeur des arbres

Plusieurs paramètres contrôlent la profondeur de croissance des arbres :

- `max_depth` : profondeur maximale de chaque arbre (par défaut : `None`, croissance jusqu'à pureté).
- `min_samples_split` : nombre minimal d'exemples requis pour diviser un nœud (souvent 2 par défaut).
- `min_samples_leaf` : nombre minimal d'exemples requis dans une feuille (souvent 1 par défaut).

Ces paramètres établissent un compromis entre la complexité du modèle et sa capacité de généralisation :

- Des arbres très profonds peuvent capturer des relations complexes mais risquent le surapprentissage.
- Des arbres peu profonds sont plus rapides mais peuvent sous-apprendre.
- En pratique, les forêts aléatoires sont robustes vis-à-vis de ces paramètres, et les valeurs par défaut donnent souvent de bons résultats.

2.5.4 Taille des échantillons bootstrap (`max_samples`)

Ce paramètre contrôle la taille des échantillons bootstrap en tant que fraction du jeu de données original.

— **Valeur par défaut :** 1.0 (taille identique au jeu de données initial).
— **Effet :** des valeurs plus faibles augmentent la diversité des arbres mais peuvent réduire la qualité individuelle de chaque arbre.
— **Quand réduire :** pour les jeux de données très volumineux, échantillonner une fraction plus faible permet d'accélérer l'entraînement avec une perte de précision souvent négligeable.

2.5.5 Stratégie pratique d'ajustement

1. Commencer avec les paramètres par défaut (ils offrent généralement une excellente base).
2. Ajuster d'abord `n_estimators` afin d'identifier le point où les performances atteignent un plateau.
3. Ajuster ensuite `max_features` à l'aide de la validation croisée, car c'est le paramètre le plus influent.
4. Modifier les paramètres de profondeur uniquement si un surapprentissage ou un sous-apprentissage est observé.
5. Utiliser l'erreur *out-of-bag* (OOB) pour une évaluation rapide sans recourir à la validation croisée.

La principale force des forêts aléatoires réside dans leur faible sensibilité aux hyperparamètres. Les valeurs par défaut constituent une base solide, et le modèle échoue rarement de manière catastrophique même en présence de choix sous-optimaux.

Paramètre clé : `max_features` est l'hyperparamètre le plus important, car il contrôle directement le compromis entre la diversité des arbres et leur puissance individuelle.

2.6 Estimation de l'erreur *Out-of-Bag* (OOB)

L'une des fonctionnalités les plus élégantes des forêts aléatoires est l'estimation de l'erreur *Out-of-Bag* (OOB), qui fournit un mécanisme de validation intégré sans nécessiter un jeu de données de test séparé.

2.6.1 Mathématiques derrière l'OOB

Lorsqu'on crée un échantillon bootstrap de n observations, on échantillonne avec remise. Cela signifie que certaines observations peuvent apparaître plusieurs fois tandis que d'autres ne sont pas sélectionnées.

La probabilité qu'une observation donnée ne soit pas sélectionnée lors d'un tirage est :

$$1 - \frac{1}{n}$$

Étant donné que l'on effectue n tirages indépendants, la probabilité qu'une observation ne soit jamais sélectionnée est :

$$P(\text{non sélectionnée}) = \left(1 - \frac{1}{n}\right)^n$$

Lorsque n tend vers l'infini, cette probabilité converge vers $1/e \approx 0,368$. Autrement dit, environ 36,8% des observations sont laissées de côté pour chaque échantillon bootstrap. Ces observations non incluses sont appelées *échantillons out-of-bag* pour cet arbre particulier.

2.6.2 Fonctionnement de l'erreur OOB

Voici l'astuce : pour chaque observation du jeu de données d'apprentissage, on peut identifier les arbres qui ne l'ont pas utilisée lors de leur entraînement (c'est-à-dire ceux pour lesquels l'observation était OOB). On effectue alors les prédictions pour cette observation uniquement avec ces arbres.

Le processus est le suivant :

1. Pour chaque observation i du jeu d'entraînement, identifier tous les arbres pour lesquels i était OOB.
2. Agréger les prédictions de ces arbres (vote majoritaire pour la classification, moyenne pour la régression).
3. Comparer la prédiction OOB à la valeur réelle.
4. Calculer l'erreur OOB globale sur l'ensemble des observations.

Mathématiquement, pour une observation i, soit C_i l'ensemble des arbres où i est OOB :

$$\hat{y}_i^{\text{OOB}} = \frac{1}{|C_i|} \sum_{k \in C_i} T_k(x_i)$$

L'erreur OOB est alors définie par :

$$\text{OOB\_Error} = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i^{\text{OOB}})$$

où L est la fonction de perte (perte 0-1 pour la classification, erreur quadratique pour la régression).

2.6.3 Valeur et avantages de l'OOB

L'erreur OOB fournit une estimation non biaisée de l'erreur de généralisation. Elle agit comme une validation croisée intégrée dans le processus d'entraînement :

— **Pas de jeu de validation séparé :** l'évaluation de performance se fait uniquement avec les données d'entraînement.
— **Efficace :** aucun entraînement supplémentaire n'est nécessaire.
— **Non biaisée :** chaque prédiction utilise uniquement les arbres n'ayant pas vu cette observation.
— **Surveillance de l'entraînement :** l'OOB permet de suivre la stabilisation de l'erreur et de déterminer le nombre d'arbres suffisant.

2.6.4 Applications pratiques

1. Sélection de modèle : comparer les erreurs OOB pour différents réglages d'hyperparamètres.

2. Critère d'arrêt : arrêter l'ajout d'arbres lorsque l'erreur OOB atteint un plateau.
3. Sélection de variables : retirer certaines caractéristiques et vérifier si l'erreur OOB s'améliore.
4. Évaluation rapide : estimer les performances sans jeu de validation (la validation finale doit toutefois utiliser un jeu de test indépendant).

2.6.5 Limites

Bien que puissante, l'erreur OOB présente certaines limites :
— Légèrement optimiste par rapport à une validation avec un jeu de test indépendant (les modèles voient l'ensemble des données collectivement).
— Nécessite un nombre suffisant d'arbres pour obtenir des estimations stables (chaque observation doit être OOB pour plusieurs arbres).
— Moins fiable avec des jeux de données très petits.

En pratique, l'erreur OOB se rapproche généralement de l'erreur obtenue par validation croisée, tout en étant beaucoup plus efficace à calculer, ce qui en fait un outil précieux pour travailler avec les forêts aléatoires.

Formule clé : probabilité qu'une observation ne soit pas sélectionnée dans un échantillon bootstrap :

$$P(\text{non sélectionnée}) = \left(1 - \frac{1}{n}\right)^n \to \frac{1}{e} \approx 0,368$$

2.7 Mesures de l'importance des variables

Comprendre quelles variables influencent les prédictions est essentiel pour l'interprétation du modèle, la sélection des caractéristiques et les insights métiers. Les forêts aléatoires offrent deux approches principales pour mesurer l'importance des variables.

2.7.1 Diminution moyenne de l'impureté (MDI)

Également appelée *Gini importance*, cette méthode est la valeur par défaut dans la plupart des implémentations. L'idée est simple : les variables importantes sont utilisées pour des divisions qui réduisent significativement l'impureté.

1. Pour chaque nœud de chaque arbre où cette variable est utilisée pour la division, mesurer la diminution d'impureté pondérée.
2. Somme des diminutions sur tous les arbres.
3. Normalisation par le nombre d'arbres.

Mathématiquement, pour une variable j :

$$\text{Importance}(j) = \sum_k \sum_{t \in T_k} I(v_t = j) \times \left(\frac{n_t}{N}\right) \times \Delta I_t$$

où :
— T_k représente tous les nœuds de l'arbre k,

— $I(v_t = j)$ est une fonction indicatrice (1 si le nœud t se divise sur la variable j, 0 sinon),
— n_t est le nombre d'échantillons au nœud t,
— N est le nombre total d'échantillons,
— ΔI_t est la diminution d'impureté au nœud t : $I_t - \frac{n_{\text{left}}}{n_t} I_{\text{left}} - \frac{n_{\text{right}}}{n_t} I_{\text{right}}$.

Avantages de MDI :
— Rapide à calculer (effectué pendant l'entraînement)
— Stable (moyenne sur de nombreux arbres)
— Pas besoin de recalculer des prédictions

Inconvénients de MDI :
— Biais vers les variables à forte cardinalité (beaucoup de valeurs uniques)
— Biais vers les variables continues par rapport aux catégorielles
— Peut attribuer de l'importance à des variables non informatives si elles sont corrélées avec des variables informatives
— Calculé sur les données d'entraînement, peut ne pas refléter la performance sur des données de test

2.7.2 Diminution moyenne de la précision (MDA) / Importance par permutation

Cette approche mesure la baisse de précision des prédictions lorsque les valeurs d'une variable sont mélangées aléatoirement, rompant sa relation avec la cible.

1. Calculer la précision de référence (souvent avec les prédictions OOB).
2. Pour chaque variable j :
   (a) Permuter aléatoirement les valeurs de j.
   (b) Recalculer les prédictions avec les données permutées.
   (c) Mesurer la diminution de précision.
3. Moyenne de la diminution sur tous les arbres.

Mathématiquement :

$$\text{Importance}(j) = \frac{1}{K} \sum_{k=1}^{K} \left[\text{Accuracy}_k - \text{Accuracy}_k(\text{permuté } j) \right]$$

Avantages de MDA :
— Non biaisé par le type de variable
— Reflète l'importance prédictive réelle
— Peut utiliser les données de test pour une estimation plus fiable
— Plus interprétable (mesure directe de l'impact sur les prédictions)

Inconvénients de MDA :
— Coûteux en calcul (nécessite de recalculer les prédictions)
— Peut être instable avec de petits jeux de données
— Sous-estime l'importance des variables corrélées (permuter une variable corrélée peut peu affecter la précision si les autres restent)

2.7.3 Considérations pratiques

1. L'échelle compte : normaliser les importances pour qu'elles somment à 1 facilite l'interprétation.
2. Problèmes de corrélation : si des variables sont fortement corrélées, l'importance peut se répartir entre elles.
3. Relations non-linéaires : certaines interactions complexes peuvent ne pas être capturées.
4. Valider les insights : utiliser l'expertise métier pour vérifier la cohérence des résultats.

2.7.4 Exemple d'interprétation

Supposons que vous prédisez le prix des maisons avec ces importances normalisées :
— Localisation : 0,35
— Surface habitable : 0,28
— Nombre de chambres : 0,15
— Âge de la maison : 0,12
— Taille du garage : 0,10

Cela indique que la localisation et la surface sont les facteurs principaux (63% de l'importance totale), tandis que les autres caractéristiques contribuent de manière modeste. Les efforts de collecte de données peuvent donc se concentrer sur ces variables clés.

Idée clé : MDI est rapide mais biaisé, MDA est non biaisé mais coûteux en calcul.

2.8 Avantages des Forêts Aléatoires

Les forêts aléatoires sont devenues l'un des algorithmes de machine learning les plus populaires, et ce pour de bonnes raisons. Elles offrent une combinaison convaincante de performance, de robustesse et de simplicité d'utilisation, ce qui les rend idéales pour de nombreuses applications réelles.

2.8.1 Robustesse au surapprentissage

Contrairement aux arbres de décision simples, qui mémorisent facilement les données d'entraînement, les forêts aléatoires résistent au surapprentissage grâce à la moyenne des prédictions de l'ensemble d'arbres. La combinaison de l'échantillonnage bootstrap et de la sélection aléatoire des caractéristiques permet que les erreurs individuelles des arbres aient tendance à s'annuler plutôt qu'à se cumuler. Dans la pratique, il est souvent possible d'ajouter davantage d'arbres sans dégrader la performance sur les données de test, bien que les gains deviennent décroissants.

2.8.2 Prétraitement minimal des données

Les forêts aléatoires gèrent remarquablement bien les données brutes :
— Pas besoin de normalisation des caractéristiques : les arbres décident selon les points de séparation, pas selon les distances.

FIGURE 2.4 – Overfitting vs Underfitting

— Gestion des types de données mixtes : elles traitent naturellement les variables numériques et catégorielles.
— Pas besoin d'ingénierie de caractéristiques : elles peuvent découvrir automatiquement des interactions complexes.
— Robustesse aux valeurs aberrantes : les divisions sont basées sur des valeurs ordonnées, donc les extrêmes ont un impact limité.

Cela rend les forêts aléatoires particulièrement utiles lorsque l'on souhaite obtenir rapidement des insights sans nettoyage de données approfondi.

2.8.3 Gestion des valeurs manquantes

Selon les implémentations, les forêts aléatoires peuvent traiter les données manquantes de plusieurs manières :

— *Surrogate splits* : utiliser d'autres variables pour approximer la division lorsque la variable principale est manquante.
— Imputation par moyenne ou mode : remplir les valeurs manquantes par la moyenne ou le mode.
— Catégorie séparée : traiter les valeurs manquantes comme une valeur distincte.
— Proximité OOB : imputer en fonction d'observations similaires.

2.8.4 Non-linéarité naturelle

Les arbres capturent automatiquement les relations non linéaires et les interactions sans spécification explicite. Si l'âge et le revenu interagissent de manière complexe, les forêts aléatoires le détecteront grâce à la structure des arbres, sans avoir besoin de créer manuellement des termes d'interaction âge×revenu.

2.8.5 Parallélisation et scalabilité

Chaque arbre s'entraîne indépendamment, ce qui rend les forêts aléatoires facilement parallélisables. Les implémentations modernes exploitent plusieurs cœurs CPU, réduisant ainsi considérablement le temps d'entraînement. Cette scalabilité s'étend également aux très grands jeux de données contenant des millions d'observations.

2.8.6 Validation intégrée

L'erreur *Out-of-Bag* (OOB) fournit une validation croisée automatique sans jeu de validation séparé, économisant à la fois des données et du temps de calcul. Cela est

particulièrement précieux lorsque les données sont limitées.

2.8.7 Avantages supplémentaires

— Importance des variables : contrairement aux réseaux de neurones *black-box*, les forêts aléatoires fournissent des scores d'importance interprétables.
— Estimations de probabilité : en classification, elles fournissent naturellement la probabilité d'appartenance à chaque classe via la proportion d'arbres votant pour chaque classe.
— Gestion des classes déséquilibrées : avec les bons paramètres, elles traitent mieux les classes déséquilibrées que de nombreux autres algorithmes.
— Performances *out-of-the-box* : elles fonctionnent souvent très bien avec les hyperparamètres par défaut.
— Polyvalence : elles excellent dans diverses tâches : classification, régression, sélection de variables, détection d'outliers, clustering et apprentissage semi-supervisé.

Cette polyvalence, combinée à la robustesse et à la simplicité d'utilisation, explique pourquoi les forêts aléatoires restent un algorithme de référence près de 25 ans après leur introduction. Elles offrent une combinaison rare de puissance et de simplicité difficile à surpasser.

Avantage clé : Les forêts aléatoires fonctionnent bien avec un prétraitement minimal et fournissent une validation intégrée via l'erreur OOB.

2.9 Limitations et Défis

Malgré leurs nombreux atouts, les forêts aléatoires présentent des limitations importantes qu'il est essentiel de connaître avant de les déployer en production.

2.9.1 Difficultés d'interprétation

Alors que les arbres de décision individuels sont très interprétables, les forêts aléatoires sacrifient cette transparence au profit de la précision. Comprendre pourquoi une forêt aléatoire a produit une prédiction spécifique est difficile :

— Impossible de visualiser facilement 500 arbres.
— L'importance des variables aide, mais n'explique pas les prédictions individuelles.
— Le chemin de décision pour une seule prédiction implique des centaines de nœuds.
— Expliquer le modèle à des parties prenantes devient complexe.

Pour les applications nécessitant la confiance humaine ou la conformité réglementaire (diagnostic médical, approbation de prêts), cette opacité peut poser problème.

2.9.2 Besoins en mémoire

Les forêts aléatoires stockent chaque arbre en mémoire, et chaque arbre peut être volumineux :

— Un arbre peut contenir des milliers de nœuds.
— Chaque nœud stocke le critère de division, le seuil et la prédiction.
— Avec 500 à 1000 arbres, la consommation mémoire augmente considérablement.

Pour un jeu de données de 1 million d'échantillons et 100 variables, une forêt aléatoire peut nécessiter plusieurs gigaoctets de RAM, ce qui devient problématique pour des déploiements mobiles ou embarqués.

2.9.3 Vitesse de prédiction

Bien que l'entraînement soit facilement parallélisable, la prédiction nécessite généralement de parcourir tous les arbres séquentiellement :

— Chaque prédiction traverse des centaines d'arbres.
— Plus lente que les modèles linéaires ou les arbres uniques.
— Peut poser problème pour des applications en temps réel (trading haute fréquence, publicité en ligne).

Pour une forêt de 1000 arbres, prédire sur de grands jeux de données peut être long.

2.9.4 Performance sur données hautement dimensionnelles et clairsemées

Les forêts aléatoires peuvent être moins performantes sur des données très hautes dimensionnelles et clairsemées, fréquentes dans :

— Classification de texte (milliers de mots, majoritairement zéros)
— Génomique (millions de marqueurs génétiques)
— Systèmes de recommandation (millions d'utilisateurs/objets)

La sélection aléatoire des variables devient moins efficace lorsque la plupart des variables sont non pertinentes, et les modèles basés sur les arbres gèrent moins bien la rareté que les modèles linéaires ou les réseaux de neurones.

2.9.5 Limites d'extrapolation

Ce problème est particulièrement important en régression. Les forêts aléatoires ne peuvent pas prédire des valeurs en dehors de l'intervalle observé dans les données d'entraînement :

— Si les cibles d'entraînement vont de 10 à 100, les prédictions resteront dans cet intervalle.
— Les arbres divisent les données en régions et font la moyenne des valeurs dans chaque région.
— Aucune région ne peut générer une valeur hors de l'intervalle de l'entraînement.

Exemple : prédire le prix des maisons lorsque les données d'entraînement varient de 100k$ à 500k$. Le modèle ne pourra jamais prédire 600k$.

2.9.6 Autres limitations

— Importance des variables biaisée : MDI favorise les variables continues ou à forte cardinalité.
— Manque de lissage : les prédictions sont par morceaux constants (régression) ou votes majoritaires (classification).
— Difficulté avec certaines relations : peut avoir du mal avec des relations linéaires, des événements rares et des dépendances temporelles.

— Réglage des hyperparamètres peu intuitif : comprendre pourquoi certains paramètres aident est moins clair que pour d'autres algorithmes.
— Pas toujours le meilleur choix : souvent surpassé par le gradient boosting sur données structurées, le deep learning sur données complexes, et les modèles linéaires sur données clairsemées.

Comprendre ces limitations permet de décider judicieusement quand utiliser les forêts aléatoires et quand d'autres méthodes peuvent mieux convenir.

Limitation clé : Les forêts aléatoires ne peuvent pas extrapoler en dehors de l'intervalle des données d'entraînement pour les tâches de régression.

Conclusion

Les forêts aléatoires constituent un algorithme de machine learning puissant et polyvalent. Grâce à la combinaison de l'échantillonnage bootstrap et de la sélection aléatoire de variables, elles offrent une grande robustesse au surapprentissage, une capacité naturelle à gérer les relations non linéaires, et une validation intégrée via l'erreur *Out-of-Bag*. Bien qu'elles nécessitent plus de mémoire et soient moins interprétables que les arbres simples, leur performance élevée, leur résistance aux valeurs aberrantes et leur aptitude à traiter différents types de données en font un outil incontournable pour la classification, la régression et d'autres applications analytiques. Les forêts aléatoires illustrent parfaitement la puissance des méthodes d'ensemble dans l'apprentissage automatique.

Chapitre 3

XGBoost (eXtreme Gradient Boosting)

3.1 Introduction

XGBoost, acronyme de *eXtreme Gradient Boosting*, représente une avancée majeure dans le domaine des algorithmes d'apprentissage automatique supervisé. Il s'agit d'un algorithme de gradient boosting hautement optimisé et scalable, positionné comme une référence pour les tâches de classification et de régression sur données tabulaires. Introduit par Tianqi Chen et Carlos Guestrin dans leur article séminal de 2016 Chen and Guestrin (2016), XGBoost a rapidement conquis la communauté scientifique et industrielle grâce à sa combinaison unique de précision, de vitesse et de robustesse.

Son succès est attesté par ses victoires répétées dans les compétitions de data science sur Kaggle, où il a souvent dominé les classements, ainsi que par son adoption massive dans des secteurs variés tels que la finance (détection de fraude), la santé (prédiction de maladies), l'e-commerce (recommandations) et les transports (prévision de trafic). Ces applications réelles soulignent sa capacité à gérer des volumes de données massifs avec une efficacité remarquable.

Par rapport à Random Forest, qui repose sur le *bagging* (construction parallèle d'arbres indépendants pour minimiser la variance), XGBoost utilise le *boosting* : une méthode séquentielle où chaque arbre corrige les erreurs des précédents, réduisant ainsi le biais global du modèle.

Ce chapitre vise à introduire les principes fondamentaux du gradient boosting, à détailler les innovations mathématiques et algorithmiques propres à XGBoost, à explorer ses fonctionnalités avancées et optimisations systèmes, et à discuter de ses avantages, limites et comparaisons avec d'autres approches, afin de fournir une compréhension complète et pratique de cet algorithme essentiel.

3.2 Principe du Gradient Boosting

3.2.1 Intuition : apprentissage par correction d'erreurs

Le principe clé du gradient boosting est l'apprentissage itératif. Contrairement à la construction parallèle des forêts aléatoires, le boosting construit le modèle étape par étape. Chaque nouvel arbre est entraîné non pas sur la variable cible originale, mais sur les erreurs résiduelles des arbres précédents.

Cette méthode peut être vue comme une descente de gradient dans l'espace fonctionnel. Si l'on considère le modèle global comme un point dans l'espace des fonctions possibles, chaque nouvel arbre ajouté représente un pas dans la direction opposée du gradient de la fonction de perte, minimisant ainsi l'erreur globale de manière incrémentale.

3.2.2 Formulation mathématique du gradient boosting

L'objectif est de trouver une fonction F qui minimise la perte globale sur un ensemble de n observations :

$$\min_F \sum_{i=1}^{n} L(y_i, F(x_i)) \tag{3.1}$$

où L est une fonction de perte convexe et différentiable.

Le modèle est construit de manière additive. À l'étape m, la prédiction $F_m(x)$ est mise à jour selon :

$$F_m(x) = F_{m-1}(x) + \eta h_m(x) \tag{3.2}$$

Ici, $h_m(x)$ est le nouvel arbre ajouté et η (taux d'apprentissage) contrôle l'amplitude de la mise à jour pour éviter le surapprentissage.

Le nouvel arbre h_m est entraîné pour approximer le gradient négatif de la perte, aussi appelé pseudo-résidu g_i pour chaque observation i :

$$g_i = - \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} \tag{3.3}$$

Différentes fonctions de perte peuvent être utilisées selon le problème : erreur quadratique (L2) pour la régression, *log-loss* pour la classification binaire, ou erreur absolue (L1) pour une plus grande robustesse.

3.2.3 Limitations du gradient boosting classique

Le Gradient Boosting Machine (GBM) traditionnel souffre de certaines limitations :

- **Absence de régularisation explicite :** Le risque de surapprentissage est élevé si le nombre d'arbres est trop grand ou leur profondeur trop importante.
- **Optimisation de premier ordre uniquement :** L'utilisation seule du gradient limite la précision de l'approximation de la perte.
- **Gestion sous-optimale des valeurs manquantes :** Nécessite souvent un prétraitement coûteux.
- **Performances limitées sur grandes données :** L'implémentation standard manque d'optimisations systèmes pour passer à l'échelle.

3.3 XGBoost : Innovations Mathématiques

XGBoost adresse ces limitations par une formulation mathématique régularisée et une approximation de second ordre.

3.3.1 Fonction objectif régularisée

La fonction objectif à minimiser à chaque étape t inclut désormais un terme de régularisation Ω :

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{3.4}$$

Le terme de régularisation pour un arbre f dépanne de sa structure et de ses poids :

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 + \alpha \sum_{j=1}^{T} |w_j| \tag{3.5}$$

Chaque terme a une signification précise :
- γ (gamma) : pénalité sur le nombre de feuilles T, contrôlant la complexité structurelle de l'arbre.
- λ (lambda) : régularisation L2 sur les poids des feuilles w_j, favorisant le lissage (analogue à Ridge).
- α (alpha) : régularisation L1 sur les poids, favorisant la parcimonie (analogue à Lasso).

3.3.2 Approximation de Taylor de second ordre

Pour optimiser cette fonction objectif, XGBoost utilise un développement de Taylor de second ordre autour de la prédiction précédente $\hat{y}^{(t-1)}$:

$$L(y_i, \hat{y}^{(t-1)} + f_t(x_i)) \approx L(y_i, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \tag{3.6}$$

où g_i et h_i sont définis comme :

$$g_i = \partial_{\hat{y}^{(t-1)}} L(y_i, \hat{y}^{(t-1)}), \quad h_i = \partial^2_{\hat{y}^{(t-1)}} L(y_i, \hat{y}^{(t-1)}) \tag{3.7}$$

L'ajout de l'information de courbure (via la Hessienne h_i) permet une convergence plus rapide et une meilleure stabilité numérique par rapport à l'approche de premier ordre utilisée dans le GBM classique.

3.3.3 Calcul du poids optimal des feuilles

En regroupant les instances par feuille j (ensemble $I_j = \{i | q(x_i) = j\}$), on peut réécrire l'objectif approximé et trouver analytiquement le poids optimal w_j^* :

$$\tilde{\mathcal{L}}^{(t)} = \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \tag{3.8}$$

La solution qui minimise cette équation quadratique est :

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{3.9}$$

3.3.4 Score de qualité d'une structure d'arbre

En réinjectant w_j^* dans l'objectif, on obtient un score de structure permettant d'évaluer la qualité d'un arbre. Cela permet de définir le gain obtenu lors d'une division d'une feuille en deux (gauche L et droite R) :

$$\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma \tag{3.10}$$

où $G = \sum g_i$ et $H = \sum h_i$. Cette formule sert de critère pour décider s'il faut effectuer un split. Le terme γ agit comme un seuil minimal : si le gain calculé est inférieur à γ, la branche n'est pas créée. Cela réalise un élagage (*pruning*) automatique pendant la construction de l'arbre.

3.4 Fonctionnalités Spécifiques de XGBoost

3.4.1 Gestion native des valeurs manquantes

XGBoost utilise un algorithme de *Sparsity-aware Split Finding*. Lors de la recherche du meilleur split, il teste automatiquement d'envoyer les valeurs manquantes vers la branche gauche ou droite et choisit la direction qui maximise le gain. Cela permet de gérer les données incomplètes nativement, sans imputation préalable.

3.4.2 Gestion du déséquilibre de classes

Pour les jeux de données déséquilibrés, le paramètre `scale_pos_weight` permet de rééquilibrer l'importance des classes :

$$\text{scale\_pos\_weight} = \frac{n_{\text{classe majoritaire}}}{n_{\text{classe minoritaire}}} \tag{3.11}$$

Ce poids est appliqué aux gradients et hessiennes des exemples positifs, augmentant leur influence lors de l'entraînement. C'est crucial pour des applications comme la détection de fraude ou le diagnostic de maladies rares.

3.4.3 Subsampling stochastique

Pour améliorer la généralisation, XGBoost emprunte le principe de sous-échantillonnage aux Random Forests :

- `subsample` : fraction des observations tirées aléatoirement pour construire chaque arbre.
- `colsample_bytree`, `colsample_bylevel`, `colsample_bynode` : fraction des colonnes (features) utilisées à différents niveaux de construction.

Cela permet de décorréler les arbres et d'ajouter une couche supplémentaire de régularisation.

3.4.4 Early stopping

L'entraînement peut être configuré pour s'arrêter prématurément si la performance sur un jeu de validation ne s'améliore pas après un certain nombre d'itérations (*patience*). Cela détermine automatiquement le nombre optimal d'arbres et prévient le surapprentissage tardif.

3.5 Optimisations Systèmes

3.5.1 Parallélisation

Bien que le boosting soit séquentiel par nature (l'arbre t dépend de l'arbre $t-1$), XGBoost parallélise la construction à l'intérieur de chaque arbre. La recherche du meilleur split sur toutes les features est effectuée en parallèle grâce à une structure de données en colonnes (*Column Block*).

3.5.2 Scalabilité

XGBoost implémente plusieurs techniques pour gérer les données volumineuses :
- **Histogram-based splits :** Les données continues sont discrétisées en *bins*, réduisant la complexité de recherche de split.
- **Out-of-core computing :** Optimisation des accès disques pour traiter des données ne tenant pas en mémoire vive.
- **Weighted Quantile Sketch :** Un algorithme d'approximation distribué pour trouver les quantiles sur des données pondérées, facilitant la création d'histogrammes précis.

3.6 Hyperparamètres Clés

3.6.1 Contrôle de la complexité

- `max_depth` : Profondeur maximale de l'arbre (typiquement entre 3 et 10).
- `min_child_weight` : Somme minimale des poids (Hessienne) requise dans une feuille pour autoriser une partition.
- `gamma` : Réduction de perte minimale requise pour faire un split.

3.6.2 Régularisation et Apprentissage

- `lambda` (λ) et `alpha` (α) : Termes de régularisation L2 et L1.
- `learning_rate` (η) : Facteur de réduction de l'impact de chaque nouvel arbre.

3.6.3 Sampling et Autres

- `subsample` et `colsample_bytree` : Contrôle du caractère stochastique.
- `scale_pos_weight` : Gestion du déséquilibre.
- `n_estimators` associé à `early_stopping_rounds`.

3.7 Avantages et Limites

3.7.1 Avantages

XGBoost offre une combinaison unique de performance (état de l'art sur données tabulaires), de robustesse (régularisation intégrée, gestion valeurs manquantes) et de ra-

pidité (optimisations systèmes). Sa flexibilité permet de définir des fonctions de perte et métriques personnalisées.

3.7.2 Limites

Cependant, il reste une "boîte noire" nécessitant des outils d'explicabilité comme SHAP. Son réglage est complexe avec plus de 20 hyperparamètres, et il peut être sensible au bruit. Sa nature séquentielle peut être un frein par rapport à la parallélisation totale des Random Forests lors de l'entraînement.

3.7.3 Comparaison : XGBoost vs Random Forest

Critère	**XGBoost**	**Random Forest**
Performance (Données Tabulaires)	Excellente	Très Bonne
Vitesse d'entraînement	Moyenne	Très Rapide
Interprétabilité	Faible	Moyenne
Facilité d'usage	Moyenne	Très Facile
Résistance au surapprentissage	Très Bonne	Excellente

TABLE 3.1 – Comparaison synthétique : XGBoost vs Random Forest

3.8 Conclusion

XGBoost représente l'état de l'art du gradient boosting grâce à ses trois piliers : la précision (via l'optimisation de second ordre régularisée), l'efficacité (parallélisation et gestion mémoire), et la praticité (fonctionnalités natives). C'est le choix optimal lorsque la performance prédictive est le critère critique, surpassant souvent les méthodes classiques malgré une complexité de mise en œuvre plus élevée. Bien que des successeurs comme LightGBM et CatBoost existent, XGBoost demeure une référence incontournable.

Chapitre 4

Commandes utiles

4.1 Quelques commandes

Voici quelques commandes utiles : Mittelbach et al. (2004)

4.1.1 Commande pour insérer et citer une image centralisée

FIGURE 4.1 – Légende de la figure

Ici, je cite l'image 4.1

4.1.2 Commande pour insérer et citer une équation

$$\rho + \Delta = 42 \tag{4.1}$$

L'équation 4.1 est cité ici. Knuth (1984)

Pour écrire des variables dans le texte, il suffit de mettre le symbole $ entre le texte souhaité comme : constante ρ.

4.1.3 Commande pour ecrire un code

```c
#ifndef DECLARATION_H_INCLUDED
#define DECLARATION_H_INCLUDED

//declaration de structure qui englobe les donnees d'un adherant
typedef struct _donnee
{
    char code[20];
    char nom[20];
    char domaineInteret[20];
}donnees;

#endif // DECLARATION_H_INCLUDED
```

Listing 1 – exemple de caption de code C

One-line code formatting also works with `minted`. For example, a small fragment of HTML like this :

```
<h2>Something <b>here</b></h2>
```

can be formatted correctly.

LaTeX Lamport (1994) is a set of macros built atop TeX Knuth (1986).

Chapitre 5

Quatrième Chapitre

Chapitre 6

Cinquième Chapitre

Conclusion Générale

Bibliographie

Bertsekas, D. (2009). *Convex optimization theory*, volume 1. Athena Scientific.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152.

Chen, T. and Guestrin, C. (2016). Xgboost : A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794.

Devroye, L., Györfi, L., and Lugosi, G. (1996). Vapnik-chervonenkis theory. In *A probabilistic theory of pattern recognition*, pages 187–213. Springer.

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2) :97–111.

Knuth, D. E. (1986). *The TEX Book*. Addison-Wesley Professional.

Lamport, L. (1994). *LATEX : a Document Preparation System*. Addison Wesley, Massachusetts, 2 edition.

Mittelbach, F., Gossens, M., Braams, J., Carlisle, D., and Rowley, C. (2004). *The LATEX Companion*. Addison-Wesley Professional, 2 edition.

Annexes