UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

Travail d'étude et de recherche
réalisé dans le cadre du
Master 1 MAEF

RAPPORT DE TER

Boosting en apprentissage
statistique et **XGBoost**

Mohamed-Ayoub Bouzid
Alexis Sorre

Master 1 mathématiques appliquées
à l'économie et à la finance
à l'Université Paris 1 Panthéon Sorbonne

Encadrant : M. Jean-Marc Bardet
Février/Mai 2025

Table des matières

1 Introduction

L'apprentissage statistique occupe une place centrale dans de nombreux domaines, allant de la médecine à la finance, en passant par l'industrie et les sciences sociales. Il s'agit d'un ensemble de méthodes permettant d'extraire des connaissances à partir de données, afin de faire des prédictions, de classer des observations, ou encore de détecter des relations cachées entre différentes variables.
Les situations dans lesquelles ces techniques sont mobilisées sont extrêmement variées : estimer l'évolution future d'un cours boursier à partir d'indicateurs économiques, ou encore reconnaître automatiquement des chiffres manuscrits dans une image.

Dans tous ces cas, on dispose d'un ensemble d'observations appelées données d'apprentissage, contenant à la fois des résultats (variables à expliquer « targets ») et des caractéristiques associées (les variables explicatives « features »). Le but est alors de construire un modèle, ou apprenant (« learner »), capable de généraliser à de nouvelles situations en apprenant les régularités présentes dans ces données.

C'est précisément ce que l'on appelle l'apprentissage supervisé, une forme d'apprentissage statistique dans laquelle la variable cible guide l'entraînement du modèle.

Un problème d'apprentissage est dit supervisé lorsque les données d'apprentissage comprennent, pour chaque observation, les valeurs des variables explicatives ainsi que la valeur du résultat à prédire. L'apprentissage est donc guidé par cette variable cible, et le modèle apprend à établir un lien entre les entrées et la sortie attendue. Un exemple classique de problème d'apprentissage supervisé est la détection de courriels indésirables. Dans ce cas, chaque email représente une observation, et l'objectif est de prédire s'il s'agit d'un spam ou d'un message légitime. Pour cela, on dispose d'un ensemble d'emails préalablement étiquetés (on sait pour chacun s'il est un spam ou non (targets)), ainsi que dans notre cas la présence ou non de certains mots clés.

	george	you	your	hp	free	hpl	!	our	re	edu	remove
spam	0.00	2.26	1.38	0.02	0.52	0.01	0.51	0.51	0.13	0.01	0.28
email	1.27	1.27	0.44	0.90	0.07	0.43	0.11	0.18	0.42	0.29	0.01

FIGURE 1 – Ce tableau montre le pourcentage moyen d'apparition de certains mots ou caractères dans des messages spam par rapport à des emails normaux (non-spam). Les mots et caractères affichés sont ceux qui montrent la plus grande différence entre les deux types d'email

En analysant le tableau, on voit que le mot you apparaît en moyenne 2.26 fois dans un spam contre 1.27 dans un email.
Ainsi, notre méthode d'apprentissage doit décider de quel caractère utiliser et comment. Par exemple, on a :

```
if (%george < 0.4) & (%you > 1.5) then spam
else email
```

Selon la nature de la variable à prédire, on distingue deux grandes catégories de tâches supervisées :

1. La classification : Prédire une étiquette ou une classe parmi un ensemble fini (par exemple : spam / non spam, malade / sain).

2. La régression : Estimer une valeur numérique continue (comme le prix d'un bien, une température, ou une concentration chimique). Dans les deux cas, l'objectif est de développer un modèle prédictif performant, capable d'exploiter les régularités contenues dans les données passées pour effectuer des prédictions fiables sur de nouvelles données.

Une fois le cadre de l'apprentissage supervisé posé, il est essentiel de distinguer les différents types de modèles que l'on peut mobiliser pour cette tâche. On distingue notamment deux grandes approches : les modèles simples et les modèles d'ensemble.

Un modèle simple repose sur un unique algorithme d'apprentissage, appliqué directement aux données d'entraînement. C'est le cas, par exemple, de la régression linéaire, des arbres de décision. Ces modèles sont généralement faciles à mettre en œuvre, rapides à entraîner et souvent interprétables, ce qui en fait des outils privilégiés pour une première exploration des données ou pour des problèmes dont la structure reste relativement simple. Toutefois, leur capacité à capturer des

relations complexes ou des interactions non linéaires entre les variables peut être limitée, ce qui peut freiner leur performance sur certains jeux de données plus riches ou plus bruités. À l'inverse, les modèles d'ensemble reposent sur la combinaison de plusieurs modèles de base, souvent appelés apprenants faibles dans le but d'améliorer la robustesse des prédictions. L'idée est que, même si chaque modèle pris isolément est imparfait, leur agrégation permet de réduire les erreurs, de limiter le sur-apprentissage, et d'améliorer la généralisation sur de nouvelles données. Parmi les principales méthodes d'ensemble, on peut citer :

1. Le bagging, illustré notamment par les forêts aléatoires (**Random Forests**), qui consiste à entraîner plusieurs modèles sur des sous-échantillons aléatoires des données, puis à agréger leurs prédictions.

2. Le boosting, utilisé dans des algorithmes tels que **AdaBoost**, **Gradient Boosting** ou **XGBoost**, qui corrige progressivement les erreurs des modèles précédents en les entraînant de manière séquentielle.

3. Le stacking, qui combine les prédictions de plusieurs modèles hétérogènes à l'aide d'un méta-modèle chargé d'apprendre comment les combiner de façon optimale.

Les modèles d'ensemble sont aujourd'hui largement utilisés en raison de leur excellente performance prédictive. Leur principal inconvénient réside toutefois dans leur complexité : ils sont généralement plus longs à entraîner, moins interprétables, et nécessitent une infrastructure de calcul plus robuste.

2 Le modèle CART

2.1 Arbres de décision

Avant de commencer à nous intéresser au modèle **CART (Classification and Regression Trees)**, il est essentiel de commencer par présenter ce que sont les arbres de décision.

Un arbre de décision est une méthode d'apprentissage supervisé dont l'idée principale est de partitionner l'espace des variables explicatives en plusieurs sous espaces et ajuster un modèle simple dans chacun d'eux (cf. figure 2). Par exemple, dans le cas d'une régression,on considère Y la variable endogène, X_1 et X_2 les variables explicatives et R_1, R_2, R_3 les sous espaces de la partition. Si $(X_1, X_2) \in R_n$ avec $n = 1, 2, 3$ alors on prédit Y par une constante c_m qui peut par exemple être la moyenne des Y dans cette région. Le modèle de régression associé sera alors de la forme :

$$\hat{f}(X) = \sum_{n=1}^{3} c_m \, \mathbf{1}_{\{(X_1,X_2)\in R_m\}}.$$

Le partitionnement de cet espace peut être plus ou moins difficile à décrire toutefois dans le cadre de notre étude, nous nous limiterons aux partitions binaires récursives utilisés pour le modèle CART.

Ce type de partitionnement peut être représenté par un arbre binaire où à chaque noeud, nous avons deux branches, les données sont dirigées vers l'une des deux branches en fonction de la règle de décision choisie à chaque noeud. Chaque chemin de l'arbre résulte en une feuille donnant une prédiction finale.

Le principal avantage d'utiliser d'utiliser les arbres binaires récursifs est leur interprétabilité. En effet, on peut décrire la partition sous la forme d'un seul arbre tandis que lorsqu'on a plus de variables d'entrées la représentation peut être bien plus difficile.

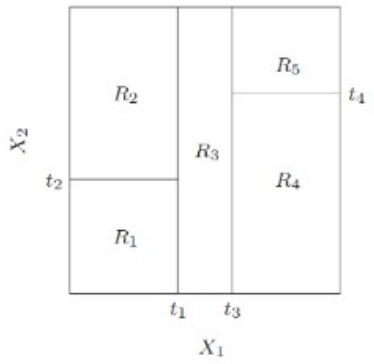

FIGURE 2 – Partitionnement de l'espace des variables explicatives constitué de X_1 et X_2

2.2 Présentation du modèle CART

Le modèle **CART** est une méthode d'apprentissage supervisé utilisée pour des tâches de classification et de régression. Il permet d'expliquer comment une variable peut être expliquée en fonction d'autres facteurs. Nous allons présenter ici les deux catégories d'arbres de décision à savoir les arbres de régression et de classification.

2.2.1 Arbres de régression

On va considérer dans cette partie Y une variable à prédire, p variables endogènes $X_1, ..., X_p$ et N observations (x_i, y_i) avec $i = 1, ..., N$. Une première méthode que l'on peut envisager pour trouver la meilleure prédiction de Y possible est de choisir la partition qui minimise le risque empirique.
Si on suppose que l'espace des variables se décompose en 2 partitions R_1, R_2 pour simplifier et que la prédiction de Y est c_m dans la région m avec $m \in \{1, 2\}$. Le modèle est donc de la forme :

$$f(x) = c_1 \, \mathbb{1}_{\{x\in R_1\}} + c_2 \, \mathbb{1}_{\{x\in R_2\}}$$

Nous voulons alors minimiser :

$$\text{SSE}(R_1, R_2) = \sum_{x_i \in R_1} \big(y_i - f(x_i)\big)^2 + \sum_{x_i \in R_2} \big(y_i - f(x_i)\big)^2.$$

D'autre part, nous déterminons les estimateurs $\hat{c}_m$ des c_m avec $m = 1, 2$ en minimisant :

$$\sum_{x_i \in R_m} (y_i - f(x_i))^2$$

On considère que $\forall x_i \in R_m$ on a $f(x_i) = c$. On a donc $Q_m(c) = \sum_{x_i \in R_m} (y_i - c)^2$. On va trouver le minimum en résolvant :

$$\frac{dQ_m(c)}{dc} = 0$$

En développant, on a :

$$Q_m(c) = \sum_{x_i \in R_m} (y_i - c)^2 = \sum_{x_i \in R_m} (y_i^2 - 2y_i c + c^2) = \sum_{x_i \in R_m} y_i^2 + 2c \sum_{x_i \in R_m} y_i + c^2 N_m$$

où $N_m = card(x_i \in R_m)$. Donc en dérivant l'expression on obtient :

$$\frac{dQ_m(c)}{dc} = -2 \sum_{x_i \in R_m} y_i + 2cN_m = 0$$

puis en isolant c, on trouve finalement que la valeur optimale de c est la moyenne des y_i dans la région R_m :

$$\boxed{\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i} \tag{1}$$

On vérifie qu'il s'agit bien d'un minimum en montrant que Q_m est bien convexe :

$$\frac{d^2 Q_m(c)}{dc^2} = 2N_n > 0.$$

Cependant, il apparaît que cette méthode est irréalisable car trop coûteuse en calculs. En effet, elle implique de tester toutes les façons possibles de découper l'espace i.e 2^N possibilité dans le cas où on veut découper l'espace en 2 partitions.

Détermination des variables et seuils de séparation :
Une solution à ce problème réside dans l'utilisation du modèle **CART**. L'idée derrière cet algorithme est que l'on cherche à prédire Y en déterminant des variables de séparation X_j, les points de séparation et la forme que l'arbre doit avoir. A chaque étape, l'algorithme choisit la variable et le point de séparation qui minimisent un critère d'erreur. Ce processus est répété jusqu'à atteindre un critère d'arrêt.

On considère donc deux demi plans :

$$R_1(j, s) = \{X \mid X_j < s\} \quad \text{et} \quad R_2(j, s) = \{X \mid X_j > s\}. \tag{2}$$

Pour une étape donnée, on va chercher la variable de séparation j et le point de séparation s qui résolvent :

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right] \tag{3}$$

Commençons par minimiser dans un premier temps la fonction :

$$L(c_1) = \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2$$

Cette fonction étant convexe, on obtient en minimum on trouve un minimum en résolvant l'équation :

$$\frac{d}{dc_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 = 0$$

Après développement puis dérivation, on obtient :

$$\sum_{x_i \in R_1(j,s)} y_i + 2c_1 \sum_{x_i \in R_1(j,s)} 1 = 0$$

Finalement, on obtient :

$$c_1 = \frac{1}{|R_1|} \sum_{x_i \in R_1(j,s)} y_i \tag{4}$$

Ce qui correspond à la moyenne des y_i dans R_1 :

$$\hat{c}_1 = \text{moyenne}(y_i | x_i \in R_1(j,s)) \tag{5}$$

On va également minimiser :

$$L(c_2) = \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2$$

Par un raisonnement analogue, on trouve également :

$$\hat{c}_2 = \text{moyenne}(y_i | x_i \in R_2(j,s)) \tag{6}$$

Ces valeurs $\hat{c}_1$ et $\hat{c}_2$ sont calculés pour chaque couple (j,s) puis on évalue l'erreur quadratique :

$$\sum_{x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2$$

On choisit le couple (j,s) qui donne l'erreur quadratique la plus petite qui seront utilisés pour réaliser la première division de l'arbre. Le processus est ensuite répété sur toutes les régions résultantes.

Détermination de la taille optimale de l'arbre :

Le deuxième élément cruciale dans la construction d'un arbre de régression est la détermination de sa taille. En effet, une fois que l'arbre qu'on notera T_0 a été construit à l'aide de la méthode précédemment décrit, il se peut que cet arbre soit trop grand ce qui aura comme conséquence un surajustement du modèle : l'arbre sera très performant sur les données d'entrainement mais mal généralisable aux nouvelles données car trop spécifique. Pour remédier à cela, nous allons tenter d'obtenir une sorte d'optimalité dans la taille de l'arbre.

Une première idée serait de diviser l'arbre en un noeud que si la somme des carrés des résidus en ce noeud ne dépasse pas un certain seuil que nous fixons. Cependant, cette approche ne semble pas pertinente car il se peut que pour un noeud donné la somme des carrés des résidus dépasse ce seuil mais que en continuant de diviser on obtient des prédictions rendant cette somme des carrés des résidus.

Nous allons au lieu de cela élaguer l'arbre avec la méthode "élagage par complexité-coût". On considère $T \subset T_0$ un arbre obtenu par élagage à partir de T_0 On indexe chaque noeud terminal par m. Nous définissons, $|T|$ comme le nombre de noeuds terminaux dans T, $N_m = \text{card}(x_i \in R_m)$ et $\hat{c}_m$ comme dans l'équation. De plus, l'erreur quadratique moyenne sur R_m est donnée par :

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2 \tag{7}$$

Le **critère de complexité-coût** est défini par :

$$\boxed{C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|} \tag{8}$$

Pour une valeur de $\alpha \geq 0$, l'arbre optimal T_α sera celui qui minimise ce critère :

$$T_\alpha = \arg\min_T C_\alpha(T) \tag{9}$$

Nous devrons choisir un α réalisant un compromis entre la taille de l'arbre et sa qualité d'ajustement. Si α est grand, les grands arbres sont pénalisés et T_α sera petit. Si α est petit, l'ajustement est privilégié et plus de noeuds sont gardés.

2.2.2 Arbres de classification

Dans de nombreuses situations, la variable à prédire Y n'est pas une variable numériques mais qualitatifs : malade ou non, client fidèle ou à risque, spam ou message légitime. Ces problèmes relèvent tous de la classification. Nous considérons que Y prend les valeurs $1, ..., K$. Ces valeurs représentent les différentes classes considérées.

Le principe est ici le même que précédemment c'est à dire que nous commençons par construire le grand arbre T_0 dont le critère d'arrêt pour sa construction par exemple si le nombre minimal d'observation dans un noeud est atteint. Nous verrons qu'un autre critère d'arrêt peut également être utilisé. Pour chaque variable d'entrée $X_1, ..., X_p$, on va tester tous les seuils de coupure possibles. Les seuils de coupures sont déterminés en commençant par trier les observations pour une variable explicative donnée et ensuite nous faisons la moyenne entre deux observations consécutives. Si on N observations pour chaque variable explicative, nous obtenons $N - 1$ seuils de coupure à tester.
Nous introduisons désormais de nouvelles notions. Nous considérons pour commencer un noeud m associé à une région R_m contenant N_m observations. Nous définissons **la proportion d'observation de classe k dans le noeud m** comme :

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} 1(y_i = k) \tag{10}$$

Un noeud sera dit impur s'il ne contient pas uniquement des observations d'une seule classe. On définit donc **la classe majoritaire dans le noeud m** comme :

$$k(m) = \arg\max_k \hat{p}_{mk} \tag{11}$$

Pour chaque noeud, l'objectif sera de mesurer l'impureté et de trouver le couple de variables de coupure et de seuil de coupure qui minimisent l'impureté globale (définie ci-dessous). Les deux mesures d'impureté communément utilisées sont l'indice Gini et l'entropie croisée. Ils sont respectivement définis par :

$$\text{Indice Gini} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{12}$$

et

$$\text{Entropie croisee} = -\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk}) \tag{13}$$

On utilise l'une de ces deux mesures pour mesurer l'impureté globale d'une division et on choisira la division qui la minimise :

$$\text{Impureté Globale} = \frac{N_L}{N} Q_L + \frac{N_R}{N} Q_R \tag{14}$$

avec N_L, N_R respectivement le nombre d'observations dans les sous-noeuds gauche et droit et N le nombre totale d'observation dans les deux sous noeuds.
On répète récursivement ce processus dans chaque sous noeud jusqu'à atteindre un critère d'arrêt. Nous en avons évoqué un ci-dessus. Il est à noter toutefois qu'un autre critère pouvant être utilisé dans le cas de classification est le gain d'impureté c'est à dire qu'on s'arrête si une division ne diminue pas de manière assez significative l'impureté globale.
Une fois cet arbre, construit nous allons l'élaguer comme précédemment c'est à dire en trouvant l'arbre T minimisant le critère complexité-coût mais cette fois $Q_m(T)$ correspondra à la mesure d'impureté dans le noeud m.

3 Théorie du Boosting

3.1 Motivation

Les arbres de décision pour la régression et la classification permettent une prédiction assez intuitives de la variable cible Y. Toutefois pris isolément, il apparaît que ces modèles de prédiction présentent de nombreuses limites. L'une des principales limitations est que ces modèles sont sensibles aux variations des données. Une modification, même minime, de l'échantillon d'apprentissage peut engendrer une structure significativement différente. Cela peut avoir pour conséquence une variance élevée des prédictions compromettant ainsi la robustesse des prédictions et limitant la capacité de généralisation du modèle.

Par ailleurs, lorsque les arbres de décision sont profonds, le nombre de partition possible explose de manière exponentielle. Pour un prédicteur ayant q valeurs possibles le nombre de partitions possibles est de 2^q pouvant provoquer des difficultés de computation un surapprentissage du modèle. Présentons maintenant une méthode permettant de travailler non pas avec un seul arbre complexe mais combiner plusieurs arbres peu profond afin d'obtenir un apprenant global plus stable et souvent plus robuste.

3.2 Classificateur faible et théorème de Schapire

Dans cette partie, nous nous intéressons à ce qui constitue la base du boosting à savoir les classificateurs faibles. Ce sont ces classificateurs qui sont combinés pour former un apprenant global plus performant. Pour montrer que cela est possible, nous allons énoncer le **théorème de Schapire** et en proposer une esquisse de preuve.

Définition 1. *Un classificateur faible est un algorithme d'apprentissage qui donne des résultats au moins meilleurs que le hasard. Cela signifie que sa précision est supérieure à* 50% *dans un problème de classification binaire. Les classificateurs faibles considérés sont souvent de petits arbres CART peu profond.*

Afin d'énoncer ce qu'est le théorème de Schapire, nous avons besoin de poser le cadre théorique nécessaire. On se place dans le cadre du modèle PAC introduit par Valiant en 1984. Dans ce cadre, un **concept** c est une fonction borélienne $c : X \to \{0, 1\}$ attribuant à chaque donnée une étiquette. De plus, une **classe de concept** C est un ensemble de ces fonctions. On peut la décomposer en sous-classe C_n de sorte que chacune de ces sous-classes agit sur un ensemble de données X_n. Par ailleurs, on considère que l'algorithme n'a pas accès directement à la fonction c mais uniquement à l'ensemble d'entraînement. De plus, l'algorithme reçoit des données $x \in X_n$ choisit aléatoirement selon une distribution inconnue D, et pour chaque donnée, la réponse correcte $c(x)$ est connue et déterminée par la fonction cible. Cela permet de traduire le fait qu'en pratique, l'algorithme apprend à partir d'un échantillon aléatoire sans connaître la loi qui régit la génération des données.
A partir des données observées, l'algorithme va produire une fonction $h : X_n \to \{0, 1\}$ appelée hypothèse. C'est cette fonction qui sert à prédire les étiquettes de nouvelles données. Nous mesurerons la qualité de cette hypothèse avec le taux d'erreur :

$$\mathrm{err}_D(h) = P(h(x) \neq c(x)) \tag{15}$$

Lorsque cette erreur est inférieur à $\epsilon > 0$, on dit que h est ϵ-proche du concept cible. Nous définissons de façon plus formelle ce que sont les apprenants forts et faibles.

Définition 2. *Une classe de concept C est dite fortement apprenable s'il existe un algorithme capable pour tout concept $c \in C$, toute distribution D, et pour tout seuil d'erreur ϵ et de confiance δ, de produire avec une probabilité d'au moins $1 - \delta$ une hypothèse h telle que :*

$$err_D(h) \leq \epsilon$$

Définition 3. *Un algorithme est dit faiblement apprenable s'il peut produire une hypothèse légèrement meilleur que le hasard :*

$$err_D(h) \leq \frac{1}{2} - \gamma$$

Enfin, le dernier concept clé dans ce cadre est celui de **l'oracle d'exemple EX**.

Définition 4. *Un oracle d'exemple pour un concept c sur un domaine X_n, associé à une distribution D, est une procédure qui à chaque appel tire une instance $x \in X_n$ selon une distribution D et retourne la paire $(x, c(x))$, i.e l'instance et son étiquette correcte donnée par le concept cible c. L'oracle d'exemple est donc ce qui génère les données d'entraînement.*

Le cadre théorique étant posé, nous pouvons maintenant énoncer le théorème de Schapire :

Théorème 1. *Toute classe de concepts qui est faiblement apprenable est également fortement apprenable. Plus précisément, si un algorithme d'apprentissage produit une hypothèse dont l'erreur est inférieure à $\frac{1}{2} - \gamma$ pour une constante $\gamma > 0$ alors il est possible de construire à partir de celui-ci, un nouvel algorithme qui produit une hypothèse avec une erreur arbitrairement petite.*

Preuve. L'idée générale de la preuve que propose Schapire est qu'il construit un algorithme appelé **Learn** qui simule un algorithme faible A trois fois sur des distributions modifiées des données d'entraînement.

Learn(ϵ, δ, EX)

Input: error parameter ϵ
confidence parameter δ
examples oracle EX
(implicit) size parameters s and n

Return: hypothesis h that is ϵ-close to the target concept c with probability $\geq 1 - \delta$

Procedure:
if $\epsilon \geq 1/2 - 1/p(n, s)$ **then return** **WeakLearn**(δ, EX)
$\alpha \leftarrow g^{-1}(\epsilon)$

$EX_1 \leftarrow EX$
$h_1 \leftarrow$ **Learn**$(\alpha, \delta/5, EX_1)$
$\tau_1 \leftarrow \epsilon/3$
let $\hat{a}_1$ be an estimate of $a_1 = \Pr_{v \in D}[h_1(v) \neq c(v)]$:
choose a sample sufficiently large that $|a_1 - \hat{a}_1| \leq \tau_1$ with probability $\geq 1 - \delta/5$
if $\hat{a}_1 \leq \epsilon - \tau_1$ **then return** h_1

defun $EX_2()$
{ flip coin
if *heads*, **return** the first instance v from EX for which $h_1(v) = c(v)$
else **return** the first instance v from EX for which $h_1(v) \neq c(v)$ }
$h_2 \leftarrow$ **Learn**$(\alpha, \delta/5, EX_2)$
$\tau_2 \leftarrow (1 - 2\alpha)\epsilon/8$
let $\hat{e}$ be an estimate of $e = \Pr_{v \in D}[h_2(v) \neq c(v)]$:
choose a sample sufficiently large that $|e - \hat{e}| \leq \tau_2$ with probability $\geq 1 - \delta/5$
if $\hat{e} \leq \epsilon - \tau_2$ **then return** h_2

defun $EX_3()$
{ **return** the first instance v from EX for which $h_1(v) \neq h_2(v)$ }
$h_3 \leftarrow$ **Learn**$(\alpha, \delta/5, EX_3)$

defun $h(v)$
{ $b_1 \leftarrow h_1(v)$, $b_2 \leftarrow h_2(v)$
if $b_1 = b_2$ **then return** b_1
else **return** $h_3(v)$ }
return h

FIGURE 3 – Algorithme Learn extrait de "The strength of weak learnability" (Schapire,1990), p.7

Nous commençons par fixer l'erreur α tel que $g(\alpha) = \epsilon$ avec $g(\alpha) = 3\alpha^2 - 2\alpha^3$. On appelle une première fois Learn$(\alpha, \frac{\delta}{5}, \text{EX}_1)$ et on considère un estimateur $\hat{a}_1$ de $a_1 = P(h_1(v) \neq c(v))$ qui soit suffisamment proche de a_1 avec un probabilité supérieur ou égale à $1 - \frac{\delta}{5}$. Si, $\hat{a}_1 \leq \epsilon$ alors on renvoie h_1 car cela signifierait que l'algorithme est un apprenant fort. Sinon, on construit une distribution

EX_2 qui donne avec 50% de chance un exemple bien classé par h_1 et 50% un exemple mal classé. Cela va forcer le prochaine appel à se concentrer autant sur les erreurs de h_1 que sur ses succès. On appelle ensuite Learn$(\alpha, \frac{\delta}{5}, EX_2)$ pour obtenir une hypothèse h_2 et si elle est suffisamment bonne on la retourne. Sinon, on construit une troisième distribution EX_3 qui sélectionne uniquement les exemples où $h_1 \neq h_2$ puis on entraîne une troisième hypothèse avec Learn$(\alpha, \frac{\delta}{5}, EX_3)$. Enfin, si $h_1 = h_2$ alors on choisit h_1 sinon on choisit h_3.

N.B : Nous remarquons que lorsque Learn est appelé récursivement, $\frac{\delta}{5}$ est pris en paramètre au lieu de δ. Cela est dû au fait qu'il y a 5 sources d'échecs lors de l'exécution de l'algorithme : l'appel récursif pour h_1, l'estimation statistique de l'erreur $\hat{a}_1$, l'appel récursif de h_2, l'estimateur de $\hat{e}$ et l'appel récursif pour h_3. Si chaque évènement a une probabilité inférieur ou égale à $\frac{\delta}{5}$ alors P(au moins une étape échoue) $\leq 5.\frac{\delta}{5} = \delta$. Donc la probabilité que l'algorithme réussisse est supérieur à $1 - \delta$.
Nous montrons maintenant le résultat suivant permettant de conclure la preuve du théorème 1 :

Théorème 2. *Soit $0 < \epsilon < \frac{1}{2}$ et $0 \leq \delta \leq 1$. Alors l'algorithme Learn(ϵ, δ, EX) retourne avec une probabilité au moins $1 - \delta$, une hypothèse h telle que $P(h(x) \neq c(x)) \leq \epsilon$*

Démonstration. Nous avons déjà expliquer dans la remarque pourquoi la probabilité qu'une bonne exécution ait lieu est au moins $1 - \delta$. Il nous reste à montrer que si Learn effectue une bonne exécution, alors l'hypothèse de sortie est $\epsilon-$proche du concept cible.
1er cas :
Dans le cas où l'estimateur $\hat{a}_1$ ou $\hat{e}$ son trouvés plus petit que $\epsilon - \tau_1$ ou $\epsilon - \tau_2$ respectivement avec $\hat{a}_1$ et $\hat{e}$ suffisamment proche de a_1 et e respectivement alors l'hypothèse retournée est $\epsilon-$proche du concept cible.
Cas général :
Soit a_i l'erreur de h_i sous la distribution D_i induite par l'oracle EX_i au i^{eme} appel récursif. Par hypothèse de récurrence nous avons pour $i = 1, 2, 3$, $a_i \leq \alpha$ car Learn est appelé récursivement avec un paramètre d'erreur α. On introduit les notations suivantes :

$$p_i(v) = P(h_i(v) \neq c(v)) \tag{16}$$

$$q(v) = P(h_1(v) \neq h_2(v)) \tag{17}$$

$$w = \underset{v \sim D}{P}(h_2(v) \neq h_1(v) = c) \tag{18}$$

$$x = \underset{v \sim D}{P}(h_1(v) = h_2(v) = c) \tag{19}$$

$$y = \underset{v \sim D}{P}(h_1(v) \neq h_2(v) = c) \tag{20}$$

$$z = \underset{v \sim D}{P}(h_1(v) = h_2(v) \neq c) \tag{21}$$

Remarque et notation : La notation $\underset{v \sim D}{P}(\pi(v))$ désigne la probabilité qu'un prédicteur τ soit vérifié sur des instances v tirées de Xn selon la distribution D.
X_n est l'ensemble des instances possibles pour le problème d'apprentissage.
Puis par la formule des probabilités totales, nous obtenons :

$$\underset{v \sim D}{P}(\tau(v)) = \sum_{v \in X_n} P(v)P(\tau(v)|v) = \sum_{v \in X_n} D(v)P(\tau(v)) \tag{22}$$

Dans la suite, afin d'alléger la notation, P_D représentera $\underset{v \sim D}{P}$.

Nous exprimons maintenant les lois de probabilité selon lesquelles les oracles EX_i génèrent les instances v. Cela nous permettra ensuite de calculer l'erreur finale de l'hypothèse h construite par Learn.
Soit $D(v)$ la probabilité que $v \in X_n$ soit tirés aléatoirement l'oracle d'exemple EX alors on a que :

$$D(v) = D_1(v) \tag{23}$$

Pour la distribution D_2, l'oracle d'exemple EX_2 est construit en lançant une pièce équilibré, puis si la pièce tombe sur pile, on retourne la première instance v tel que $h_1(v) = c(v)$. Sinon, l'algorithme retourne la première instance v de EX telle que $h_1(v) \neq c(v)$. Nous en déduisons donc que :

$$D_2(v) = \frac{1}{2}P(v|h_1(v) = c(v)) + \frac{1}{2}P(v|h_1(v) \neq c(v)) \tag{24}$$

Or par la formule de Bayes, nous avons :

$$P(v|h_1(v) \neq c(v)) = \frac{P(v \text{ et } h_1(v) \neq c(v))}{P(h_1(v) \neq c(v))} = \frac{D(v)p_1(v)}{a_1} \tag{25}$$

$$P(v|h_1(v) = c(v)) = \frac{D(v)(1 - p_1(v))}{1 - a_1} \tag{26}$$

avec $1 - a_1 = w + x = P_D(h_1(v) = c(v))$.
Finalement, en injectant ces expressions dans (23), on obtient :

$$D_2(v) = \frac{D(v)}{2}\left(\frac{p_1(v)}{a_1} + \frac{1 - p_1(v)}{1 - a_1}\right) \tag{27}$$

La distribution D_3 ne considère que les instances où les hypothèses h_1 et h_2 sont en désaccord. Nous avons donc :

$$D_3(v) = P(v|h_1(v) \neq h_2(v)) = \frac{P(v \text{ et } h_1(v) \neq h_2(v))}{P(h_1(v) \neq h_2(v))} = \frac{D(v)q(v)}{w + y} \tag{28}$$

Nous calculons maintenant l'erreur de l'hypothèse h construite à partir de h_1, h_2, h_3 i.e

$$P_D(h(v) \neq c(v)) \tag{29}$$

où $c(v)$ est la bonne étiquette de v.
D'après la définition de h, deux cas peuvent se produire :
1er cas : $h_1(v) = h_2(v)$, donc $h(v) = h_1(v) = h_2(v)$. L'algorithme fait une erreur si cette prédiction commune est fausse.
Deuxième cas : $h_1(v) \neq h_2(v)$ et dans ce cas $h(v) = h_3(v)$. Dans ce cas, l'algorithme fait une erreur si $h_3(v) \neq c(v)$.
On obtient donc :

$$P_D(h(v) \neq c(v)) = P_D(h_1(v) = h_2(v) = h(v)) + P_D((h_1(v) \neq h_2(v)) \cap (h_3(v) \neq c(v))) \tag{30}$$

Puis en utilisant le résultat de la remarque, nous avons :

$$P_D(h(v) \neq c(v)) = z + \sum_{v \in X_n} D(v)q(v)p_3(v) = z + \sum_{v \in X_n} (w + y)D_3(v)q(v)p_3(v) \tag{31}$$

Le lemme suivant donne les résultats qui permettent de conclure la preuve :

Lemme 1. *Nous avons les 3 équations suivantes :*

$$1 - a_2 = \frac{y}{2a_1} + \frac{z}{2(1 - a_1)} \tag{32}$$

$$z = a_1 - y \tag{33}$$

$$w = (2a_2 - 1)(1 - a_1) + \frac{y(1 - a_1)}{a_1} \tag{34}$$

Démonstration. On commence par montrer (32). Nous avons par définition :

$$1 - a_2 = P_{D_2}(h_2(v) = c(v)) = \sum_{v \in X_n} D_2(v)(1 - p_2(v))$$

Puis en injectant l'expression de $D_2(v)$ dans la somme, nous obtenons

$$1 - a_2 = P_{D_2}(h_2(v) = c(v)) = \sum_{v \in X_n} \left(\frac{1}{2} \frac{D(v) p_1(v)}{a_1} + \frac{1}{2} \frac{D(v)(1 - p_1(v))}{1 - a_1} \right) (1 - p_2(v))$$

$$\iff 1 - a_2 = \sum_{v \in X_n} \frac{1}{2} \frac{D(v) p_1(v)}{a_1} (1 - p_2(v)) + \frac{1}{2} \frac{D(v)(1 - p_1(v))}{1 - a_1} (1 - p_2(v))$$

Nous obtenons finalement en utilisant encore une fois la remarque que :

$$1 - a_2 = \frac{y}{2a_1} + \frac{z}{2(1 - a_1)}$$

Nous démontrons maintenant (33). Il est clair que :

$$\{h_1(v) \neq h_2(v), h_2(v) = c(v)\} \cap \{h_1(v) = h_2(v), h_2(v) = c(v)\} = \emptyset$$

Nous avons donc :

$$\begin{aligned} y + z &= P_D(\{h_1(v) \neq h_2(v),\ h_2(v) = c(v)\}) + P_D(\{h_1(v) = h_2(v),\ h_2(v) = c(v)\}) \\ &= P_D(\{h_1(v) \neq h_2(v),\ h_2(v) = c(v)\} \cup \{h_1(v) = h_2(v),\ h_2(v) = c(v)\}) \\ &= P_D(\{h_1(v) \neq c(v)\}) = a_1 \end{aligned}$$

Nous terminons la démonstration de ce lemme en montrant (34). On sait que par définition, on a :

$$\begin{aligned} w &= \sum_{v \in X_n} D(v)(1 - p_1(v)) p_2(v) \\ &= \sum_{v \in X_n} D(v)(1 - p_1(v))(1 - (1 - p_2(v))) \\ &= (1 - a_1) - \sum_{v \in X_n} D(v)(1 - p_1(v))(1 - p_2(v)) \\ &= (1 - a_1) - z \end{aligned}$$

D'autre part, en reprenant (32), nous avons :

$$\begin{aligned} & 1 - a_2 = \frac{y}{2a_1} + \frac{z}{2(1 - a_1)} \\ \iff\ & 2(1 - a_2) = \frac{y}{a_1} + \frac{z}{1 - a_1} \\ \iff\ & z = (1 - a_1) \left(2(1 - a_2) - \frac{y}{a_1} \right) \end{aligned}$$

Nous obtenons donc :

$$\begin{aligned} w &= (1 - a_1) - (1 - a_1) \left(2(1 - a_2) - \frac{y}{a_1} \right) \\ &= (1 - a_1) \left(1 - \left(2(1 - a_2) - \frac{y}{a_1} \right) \right) \\ &= (1 - a_1) \left(2a_1 - 1 + \frac{y}{a_1} \right) \\ &= (2a_2 - 1)(1 - a_1) + \frac{y(1 - a_1)}{a_1} \end{aligned}$$

□

Nous pouvons maintenant conclure la preuve de ce théorème en repartant de l'équation (31), nous obtenons :

$$\begin{aligned}
P_D(h(v) \neq c(v)) &= z + a_3(w + y) \\
&\leq z + \alpha(w + y) \quad \text{par récursivité} \\
&= a_1 - y + \alpha(2a_2 - 1)(1 - a_1) + \alpha\frac{y(1 - a_1)}{a_1} + \alpha y \quad \text{en injectant (33) et (34)} \\
&= \alpha(2a_2 - 1)(1 - a_1) + a_1 + y\left(-1 + \alpha\left(1 + \frac{1 - a_1}{a_1}\right)\right) \\
&= \alpha(2a_2 - 1)(1 - a_1) + a_1 + \frac{y(\alpha - a_1)}{a_1} \\
&= \alpha(2a_2 - 1)(1 - a_1) + a_1 + y\left(-1 + \alpha\left(1 + \frac{1 - a_1}{a_1}\right)\right) \\
&\leq \alpha(2a_2 - 1)(1 - a_1) + a_1 + \alpha - a_1 \quad \text{car} \quad y \leq a_1 \\
&= \alpha(2\alpha - 1)(1 - \alpha) \\
&= 3\alpha^2 - 2\alpha^3 = g(\alpha) = \epsilon
\end{aligned}$$

Ce qui conclut la preuve du théorème 2. □

D'après le théorème 2, nous savons que l'algorithme Learn(ϵ, δ,EX) retourne avec une probabilité au moins $1 - \delta$ une hypothèse h tel que $\text{err}_D(h) \leq \epsilon$. Cela nous permet d'établir que tout algorithme faible peut être transformé en algorithme fort, c'est à dire capable d'atteindre une erreur arbitrairement petite. □

3.3 Schéma général de renforcement

Maintenant que nous avons expliqué ce que sont les classificateurs faibles et établi le **théorème de Schapire**, nous proposons dans cette section une approche plus intuitive et algorithmique du boosting.
Le boosting repose sur l'idée de construire un classificateur faible qui peut par exemple être construit à partir du modèle CART, que l'on combine pour obtenir un modèle global plus performant. A chaque itération, un nouveau modèle est entraîné pour corriger les erreurs du précédents de sorte à ce que chaque étape vient affiner les prédictions du modèle précédent. Nous pouvons alors décrire le boosting comme une construction additive du type :

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m) \tag{35}$$

avec $b(x, \gamma_m)$ sont les apprenants faibles et β_m sont les coefficients déterminant leur importance dans le modèle final. Cette construction est appelé **forward stagewise additive modelling** et peut être vu comme une somme de corrections successives apportées à un modèle initialement trivial.
L'intuition centrale du boosting est que l'attention du modèle est déplacée vers les observations mal prédites. Autrement dit, à chaque étape, on entraîne un classificateur sur une version pondérée des données où les observations bien prédites voient leur poids diminuer tandis que celles mal prédites voient leurs poids augmenter.
Le boosting modifie donc implicitement à chaque itération la distribution des données en accentuant l'importance des cas difficiles. Cette idée déjà présente dans la construction théorique du boosting de Schapire en 1990 se retrouve également au coeur des algorithmes modernes tel que **AdaBoost** ou **XGBoost**.
Dans la suite, nous allons présenter plus en détails les deux plus importants algorithmes de boosting que sont AdaBoost et XGBoost.

4 AdaBoost

4.1 Présentation générale d'Adaboost

AdaBoost est un algorithme de boosting introduit en 1997 par Yoav Freund et Robert Schapire. Il a été l'un des premiers algorithmes qui a traduit les fondements théoriques du boosting en une méthode concrète.
Dans AdaBoost, les modèles faibles les plus fréquemment utilisés sont des arbres de décision de profondeur 1, appelés stumps. Ce sont donc des arbres qui effectuent une seule coupure sur une variables explicative ce qui les rend très simple mais également très limités individuellement. Nous les construisons, en utilisant la méthode de construction des arbres de classification décrite dans la partie traitant du modèle **CART** c'est à dire que le stump cherche la variable et le seuil qui mesure d'impureté comme l'indice Gini ou l'entropie croisée. Néanmoins, puisqu'il ne peut utiliser qu'une seule variable, un stump ne peut capter qu'une information partielle sur le problème.

Exemple : Si nous souhaitons prédire le risque de défaut d'un emprunteur, nous pourrions utiliser plusieurs caractéristiques telles que le revenu, la situation professionnelle, le statut marital etc... Un stump ne peut se baser que sur l'une de ces variables ce qui limite fortement sa prédiction. Ce genre d'arbre est considéré comme des classificateurs faibles. AdaBoost entraîne de façon séquentielle ces stumps en modifiant de façon dynamique le poids des observations à chaque itération. Au départ, chaque observation a le même poids et un premier stump noté G_1 est entraîné sur ses données pondérées. Ensuite, les poids sont mis à jour de façon à ce que les observations mal classées voient leurs poids augmenter tandis que les observations bien classées voient leur poids diminuer. Le stump suivant se concentrera davantage sur les observations avec un poids plus important et ce processus est répété un nombre de fois M fixé. Ainsi, l'ordre des stumps a une importance car chaque modèle dépend de ceux qui le précèdent. A la fin, les stumps sont combinés via un vote pondéré où chaque stump reçoit un poids α_m. Dans le cas d'une classification binaire la prédiction finale est donnée par :

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{36}$$

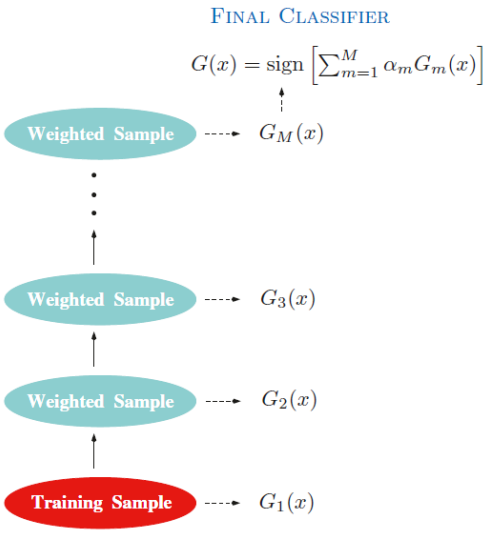

Figure 4 – Schéma de fonctionnement d'AdaBoost
Source : Hastie, Tibshirani, Friedman – *The Elements of Statistical Learning*, 2e édition, Springer (2009), Figure 10.1.

4.2 Description détaillée de l'algorithme

Nous présentons dans cette partie l'algorithme **AdaBoost** et nous donnons également un exemple illustratif permettant de bien le comprendre. Nous considérons un problème de classification à deux classes. La variable à prédire Y prend donc deux valeurs : 1 et -1. Pour un "feature" X, le classificateur produit $G(X)$ à valeurs dans $\{-1, 1\}$. L'algorithme AdaBoost se présente comme suit :

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

FIGURE 5 – Pseudo code de l'algorithme AdaBoost.M1.
Source : Hastie, Tibshirani, Friedman – *The Elements of Statistical Learning*, 2e édition, Springer (2009), Algorithm 10.1.

Nous considérons les observations $\{(x_1, y_1), ..., (x_N, y_N)\}$. Nous commençons dans un premier temps à attribuer à toutes les observations un poids identique :

$$w_i^{(1)} = \frac{1}{N} \text{ pour } i = 1, ..., N$$

Ensuite, pour l'itération $m \in \{1, ..., M\}$, nous entraînons le classificateur faible $G_m(x)$ sur les données d'entraînement pondérées selon les poids $w_i^{(m)}$. Les erreurs sur les observations les plus pondérées sont pénalisées davantage. Pour cette itération, nous calculons l'erreur pondérée qui est définie par :

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i} \tag{37}$$

L'intérêt de normaliser l'erreur de prédiction est d'avoir une interprétation probabiliste de cette erreur qui ne peut prendre que des valeurs comprises dans l'intervalle $[0, 1]$. Cela nous permet donc d'avoir une interprétation homogène de la performance du modèle courant. Autrement dit, cela permet de faire en sorte que l'erreur reste à la même échelle et donc que les poids conservent une cohérence à chaque itération.
Une fois que nous calculons l'erreur pondérée, ϵ_m du classificateur G_m, nous quantifions son importance dans le modèle final. Nous calculons son poids α_m définie par :

$$\alpha_m = \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right) \tag{38}$$

Ce poids va permettre de mesurer la fiabilité du classificateur. En effet, plus err_m sera faible, plus α_m sera grand augmentant ainsi son influence dans la prédiction finale. A l'inverse, lorsque err_m se rapproche de 0.5 c'est à dire que le classificateur a une précision équivalente à celle du hasard alors α_m tend vers 0. Il est important de noter que la forme de α_m n'est pas arbitraire et découle de la minimisation d'une fonction de perte exponentielle dans un modèle additif. Nous justifions en détail cette affirmation dans la section 4.3.

Nous mettons ensuite à jour le poids des observations w_i. L'idée à cette étape est d'augmenter l'importance des observations mal classées et de réduire l'importance de celles qui ont été bien classées. En effet, si une observation i est bien classée alors $I(y_i \neq G_m(x)) = 0$ et

$$w_i \exp(0) = w_i$$

donc son poids reste inchangé. A l'inverse si une observation est mal classée alors $I(y_i \neq G_m(x)) = 1$ et son poids est multiplié par un facteur supérieur ou égale à 1. Intuitivement, cela signifie que l'algorithme va davantage sur les observations mal classées. Nous expliquerons la forme de la mise à jour des poids w_i dans la section 4.3. Enfin, après M itérations, le classificateur final sera :

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{39}$$

Chaque G_m vote pour la classe 1 ou -1 sur la donnée x et son influence est déterminée par son poids α_m. Les votes des classificateurs les plus performants comptent davantage dans le vote final.

Remarque : Dans l'algorithme présenté, nous utilisons deux notions de poids qu'il est important de ne pas confondre. D'une part, w_i représente le poids de l'observation i dans l'apprentissage du prochain classificateur. Il est ajusté de manière à ce que les observations mal classés obtiennent un poids plus important. Par ailleurs, α_m est le poids du classificateur m et détermine son influence dans la prédiction finale.

Exemple : On considère le jeu de données suivant contenant trois observations :

x_i	y_i
A	1
B	-1
C	1

FIGURE 6 – Jeu de données simplifié à trois observations pour illustrer le fonctionnement de l'algorithme AdaBoost

Nous appliquons AdaBoost pendant 2 itérations. A la première itération, les trois observations ont un poids $w_i^{(1)} = \frac{1}{3}$ pour $i = 1, 2, 3$. Nous supposons que le premier classifieur G_1 prédit correctement les points 1 et 3 mais se trompe sur le point 2. En calculant l'erreur pondérée on trouve $\text{err}_m = \frac{1}{3}$ et le poids du classifieur est $\alpha_1 = \log(2)$. Enfin actualisant les poids w_i, nous avons $w_1 = \frac{1}{3}, w_2 = \frac{2}{3}, w_3 = \frac{1}{3}$. Donc nous retrouvons bien l'idée que les observations mal classés se voient attribués un poids plus important. A la deuxième itération, nous supposons que le classifieur G_2 prédit correctement toutes les observations. Nous avons $\text{err}_2 = 0$ et $\alpha_2 \to +\infty$. En pratique, nous fixons un ϵ très petit par exemple $\epsilon = 10^{-10}$ donc $\alpha_2 \approx 10\log(10) \approx 23$. Finalement, la sortie de l'algorithme sera $G(x) = \text{sign}(\alpha_1 G_1(x) + \alpha_2 G_2(x))$ et comme $\alpha_2 >> \alpha_1$, le vote du classificateur faible aura plus d'importance en cohérence avec ce qui a été expliqué.

4.3 Analyse théorique

4.3.1 AdaBoost comme modèle additif

Un modèle additif est un modèle qui s'écrit sous la forme :

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m) \tag{40}$$

avec β_m les coefficients d'expansion et $b(x, \gamma_m)$ est une fonction définie sur x et paramétré par γ_m. Il apparaît donc que dans ce modèle, les paramètres inconnus sont les β_m et γ_m. Afin de les estimer,

nous voudrions déterminer la fonction f qui minimise une fonction de perte sur l'échantillon d'apprentissage c'est à dire qui permettrait d'obtenir la plus grande précision sur la prédiction globale. Le problème d'optimisation serait donc le suivant :

$$\min_{\{(\beta_m,\gamma_m)\}_{m=1}^{M}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m\, b(x_i;\gamma_m)\right) \tag{41}$$

Résoudre ce problème d'optimisation nous permettrait d'ajuster tous les paramètres simultanément de sorte à minimiser l'erreur globale. Toutefois, il apparait que ce problème est souvent infaisable en terme de complexité de calcul. Une alternative va consister à déterminer les paramètres (β_m, γ_m) en minimisant la fonction de perte par étapes successives. Cette procédure est connue sous le nom de **modélisation additive par étapes**. Elle est décrite dans l'algorithme suivant qui constitue une méthode générale d'optimisation pour les modèles additifs.

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M:
   (a) Compute
   $$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i;\gamma)).$$
   (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x;\gamma_m)$.

FIGURE 7 – Pseudo-code de la modélisation additive par étapes
Source : Hastie, Tibshirani, Friedman – *The Elements of Statistical Learning*, 2e édition, Springer (2009), Algorithm 10.2.

Nous commençons par initialiser la fonction f à 0 c'est à dire que nous ne faisons aucune prédiction au départ. Puis pour chaque étape $m \in \{1, ..., M\}$, nous recherchons parmi toutes les fonctions de base possible $b(x, \gamma)$ et tous les coefficients β permettant de réduire la perte globale si nous l'ajoutons au modèle. Autrement dit, nous cherchons la fonction de base qui permet d'améliorer le plus la prédiction globale. Ceci se traduit dans l'algorithme par :

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, f_{m-1}(x_i) + \beta\, b(x_i;\gamma)\right) \tag{42}$$

Une fois ce couple de paramètres trouvés, nous mettons à jour le modèle :

$$f_m(x) = f_{m-1}(x) + \beta_m b(x, \gamma_m) \tag{43}$$

A la fin des M itérations, nous trouvons la forme finale de f.
A partir de l'équation (39), nous pouvons voir qu'**AdaBoost** peut être interprété comme un modèle additif construit de manière incrémentale. Dans notre cas, les fonctions $b(x, \gamma_m)$ correspondent aux classificateurs faibles $G_m(x) \in \{-1, 1\}$ et les coefficients β_m sont les poids α_m attribués à chaque classifieurs. Nous allons montrer dans la partie qui suit, qu'AdaBoost correspond à la minimisation de la fonction de perte exponentielle au sein de ce cadre additif.

4.3.2 Fonction de perte exponentielle

Nous allons montrer dans cette partie qu'AdaBoost est équivalent à l'algorithme de modélisation additive par étapes lorsqu'on choisit la fonction de perte :

$$L(x, f(x)) = \exp(-yf(x)) \tag{44}$$

qu'on appelle **fonction de perte exponentielle**.

Démonstration. Nous reprenons l'algorithme présenté à la figure 6 et nous supposons qu'à l'itération $m-1$, nous ayons construit le modèle partiel $f_{m-1}(x)$. Nous cherchons alors le terme $\beta G(x)$ qui permet d'obtenir $f_m(x) = f_{m-1}(x) + \beta G(x)$ de sorte à minimiser la perte totale :

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{N} \exp\left(-y_i \left(f_{m-1}(x_i) + \beta G(x_i)\right)\right) \tag{45}$$

Nous définissons les poids mesurant à quel point l'observation i est mal classée par :

$$w_i^{(m)} = \exp(-y_i f_{m-1}(x_i)) \tag{46}$$

Lorsque la prédiction est incorrecte, $y_i f_{m-1}(x_i) < 0$ et par croissance de la fonction exponentielle le poids $w_i^{(m)}$ est plus élevé que lorsque la prédiction est correcte en cohérence avec ce que nous avons expliqué lorsque nous avons traité l'algorithme **AdaBoost**. On réecrit alors (45) de la façon suivante :

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{N} w_i^{(m)} \exp\left(-y_i \beta G(x_i)\right) \tag{47}$$

Par ailleurs, comme on considère que $y_i, G(x_i) \in \{-1, 1\}$, on a alors que $\exp(-\beta y_i G(x_i)) = \exp(-\beta)$ si $y_i = G(x_i)$ et $\exp(-\beta y_i G(x_i)) = \exp(\beta)$ si $y_i \neq G(x_i)$. Nous posons :

$$C = \{i : y_i = G(x_i)\}$$
$$M = \{i; y_i \neq G(x_i)\}$$

L'équation (47) devient alors :

$$(\beta_m, G_m) = \arg\min_{\beta, G} \exp(-\beta) \sum_{i \in C} w_i^{(m)} + \exp(\beta) \sum_{i \in M} w_i^{(m)} \tag{48}$$

Par ailleurs, on sait aussi que :

$$\sum_{i=1}^{N} w_i^{(m)} = \sum_{i \in C} w_i^{(m)} + \sum_{i \in M} w_i^{(m)} \tag{49}$$

Puis en injectant l'expression de $\sum_{i \in C} w_i^{(m)}$ dans (48), on obtient :

$$\begin{aligned}
\exp(-\beta) \sum_{i \in C} w_i^{(m)} + \exp(\beta) \sum_{i \in M} w_i^{(m)} &= \exp(-\beta) \left(\sum_{i=1}^{N} w_i^{(m)} - \sum_{i \in M} w_i^{(m)} \right) + \exp(\beta) \sum_{i \in M} w_i^{(m)} \\
&= \exp(-\beta) \sum_{i=1}^{N} w_i^{(m)} + (\exp(\beta) - \exp(-\beta)) \sum_{i \in M} w_i^{(m)}
\end{aligned}$$

Pour β fixé, minimiser (48) revient à minimiser la somme pondérée des erreurs :

$$G_m = \arg\min_{G} \sum_{i \in M} w_i^{(m)} = \arg\min_{G} \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)) \tag{50}$$

ce qui est exactement fait par AdaBoost i.e à chaque itération G_m est ajusté de sorte à minimiser la somme pondérée des erreurs. Nous fixons maintenant G_m et nous définissons :

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i^{(m)}} \tag{51}$$

En reprenant encore une fois l'expression à minimiser dans (48), nous obtenons :

$$L(\beta) = \exp(-\beta)(1 - \text{err}_m) \sum_{i=1}^{N} w_i^{(m)} + \exp(\beta)\, \text{err}_m \sum_{i=1}^{N} w_i^{(m)} \tag{52}$$

Finalement pour obtenir la valeur de β qui minimise L, nous résolvons l'équation :

$$\begin{aligned}
\frac{dL}{d\beta}(\beta) = 0 &\iff -(1 - \text{err}_m)\exp(-\beta) + \text{err}_m \exp(\beta) = 0 \\
&\iff \exp(-2\beta) = \frac{\text{err}_m}{1 - \text{err}_m} \\
&\iff \beta_m = \frac{1}{2}\log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)
\end{aligned}$$

C'est exactement la formule à un facteur près du poids du classificateur m utilisée dans AdaBoost. Finalement, pour une étape donnée, l'algorithme se termine par la mise à jour du modèle :

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x) \tag{53}$$

Le poids à l'étape $m + 1$ devient donc :

$$w_i^{(m+1)} = \exp(-y_i f_m(x_i)) = w_i^{(m)} \exp(-\beta_m y_i G_m(x_i)) \tag{54}$$

et en utilisant le fait que $-y_i G_m(x_i) = 2I(y_i \neq G_m(x_i)) - 1$, nous obtenons donc que :

$$w_i^{(m+1)} = w_i^{(m)} \exp\left(\alpha_m \mathbb{I}(y_i \neq G_m(x_i))\right) \exp(-\beta_m) \tag{55}$$

avec $\alpha_m = 2\beta_m$. Comme le facteur $\exp(-\beta_m)$ multiplie tous les poids, il n'a aucun effet. □

Nous venons donc de montrer que la modélisation additive par étape avec une fonction de perte exponentielle est équivalente à appliquer l'algorithme AdaBoost. Nous allons maintenant essayer de comprendre pour la fonction de perte exponentielle permet de retrouver Adaboost et en tirer une interprétation probabiliste.

4.3.3 Interprétation probabiliste

Dans la section précédente, nous avons présenté perte exponentielle empirique c'est à dire que les différentes minimisation s'effectuent sur un jeu de données $\{(x_i, y_i)\}_{i=1}^N$. Nous allons maintenant, dans cette section considéré la forme théorique de la perte exponentielle afin de déterminer la fonction idéale qu'AdaBoost essaie d'estimer. Nous introduisons alors la fonction de perte exponentielle théorique comme :

$$L(f) = \mathbb{E}_{Y|x}[\exp(-Yf(x))] \tag{56}$$

où l'espérance est prise selon la loi de Y conditionnellement à l'entrée x. Cela est justifié par le fait que nous connaissons l'entrée x et nous voulons chercher la valeur optimal de $f^*(x)$ qui minimiser la perte moyenne sur la distribution de Y sachant cette entrée x. Nous allons alors chercher la fonction f^* tel que :

$$f^*(x) = \arg\min_f \mathbb{E}_{Y|x}\left[\exp\left(-Yf(x)\right)\right] \tag{57}$$

Comme $Y \in \{-1, 1\}$, nous notons $p = P(Y = 1|x)$ et donc $1 - p = P(Y = -1|x)$. Nous obtenons donc :

$$L(f) = p \cdot \exp(-f) + (1 - p) \cdot \exp(f)$$

Pour minimiser cette expression, nous résolvons :

$$\begin{aligned}
&\frac{dL}{df} = -p \cdot \exp(-f) + (1 - p) \cdot \exp(f) = 0 \\
\iff\quad & (1 - p) \cdot \exp(f) = p \cdot \exp(-f) \\
\iff\quad & \exp(2f) = \frac{p}{1 - p} \\
\iff\quad & f^*(x) = \frac{1}{2}\log\left(\frac{p}{1 - p}\right)
\end{aligned}$$

Ce résultat peut se réécrire comme :

$$f^*(x) = \frac{1}{2}\log\left(\frac{P(Y=1|x)}{P(Y=-1|x)}\right) \tag{58}$$

ou de façon équivalente :

$$\Pr(Y=1 \mid x) = \frac{1}{1+e^{-2f^*(x)}} \tag{59}$$

Nous constatons alors que le score f appris par AdaBoost est la moitié d'un logit de la probabilité conditionnelle $P(Y=1|x)$. Donc Adaboost peut également être interprété comme un estimateur probabiliste apprenant une approximation du logit.
Par ailleurs, nous introduisons un autre critère utilisé pour modéliser la probabilité conditionnelle $P(Y=1|x)$ est la log-vraisemblance binomiale négative également appelée déviance définie par :

$$l(Y, f(x)) = \log(1+\exp(-2Yf(x))) \tag{60}$$

Cette fonction va nous permettre de réaliser une comparaison de robustesse avec le perte exponentielle car nous pouvons montrer que cette fonction est minimisé comme la perte exponentielle par :

$$f^*(x) = \frac{1}{2}\log\left(\frac{P(Y=1|x)}{P(Y=-1|x)}\right)$$

et donc visent le même objectif théorique.

4.3.4 Limite de la perte exponentielle et problèmes de robustesse

Nous nous intéressons dans cette section à la robustesse d'**Adaboost** et nous allons notamment la comparer avec celle de la déviance. Nous rappelons que la fonction score $f(x)$ associée à une observation permet de mesurer la confiance du modèle dans sa prédiction. Plus cette fonction score est grande et plus le modèle a confiance. Lorsque $yf(x) > 0$, l'observation est bien classé et lorsque cette quantité est négative elle est mal classée. Quand $yf(x) = 0$, le modèle se trouve exactement sur la frontière de décision. Par ailleurs, nous savons que l'objectif de tout algorithme de classification est de produire des marges positives aussi fréquemment que possible.

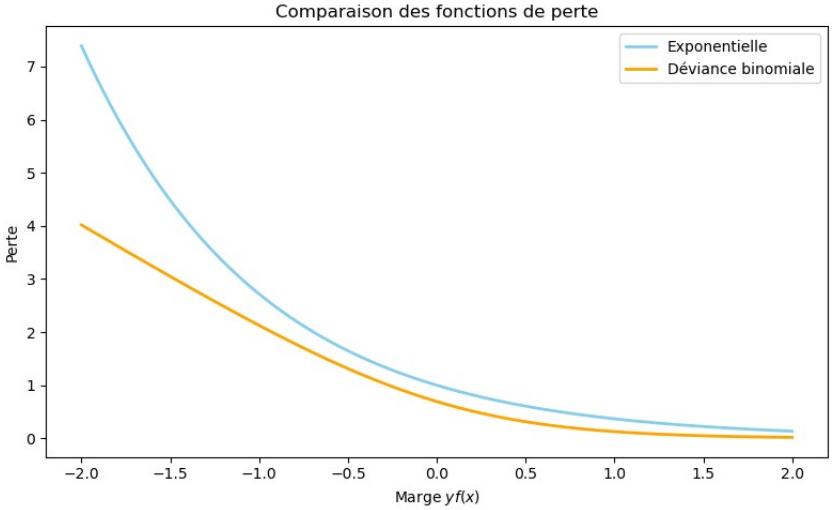

FIGURE 8 – Comparaison entre la perte exponentielle et la déviance binomiale en fonction de la marge $yf(x)$.

La figure ci-dessus compare la perte en fonction de la marge $yf(x)$ de la fonction de perte exponentielle et de la déviance. Nous pouvons observer que la déviance augmente de façon assez linéaire pour les marges négatives tandis que la perte exponentielle croit de façon exponentielles pour ces

même marges. Cela implique que le critère exponentielle concentre une influence plus forte sur les observations mal classés que la déviance qui elle, répartit l'influence plus uniformément sur toutes les observations.
Cela rend donc la déviance nettement plus robuste dans un contexte bruité. En effet, une observation est mal étiquetée ou aberrante alors la perte exponentielle va donner un poids significatif à cette observation quitte à modifier fortement le modèle. Cela peut donc détériorer les prédictions sur tout le reste du jeu de données. La perte exponentielle possède alors un risque de surapprentissage localisé. Cela est moins le cas de la déviance qui prend en compte ce genre de points mais sans leur données une importance significative.
Nous venons ici de mettre en évidence certaines limites d'AdaBoost et notamment sa grande sensibilité aux observations mal classées. Nous allons alors chercher à développer un algorithme plus robuste et donc nous intéresser à XGBoost.

5 XGBoost

5.1 Motivation et introduction

Après avoir présenté **AdaBoost**, qui repose sur une pondération adaptative des observations mal classées pour combiner plusieurs classificateurs faibles, nous allons introduire une approche plus générale et plus puissante : **le Gradient Boosting.**

L'idée fondamentale derrière le Gradient Boosting est de reformuler le problème du boosting comme un problème d'optimisation, où chaque nouvel apprenant faible est construit dans le but de réduire l'erreur résiduelle du modèle précédent, à l'aide de la descente de gradient.

Plutôt que de se focaliser uniquement sur les erreurs de classification comme dans AdaBoost, le Gradient Boosting adopte une approche plus flexible : il minimise une fonction de perte différentiable quelconque (par exemple, l'erreur quadratique en régression) en ajoutant itérativement des prédicteurs qui pointent dans la direction du négatif du gradient de la fonction de perte, calculé par rapport aux prédictions actuelles du modèle.

C'est dans ce cadre qu'est introduit **XGBoost** (eXtreme Gradient Boosting) par Tianqi Chen et Carlos Guestrin en 2016 dans *XGBoost : A Scalable Tree Boosting System* , une version optimisée et hautement performante du Gradient Boosting, largement utilisée en pratique et ayant gagné de nombreuses compétitions de science des données.

Comparé au Gradient Boosting classique et à d'autres méthodes basées sur les arbres, XGBoost présente deux avantages majeurs.

Premièrement, il est efficace sur le plan computationnel, car il traite les données sous forme de blocs compressés, ce qui permet un tri et un traitement rapides en parallèle. Cela lui confère un temps d'entraînement nettement inférieur à celui du Gradient Boosting classique (cf. graphique ci-dessous).

Deuxièmement, il utilise un développement de Taylor d'ordre deux pour minimiser la fonction objectif. Cela signifie qu'il prend en compte la courbure de la fonction objectif, ce qui lui permet de converger vers un minimum de meilleure qualité que d'autres méthodes.

Il est également important de noter que XGBoost ne s'appuie pas sur les critères classiques utilisés pour la construction des arbres de décision, tels que l'entropie croisée ou l'indice de Gini définis précédemment. Il utilise à la place des critères spécifiques, qui sont présentés et expliqués en détail dans ce document.

Troisièmement, XGBoost permet via des mécanismes de régularisation intégrés de limiter le surapprentissage.

Enfin, il permet également de gérer automatiquement les données manquantes de manière intelligente. Lors de l'entraînement, le modèle apprend quelle direction (gauche ou droite dans l'arbre) est la plus optimale pour une valeur manquante, ce qui évite d'avoir à forcer une imputation artificielle.

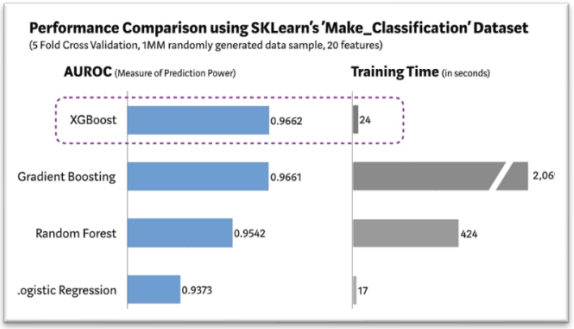

FIGURE 9 – Enter Caption

5.2 Principe général du Gradient Boosting

Le Gradient Boosting repose sur l'utilisation séquentielle de plusieurs arbres de décision. En effet, contrairement à la forêt aléatoire, les arbres de décision ne sont pas construits indépendamment. Le

nouvel arbre est entraîné par les erreurs commises par l'ensemble des arbres précédents. Pour chaque observation, chaque arbre fournit une prédiction, et la prédiction finale est obtenue en additionnant les contributions de l'ensemble des arbres.

Avant de discuter du Gradient Boosting, il est important de revenir sur l'idée du boosting glouton. Le boosting traditionnel consiste à ajouter des arbres de manière séquentielle, avec l'objectif de corriger les erreurs faites par les arbres précédents. À chaque itération, un nouvel arbre est ajouté pour améliorer la prédiction globale. Cette méthode peut être considérée comme gloutonne dans la mesure où, à chaque étape, l'algorithme se concentre uniquement sur l'erreur locale (les erreurs résiduelles des arbres précédents) sans considérer de manière globale la minimisation de la fonction de perte.

Le Gradient Boosting améliore cette approche en introduisant une optimisation plus globale qui prend en compte la fonction de perte dans son ensemble, en suivant le gradient de celle-ci. L'idée est de guider l'algorithme dans une direction plus précise à chaque itération pour minimiser la fonction de perte de manière optimale, et non simplement de réduire les erreurs résiduelles locales.

Le Gradient Boosting repose sur un concept fondamental : suivre le gradient de la fonction de perte. Cela signifie que chaque nouvel arbre construit à l'itération m est ajusté non seulement pour corriger les erreurs résiduelles du modèle précédent, mais pour suivre la direction du gradient de la fonction de perte. Le gradient représente la direction dans laquelle la fonction de perte diminue le plus rapidement, et c'est dans cette direction que l'arbre doit être ajusté.
Pour calculer ce gradient, on prend la dérivée de la fonction de perte par rapport à la prédiction du modèle, ce qui nous donne une idée claire de l'ajustement nécessaire pour réduire l'erreur.

Au début du processus de boosting, chaque nouvel arbre est ajouté pour corriger les erreurs des prédictions précédentes. À chaque itération m, on calcule le gradient de la fonction de perte $L(y_i, f(x_i))$ par rapport aux prédictions actuelles $f_{m-1}(x_i)$ (après m-1 itérations), c'est-à-dire celles issues des arbres précédemment ajoutés.
Ce gradient est défini comme suit :

$$g_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\bigg|_{f(x_i)=f_{m-1}(x_i)}$$

Cela correspond à la dérivée de la fonction de perte par rapport à la prédiction actuelle du modèle. Le gradient indique la direction dans laquelle les prédictions doivent être ajustées pour minimiser l'erreur. Plus précisément, il montre à quel point il faut ajuster les prédictions pour réduire la fonction de perte. C'est dans cette direction que le nouvel arbre sera construit.

Une fois le gradient calculé, un nouvel arbre de décision $T_m(x_i)$ est construit. Cet arbre a pour objectif de minimiser l'erreur sur les résidus, c'est-à-dire d'ajuster les prédictions en suivant la direction du gradient. L'ajustement de l'arbre permet de corriger les erreurs du modèle actuel en suivant la pente négative du gradient de la fonction de perte.

L'arbre est ajusté de manière à réduire les erreurs sur les données, et la prédiction est mise à jour comme suit :

$$f_m(x) = f_{m-1}(x) + T_m(x)$$

Le modèle est mis à jour après l'ajout de chaque arbre. La prédiction totale $f_M(x)$ après M itérations est donnée par la somme des prédictions successives :

$$f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m)$$

À chaque itération, chaque arbre $T(x;\Theta_m)$ est ajusté de manière à minimiser les erreurs des arbres précédents, en suivant la direction du gradient de la fonction de perte. Cela permet de réduire l'erreur globale de manière plus efficace que dans le boosting traditionnel.

Le calcul des régions R_j et des prédictions γ_j est au cœur de l'optimisation du boosting par gradient. Chaque région R_j est déterminée pour minimiser la fonction de perte $L(y_i, \gamma_j)$, en ajustant les partitions de l'espace de manière à réduire les résidus.

Les régions R_j peuvent être calculées de manière itérative, et la solution optimale pour chaque γ_j dans une région R_j est trouvée par un algorithme de partitionnement de type régression, qui vise à minimiser l'erreur quadratique dans chaque région.

Dans le gradient boosting, une approche de **descente de gradient** est utilisée pour résoudre le problème d'optimisation. La descente de gradient est une méthode itérative qui permet de trouver les paramètres Θ_m qui minimisent la fonction de perte. À chaque itération, le modèle est ajusté en suivant la direction du gradient de la fonction de perte.

La mise à jour du modèle $f_m(x)$ en ajoutant un arbre $T_m(x)$ est réalisée en utilisant la solution de l'optimisation suivante :

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} L\left(y_i, f_{m-1}(x_i) + T(x_i;\Theta_m)\right)$$

Cela permet d'ajuster progressivement le modèle pour réduire l'erreur sur l'ensemble des données, en ajoutant des arbres successifs qui suivent la direction du gradient.

On a ainsi l'algorithme de Boosting d'Arbres de Décision par Gradient :

1. **Initialisation** : Initialiser le modèle $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$, avec une prédiction simple (par exemple, la moyenne des y_i).
2. **Pour** $m = 1$ **à** M :
   (a) Pour $i = 1, 2, \ldots, N$, calculer les résidus r_{im} comme suit :
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}.$$
   (b) Ajuster un arbre de régression $T(x;\Theta_m)$ aux cibles r_{im}, ce qui donne les régions terminales $R_{jm}, j = 1, 2, \ldots, J_m$.
   (c) Pour $j = 1, 2, \ldots, J_m$, calculer les prédictions γ_{jm} en résolvant :
   $$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Mettre à jour la prédiction du modèle :
   $$f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm}).$$
3. **Sortie** : La prédiction finale du modèle est donnée par $\hat{f}(x) = f_M(x)$.

5.3 Du Gradient Boosting à XGBoost

On a vu que le Gradient Boosting repose sur l'idée de construire des modèles successifs pour réduire les erreurs résiduelles des arbres précédents, en ajustant les prédictions vers la direction du gradient de la fonction de perte. Cela est réalisé en suivant une approche itérative où, à chaque itération, un nouvel arbre est ajouté pour améliorer les prédictions. Bien que

ce processus soit puissante, il présente certaines limites. Il peut être lent à l'exécution sur de grands volumes de données en raison de son apprentissage séquentiel.Ce modèle est également sujet à un risque de surapprentissage si les arbres sont trop complexes. De plus, il gère mal les valeurs manquantes, nécessitant souvent une imputation préalable. Enfin, son optimisation repose uniquement sur le gradient de premier ordre, sans prendre en compte la courbure de la fonction de perte, ce qui peut limiter sa précision.

XGBoost (eXtreme Gradient Boosting) surmonte ces limitations en introduisant plusieurs améliorations notables, tant sur le plan de l'optimisation que de la régularisation.

5.4 XGBoost : Optimisation et Régularisation

XGBoost prend les principes du Gradient Boosting et les améliore avec plusieurs optimisations clés, qui sont détaillées ci-dessous.

5.4.1 Régularisation

L'une des principales innovations de XGBoost par rapport au Gradient Boosting classique est l'ajout de régularisation dans la fonction objectif. La régularisation permet de mieux contrôler la complexité des arbres et d'éviter le surajustement, en limitant la profondeur des arbres et la norme des poids.

La fonction objectif dans XGBoost comprend donc un terme de régularisation, $\Omega(T)$, ajouté à la fonction de perte. La fonction objectif totale devient ainsi :

$$\mathcal{L}(\Theta) = \sum_{i=1}^{N} L(y_i, \hat{y}_i) + \lambda \sum_{j=1}^{J} \|\gamma_j\|^2 + \gamma \sum_{k=1}^{K} d_k$$

— $\sum_{i=1}^{N} L(y_i, \hat{y}_i)$: fonction de perte qui mesure l'écart entre la prédiction $\hat{y}_i$ et la vérité y_i. Cette fonction doit être différentiable et convexe (afin de pouvoir être optimisée facilement et atteindre ainsi son minimum)

— Pénalisation des poids :

$$\lambda \sum_{j=1}^{J} \|\gamma_j\|^2$$

Cette somme pénalise la norme des poids (ou scores) associés aux feuilles des arbres. Cela évite que les poids prennent des valeurs trop extrêmes.

— Signification des symboles :
  — γ_j : poids attribué à la feuille j dans un arbre.
  — $\|\gamma_j\|^2$: carré de la norme (souvent γ_j^2 si γ_j est scalaire), induisant une **pénalisation quadratique**.
  — $\sum_{j=1}^{J}$: cette régularisation est appliquée à toutes les feuilles $j = 1, \ldots, J$.
  — λ : hyperparamètre qui permet de réduire la complexité du modèle et d'éviter l'overfitting. λ contrôle la pénalisation des poids (plus λ est grand, plus on freine les valeurs extrêmes de γ_j).

— Pénalisation structurelle :

$$\gamma \sum_{k=1}^{K} d_k$$

Cette somme pénalise la complexité structurelle des arbres du modèle en limitant leur profondeur.

— Signification des symboles :
  — d_k : profondeur ou complexité de l'arbre k, par exemple le nombre de feuilles.
  — $\sum_{k=1}^{K} d_k$: somme de toutes les profondeurs des arbres $k = 1, \ldots, K$.
  — γ : hyperparamètre qui contrôle la pénalisation sur la taille des arbres (plus γ est élevé, plus le modèle favorise des arbres simples, moins profonds).

5.4.2 Optimisation de la fonction objectif via l'approximation de Taylor

L'une des contributions essentielles de XGBoost est l'utilisation d'une écriture de la fonction objectif permettant une optimisation efficace à chaque itération m. Cela repose sur un développement de Taylor à l'ordre 2 de la fonction de perte, qui facilite l'utilisation conjointe du gradient et de la hessienne pour accélérer la convergence.

Approximation de la fonction objectif

On cherche à minimiser la fonction objectif suivante à l'itération m :

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

En développant $L(y_i, \hat{y}_i^{(m-1)} + f_m(x_i))$ au voisinage de $\hat{y}_i^{(m-1)}$ avec un développement de Taylor à l'ordre 2, on obtient :

$$\mathcal{L}^{(m)}(f_m) \approx \sum_{i=1}^{N} \left[g_i \cdot f_m(x_i) + \frac{1}{2} h_i \cdot f_m^2(x_i) \right] + \Omega(f_m)$$

avec :

— $g_i = \left. \frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i} \right|_{\hat{y}_i = \hat{y}_i^{(m-1)}}$: le gradient (dérivée première)

— $h_i = \left. \frac{\partial^2 L(y_i, \hat{y}_i)}{\partial \hat{y}_i^2} \right|_{\hat{y}_i = \hat{y}_i^{(m-1)}}$: la hessienne (dérivée seconde)

Structure de f_m et simplification de l'écriture

On suppose que $f_m(x_i)$ correspond au poids w_j de la feuille j de l'arbre dans laquelle tombe l'observation x_i. On note I_j l'ensemble des indices i tels que x_i est assigné à la feuille j, et J le nombre total de feuilles de l'arbre.

En remplaçant $f_m(x_i) = w_j$ pour $i \in I_j$, on obtient :

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

Interprétation des termes de régularisation

— λ est un hyperparamètre qui pénalise les valeurs élevées des poids w_j, pour éviter des scores extrêmes et donc du surapprentissage.

— γ pénalise le nombre total de feuilles J, ce qui favorise les arbres plus simples.

Calcul du poids optimal w_j de chaque feuille

On minimise la fonction objectif par rapport à chaque w_j :

$$\frac{\partial \mathcal{L}^{(m)}}{\partial w_j} = \left(\sum_{i \in I_j} g_i \right) + \left(\sum_{i \in I_j} h_i + \lambda \right) w_j = 0$$

Ce qui donne la solution optimale :

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

Perte optimisée associée à un arbre

En remplaçant w_j^* dans l'expression de $\mathcal{L}^{(m)}$, on obtient la perte finale de l'arbre construit à l'itération m :

$$\mathcal{L}^{(m)} = -\frac{1}{2}\sum_{j=1}^{J}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma J$$

Cette quantité est utilisée pour déterminer le gain d'information G lors du choix optimal de split dans l'arbre.

5.4.3 Critère de Split Optimal dans XGBoost

Pour qu'une séparation (split) d'une feuille en deux nouvelles feuilles soit considérée comme intéressante, il est nécessaire que la somme des pertes pénalisées des deux nouvelles feuilles soit inférieure à la perte de la feuille initiale, c'est-à-dire :

$$\text{Loss}_{\text{Feuille Gauche}} + \text{Loss}_{\text{Feuille Droite}} < \text{Loss}_{\text{Feuille Initiale}}$$

L'expression de la perte totale pour un arbre construit à l'itération m, notée $\mathcal{L}^{(m)}$, est donnée par :

$$\mathcal{L}^{(m)} = -\frac{1}{2}\sum_{j=1}^{J}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma J$$

Pour une feuille unique (avant split), la perte est donc :

$$-\frac{1}{2}\frac{\left(\sum_{i\in I_{\text{FI}}} g_i\right)^2}{\sum_{i\in I_{\text{FI}}} h_i + \lambda} + \gamma$$

où :

— I_{FI} représente les indices des observations dans la feuille initiale,
— I_{FG} et I_{FD} pour les feuilles gauche et droite.

Ainsi, le split est bénéfique si :

$$-\frac{1}{2}\frac{\left(\sum_{i\in I_{\text{FG}}} g_i\right)^2}{\sum_{i\in I_{\text{FG}}} h_i + \lambda} - \frac{1}{2}\frac{\left(\sum_{i\in I_{\text{FD}}} g_i\right)^2}{\sum_{i\in I_{\text{FD}}} h_i + \lambda} + 2\gamma < -\frac{1}{2}\frac{\left(\sum_{i\in I_{\text{FI}}} g_i\right)^2}{\sum_{i\in I_{\text{FI}}} h_i + \lambda} + \gamma$$

Ce qui peut se réécrire en simplifiant :

$$\frac{\left(\sum_{i\in I_{\text{FG}}} g_i\right)^2}{\sum_{i\in I_{\text{FG}}} h_i + \lambda} + \frac{\left(\sum_{i\in I_{\text{FD}}} g_i\right)^2}{\sum_{i\in I_{\text{FD}}} h_i + \lambda} - \frac{\left(\sum_{i\in I_{\text{FI}}} g_i\right)^2}{\sum_{i\in I_{\text{FI}}} h_i + \lambda} - 2\gamma > 0$$

Ainsi ce critère signifie qu'une séparation est avantageuse si elle réduit la perte globale, en tenant compte des pénalités de complexité. En pratique, XGBoost applique ce critère à chaque possible séparation d'une feuille, et choisit celle qui maximise ce gain de perte.

5.4.4 Parallélisation et Traitement des Valeurs Manquantes

XGBoost est conçu pour être rapide, même sur des ensembles de données volumineux, en tirant parti de la parallélisation. Les calculs de gradients et de hessiennes sont parallélisés, permettant une exécution plus rapide. De plus, XGBoost gère les **valeurs manquantes** de manière efficace, en décidant automatiquement de la meilleure façon de traiter ces valeurs lors de la construction des arbres.

5.5 Synthèse

En conclusion, XGBoost améliore le Gradient Boosting par l'introduction de la régularisation, de la descente de gradient second ordre, de la parallélisation, et d'optimisations sophistiquées dans la recherche des splits. Ces améliorations permettent à XGBoost de surpasser les méthodes traditionnelles en termes de rapidité et de performance, ce qui en fait un choix privilégié pour des tâches complexes de régression et de classification.

5.6 Application numérique dans le cadre d'une régression

Soit le jeu de données suivant, qui mesure l'efficacité d'un médicament en fonction de son dosage. Soit x le dosage et y la mesure de l'efficacité. L'objectif est de construire une suite d'arbres pour prédire en fonction de x la valeur de y :

X (dosage)	10	20	25	35
Y (efficacité)	-10	7	8	-7

Ce jeu de données est simpliste : 4 observations et 1 feature. Cela permettra d'illustrer parfaitement l'algorithme XGBoost pour la régression.

Étapes de l'algorithme

— Étape 0 : 1er estimateur constant $\hat{y}^{(0)}$
— Étape 1 : calcul des résidus $y_i - \hat{y}^{(0)}$
— Étape 2 : construction du premier arbre
  — identification du split optimal
  — calcul des estimations des feuilles
  — itération jusqu'à l'absence de split possible
— Étape 3 : calcul des résidus pour l'arbre suivant

Étape 0 : initialisation

— On prend $\hat{y}^{(0)} = 0.5$

Étape 1 : calcul des résidus

X (dosage)	10	20	25	35
Y (efficacité)	-10	7	8	-7
1^{er} est. $\hat{y}^{(0)}$	0.5	0.5	0.5	0.5
Résidus	-10.5	6.5	7.5	-7.5

Étape 2 : construction de l'arbre

Dans le cadre de la régression, nous choisissons d'utiliser la fonction de perte quadratique classique, définie comme suit :

$$L(y_i, \hat{y}_i) = \frac{1}{2} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

où $\hat{y}_i$ est la prédiction du modèle pour l'observation i, et y_i est la valeur réelle.

Calcul du gradient

On souhaite dériver la fonction de perte par rapport à $\hat{y}_i$ pour obtenir le gradient :

$$g_i = \frac{\partial}{\partial \hat{y}_i} L(y_i, \hat{y}_i) = \frac{\partial}{\partial \hat{y}_i} \left(\frac{1}{2} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \right) = (\hat{y}_i - y_i)$$

Ainsi, on obtient :

$$g_i = \hat{y}_i - y_i$$

Calcul du hessien

On dérive à nouveau pour obtenir la dérivée seconde :

$$h_i = \frac{\partial^2}{\partial \hat{y}_i^2} L(y_i, \hat{y}_i) = \frac{\partial}{\partial \hat{y}_i}(\hat{y}_i - y_i) = 1$$

Le hessien est donc constant :

$$h_i = 1 \quad \text{(valeur constante dans le cas quadratique)}$$

Formule du score :

$$\text{Score} = \frac{(\sum g_i)^2}{\sum h_i + \lambda} + \frac{(\sum g_i)^2}{\sum h_i + \lambda} - \frac{(\sum g_i)^2}{\sum h_i + \lambda} - 2\gamma$$

Cas 1 : $\lambda = 0$, $\gamma = 0$.

Split du 1er niveau de l'arbre : comparons le score de chacun des splits. On a 4 valeurs dans cet exemple, donc 3 splits possibles.

	Split 1.1 (x < 15)	**Split 1.2 (x < 22.5)**	**Split 1.3 (x < 30)**
Feuille initiale – Résidus	[-10.5, 6.5, 7.5, -7.5]	[-10.5, 6.5, 7.5, -7.5]	[-10.5, 6.5, 7.5, -7.5]
Feuille initiale – Score ($\lambda = 0$)	$\frac{(-10.5+6.5+7.5-7.5)^2}{4} = 4$		
Feuille gauche – Résidus	[-10.5]	[-10.5, 6.5]	[-10.5, 6.5, 7.5]
Feuille gauche – Score ($\lambda = 0$)	$\frac{(-10.5)^2}{1} = 110.25$	$\frac{(-10.5+6.5)^2}{2} = 8$	$\frac{(-10.5+6.5+7.5)^2}{3} = 4.08$
Feuille droite – Résidus	[6.5, 7.5, -7.5]	[7.5, -7.5]	[-7.5]
Feuille droite – Score ($\lambda = 0$)	$\frac{(6.5+7.5-7.5)^2}{3} = 14.08$	$\frac{(7.5-7.5)^2}{2} = 0$	$\frac{(-7.5)^2}{1} = 56.25$
Total Score ($\lambda = 0, \gamma = 0$)	$110.25 + 14.08 - 4 = 120.33$	$8 + 0 - 4 = 4$	$4.08 + 56.25 - 4 = 56.33$

TABLE 1 – Comparaison des splits du premier niveau de l'arbre ($\lambda = 0, \gamma = 0$)

Le split 1.1 (x<15) a le plus fort score, il est donc conservé dans la construction de l'arbre au premier niveau

Maintenant nous allons spliter sur la feuille de droite.

On split la feuille de droite.

	Split 2.1 (x < 22.5)	**Split 2.2 (x < 30)**
Feuille initiale – Résidus	[6.5, 7.5, -7.5]	[6.5, 7.5, -7.5]
Feuille initiale – Score ($\lambda = 0$)	$\frac{(6.5+7.5-7.5)^2}{3} = 14.08$	
Feuille gauche - Résidus	[6.5]	[-6.5, 7.5]
Feuille gauche – Score ($\lambda = 0$)	$\frac{(6.5)^2}{1} = 42.25$	$\frac{(6.5+7.5)^2}{2} = 98$
Feuille droite - Résidus	[7.5, -7.5]	[-7.5]
Feuille droite – Score ($\lambda = 0$)	$\frac{(7.5-7.5)^2}{2} = 0$	$\frac{(-7.5)^2}{1} = 56.25$
Total Score ($\lambda = 0, \gamma = 0$)	$42.25 + 0 - 14.08 = 28.17$	$98 + 56.25 - 14.08 = 140.17$

TABLE 2 – Scores des splits du deuxième niveau de l'arbre (cas $\lambda = 0, \gamma = 0$)

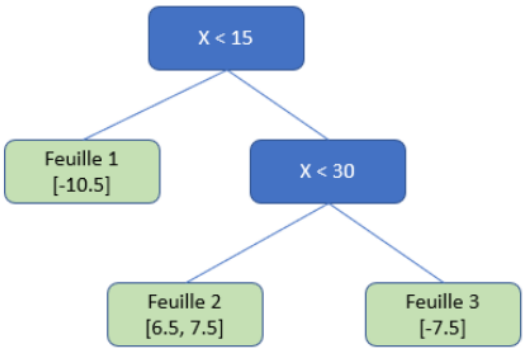

FIGURE 10 – Arbre obtenu

Le split 2.2 (x<30) a le plus fort gain, il est donc conservé dans la construction de l'arbre On suppose maintenant que notre arbre est complètement construit, on calcule maintenant les estimations optimales par feuille. On se sert des résidus restant par feuille

Calcul des sorties par feuille

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

Feuille	Résidus	Output ($\lambda = 0$)
Feuille 1	[-10.5]	-10.5
Feuille 2	[6.5, 7.5]	7
Feuille 3	[-7.5]	-7.5

Nouvelle estimation :

$$\hat{y}^{(1)} = \hat{y}^{(0)} + f_1(x_i)$$

X	10	20	25	35
Y	-10	7	8	-7
$\hat{y}^{(0)}$	0.5	0.5	0.5	0.5
Résidu	-10.5	6.5	7.5	-7.5
$f_1(x_i)$	-10.5	7	7	-7.5
$\hat{y}^{(1)}$	-10	7.5	7.5	-7

Cas 2 : $\lambda = 1$, $\gamma = 0$

Split du 1er niveau de l'arbre :

Split du 1er niveau de l'arbre	**Split 1.1** ($x < 15$)	**Split 1.2** ($x < 22.5$)	**Split 1.3** ($x < 30$)
Feuille initiale – Résidus	$[-10.5, 6.5, 7.5, -7.5]$	$[-10.5, 6.5, 7.5, -7.5]$	$[-10.5, 6.5, 7.5, -7.5]$
Feuille initiale – Score ($\lambda = 1$)	$\frac{(-10.5+6.5+7.5-7.5)^2}{4+1} = 3.2$		
Feuille gauche – Résidus	$[-10.5]$	$[-10.5, 6.5]$	$[-10.5, 6.5, 7.5]$
Feuille gauche – Score ($\lambda = 1$)	$\frac{(-10.5)^2}{2} = 55.12$	$\frac{(-10.5+6.5)^2}{3} = 5.33$	$\frac{(-10.5+6.5+7.5)^2}{4} = 3.06$
Feuille droite – Résidus	$[6.5, 7.5, -7.5]$	$[7.5, -7.5]$	$[-7.5]$
Feuille droite – Score ($\lambda = 1$)	$\frac{(6.5+7.5-7.5)^2}{4} = 10.56$	$\frac{(7.5-7.5)^2}{3} = 0$	$\frac{(-7.5)^2}{2} = 28.1$
Total Score ($\lambda = 1, \gamma = 0$)	$55.12 + 10.56 - 3.2 = 62.48$	$5.33 - 3.2 = 2.13$	$3.06 + 28.1 - 3.2 = 27.96$

TABLE 3 – Comparaison des splits au 1er niveau de l'arbre avec pénalisation $\lambda = 1$

On voit que c'est toujours le split 1.1 (x<15) qui a le plus fort score, il est donc conservé dans la construction de l'arbre au premier niveau. Par contre, l'ajout d'une pénalisation λ= 1 abaisse fortement les scores, ce qui amène à des arbres beaucoup moins profonds que sans pénalisation. C'est bien le but recherché : avoir des arbres moins profonds permettant de limiter les risques de sur apprentissage.

Nous obtenons les mêmes splits que pour le premier cas. Maintenant en calculant les sorties par feuille on obtient :

Feuille	**Résidus**	**Output** ($\lambda = 0$)	**Output** ($\lambda = 1$)
Feuille 1	$[-10.5]$	$\frac{-10.5}{1} = -10.5$	$\frac{-10.5}{2} = -5.25$
Feuille 2	[6.5, 7.5]	$\frac{6.5+7.5}{2} = 7$	$\frac{6.5+7.5}{3} = 4.6$
Feuille 3	$[-7.5]$	$\frac{-7.5}{1} = -7.5$	$\frac{-7.5}{2} = -3.75$

TABLE 4 – Estimations optimales par feuille pour $\lambda = 0$ et $\lambda = 1$

Nouvelle estimation :

$$\hat{y}^{(1)} = \hat{y}^{(0)} + f_1(x_i)$$

X	10	20	25	35
Y	-10	7	8	-7
$\hat{y}^{(0)}$	0.5	0.5	0.5	0.5
Résidu	-10.5	6.5	7.5	-7.5
$f_1(x_i)$	-5.25	4.6	4.6	3.75
$\hat{y}^{(1)}$	-4.75	5.1	5.1	-3.25

Cas 3 : $\gamma > 0$

$$\text{Score} = \text{Score}_{FG} + \text{Score}_{FD} - \text{Score}_{FI} - 2\gamma$$

Avec $\gamma > 0$ cela réduit les possibilités qu'un split soit valide (il faut que le score soit positif). Et donc par conséquent, cela limite l profondeur des arbres, et donc les risques de sur-apprentissage.

6 Simulation

6.1 Cadre de la simulation

Nous allons maintenant présenter la simulation que nous avons mise en place afin de pouvoir évaluer les performances du modèle CART et des algorithmes **AdaBoost** et **XGBoost** dans le cadre de la détection des maladies cardiaques grâce à différentes variables.
Notre simulation repose sur un jeu de données regroupant plus de 900 cas comportant une maladie ou non. Pour cela nous avons mis en place une classification supervisée sur la présence ou non de maladie cardiaque. Pour ce qui est de nos variables explicatives nous avons :

- des variables biologiques avec l'âge du patient et son sexe (encodé : 1=homme et 0=femme)
- des variables cliniques avec la pression artérielle(en mm Hg) au repos, le taux de cholestérol ((en mg/dL), la glycémie à jeun (encodé 1 si $>$ 120 mg/dL, sinon 0), la fréquence cardiaque maximale pendant un effort et la dépression du segment ST induite par l'exercice (Le segment ST est une partie de l'électrocardiogramme qui représente la période entre la dépolarisation et la repolarisation des ventricules du cœur. Ainsi une dépression de ce segment le muscle cardiaque ne reçoit pas assez d'oxygène à ce moment-là)
- une variable de douleur à l'effort avec la présence ou non d'angine de poitrine provoquée par l'effort.
- des variables analysant les douleurs thoraciques (angineuse ou non)
- une variable d'analyse des électrocardiogrammes des patients
- des variables analysant la pente du segment ST (plate ou montante)

Nous avons ensuite entraîné nos trois modèles de prédiction sur ces variables à travers un code Python (cf Annexe) et affiché les métriques de prédiction de nos modèles ainsi que les matrices de confusion.

Rappel sur les métriques d'évaluation

Pour évaluer les performances de nos modèles de classification, nous utilisons les métriques suivantes :

- **Précision** : proportion de vraies prédictions positives parmi toutes les prédictions positives.

$$\text{Précision} = \frac{VP}{VP + FP}$$

  où VP est le nombre de vrais positifs et FP le nombre de faux positifs.
- **Rappel** : proportion de vraies prédictions positives parmi toutes les instances positives réelles.

$$\text{Rappel} = \frac{VP}{VP + FN}$$

  où FN est le nombre de faux négatifs.
- **F1-score** : moyenne harmonique entre la précision et le rappel, utile en cas de classes déséquilibrées.

$$F1 = 2 \times \frac{\text{Précision} \times \text{Rappel}}{\text{Précision} + \text{Rappel}}$$

6.2 Comportement modèle CART

Le modèle a été testé sur 368 données (40% de test et 60% d'entraînement) et voici ces métriques de performance par classe :

Classe	Précision	Rappel	F1-score	Nombre d'exemples
0 (pas de maladie)	0.76	0.75	0.76	173
1 (maladie cardiaque)	0.78	0.79	0.79	195

Table 5 – Métriques de performance du modèle CART par classe

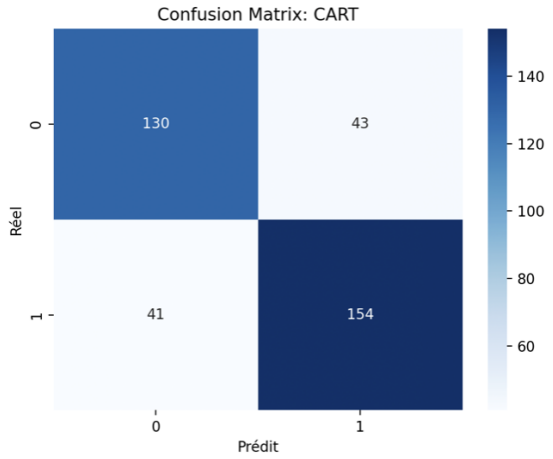

FIGURE 11 – Matrice de confusion du modèle CART

Le modèle parvient à équilibrer correctement les prédictions entre les deux classes. Le nombre d'erreurs de prédiction reste relativement similaire pour chaque classe : on observe 41 faux négatifs et 43 faux positifs. Cela montre que le modèle ne favorise pas une classe au détriment de l'autre. Le F1-score moyen atteint 0,77, ce qui reflète une performance globale raisonnable et équilibrée.

6.3 Comportement d'AdaBoost

Comme pour le modèle **CART**, **AdaBoost** a été testé sur 368 données (40% de test et 60% d'entraînement) et voici ces métriques de performance par classe :

Classe (réelle)	Précision	Rappel	F1-score	Nombre d'exemples
0 (pas de maladie)	0.82	0.83	0.82	173
1 (maladie cardiaque)	0.85	0.84	0.84	195

TABLE 6 – Métriques de performance du modèle AdaBoost par classe

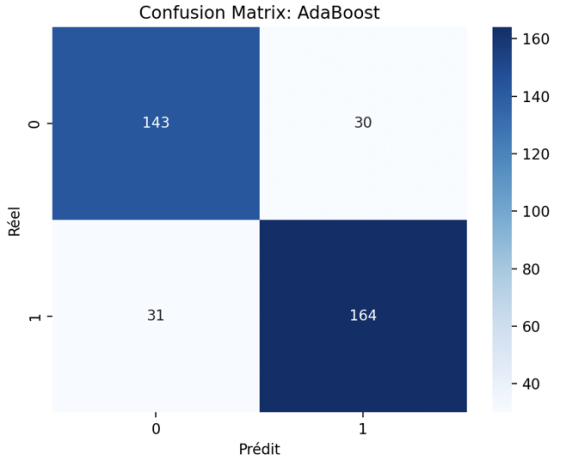

FIGURE 12 – Matrice de confusion d'AdaBoost

Bien que l'on observe un léger déséquilibre entre les prédictions des deux classes, AdaBoost montre de meilleures performances que CART. Il obtient une précision de 0,82 pour la classe 0 et de 0,85 pour la classe 1. La matrice de confusion indique une réduction du nombre d'erreurs : 30 faux positifs et 31 faux négatifs. Globalement, le F1-score moyen de 0,83 témoigne d'une amélioration par rapport au modèle CART, notamment en termes de précision et de rappel pour chaque classe.

6.4 Comportement de XGBoost

Comme pour les deux précédents cas, **XGBoost** a été testé sur 368 données (40% de test et 60% d'entraînement) et voici ces métriques de performance par classe :

Classe (réelle)	Précision	Rappel	F1-score	Nombre d'exemples
0 (pas de maladie)	0.86	0.82	0.84	173
1 (maladie cardiaque)	0.85	0.88	0.86	195

TABLE 7 – Métriques de performance du modèle XGBoost par classe

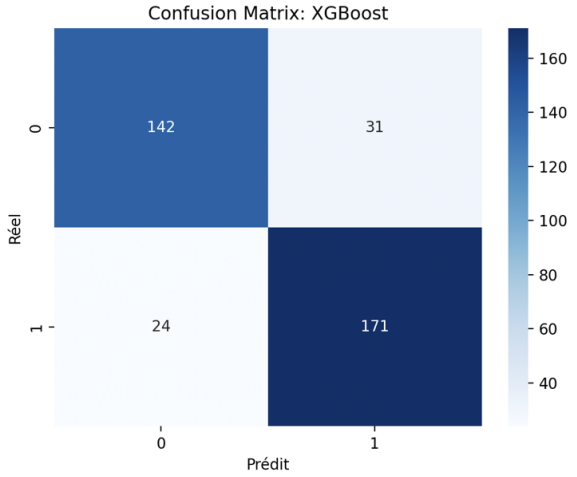

FIGURE 13 – Matrice de confusion pour XGBoost

Nous remarquons que le modèle XGBoost affiche des performances légèrement supérieures à **AdaBoost**, notamment avec un F1-score global de 0,85. Ce bon résultat s'explique par un équilibre efficace entre la précision et le rappel pour chaque classe.

Pour la classe 0 (absence de maladie), la précision est de 0,86 et le rappel de 0,82, ce qui traduit une bonne capacité du modèle à éviter les faux positifs. Pour la classe 1 (présence d'une maladie cardiaque), le rappel est élevé à 0,88, ce qui signifie que XGBoost détecte efficacement les cas de patients malades, ce qui est crucial dans un contexte médical.

La matrice de confusion montre que le modèle commet 24 faux négatifs et 31 faux positifs, ce qui est meilleur que le modèle AdaBoost, notamment sur les faux négatifs, ce qui diminue le risque de ne pas diagnostiquer une maladie existante.

6.5 Comparaison globale

Après avoir analysé séparément les performances de chaque modèle (**CART**, **AdaBoost** et **XGBoost**), voyons ici une synthèse comparative. Les principales métriques de classification (précision, rappel, F1-score et exactitude) sont rassemblées dans le tableau suivant :

Modèle	Précision	Rappel	F1-Score	Accuracy
CART	0.77	0.77	0.77	0.77
AdaBoost	0.83	0.83	0.83	0.83
XGBoost	0.85	0.85	0.85	0.85

TABLE 8 – Résultats des modèles sur la prédiction de la maladie cardiaque

L'évaluation conjointe des trois modèles, CART, AdaBoost et XGBoost, met en évidence des différences notables en termes de performances, de stabilité et de pertinence dans un contexte de prédiction médicale.

Le modèle CART, basé sur des arbres de décision simples, constitue une solution rapide et interprétable. Toutefois, ses performances restent limitées avec un F1-score de 0,77. Il tend à générer davantage de faux positifs et de faux négatifs, ce qui peut poser problème dans une tâche sensible comme la détection de maladies.

L'introduction du boosting avec AdaBoost permet une nette amélioration. En combinant plusieurs arbres faibles, le modèle parvient à corriger progressivement les erreurs du modèle de base. Cette approche se traduit par une hausse significative des performances (F1-score de 0,83) et une réduction des erreurs critiques. AdaBoost montre ainsi sa capacité à mieux généraliser sans surapprentissage excessif.

Enfin, XGBoost pousse cette logique encore plus loin, grâce à une régularisation fine et une optimisation efficace de la fonction objective. Il affiche les meilleures performances (F1-score de 0,85) tout en maintenant un bon équilibre entre précision et rappel. De plus, il parvient à réduire davantage les faux négatifs, ce qui est essentiel dans le cadre d'un diagnostic médical où les erreurs d'omission peuvent avoir des conséquences graves.

En croisant les résultats, on observe une progression cohérente : chaque modèle apporte un gain par rapport au précédent, aussi bien en précision qu'en robustesse. CART pose les bases, AdaBoost les améliore, et XGBoost les consolide. Ce constat justifie pleinement le recours aux modèles d'ensemble pour les problèmes de classification médicale, en particulier lorsque la sensibilité du modèle (rappel) est un enjeu majeur.

7 Conclusion

Ce travail de recherche a permis d'explorer en profondeur les fondements, les mécanismes et les performances des algorithmes de boosting appliqués à l'apprentissage supervisé, que ce soit dans un contexte de régression ou de classification.En partant du modèle de base CART, nous avons mis en évidence les limites des arbres de décision lorsqu'ils sont utilisés isolément, notamment leur sensibilité au surapprentissage et leur manque de robustesse sur des jeux de données complexes ou bruités.

L'introduction d'AdaBoost a marqué une avancée significative. En combinant plusieurs classificateurs faibles de manière séquentielle et pondérée, cet algorithme améliore considérablement la capacité de généralisation du modèle. Nous avons démontré que son fondement repose sur une interprétation probabiliste et une minimisation de la perte exponentielle. Toutefois, nous avons également mis en lumière certaines de ses limites, notamment sa sensibilité aux données aberrantes.

Pour pallier ces limites, XGBoost s'impose comme une solution de pointe. En capitalisant sur les principes du gradient boosting, tout en y ajoutant des techniques avancées telles que la régularisation, l'optimisation via un développement de Taylor et une gestion intelligente des données manquantes, XGBoost se démarque à la fois par sa performance statistique et son efficacité computationnelle. Les expérimentations menées dans le cadre de ce travail confirment la supériorité de ce modèle, notamment dans des contextes où la minimisation des erreurs critiques comme les faux négatifs est primordiale, à l'instar des applications médicales.

Il est important de souligner que ce travail ne s'est pas limité à une étude empirique des algorithmes. Il s'est également appuyé sur une formalisation théorique rigoureuse du boosting, en s'appuyant notamment sur le théorème de Schapire et la modélisation additive progressive. Ces fondements ont permis de mieux comprendre pourquoi les algorithmes comme AdaBoost ou XGBoost parviennent à améliorer la performance des classificateurs faibles, en concentrant progressivement l'apprentissage sur les erreurs commises par les modèles précédents. Cette articulation entre théorie et pratique a constitué l'un des fils conducteurs du mémoire.

Plus globalement, ce TER a été l'occasion de mettre en pratique les concepts théoriques de l'apprentissage statistique à travers une étude comparative rigoureuse et des implémentations concrètes. Il illustre de manière concrète comment les avancées algorithmiques récentes, notamment dans le domaine du boosting, permettent de relever efficacement les défis posés par la classification supervisée.

Annexe

```python
import numpy as np
import matplotlib.pyplot as plt

# Nous d finissons un vecteur de marges yf(x)
marge = np.linspace(-2, 2, 400)

# Nous d finissons les fonctions de perte
def perte_exponentielle(yf):
    return np.exp(-yf)

def perte_deviance(yf):
    return np.log(1 + np.exp(-2 * yf))

# Calcul des pertes
loss_exp = perte_exponentielle(marge)
loss_dev = perte_deviance(marge)

# Trac  des graphiques
plt.figure(figsize=(8, 5))
plt.plot(marge, loss_exp, label="Exponentielle", color="skyblue", linewidth=2)
plt.plot(marge, loss_dev, label="D viance binomiale", color="orange", linewidth=2)

# Mise en forme
plt.xlabel("Marge $yf(x)$")
plt.ylabel("Perte")
plt.title("Comparaison des fonctions de perte")
plt.legend()
plt.tight_layout()
plt.show()
```

Listing 1 – Code Python utilisé pour tracer les fonctions de perte exponentielle et déviance binomiale en fonction de la marge

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from xgboost import XGBClassifier
import matplotlib.pyplot as plt
import seaborn as sns

# Chargement du jeu de donn es
df = pd.read_csv("heart.csv")

print(df.columns.tolist())

# S lection manuelle des variables explicatives
features = [
    'Age',
    'RestingBP', # Pression art rielle au repos (mm/Hg)
    'Cholesterol', # Taux de cholest rol s rique (mg/dL)
    'FastingBS', # Glyc mie    jeun
    'MaxHR', # Fr quence cardiaque maximale    l'effort
    'Oldpeak', # D pression du segment ST (exercice)
    'Sex_M',
    'ChestPainType_ATA', #Douleur thoracique type ATA
    'ChestPainType_NAP', #Douleur thoracique type NAP
    'ChestPainType_TA', #Douleur thoracique type TA
    'RestingECG_Normal', #Electrocardiagramme normal
    'RestingECG_ST', #Electrocardiagramme avec anomalie
    'ExerciseAngina_Y', #Douleur angineuse pendant l'effort
    'ST_Slope_Flat', #Pente du segment ST plate
    'ST_Slope_Up' #Pente du segment ST ascendante
]

# X = variables explicatives, Y = cible (maladie cardiaque)
X = df[features]
```

```
Y = df['HeartDisease']

# S paration des donn es (60% entra nement , 40% test)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4, random_state
    =14)

# Fonction d' valuation
def eval_model(Y_true, Y_pred, title):
    print(f"--- {title} ---")
    print(classification_report(Y_true, Y_pred))
    sns.heatmap(confusion_matrix(Y_true, Y_pred), annot=True, fmt='d', cmap='Blues')
    plt.title(f"Confusion Matrix: {title}")
    plt.xlabel("Pr dit")
    plt.ylabel("R el")
    plt.show()

# Mod le CART
cart = DecisionTreeClassifier(random_state=14)
cart.fit(X_train, Y_train)
Y_pred_cart = cart.predict(X_test)
eval_model(Y_test, Y_pred_cart, "CART")

# Mod le AdaBoost
ada = AdaBoostClassifier(n_estimators=100, random_state=14)
ada.fit(X_train, Y_train)
Y_pred_ada = ada.predict(X_test)
eval_model(Y_test, Y_pred_ada, "AdaBoost")

# Mod le XGBoost
xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=14)
xgb.fit(X_train, Y_train)
Y_pred_xgb = xgb.predict(X_test)
eval_model(Y_test, Y_pred_xgb, "XGBoost")
```

Listing 2 – Code Python pour entraîner et évaluer les modèles CART, AdaBoost et XGBoost sur les données cardiaques

8 English version

9 Introduction

Statistical learning plays a central role in many fields, ranging from medicine to finance, including industry and social sciences. It is a set of methods that allow us to extract knowledge from data, in order to make predictions, classify observations, or detect hidden relationships between different variables.
The situations in which these techniques are used are extremely varied : estimating the future evolution of a stock price from economic indicators, or automatically recognizing handwritten digits in an image.

In all these cases, we have a set of observations called training data, containing both results (target variables) and associated characteristics (explanatory variables, or features). The goal is then to build a model, or learner, capable of generalizing to new situations by learning the regularities present in this data.

This is precisely what is called supervised learning, a form of statistical learning in which the target variable guides the training of the model.

A learning problem is said to be supervised when the training data includes, for each observation, the values of the explanatory variables as well as the value of the result to predict. The learning is thus guided by this target variable, and the model learns to establish a link between the inputs and the expected output. A classic example of a supervised learning problem is the detection of spam emails. In this case, each email represents an observation, and the goal is to predict whether it is spam or a legitimate message. To do this, we have a set of pre-labeled emails (we know for

each whether it is spam or not (targets)), as well as in our case the presence or absence of certain keywords.

	george	you	your	hp	free	hpl	!	our	re	edu	remove
spam	0.00	2.26	1.38	0.02	0.52	0.01	0.51	0.51	0.13	0.01	0.28
email	1.27	1.27	0.44	0.90	0.07	0.43	0.11	0.18	0.42	0.29	0.01

FIGURE 14 – This table shows the average percentage of appearance of certain words or characters in spam messages compared to normal (non-spam) emails. The displayed words and characters are those that show the greatest difference between the two types of emails

By analyzing the table, we see that the word *you* appears on average 2.26 times in a spam compared to 1.27 in a regular email.
Thus, our learning method must decide which character to use and how. For example, we have :

```
if (%george < 0.4) & (%you > 1.5) then spam
else email
```

Depending on the nature of the variable to predict, we distinguish two main categories of supervised tasks :

1. Classification : Predicting a label or a class among a finite set (for example : spam / not spam, sick / healthy).

2. Regression : Estimating a continuous numerical value (like the price of a good, a temperature, or a chemical concentration). In both cases, the objective is to develop a predictive model capable of exploiting the regularities contained in past data to make reliable predictions on new data.

Once the framework of supervised learning is set, it is essential to distinguish the different types of models that can be used for this task. We notably distinguish two main approaches : simple models and ensemble models.

A simple model relies on a single learning algorithm, applied directly to the training data. This is the case, for example, of linear regression, decision trees. These models are generally easy to implement, fast to train, and often interpretable, which makes them preferred tools for a first data exploration or for problems whose structure remains relatively simple. However, their ability to capture complex relationships or non-linear interactions between variables may be limited, which can hinder their performance on richer or noisier datasets. On the other hand, ensemble models rely on the combination of several base models, often called weak learners, in order to improve the robustness of the predictions. The idea is that, even if each model taken in isolation is imperfect, their aggregation makes it possible to reduce errors, limit overfitting, and improve generalization on new data. Among the main ensemble methods, we can cite :

1. Bagging, illustrated notably by **Random Forests**, which consists of training several models on random subsamples of the data, then aggregating their predictions.

2. Boosting, used in algorithms such as **AdaBoost**, **Gradient Boosting**, or **XGBoost**, which progressively corrects the errors of previous models by training them sequentially.

3. Stacking, which combines the predictions of several heterogeneous models using a meta-model in charge of learning how to optimally combine them.

Ensemble models are now widely used due to their excellent predictive performance. However, their main drawback lies in their complexity : they are generally longer to train, less interpretable, and require a more robust computing infrastructure.

10 The CART Model

10.1 Decision Trees

Before starting to focus on the **CART (Classification and Regression Trees)** model, it is essential to begin by presenting what decision trees are.

A decision tree is a supervised learning method whose main idea is to partition the space of explanatory variables into several subspaces and adjust a simple model in each of them (see figure 2). For example, in the case of a regression, we consider Y the endogenous variable, X_1 and X_2 the explanatory variables, and R_1, R_2, R_3 the subspaces of the partition. If $(X_1, X_2) \in R_n$ with $n = 1, 2, 3$, then we predict Y with a constant c_m which can for example be the mean of the Y in this region. The associated regression model will then be of the form :

$\hat{f}(X) = \sum_{n=1}^{3} c_m \, \mathbf{1}_{\{(X_1,X_2)\in R_m\}}$. The partitioning of this space can be more or less difficult to describe, however in the context of our study, we will limit ourselves to recursive binary partitions used for the CART model.

This type of partitioning can be represented by a binary tree where at each node, we have two branches, the data is directed towards one of the two branches depending on the decision rule chosen at each node. Each path of the tree results in a leaf giving a final prediction.

The main advantage of using recursive binary trees is their interpretability. Indeed, one can describe the partition in the form of a single tree while when we have more input variables the representation can be much more difficult.

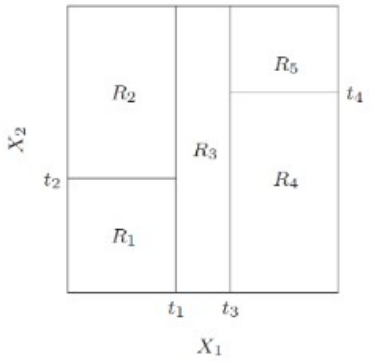

FIGURE 15 – Partitioning of the explanatory variable space composed of X_1 and X_2

10.2 Presentation of the CART model

The **CART** model is a supervised learning method used for classification and regression tasks. It allows us to explain how a variable can be explained based on other factors. We will present here the two categories of decision trees, namely regression and classification trees.

10.2.1 Regression Trees

We will consider in this part Y a variable to predict, p endogenous variables $X_1, ..., X_p$ and N observations (x_i, y_i) with $i = 1, ..., N$. A first method that can be considered to find the best possible prediction of Y is to choose the partition that minimizes the empirical risk.
If we assume that the variable space is divided into 2 partitions R_1, R_2 for simplicity and that the prediction of Y is c_m in region m with $m \in \{1, 2\}$. The model is therefore of the form :

$$f(x) = c_1 \, \mathbb{1}_{\{x\in R_1\}} + c_2 \, \mathbb{1}_{\{x\in R_2\}}$$

We then want to minimize :

$$\text{SSE}(R_1, R_2) = \sum_{x_i \in R_1} \big(y_i - f(x_i)\big)^2 + \sum_{x_i \in R_2} \big(y_i - f(x_i)\big)^2.$$

Moreover, we determine the estimators $\hat{c}_m$ of c_m with $m = 1, 2$ by minimizing :

$$\sum_{x_i \in R_m} (y_i - f(x_i))^2$$

We consider that $\forall x_i \in R_m$ we have $f(x_i) = c$. So we have $Q_m(c) = \sum_{x_i \in R_m} (y_i - c)^2$. We will find the minimum by solving :

$$\frac{dQ_m(c)}{dc} = 0$$

By expanding, we have :

$$Q_m(c) = \sum_{x_i \in R_m} (y_i - c)^2 = \sum_{x_i \in R_m} (y_i^2 - 2y_i c + c^2) = \sum_{x_i \in R_m} y_i^2 + 2c \sum_{x_i \in R_m} y_i + c^2 N_m$$

where $N_m = card(x_i \in R_m)$. So by deriving the expression we obtain :

$$\frac{dQ_m(c)}{dc} = -2 \sum_{x_i \in R_m} y_i + 2cN_m = 0$$

then isolating c, we finally find that the optimal value of c is the average of the y_i in region R_m :

$$\boxed{\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i} \tag{61}$$

We verify that this is indeed a minimum by showing that Q_m is convex :

$$\frac{d^2 Q_m(c)}{dc^2} = 2N_n > 0.$$

However, it appears that this method is unfeasible because it is too computationally expensive. Indeed, it involves testing all possible ways to split the space, i.e. 2^N possibilities in the case where we want to split the space into 2 partitions.

Determination of variables and split thresholds :
A solution to this problem lies in the use of the **CART** model. The idea behind this algorithm is that we seek to predict Y by determining splitting variables X_j, the splitting points, and the shape that the tree should have. At each step, the algorithm chooses the variable and the splitting point that minimize an error criterion. This process is repeated until a stopping criterion is reached.

We thus consider two half-planes :

$$R_1(j, s) = \{X \mid X_j < s\} \quad \text{and} \quad R_2(j, s) = \{X \mid X_j > s\}. \tag{62}$$

For a given step, we will search for the splitting variable j and the splitting point s that solve :

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right] \tag{63}$$

Let's begin by minimizing at first the function :

$$L(c_1) = \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2$$

This function being convex, we obtain a minimum by solving the equation :

$$\frac{d}{dc_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 = 0$$

After expanding and deriving, we obtain :

$$\sum_{x_i \in R_1(j,s)} y_i + 2c_1 \sum_{x_i \in R_1(j,s)} 1 = 0$$

Finally, we get :

$$c_1 = \frac{1}{|R_1|} \sum_{x_i \in R_1(j,s)} y_i \tag{64}$$

Which corresponds to the average of the y_i in R_1 :

$$\hat{c}_1 = \text{mean}(y_i | x_i \in R_1(j,s)) \tag{65}$$

We also minimize :

$$L(c_2) = \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2$$

By a similar reasoning, we also find :

$$\hat{c}_2 = \text{mean}(y_i \mid x_i \in R_2(j,s)) \tag{66}$$

These values $\hat{c}_1$ and $\hat{c}_2$ are computed for each pair (j, s), then the sum of squared errors is evaluated :

$$\sum_{x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2$$

We choose the pair (j, s) that gives the smallest sum of squared errors to perform the first split of the tree. This process is then repeated on all resulting regions.

Determining the Optimal Tree Size :

The second crucial element in constructing a regression tree is determining its size. Indeed, once the tree, denoted T_0, has been built using the method previously described, it may be too large, leading to overfitting : the tree will perform very well on the training data but generalize poorly to new data because it is too specific. To remedy this, we aim to find an optimal tree size.

A first idea would be to split a node only if the sum of squared residuals at that node exceeds a predefined threshold. However, this approach may not be suitable since, for a given node, the sum of squared residuals might exceed the threshold, yet further splits could result in much lower residual errors.

Instead, we prune the tree using the **cost-complexity pruning** method. We consider a tree $T \subset T_0$ obtained by pruning T_0. Each terminal node is indexed by m. Let $|T|$ be the number of terminal nodes in T, $N_m = \text{card}(x_i \in R_m)$, and $\hat{c}_m$ be defined as before. The mean squared error over R_m is given by :

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2 \tag{67}$$

The **cost-complexity criterion** is defined as :

$$\boxed{C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|} \tag{68}$$

For a given $\alpha \geq 0$, the optimal tree T_α is the one that minimizes this criterion :

$$T_\alpha = \arg\min_T C_\alpha(T) \tag{69}$$

We must choose an α that achieves a trade-off between tree size and goodness of fit. If α is large, larger trees are penalized more, and T_α will be small. If α is small, fitting quality is prioritized and more nodes are retained.

10.2.2 Classification Trees

In many situations, the response variable Y is not numerical but categorical : sick or not, loyal or at-risk customer, spam or legitimate message. These problems fall under classification. We assume Y takes values in $\{1, ..., K\}$, representing the different classes.

The principle is the same as before : we begin by growing a large tree T_0 using a stopping criterion such as a minimum number of observations in a node. Another stopping rule may also be used.
For each input variable $X_1, ..., X_p$, all possible split thresholds are tested. The thresholds are determined by first sorting the observations for a given explanatory variable, then computing the average between each pair of consecutive observations. If there are N observations, this gives $N - 1$ possible split points for each variable.

We now introduce some new concepts. Consider a node m corresponding to a region R_m with N_m observations. The **proportion of class k observations in node** m is defined as :

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} \mathbf{1}(y_i = k) \tag{70}$$

A node is said to be impure if it does not contain only observations from a single class. We define the **majority class in node** m as :

$$k(m) = \arg\max_k \hat{p}_{mk} \tag{71}$$

The objective at each node is to measure impurity and find the variable and threshold that minimize the total impurity (defined below). The two commonly used impurity measures are the Gini index and cross-entropy. They are respectively defined as :

$$\text{Gini index} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{72}$$

and

$$\text{Cross entropy} = -\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk}) \tag{73}$$

We use one of these two measures to evaluate the **overall impurity** of a split, and choose the split that minimizes it :

$$\text{Overall Impurity} = \frac{N_L}{N} Q_L + \frac{N_R}{N} Q_R \tag{74}$$

where N_L and N_R are the number of observations in the left and right sub-nodes respectively, and N is the total number of observations in both sub-nodes.

We recursively repeat this process in each sub-node until a stopping criterion is met. One such criterion was discussed above. However, another commonly used criterion in classification is the impurity gain meaning we stop the split if it does not significantly reduce the overall impurity.

Once the tree has been constructed, we prune it as before, by finding the tree T that minimizes the cost-complexity criterion. This time, $Q_m(T)$ corresponds to the impurity measure at node m.

11 Boosting Theory

11.1 Motivation

Decision trees for regression and classification allow for fairly intuitive predictions of the target variable Y. However, when taken individually, these models present several limitations. One major limitation is that they are sensitive to variations in the data. Even small changes in the training set can result in significantly different tree structures. This can lead to high prediction variance, undermining robustness and limiting the model's ability to generalize.

Moreover, when decision trees are deep, the number of possible partitions grows exponentially. For a predictor with q possible values, the number of possible partitions is 2^q, which can lead to computational difficulties and model overfitting.

We now present a method that avoids using a single complex tree by combining multiple shallow trees to obtain a more stable and often more robust global learner.

11.2 Weak Classifier and Schapire's Theorem

In this section, we focus on the foundation of boosting : weak classifiers. These are the classifiers that will be combined to form a more powerful global learner. To show this is feasible, we state the **Schapire's Theorem** and outline a sketch of its proof.

Définition 5. *A weak classifier is a learning algorithm that performs only slightly better than random guessing. That is, its accuracy exceeds* 50% *in a binary classification problem. Weak classifiers are often small, shallow CART trees.*

To state Schapire's Theorem, we first introduce the necessary theoretical framework. We adopt the PAC learning model introduced by Valiant in 1984. In this framework, a **concept** c is a Borel function $c : X \to \{0, 1\}$ assigning a label to each data point.

Furthermore, a **concept class** C is a set of such functions. It can be decomposed into subclasses C_n such that each subclass operates over a data set X_n. Additionally, the learning algorithm does not have direct access to the function c, but only to the training data. The algorithm receives data $x \in X_n$ drawn randomly according to an unknown distribution D, and for each data point, the correct label $c(x)$ is known, determined by the target concept. This represents the real-world scenario where the algorithm learns from a random sample without knowledge of the underlying data-generating distribution.

From the observed data, the algorithm produces a function $h : X_n \to \{0, 1\}$ called a hypothesis. This function is used to predict the labels of new data points. The quality of this hypothesis is measured by the error rate :

$$\mathrm{err}_D(h) = P(h(x) \neq c(x)) \tag{75}$$

When this error is below $\epsilon > 0$, we say that h is ϵ-close to the target concept. We now define strong and weak learners more formally.

Définition 6. *A concept class C is said to be strongly learnable if there exists an algorithm such that, for any concept $c \in C$, any distribution D, and for any error threshold ϵ and confidence level δ, it produces, with probability at least $1 - \delta$, a hypothesis h satisfying :*

$$err_D(h) \leq \epsilon$$

Définition 7. *An algorithm is said to be weakly learnable if it can produce a hypothesis that performs slightly better than random guessing :*

$$err_D(h) \leq \frac{1}{2} - \gamma$$

The final key concept in this framework is the **example oracle EX**.

Définition 8. *An example oracle for a concept c over domain X_n, with respect to a distribution D, is a procedure that, upon each call, draws an instance $x \in X_n$ from D and returns the pair $(x, c(x))$, that is, the instance and its correct label according to the target concept c.*

The example oracle thus simulates the generation of training data.

With this theoretical foundation in place, we can now state Schapire's Theorem :

Théorème 3. *Any concept class that is weakly learnable is also strongly learnable. More precisely, if a learning algorithm produces a hypothesis with error less than* $\frac{1}{2} - \gamma$ *for some constant* $\gamma > 0$*, then it is possible to construct from it a new algorithm that yields a hypothesis with arbitrarily small error.*

Sketch of Proof. The general idea of Schapire's proof is to build an algorithm called **Learn**, which simulates a weak algorithm A three times on modified distributions of the training data.

```
Learn(ε, δ, EX)

Input:   error parameter ε
         confidence parameter δ
         examples oracle EX
         (implicit) size parameters s and n

Return:  hypothesis h that is ε-close to the target concept c with probability ≥ 1 − δ

Procedure:
  if ε ≥ 1/2 − 1/p(n, s) then return WeakLearn(δ, EX)
  α ← g^{-1}(ε)

  EX_1 ← EX
  h_1 ← Learn(α, δ/5, EX_1)
  τ_1 ← ε/3
  let â_1 be an estimate of a_1 = Pr_{v∈D}[h_1(v) ≠ c(v)]:
     choose a sample sufficiently large that |a_1 − â_1| ≤ τ_1 with probability ≥ 1 − δ/5
  if â_1 ≤ ε − τ_1 then return h_1

  defun EX_2()
     { flip coin
       if heads, return the first instance v from EX for which h_1(v) = c(v)
       else      return the first instance v from EX for which h_1(v) ≠ c(v) }
  h_2 ← Learn(α, δ/5, EX_2)
  τ_2 ← (1 − 2α)ε/8
  let ê be an estimate of e = Pr_{v∈D}[h_2(v) ≠ c(v)]:
     choose a sample sufficiently large that |e − ê| ≤ τ_2 with probability ≥ 1 − δ/5
  if ê ≤ ε − τ_2 then return h_2

  defun EX_3()
     { return the first instance v from EX for which h_1(v) ≠ h_2(v) }
  h_3 ← Learn(α, δ/5, EX_3)

  defun h(v)
     { b_1 ← h_1(v), b_2 ← h_2(v)
       if b_1 = b_2 then return b_1
       else              return h_3(v) }
  return h
```

FIGURE 16 – Learn Algorithm from "The Strength of Weak Learnability" (Schapire, 1990), p.7

We start by fixing the error α such that $g(\alpha) = \epsilon$, where $g(\alpha) = 3\alpha^2 - 2\alpha^3$. We call Learn$(\alpha, \frac{\delta}{5}, \text{EX}_1)$ and compute an estimator $\hat{a}_1$ for $a_1 = P(h_1(v) \neq c(v))$ which is sufficiently close to a_1 with probability at least $1 - \frac{\delta}{5}$. If $\hat{a}_1 \leq \epsilon$, we return h_1, indicating a strong learner.

Otherwise, we construct a distribution EX_2 that gives a 50% chance of selecting an example correctly classified by h_1 and 50% for an incorrectly classified one. This forces the next call to focus equally on h_1's errors and successes. We then call Learn$(\alpha, \frac{\delta}{5}, \text{EX}_2)$ to obtain a second hypothesis h_2. If h_2 is good enough, we return it. Otherwise, we construct a third distribution EX_3 that selects only examples where $h_1 \neq h_2$ and train a third hypothesis using Learn$(\alpha, \frac{\delta}{5}, \text{EX}_3)$. If $h_1 = h_2$, we return h_1, otherwise we return h_3.

Note : The use of $\frac{\delta}{5}$ instead of δ in recursive calls to Learn is due to the five possible failure points in the algorithm : the recursive call for h_1, the statistical estimate of error $\hat{a}_1$, the recursive call for h_2, the estimator for $\hat{e}$, and the recursive call for h_3. If each has failure probability at most $\frac{\delta}{5}$, then the probability that at least one fails is at most $5 \cdot \frac{\delta}{5} = \delta$. Therefore, the probability of overall success is at least $1 - \delta$.

We now show the following result to complete the proof of Theorem 1 :

Théorème 4. *Let* $0 < \epsilon < \frac{1}{2}$ *and* $0 \leq \delta \leq 1$. *Then the algorithm Learn*(ϵ, δ, EX) *returns, with probability at least* $1 - \delta$, *a hypothesis* h *such that* $P(h(x) \neq c(x)) \leq \epsilon$.

Démonstration. As explained in the remark, the probability of a successful execution is at least $1-\delta$. It remains to show that if Learn executes successfully, then the output hypothesis is ϵ-close to the target concept.

First Case :
In the case where the estimator $\hat{a}_1$ or $\hat{e}$ is found to be smaller than $\epsilon - \tau_1$ or $\epsilon - \tau_2$ respectively, and $\hat{a}_1$ and $\hat{e}$ are sufficiently close to a_1 and e respectively, then the returned hypothesis is ϵ-close to the target concept.

General Case :
Let a_i be the error of h_i under the distribution D_i induced by the oracle EX_i at the i^{th} recursive call. By the induction hypothesis, we have for $i = 1, 2, 3$, $a_i \leq \alpha$ because Learn is called recursively with error parameter α. We introduce the following notations :

$$p_i(v) = P(h_i(v) \neq c(v)) \tag{76}$$

$$q(v) = P(h_1(v) \neq h_2(v)) \tag{77}$$

$$w = \underset{v \sim D}{P}(h_2(v) \neq h_1(v) = c) \tag{78}$$

$$x = \underset{v \sim D}{P}(h_1(v) = h_2(v) = c) \tag{79}$$

$$y = \underset{v \sim D}{P}(h_1(v) \neq h_2(v) = c) \tag{80}$$

$$z = \underset{v \sim D}{P}(h_1(v) = h_2(v) \neq c) \tag{81}$$

Note and Notation : The notation $\underset{v \sim D}{P}(\pi(v))$ denotes the probability that a predicate τ is satisfied over instances v drawn from X_n according to the distribution D.
X_n is the set of all possible instances for the learning problem.
Then, by the law of total probability, we have :

$$\underset{v \sim D}{P}(\tau(v)) = \sum_{v \in X_n} P(v)P(\tau(v)|v) = \sum_{v \in X_n} D(v)P(\tau(v)) \tag{82}$$

From now on, to simplify notation, P_D will denote $\underset{v \sim D}{P}$.

We now express the probability laws according to which the oracles EX_i generate the instances v. This will allow us to compute the final error of the hypothesis h constructed by Learn.
Let $D(v)$ be the probability that $v \in X_n$ is randomly drawn by the example oracle EX, then :

$$D(v) = D_1(v) \tag{83}$$

For the distribution D_2, the example oracle EX_2 is built by flipping a fair coin. If the coin lands heads, it returns the first instance v such that $h_1(v) = c(v)$. Otherwise, the algorithm returns the first instance v from EX such that $h_1(v) \neq c(v)$. Thus :

$$D_2(v) = \frac{1}{2}P(v|h_1(v) = c(v)) + \frac{1}{2}P(v|h_1(v) \neq c(v)) \tag{84}$$

Now, using Bayes' theorem :

$$P(v|h_1(v) \neq c(v)) = \frac{P(v \text{ and } h_1(v) \neq c(v))}{P(h_1(v) \neq c(v))} = \frac{D(v)p_1(v)}{a_1} \tag{85}$$

$$P(v|h_1(v) = c(v)) = \frac{D(v)(1 - p_1(v))}{1 - a_1} \tag{86}$$

with $1 - a_1 = w + x = P_D(h_1(v) = c(v))$.
Substituting these into (23), we obtain :

$$D_2(v) = \frac{D(v)}{2}\left(\frac{p_1(v)}{a_1} + \frac{1 - p_1(v)}{1 - a_1}\right) \tag{87}$$

Distribution D_3 only considers instances where hypotheses h_1 and h_2 disagree. So :

$$D_3(v) = P(v|h_1(v) \neq h_2(v)) = \frac{P(v \text{ and } h_1(v) \neq h_2(v))}{P(h_1(v) \neq h_2(v))} = \frac{D(v)q(v)}{w + y} \tag{88}$$

Now we compute the error of the hypothesis h built from h_1, h_2, h_3, i.e.

$$P_D(h(v) \neq c(v)) \tag{89}$$

where $c(v)$ is the correct label of v.
By the definition of h, two cases may occur :
First case : $h_1(v) = h_2(v)$, so $h(v) = h_1(v) = h_2(v)$. The algorithm makes an error if this common prediction is wrong.
Second case : $h_1(v) \neq h_2(v)$ and in this case $h(v) = h_3(v)$. Then, the algorithm errs if $h_3(v) \neq c(v)$.
Hence :

$$P_D(h(v) \neq c(v)) = P_D(h_1(v) = h_2(v) = h(v) \neq c(v)) + P_D((h_1(v) \neq h_2(v)) \cap (h_3(v) \neq c(v))) \tag{90}$$

Using the result of the note, we get :

$$P_D(h(v) \neq c(v)) = z + \sum_{v \in X_n} D(v)q(v)p_3(v) = z + \sum_{v \in X_n} (w + y)D_3(v)q(v)p_3(v) \tag{91}$$

The following lemma provides the results needed to complete the proof :

Lemme 2. *We have the following three equations :*

$$1 - a_2 = \frac{y}{2a_1} + \frac{z}{2(1 - a_1)} \tag{92}$$

$$z = a_1 - y \tag{93}$$

$$w = (2a_2 - 1)(1 - a_1) + \frac{y(1 - a_1)}{a_1} \tag{94}$$

Démonstration. We begin by proving (32). By definition :

$$1 - a_2 = P_{D_2}(h_2(v) = c(v)) = \sum_{v \in X_n} D_2(v)(1 - p_2(v))$$

Substituting the expression for $D_2(v)$ into the sum, we get :

$$1 - a_2 = P_{D_2}(h_2(v) = c(v)) = \sum_{v \in X_n} \left(\frac{1}{2}\frac{D(v)p_1(v)}{a_1} + \frac{1}{2}\frac{D(v)(1 - p_1(v))}{1 - a_1}\right)(1 - p_2(v))$$

$$\iff 1 - a_2 = \sum_{v \in X_n} \frac{1}{2}\frac{D(v)p_1(v)}{a_1}(1 - p_2(v)) + \frac{1}{2}\frac{D(v)(1 - p_1(v))}{1 - a_1}(1 - p_2(v))$$

Finally, using the remark again :

$$1 - a_2 = \frac{y}{2a_1} + \frac{z}{2(1 - a_1)}$$

We now prove (33). It is clear that :

$$\{h_1(v) \neq h_2(v), h_2(v) = c(v)\} \cap \{h_1(v) = h_2(v), h_2(v) = c(v)\} = \emptyset$$

Therefore :

$$\begin{aligned} y+z &= P_D(\{h_1(v) \neq h_2(v),\ h_2(v) = c(v)\}) + P_D(\{h_1(v) = h_2(v),\ h_2(v) = c(v)\}) \\ &= P_D(\{h_1(v) \neq h_2(v),\ h_2(v) = c(v)\} \cup \{h_1(v) = h_2(v),\ h_2(v) = c(v)\}) \\ &= P_D(\{h_1(v) = c(v)\}) = a_1 \end{aligned}$$

To conclude the proof of this lemma, we prove (34). From the definition :

$$\begin{aligned} w &= \sum_{v \in X_n} D(v)(1-p_1(v))p_2(v) \\ &= \sum_{v \in X_n} D(v)(1-p_1(v))(1-(1-p_2(v))) \\ &= (1-a_1) - \sum_{v \in X_n} D(v)(1-p_1(v))(1-p_2(v)) \\ &= (1-a_1) - z \end{aligned}$$

On the other hand, from (32), we have :

$$\begin{aligned} & 1 - a_2 = \frac{y}{2a_1} + \frac{z}{2(1-a_1)} \\ \iff \quad & 2(1-a_2) = \frac{y}{a_1} + \frac{z}{1-a_1} \\ \iff \quad & z = (1-a_1)\left(2(1-a_2) - \frac{y}{a_1}\right) \end{aligned}$$

□

We thus obtain :

$$\begin{aligned} w &= (1-a_1) - (1-a_1)\left(2(1-a_2) - \frac{y}{a_1}\right) \\ &= (1-a_1)\left(1 - \left(2(1-a_2) - \frac{y}{a_1}\right)\right) \\ &= (1-a_1)\left(2a_1 - 1 + \frac{y}{a_1}\right) \\ &= (2a_2-1)(1-a_1) + \frac{y(1-a_1)}{a_1} \end{aligned}$$

□

We can now conclude the proof of this theorem by returning to equation (31). We obtain :

$$\begin{aligned} P_D(h(v) \neq c(v)) &= z + a_3(w+y) \\ &\leq z + \alpha(w+y) \quad \text{by recursion} \\ &= a_1 - y + \alpha(2a_2-1)(1-a_1) + \alpha\frac{y(1-a_1)}{a_1} + \alpha y \quad \text{by substituting (33) and (34)} \\ &= \alpha(2a_2-1)(1-a_1) + a_1 + y\left(-1 + \alpha\left(1 + \frac{1-a_1}{a_1}\right)\right) \\ &= \alpha(2a_2-1)(1-a_1) + a_1 + \frac{y(\alpha - a_1)}{a_1} \\ &= \alpha(2a_2-1)(1-a_1) + a_1 + y\left(-1 + \alpha\left(1 + \frac{1-a_1}{a_1}\right)\right) \\ &\leq \alpha(2a_2-1)(1-a_1) + a_1 + \alpha - a_1 \quad \text{since} \quad y \leq a_1 \\ &= \alpha(2\alpha-1)(1-\alpha) \\ &= 3\alpha^2 - 2\alpha^3 = g(\alpha) = \epsilon \end{aligned}$$

Which concludes the proof of Theorem 2. □

According to Theorem 2, we know that the Learn(ϵ, δ,EX) algorithm returns, with probability at least $1 - \delta$, a hypothesis h such that $\text{err}_D(h) \leq \epsilon$. This allows us to conclude that any weak learning algorithm can be converted into a strong one, meaning it can achieve arbitrarily small error.

11.3 General Boosting Framework

Now that we have explained what weak classifiers are and established **Schapire's theorem**, we present in this section a more intuitive and algorithmic approach to boosting.
Boosting is based on the idea of constructing a weak classifier, for example using a CART model, and combining several such classifiers to obtain a more powerful global model. At each iteration, a new model is trained to correct the errors of the previous one, so that each step refines the predictions of the preceding model. We can describe boosting as an additive construction of the form :

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m) \tag{95}$$

where $b(x, \gamma_m)$ are the weak learners and β_m are the coefficients that determine their importance in the final model. This construction is called **forward stagewise additive modeling** and can be viewed as a series of successive corrections applied to a trivially initialized model.
The central intuition of boosting is that the model's attention is focused on the poorly predicted observations. In other words, at each step, a classifier is trained on a weighted version of the data in which well-predicted observations receive reduced weight, while misclassified ones receive increased weight.
Boosting thus implicitly modifies the data distribution at each iteration, emphasizing hard-to-classify cases. This idea, already present in Schapire's 1990 theoretical construction, is also at the heart of modern algorithms such as **AdaBoost** and **XGBoost**.
In the following, we present in more detail the two most important boosting algorithms : AdaBoost and XGBoost.

12 AdaBoost

12.1 General Overview of AdaBoost

AdaBoost is a boosting algorithm introduced in 1997 by Yoav Freund and Robert Schapire. It was one of the first algorithms to translate the theoretical foundations of boosting into a practical method.
In AdaBoost, the most commonly used weak models are decision trees of depth 1, called stumps. These trees make a single split on one explanatory variable, making them very simple but also individually limited. We construct them using the classification tree method described in the **CART** model section, i.e., the stump seeks the variable and threshold that minimize an impurity measure such as Gini index or cross-entropy. However, since it can use only one variable, a stump captures only partial information about the problem.

Example : If we want to predict a borrower's default risk, we could use several features such as income, employment status, marital status, etc. A stump can rely on only one of these variables, which significantly limits its predictive power.
Such trees are considered weak classifiers. AdaBoost sequentially trains these stumps by dynamically adjusting the observation weights at each iteration. Initially, each observation has the same weight, and a first stump, denoted G_1, is trained on this weighted data. Then, the weights are updated so that misclassified observations receive increased weight, while correctly classified ones receive reduced weight. The next stump will therefore focus more on the higher-weighted observations, and this process is repeated for a fixed number M of iterations. Thus, the order of the stumps matters since each model depends on the ones before it. Finally, the stumps are combined via a weighted vote, where each stump receives a weight α_m. In binary classification, the final prediction is given by :

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{96}$$

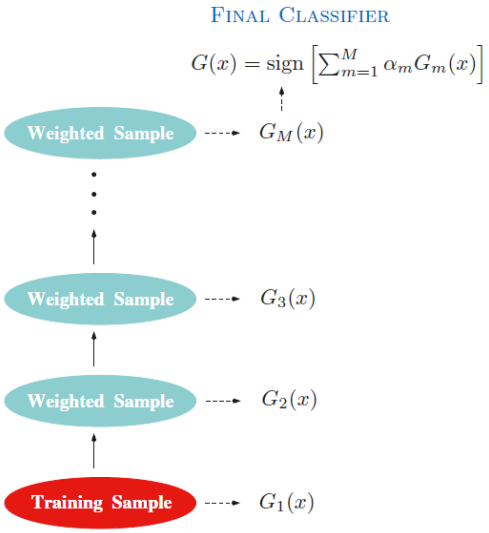

Figure 17 – Functioning of AdaBoost
Source : Hastie, Tibshirani, Friedman – *The Elements of Statistical Learning*, 2nd edition, Springer (2009), Figure 10.1.

12.2 Detailed Description of the Algorithm

In this section, we present the **AdaBoost** algorithm and provide an illustrative example to clarify its functioning. We consider a binary classification problem. The target variable Y thus takes values

in $\{1, -1\}$. For a feature X, the classifier outputs $G(X)$ in $\{-1, 1\}$. The AdaBoost algorithm is as follows :

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

FIGURE 18 – Pseudo-code of the AdaBoost.M1 algorithm
Source : Hastie, Tibshirani, Friedman – *The Elements of Statistical Learning*, 2nd edition, Springer (2009), Algorithm 10.1.

We consider the training set $\{(x_1, y_1), ..., (x_N, y_N)\}$. Initially, we assign equal weights to all observations :

$$w_i^{(1)} = \frac{1}{N} \text{ for } i = 1, ..., N$$

Then, for each iteration $m \in \{1, ..., M\}$, we train the weak classifier $G_m(x)$ on the training data weighted by $w_i^{(m)}$. Errors on higher-weighted observations are penalized more heavily. We compute the weighted error for this iteration as :

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i} \tag{97}$$

Normalizing the prediction error allows us to interpret it probabilistically it only takes values in $[0, 1]$. This ensures that model performance is measured on a consistent scale and maintains coherence across iterations.
Once we compute the classifier's weighted error ϵ_m, we determine its importance in the final model via the weight α_m :

$$\alpha_m = \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right) \tag{98}$$

This weight measures the classifier's reliability : the smaller err_m, the larger α_m, increasing its influence in the final prediction. Conversely, when err_m approaches 0.5 (random guessing), then $\alpha_m \to 0$. The form of α_m is not arbitrary and arises from minimizing an exponential loss function in an additive model this is justified in detail in section 4.3.
Next, we update the observation weights w_i. The idea is to increase the weights of misclassified examples and reduce the weights of correctly classified ones. For correctly classified i, $I(y_i \neq G_m(x)) = 0$ and :

$$w_i \exp(0) = w_i$$

so its weight remains unchanged. For misclassified ones, $I(y_i \neq G_m(x)) = 1$, and the weight is multiplied by a factor ≥ 1. Intuitively, this means the algorithm focuses more on misclassified

observations. We explain the form of the weight update in section 4.3. Finally, after M iterations, the final classifier is :

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{99}$$

Each G_m votes for class 1 or -1 for input x, and its vote is weighted by α_m. The most accurate classifiers contribute more to the final vote.

Note : In the algorithm above, we use two notions of weights which should not be confused. w_i is the weight of observation i used for training the next classifier. It is adjusted so that misclassified observations receive higher weights. On the other hand, α_m is the weight of the classifier m and determines its influence in the final prediction.

Example : Consider the following dataset with three observations :

x_i	y_i
A	1
B	-1
C	1

FIGURE 19 – Toy dataset with three observations to illustrate AdaBoost

We apply AdaBoost for two iterations. At the first iteration, all observations have weight $w_i^{(1)} = \frac{1}{3}$ for $i = 1, 2, 3$. Suppose that the first classifier G_1 correctly predicts points 1 and 3 but misclassifies point 2. The weighted error is then $\text{err}_m = \frac{1}{3}$ and the classifier's weight is $\alpha_1 = \log(2)$. After updating weights, we get $w_1 = \frac{1}{3}, w_2 = \frac{2}{3}, w_3 = \frac{1}{3}$ confirming that the misclassified observation is now more important. At iteration two, suppose classifier G_2 correctly classifies all observations. Then $\text{err}_2 = 0$ and $\alpha_2 \to +\infty$. In practice, we set $\epsilon = 10^{-10}$ so $\alpha_2 \approx 10\log(10) \approx 23$. Finally, the algorithm outputs $G(x) = \text{sign}(\alpha_1 G_1(x) + \alpha_2 G_2(x))$, and since $\alpha_2 \gg \alpha_1$, the final decision is mostly determined by the more accurate classifier, as expected.

12.3 Theoretical Analysis

12.3.1 AdaBoost as an Additive Model

An additive model is a model that can be written as :

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m) \tag{100}$$

where β_m are the expansion coefficients and $b(x, \gamma_m)$ is a function defined over x and parameterized by γ_m.
In this model, the unknown parameters are thus β_m and γ_m. To estimate them, we aim to determine the function f that minimizes a loss function over the training set, i.e., that yields the highest overall prediction accuracy. The optimization problem is therefore :

$$\min_{\{(\beta_m, \gamma_m)\}_{m=1}^{M}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m\, b(x_i; \gamma_m)\right) \tag{101}$$

Solving this optimization problem would allow us to adjust all parameters simultaneously in order to minimize the global error. However, this problem is often computationally infeasible. An alternative approach consists of determining the parameters (β_m, γ_m) by minimizing the loss function in a stepwise manner. This procedure is known as **forward stagewise additive modeling**, described in the algorithm below, which provides a general optimization method for additive models.

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M:
   (a) Compute
   $$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$
   (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.

FIGURE 20 – Pseudo-code of forward stagewise additive modeling
Source : Hastie, Tibshirani, Friedman – *The Elements of Statistical Learning*, 2nd edition, Springer (2009), Algorithm 10.2.

We start by initializing the function f to 0, meaning we make no prediction at the beginning. Then, for each step $m \in \{1, ..., M\}$, we search among all possible base functions $b(x, \gamma)$ and all coefficients β to reduce the overall loss if added to the model. In other words, we seek the base function that most improves the overall prediction. This is translated into the algorithm as :

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, f_{m-1}(x_i) + \beta\, b(x_i; \gamma)\right) \tag{102}$$

Once this pair of parameters is found, we update the model :

$$f_m(x) = f_{m-1}(x) + \beta_m b(x, \gamma_m) \tag{103}$$

After M iterations, we obtain the final form of f.
From equation (39), we can see that **AdaBoost** can be interpreted as an additive model built incrementally. In our case, the functions $b(x, \gamma_m)$ correspond to the weak classifiers $G_m(x) \in \{-1, 1\}$ and the coefficients β_m are the weights α_m assigned to each classifier. We will show in the next section that AdaBoost corresponds to minimizing the exponential loss function in this additive framework.

12.3.2 Exponential Loss Function

We will show in this section that AdaBoost is equivalent to the forward stagewise additive modeling algorithm when using the following loss function :

$$L(x, f(x)) = \exp(-yf(x)) \tag{104}$$

known as the **exponential loss function**.

Démonstration. We revisit the algorithm presented in Figure 6 and assume that at iteration $m-1$, we have constructed the partial model $f_{m-1}(x)$. We then seek the term $\beta G(x)$ such that $f_m(x) = f_{m-1}(x) + \beta G(x)$ in order to minimize the total loss :

$$(\beta_m, G_m) = \arg\min_{\beta,G} \sum_{i=1}^{N} \exp\left(-y_i \left(f_{m-1}(x_i) + \beta G(x_i)\right)\right) \tag{105}$$

We define the weights measuring how poorly each observation i is classified as :

$$w_i^{(m)} = \exp(-y_i f_{m-1}(x_i)) \tag{106}$$

When the prediction is incorrect, $y_i f_{m-1}(x_i) < 0$, and due to the exponential function's growth, the weight $w_i^{(m)}$ is higher than when the prediction is correct. This is consistent with what we explained when discussing the **AdaBoost** algorithm. Equation (45) can thus be rewritten as :

$$(\beta_m, G_m) = \arg\min_{\beta,G} \sum_{i=1}^{N} w_i^{(m)} \exp\left(-y_i \beta G(x_i)\right) \tag{107}$$

Moreover, since we assume $y_i, G(x_i) \in \{-1, 1\}$, we have that $\exp(-\beta y_i G(x_i)) = \exp(-\beta)$ if $y_i = G(x_i)$, and $\exp(\beta)$ if $y_i \neq G(x_i)$. We define :

$$C = \{i : y_i = G(x_i)\}$$
$$M = \{i : y_i \neq G(x_i)\}$$

Equation (47) then becomes :

$$(\beta_m, G_m) = \arg\min_{\beta, G} \exp(-\beta) \sum_{i \in C} w_i^{(m)} + \exp(\beta) \sum_{i \in M} w_i^{(m)} \tag{108}$$

Additionally, we also know that :

$$\sum_{i=1}^{N} w_i^{(m)} = \sum_{i \in C} w_i^{(m)} + \sum_{i \in M} w_i^{(m)} \tag{109}$$

Plugging the expression for $\sum_{i \in C} w_i^{(m)}$ into (48), we get :

$$\begin{aligned}
\exp(-\beta) \sum_{i \in C} w_i^{(m)} + \exp(\beta) \sum_{i \in M} w_i^{(m)} &= \exp(-\beta) \left(\sum_{i=1}^{N} w_i^{(m)} - \sum_{i \in M} w_i^{(m)} \right) + \exp(\beta) \sum_{i \in M} w_i^{(m)} \\
&= \exp(-\beta) \sum_{i=1}^{N} w_i^{(m)} + (\exp(\beta) - \exp(-\beta)) \sum_{i \in M} w_i^{(m)}
\end{aligned}$$

For a fixed β, minimizing (48) is equivalent to minimizing the weighted sum of the errors :

$$G_m = \arg\min_{G} \sum_{i \in M} w_i^{(m)} = \arg\min_{G} \sum_{i=1}^{N} w_i^{(m)} \mathbb{I}(y_i \neq G(x_i)) \tag{110}$$

which is exactly what AdaBoost does, i.e., at each iteration G_m is fitted so as to minimize the weighted sum of errors. We now fix G_m and define :

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i^{(m)} \mathbb{I}(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i^{(m)}} \tag{111}$$

Revisiting again the expression to minimize in (48), we obtain :

$$L(\beta) = \exp(-\beta)(1 - \text{err}_m) \sum_{i=1}^{N} w_i^{(m)} + \exp(\beta)\, \text{err}_m \sum_{i=1}^{N} w_i^{(m)} \tag{112}$$

Finally, to find the value of β that minimizes L, we solve :

$$\begin{aligned}
\frac{dL}{d\beta}(\beta) = 0 &\iff -(1 - \text{err}_m) \exp(-\beta) + \text{err}_m \exp(\beta) = 0 \\
&\iff \exp(-2\beta) = \frac{\text{err}_m}{1 - \text{err}_m} \\
&\iff \beta_m = \frac{1}{2} \log \left(\frac{1 - \text{err}_m}{\text{err}_m} \right)
\end{aligned}$$

This is exactly the AdaBoost formula for the weight of classifier m, up to a factor.
Finally, for a given step, the algorithm ends by updating the model :

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x) \tag{113}$$

The weight at step $m+1$ then becomes :

$$w_i^{(m+1)} = \exp(-y_i f_m(x_i)) = w_i^{(m)} \exp(-\beta_m y_i G_m(x_i)) \tag{114}$$

Using the fact that $-y_i G_m(x_i) = 2\mathbb{I}(y_i \neq G_m(x_i)) - 1$, we obtain :

$$w_i^{(m+1)} = w_i^{(m)} \exp\left(\alpha_m \mathbb{I}(y_i \neq G_m(x_i))\right) \exp(-\beta_m) \tag{115}$$

with $\alpha_m = 2\beta_m$. Since the factor $\exp(-\beta_m)$ multiplies all weights equally, it has no effect. □

We have therefore shown that forward stagewise additive modeling using the exponential loss is equivalent to applying the AdaBoost algorithm. We now explore why the exponential loss leads to AdaBoost and derive a probabilistic interpretation.

12.3.3 Probabilistic Interpretation

In the previous section, we introduced the empirical exponential loss, i.e., minimization is performed on a dataset $\{(x_i, y_i)\}_{i=1}^N$. We now consider the theoretical form of the exponential loss to determine the ideal function that **AdaBoost** attempts to estimate. We define the theoretical exponential loss as :

$$L(f) = \mathbb{E}_{Y|x}[\exp(-Yf(x))] \tag{116}$$

where the expectation is taken over the distribution of Y given the input x. This is justified since we know the input x and want to find the optimal value of $f^*(x)$ that minimizes the expected loss over the distribution of Y given x. We seek a function f^* such that :

$$f^*(x) = \arg\min_f \mathbb{E}_{Y|x}\left[\exp\left(-Yf(x)\right)\right] \tag{117}$$

Since $Y \in \{-1, 1\}$, let $p = P(Y = 1|x)$ and thus $1 - p = P(Y = -1|x)$. We obtain :

$$L(f) = p \cdot \exp(-f) + (1-p) \cdot \exp(f)$$

To minimize this expression :

$$\begin{aligned}
& \frac{dL}{df} = -p \cdot \exp(-f) + (1-p) \cdot \exp(f) = 0 \\
\iff \quad & (1-p) \cdot \exp(f) = p \cdot \exp(-f) \\
\iff \quad & \exp(2f) = \frac{p}{1-p} \\
\iff \quad & f^*(x) = \frac{1}{2} \log\left(\frac{p}{1-p}\right)
\end{aligned}$$

This result can be rewritten as :

$$f^*(x) = \frac{1}{2} \log\left(\frac{P(Y=1|x)}{P(Y=-1|x)}\right) \tag{118}$$

or equivalently :

$$\Pr(Y = 1 \mid x) = \frac{1}{1 + e^{-2f^*(x)}} \tag{119}$$

We observe that the score f learned by AdaBoost is half of the logit of the conditional probability $P(Y = 1|x)$. Therefore, AdaBoost can also be interpreted as a probabilistic estimator approximating the logit function.
We also introduce another criterion for modeling the conditional probability $P(Y = 1|x)$, namely the negative binomial log-likelihood, also called deviance, defined by :

$$l(Y, f(x)) = \log(1 + \exp(-2Yf(x))) \tag{120}$$

This function allows us to compare robustness with the exponential loss, and we can show that this function is minimized — like the exponential loss — by :

$$f^*(x) = \frac{1}{2} \log\left(\frac{P(Y=1|x)}{P(Y=-1|x)}\right)$$

and thus targets the same theoretical optimum.

12.3.4 Limitations of the Exponential Loss and Robustness Issues

In this section, we focus on the robustness of **AdaBoost** and compare it with the deviance. Recall that the score function $f(x)$ associated with an observation measures the model's confidence in its prediction. The larger this score, the more confident the model is. When $yf(x) > 0$, the observation is correctly classified ; when negative, it is misclassified. When $yf(x) = 0$, the model lies exactly on the decision boundary. We also know that the goal of any classification algorithm is to produce as many positive margins as possible.

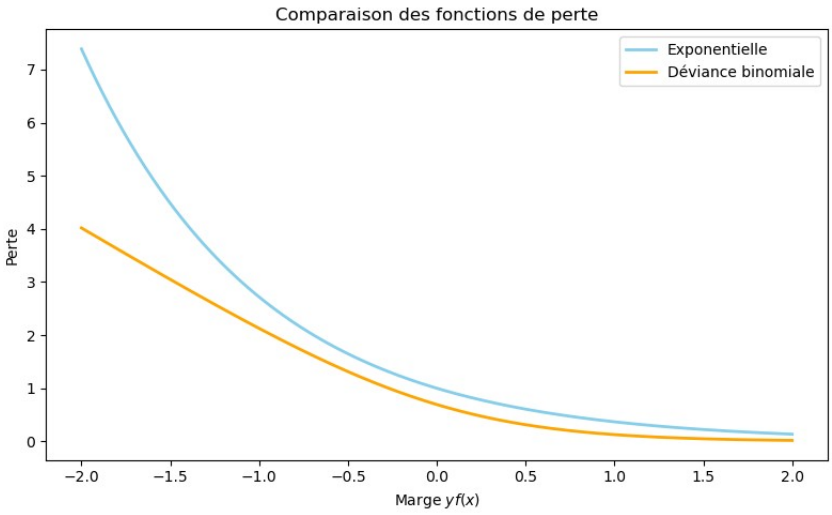

FIGURE 21 – Comparison between exponential loss and binomial deviance as a function of the margin $yf(x)$.

The figure above compares the loss as a function of the margin $yf(x)$ for both exponential loss and deviance. We observe that the deviance increases fairly linearly for negative margins, whereas the exponential loss grows exponentially. This implies that the exponential loss places greater emphasis on misclassified observations than the deviance, which distributes influence more evenly across all observations.
Thus, deviance is considerably more robust in noisy settings. If an observation is mislabeled or an outlier, the exponential loss will assign it a significant weight, potentially distorting the model heavily and degrading predictions on the rest of the dataset. The exponential loss is therefore subject to a risk of *localized overfitting*. This is less the case with deviance, which accounts for such points without assigning them excessive importance.
We have highlighted some of AdaBoost's limitations, notably its high sensitivity to misclassified observations. We will now aim to develop a more robust algorithm and turn our attention to XGBoost.

13 XGBoost

13.1 Motivation and Introduction

After presenting **AdaBoost**, which relies on adaptive reweighting of misclassified observations to combine several weak classifiers, we now introduce a more general and powerful approach : **Gradient Boosting**.

The fundamental idea behind Gradient Boosting is to reformulate the boosting problem as an optimization problem, where each new weak learner is constructed to reduce the residual error of the previous model using gradient descent.

Rather than focusing solely on classification errors as in AdaBoost, Gradient Boosting adopts a more flexible approach : it minimizes a general differentiable loss function (e.g., squared error in regression) by iteratively adding predictors that point in the direction of the negative gradient of the loss function, computed with respect to the current model predictions.

Within this framework, **XGBoost** (eXtreme Gradient Boosting) was introduced by Tianqi Chen and Carlos Guestrin in 2016 in *XGBoost : A Scalable Tree Boosting System*. It is an optimized and highly efficient version of Gradient Boosting, widely used in practice and a frequent winner of data science competitions.

Compared to classic Gradient Boosting and other tree-based methods, XGBoost offers two major advantages.

First, it is computationally efficient because it processes data in compressed blocks, allowing fast sorting and parallel processing. This results in a significantly shorter training time compared to traditional Gradient Boosting (see figure below).

Second, it uses a second-order Taylor expansion to minimize the objective function. This means it considers the curvature of the objective function, enabling it to converge to a higher-quality minimum than other methods.

It is also important to note that XGBoost does not rely on the traditional decision tree construction criteria, such as cross-entropy or Gini index. Instead, it uses specific criteria, which are presented and explained in detail in this document.

Third, XGBoost integrates built-in regularization mechanisms to limit overfitting.

Finally, it can also intelligently handle missing data. During training, the model learns which direction (left or right in the tree) is optimal for a missing value, eliminating the need for artificial imputation.

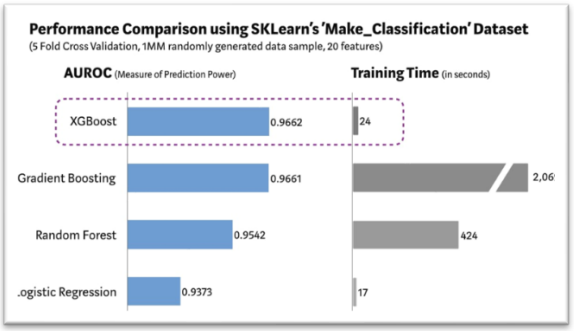

FIGURE 22 – Enter Caption

13.2 General Principle of Gradient Boosting

Gradient Boosting relies on the sequential use of multiple decision trees. Unlike random forests, the decision trees are not built independently. The new tree is trained on the errors made by the ensemble of previous trees. For each observation, each tree provides a prediction, and the final prediction is obtained by summing the contributions of all trees.

Before discussing Gradient Boosting, it is important to revisit the idea of greedy boosting. Traditional boosting consists of sequentially adding trees with the goal of correcting the errors made by previous ones. At each iteration, a new tree is added to improve the overall prediction. This method can be considered greedy in the sense that at each step, the algorithm focuses only on the local error (the residual errors of the previous trees) without globally considering the loss function minimization.

Gradient Boosting improves this approach by introducing a more global optimization that considers the entire loss function by following its gradient. The idea is to guide the algorithm more precisely at each iteration to optimally minimize the loss function, rather than just reduce local residual errors.

Gradient Boosting is based on a fundamental concept : following the gradient of the loss function. This means that each new tree built at iteration m is trained not only to correct residual errors from the previous model but to follow the direction of the loss function gradient. The gradient represents the direction in which the loss decreases most rapidly, and it is in this direction that the tree should be adjusted.

To compute this gradient, we take the derivative of the loss function with respect to the model prediction, which provides a clear idea of the adjustment needed to reduce the error.

At the beginning of the boosting process, each new tree is added to correct the prediction errors. At each iteration m, the gradient of the loss function $L(y_i, f(x_i))$ with respect to the current predictions $f_{m-1}(x_i)$ (after $m-1$ iterations) is computed :

$$g_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\bigg|_{f(x_i)=f_{m-1}(x_i)}$$

This is the derivative of the loss function with respect to the current model prediction. The gradient shows the direction in which predictions should be adjusted to minimize the error. More specifically, it indicates how much predictions should be adjusted to reduce the loss. This is the direction in which the new tree will be built.

Once the gradient is calculated, a new decision tree $T_m(x_i)$ is constructed. This tree aims to minimize the error on the residuals by adjusting the predictions in the direction of the gradient. The tree is trained to reduce the model errors by following the negative slope of the loss function gradient.
The prediction is then updated as follows :

$$f_m(x) = f_{m-1}(x) + T_m(x)$$

The model is updated after each tree is added. The final prediction $f_M(x)$ after M iterations is given by the sum of successive predictions :

$$f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m)$$

At each iteration, each tree $T(x; \Theta_m)$ is trained to minimize the errors of the previous trees by following the direction of the loss function gradient. This allows for a more effective reduction of the global error compared to traditional boosting.

The calculation of regions R_j and predictions γ_j is central to optimizing Gradient Boosting. Each region R_j is determined to minimize the loss function $L(y_i, \gamma_j)$, adjusting the partitions of the space to reduce the residuals.

The regions R_j can be computed iteratively, and the optimal solution for each γ_j in region R_j is found using a regression-style partitioning algorithm that minimizes the squared error in each region.

In Gradient Boosting, a **gradient descent** approach is used to solve the optimization problem. Gradient descent is an iterative method that finds the parameters Θ_m minimizing the loss function. At each iteration, the model is adjusted by following the direction of the loss function gradient.
The model $f_m(x)$ is updated by adding a tree $T_m(x)$, using the following optimization :

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} L\left(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)\right)$$

This progressively adjusts the model to reduce the error over the entire dataset by adding successive trees that follow the gradient direction.

The algorithm for Gradient Tree Boosting is as follows :

1. **Initialization** : Initialize the model $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$, with a simple prediction (e.g., the mean of y_i).
2. **For** $m = 1$ **to** M :
   (a) For $i = 1, 2, \ldots, N$, compute residuals r_{im} as :

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}.$$

   (b) Fit a regression tree $T(x; \Theta_m)$ to the targets r_{im}, yielding terminal regions $R_{jm}, j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$, compute predictions γ_{jm} by solving :

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$

   (d) Update the model prediction :

$$f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm}).$$

3. **Output** : The final model prediction is $\hat{f}(x) = f_M(x)$.

13.3 From Gradient Boosting to XGBoost

We have seen that Gradient Boosting relies on the idea of building successive models to reduce residual errors of previous trees by adjusting predictions in the direction of the loss function gradient. This is done iteratively by adding a new tree at each step to improve predictions. While powerful, this process has limitations. It can be slow on large datasets due to its sequential nature. It is also prone to overfitting if trees are too complex. Moreover, it handles missing values poorly, often requiring prior imputation. Lastly, its optimization is based solely on first-order gradients, without accounting for the curvature of the loss function, which can limit its precision.

XGBoost (eXtreme Gradient Boosting) overcomes these limitations by introducing several notable improvements in both optimization and regularization.

13.4 XGBoost : Optimization and Regularization

XGBoost builds on the principles of Gradient Boosting and enhances them with several key optimizations, which are detailed below.

13.4.1 Regularization

One of the main innovations of XGBoost compared to traditional Gradient Boosting is the addition of **regularization** in the objective function. Regularization helps control tree complexity and prevent overfitting by limiting tree depth and weight magnitude.

The objective function in XGBoost thus includes a regularization term, $\Omega(T)$, added to the loss function. The complete objective function becomes :

$$\mathcal{L}(\Theta) = \sum_{i=1}^{N} L(y_i, \hat{y}_i) + \lambda \sum_{j=1}^{J} \|\gamma_j\|^2 + \gamma \sum_{k=1}^{K} d_k$$

— $\sum_{i=1}^{N} L(y_i, \hat{y}_i)$: Loss function measuring the discrepancy between prediction $\hat{y}_i$ and ground truth y_i. It must be differentiable and convex (to allow easy optimization to its minimum).

— **Weight penalty :**

$$\lambda \sum_{j=1}^{J} \|\gamma_j\|^2$$

This sum **penalizes the norm of the weights** (or scores) associated with the tree leaves. It prevents extreme values of the weights.

— **Meaning of symbols :**
  — γ_j : weight assigned to leaf j in a tree.
  — $\|\gamma_j\|^2$: squared norm (often γ_j^2 if scalar), implying a **quadratic penalty**.
  — $\sum_{j=1}^{J}$: this regularization is applied **to all leaves** $j = 1, \ldots, J$.
  — λ : hyperparameter that reduces model complexity and prevents overfitting. The higher λ, the more extreme weights γ_j are suppressed.

— **Structural penalty :**

$$\gamma \sum_{k=1}^{K} d_k$$

This sum **penalizes the structural complexity** of trees in the model by limiting their **depth**.

— **Meaning of symbols :**
  — d_k : depth or complexity of tree k, e.g., number of leaves.
  — $\sum_{k=1}^{K} d_k$: sum of all tree depths for $k = 1, \ldots, K$.
  — γ : hyperparameter controlling the penalty on tree size (larger γ favors simpler, shallower trees).

13.4.2 Objective Function Optimization via Taylor Approximation

A key contribution of XGBoost is expressing the objective function in a form that allows efficient optimization at each iteration m. This relies on a second-order Taylor expansion of the loss function, enabling the use of both the gradient and Hessian to accelerate convergence.

Objective Function Approximation

We aim to minimize the objective function at iteration m :

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

Expanding $L(y_i, \hat{y}_i^{(m-1)} + f_m(x_i))$ around $\hat{y}_i^{(m-1)}$ using a second-order Taylor expansion :

$$\mathcal{L}^{(m)}(f_m) \approx \sum_{i=1}^{N} \left[g_i \cdot f_m(x_i) + \frac{1}{2} h_i \cdot f_m^2(x_i) \right] + \Omega(f_m)$$

where :
— $g_i = \frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i}\Big|_{\hat{y}_i = \hat{y}_i^{(m-1)}}$: the gradient (first derivative)
— $h_i = \frac{\partial^2 L(y_i, \hat{y}_i)}{\partial \hat{y}_i^2}\Big|_{\hat{y}_i = \hat{y}_i^{(m-1)}}$: the Hessian (second derivative)

Structure of f_m and Simplified Expression

Assume $f_m(x_i)$ equals the weight w_j of the leaf j that observation x_i falls into. Let I_j be the set of indices i assigned to leaf j, and J the total number of leaves.

Replacing $f_m(x_i) = w_j$ for $i \in I_j$:

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

Interpretation of Regularization Terms

— λ penalizes large weights w_j, avoiding extreme scores and overfitting.
— γ penalizes the total number of leaves J, encouraging simpler trees.

Optimal Weight w_j of Each Leaf

We minimize the objective with respect to each w_j :

$$\frac{\partial \mathcal{L}^{(m)}}{\partial w_j} = \left(\sum_{i \in I_j} g_i \right) + \left(\sum_{i \in I_j} h_i + \lambda \right) w_j = 0$$

Thus, the optimal solution is :

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

Optimized Loss Associated with a Tree

Substituting w_j^* into $\mathcal{L}^{(m)}$, we get the final loss of the tree at iteration m :

$$\mathcal{L}^{(m)} = -\frac{1}{2} \sum_{j=1}^{J} \frac{\left(\sum_{i \in I_j} g_i \right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma J$$

This expression is used to compute the information gain G for selecting the best split.

13.4.3 Optimal Split Criterion in XGBoost

A split is considered beneficial if the penalized loss sum of the resulting leaves is lower than that of the original leaf :

$$\text{Loss}_{\text{Left}} + \text{Loss}_{\text{Right}} < \text{Loss}_{\text{Parent}}$$

The full tree loss at iteration m is :

$$\mathcal{L}^{(m)} = -\frac{1}{2} \sum_{j=1}^{J} \frac{\left(\sum_{i \in I_j} g_i \right)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma J$$

For a single leaf (before splitting), the loss is :

$$-\frac{1}{2}\frac{\left(\sum_{i\in I_{\mathrm{P}}} g_i\right)^2}{\sum_{i\in I_{\mathrm{P}}} h_i+\lambda}+\gamma$$

with :
— I_{P} : indices of observations in the parent leaf,
— I_{L}, I_{R} : indices in left and right child leaves.
The split is beneficial if :

$$-\frac{1}{2}\frac{\left(\sum_{i\in I_{\mathrm{L}}} g_i\right)^2}{\sum_{i\in I_{\mathrm{L}}} h_i+\lambda}-\frac{1}{2}\frac{\left(\sum_{i\in I_{\mathrm{R}}} g_i\right)^2}{\sum_{i\in I_{\mathrm{R}}} h_i+\lambda}+2\gamma<-\frac{1}{2}\frac{\left(\sum_{i\in I_{\mathrm{P}}} g_i\right)^2}{\sum_{i\in I_{\mathrm{P}}} h_i+\lambda}+\gamma$$

Which simplifies to :

$$\frac{\left(\sum_{i\in I_{\mathrm{L}}} g_i\right)^2}{\sum_{i\in I_{\mathrm{L}}} h_i+\lambda}+\frac{\left(\sum_{i\in I_{\mathrm{R}}} g_i\right)^2}{\sum_{i\in I_{\mathrm{R}}} h_i+\lambda}-\frac{\left(\sum_{i\in I_{\mathrm{P}}} g_i\right)^2}{\sum_{i\in I_{\mathrm{P}}} h_i+\lambda}-2\gamma>0$$

This criterion means a split is beneficial if it reduces overall loss, including complexity penalties. XGBoost evaluates all potential splits and selects the one maximizing this gain.

13.4.4 Parallelization and Missing Value Handling

XGBoost is designed for speed, even on large datasets, by leveraging parallel computation. Gradient and Hessian calculations are parallelized for faster execution. Additionally, XGBoost handles **missing values** efficiently by automatically determining the best way to handle them during tree construction.

13.5 Summary

In conclusion, XGBoost enhances Gradient Boosting through regularization, second-order gradient descent, parallelization, and advanced split search techniques. These improvements allow XGBoost to outperform traditional methods in terms of speed and accuracy, making it a top choice for complex regression and classification tasks.

13.6 Numerical Example : Regression

Let us consider the following dataset, which measures the effectiveness of a drug based on its dosage. Let x denote dosage and y the measured effectiveness. The goal is to build a sequence of trees to predict y based on x :

X (dosage)	10	20	25	35
Y (effectiveness)	-10	7	8	-7

This is a simple dataset with 4 observations and 1 feature, ideal for illustrating the XGBoost regression algorithm.

Algorithm Steps

— Step 0 : Initialize the first estimator $\hat{y}^{(0)}$
— Step 1 : Compute residuals $y_i - \hat{y}^{(0)}$
— Step 2 : Build the first tree
  — Find optimal split
  — Compute leaf predictions
  — Iterate until no more splits are possible
— Step 3 : Compute residuals for next tree

Step 0 : Initialization

— We take $\hat{y}^{(0)} = 0.5$

Step 1 : Compute Residuals

X (dosage)	10	20	25	35
Y (effectiveness)	-10	7	8	-7
First est. $\hat{y}^{(0)}$	0.5	0.5	0.5	0.5
Residuals	-10.5	6.5	7.5	-7.5

Step 2 : Tree Construction

For regression, we use the classic quadratic loss function :

$$L(y_i, \hat{y}_i) = \frac{1}{2} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

where $\hat{y}_i$ is the model prediction and y_i the true value.

Gradient Calculation

We differentiate the loss function with respect to $\hat{y}_i$:

$$g_i = \frac{\partial}{\partial \hat{y}_i} L(y_i, \hat{y}_i) = (\hat{y}_i - y_i)$$

Hessian Calculation

Differentiating again yields the second derivative :

$$h_i = \frac{\partial^2}{\partial \hat{y}_i^2} L(y_i, \hat{y}_i) = \frac{\partial}{\partial \hat{y}_i} (\hat{y}_i - y_i) = 1$$

So the Hessian is constant :

$$h_i = 1 \quad \text{(constant value in the quadratic case)}$$

Score formula :

$$\text{Score} = \frac{(\sum g_i)^2}{\sum h_i + \lambda} + \frac{(\sum g_i)^2}{\sum h_i + \lambda} - \frac{(\sum g_i)^2}{\sum h_i + \lambda} - 2\gamma$$

Case 1 : $\lambda = 0$, $\gamma = 0$

First-level tree split : comparing the score of each split. We have 4 values in this example, so 3 possible splits.

	Split 1.1 (x < 15)	**Split 1.2 (x < 22.5)**	**Split 1.3 (x < 30)**
Initial leaf – Residuals	[-10.5, 6.5, 7.5, -7.5]	[-10.5, 6.5, 7.5, -7.5]	[-10.5, 6.5, 7.5, -7.5]
Initial leaf – Score ($\lambda = 0$)	$\frac{(-10.5 + 6.5 + 7.5 - 7.5)^2}{4} = 4$		
Left leaf – Residuals	[-10.5]	[-10.5, 6.5]	[-10.5, 6.5, 7.5]
Left leaf – Score ($\lambda = 0$)	$\frac{(-10.5)^2}{1} = 110.25$	$\frac{(-10.5 + 6.5)^2}{2} = 8$	$\frac{(-10.5 + 6.5 + 7.5)^2}{3} = 4.08$
Right leaf – Residuals	[6.5, 7.5, -7.5]	[7.5, -7.5]	[-7.5]
Right leaf – Score ($\lambda = 0$)	$\frac{(6.5 + 7.5 - 7.5)^2}{3} = 14.08$	$\frac{(7.5 - 7.5)^2}{2} = 0$	$\frac{(-7.5)^2}{1} = 56.25$
Total Score ($\lambda = 0, \gamma = 0$)	$110.25 + 14.08 - 4 = 120.33$	$8 + 0 - 4 = 4$	$4.08 + 56.25 - 4 = 56.33$

TABLE 9 – Comparison of first-level tree splits ($\lambda = 0, \gamma = 0$)

Split 1.1 (x $<$ 15) has the highest score, so it is retained in the tree construction at the first level.

Now we split the right leaf.

	Split 2.1 (x $<$ 22.5)	**Split 2.2 (x $<$ 30)**
Initial leaf – Residuals	[6.5, 7.5, -7.5]	[6.5, 7.5, -7.5]
Initial leaf – Score ($\lambda = 0$)	$\frac{(6.5+7.5-7.5)^2}{3} = 14.08$	
Left leaf – Residuals	[6.5]	[6.5, 7.5]
Left leaf – Score ($\lambda = 0$)	$\frac{(6.5)^2}{1} = 42.25$	$\frac{(6.5+7.5)^2}{2} = 98$
Right leaf – Residuals	[7.5, -7.5]	[-7.5]
Right leaf – Score ($\lambda = 0$)	$\frac{(7.5-7.5)^2}{2} = 0$	$\frac{(-7.5)^2}{1} = 56.25$
Total Score ($\lambda = 0, \gamma = 0$)	$42.25 + 0 - 14.08 = 28.17$	$98 + 56.25 - 14.08 = 140.17$

TABLE 10 – Split scores at the second level of the tree (case $\lambda = 0, \gamma = 0$)

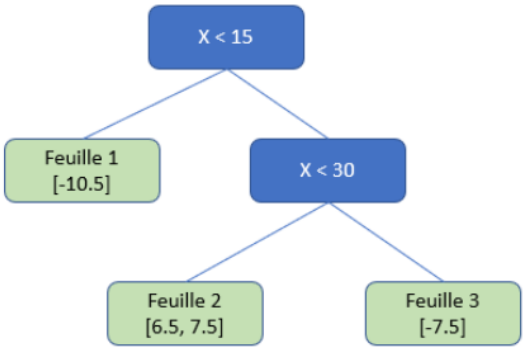

FIGURE 23 – Final tree

Split 2.2 (x $<$ 30) has the highest gain, so it is kept in the tree construction.
We now assume the tree is fully built and compute the optimal output values for each leaf using the remaining residuals.

Leaf output calculation

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

[H]

Leaf	Residuals	Output ($\lambda = 0$)
Leaf 1	[-10.5]	-10.5
Leaf 2	[6.5, 7.5]	7
Leaf 3	[-7.5]	-7.5

New prediction :

$$\hat{y}^{(1)} = \hat{y}^{(0)} + f_1(x_i)$$

X	10	20	25	35
Y	-10	7	8	-7
$\hat{y}^{(0)}$	0.5	0.5	0.5	0.5
Residual	-10.5	6.5	7.5	-7.5
$f_1(x_i)$	-10.5	7	7	-7.5
$\hat{y}^{(1)}$	-10	7.5	7.5	-7

Case 2 : $\lambda = 1$, $\gamma = 0$

First-level tree split :

First-level tree split	**Split 1.1** ($x < 15$)	**Split 1.2** ($x < 22.5$)	**Split 1.3** ($x < 30$)
Initial leaf – Residuals	$[-10.5, 6.5, 7.5, -7.5]$	$[-10.5, 6.5, 7.5, -7.5]$	$[-10.5, 6.5, 7.5, -7.5]$
Initial leaf – Score ($\lambda = 1$)	$\frac{(-10.5+6.5+7.5-7.5)^2}{4+1} = 3.2$		
Left leaf – Residuals	$[-10.5]$	$[-10.5, 6.5]$	$[-10.5, 6.5, 7.5]$
Left leaf – Score ($\lambda = 1$)	$\frac{(-10.5)^2}{2} = 55.12$	$\frac{(-10.5+6.5)^2}{3} = 5.33$	$\frac{(-10.5+6.5+7.5)^2}{4} = 3.06$
Right leaf – Residuals	$[6.5, 7.5, -7.5]$	$[7.5, -7.5]$	$[-7.5]$
Right leaf – Score ($\lambda = 1$)	$\frac{(6.5+7.5-7.5)^2}{4} = 10.56$	$\frac{(7.5-7.5)^2}{3} = 0$	$\frac{(-7.5)^2}{2} = 28.1$
Total Score ($\lambda = 1, \gamma = 0$)	$55.12 + 10.56 - 3.2 = 62.48$	$5.33 - 3.2 = 2.13$	$3.06 + 28.1 - 3.2 = 27.96$

TABLE 11 – Comparison of first-level tree splits with regularization $\lambda = 1$

Case 2 : $\lambda = 1$, $\gamma = 0$

First-level tree split :

First-level Tree Split	**Split 1.1** ($x < 15$)	**Split 1.2** ($x < 22.5$)	**Split 1.3** ($x < 30$)
Initial Leaf – Residuals	$[-10.5, 6.5, 7.5, -7.5]$	$[-10.5, 6.5, 7.5, -7.5]$	$[-10.5, 6.5, 7.5, -7.5]$
Initial Leaf – Score ($\lambda = 1$)	$\frac{(-10.5+6.5+7.5-7.5)^2}{4+1} = 3.2$		
Left Leaf – Residuals	$[-10.5]$	$[-10.5, 6.5]$	$[-10.5, 6.5, 7.5]$
Left Leaf – Score ($\lambda = 1$)	$\frac{(-10.5)^2}{2} = 55.12$	$\frac{(-10.5+6.5)^2}{3} = 5.33$	$\frac{(-10.5+6.5+7.5)^2}{4} = 3.06$
Right Leaf – Residuals	$[6.5, 7.5, -7.5]$	$[7.5, -7.5]$	$[-7.5]$
Right Leaf – Score ($\lambda = 1$)	$\frac{(6.5+7.5-7.5)^2}{4} = 10.56$	$\frac{(7.5-7.5)^2}{3} = 0$	$\frac{(-7.5)^2}{2} = 28.1$
Total Score ($\lambda = 1, \gamma = 0$)	$55.12 + 10.56 - 3.2 = 62.48$	$5.33 - 3.2 = 2.13$	$3.06 + 28.1 - 3.2 = 27.96$

TABLE 12 – Comparison of first-level splits with regularization $\lambda = 1$

We see that Split 1.1 ($x < 15$) still yields the highest score, so it is kept in the first level of the tree construction. However, adding a regularization term $\lambda = 1$ significantly reduces the scores, resulting in much shallower trees compared to the unregularized case. This is precisely the intended effect : to produce shallower trees in order to reduce the risk of overfitting.

We obtain the same splits as in Case 1. Now, calculating the leaf outputs gives :

Leaf	**Residuals**	**Output** ($\lambda = 0$)	**Output** ($\lambda = 1$)
Leaf 1	$[-10.5]$	$\frac{-10.5}{1} = -10.5$	$\frac{-10.5}{2} = -5.25$
Leaf 2	$[6.5,\ 7.5]$	$\frac{6.5+7.5}{2} = 7$	$\frac{6.5+7.5}{3} = 4.6$
Leaf 3	$[-7.5]$	$\frac{-7.5}{1} = -7.5$	$\frac{-7.5}{2} = -3.75$

TABLE 13 – Optimal output per leaf for $\lambda = 0$ and $\lambda = 1$

New prediction :

$$\hat{y}^{(1)} = \hat{y}^{(0)} + f_1(x_i)$$

X	10	20	25	35
Y	-10	7	8	-7
$\hat{y}^{(0)}$	0.5	0.5	0.5	0.5
Residual	-10.5	6.5	7.5	-7.5
$f_1(x_i)$	-5.25	4.6	4.6	-3.75
$\hat{y}^{(1)}$	-4.75	5.1	5.1	-3.25

Case 3 : $\gamma > 0$

$$\text{Score} = \text{Score}_{LeftLeaf} + \text{Score}_{RightLeaf} - \text{Score}_{InitialLeaf} - 2\gamma$$

When $\gamma > 0$, this reduces the number of valid splits (since the score must be positive). Consequently, it limits the tree depth, reducing the risk of overfitting.

14 Simulation

14.1 Simulation Framework

We now present the simulation we implemented to evaluate the performance of the CART model and the **AdaBoost** and **XGBoost** algorithms for detecting heart disease using various explanatory variables.

Our simulation is based on a dataset containing over 900 cases, each indicating the presence or absence of heart disease. We implemented a supervised classification to predict the presence of heart disease. Our explanatory variables include :

— **Biological variables :** the patient's age and sex (encoded : 1 = male, 0 = female)
— **Clinical variables :** resting blood pressure (in mm Hg), cholesterol level (in mg/dL), fasting blood sugar (encoded as 1 if > 120 mg/dL, else 0), maximum heart rate during exercise, and ST segment depression induced by exercise (The ST segment is part of the electrocardiogram that represents the period between ventricular depolarization and repolarization. A depression in this segment indicates insufficient oxygen supply to the heart muscle during this phase)
— **Exercise-induced angina :** presence or absence of chest pain triggered by physical effort
— **Chest pain variables :** distinguishing between anginal and non-anginal chest pain
— **Electrocardiogram analysis :** interpretation of the patient's ECG
— **Slope of the ST segment :** whether the ST segment is flat or upsloping

We then trained our three predictive models using these variables via Python code (see Appendix), and reported their performance metrics and confusion matrices.

Reminder : Evaluation Metrics

To evaluate the performance of our classification models, we used the following metrics :

— **Precision :** the proportion of true positive predictions among all positive predictions.

$$\text{Precision} = \frac{TP}{TP + FP}$$

where TP is true positives and FP is false positives.

— **Recall :** the proportion of true positive predictions among all actual positive instances.

$$\text{Recall} = \frac{TP}{TP + FN}$$

where FN is false negatives.

— **F1-score :** the harmonic mean of precision and recall, especially useful for imbalanced classes.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

14.2 Performance of the CART Model

The model was tested on 368 records (40% for testing and 60% for training). Here are its performance metrics by class :

Class	Precision	Recall	F1-score	Number of samples
0 (no disease)	0.76	0.75	0.76	173
1 (heart disease)	0.78	0.79	0.79	195

TABLE 14 – Performance metrics for the CART model by class

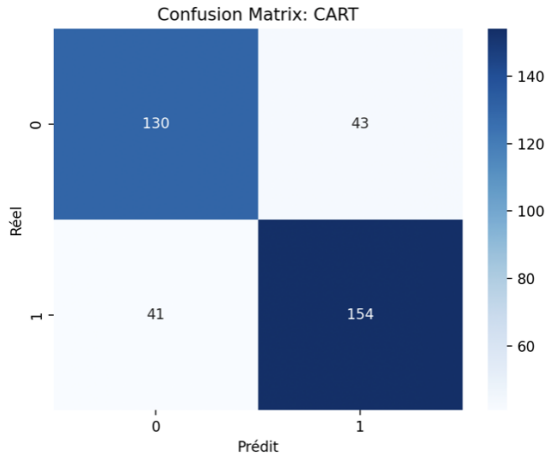

FIGURE 24 – Confusion matrix for the CART model

The model demonstrates balanced predictions between both classes. The number of prediction errors is relatively similar : 41 false negatives and 43 false positives. This indicates that the model does not favor one class over the other. The average F1-score of 0.77 reflects reasonably balanced overall performance.

14.3 Performance of AdaBoost

Like the **CART** model, **AdaBoost** was tested on 368 records (40% testing and 60% training). Here are its performance metrics by class :

Class (actual)	Precision	Recall	F1-score	Number of samples
0 (no disease)	0.82	0.83	0.82	173
1 (heart disease)	0.85	0.84	0.84	195

TABLE 15 – Performance metrics for the AdaBoost model by class

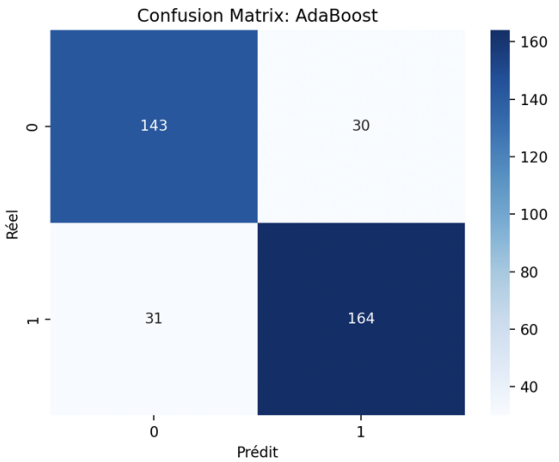

FIGURE 25 – Confusion matrix for AdaBoost

Although a slight imbalance is observed between class predictions, AdaBoost performs better than CART. It achieves a precision of 0.82 for class 0 and 0.85 for class 1. The confusion matrix shows fewer errors : 30 false positives and 31 false negatives. The average F1-score of 0.83 indicates a clear improvement over CART, especially in terms of precision and recall.

14.4 Performance of XGBoost

As with the previous models, **XGBoost** was tested on 368 records (40% testing and 60% training). Here are its performance metrics by class :

Class (actual)	Precision	Recall	F1-score	Number of samples
0 (no disease)	0.86	0.82	0.84	173
1 (heart disease)	0.85	0.88	0.86	195

TABLE 16 – Performance metrics for the XGBoost model by class

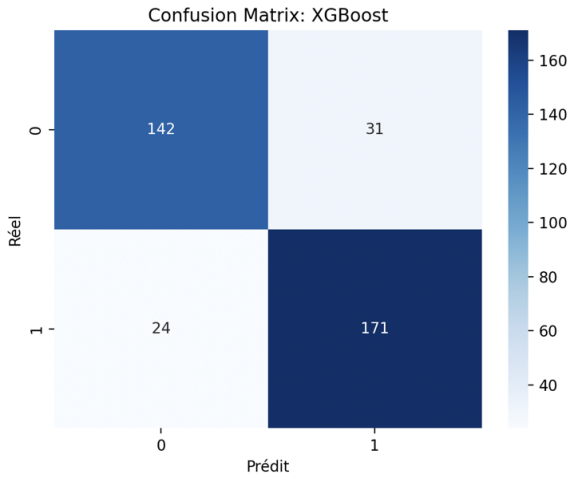

FIGURE 26 – Confusion matrix for XGBoost

The XGBoost model performs slightly better than **AdaBoost**, with an overall F1-score of 0.85. This result reflects a good balance between precision and recall for both classes.

For class 0 (no disease), the precision is 0.86 and recall is 0.82, indicating good ability to avoid false positives. For class 1 (heart disease), the recall is high at 0.88, which means that XGBoost is very effective in detecting actual cases — a key aspect in medical diagnostics.

The confusion matrix shows 24 false negatives and 31 false positives — better than AdaBoost, particularly in reducing false negatives, which is crucial to avoid missing real cases of disease.

14.5 Overall Comparison

After analyzing each model individually (**CART**, **AdaBoost**, and **XGBoost**), we summarize their classification performance (precision, recall, F1-score, and accuracy) in the following table :

Model	Precision	Recall	F1-Score	Accuracy
CART	0.77	0.77	0.77	0.77
AdaBoost	0.83	0.83	0.83	0.83
XGBoost	0.85	0.85	0.85	0.85

TABLE 17 – Model results for heart disease prediction

The joint evaluation of the three models — CART, AdaBoost, and XGBoost — reveals notable differences in performance, stability, and relevance in the medical prediction context.

The CART model, based on simple decision trees, offers a fast and interpretable solution. However, its performance is limited (F1-score of 0.77), and it tends to generate more false positives and false negatives — a critical drawback for disease detection.

Introducing boosting with AdaBoost brings significant improvement. By combining multiple weak learners, AdaBoost progressively corrects the base model's errors. It significantly improves

performance (F1-score of 0.83) while reducing critical errors, showing good generalization without excessive overfitting.

XGBoost takes this approach even further, incorporating regularization, second-order optimization (via Taylor expansion), and efficient handling of missing data. It delivers the best results (F1-score of 0.85), maintaining balance between precision and recall. It also further reduces false negatives — essential in medical contexts where missing a diagnosis can be dangerous.

Overall, the results show a consistent progression : each model improves upon the previous. CART sets the baseline, AdaBoost enhances it, and XGBoost consolidates it. This clearly justifies the use of ensemble methods in medical classification tasks, especially when model sensitivity (recall) is paramount.

15 Conclusion

This research explored in depth the foundations, mechanisms, and performance of boosting algorithms applied to supervised learning — both in regression and classification contexts. Starting from the base CART model, we highlighted the limitations of standalone decision trees, such as overfitting and lack of robustness with complex or noisy data.

Introducing AdaBoost marked a significant advancement. By combining several weak classifiers in a sequential and weighted manner, the algorithm greatly improves model generalization. We demonstrated its foundation in probabilistic interpretation and exponential loss minimization. However, we also noted its sensitivity to outliers.

To address these issues, XGBoost stands out as a state-of-the-art solution. Building on gradient boosting principles, it incorporates advanced techniques such as regularization, Taylor-series-based optimization, and smart handling of missing data. Our experiments confirm its superior performance — especially where minimizing critical errors like false negatives is vital, as in medical applications.

Importantly, this work was not limited to empirical study. It also drew on rigorous theoretical formalism of boosting — including Schapire's theorem and additive modeling — providing a deeper understanding of how algorithms like AdaBoost and XGBoost improve weak learners by progressively focusing on previous errors. This theory-practice articulation was a core thread of the report.

More broadly, this research project allowed us to apply theoretical concepts from statistical learning through a rigorous comparative study and concrete implementations. It demonstrates how modern algorithmic advances — especially in boosting — can effectively address supervised classification challenges.

Appendix

```python
import numpy as np
import matplotlib.pyplot as plt

# Define a vector of margins yf(x)
margin = np.linspace(-2, 2, 400)

# Define loss functions
def exponential_loss(yf):
    return np.exp(-yf)

def deviance_loss(yf):
    return np.log(1 + np.exp(-2 * yf))

# Compute losses
loss_exp = exponential_loss(margin)
loss_dev = deviance_loss(margin)

# Plot graphs
plt.figure(figsize=(8, 5))
plt.plot(margin, loss_exp, label="Exponential", color="skyblue", linewidth=2)
plt.plot(margin, loss_dev, label="Binomial Deviance", color="orange", linewidth=2)

# Formatting
plt.xlabel("Margin $yf(x)$")
plt.ylabel("Loss")
plt.title("Comparison of Loss Functions")
plt.legend()
plt.tight_layout()
plt.show()
```

Listing 3 – Python code used to plot exponential and binomial deviance loss functions as a function of the margin

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from xgboost import XGBClassifier
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
df = pd.read_csv("heart.csv")

print(df.columns.tolist())

# Manually select explanatory variables
features = [
    'Age',
    'RestingBP',  # Resting blood pressure (mm/Hg)
    'Cholesterol',  # Serum cholesterol level (mg/dL)
    'FastingBS',  # Fasting blood sugar
    'MaxHR',  # Maximum heart rate achieved during exercise
    'Oldpeak',  # ST segment depression induced by exercise
    'Sex_M',
    'ChestPainType_ATA',  # Chest pain type ATA
    'ChestPainType_NAP',  # Chest pain type NAP
    'ChestPainType_TA',   # Chest pain type TA
    'RestingECG_Normal',  # Normal ECG
    'RestingECG_ST',      # Abnormal ECG with ST-T wave changes
    'ExerciseAngina_Y',   # Exercise-induced angina
    'ST_Slope_Flat',      # Flat ST segment slope
    'ST_Slope_Up'         # Up-sloping ST segment
]

# X = features, Y = target (heart disease)
X = df[features]
```

```
Y = df['HeartDisease']

# Split the data (60% training, 40% testing)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4, random_state
    =14)

# Evaluation function
def eval_model(Y_true, Y_pred, title):
    print(f"--- {title} ---")
    print(classification_report(Y_true, Y_pred))
    sns.heatmap(confusion_matrix(Y_true, Y_pred), annot=True, fmt='d', cmap='Blues')
    plt.title(f"Confusion Matrix: {title}")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

# CART Model
cart = DecisionTreeClassifier(random_state=14)
cart.fit(X_train, Y_train)
Y_pred_cart = cart.predict(X_test)
eval_model(Y_test, Y_pred_cart, "CART")

# AdaBoost Model
ada = AdaBoostClassifier(n_estimators=100, random_state=14)
ada.fit(X_train, Y_train)
Y_pred_ada = ada.predict(X_test)
eval_model(Y_test, Y_pred_ada, "AdaBoost")

# XGBoost Model
xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=14)
xgb.fit(X_train, Y_train)
Y_pred_xgb = xgb.predict(X_test)
eval_model(Y_test, Y_pred_xgb, "XGBoost")
```

Listing 4 – Python code to train and evaluate CART, AdaBoost, and XGBoost models on heart disease data

Références

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

[2] StatQuest with Josh Starmer. *Regression trees clearly explained*. Youtube 2019. Disponible à l'adresse : https://www.youtube.com/watch?v=g9c66TUylZ4&t=41s&ab_channel=StatQuestwithJoshStarmer

[3] StatQuest with Josh Starmer. *Decision and classification trees clearly explained*. Youtube 2021. Disponible à l'adresse : https://www.youtube.com/watch?v=_L39rN6gz7Y&ab_channel=StatQuestwithJoshStarmer

[4] Robert E. Schapire. *The strength of weak learnability*. Volume 5, pages 197-227 Springer, 1990. Disponible à l'adresse : https://link.springer.com/article/10.1007/BF00116037

[5] Ricco Rakotomalala. *Gradient Boosting – Technique ensembliste pour l'analyse prédictive*. Université Lumière Lyon 2, 2020. Disponible à l'adresse : https://eric.univ-lyon2.fr/ricco/cours/slides/gradient_boosting.pdf

[6] Objectif Data Science avec Vincent. *XGBoost : l'algorithme star de machine learning décortiqué*. YouTube, 2022. Disponible à l'adresse : https://www.youtube.com/watch?v=iD_TL725wZs

[7] StatQuest with Josh Starmer. *XGBoost Part 1 (of 4) : Régression* . YouTube, 2020. Disponible à l'adresse : https://www.youtube.com/watch?v=OtD8wVaFm6E

[8] StatQuest with Josh Starmer. *XGBoost Part 2 : Arbres XGBoost pour la classification* . YouTube, 2020. Disponible à l'adresse : https://www.youtube.com/watch?v=8b1JEDvenQU

[9] Kaggle. *Heart Failure Prediction Dataset*, consulté en mai 2025. Disponible à l'adresse : https://www.kaggle.com/code/tanmay111999/heart-failure-prediction-cv-score-90-5-models/input