



# Introduction à l'Apprentissage Statistique

---

## Boosting

M2 Actuariat – ISFA – 2021/2022

Pierrick Piette  
Actuaire à Seyna  
[pierrick.piette@gmail.com](mailto:pierrick.piette@gmail.com)

**Seyna.**

In God we trust, all others bring data  
- William Edwards Deming





# Principes

# Ensemble learning



# Introduction

Nous avons vu un exemple de combinaison de modèles basée sur une stratégie aléatoire, permettant ainsi de diminuer la variance de l'estimateur.

L'enjeu de l'agrégation par boosting est tout à fait différent

- Il s'agit d'une stratégie adaptative

Améliore l'ajustement, i.e. le biais, par

- une construction adaptative séquentielle d'estimateurs
- puis une combinaison de ces estimateurs pour éviter le surapprentissage

Remarque : on analyse les CART en particulier car ils sont instables, mais valable pour toutes les modélisations

# Evolution par rapport au Bagging

En plus de traiter la réduction de variance, **on traite aussi le problème du biais !**

En effet, l'agrégation par bagging ne corrige pas le biais, puisque l'espérance est un opérateur linéaire, et le biais est défini par une espérance.

Or dans le cas d'arbres individuels simples, le biais peut être important.

Le boosting construit une famille de modèle récurrente : chaque modèle est une version adaptative du précédent.

# Procédure du Boosting

Grandes étapes du boosting

1. On estime le modèle  $m_1$  pour  $y$  à partir de  $x$ . On en déduit le vecteur d'erreurs  $\epsilon_1$ .
2. On estime le modèle  $m_2$  pour  $\epsilon_1$  à partir de  $x$ . On en déduit le vecteur d'erreurs  $\epsilon_2$ .
3. On itère  $k$  fois pour obtenir l'estimateur agrégé

$$m^{(k)}(x) = m_1(x) + m_2(x) + \cdots + m_k(x) = m^{(k-1)}(x) + m_k(x)$$

# Stratégie adaptative

Pour s'adapter de proche en proche, on donne **plus de poids dans l'estimation suivante aux observations mal prédites** précédemment.

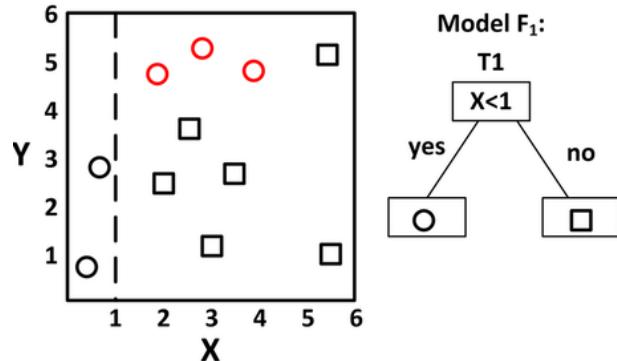
Intuitivement, l'algorithme concentre ses efforts sur les observations les plus difficiles à ajuster, tout en limitant l'overfitting par l'agrégation

Les différents algorithmes de boosting diffèrent par leurs caractéristiques

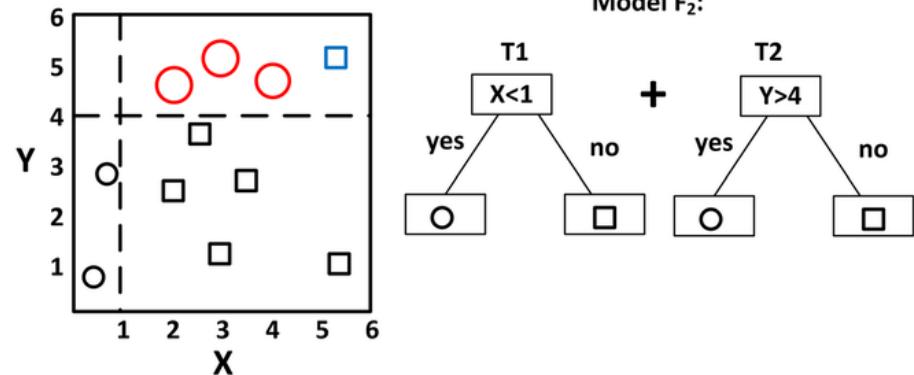
- La façon de pondérer l'importance des individus mal estimés
- La façon de pondérer les modèles lors de l'agrégation
- Leur objectif (prédirer  $Y$  réelle, binaire, etc.)
- La fonction de perte qui mesure l'erreur d'ajustement

# Illustration graphique

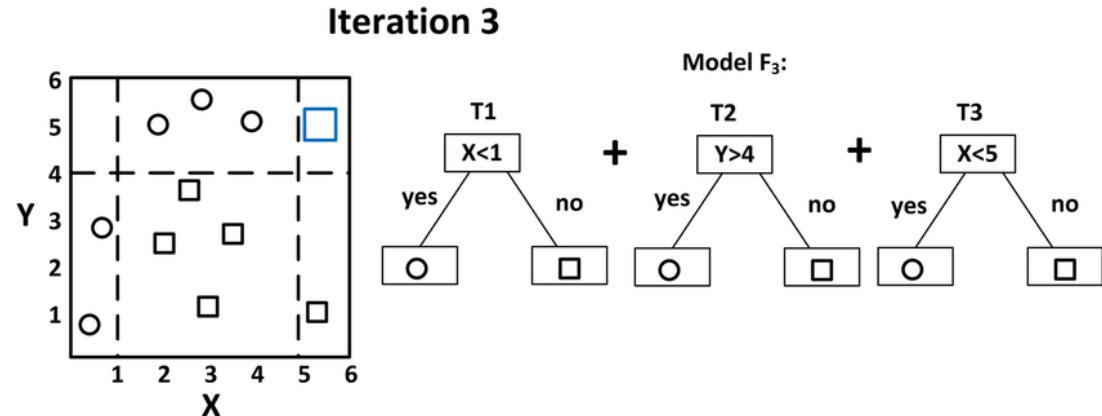
**Iteration 1**



**Iteration 2**



**Iteration 3**





# Adaboost

# Algorithme Adaboost (1/2)

Au départ, cet algorithme est proposé pour une **classification sur 2 classes**

Notons  $h$  la fonction de discrimination à valeurs dans  $\{-1,1\}$

## Algorithme

1. Soit  $y_0$  à prévoir (connaissant  $x_0$ ) et  $\mathcal{D} = \{(x_1, y_1), \dots, (x_n, y_n)\}$  un échantillon d'apprentissage
2. On initialise les poids par équipondération

$$\omega = \left\{ \omega_i = \frac{1}{n}; i = 1, \dots, n \right\}$$

# Algorithme Adaboost (2/2)

3. Pour  $m = 1, \dots, M$ 
  - a. On estime  $h_m$  sur l'échantillon pondéré par  $\omega$
  - b. On calcule le taux d'erreur apparent :  $\hat{\epsilon}_m = \frac{\sum_{i=1}^n \omega_i \mathbb{I}\{h_m(x_i) \neq y_i\}}{\sum_{i=1}^n \omega_i}$
  - c. On calcule les logits relatifs au modèle  $h_m$  :  $c_m = \ln\left(\frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}\right)$
  - d. On met à jour les pondérations  $\omega_i \leftarrow \omega_i \exp(c_m \mathbb{I}\{h_m(x_i) \neq y_i\})$
  - e. On passe au modèle suivant  $m \leftarrow m + 1$

4. Résultat du vote

$$\hat{F}_M(x_0) = \text{signe} \left[ \sum_{m=1}^M c_m h_m(x_0) \right]$$

# Remarques

Il faut vérifier à chaque étape que le modèle incrémental fait mieux qu'une prévision aléatoire, i.e.

$$\hat{\epsilon}_m < 0.5$$

Sinon le poids  $c_m$  du modèle correspondant devient négatif !

De nombreuses adaptations de cet algorithme ont été proposées, avec des fonctions de perte adaptées aux cas où

- $Y$  qualitative à plusieurs modalités
- $Y$  quantitative
- ...

# Approximation des pondérations

Parfois, on utilise des classifieurs pour lesquels il est difficile d'intégrer une pondération des observations

La stratégie revient à créer aléatoirement des échantillons

- Chaque modèle sera construit sur un nouvel échantillon
- La probabilité de tirer (avec remise) chaque observation est inversement proportionnelle à sa qualité d'ajustement dans l'itération précédente

C'est ce qu'on appelle des **arcing classifiers** (adaptively **resample and combine**)



# Gradient Boosting

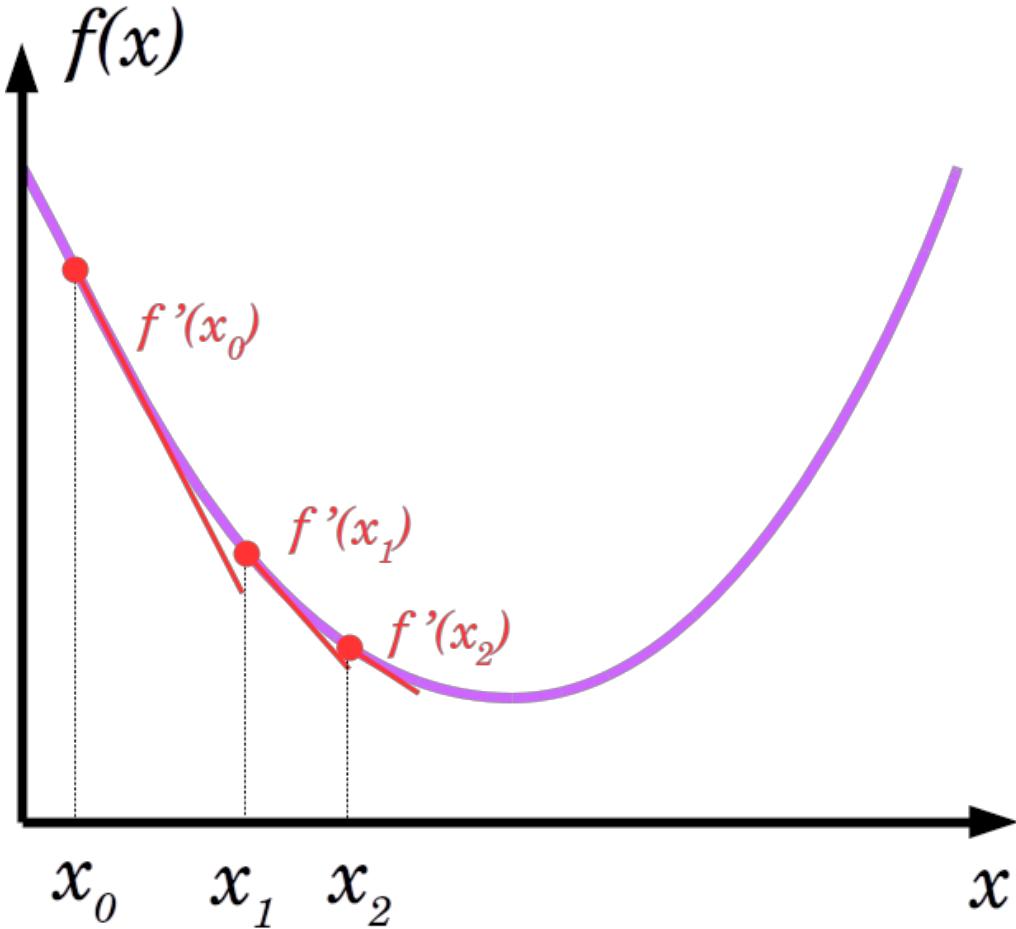
# Descente de Gradient

La descente de gradient est un algorithme d'optimisation destiné à minimiser une fonction réelle différentiable  $f$ .

Algorithme itératif fonctionnant par amélioration successives. On part d'un point  $x_0$  initial et on se fixe un seuil de tolérance  $\epsilon > 0$ . Le passage du point  $x_k$  à  $x_{k+1}$  est déterminé par :

1. **Simulation** : calcul de  $\nabla f(x_k)$
2. **Test d'arrêt** : si  $|\nabla f(x_k)| \leq \epsilon$  alors arrêt de l'algorithme
3. **Calcul du pas** :  $\alpha_k > 0$
4. **Nouveau point** :  $x_{k+1} \leftarrow x_k - \alpha_k \nabla f(x_k)$

# Descente de Gradient



# Gradient Boosting

1. Initialisation du modèle

$$F_0(\boldsymbol{x}) = \arg \min_{\rho} \sum_i^n L(y_i, \rho)$$

2. Pour  $m = 1, \dots, M$

a. Calcul des pseudo-résidus

$$\tilde{y}_{i,m} = - \left[ \frac{\partial L(y_i, F(\boldsymbol{x}_i))}{\partial F(\boldsymbol{x}_i)} \right]_{F=F_{m-1}}$$

b. Estimation du weak learner  $h(\boldsymbol{x}, \alpha_m)$  sur les pseudo-résidus  $\{\boldsymbol{x}_i, \tilde{y}_{i,m}\}_{i=1}^n$

c. Estimation du multiplicateur optimum sur les données initiales  $\{\boldsymbol{x}_i, y_{i,m}\}_{i=1}^n$

$$\rho_m = \arg \min_{\rho} \sum_i^n L(y_i, F_{m-1}(\boldsymbol{x}_i) + \rho h(\boldsymbol{x}_i, \alpha_m))$$

d. Incrémentation de l'estimateur  $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \rho_m h(\boldsymbol{x}, \alpha_m)$

# Gradient Tree Boosting

L'algorithme de Gradient Boosting est le plus souvent utilisé avec des arbres de décisions :

- le weak learner  $h(x, \alpha)$  est défini comme un arbre avec  $L$  feuilles
- le « paramètre »  $\alpha$  à optimiser peut être décrit comme l'ensemble  $\{\omega_{l,m}, R_{lm}\}_{l=1}^L$
- $R_{lm}$  les régions de l'espace définies par les feuilles
- $\omega_{l,m}$  la valeur réponse associée (parfois dénommé *weight* en anglais)

Pour limiter le surapprentissage, on impose généralement dans la pratique un taux d'apprentissage  $\eta < 1$  tel que

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \eta \times \rho_m h(\mathbf{x}, \alpha_m)$$

# Tuning des hyperparamètres

Le tuning des hyperparamètres commence à être une question opérationnelle

- Le nombre de feuilles des weak learners :  $L$
- Le nombre de weak learners à agréger :  $M$
- Le taux d'apprentissage incrémental :  $\eta$

Permet d'aborder l'arbitrage under/over-fitting avec différentes stratégies, selon l'étude réalisée, e.g.

- Weak learners nombreux, mais avec un fort biais et un apprentissage lent
- Un apprentissage rapide avec des weak learners de bonne précision mais peu nombreux



# XGBoost

# Stochastic Gradient Boosting

Dans la recherche du meilleur compromis biais-variance, des variantes du gradient boosting ont rapidement été développées

## Le Stochastic Gradient Boosting

- Reprend l'idée du bagging
- pour l'intégrer au gradient boosting
- A chaque itération, le weak learner  $h_m$  est entraîné non pas sur l'ensemble des données...
- ... mais sur un sous-échantillon (tirage sans remise)

## Double effet bénéfique

- Améliore la robustesse de l'algorithme
- Améliore les temps de calcul

# XGBoost

## eXtreme Gradient Boosting

Librairie publiée en 2015

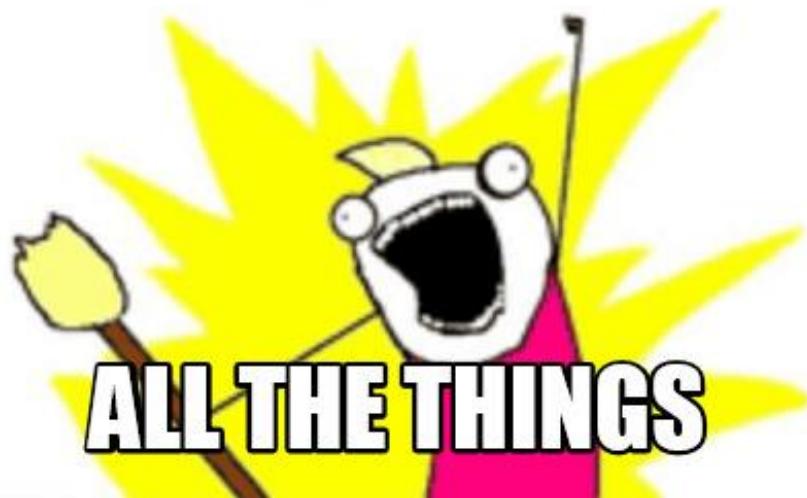
- Algorithme de Tree Boosting
- Augmenté avec de nombreuses options (et donc de nombreux hyperparamètres)
- Permet d'aborder avec plus de détails l'arbitrage biais-variance pour trouver le meilleur modèle

Grande popularité dès sa sortie

- Développé en C++ et disponible sur R et Python
- Rapidité de calcul (possibilité de paralléliser les calculs facilement CPU et GPU)
- « Why does XGBoost win every machine learning competition ? »

# XGBoost

# XGBOOST



# XGBoost – Fonction de perte objective

Fonction de perte de l’XGBoost est redéfinie comme une fonction objective

$$obj = L + \Omega$$

avec  $L$  fonction de perte initiale et  $\Omega$  une fonction de régularisation

$$\Omega(f) = \sum_{m=1}^M \left( \gamma L_m + \frac{1}{2} \lambda \|\omega_m\|_2^2 + \alpha \|\omega\|_1 \right)$$

1. **gamma** (default=0) : pénalisation sur le nombre de feuilles  $L$  de chaque arbre, équivalent au gain minimum (fonction de perte  $L$ ) pour chaque nouveau nœud
2. **lambda** (default =1) : pénalisation en norme L2 sur les valeurs réponses
3. **alpha** (default = 0) : pénalisation en norme L1 sur les valeurs réponses

# XGBoost – Stochastic jungle

Comme pour les forêt aléatoires, XGBoost va rajouter des tirages aléatoires sur les observations et les covariables afin de décorrélérer les arbres entre eux

4. `subsample` (default = 1) : ratio du nombre d'observations utilisées pour train chaque arbre. Un tirage pour chaque itération  $m$ .
5. `colsample_bytree` (default = 1) : ratio du nombre de covariables utilisées pour train chaque arbre. Un tirage pour chaque itération  $m$
6. `colsample_bynode` (default = 1) : ratio du nombre de covariables utilisées pour chaque nœud (par rapport au nombre de covariables de l'arbre). Un tirage à chaque nœud.

# XGBoost – Arbre et apprentissage

Enfin, des hyperparamètres sont aussi présent pour « forcer manuellement » la taille des arbres, sans oublier le taux d'apprentissage du boosting

7. **max\_depth** (default = 6) : taille maximal des arbres (nombre de feuille).  
Attention à l'impact sur la consommation de mémoire.
8. **min\_child\_weight** (default = 1) : poids (= # observations en équipondération) minimum pour la création d'un nœud fils.
9. **eta** (default = 0.3) : taux d'apprentissage appliqué à chaque arbre sur sa contribution finale.
10. **nrounds** : nombre d'itération  $M$  à estimer