Gradient Boosting

M2 MAS

Université de Rennes 2

2025

Programme du cours

1. Introduction : méthodes d'ensemble, bagging, vers le boosting.
2. Boosting : weak learners, AdaBoost, motivations vers le gradient.
3. Gradient Boosting : algorithme, descente de gradient, fonction de perte.
4. Choix des paramètres et bonnes pratiques.
5. TD

Méthodes d'ensemble : panorama rapide

- Objectif : combiner plusieurs modèles faibles ou instables pour améliorer **performance** et **robustesse**.
- Deux grandes familles :
  - **Bagging**
  - **Boosting**
- Idée générale : exploiter la "complémentarité" de modèles imparfaits.

Rappel sur le bagging : Agrégation d'algorithmes simples

Idée : construire un grand nombre d'algorithmes "simples" et les agréger pour obtenir une seule prévision.

$$T_1(x, D_{n,1}), \quad T_2(x, D_{n,2}), \quad \ldots, \quad T_B(x, D_{n,B})$$

$$f_n(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x, D_{n,b})$$

Questions clés :

1. Comment choisir les échantillons $D_{n,b}$?
2. Comment choisir les algorithmes T_b ?
3. Quelle taille B pour la moyenne ?

Cadre statistique et notations

Cadre : régression simplifiée

- (X, Y) : couple aléatoire avec $X \in \mathbb{R}^d$, $Y \in \mathbb{R}$.
- Échantillon i.i.d. : $D_n = \{(X_i, Y_i)\}_{i=1}^n$.
- Algorithme de prédiction : $f_n(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$.
- Hypothèse : $T_1, \ldots, T_B$ identiquement distribués conditionnellement à D_n.

Algorithme Bagging (Breiman 1996)

Bootstrap Aggregating (Bagging) :

1. Pour $b = 1, \ldots, B$:
   1. Tirer un échantillon bootstrap $D^*_{n,b}$ de taille n avec remise.
   2. Entraîner T sur $D^*_{n,b} \Rightarrow T(x, \theta_b, D_n)$
2. Agréger les prédictions :

$$f_n(x) = \frac{1}{B} \sum_{b=1}^{B} T(x, \theta_b, D_n)$$

Propriétés :

- $B \to \infty \implies f_n(x) \to \mathbb{E}_\theta[T(x, \theta, D_n)]$ (stabilisation)
- L'algorithme devient robuste aux fluctuations aléatoires.

Création d'un échantillon bootstrap

Échantillon initial : $D_n = \{1, 2, 3, 4, 5\}$
Tirage bootstrap (taille $n = 5$) avec remise :

$$D^*_{n,1} = \{2, 3, 2, 5, 1\} \quad \text{(indices tirés aléatoirement parmi } \{1, 2, 3, 4, 5\})$$

Points clés :

- Chaque tirage est fait **avec remise** certains points apparaissent plusieurs fois, d'autres pas
- Taille du bootstrap = taille de l'échantillon original n
- Permet d'introduire de la variation entre les modèles T_b pour l'agrégation

Variance et corrélation des estimateurs

Soit $T_1(x), \ldots, T_B(x)$ identiquement distribués, avec corrélation

$$\rho(x) = \mathrm{corr}[T_1(x), T_2(x)].$$

Alors :

$$\mathbb{E}[f_n(x)] = \mathbb{E}[T_1(x)]$$

$$\mathrm{Var}[f_n(x)] = \rho(x)\,\mathrm{Var}[T_1(x)] + \frac{1-\rho(x)}{B}\mathrm{Var}[T_1(x)]$$

Conséquences :

- Le biais n'est pas modifié.
- La variance diminue lorsque $B \to \infty$ et $\rho(x) \to 0$.
- Répéter le même algorithme sur le même échantillon est inutile.
- L'intérêt du bootstrap : produire des modèles corrélés faiblement (ρ petit) pour bénéficier de l'agrégation.

Forêts aléatoires : principes et variantes

- Construire des arbres de décision sur échantillons bootstrap (bagging).
- À chaque noeud, au lieu de regarder toutes les variables, on choisit au hasard un sous-ensemble de m caractéristiques (*feature subsampling*).
- Choix courant : $m = \sqrt{p}$ pour classification, $m = p/3$ pour régression (règles empiriques).
- Avantage : baisse de la corrélation entre arbres (diminue ρ dans la formule précédente) ⇒ meilleure réduction de variance.

Remarques pratiques :

- Robustesse aux outliers, peu d'overfitting si arbres non taillés.
- Importance des variables pour interprétabilité partielle.
- Estimer l'erreur OOB (out-of-bag) : estimateur interne de généralisation.

Biais / Variance : interprétation et limites du bagging

Rappel : l'erreur quadratique espérée se décompose en :

$$\mathbb{E}\big[(Y - \hat{f}(X))^2\big] = \underbrace{\big(f(X) - \mathbb{E}[\hat{f}(X)]\big)^2}_{\text{biais}^2} + \underbrace{\mathrm{Var}[\hat{f}(X)]}_{\text{variance}} + \mathrm{Var}(\varepsilon).$$

- Le bagging réduit la **variance** sans toucher au biais (la moyenne garde l'espérance de l'estimateur).
- Si le modèle de base a un **fort biais** (sous-adaptation), le bagging ne corrigera pas ce biais.

Pourquoi aller vers le boosting ?

- Limite constatée : si le modèle de base est biaisé, la simple agrégation (bagging) ne réduit pas le biais.
- Objectif : diminuer le biais en construisant une combinaison **séquentielle** de modèles qui corrigent explicitement les erreurs précédentes.
- Caractéristiques du boosting :
  - Construction itérative (modèles dépendants les uns des autres).
  - Chaque nouveau modèle vise à corriger les erreurs résiduelles.

Boosting

Weak Learners, stump, AdaBoost, ...

Idée clé : Boosting et Weak Learners

Idée générale :

- Un **weak learner** (ou apprenant faible) est un modèle simple, à peine meilleur que le hasard.
- Exemple : dans une tâche de classification binaire équilibrée, il a une précision juste supérieure à 50 %.

Exemples de weak learners :

- Arbre de décision de profondeur 1 (*decision stump*).
- Régression logistique simple sur une seule variable.

Propriétés des weak learners :

- **Biais élevé** : le modèle est trop simple pour bien capturer la structure des données.
- **Variance faible** : peu sensible aux fluctuations de l'échantillon d'apprentissage.
- **Faible performance individuelle**, mais excellente brique de base pour le **boosting**.

Choix du Weak Learner pour le Gradient Boosting

Question : Quel type de modèle utiliser comme *weak learner* dans le cadre du gradient boosting ?

Plusieurs candidats possibles :

- **Régression linéaire / logistique :**
  - Faible variance, facile à entraîner.
  - Hypothèse de linéarité trop restrictive, peu adaptés pour approximer des gradients complexes.
- **Perceptron (un seul neurone) :**
  - Simple, interprétable.
  - Apprentissage instable, très dépendant du taux d'apprentissage.
- **k-plus proches voisins (k-NN) :**
  - Intuitif, sans paramétrage lourd.
  - Non différentiable et non paramétrique..
- **Petits réseaux de neurones :**
  - Peuvent capturer des relations non linéaires.
  - Trop lents à entraîner à chaque itération du boosting.

Pourquoi les arbres comme Weak Learners ?

Les arbres de décision présentent plusieurs avantages :

- **Flexibilité** : captent naturellement les relations non linéaires, sans nécessiter de transformation des variables.
- **Gestion des interactions** : identifient automatiquement les interactions entre variables.
- **Faible variance** lorsqu'ils sont peu profonds.
- **Peu sensibles à l'échelle** des variables (pas besoin de normalisation).
- **Rapides à entraîner** et facilement interprétables.

En pratique :

- Chaque arbre utilisé comme weak learner a une **profondeur modérée** (généralement entre 3 et 5).
- Ce compromis offre un **biais modéré** et une **variance contrôlée**, ce qui en fait une base idéale pour le boosting.

Historique : des Weak Learners à AdaBoost

Question de départ : *"Peut-on combiner plusieurs modèles faibles pour obtenir un modèle fort ?"* (Kearns et Valiant, 1988)

Première réponse : AdaBoost (Freund et Schapire, 1997)

- Entraîne une suite de **modèles faibles** (souvent des arbres de décision peu profonds).
- À chaque itération :
  - Les observations sont **repondérées** : les exemples mal classés voient leur poids augmenter.
  - Le nouveau modèle se concentre sur les erreurs des modèles précédents.
- Le modèle final est une **combinaison pondérée** de tous les modèles successifs.

Idée clé : Transformer une collection de modèles faibles, chacun légèrement meilleur que le hasard, en un modèle globalement très performant.

Limites et sensibilités d'AdaBoost

Sensibilité au bruit :

- Les points mal étiquetés se voient attribuer des poids élevés.
- Cela peut fortement dégrader la performance du modèle.

Limites principales :

- **Fonction de perte rigide** : AdaBoost minimise implicitement une perte exponentielle, ce qui peut amplifier les erreurs.
- **Peu flexible** : difficile à adapter à d'autres tâches, comme la régression ou l'estimation de quantiles.

Vers le Gradient Boosting

Idée clé :

- Plutôt que de corriger les erreurs par repondération (comme dans AdaBoost), on peut les corriger en suivant la **direction du gradient** de la fonction de perte.

Friedman (2001) :

- Le boosting peut être vu comme une **descente de gradient fonctionnelle**, où chaque weak learner approxime le gradient de la perte.

Avantages du Gradient Boosting :

- Utilisation possible de **toutes sortes de fonctions de perte différentiables**.
- Contrôle fin de l'apprentissage : *learning rate*, régularisation, nombre d'itérations.
- **Robuste au bruit** comparé à AdaBoost.

Petit résumé : AdaBoost vs Gradient Boosting

Idée générale :

- Combiner plusieurs modèles faibles (*weak learners*) pour créer un modèle fort.

AdaBoost :

- Corrige les erreurs en **repondérant les observations**.
- Chaque arbre apprend à mieux prédire les exemples mal classés.
- Simple et efficace, mais **sensible au bruit**.

Gradient Boosting (Friedman, 2001) :

- Corrige les erreurs en suivant le **gradient de la fonction de perte**.
- Plus **robuste** et plus **flexible** qu'AdaBoost.

En résumé :

- **AdaBoost** : ajuste les **poids des données**.
- **Gradient Boosting** : ajuste les **prédictions elles-mêmes**.

Gradient Boosting

L'algorithme étape par étape

Intuition générale du Gradient Boosting

Principe fondamental :

- Le Gradient Boosting construit une **série de modèles**, où chaque nouveau modèle se concentre sur la correction des erreurs commises par l'ensemble des modèles précédents.
- Plutôt que d'apprendre directement la cible, on apprend **itérativement les résidus** : les écarts non expliqués par les modèles précédents.

Formulation mathématique du Gradient Boosting

Étapes principales :

1. **Initialisation :** On commence avec un modèle simple $F_0(x)$, souvent une constante.
2. **Itération :** À chaque étape m, on entraîne un nouveau modèle $h_m(x)$ qui approxime le **gradient négatif** de la fonction de perte par rapport aux prédictions actuelles.
3. **Mise à jour :** Le modèle global devient :

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x),$$

où ν est le **taux d'apprentissage (learning rate)**.

Remarque : Cette approche réalise une **descente de gradient dans l'espace fonctionnel** plutôt que dans l'espace des paramètres.

Espace de fonctions et descente de gradient fonctionnelle

Optimisation classique vs Gradient Boosting :

- En optimisation classique, on ajuste des **paramètres** d'un modèle.
- En Gradient Boosting, on optimise dans un **espace de fonctions**.

Principe d'ajout de fonction :

- À chaque étape, on se déplace dans la direction qui réduit le plus le gradient de la fonction de perte $L(y, F(x))$.
- On ajoute une **nouvelle fonction** (weak learner) plutôt que de modifier des poids ou des paramètres existants.

Fonction de perte et résidus

Exemple : perte quadratique (L2)

- La perte quadratique mesure l'écart au carré entre la prédiction et la valeur réelle :

$$L(y, F(x)) = \frac{1}{2}(y - F(x))^2$$

- Le gradient négatif de la perte par rapport à $F(x)$ est :

$$-\frac{\partial L(y, F(x))}{\partial F(x)} = y - F(x)$$

- On voit que ce gradient correspond exactement aux **résidus** :

$$r_i = y_i - F_{m-1}(x_i)$$

- Chaque nouveau modèle $h_m(x)$ apprend donc à approximer ces résidus pour améliorer la prédiction globale :

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

Le résidu comme signal d'apprentissage

Prédiction actuelle :

- Le modèle $F_{m-1}(x)$ produit une estimation.

Calcul du résidu :

$$r_i = y_i - F_{m-1}(x_i) \quad \text{pour chaque exemple}$$

Apprentissage :

- Chaque nouvel arbre $h_m(x)$ est entraîné pour prédire ces résidus.
- Pour la perte L^2, le processus est particulièrement intuitif : chaque nouvel arbre prédit directement ce qui manque aux prédictions précédentes.

Approximation du gradient par un arbre

Dans le Gradient Boosting, on utilise généralement des **arbres de décision peu profonds (weak learners)** pour approximer le gradient de la perte.
Avantages :

- Capturent naturellement les **interactions non-linéaires** entre variables.
- Gèrent facilement les **variables mixtes** (numériques et catégorielles).
- Leur **faible profondeur** limite le surapprentissage et favorise la régularisation.

Lien avec la descente de gradient classique

Descente de gradient standard

- Optimisation dans un espace **paramétrique**
- Mise à jour : $\theta \leftarrow \theta - \eta \nabla L(\theta)$
- Gradient calculé **analytique**
- Direction de descente **exacte**

Descente de grad' fonctionnelle

- Optimisation dans un espace **de fonctions**
- Mise à jour : $F \leftarrow F + \nu \cdot h(x)$
- Gradient **approximé par un modèle**
- Direction de descente **approchée via le weak learner**

Gradient Boosting : schéma

Étape 0 : Initialisation

Modèle simple $F_0(x)$

Étape m : Calcul des résidus

$$r_i = y_i - F_{m-1}(x_i)$$

Entraînement du nouvel arbre $h_m(x)$

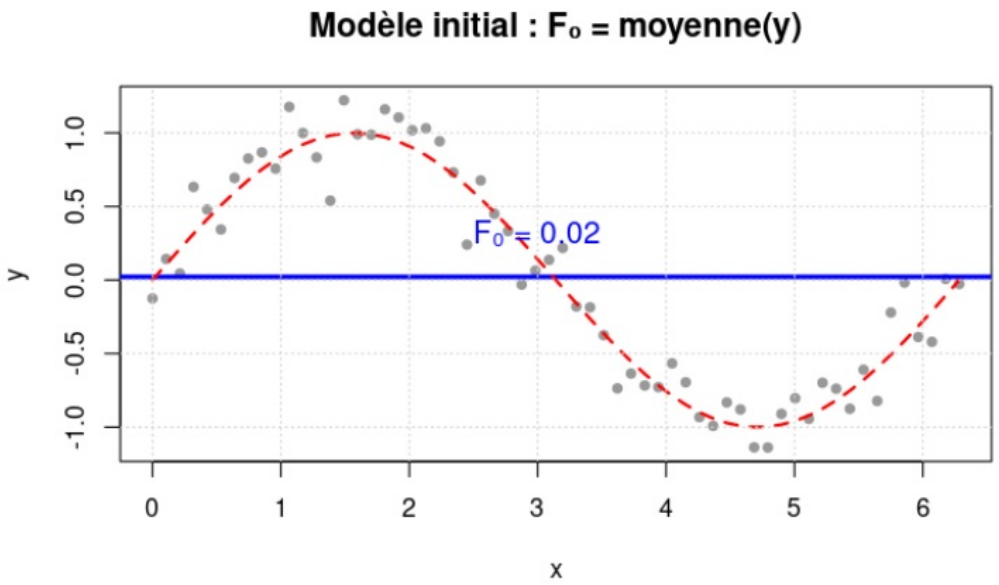
Mise à jour : $F_m(x) = F_{m-1}(x) + \nu h_m(x)$

Répéter jusqu'à convergence

Convergence : Modèle final

$$F_M(x) = F_0(x) + \nu \sum_{m=1}^{M} h_m(x)$$

Gradient Boosting : Étape 0 Initialisation



Les fondamentaux :

- Une fonction de perte (MSE)
- Initialisation $F_0(x)$ avec une constante
- Choix "intuitif" : la **moyenne des observations**

Figure – 30 points bruités de $\sin(x)$ et initialisation $F_0(x)$

Gradient Boosting : Étape 1 Calcul des résidus et mise à jour

A) Calcul des pseudo-résidus :

$$r_{m,i} = y_i - F_{m-1}(x_i)$$

- Mesure l'erreur de notre modèle au point i

B) Apprentissage d'un weak learner sur les résidus :

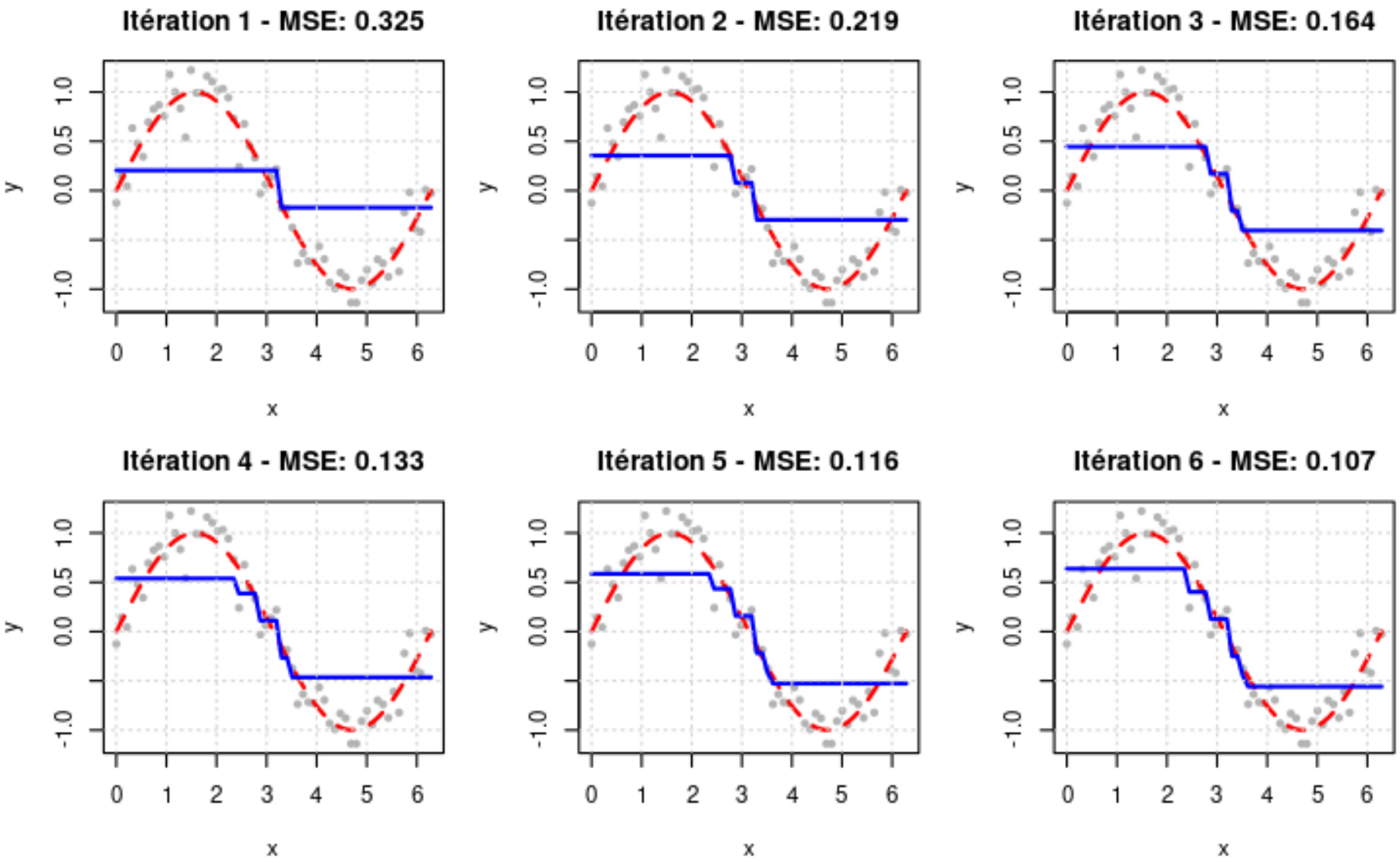
- On entraîne un weak learner pour prédire les résidus $r_{m,i}$
- Exemple : un **stump** (arbre de profondeur 1)
- On cherche la meilleure coupure c^* minimisant :

$$\text{erreur}_c = \sum_i (r_{m,i} - \hat{r}_i)^2$$

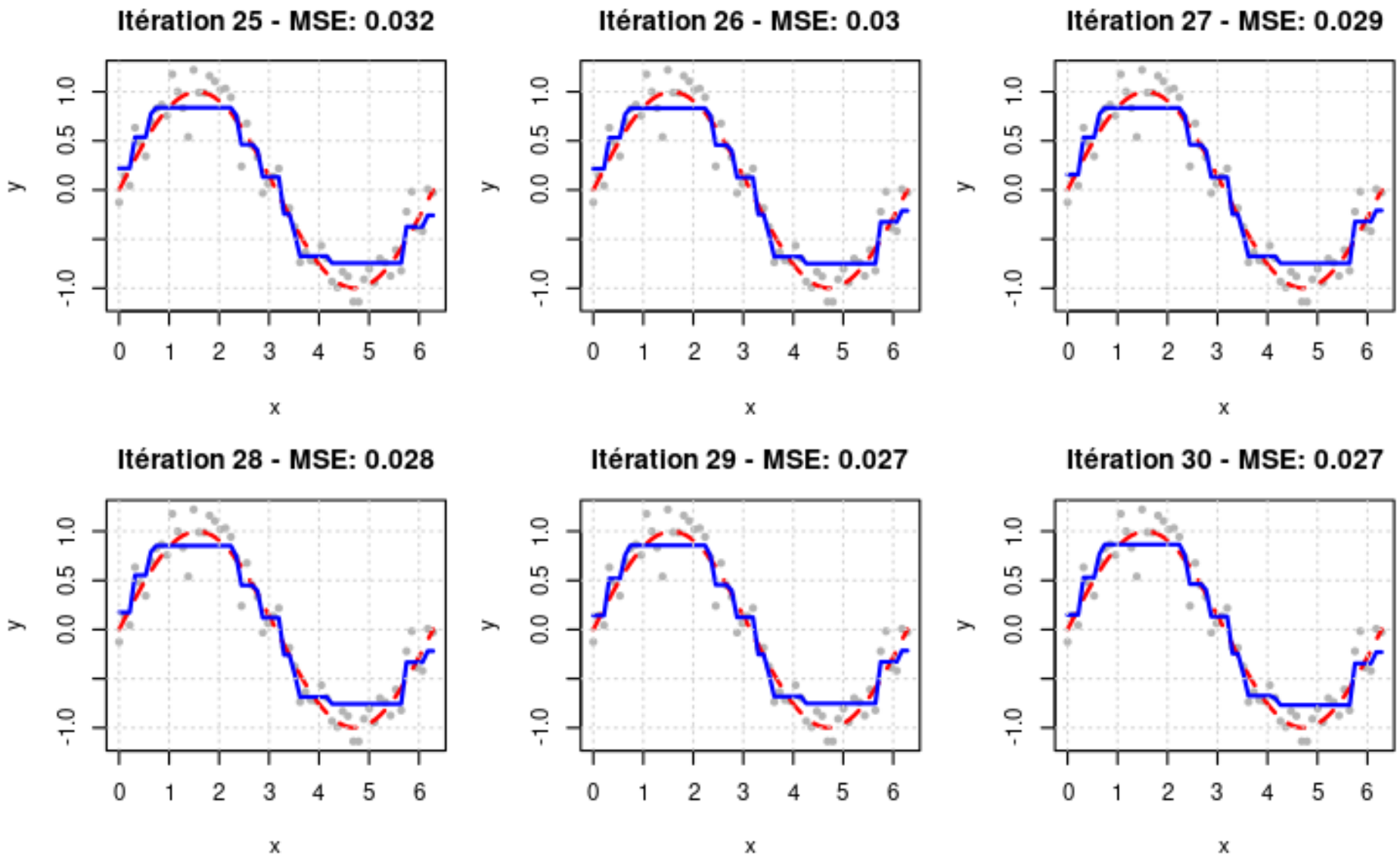
C) Mise à jour du modèle :

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

Gradient Boosting : itérations 1 à 6



Gradient Boosting : itérations 25 à 30

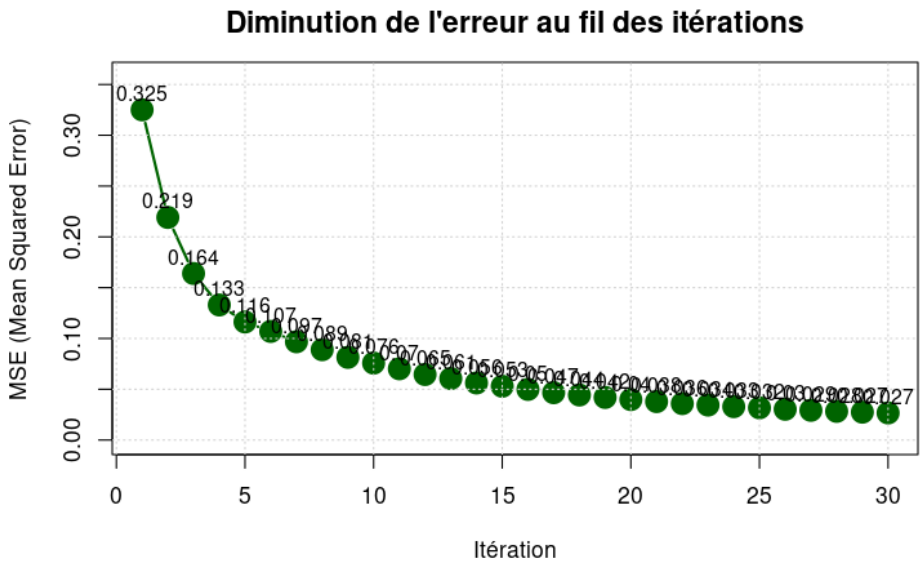


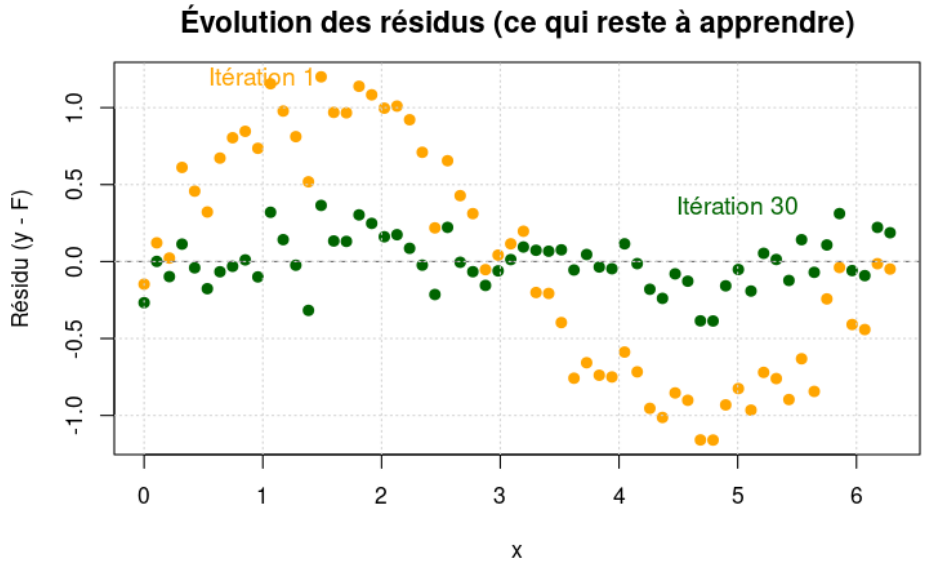
Gradient Boosting : Étape finale

Étape Finale : Après M itérations, notre modèle final est :

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \nu \cdot h_m(x)$$

C'est la somme pondérée de tous les modèles entraînés à chaque étape.





Extension à d'autres pertes

Régression :

- **MSE** : classique, minimise l'erreur quadratique, sensible aux outliers

$$L(y, F) = \frac{1}{2}(y - F)^2$$

- **Quantile loss** : prédit un quantile, robuste aux valeurs aberrantes

$$L(y, F) = (y - F) \cdot (\alpha - \mathbf{1}_{\{y<F\}})$$

où $\alpha =$ quantile à prédire

- **Huber loss** : régression robuste, mix quadratique/linéaire

$$L(y, F) = \begin{cases} \frac{1}{2}(y - F)^2 & |y - F| \leq \delta \\ \delta|y - F| - \frac{1}{2}\delta^2 & \text{sinon} \end{cases}$$

- **Log-loss** : prédit 0/1, probabilités calibrées

$$L(y, F) = -\left[y \log(p) + (1 - y) \log(1 - p)\right], \quad p = \frac{1}{1 + e^{-F}}$$

Le choix des paramètres

Learning rate, nombre d'arbres, ...

Le choix des paramètres dans le Gradient Boosting

Pourquoi le choix des hyperparamètres est crucial ?

- **Éviter le sous-apprentissage :**
  - Modèle trop simple ne capture pas la structure des données
  - Biais élevé mauvaise approximation
- **Éviter le surapprentissage :**
  - Modèle trop complexe s'adapte au bruit
  - Variance élevée mauvaise généralisation
- **Objectif :** trouver le juste milieu entre complexité et robustesse
- **Règle pratique :** ajuster le nombre d'arbres, la profondeur et le learning rate pour contrôler ce compromis

Paramètres principaux : nombre d'arbres et learning rate

Nombre d'arbres (M) :

- Plus d'arbres meilleure approximation
- Trop d'arbres risque d'overfitting
- Toujours à relier avec le learning rate
- On arrête d'ajouter des arbres dès que l'erreur sur validation stagne ou remonte

Learning rate (ν) :

- Petit ν apprentissage lent mais plus stable
- Grand ν rapide mais risque de surapprentissage
- Relation : petit ν grand nombre d'arbres
- Bonnes pratiques : $\nu = 0.05$ 0.1 souvent un bon départ

Paramètres principaux : profondeur et subsample

Profondeur des arbres :

- Profondeur = complexité des interactions apprises
- 35 = souvent suffisant
- Trop profond variance élevée overfitting

Subsample :

- Sous-échantillonnage des données à chaque itération
- Introduit de la stochasticité régularisation naturelle
- En général : 0.5 0.9
- Cela correspond au stochastic gradient boosting

Méthodes de réglage des hyperparamètres

Techniques :

- Grid search : teste toutes les combinaisons
- Random search : échantillonne aléatoirement
- Validation croisée obligatoire !

Exemple :

```
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 4, 5],
    'subsample': [0.8, 1.0]
}
= 54 modèles
```

Bonnes pratiques générales

- Les paramètres contrôlent le biais/variance du modèle
- Pas de solution universelle dépend du jeu de données
- Bon point de départ :
  - Petit learning rate
  - Arbres peu profonds
  - Subsample < 1

Conclusion

Conclusion : Domaines d'application

- **Finance** : prévision du risque de crédit, détection de fraude
- **Marketing** : prédiction du comportement client, scoring de prospects
- **Sport et santé** : modélisation de la performance, détection d'anomalies

Conclusion : Limites pratiques

- Temps de calcul élevé, surtout avec de nombreux hyperparamètres
- Risque de surapprentissage si mal réglé (trop d'arbres ou learning rate trop grand)
- Complexité du réglage : nécessite Grid Search ou Random Search
- Moins interprétable qu'un modèle linéaire ou un arbre unique
- Données non tabulaires (images, texte, son) : réseaux de neurones souvent plus performants
- Données en temps réel : nécessité de réentraînement du modèle

Conclusion : Perspective mathématique

- Gradient Boosting = descente de gradient fonctionnelle
- Chaque modèle $h_m(x)$ approxime le gradient négatif de la fonction de perte
- Modèle final :

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \nu \, h_m(x)$$

- Réduction progressive de l'erreur sur l'ensemble d'apprentissage par accumulation des corrections
- Biais/Variance contrôlables via profondeur des arbres et learning rate

TP Machine

Travaux Pratiques : Gradient Boosting

Prochain TD : Mise en pratique du Gradient Boosting sur des jeux de données simulés et réels avec le package gbm.

Ressources : Le TD est disponible sur GitHub :

https://github.com/ziraax/GradientBoostingCourse

Objectifs du TD :

- Implémenter un Gradient Boosting simple
- Visualiser l'évolution du modèle itération par itération
- Expérimenter avec différents hyperparamètres (nombre d'arbres, learning rate, profondeur)