

# Machine learning - Boosting

---

L. Rouvière

[laurent.rouviere@univ-rennes2.fr](mailto:laurent.rouviere@univ-rennes2.fr)

NOVEMBRE 2022

- **Objectifs** : comprendre les aspects **théoriques** et **pratiques** des algorithmes de gradient boosting.
- **Pré-requis** : théorie des probabilités, modélisation statistique, machine learning, méthodes par arbres. R, niveau avancé.
- **Enseignant** : Laurent Rouvière [laurent.rouviere@univ-rennes2.fr](mailto:laurent.rouviere@univ-rennes2.fr)
  - **Recherche** : statistique non paramétrique, apprentissage statistique
  - **Enseignements** : statistique et probabilités (Université, école d'ingénieur et de commerce, formation continue).
  - **Consulting** : énergie, finance, marketing, sport.

- Matériel :
  - slides : [https://lrouviere.github.io/page\\_perso/apprentissage\\_sup.html](https://lrouviere.github.io/page_perso/apprentissage_sup.html)
  - Tutoriel : [https://lrouviere.github.io/TUTO\\_ARBRES/](https://lrouviere.github.io/TUTO_ARBRES/)
- 3 parties :
  1. Rappels sur les fondamentaux du machine Learning
  2. Les algorithmes de Gradient Boosting
  3. Xgboost : Extreme Gradient Boosting.

## Rappels

## Boosting

- Algorithme de gradient boosting

- Choix des paramètres

- Compléments/conclusion

- Xgboost

## Bibliographie

## Le problème

Prédire/expliquer une sortie  $Y$  par des entrées  $X = (X_1, \dots, X_d)$

- Fonction de prévision :  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ .

# Vocabulaire

- **Fonction de prévision** :  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ .
- **Fonction de perte** :  $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+$  telle que

$$\begin{cases} \ell(y, y') = 0 & \text{si } y = y' \\ \ell(y, y') > 0 & \text{si } y \neq y'. \end{cases}$$

- **Fonction de prévision** :  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ .
- **Fonction de perte** :  $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+$  telle que

$$\begin{cases} \ell(y, y') = 0 & \text{si } y = y' \\ \ell(y, y') > 0 & \text{si } y \neq y'. \end{cases}$$

- **Risque** :  $\mathcal{R}(f) = \mathbf{E}[\ell(Y, f(X))]$ .



- **Fonction de prévision** :  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ .
- **Fonction de perte** :  $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+$  telle que

$$\begin{cases} \ell(y, y') = 0 & \text{si } y = y' \\ \ell(y, y') > 0 & \text{si } y \neq y'. \end{cases}$$

- **Risque** :  $\mathcal{R}(f) = \mathbf{E}[\ell(Y, f(X))]$ .
- **Champion** ou **fonction de prévision optimale**

$$f^* \in \underset{f}{\operatorname{argmin}} \mathcal{R}(f) \iff \mathcal{R}(f^*) \leq \mathcal{R}(f) \quad \forall f$$

# Vocabulaire

- **Fonction de prévision** :  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ .
- **Fonction de perte** :  $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+$  telle que

$$\begin{cases} \ell(y, y') = 0 & \text{si } y = y' \\ \ell(y, y') > 0 & \text{si } y \neq y'. \end{cases}$$

- **Risque** :  $\mathcal{R}(f) = \mathbf{E}[\ell(Y, f(X))]$ .
- **Champion** ou **fonction de prévision optimale**

$$f^* \in \operatorname{argmin}_f \mathcal{R}(f) \iff \mathcal{R}(f^*) \leq \mathcal{R}(f) \quad \forall f$$

## Problème

$f^*$  est toujours **inconnu**.

- Les données  $\mathcal{D}_n = \{(x_1, y_1), \dots, (x_n, y_n)\}$ .

- Les données  $\mathcal{D}_n = \{(x_1, y_1), \dots, (x_n, y_n)\}$ .

### Le problème pratique

Trouver un **algorithme de prévision**  $f_n(.) = f_n(., \mathcal{D}_n)$  tel que  $\mathcal{R}(f_n) \approx \mathcal{R}(f^*)$ .

- Les données  $\mathcal{D}_n = \{(x_1, y_1), \dots, (x_n, y_n)\}$ .

## Le problème pratique

Trouver un **algorithme de prévision**  $f_n(.) = f_n(., \mathcal{D}_n)$  tel que  $\mathcal{R}(f_n) \approx \mathcal{R}(f^*)$ .

## Exemples

- Algorithme linéaires ou logistique : MCO, ridge, lasso
- Plus proches voisins, SVM
- Arbres, forêts aléatoires, boosting...

## Conséquence

Crucial de savoir calculer/estimer le risque d'un algorithme de prévision

$$\mathcal{R}(f_n) = \mathbf{E}[\ell(Y, f_n(X))].$$

## Conséquence

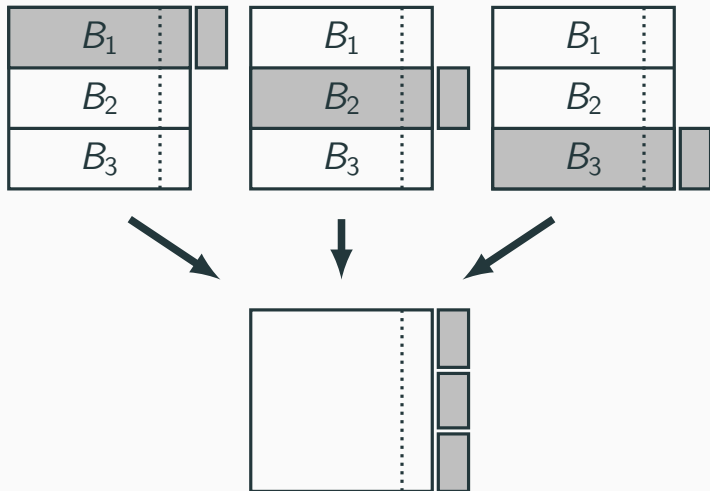
Crucial de savoir calculer/estimer le risque d'un algorithme de prévision

$$\mathcal{R}(f_n) = \mathbf{E}[\ell(Y, f_n(X))].$$

Cela s'effectue généralement à l'aide de méthodes de ré-échantillonnage :

- Validation hold-out (on coupe en deux)
- Validation croisée (on coupe en blocs)
- Bootstrap (tirages avec ou sans remise)

## Validation croisée





## Algorithme - CV

**Entrée :**  $\{B_1, \dots, B_K\}$  une partition de  $\{1, \dots, n\}$  en  $K$  blocs.

Pour  $k = 1, \dots, K$  :

1. Ajuster l'algorithme de prévision en utilisant **l'ensemble des données privé du  $k^{\text{e}}$  bloc**, c'est-à-dire  $\mathcal{B}_k = \{(x_i, y_i) : i \in \{1, \dots, n\} \setminus B_k\}$ . On désigne par  $f_k(\cdot) = f_k(\cdot, \mathcal{B}_k)$  l'algorithme obtenu.
2. Calculer la valeur prédite par l'algorithme pour chaque observation du bloc  $k$  :  $f_k(x_i), i \in B_k$  et en déduire le **risque sur le bloc  $k$**  :

$$\hat{\mathcal{R}}(f_k) = \frac{1}{|B_k|} \sum_{i \in B_k} \ell(y_i, f_k(x_i)).$$

**Retourner :**  $\frac{1}{K} \sum_{k=1}^K \hat{\mathcal{R}}(f_k).$

## Le sur-apprentissage

- La plupart des modèles statistiques renvoient des estimateurs qui dépendent de paramètres  $\lambda$  à calibrer.

# Le sur-apprentissage

- La plupart des modèles statistiques renvoient des estimateurs qui dépendent de **paramètres  $\lambda$  à calibrer**.

## Exemples

- nombres de variables dans un modèle linéaire ou logistique.
- paramètre de pénalités pour les régressions pénalisées.
- profondeur des arbres.
- nombre de plus proches voisins.
- nombre d'itérations en boosting.
- ...

# Le sur-apprentissage

- La plupart des modèles statistiques renvoient des estimateurs qui dépendent de **paramètres  $\lambda$  à calibrer**.

## Exemples

- nombres de variables dans un modèle linéaire ou logistique.
- paramètre de pénalités pour les régressions pénalisées.
- profondeur des arbres.
- nombre de plus proches voisins.
- nombre d'itérations en boosting.
- ...

## Remarque importante

Le choix de ces paramètres est le plus souvent **crucial** pour la **performance de l'estimateur sélectionné**.

- Le paramètre  $\lambda$  à sélectionner représente la complexité du modèle :

- Le paramètre  $\lambda$  à sélectionner représente la complexité du modèle :

### Complexité $\implies$ compromis biais/variance

- $\lambda$  petit  $\implies$  modèle peu flexible  $\implies$  mauvaise adéquation sur les données  $\implies$  biais  $\nearrow$ , variance  $\searrow$ .

- Le paramètre  $\lambda$  à sélectionner représente la complexité du modèle :

### Complexité $\implies$ compromis biais/variance

- $\lambda$  petit  $\implies$  modèle peu flexible  $\implies$  mauvaise adéquation sur les données  $\implies$  biais  $\nearrow$ , variance  $\searrow$ .
- $\lambda$  grand  $\implies$  modèle trop flexible  $\implies$  sur-ajustement  $\implies$  biais  $\searrow$ , variance  $\nearrow$ .

- Le paramètre  $\lambda$  à sélectionner représente la **complexité du modèle** :

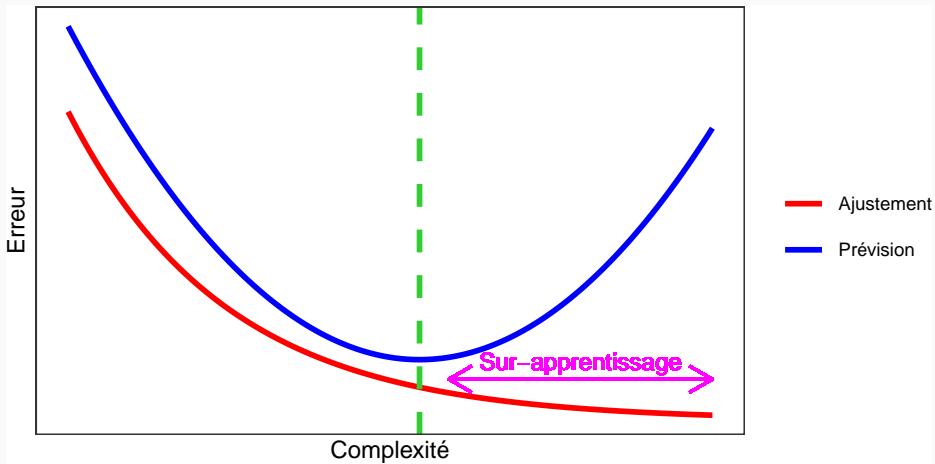
### Complexité $\implies$ compromis biais/variance

- $\lambda$  petit  $\implies$  modèle peu flexible  $\implies$  mauvaise adéquation sur les données  $\implies$  biais  $\nearrow$ , variance  $\searrow$ .
- $\lambda$  grand  $\implies$  modèle trop flexible  $\implies$  **sur-ajustement**  $\implies$  biais  $\searrow$ , variance  $\nearrow$ .

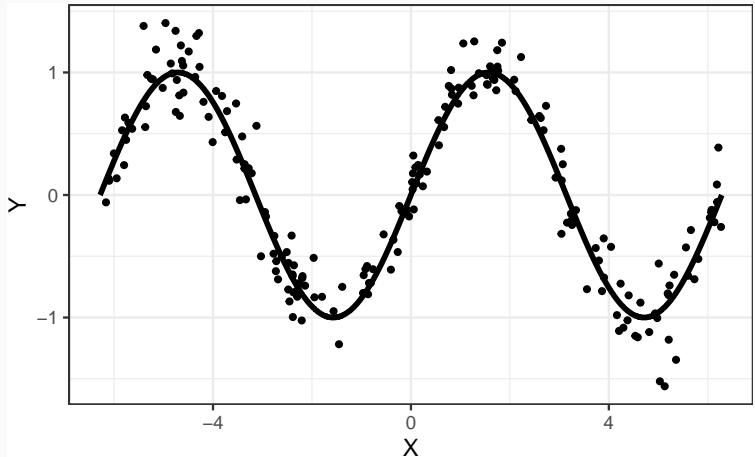
### Overfitting

**Sur-ajuster** signifie que le modèle va (trop) bien ajuster les données d'apprentissage, il aura du mal à s'adapter à de nouveaux individus.

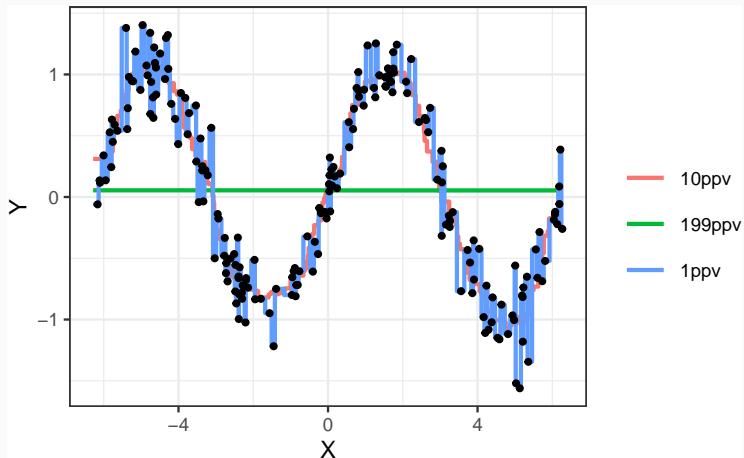




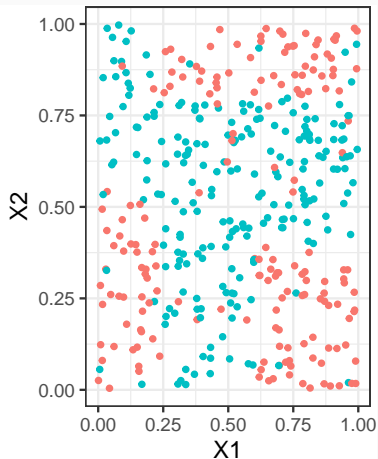
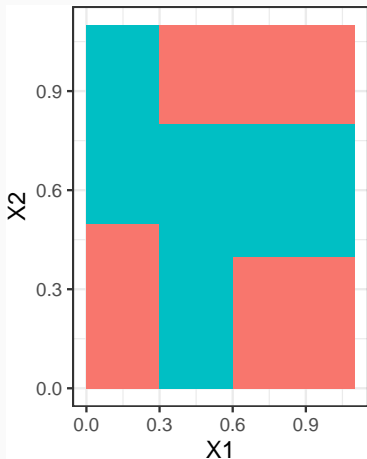
# Overfitting en régression



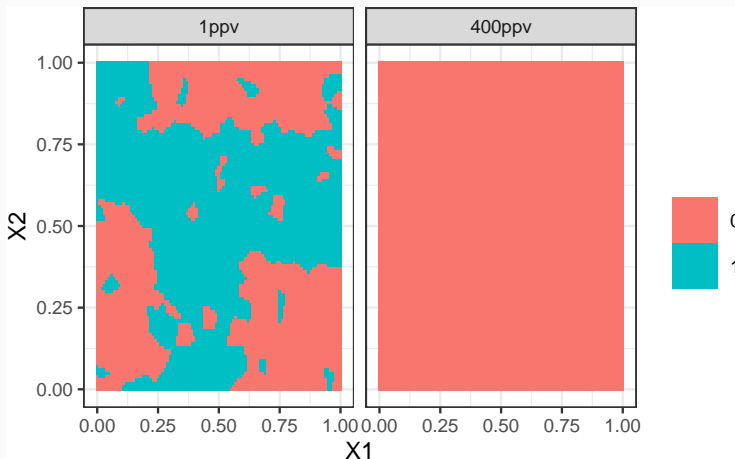
# Overfitting en régression



# Overfitting en classification supervisée



# Overfitting en classification supervisée

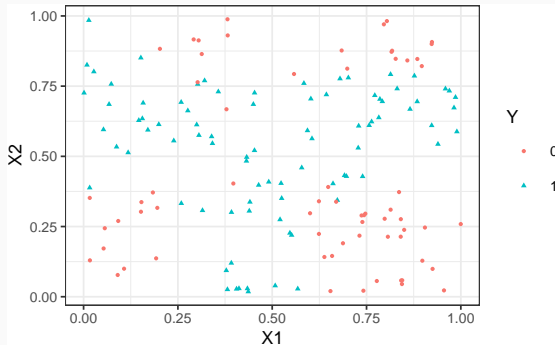


Application shiny

[https://lrouviere.shinyapps.io/overfitting\\_app/](https://lrouviere.shinyapps.io/overfitting_app/)

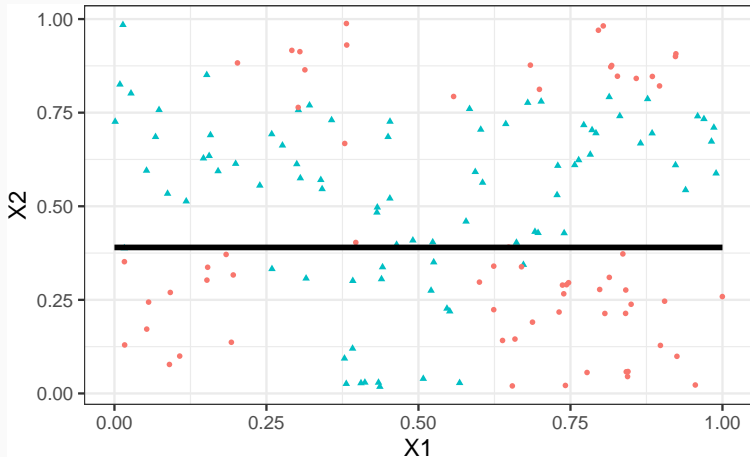
# Les arbres

- A chaque étape, la méthode cherche une **nouvelle division** : une **variable** et un **seuil** de coupure.



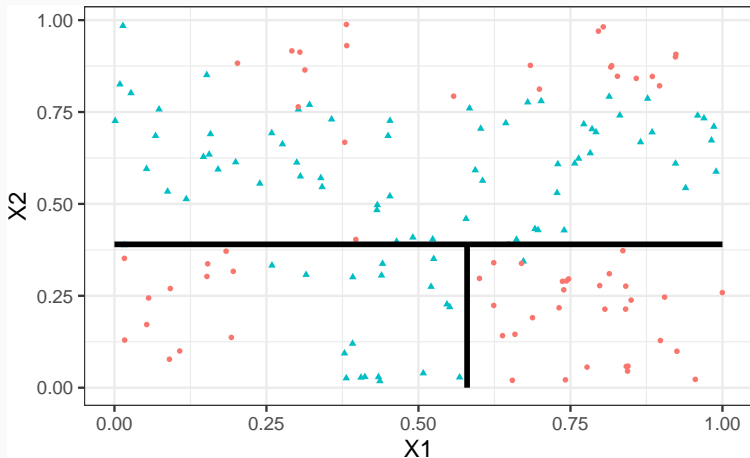
# Les arbres

- A chaque étape, la méthode cherche une **nouvelle division** : une **variable** et un **seuil** de coupure.



# Les arbres

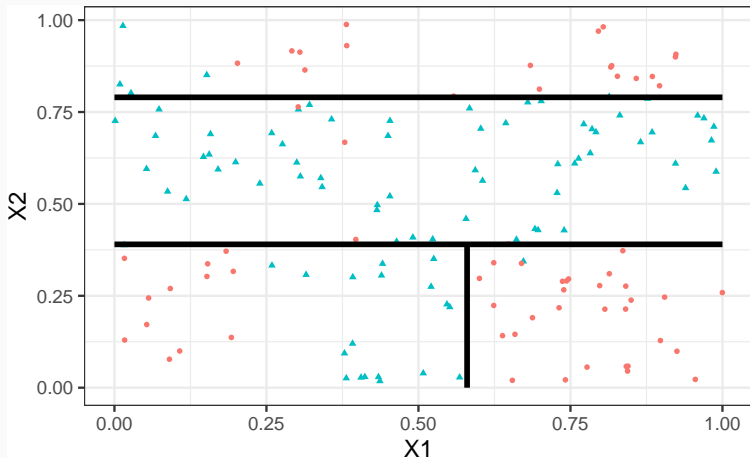
- A chaque étape, la méthode cherche une **nouvelle division** : une **variable** et un **seuil** de coupure.





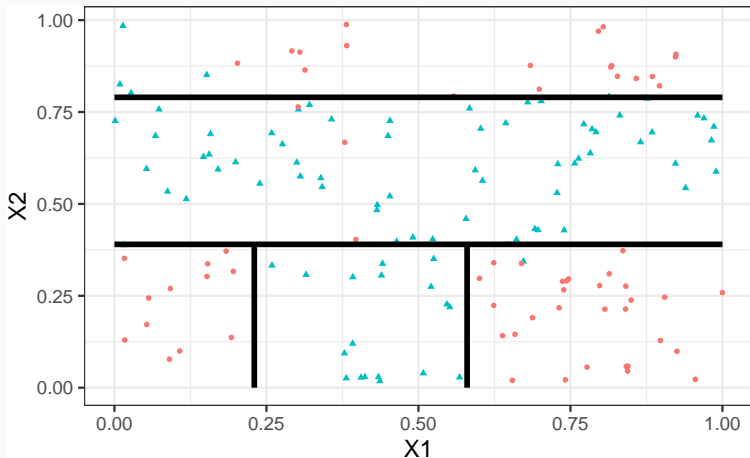
# Les arbres

- A chaque étape, la méthode cherche une **nouvelle division** : une **variable** et un **seuil** de coupure.



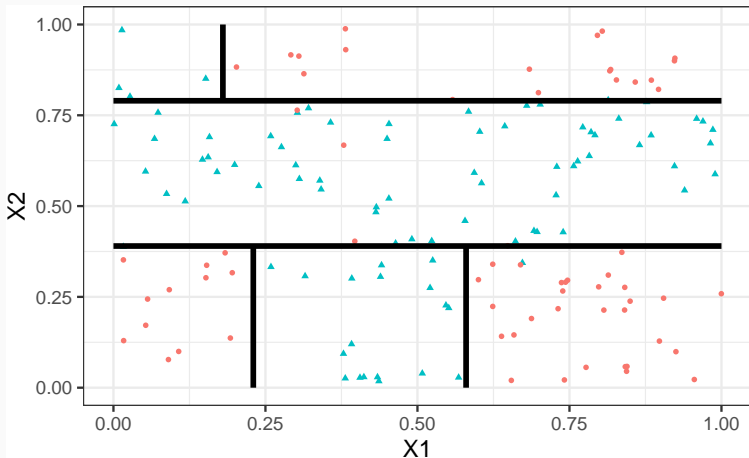
# Les arbres

- A chaque étape, la méthode cherche une **nouvelle division** : une **variable** et un **seuil** de coupure.

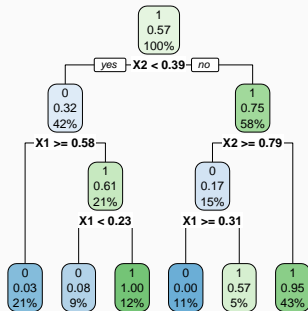
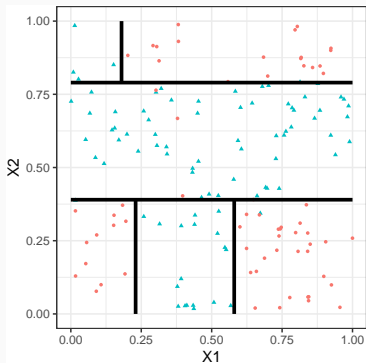


# Les arbres

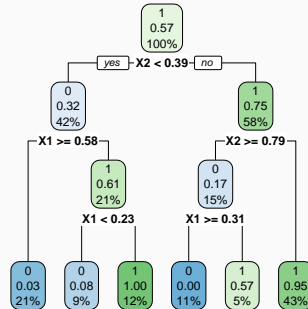
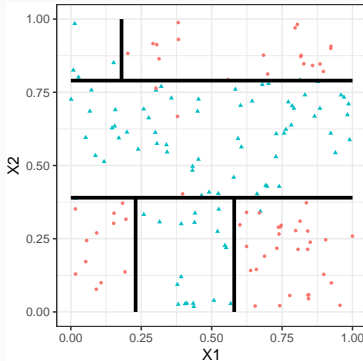
- A chaque étape, la méthode cherche une **nouvelle division** : une **variable** et un **seuil** de coupure.



# Représentation de l'arbre



# Représentation de l'arbre



## Remarque

Visuel de **droite** plus pertinent :

- Plus d'information.
- Généralisation à plus de deux dimensions.

- La complexité d'un arbre est caractérisé par sa profondeur ou son nombre de coupures :

- La **complexité** d'un arbre est caractérisé par sa **profondeur** ou son **nombre de coupures** :
  - Arbres trop profond  $\implies$  sur-ajustement, peu de biais mais trop de variance
  - Arbre peu profond  $\implies$  sous-ajustement, peu de variance mais beaucoup de biais.

- La **complexité** d'un arbre est caractérisé par sa **profondeur** ou son **nombre de coupures** :
  - Arbres trop profond  $\implies$  sur-ajustement, peu de biais mais trop de variance
  - Arbre peu profond  $\implies$  sous-ajustement, peu de variance mais beaucoup de biais.

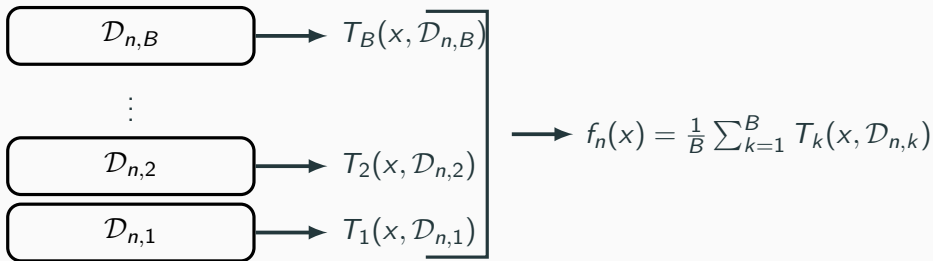
## **Solution : élagage [Breiman et al., 1984]**

1. Construire un arbre très/trop profond ;
2. Retirer les branches inutiles ou peu informatives.



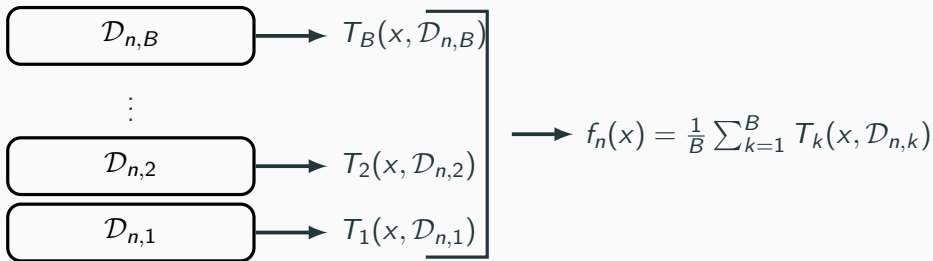
# Agrégation

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



# Agrégation

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



## Questions

1. Comment choisir les **échantillons**  $\mathcal{D}_{n,b}$  ?
2. Comment choisir les **algorithmes** ?
3. ...

- Le **bagging** désigne un ensemble de méthodes introduit par Léo Breiman [**Breiman, 1996**].
- **Bagging** : vient de la contraction de **B**ootstrap **A**ggregating.
- **Idée** : plutôt que de construire un seul estimateur, en construire un grand nombre (sur des échantillons **bootstrap**) et les **agréger**.

## Idée : échantillons bootstrap

- Echantillon **initial** :

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

## Idée : échantillons bootstrap

- Echantillon **initial** :

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

- Echantillons **bootstrap** : tirage de taille  $n$  avec remise

3	4	6	10	3	9	10	7	7	1	$T_1$
2	8	6	2	10	10	2	9	5	6	$T_2$
2	9	4	4	7	7	2	3	6	7	$T_3$
6	1	3	3	9	3	8	10	10	1	$T_4$
3	7	10	3	2	8	6	9	10	2	$T_5$
	$\vdots$								$\vdots$	
7	10	3	4	9	10	10	8	6	1	$T_B$

## Idée : échantillons bootstrap

- Echantillon **initial** :

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

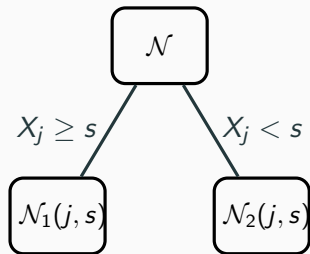
- Echantillons **bootstrap** : tirage de taille  $n$  avec remise

3	4	6	10	3	9	10	7	7	1	$T_1$
2	8	6	2	10	10	2	9	5	6	$T_2$
2	9	4	4	7	7	2	3	6	7	$T_3$
6	1	3	3	9	3	8	10	10	1	$T_4$
3	7	10	3	2	8	6	9	10	2	$T_5$
	$\vdots$								$\vdots$	
7	10	3	4	9	10	10	8	6	1	$T_B$

- A la fin, on **agrège** :

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

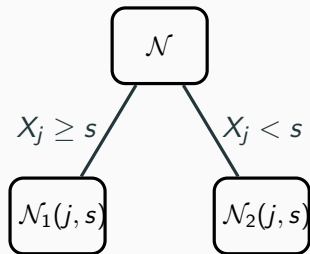
## Coupures "aléatoires"



### Arbres pour forêt

- Breiman propose de sélectionner la "meilleure" variable dans un ensemble composé **uniquement de  $m$  try variables choisies aléatoirement parmi les  $d$  variables initiales.**

## Coupures "aléatoires"



### Arbres pour forêt

- Breiman propose de sélectionner la "meilleure" variable dans un ensemble composé **uniquement de  $m$  try variables choisies aléatoirement parmi les  $d$  variables initiales.**
- **Objectif :** **diminuer la corrélation** entre les arbres que l'on agrège.



# Algorithme forêts aléatoires

Entrées :

- $B$  un entier positif ;
- **mtry** un entier entre 1 et  $d$  ;
- **min.node.size** un entier plus petit que  $n$ .

Pour  $b$  entre 1 et  $B$  :

1. Faire un tirage aléatoire avec remise de taille  $n$  dans  $\{1, \dots, n\}$ . On note  $\mathcal{I}_b$  l'ensemble des indices sélectionnés et  $\mathcal{D}_{n,b}^* = \{(x_i, y_i), i \in \mathcal{I}_b\}$  l'échantillon bootstrap associé.
2. Construire un arbre CART à partir de  $\mathcal{D}_{n,b}^*$  en découpant chaque nœud de la façon suivante :
  - 2.1 Choisir **mtry** variables au hasard parmi les  $d$  variables explicatives ;
  - 2.2 Sélectionner la meilleure coupure  $X_j \leq s$  en ne considérant que les **mtry** variables sélectionnées ;
  - 2.3 Ne pas découper un nœud s'il contient moins de **min.node.size** observations.
3. On note  $T(\cdot, \theta_b, \mathcal{D}_n)$  l'arbre obtenu.

Retourner :  $f_n(x) = \frac{1}{B} \sum_{b=1}^B T(x, \theta_b, \mathcal{D}_n)$ .

# Le coin R

- Notamment 2 packages avec à peu près la même syntaxe.
- **randomforest** : le plus ancien et probablement encore le plus utilisé.
- **ranger** [Wright and Ziegler, 2017] : plus efficace au niveau temps de calcul (codé en C++).

```
> library(ranger)
> set.seed(12345)
> foret <- ranger(type~.,data=spam)
> foret
## ranger(type ~ ., data = spam)
## Type: Classification
## Number of trees: 500
## Sample size: 4601
## Number of independent variables: 57
## Mtry: 7
## Target node size: 1
## Variable importance mode: none
## Splitrule: gini
## OOB prediction error: 4.59 %
```

- $B \implies$  le plus grand possible.

En pratique on pourra s'assurer que le **courbe d'erreur** en fonction du nombre d'arbres est **stabilisée**.

- $B \implies$  le plus grand possible.

En pratique on pourra s'assurer que le **courbe d'erreur** en fonction du nombre d'arbres est **stabilisée**.

- ***min.node.size*** petit  $\implies$  bagging = réduction de variance  $\implies$  il faut des arbres profonds. Par défaut
  - ***min.node.size*** = 5 en régression
  - ***min.node.size*** = 1 en classification

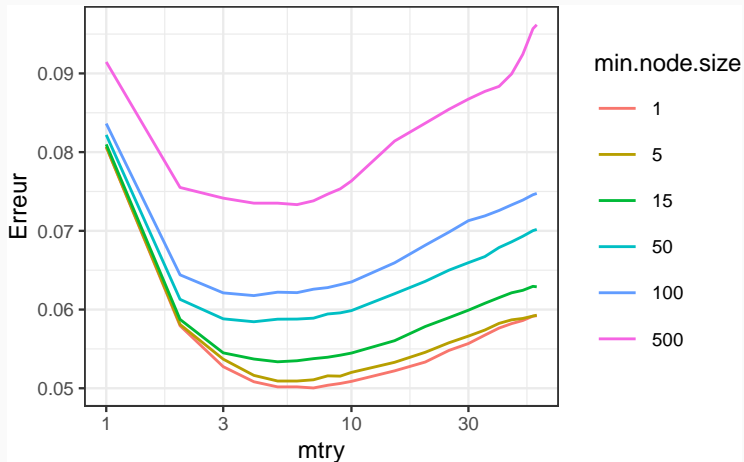
# Choix des paramètres

- $B \implies$  le plus grand possible.

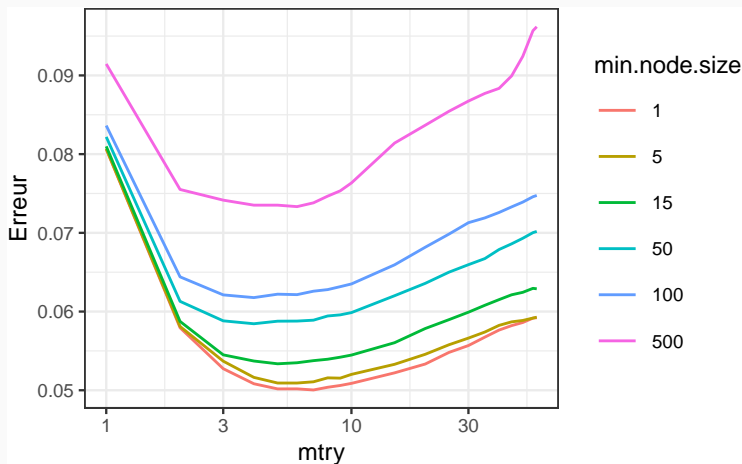
En pratique on pourra s'assurer que le **courbe d'erreur** en fonction du nombre d'arbres est **stabilisée**.

- ***min.node.size*** petit  $\implies$  bagging = réduction de variance  $\implies$  il faut des arbres profonds. Par défaut
  - ***min.node.size*** = 5 en régression
  - ***min.node.size*** = 1 en classification
- Par défaut ***mtry*** =  $d/3$  en régression et  $\sqrt{d}$  en classification mais à calibrer (estimation du risque).

- Visualisation d'erreur en fonction de `min.node.size` et `mtry`



- Visualisation d'erreur en fonction de `min.node.size` et `mtry`



## Commentaires

`min.node.size` petit et `mtry` à calibrer.

- On peut bien entendu **calibrer ces paramètres** avec les approches traditionnelles mais...



- On peut bien entendu **calibrer ces paramètres** avec les approches traditionnelles mais...
- les valeurs par défaut sont souvent performantes !

- On peut bien entendu **calibrer ces paramètres** avec les approches traditionnelles mais...
- les valeurs par défaut sont souvent performantes !
- On pourra quand même faire quelques essais, notamment pour **mtry**.

# Un exemple avec tidymodels

## 1. Initialisation du workflow :

```
> tune_spec <- rand_forest(mtry = tune(), min_n = tune()) %>%  
+   set_engine("ranger") %>%  
+   set_mode("classification")  
> rf_wf <- workflow() %>% add_model(tune_spec) %>% add_formula(type ~ .)
```

## 2. Ré-échantillonnage et grille de paramètres :

```
> blocs <- vfold_cv(spam, v = 10, repeats = 5)  
> rf_grid <- expand_grid(mtry=c(seq(1,55,by=5),57),  
+                       min_n=c(1,5,15,50,100,500))
```

### 3. Calcul des erreurs :

```
> rf_res <- rf_wf %>% tune_grid(resamples = blocs, grid = rf_grid)
```

### 4. Visualisation des résultats (AUC et accuracy) :

```
> rf_res %>% show_best("roc_auc") %>% select(-8)
```

```
## # A tibble: 5 x 7
```

	mtry	min_n	.metric	.estimator	mean	n	std_err
	<dbl>	<dbl>	<chr>	<chr>	<dbl>	<int>	<dbl>
## 1	4	1	roc_auc	binary	0.988	50	0.000614
## 2	5	1	roc_auc	binary	0.988	50	0.000623
## 3	6	1	roc_auc	binary	0.988	50	0.000617
## 4	5	5	roc_auc	binary	0.988	50	0.000621
## 5	7	1	roc_auc	binary	0.988	50	0.000645

```
> rf_res %>% show_best("accuracy") %>% select(-8)
```

```
## # A tibble: 5 x 7
```

	mtry	min_n	.metric	.estimator	mean	n	std_err
	<dbl>	<dbl>	<chr>	<chr>	<dbl>	<int>	<dbl>
## 1	4	1	accuracy	binary	0.954	50	0.00159
## 2	6	1	accuracy	binary	0.954	50	0.00141
## 3	7	1	accuracy	binary	0.954	50	0.00149
## 4	5	1	accuracy	binary	0.954	50	0.00153
## 5	8	1	accuracy	binary	0.953	50	0.00146

## Remarque

On retrouve bien `min.node.size` petit et `mtry` proche de la valeur par défaut (7).

## Remarque

On retrouve bien **min.node.size** petit et **mtry** proche de la valeur par défaut (7).

### 5. Ajustement de l'algorithme final :

```
> foret_finale <- rf_wf %>%  
+   finalize_workflow(list(mtry=7,min_n=1)) %>%  
+   fit(data=spam)
```

## Beaucoup d'avantages

- Bonnes performances prédictives  $\implies$  souvent parmi les algorithmes de tête dans les compétitions [Fernández-Delgado et al., 2014].
- Facile à calibrer.

# Conclusion

## Beaucoup d'avantages

- Bonnes performances prédictives  $\implies$  souvent parmi les algorithmes de tête dans les compétitions [Fernández-Delgado et al., 2014].
- Facile à calibrer.

## Assez peu d'inconvénients

Coté boîte noire (mais guère plus que les autres méthodes...)



Rappels

Boosting

- Algorithme de gradient boosting

- Choix des paramètres

- Compléments/conclusion

- Xgboost

Bibliographie

- Le terme **Boosting** s'applique à des méthodes générales permettant de produire des décisions précises à partir de **règles faibles** (weaklearner).

- Le terme **Boosting** s'applique à des méthodes générales permettant de produire des décisions précises à partir de **règles faibles** (weaklearner).
- Historiquement, le **premier** algorithme boosting est **adaboost** [Freund and Schapire, 1996].
- Beaucoup de travaux ont par la suite été développés pour **comprendre et généraliser** ces algorithmes (voir [Hastie et al., 2009]) :
  - modèle additif
  - **descente de gradient**  $\implies$  gradient boosting machine, extreme gradient boosting (Xgboost).
  - ...

- Le terme **Boosting** s'applique à des méthodes générales permettant de produire des décisions précises à partir de **règles faibles** (weaklearner).
- Historiquement, le **premier** algorithme boosting est **adaboost** [Freund and Schapire, 1996].
- Beaucoup de travaux ont par la suite été développés pour **comprendre et généraliser** ces algorithmes (voir [Hastie et al., 2009]) :
  - modèle additif
  - **descente de gradient**  $\implies$  gradient boosting machine, extreme gradient bossting (Xgboost).
  - ...
- Dans cette partie  $\implies$  descente de gradient.

- Machine learning  $\implies$  objectifs prédictifs  $\implies$  minimisation de risque.

- Machine learning  $\implies$  objectifs **prédictifs**  $\implies$  minimisation de risque.
- Risque d'une fonction de prévision  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  :

$$\mathcal{R}(f) = \mathbf{E}[\ell(Y, f(X))].$$

- Machine learning  $\implies$  objectifs **prédictifs**  $\implies$  minimisation de risque.
- Risque d'une fonction de prévision  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  :

$$\mathcal{R}(f) = \mathbf{E}[\ell(Y, f(X))].$$

- $\mathcal{R}(f)$  inconnu  $\implies$  version empirique

$$\mathcal{R}_n(f) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i)).$$

## Retour aux sources...

- Machine learning  $\implies$  objectifs **prédictifs**  $\implies$  minimisation de risque.
- Risque d'une fonction de prévision  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  :

$$\mathcal{R}(f) = \mathbf{E}[\ell(Y, f(X))].$$

- $\mathcal{R}(f)$  inconnu  $\implies$  version empirique

$$\mathcal{R}_n(f) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i)).$$

### Idée

Minimiser  $\mathcal{R}_n(f)$  sur une classe d'algorithmes  $\mathcal{F}$ .



## Choix de $\mathcal{F}$

- Il est bien entendu **crucial**.
- $\mathcal{F}$  riche/complexité élevée  $\implies$

## Choix de $\mathcal{F}$

- Il est bien entendu **crucial**.
- $\mathcal{F}$  riche/complexité élevée  $\implies \mathcal{R}_n(f) \searrow \implies f(x_i) \approx y_i, i = 1, \dots, n$   
 $\implies$  **sur-ajustement**.

## Choix de $\mathcal{F}$

- Il est bien entendu **crucial**.
- $\mathcal{F}$  riche/complexité élevée  $\implies \mathcal{R}_n(f) \searrow \implies f(x_i) \approx y_i, i = 1, \dots, n$   
 $\implies$  **sur-ajustement**.
- et réciproquement pour des classes  $\mathcal{F}$  simple/complexité faible.

# Choix de $\mathcal{F}$

- Il est bien entendu **crucial**.
- $\mathcal{F}$  riche/complexité élevée  $\implies \mathcal{R}_n(f) \searrow \implies f(x_i) \approx y_i, i = 1, \dots, n$   
 $\implies$  **sur-ajustement**.
- et réciproquement pour des classes  $\mathcal{F}$  simple/complexité faible.

## Combinaisons d'arbres

- [Friedman, 2001, Friedman, 2002] propose de se restreindre à des combinaisons d'arbres :

$$\mathcal{F} = \left\{ \sum_{b=1}^B \lambda_b T(x, \theta_b), \lambda_b \in \mathbb{R}, \theta_b \in \Theta \right\}$$

où  $\theta_b$  désigne les paramètres de l'arbre (impureté, profondeur)...

- **Rappel** : un arbre peut s'écrire

$$T(x, \theta_b) = \sum_{\ell=1}^L \gamma_{b\ell} \mathbf{1}_{x \in \mathcal{N}_{b\ell}}$$

où  $\mathcal{N}_{b\ell}$  désigne les feuilles et  $\gamma_{b\ell}$  les prévisions dans les feuilles.

- **Rappel** : un arbre peut s'écrire

$$T(x, \theta_b) = \sum_{\ell=1}^L \gamma_{b\ell} \mathbf{1}_{x \in \mathcal{N}_{b\ell}}$$

où  $\mathcal{N}_{b\ell}$  désigne les feuilles et  $\gamma_{b\ell}$  les prévisions dans les feuilles.

- Les paramètres  $B, \theta_b$  définissent la **complexité** de  $\mathcal{F}$ .
- Il faudra les **calibrer** à un moment mais nous les considérons **fixés** pour l'instant.

- **Rappel** : un arbre peut s'écrire

$$T(x, \theta_b) = \sum_{\ell=1}^L \gamma_{b\ell} \mathbf{1}_{x \in \mathcal{N}_{b\ell}}$$

où  $\mathcal{N}_{b\ell}$  désigne les feuilles et  $\gamma_{b\ell}$  les prévisions dans les feuilles.

- Les paramètres  $B, \theta_b$  définissent la **complexité** de  $\mathcal{F}$ .
- Il faudra les **calibrer** à un moment mais nous les considérons **fixés** pour l'instant.

## Un premier problème

Chercher  $f \in \mathcal{F}$  qui minimise  $\mathcal{R}_n(f)$ .

- **Rappel** : un arbre peut s'écrire

$$T(x, \theta_b) = \sum_{\ell=1}^L \gamma_{b\ell} \mathbf{1}_{x \in \mathcal{N}_{b\ell}}$$

où  $\mathcal{N}_{b\ell}$  désigne les feuilles et  $\gamma_{b\ell}$  les prévisions dans les feuilles.

- Les paramètres  $B, \theta_b$  définissent la **complexité** de  $\mathcal{F}$ .
- Il faudra les **calibrer** à un moment mais nous les considérons **fixés** pour l'instant.

## Un premier problème

Chercher  $f \in \mathcal{F}$  qui minimise  $\mathcal{R}_n(f)$ .

- Résolution numérique **trop difficile**.
- Nécessité de trouver un **algorithme** qui approche la solution.



Rappels

Boosting

- Algorithme de gradient boosting

- Choix des paramètres

- Compléments/conclusion

- Xgboost

Bibliographie

# Descentes de gradient

- Définissent des **suites** qui convergent vers des **extrema locaux** de fonctions  $\mathbb{R}^p \rightarrow \mathbb{R}$ .

# Descentes de gradient

- Définissent des suites qui convergent vers des extrema locaux de fonctions  $\mathbb{R}^p \rightarrow \mathbb{R}$ .
- Le risque  $\mathcal{R}_n(f)$  ne dépend que des valeurs de  $f$  aux points  $x_j$ .

# Descentes de gradient

- Définissent des **suites** qui convergent vers des **extrema locaux** de fonctions  $\mathbb{R}^p \rightarrow \mathbb{R}$ .
- Le risque  $\mathcal{R}_n(f)$  ne dépend que des valeurs de **f aux points  $x_i$** .
- En notant  $\mathbf{f} = (\mathbf{f}(x_1), \dots, \mathbf{f}(x_n)) \in \mathbb{R}^n$ , on a

$$\mathcal{R}_n(f) = \tilde{\mathcal{R}}_n(\mathbf{f}) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, \mathbf{f}(x_i))$$

avec  $\tilde{\mathcal{R}}_n : \mathbb{R}^n \rightarrow \mathbb{R}$ .

# Descentes de gradient

- Définissent des **suites** qui convergent vers des **extrema locaux** de fonctions  $\mathbb{R}^p \rightarrow \mathbb{R}$ .
- Le risque  $\mathcal{R}_n(f)$  ne dépend que des valeurs de **f aux points  $x_i$** .
- En notant  $\mathbf{f} = (\mathbf{f}(x_1), \dots, \mathbf{f}(x_n)) \in \mathbb{R}^n$ , on a

$$\mathcal{R}_n(f) = \tilde{\mathcal{R}}_n(\mathbf{f}) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, \mathbf{f}(x_i))$$

avec  $\tilde{\mathcal{R}}_n : \mathbb{R}^n \rightarrow \mathbb{R}$ .

## Nouveau problème

Minimiser  $\tilde{\mathcal{R}}_n$ .

$\implies$  en gardant en tête que **minimiser de  $\mathcal{R}_n(f)$**  n'est pas équivalent à **minimiser  $\tilde{\mathcal{R}}_n(\mathbf{f})$** .

- Descente de gradient  $\implies$  suite  $(\mathbf{f}_b)_b$  de vecteurs de  $\mathbb{R}^n$  qui convergent vers des extrema (locaux) de  $\tilde{\mathcal{R}}_n$ .

- Descente de gradient  $\implies$  suite  $(\mathbf{f}_b)_b$  de vecteurs de  $\mathbb{R}^n$  qui convergent vers des extrema (locaux) de  $\widetilde{\mathcal{R}}_n$ .
- Suite réursive :

$$\mathbf{f}_b = \mathbf{f}_{b-1} - \rho_b \nabla \widetilde{\mathcal{R}}_n(\mathbf{f}_{b-1}),$$

où  $\nabla \widetilde{\mathcal{R}}_n(\mathbf{f}_{b-1})$  désigne le vecteur gradient de  $\widetilde{\mathcal{R}}_n$  évalué en  $\mathbf{f}_{b-1}$ .

$\implies$  vecteur de  $\mathbb{R}^n$  donc la  $i^{\text{e}}$  coordonnée vaut

$$\frac{\partial \widetilde{\mathcal{R}}_n(\mathbf{f})}{\partial \mathbf{f}(x_i)}(\mathbf{f}_{b-1}) = \frac{\partial \ell(y_i, \mathbf{f}(x_i))}{\partial \mathbf{f}(x_i)}(\mathbf{f}_{b-1}(x_i)).$$

- Descente de gradient  $\implies$  suite  $(\mathbf{f}_b)_b$  de vecteurs de  $\mathbb{R}^n$  qui convergent vers des extrema (locaux) de  $\widetilde{\mathcal{R}}_n$ .
- Suite réursive :

$$\mathbf{f}_b = \mathbf{f}_{b-1} - \rho_b \nabla \widetilde{\mathcal{R}}_n(\mathbf{f}_{b-1}),$$

où  $\nabla \widetilde{\mathcal{R}}_n(\mathbf{f}_{b-1})$  désigne le vecteur gradient de  $\widetilde{\mathcal{R}}_n$  évalué en  $\mathbf{f}_{b-1}$ .

$\implies$  vecteur de  $\mathbb{R}^n$  donc la  $i^{\text{e}}$  coordonnée vaut

$$\frac{\partial \widetilde{\mathcal{R}}_n(\mathbf{f})}{\partial \mathbf{f}(x_i)}(\mathbf{f}_{b-1}) = \frac{\partial \ell(y_i, \mathbf{f}(x_i))}{\partial \mathbf{f}(x_i)}(\mathbf{f}_{b-1}(x_i)).$$

## Exemple

Si  $\ell(y, f(x)) = 1/2(y - f(x))^2$  alors

$$-\frac{\partial \ell(y_i, \mathbf{f}(x_i))}{\partial \mathbf{f}(x_i)}(\mathbf{f}_{b-1}(x_i)) = y_i - \mathbf{f}_{b-1}(x_i),$$

$\implies$  résidu de  $\mathbf{f}_{b-1}(x_i)$ .



- Si tout se passe bien... la suite  $(\mathbf{f}_b)_b$  doit **converger** vers un minimum de  $\widetilde{\mathcal{R}}_n$ .

## Deux problèmes

- Si tout se passe bien... la suite  $(\mathbf{f}_b)_b$  doit **converger** vers un minimum de  $\widetilde{\mathcal{R}}_n$ .

## Deux problèmes

1. Cette suite définit des prévisions uniquement aux points  $x_i \implies$   
impossible de prédire en tout  $x$ .

- Si tout se passe bien... la suite  $(\mathbf{f}_b)_b$  doit **converger** vers un minimum de  $\widetilde{\mathcal{R}}_n$ .

## Deux problèmes

1. Cette suite définit des prévisions uniquement aux points  $x_i \implies$  impossible de prédire en tout  $x$ .
2. Les éléments de la suite ne s'écrivent pas comme des combinaisons d'arbres.

- Si tout se passe bien... la suite  $(\mathbf{f}_b)_b$  doit **converger** vers un minimum de  $\widetilde{\mathcal{R}}_n$ .

## Deux problèmes

1. Cette suite définit des prévisions uniquement aux points  $x_i \implies$  impossible de prédire en tout  $x$ .
2. Les éléments de la suite ne s'écrivent **pas** comme des **combinaisons d'arbres**.

## Une solution

[Friedman, 2001] propose d'**ajuster un arbre sur les valeurs du gradient** à chaque étape de la descente.

## Algorithme de gradient boosting

1. Initialisation :  $f_0(.) = \operatorname{argmin}_c \frac{1}{n} \sum_{i=1}^n \ell(y_i, c)$

## Algorithme de gradient boosting

1. Initialisation :  $f_0(.) = \operatorname{argmin}_c \frac{1}{n} \sum_{i=1}^n \ell(y_i, c)$
2. Pour  $b = 1$  à  $B$  :
  - 2.1 Calculer l'opposé du gradient  $-\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i))$  et l'évaluer aux points  $f_{b-1}(x_i)$  :

$$u_i = -\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i)) \Big|_{f(x_i)=f_{b-1}(x_i)}, \quad i = 1, \dots, n.$$

## Algorithme de gradient boosting

1. Initialisation :  $f_0(.) = \operatorname{argmin}_c \frac{1}{n} \sum_{i=1}^n \ell(y_i, c)$
2. Pour  $b = 1$  à  $B$  :
  - 2.1 Calculer l'opposé du gradient  $-\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i))$  et l'évaluer aux points  $f_{b-1}(x_i)$  :

$$u_i = -\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i)) \Big|_{f(x_i)=f_{b-1}(x_i)}, \quad i = 1, \dots, n.$$

- 2.2 Ajuster un arbre de régression à  $J$  feuilles sur  $(x_i, u_i), \dots, (x_n, u_n)$ .

## Algorithme de gradient boosting

1. Initialisation :  $f_0(.) = \operatorname{argmin}_c \frac{1}{n} \sum_{i=1}^n \ell(y_i, c)$
2. Pour  $b = 1$  à  $B$  :
  - 2.1 Calculer l'opposé du gradient  $-\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i))$  et l'évaluer aux points  $f_{b-1}(x_i)$  :

$$u_i = -\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i)) \Big|_{f(x_i)=f_{b-1}(x_i)}, \quad i = 1, \dots, n.$$

- 2.2 Ajuster un arbre de régression à  $J$  feuilles sur  $(x_i, u_i), \dots, (x_n, u_n)$ .
  - 2.3 Calculer les valeurs prédites dans chaque feuille

$$\gamma_{jb} = \operatorname{argmin}_{\gamma} \sum_{i: x_i \in \mathcal{N}_{jb}}^n \ell(y_i, f_{b-1}(x_i) + \gamma).$$



# Algorithme de gradient boosting

1. Initialisation :  $f_0(.) = \operatorname{argmin}_c \frac{1}{n} \sum_{i=1}^n \ell(y_i, c)$
2. Pour  $b = 1$  à  $B$  :
  - 2.1 Calculer l'opposé du gradient  $-\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i))$  et l'évaluer aux points  $f_{b-1}(x_i)$  :

$$u_i = -\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i)) \Big|_{f(x_i)=f_{b-1}(x_i)}, \quad i = 1, \dots, n.$$

- 2.2 Ajuster un arbre de régression à  $J$  feuilles sur  $(x_i, u_i), \dots, (x_n, u_n)$ .
  - 2.3 Calculer les valeurs prédites dans chaque feuille

$$\gamma_{jb} = \operatorname{argmin}_{\gamma} \sum_{i: x_i \in \mathcal{N}_{jb}}^n \ell(y_i, f_{b-1}(x_i) + \gamma).$$

- 2.4 Mise à jour :  $f_b(x) = f_{b-1}(x) + \sum_{j=1}^J \gamma_{jb} \mathbf{1}_{x \in \mathcal{N}_{jb}}$ .

## Algorithme de gradient boosting

1. Initialisation :  $f_0(.) = \operatorname{argmin}_c \frac{1}{n} \sum_{i=1}^n \ell(y_i, c)$
2. Pour  $b = 1$  à  $B$  :
  - 2.1 Calculer l'opposé du gradient  $-\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i))$  et l'évaluer aux points  $f_{b-1}(x_i)$  :

$$u_i = -\frac{\partial}{\partial f(x_i)} \ell(y_i, f(x_i)) \Big|_{f(x_i)=f_{b-1}(x_i)}, \quad i = 1, \dots, n.$$

- 2.2 Ajuster un arbre de régression à  $J$  feuilles sur  $(x_i, u_i), \dots, (x_n, u_n)$ .
  - 2.3 Calculer les valeurs prédites dans chaque feuille

$$\gamma_{jb} = \operatorname{argmin}_{\gamma} \sum_{i: x_i \in \mathcal{N}_{jb}}^n \ell(y_i, f_{b-1}(x_i) + \gamma).$$

- 2.4 Mise à jour :  $f_b(x) = f_{b-1}(x) + \sum_{j=1}^J \gamma_{jb} \mathbf{1}_{x \in \mathcal{N}_{jb}}$ .

**Retourner** : l'algorithme  $f_n(x) = f_B(x)$ .

# Paramètres

Nous donnons les correspondances entre les paramètres et les options de la fonction `gbm` :

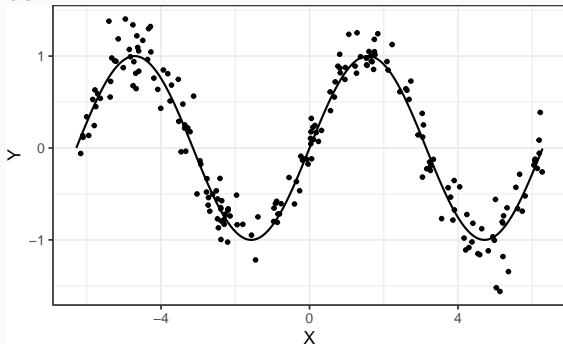
- $\ell$  la fonction de perte  $\implies$  `distribution`
- $B$  nombre d'itérations  $\implies$  `n.tree`
- $J$  le nombre de feuilles des arbres  $\implies$  `interaction.dept` ( $=J - 1$ )
- $\lambda$  le paramètre de rétrécissement  $\implies$  `shrinkage`.

## Stochastic gradient boosting

[Friedman, 2002] montre qu'`ajuster les arbres sur des sous-échantillons` (tirage sans remise) améliore souvent les performances de l'algorithme.  
 $\implies$  `bag.fraction` : taille des sous-échantillons.

# Exemple

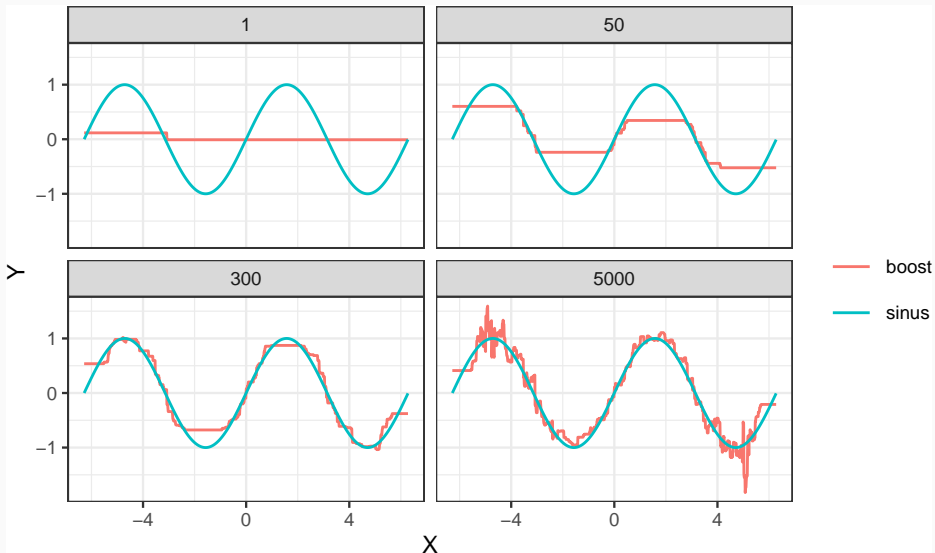
- Données **sinus**



- On entraîne l'algorithme :

```
> set.seed(1234)
> library(gbm)
> boost.5000 <- gbm(Y~.,data=data.sinus,
+                   distribution="gaussian",shrinkage=0.1,n.trees = 5000)
```

- On visualise les prévisions en fonction du nombre d'itérations :



Rappels

Boosting

Algorithme de gradient boosting

Choix des paramètres

Compléments/conclusion

Xgboost

Bibliographie

# Fonction de perte

- Pas vraiment un paramètre...
- Elle doit
  1. mesurer un coût (comme d'habitude).  
 $\implies$  elle caractérise la fonction de prévision à estimer  $\implies f_n$  est en effet un estimateur de

$$f^* \in \operatorname{argmin}_{f: \mathbb{R}^d \rightarrow \mathbb{R}} \mathbf{E}[\ell(Y, f(X))].$$

# Fonction de perte

- Pas vraiment un paramètre...
- Elle doit
  1. mesurer un **coût** (comme d'habitude).  
 $\implies$  elle caractérise la fonction de prévision à estimer  $\implies f_n$  est en effet un **estimateur** de

$$f^* \in \operatorname{argmin}_{f: \mathbb{R}^d \rightarrow \mathbb{R}} \mathbf{E}[\ell(Y, f(X))].$$

2. être **convexe** et **dérivable** par rapport à son second argument (spécificité gradient).



## $L_2$ -boosting en régression

- Correspond à la **perte quadratique**

$$\ell(y, f(x)) = \frac{1}{2}(y - f(x))^2.$$

## $L_2$ -boosting en régression

- Correspond à la **perte quadratique**

$$\ell(y, f(x)) = \frac{1}{2}(y - f(x))^2.$$

- **fonction de prévision optimale** :  $f^*(x) = \mathbf{E}[Y|X = x]$ .

## $L_2$ -boosting en régression

- Correspond à la **perte quadratique**

$$\ell(y, f(x)) = \frac{1}{2}(y - f(x))^2.$$

- fonction de prévision optimale** :  $f^*(x) = \mathbf{E}[Y|X = x]$ .

### Remarque

- Avec cette perte, les  $u_i$  sont donnés par

$$u_i = -\frac{\partial \ell(y_i, f(x_i))}{\partial f(x_i)}(f_{b-1}(x_i)) = y_i - f_{b-1}(x_i),$$

## $L_2$ -boosting en régression

- Correspond à la **perte quadratique**

$$\ell(y, f(x)) = \frac{1}{2}(y - f(x))^2.$$

- fonction de prévision optimale** :  $f^*(x) = \mathbf{E}[Y|X = x]$ .

### Remarque

- Avec cette perte, les  $u_i$  sont donnés par

$$u_i = -\frac{\partial \ell(y_i, f(x_i))}{\partial f(x_i)}(f_{b-1}(x_i)) = y_i - f_{b-1}(x_i),$$

- $f_b$  s'obtient donc en **corrigeant**  $f_{b-1}$  avec une **régression sur ses résidus**.

## Version simplifiée du $L_2$ -boosting

La boucle de l'algorithme de gradient boosting peut se réécrire :

1. Calculer les résidus  $u_i = y_i - f_{b-1}(x_i), i = 1, \dots, n$  ;
2. Ajuster un arbre de régression pour expliquer les résidus  $u_i$  par les  $x_i$  ;
3. Corriger  $f_{b-1}$  en lui ajoutant l'arbre construit.

## Version simplifiée du $L_2$ -boosting

La boucle de l'algorithme de gradient boosting peut se réécrire :

1. Calculer les résidus  $u_i = y_i - f_{b-1}(x_i), i = 1, \dots, n$  ;
2. Ajuster un arbre de régression pour expliquer les résidus  $u_i$  par les  $x_i$  ;
3. Corriger  $f_{b-1}$  en lui ajoutant l'arbre construit.

## Interprétation

- On "corrige"  $f_{b-1}$  en cherchant à expliquer "l'information restante" qui est contenue dans les résidus.
- Meilleur ajustement lorsque  $b \nearrow \implies$  biais  $\searrow$  (mais variance  $\nearrow$ ).

- Classification binaire avec  $Y$  dans  $\{-1, 1\}$  et  $\tilde{Y} = (Y + 1)/2$  dans  $\{0, 1\}$ .

- **Classification binaire** avec  $Y$  dans  $\{-1, 1\}$  et  $\tilde{Y} = (Y + 1)/2$  dans  $\{0, 1\}$ .
- **Log-vraisemblance binomiale** de la prévision  $p(x) \in [0, 1]$  par rapport à l'observation  $\tilde{y}$  :

$$\mathcal{L}(\tilde{y}, p(x)) = \tilde{y} \log p(x) + (1 - \tilde{y}) \log(1 - p(x)).$$



- **Classification binaire** avec  $Y$  dans  $\{-1, 1\}$  et  $\tilde{Y} = (Y + 1)/2$  dans  $\{0, 1\}$ .
- **Log-vraisemblance binomiale** de la prévision  $p(x) \in [0, 1]$  par rapport à l'observation  $\tilde{y}$  :

$$\mathcal{L}(\tilde{y}, p(x)) = \tilde{y} \log p(x) + (1 - \tilde{y}) \log(1 - p(x)).$$

- Soit  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  telle que

$$f(x) = \frac{1}{2} \log \frac{p(x)}{1 - p(x)} \iff p(x) = \frac{1}{1 + \exp(-2f(x))}.$$

$\implies$  re-paramétrisation.

- Chercher  $p(x)$  qui maximise  $\mathcal{L}(\tilde{y}, p(x))$  revient à chercher  $f(x)$  qui minimise son opposé :

$$\begin{aligned} -\mathcal{L}(y, f(x)) &= -\frac{y+1}{2} \log p(x) - \left(1 - \frac{y+1}{2}\right) \log(1 - p(x)) \\ &= \frac{y+1}{2} \log(1 + \exp(-2f(x))) + \\ &\quad \left(1 - \frac{y+1}{2}\right) \log(1 + \exp(2f(x))) \\ &= \log(1 + \exp(-2yf(x))). \end{aligned}$$

- Chercher  $p(x)$  qui maximise  $\mathcal{L}(\tilde{y}, p(x))$  revient à chercher  $f(x)$  qui minimise son opposé :

$$\begin{aligned} -\mathcal{L}(y, f(x)) &= -\frac{y+1}{2} \log p(x) - \left(1 - \frac{y+1}{2}\right) \log(1 - p(x)) \\ &= \frac{y+1}{2} \log(1 + \exp(-2f(x))) + \\ &\quad \left(1 - \frac{y+1}{2}\right) \log(1 + \exp(2f(x))) \\ &= \log(1 + \exp(-2yf(x))). \end{aligned}$$

### Remarque

$f(x) \mapsto \log(1 + \exp(-2yf(x)))$  est **convexe** et **dérivable**.

## Logitboost

Algorithme de gradient boosting avec la fonction de perte

$$\ell(y, f(x)) = \log(1 + \exp(-2yf(x))).$$

## Logitboost

Algorithme de gradient boosting avec la fonction de perte

$$\ell(y, f(x)) = \log(1 + \exp(-2yf(x))).$$

- Fonction optimale

$$f^*(x) = \frac{1}{2} \log \frac{\mathbf{P}(Y = 1|X = x)}{1 - \mathbf{P}(Y = 1|X = x)}.$$

## Logitboost

Algorithme de gradient boosting avec la fonction de perte

$$\ell(y, f(x)) = \log(1 + \exp(-2yf(x))).$$

- Fonction optimale

$$f^*(x) = \frac{1}{2} \log \frac{\mathbf{P}(Y = 1|X = x)}{1 - \mathbf{P}(Y = 1|X = x)}.$$

- $f_n$  estimant  $f^*$ , on estime  $\mathbf{P}(Y = 1|X = x)$  avec

$$\frac{1}{1 + \exp(-2f_n(x))}.$$

- Remarque :  $f(x) \mapsto \exp(-yf(x))$  est aussi convexe et dérivable.

## Adaboost

Algorithme de gradient boosting avec la fonction de perte

$$\ell(y, f(x)) = \exp(-yf(x)).$$

# Adaboost

- Remarque :  $f(x) \mapsto \exp(-yf(x))$  est aussi convexe et dérivable.

## Adaboost

Algorithme de gradient boosting avec la fonction de perte

$$\ell(y, f(x)) = \exp(-yf(x)).$$

## Remarque

- Même nom que l'algorithme initial de [Freund and Schapire, 1996] car



# Adaboost

- Remarque :  $f(x) \mapsto \exp(-yf(x))$  est aussi convexe et dérivable.

## Adaboost

Algorithme de gradient boosting avec la fonction de perte

$$\ell(y, f(x)) = \exp(-yf(x)).$$

## Remarque

- Même nom que l'algorithme initial de [Freund and Schapire, 1996] car quasi-similaire [Hastie et al., 2009].
- Même  $f^*$  que logitboost.

# Adaboost - version 1

## Algorithme [Freund and Schapire, 1996]

**Entrées** : une règle faible,  $M$  nombre d'itérations.

1. Initialiser les poids  $w_i = 1/n$ ,  $i = 1, \dots, n$
2. **Pour**  $m = 1$  à  $M$  :
  - a) Ajuster la règle faible sur l'échantillon  $d_n$  pondéré par les poids  $w_1, \dots, w_n$ , on note  $g_m(x)$  l'estimateur issu de cet ajustement
  - b) Calculer le taux d'erreur :
$$e_m = \frac{\sum_{i=1}^n w_i \mathbf{1}_{y_i \neq g_m(x_i)}}{\sum_{i=1}^n w_i}.$$
  - c) Calculer :  $\alpha_m = \log((1 - e_m)/e_m)$
  - d) Réajuster les poids :  $w_i = w_i \exp(\alpha_m \mathbf{1}_{y_i \neq g_m(x_i)})$ ,  $i = 1, \dots, n$

**Sorties** : l'algorithme de prévision  $\sum_{m=1}^M \alpha_m g_m(x)$ .

# Récapitulatif

- Les principales fonctions de perte pour la régression et classification sont résumées dans le tableau :

	$Y$	Perte	Prév. optimale
$L_2$ -boosting	$\mathbb{R}$	$(y - f(x))^2$	$\mathbf{E}[Y X = x]$
Logitboost	$\{-1, 1\}$	$\log(1 + \exp(-2yf(x)))$	$\frac{1}{2} \log \frac{\mathbf{P}(Y=1 X=x)}{1 - \mathbf{P}(Y=1 X=x)}$
Adaboost	$\{-1, 1\}$	$\exp(-yf(x))$	$\frac{1}{2} \log \frac{\mathbf{P}(Y=1 X=x)}{1 - \mathbf{P}(Y=1 X=x)}$

- Dans gbm on utilise distribution=
  - gaussian pour le  $L_2$ -boosting.
  - bernoulli pour logitboost.
  - adaboost pour adaboost.

- **interaction.depth** qui correspond au **nombre de coupures**  $\implies$  nombre de feuilles  $J - 1$ .
- On parle d'**interaction** car ce paramètre est associé au **degrés d'interactions** que l'algorithme peut identifier :

$$f^*(x) = \sum_{1 \leq j \leq d} f_j(x_j) + \sum_{1 \leq j, k \leq d} f_{j,k}(x_j, x_k) + \sum_{1 \leq j, k, \ell \leq d} f_{j,k,\ell}(x_j, x_k, x_\ell) + \dots$$

$\implies$  **interaction.depth** =

- 1  $\implies$  premier terme
- 2  $\implies$  second terme (interactions d'ordre 2)
- ...

- Boosting : réduction de biais.

- Boosting : réduction de biais.
- Nécessité d'utiliser des arbres biaisés  $\implies$  peu de coupures.

- Boosting : réduction de biais.
- Nécessité d'utiliser des arbres biaisés  $\implies$  peu de coupures.

## Recommandation

Choisir `interaction.depth` entre 2 et 5.

# Nombre d'itérations

- Le **nombre d'arbres** `n.trees` mesure la **complexité** de l'algorithme.
- Plus on itère, mieux on ajuste  
⇒ si on itère trop, on **sur-ajuste**.



# Nombre d'itérations

- Le nombre d'arbres `n.trees` mesure la complexité de l'algorithme.
- Plus on itère, mieux on ajuste  
⇒ si on itère trop, on sur-ajuste.
- Nécessité de calibrer correctement ce paramètre.

Comment ?

# Nombre d'itérations

- Le **nombre d'arbres** **n.trees** mesure la **complexité** de l'algorithme.
- Plus on itère, mieux on ajuste  
⇒ si on itère trop, on **sur-ajuste**.
- Nécessité de **calibrer correctement** ce paramètre.

## Comment ?

Avec des méthodes classiques d'**estimation du risque**.

## Sélection de `n.trees` dans `gbm`

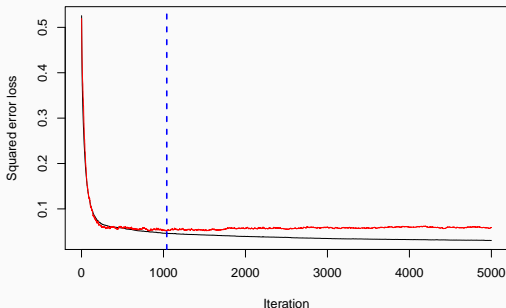
- `gbm` propose d'estimer le risque associé au paramètre `distribution` par ré-échantillonnage :
  - `bag.fraction` pour du Out Of Bag.
  - `train.fraction` pour de la validation hold out.
  - `cv.folds` pour de la validation croisée.

## Sélection de `n.trees` dans `gbm`

- `gbm` propose d'estimer le risque associé au paramètre `distribution` par ré-échantillonnage :
  - `bag.fraction` pour du Out Of Bag.
  - `train.fraction` pour de la validation hold out.
  - `cv.folds` pour de la validation croisée.
- La valeur sélectionnée s'obtient avec `gbm.perf`.

# Exemple

```
> set.seed(321)
> boost.5000 <- gbm(Y~.,data=data.sinus,train.fraction = 0.75,
+                  distribution="gaussian",shrinkage=0.1,n.trees = 5000)
> gbm.perf(boost.5000)
## [1] 1040
```



⇒ Risque quadratique estimé par hold out avec 75% d'observations dans l'échantillon d'apprentissage.

- shrinkage dans gbm.
- Correspond au pas de la descente de gradient : shrinkage  $\nearrow \Rightarrow$  minimisation plus rapide.

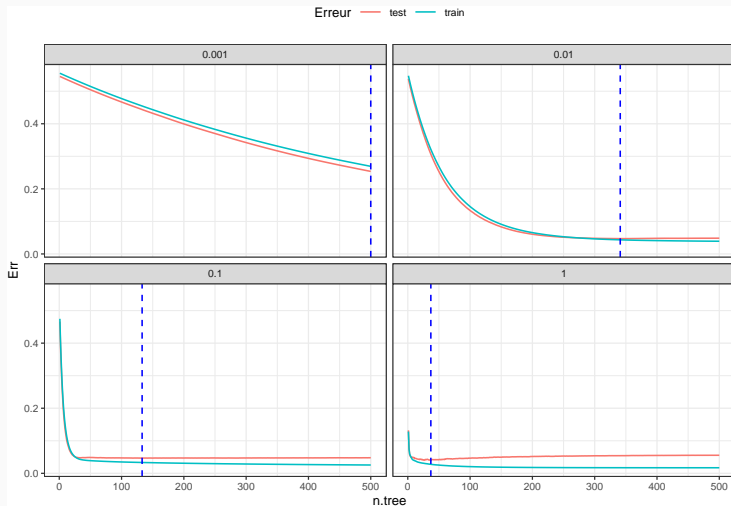
- shrinkage dans gbm.
- Correspond au pas de la descente de gradient : shrinkage  $\nearrow \implies$  minimisation plus rapide.

## Conséquence

shrinkage est lié à n.trees :

- shrinkage  $\nearrow \implies$  n.trees  $\searrow$ .
- shrinkage  $\searrow \implies$  n.trees  $\nearrow$ .

# Illustration



## Remarque

Le nombre d'itération optimal diminue lorsque **shrinkage** augmente.



- Pas nécessaire de trop optimiser **shrinkage**.
- Tester 3 ou 4 valeurs (0.01, 0.1, 0.5...) et regarder les **courbes de risque**.

- Pas nécessaire de trop optimiser **shrinkage**.
- Tester 3 ou 4 valeurs (0.01, 0.1, 0.5...) et regarder les **courbes de risque**.
- S'assurer que le **nombre d'itérations optimal** se trouve sur un **"plateau"** pour des raisons de **stabilité**.

Rappels

Boosting

Algorithme de gradient boosting

Choix des paramètres

Compléments/conclusion

Xgboost

Bibliographie

- Similaire aux forêts aléatoires.
- Score d'impureté :

$$\mathcal{I}_j^{\text{imp}} = \frac{1}{B} \sum_{b=1}^B \mathcal{I}_j(T_b).$$

- Visualisation avec vip.

# Comparaison Boosting/Forêts aléatoires

- Deux algorithmes qui agrègent des arbres :

$$f_n(x) = \sum_{b=1}^B \alpha_b T_b(x).$$

- **Indépendance** pour les forêts  $\implies T_b$  se construit indépendamment de  $T_{b-1}$ .

# Comparaison Boosting/Forêts aléatoires

- Deux algorithmes qui agrègent des arbres :

$$f_n(x) = \sum_{b=1}^B \alpha_b T_b(x).$$

- **Indépendance** pour les forêts  $\implies T_b$  se construit indépendamment de  $T_{b-1}$ .
- **Récursivité** pour le boosting  $\implies T_b$  se construit à partir de  $T_{b-1}$ .

# Comparaison Boosting/Forêts aléatoires

- Deux algorithmes qui agrègent des arbres :

$$f_n(x) = \sum_{b=1}^B \alpha_b T_b(x).$$

- **Indépendance** pour les forêts  $\implies T_b$  se construit indépendamment de  $T_{b-1}$ .
- **Récursivité** pour le boosting  $\implies T_b$  se construit à partir de  $T_{b-1}$ .

## Interprétation statistique

- **Boosting** : réduction de biais  $\implies$  arbres peu profonds.
- **Random Forest** : réduction de variance  $\implies$  arbres très profonds.

# Comparaison Boosting/Forêts aléatoires

- Deux algorithmes qui agrègent des arbres :

$$f_n(x) = \sum_{b=1}^B \alpha_b T_b(x).$$

- **Indépendance** pour les forêts  $\implies T_b$  se construit indépendamment de  $T_{b-1}$ .
- **Récursivité** pour le boosting  $\implies T_b$  se construit à partir de  $T_{b-1}$ .

## Interprétation statistique

- **Boosting** : réduction de biais  $\implies$  arbres peu profonds.
- **Random Forest** : réduction de variance  $\implies$  arbres très profonds.

$\implies$  les arbres sont ajustés de **façon différente** pour ces deux algorithmes.

$\implies$  dans les deux cas, il faut des **arbres "mauvais"**.



Rappels

Boosting

Algorithme de gradient boosting

Choix des paramètres

Compléments/conclusion

Xgboost

Bibliographie

- Pour **Extreme Gradient Boosting** [Chen and Guestrin, 2016]
- Version plus “sophistiquée” de l’algorithme de gradient boosting.

- Pour **Extreme Gradient Boosting** [Chen and Guestrin, 2016]
- Version plus “sophistiquée” de l’algorithme de gradient boosting.
- **Idée** : ajouter de la **régularisation** dans le procédé itératif d’entraînement des arbres.

## Références

- <https://xgboost.readthedocs.io/en/stable/tutorials/model.html>
- <https://arxiv.org/pdf/1603.02754.pdf>

# Le problème d'optimisation

- On cherche toujours des combinaisons d'arbres

$$f_b(x) = f_{b-1}(x) + h_b(x) \quad \text{où} \quad h_b(x) = w_{q(x)}$$

est un arbre à  $T$  feuilles :  $w \in \mathbb{R}^T$  et  $q : \mathbb{R}^d \rightarrow \{1, 2, \dots, T\}$ .

# Le problème d'optimisation

- On cherche toujours des **combinaisons d'arbres**

$$f_b(x) = f_{b-1}(x) + h_b(x) \quad \text{où} \quad h_b(x) = w_{q(x)}$$

est un arbre à  $T$  feuilles :  $w \in \mathbb{R}^T$  et  $q : \mathbb{R}^d \rightarrow \{1, 2, \dots, T\}$ .

- À l'étape  $b$ , on cherche l'arbre qui minimise la **fonction objectif** de la forme

$$\begin{aligned} \text{obj}^{(b)} &= \sum_{i=1}^n \ell(y_i, f_b(x_i)) + \sum_{j=1}^b \Omega(h_j) \\ &= \sum_{i=1}^n \ell(y_i, f_{b-1}(x_i) + h_b(x_i)) + \sum_{j=1}^b \Omega(h_j) \end{aligned}$$

où  $\Omega(h_j)$  est un terme de **régularisation** qui va pénaliser  $h_j$  en fonction de son nombre de feuilles  $T$  et des valeurs prédites  $w$ .

- Un développement limité à l'ordre 2 donne

$$\ell(y_i, f_{b-1}(x_i) + h_b(x_i)) = \ell(y_i, f_{b-1}(x_i) + h_b(x_i)) + \ell_i^{(1)} h_b(x_i) + \frac{1}{2} \ell_i^{(2)} h_b^2(x_i)$$

où

$$\ell_i^{(1)} = \frac{\partial \ell(y_i, f(x))}{\partial f(x)}(f_{b-1}(x_i)) \quad \text{et} \quad \ell_i^{(2)} = \frac{\partial^2 \ell(y_i, f(x))}{\partial f(x)^2}(f_{b-1}(x_i)).$$

- Un développement limité à l'ordre 2 donne

$$\ell(y_i, f_{b-1}(x_i) + h_b(x_i)) = \ell(y_i, f_{b-1}(x_i) + h_b(x_i)) + \ell_i^{(1)} h_b(x_i) + \frac{1}{2} \ell_i^{(2)} h_b^2(x_i)$$

où

$$\ell_i^{(1)} = \frac{\partial \ell(y_i, f(x))}{\partial f(x)}(f_{b-1}(x_i)) \quad \text{et} \quad \ell_i^{(2)} = \frac{\partial^2 \ell(y_i, f(x))}{\partial f(x)^2}(f_{b-1}(x_i)).$$

## Conséquence

La fonction objectif peut se ré-écrire

$$\text{obj}^{(b)} = \sum_{i=1}^n [\ell_i^{(1)} h_b(x_i) + \frac{1}{2} \ell_i^{(2)} h_b^2(x_i)] + \Omega(h_b) + \text{constantes.}$$

# La fonction de régularisation

- Elle doit prendre des valeurs élevées pour des arbres profonds et des valeurs ajustées élevées.



# La fonction de régularisation

- Elle doit prendre des valeurs élevées pour des arbres profonds et des valeurs ajustées élevées.
- On utilise généralement

$$\Omega(h) = \Omega(T, w) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j.$$

- Les paramètres  $\gamma$  et  $\lambda$  contrôlent le poids que l'on donne aux paramètres de l'arbre.

## Xgboost

1. Initialisation  $f_0 = h_0$ .
2. Pour  $b = 1, \dots, B$ 
  - 2.1 Ajuster un arbre  $h_b$  à  $T$  feuilles qui minimise

$$\sum_{i=1}^n [\ell_i^{(1)} h_b(x_i) + \frac{1}{2} \ell_i^{(2)} h_b^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j.$$

- 2.2 Mettre à jour

$$f_b(x) = f_{b-1}(x) + h_b(x).$$

3. Sortie : la suite d'algorithmes  $(f_b)_b$ .

## Choix des paramètres

On donne ici les principaux paramètres (il existe des variantes) et leur équivalent dans la fonction `xgboost` du package `xgboost` :

- **Fonction de perte (objective)** : idem au gradient boosting, par exemple
  - **reg :squareerror** : erreur quadratique  $\ell(y, f(x)) = (y - f(x))^2$ .
  - **reg :logistic** : vraisemblance multinomiale
  - **binary :logistic** : vraisemblance binomiale avec les probabilités en sortie

# Choix des paramètres

On donne ici les principaux paramètres (il existe des variantes) et leur équivalent dans la fonction `xgboost` du package `xgboost` :

- **Fonction de perte (objective)** : idem au gradient boosting, par exemple
  - **reg:squarederror** : erreur quadratique  $\ell(y, f(x)) = (y - f(x))^2$ .
  - **reg:logistic** : vraisemblance multinomiale
  - **binary:logistic** : vraisemblance binomiale avec les probabilités en sortie
- **Nombre d'itérations (nrounds)**.

## Choix des paramètres

On donne ici les principaux paramètres (il existe des variantes) et leur équivalent dans la fonction `xgboost` du package `xgboost` :

- **Fonction de perte (objective)** : idem au gradient boosting, par exemple
  - **reg:squarederror** : erreur quadratique  $\ell(y, f(x)) = (y - f(x))^2$ .
  - **reg:logistic** : vraisemblance multinomiale
  - **binary:logistic** : vraisemblance binomiale avec les probabilités en sortie
- **Nombre d'itérations (nrounds)**.
- **Learning rate (eta)** : idem au gradient boosting pour la mise à jour

$$f_b(x) = f_{b-1}(x) + \text{eta} h_b(x).$$

- Early stopping (`early_stopping_rounds`) : nombre d'itérations avant de stopper l'algorithme si il ne progresse pas.

- Early stopping (`early_stopping_rounds`) : nombre d'itérations avant de stopper l'algorithme si il ne progresse pas.
- Profondeur des arbres (`max_depth`)

- Early stopping (`early_stopping_rounds`) : nombre d'itérations avant de stopper l'algorithme si il ne progresse pas.
- Profondeur des arbres (`max_depth`)
- Régularisation  $L_2$  (`lambda`)
- ...



# Conclusion

- Plus général que le gradient boosting mais

# Conclusion

- Plus général que le gradient boosting mais
- plus difficile à calibrer.

# Conclusion

- Plus général que le gradient boosting mais
- plus difficile à calibrer.
- Se révèle souvent très efficace si bien calibré.

# Discussion/comparaison des algorithmes

	Linéaire	SVM	Réseau	Arbre	Forêt	Boosting
Performance	■	■	■	▼	▲	▲
Calibration	▼	▼	▼	▲	▲	▲
Coût calc.	■	▼	▼	▲	▲	▲
Interprétation	▲	▼	▼	■	▼	▼

## Commentaires

- Résultats pour **données tabulaires**.
- Différent pour **données structurées** (image, texte..)  
⇒ performance ↗ réseaux pré-entraînés ⇒ **apprentissage profond/deep learning**.

Rappels

Boosting

- Algorithme de gradient boosting

- Choix des paramètres

- Compléments/conclusion

- Xgboost

Bibliographie



Breiman, L. (1996).

**Bagging predictors.**

*Machine Learning*, 26(2) :123–140.



Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984).

**Classification and regression trees.**


Wadsworth & Brooks.



Chen, T. and Guestrin, C. (2016).

**XGBoost : A scalable tree boosting system.**

In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

 Fernández-Delgado, M., Cernadas, E., Barro, S., and Amorim, D. (2014).

**Do we need hundreds of classifiers to solve real world classification problems ?**

*Journal of Machine Learning Research*, 15 :3133–3181.

 Freund, Y. and Schapire, R. (1996).

**Experiments with a new boosting algorithm.**

In *Proceedings of the Thirteenth International Conference on Machine Learning*.

 Friedman, J. H. (2001).

**Greedy function approximation : A gradient boosting machine.**

*Annals of Statistics*, 29 :1189–1232.



Friedman, J. H. (2002).

**Stochastic gradient boosting.**

*Computational Statistics & Data Analysis*, 28 :367–378.



Hastie, T., Tibshirani, R., and Friedman, J. (2009).

**The Elements of Statistical Learning : Data Mining, Inference, and Prediction.**

Springer, second edition.



Wright, M. and Ziegler, A. (2017).

**ranger : A fast implementation of random forests for high dimensional data in c++ and r.**

*Journal of Statistical Software*, 17(1).