

# Machine learning

L. Rouvière

*laurent.rouviere@univ-rennes2.fr*

MAI 2021

## Table des matières

<b>I</b>	<b>Apprentissage : contexte et formalisation</b>	<b>4</b>
1	Motivations	4
2	Quelques exemples	6
3	Cadre statistique pour l'apprentissage supervisé	8
4	Exemples de fonction de perte	10
5	Estimation du risque	15
6	Le sur-apprentissage	17
7	Le package tidymodels	19
8	Annexe : le package caret	22
9	Bibliographie	25
<b>II</b>	<b>Algorithmes linéaires</b>	<b>27</b>
1	Estimation par moindres carrés	28
2	Sélection de variables	30
3	Régularisation	33
3.1	Régression ridge	34
3.2	Régression Lasso	36
3.3	Variantes de ridge/lasso	39
3.4	Discrimination binaire	41
4	Support vector machine	42
4.1	SVM - cas séparable	43
4.2	SVM : cas non séparable	47
4.3	SVM non linéaire : astuce du noyau	52
4.4	Scores et probabilités	56
4.5	Compléments : SVM multi-classes et SVR	57
4.5.1	SVM multiclassés	57
4.5.2	Support vector regression (SVR)	59
5	Bibliographie	62
<b>III</b>	<b>Algorithmes non linéaires</b>	<b>64</b>

<b>1 Arbres</b>	<b>64</b>
1.1 Arbres binaires	64
1.2 Choix des coupures	67
1.2.1 Cas de la régression	68
1.2.2 Cas de la classification supervisée	68
1.3 Elagage	70
1.4 Importance des variables	74
<b>2 Réseaux de neurones</b>	<b>76</b>
2.1 Introduction	76
2.2 Le perceptron simple	77
2.3 Perceptron multicouches	79
2.4 Estimation	81
2.5 Choix des paramètres et surapprentissage	84
<b>3 Bibliographie</b>	<b>86</b>
<b>IV Agrégation</b>	<b>87</b>
<b>1 Bagging et forêts aléatoires</b>	<b>87</b>
1.1 Bagging	88
1.2 Forêts aléatoires	89
1.2.1 Algorithme	90
1.2.2 Choix des paramètres	91
1.2.3 Erreur OOB et importance des variables	92
<b>2 Boosting</b>	<b>95</b>
2.1 Algorithme de gradient boosting	96
2.2 Choix des paramètres	97
2.3 Compléments/conclusion	101
<b>3 Bibliographie</b>	<b>102</b>

## Présentation

- *Objectifs* : comprendre les aspects **théoriques** et **pratiques** des algorithmes machine learning de référence.
- *Pré-requis* : théorie des probabilités, modélisation statistique, régression (linéaire et logistique). 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** : energie, finance, marketing, sport.

## Programme

- *Matériel* :
  - slides : [https://lrouviere.github.io/machine\\_learning/](https://lrouviere.github.io/machine_learning/)
  - Tutoriel long : [https://lrouviere.github.io/TUTO\\_ML/](https://lrouviere.github.io/TUTO_ML/)
  - Tutoriel court : [https://lrouviere.github.io/machine\\_learning/tuto\\_court\\_ml\\_sans\\_correc.html](https://lrouviere.github.io/machine_learning/tuto_court_ml_sans_correc.html)
- *4 parties* :
  1. **Machine Learning** : cadre, objectif, risque...
  2. **Algorithmes linéaires** : MCO, régularisation (ridge, lasso), SVM
  3. **Algorithmes linéaires** : arbres et réseaux de neurones
  4. **Agrégation** : forêts aléatoires et boosting

## Objectifs/questions

- *Buzzword* : machine learning, big data, data mining, intelligence artificielle...
- *Machine learning* versus *statistique* (traditionnelle)
- *Risque*  $\implies$  calcul ou estimation : ré-échantillonnage, validation croisée...
- Algorithmes versus estimateurs...
- *Classification* des algorithmes. Tous équivalents? Cadre propice...
- ...

## Première partie

# Apprentissage : contexte et formalisation

## 1 Motivations

### Apprentissage statistique ?

#### Plusieurs "définitions"

1. "... explores way of **estimating functional dependency** from a given collection of data" [Vapnik, 2000].
2. "...vast set of tools for modelling and understanding **complex data**" [James et al., 2015]

#### Wikipedia

L'**apprentissage automatique** (en anglais : machine learning), **apprentissage artificiel** ou **apprentissage statistique** est un champ d'étude de l'**intelligence artificielle** qui se fonde sur des approches *mathématiques et statistiques* pour donner aux **ordinateurs** la capacité d'apprendre à partir de donnée...

⇒ *Interface* : Mathématiques-statistiques/informatique.

#### Constat

- Le *développement des moyens informatiques* fait que l'on est confronté à des données de plus en plus *complexes*.
- Les méthodes *traditionnelles* se révèlent souvent *peu efficaces* face à ce type de données.
- Nécessité de proposer des **algorithmes/modèles statistiques** qui apprennent directement à partir des données.

### Un peu d'histoire - voir [Besse, 2018]

Période	Mémoire	Ordre de grandeur
1940-70	Octet	$n = 30, p \leq 10$
1970	kO	$n = 500, p \leq 10$
1980	MO	Machine Learning
1990	GO	Data-Mining
2000	TO	$p > n$ , apprentissage statistique
2010	PO	$n$ explose, cloud, cluster...
2013	serveurs	Big data
2017	??	Intelligence artificielle...

#### Conclusion

Capacités informatiques ⇒ Data Mining ⇒ Apprentissage statistique ⇒ Big Data ⇒ Intelligence artificielle...

### Approche statistique

#### Objectif ⇒ *expliquer*

- notion de **modèle** ;
- retrouver des lois de probabilités ;
- décisions prises à l'aide de **tests statistiques, intervalles de confiance**.

#### Exemples

- Tests indépendance/adéquation...
- Modèle linéaire : estimation, sélection de variables, analyse des résidus...
- Régression logistique...
- Séries temporelles...

## Approche machine learning

**Objectif**  $\implies$  *prédire*

- notion d'**algorithmes de prévision** ;
- critères d'**erreur de prévision** ;
- **calibration** de paramètres (tuning).

## Exemples

- Algorithmes linéaires (moindres carrés, régularisation, "SVM") ;
- Arbres, réseaux de neurones ;
- **Agrégation** : boosting, bagging (forêts aléatoires) ;
- Deep learning (apprentissage profond).

## Statistique vs apprentissage

- Les objectifs *différent* :
  - recherche de **complexité minimale** en statistique  $\implies$  le modèle doit être **interprétable** !
  - **complexité moins importante** en machine learning  $\implies$  on veut "juste bien prédire".
- Approches néanmoins *complémentaires* :
  - bien expliquer  $\implies$  bien prédire ;
  - "Récentes" évolutions d'aide à l'**interprétation des algorithmes ML**  $\implies$  **scores d'importance** des variables...
  - Un bon algorithme doit posséder des **bonnes propriétés statistiques** (convergence, biais, variance...).

## Conclusion

Ne **pas dissocier** les deux approches.

## Problématiques associées à l'apprentissage

- **Apprentissage supervisé** : prédire une sortie  $y \in \mathcal{Y}$  à partir d'entrées  $x \in \mathcal{X}$  ;
- **Apprentissage non supervisé** : établir une typologie des observations ;
- **Règles d'association** : identifier des liens entre différents produits ;
- **Systèmes de recommandation** : identifier les produits susceptibles d'intéresser des consommateurs.

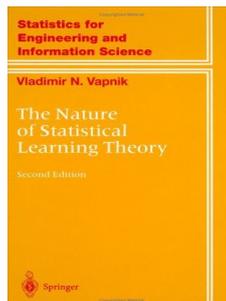
## Nombreuses applications

finance, économie, marketing, biologie, médecine...

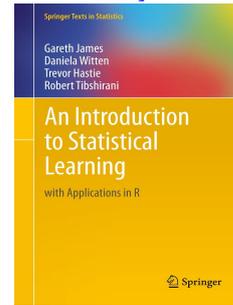
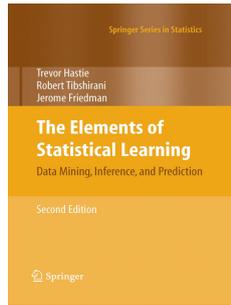
## Théorie de l'apprentissage statistique

**Approche mathématique**

- **Ouvrage fondateur** : [Vapnik, 2000]
- voir aussi [Bousquet et al., 2003].



## The Elements of Statistical Learning [Hastie et al., 2009, James et al., 2015]



— Disponibles (avec jeux de données, codes...) aux url :

<https://web.stanford.edu/~hastie/ElemStatLearn/> <http://www-bcf.usc.edu/~gareth/ISL/>

## Wikistat

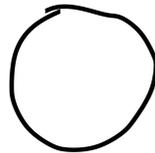
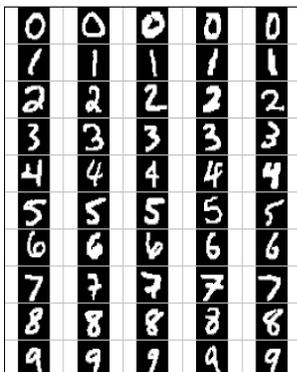
- Page de cours et tutoriels très bien faits sur la *statistique classique et moderne*.
- On pourra notamment regarder les *vignettes* sur la partie *apprentissage* :
  - [Wikistat, 2020a]
  - [Wikistat, 2020b]
  - ...
- Plusieurs parties de ce cours sont *inspirées de ces vignettes*.

## 2 Quelques exemples

### Reconnaissance de l'écriture

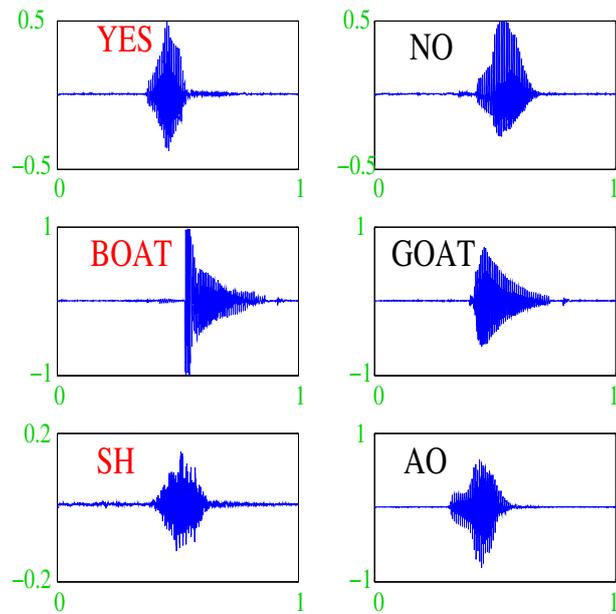
#### *Apprentissage statistique*

Comprendre et apprendre un comportement à partir d'exemples.

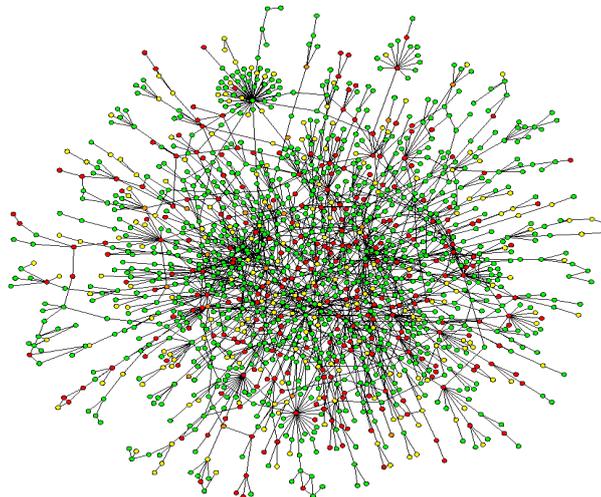


*Qu'est-ce qui est écrit ? 0, 1, 2... ?*

### Reconnaissance de la parole



### Apprentissage sur les réseaux



### Prévision de pics d'ozone

- On a mesuré pendant 366 jours la *concentration maximale* en ozone (V4) ;
- On dispose également d'autres variables météorologiques (température, nébulosité, vent...).

```
> head(Ozone)
##   V1 V2 V3 V4   V5 V6 V7 V8   V9 V10 V11  V12 V13
## 1  1  1  4  3 5480 8 20 NA   NA 5000 -15 30.56 200
## 2  1  2  5  3 5660 6 NA 38   NA  NA  -14  NA  300
## 3  1  3  6  3 5710 4 28 40   NA 2693 -25 47.66 250
## 4  1  4  7  5 5700 3 37 45   NA  590 -24 55.04 100
## 5  1  5  1  5 5760 3 51 54 45.32 1450 25 57.02  60
## 6  1  6  2  6 5720 4 69 35 49.64 1568 15 53.78  60
```

### Question

Peut-on *prédire* la concentration maximale en ozone du *lendemain* à partir des prévisions météorologiques ?

### Détection de spam

- Sur 4601 mails, on a pu identifier *1813 spams*.

- On a également mesuré sur chacun de ces mails la présence ou absence de *57 mots*.

```
> spam %>% select(c(1:8,58)) %>% head()
##   make address all num3d our over remove internet type
## 1 0.00    0.64 0.64    0 0.32 0.00  0.00    0.00 spam
## 2 0.21    0.28 0.50    0 0.14 0.28  0.21    0.07 spam
## 3 0.06    0.00 0.71    0 1.23 0.19  0.19    0.12 spam
## 4 0.00    0.00 0.00    0 0.63 0.00  0.31    0.63 spam
## 5 0.00    0.00 0.00    0 0.63 0.00  0.31    0.63 spam
## 6 0.00    0.00 0.00    0 1.85 0.00  0.00    1.85 spam
```

### Question

Peut-on construire à partir de ces données une méthode de *détection automatique* de spam ?

## 3 Cadre statistique pour l'apprentissage supervisé

### Régression vs classification

- *Données* de type *entrée-sortie* :  $d_n = (x_1, y_1), \dots, (x_n, y_n)$  où  $x_i \in \mathcal{X}$  représente l'entrée et  $y_i \in \mathcal{Y}$  la sortie.

### Objectifs

1. **Expliquer** le(s) mécanisme(s) liant les entrée  $x_i$  aux sorties  $y_i$  ;
2. **Prédire** « au mieux » la sortie  $y$  associée à une nouvelle entrée  $x \in \mathcal{X}$ .

### Vocabulaire

- Lorsque la variable à expliquer est quantitative ( $\mathcal{Y} \subseteq \mathbb{R}$ ), on parle de *régression*.
- Lorsqu'elle est qualitative ( $\text{Card}(\mathcal{Y})$  fini), on parle de *classification (supervisée)*.

### Exemples

- La plupart des problèmes présentés précédemment peuvent être appréhendés dans un contexte d'*apprentissage supervisé* : on cherche à expliquer une sortie  $y$  par des entrées  $x$  :

$y_i$	$x_i$	
Chiffre	image	Classification
Mot	courbe	Classification
Spam	présence/absence de mots	Classification
C. en $O_3$	données météo.	Régression

### Remarque

La nature des variables associées aux *entrées*  $x_i$  est *variée* (quanti, quali, fonctionnelle...).

### Un début de formalisation mathématique

- Etant données des observations  $d_n = \{(x_1, y_1), \dots, (x_n, y_n)\}$  on cherche à *expliquer/prédire* les sorties  $y_i \in \mathcal{Y}$  à partir des entrées  $x_i \in \mathcal{X}$ .
- Il s'agit donc de trouver *une fonction de prévision*  $f : \mathcal{X} \rightarrow \mathcal{Y}$  telle que

$$f(x_i) \approx y_i, i = 1, \dots, n.$$

- Nécessité de se donner un *critère* qui permette de mesurer la qualité des fonctions de prévision  $f$ .
- Le plus souvent, on utilise une *fonction de perte*  $\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$  telle que

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

## Vision statistique

- On suppose que les données  $d_n = \{(x_1, y_1), \dots, (x_n, y_n)\}$  sont des *réalisations d'un n-échantillon*  $\mathcal{D}_n = \{(X_1, Y_1), \dots, (X_n, Y_n)\}$  de loi *inconnue*.
- Les  $X_i$  sont des *variables aléatoires* à valeurs dans  $\mathcal{X}$ , les  $Y_i$  dans  $\mathcal{Y}$ .
- Le plus souvent on supposera que les couples  $(X_i, Y_i), i = 1, \dots, n$  sont *i.i.d* de loi  $\mathbf{P}$ .

## Performance d'une fonction de prévision

- Etant donné une *fonction de perte*  $\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$ , la performance d'une *fonction de prévision*  $f : \mathcal{X} \rightarrow \mathcal{Y}$  est mesurée par

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

où  $(X, Y)$  est indépendant des  $(X_i, Y_i)$  et de même loi  $P$ .

- $\mathcal{R}(f)$  est appelé *risque* ou *erreur de généralisation* de  $f$ .

## Fonction de prévision optimale

- $\mathcal{R}(f) \implies$  "*Erreur moyenne*" de  $f$  par rapport à la loi ds données.
- *Idée* : trouver  $f$  qui a la *plus petite erreur*.

## Aspect théorique

- Pour une fonction de perte  $\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$  donnée, le problème *théorique* consiste à trouver

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

- Une telle fonction  $f^*$  (si elle existe) est appelée *fonction de prévision optimale* pour la perte  $\ell$ .

## Aspect pratique

- La fonction de prévision optimale  $f^*$  dépend le plus souvent de la loi  $\mathbf{P}$  des  $(X, Y)$  qui est en pratique *inconnue*.
- Le job du statisticien est de trouver un *estimateur*  $f_n = f_n(\cdot, \mathcal{D}_n)$  tel que  $\mathcal{R}(f_n) \approx \mathcal{R}(f^*)$ .

## Définition

Un *algorithme de prévision* est représenté par une suite  $(f_n)_n$  d'applications (mesurables) telles que pour  $n \geq 1$ ,  $f_n : \mathcal{X} \times (\mathcal{X} \times \mathcal{Y})^n \rightarrow \mathcal{Y}$ .

## Propriétés statistiques d'un algorithme

- 1 un *algorithme* : 1 *estimateur*  $f_n(\cdot) = f_n(\cdot, \mathcal{D}_n)$  de  $f^*$ .

## Propriétés statistiques

- *Biais* :  $\mathbf{E}[f_n(x)] - f^*(x) \implies$  prévisions "en moyenne";
- *Variance* :  $\mathbf{V}[f_n(x)] \implies$  stabilité des prévisions;
- *Consistance* :  $\lim_{n \rightarrow \infty} \mathcal{R}(f_n) = \mathcal{R}(f^*) \implies$  précision quand  $n$  augmente;
- ...

## 4 Exemples de fonction de perte

### Choix de la fonction de perte

- Le cadre mathématique développé précédemment sous-entend qu'une fonction est *performante* (voire *optimale*) vis-à-vis d'un *critère* (représenté par la *fonction de perte*  $\ell$ ).
- Un algorithme de prévision performant pour un critère ne sera *pas forcément performant pour un autre*.

### Conséquence pratique

Avant de s'attacher à construire un algorithme de prévision, il est *capital* de savoir *mesurer la performance* d'un algorithme de prévision.

- Une fonction de perte  $\ell : \mathcal{Y} \times \tilde{\mathcal{Y}} \rightarrow \mathbb{R}^+$  dépend de l'*espace des observations*  $\mathcal{Y}$  et de celui des *prévisions*  $\tilde{\mathcal{Y}}$ .
- On distingue **3 catégories** de fonction de perte en fonction de ces espaces :
  1. Prévisions *numériques* : problème de *régression* où on cherche à prédire la *valeur* de  $Y : \ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+$  ;
  2. Prévision *de groupes* : problème de *classification* où on veut prédire un *label* :  $\ell : \{1, \dots, K\} \times \{1, \dots, K\} \rightarrow \mathbb{R}^+$  ;
  3. Prévision de *probabilités* : problème de *classification* où on veut prédire les *probabilités*  $\mathbf{P}(Y = k|X = x)$  :  $\ell : \{1, \dots, K\} \times \mathbb{R}^K \rightarrow \mathbb{R}^+$ .

### Régression

- $\mathcal{Y} = \mathbb{R}$ , une prévision = un réel  $\implies m : \mathcal{X} \rightarrow \mathbb{R}$  ;
- Une perte = une *distance* entre deux nombres, par exemple la *perte quadratique* :

$$\begin{aligned}\ell : \mathbb{R} \times \mathbb{R} &\rightarrow \mathbb{R}^+ \\ (y, y') &\mapsto (y - y')^2\end{aligned}$$

- Le *risque* (*risque quadratique*) est alors donné par

$$\mathcal{R}(m) = \mathbf{E}[(Y - m(X))^2]$$

- et la *fonction optimale (inconnue)*, appelée *fonction de régression*, par

$$m^*(x) = \mathbf{E}[Y|X = x].$$

### Classification

- $\mathcal{Y} = \{1, \dots, K\}$ , une prévision = un *groupe*  $\implies g : \mathcal{X} \rightarrow \{1, \dots, K\}$  ;
- Une perte = 1 *coût* pour une mauvaise prévision, par exemple la *perte indicatrice*

$$\begin{aligned}\ell : \{1, \dots, K\} \times \{1, \dots, K\} &\rightarrow \mathbb{R}^+ \\ (y, y') &\mapsto \mathbf{1}_{y \neq y'}\end{aligned}$$

- Le *risque* (*erreur de classification*) est alors donné par

$$\mathcal{R}(g) = \mathbf{E}[\mathbf{1}_{g(X) \neq Y}] = \mathbf{P}(g(X) \neq Y).$$

- et la *fonction optimale (inconnue)*, appelée *règle de Bayes*, par

$$g^*(x) = \operatorname{argmax}_k \mathbf{P}(Y = k|X = x).$$

## Classification binaire

- $\mathcal{Y} = \{-1, 1\}$ , une prévision = un **groupe**  $\implies g : \mathcal{X} \rightarrow \{-1, 1\}$ .
- Ce cadre permet une *analyse plus fine* des différents types d'erreur.
- En effet, seules **4 situations** peuvent se produire

	$g(x) = -1$	$g(x) = 1$
$y = -1$	VN	FP
$y = 1$	FN	VP

- On peut les quantifier en terme de *probabilités*.

Pour aller plus vite

## Erreurs binaires

- *Spécificité*  $\implies$  bien prédire les négatifs :

$$\text{sp}(g) = \mathbf{P}(g(X) = -1|Y = -1),$$

- *Sensibilité*  $\implies$  bien prédire les positifs :

$$\text{se}(g) = \mathbf{P}(g(X) = 1|Y = 1),$$

- *Taux de faux négatifs*  $\implies$  prédire négatif à tort :

$$\text{fn}(g) = \mathbf{P}(g(X) = -1|Y = 1),$$

- *Taux de faux positifs*  $\implies$  prédire positif à tort :

$$\text{fp}(g) = \mathbf{P}(g(X) = 1|Y = -1).$$

## Critères binaires

De nombreux critères s'obtiennent en combinant ces probabilités :

$$\text{EC}(g) = \mathbf{P}(g(X) \neq Y) = \text{fp}(g)\mathbf{P}(Y = -1) + \text{fn}(g)\mathbf{P}(Y = 1).$$

## Quelques critères binaires

- *Balanced Accuracy* :

$$\frac{1}{2}\mathbf{P}(g(X) = -1|Y = -1) + \frac{1}{2}\mathbf{P}(g(X) = 1|Y = 1) = \frac{1}{2}(\text{se}(g) + \text{sp}(g)).$$

- *F<sub>1</sub>-score* :

$$2 \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}},$$

avec

$$\text{Precision } \mathbf{P}(Y = 1|g(X) = 1) \quad \text{et} \quad \text{Recall} = \mathbf{P}(g(X) = 1|Y = 1).$$

- *Kappa de Cohen*...

## Remarque

Mieux adapté que l'erreur de classification au cas de *données déséquilibrées*.

## Classification (pour des probabilités)

- $\mathcal{Y} = \{1, \dots, K\}$ , une prévision =  **$K - 1$  probabilités**  $p_k(x) = \mathbf{P}(Y = k|X = x), k = 1, \dots, K - 1$ .
- Les fonctions de perte sont généralement définies comme généralisation de pertes spécifiques au problème de *classification binaire*.
- **Classification binaire** avec  $\mathcal{Y} = \{-1, 1\}$  et  $S : \mathcal{X} \rightarrow \mathbb{R}$  ( $S(x) = \mathbf{P}(Y = 1|X = x)$  ou une transformation bijective de cette probabilité)  $\implies$  fonction de **score**.



### Fonction de score

- *Objectif d'un score* : ordonner
- avant (d'éventuellement) classer en fixant un seuil  $s \in \mathbb{R}$  :

$$g_s(x) = \begin{cases} 1 & \text{si } S(x) > s \\ -1 & \text{sinon.} \end{cases}$$

- Pour un seuil  $s$  donné, on a les erreurs (FP et FN)

$$\alpha(s) = \mathbf{P}(S(X) > s | Y = -1) = 1 - \text{sp}(s)$$

et

$$\beta(s) = \mathbf{P}(S(X) \leq s | Y = 1) = 1 - \text{se}(s).$$

### Courbe ROC

- *Idée* : s'affranchir du choix de  $s$  en visualisant les erreurs  $\alpha(s)$  et  $\beta(s)$  sur un graphe 2D pour toutes les valeurs de  $s$ .

#### Définition

La *courbe ROC* de  $S$  est la courbe paramétrée par les valeurs de seuil  $s$  dont les abscisses et ordonnées sont définies par

$$\begin{cases} x(s) = \mathbf{P}(S(X) > s | Y = -1) = \alpha(s) \\ y(s) = \mathbf{P}(S(X) > s | Y = 1) = 1 - \beta(s). \end{cases}$$

#### Visualisation

- *Abscisses* : les faux positifs ou la spécificité ;
- *Ordonnées* : les faux négatifs ou la sensibilité.

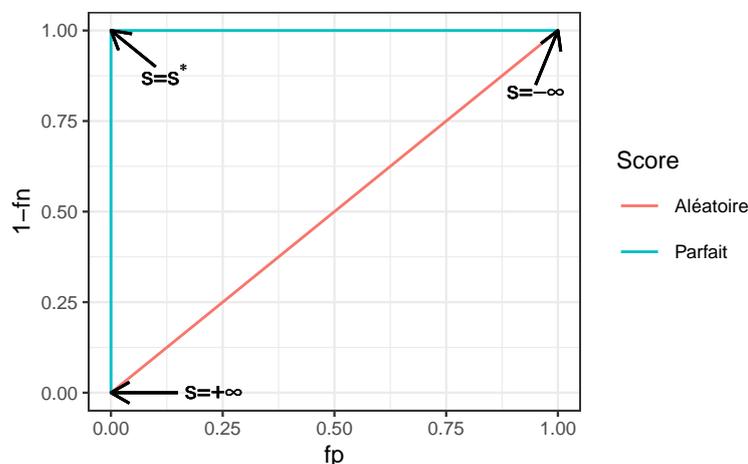
### Analyse de la courbe ROC

- Une proba est entre 0 et 1  $\implies$  ROC vit dans le carré  $[0, 1]^2$ .
- $x(-\infty) = y(-\infty) = 1$  et  $x(+\infty) = y(+\infty) = 0 \implies$  ROC part du point  $(1, 1)$  pour arriver en  $(0, 0)$ .
- *ROC parfaite* : il existe  $s^*$  tel que  $\alpha(s^*) = \beta(s^*) = 0 \implies$  ROC est définie par l'union des segments

$$[(1, 1); (0, 1)] \quad \text{et} \quad [(0, 1); (0, 0)].$$

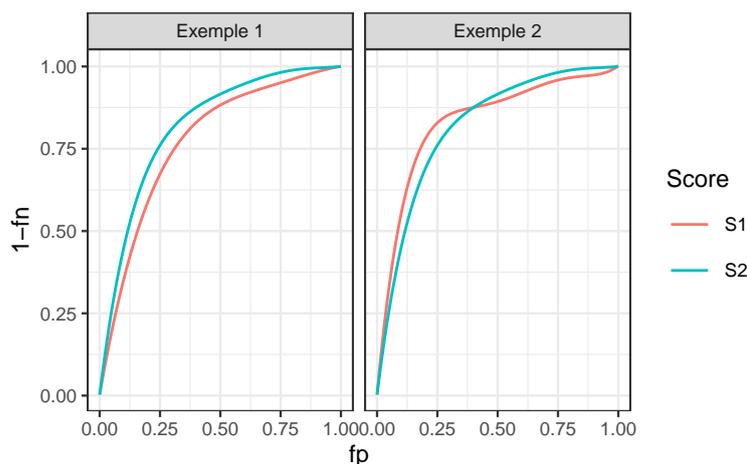
- *Mauvaise ROC* :  $S(X)$  et  $Y$  sont indépendantes  $\implies x(s) = y(s)$  pour tout  $s \in \mathbb{R}$  et ROC correspond à la première bissectrice.

### Visualisation



## Interprétation

On évalue la *performance d'un score* par sa *capacité à se rapprocher* le plus vite possible de la droite  $y = 1$ .



## Comparaison

- Exemple 1 : S2 meilleur que S1.
- Exemple 2 : il y a débat...
- Idée : utiliser l'*aire sous la courbe*.

## AUC

### Définition

On appelle AUC l'aire sous la courbe ROC de  $S$ .

### Propriété

- $0.5 \leq \text{AUC}(S) \leq 1$ .
- Plus l'AUC est *grand*, *meilleur* est le score.

## Interprétation de l'AUC

### Propriété

Soit  $(X_1, Y_1)$  et  $(X_2, Y_2)$  indépendants et de même loi que  $(X, Y)$ , on a

$$\begin{aligned} \text{AUC}(S) &= \mathbf{P}(S(X_1) > S(X_2) | Y_1 = 1, Y_2 = -1) \\ &\quad + \frac{1}{2} \mathbf{P}(S(X_1) = S(X_2) | Y_1 = 1, Y_2 = -1). \end{aligned}$$

En particulier si  $S(X)$  est continue alors

$$\text{AUC}(S) = \mathbf{P}(S(X_1) \geq S(X_2) | Y_1 = 1, Y_2 = -1).$$

### Interprétation

- L'AUC correspond à la probabilité que le score *ordonne correctement deux observations* prélevées aléatoirement dans les groupes -1 et 1.
- $\text{AUC}(S) = 0.9 \implies$  dans 90% des cas, le score d'un individu positif sera plus grand que le score d'un individu négatif.

## Perte AUC et score optimal

— Remarquons que

$$\text{AUC}(S) = \mathbf{E}[\mathbf{1}_{S(X_1) > S(X_2)} | Y_1 = 1, Y_2 = -1].$$

— L'AUC peut donc s'écrire comme l'*espérance d'une fonction de perte particulière*

$$\ell((y_1, y_2), (S(x_1), S(x_2))) = \mathbf{1}_{S(x_1) > S(x_2)} \quad \text{avec } y_1 = 1 \text{ et } y_2 = -1.$$

### Proposition

Le **score optimal** par rapport à l'AUC est

$$S^*(x) = \mathbf{P}(Y = 1 | X = x).$$

En effet pour tout score  $S : \mathcal{X} \rightarrow \mathbb{R}$  on a

$$\text{AUC}(S^*) \geq \text{AUC}(S).$$

## Résumé

	Perte $\ell(y, f(x))$	Risque $\mathcal{R}(f)$	Champion $f^*$
Régression	$(y - m(x))^2$	$\mathbf{E}[Y - m(X)]^2$	$\mathbf{E}[Y   X = x]$
Classif. binaire	$\mathbf{1}_{y \neq g(x)}$	$\mathbf{P}(Y \neq g(X))$	Bayes
Scoring	$\mathbf{1}_{S(x_1) > S(x_2)}$	$\text{AUC}(S)$	$\mathbf{P}(Y = 1   X = x)$

## Le package yardstick

— Nous verrons dans la section suivante que ces *critères* se **calculent** (ou plutôt s'**estiment**) en confrontant les valeurs *observées*  $y_i$  aux valeurs *prédites* d'un algorithme. Par exemple

```
> head(tbl)
## # A tibble: 6 x 3
##   obs   proba class
##   <dbl> <dbl> <dbl>
## 1  0     0.117  0
## 2  0     0.288  0
## 3  1     0.994  1
## 4  0     0.528  1
## 5  0     0.577  1
## 6  1     0.997  1
```

— Le package yardstick contient un ensemble de fonctions qui permettent de calculer les critères :

<https://yardstick.tidymodels.org/articles/metric-types.html>

## Exemples

— Erreur de classification (ou plutôt accuracy) avec **accuracy** :

```
> library(yardstick)
> tbl %>% accuracy(truth=obs, estimate=class)
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>       <dbl>
## 1 accuracy binary      0.834
```

— AUC avec **roc\_auc**

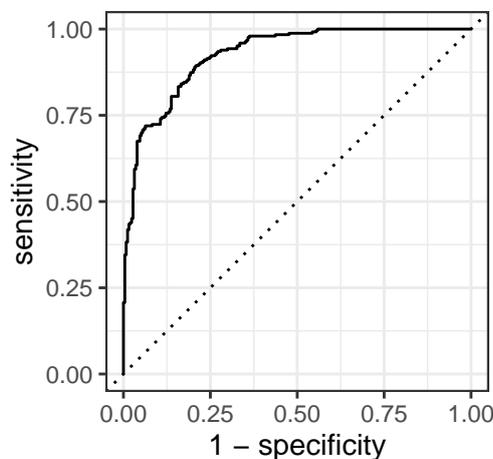
```
> tbl %>% roc_auc(truth=obs, estimate=proba, event_level="second")
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>       <dbl>
## 1 roc_auc binary       0.926
```

— On peut aussi définir plusieurs critères :

```
> multi_metric <- metric_set(accuracy, bal_accuracy, f_meas, kap)
> tbl %>% multi_metric(truth=obs, estimate=class, event_level="second")
## # A tibble: 4 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>       <dbl>
## 1 accuracy binary       0.834
## 2 bal_accuracy binary       0.834
## 3 f_meas   binary       0.832
## 4 kap      binary       0.668
```

— et tracer des courbes ROC avec `roc_curve` et `autoplot`

```
> tbl %>% roc_curve(truth=obs, estimate=proba, event_level="second") %>%
+ autoplot()
```



## 5 Estimation du risque

### Rappels

—  $n$  observations  $(X_1, Y_1), \dots, (X_n, Y_n)$  i.i.d à valeurs dans  $\mathcal{X} \times \mathcal{Y}$ .

### Objectif

Etant donnée une fonction de perte  $\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$ , on cherche un **algorithme de prévision**  $f_n(x) = f_n(x, \mathcal{D}_n)$  qui soit "proche" de l'oracle  $f^*$  défini par

$$f^* \in \underset{f}{\operatorname{argmin}} \mathcal{R}(f)$$

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

### Question

Etant donné un algorithme  $f_n$ , *que vaut son risque*  $\mathcal{R}(f_n)$  ?

### Risque empirique

— La loi de  $(X, Y)$  étant *inconnue* en pratique, il est *impossible de calculer*  $\mathcal{R}(f_n) = \mathbf{E}[\ell(Y, f_n(X))]$ .

— **Première approche** :  $\mathcal{R}(f_n)$  étant une espérance, on peut l'estimer (LGN) par sa *version empirique*

$$\mathcal{R}_n(f_n) = \frac{1}{n} \sum_{i=1}^n \ell(Y_i, f_n(X_i)).$$

## Problème

- L'échantillon  $\mathcal{D}_n$  a *déjà été utilisé* pour construire l'algorithme de prévision  $f_n \implies$  La LGN ne peut donc s'appliquer !
- *Conséquence* :  $\mathcal{R}_n(f_n)$  conduit souvent à une *sous-estimation* de  $\mathcal{R}(f_n)$ .

## Une solution

Utiliser des méthodes de type *validation croisée* ou *bootstrap*.

## Apprentissage - Validation ou Validation hold out

- Elle consiste à séparer l'échantillon  $\mathcal{D}_n$  en :
  1. un *échantillon d'apprentissage*  $\mathcal{D}_{n,app}$  pour construire  $f_n$  ;
  2. un *échantillon de validation*  $\mathcal{D}_{n,test}$  utilisé pour estimer le risque de  $f_n$ .

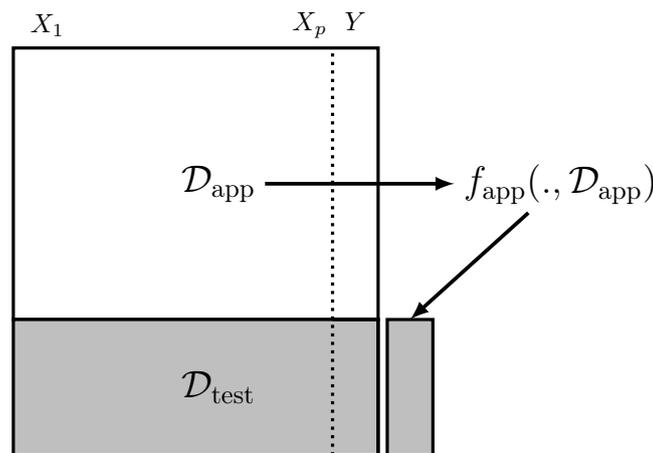
## Algorithme

**Entrée** :  $\{\mathcal{A}, \mathcal{T}\}$  une partition de  $\{1, \dots, n\}$  en deux parties.

1. Ajuster l'algorithme de prévision en utilisant *uniquement les données d'apprentissage*  $\mathcal{D}_{app} = \{(x_i, y_i) : i \in \mathcal{A}\}$ . On désigne par  $f_{app}(\cdot, \mathcal{D}_{app})$  l'algorithme obtenu.
2. Calculer les valeurs prédites  $f_{app}(x_i, \mathcal{D}_{app})$  par l'algorithme pour chaque observation de l'échantillon test  $\mathcal{D}_{test} = \{(x_i, y_i) : i \in \mathcal{T}\}$

**Retourner** :

$$\frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} \ell(y_i, f_{app}(x_i, \mathcal{D}_{app})).$$



## Commentaires

Nécessite d'avoir un *nombre suffisant d'observations* dans

1.  $\mathcal{D}_{app}$  pour bien ajuster l'algorithme de prévision ;
2.  $\mathcal{D}_{test}$  pour bien estimer l'erreur de l'algorithme.

## Validation croisée K-blocs

- **Principe** : répéter la hold out sur *différentes partitions*.

## Algorithme - CV

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

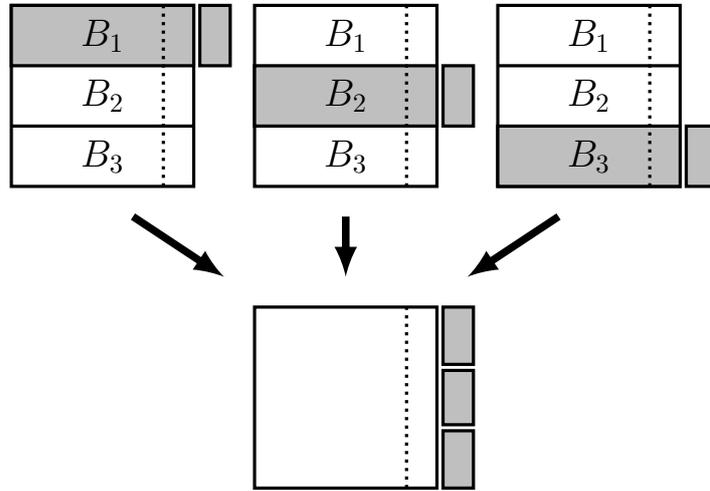
Pour  $k = 1, \dots, K$  :

1. Ajuster l'algorithme de prévision en utilisant *l'ensemble des données privé du  $k^e$  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  $k$**  :

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

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



### Commentaires

- Le *choix de  $K$*  doit être fait par l'utilisateur (souvent  $K = 10$ ).
- *Avantage* : plus adapté que la technique apprentissage/validation  $\implies$  **plus stable et précis**.
- *Inconvénient* : plus couteux en **temps de calcul**.

### Leave one out

- Lorsque  $K = n$ , on parle de validation croisée *leave one out*;
- Le risque est alors estimé par

$$\widehat{\mathcal{R}}_n(f_n) = \frac{1}{n} \sum_{i=1}^n \ell(Y_i, f_n^i(X_i))$$

où  $f_n^i$  désigne l'algorithme de prévision construit sur  $\mathcal{D}_n$  *amputé de la  $i$ -ème observation*.

$\implies$  recommandé uniquement lorsque  $n$  est petit.

### Autres approches

- *Estimation par pénalisation* : critère **ajustement/complexité**,  $C_p$  de Mallows, AIC-BIC...
- *Validation croisée Monte-Carlo* : répéter plusieurs fois la validation hold out ;
- *Bootstrap* : notamment **Out Of Bag** ;
- voir [Wikistat, 2020b].

## 6 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.

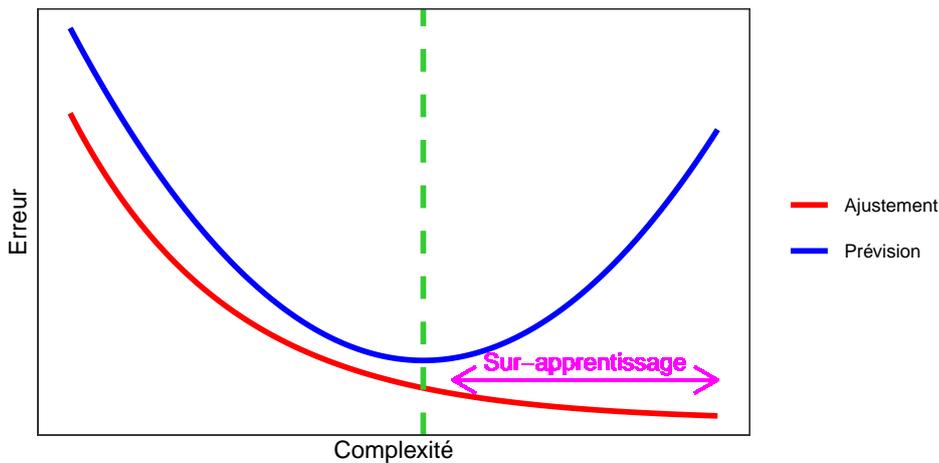
- Le paramètre  $\lambda$  à sélectionner représente la complexité du modèle :

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

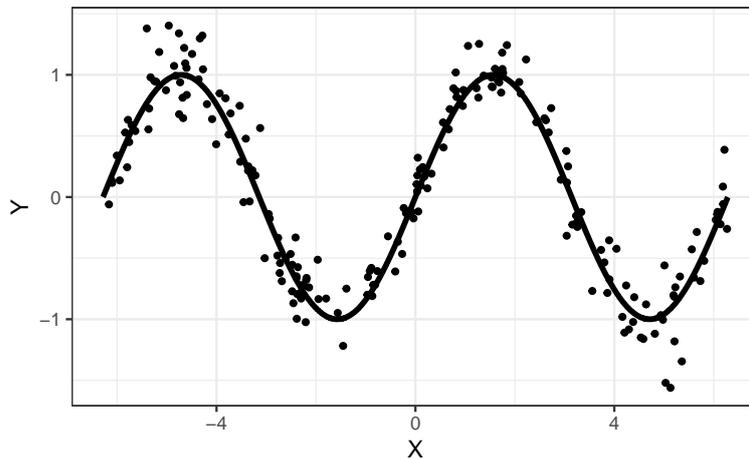
- $\lambda$  petit  $\implies$  modèle peu flexible  $\implies$  mauvaise adéquation sur les données  $\implies$  biais  $\nearrow$ , variance  $\searrow$ .
- $\lambda$  grand  $\implies$  modèle trop flexible  $\implies$  sur-ajustement  $\implies$  biais  $\searrow$ , variance  $\nearrow$ .

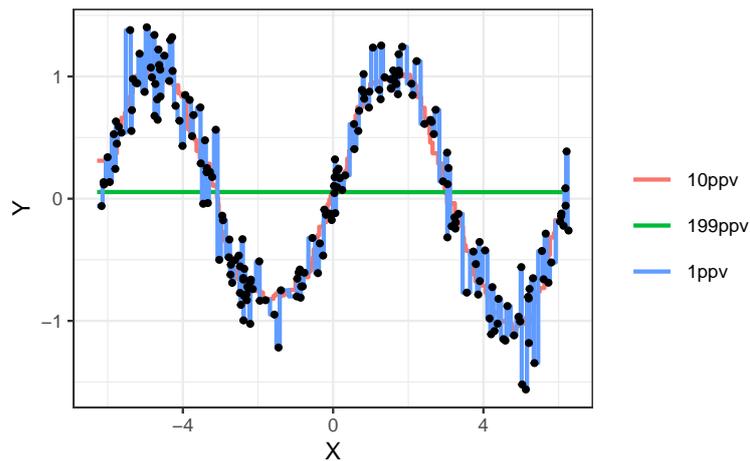
**Overfitting**

*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.

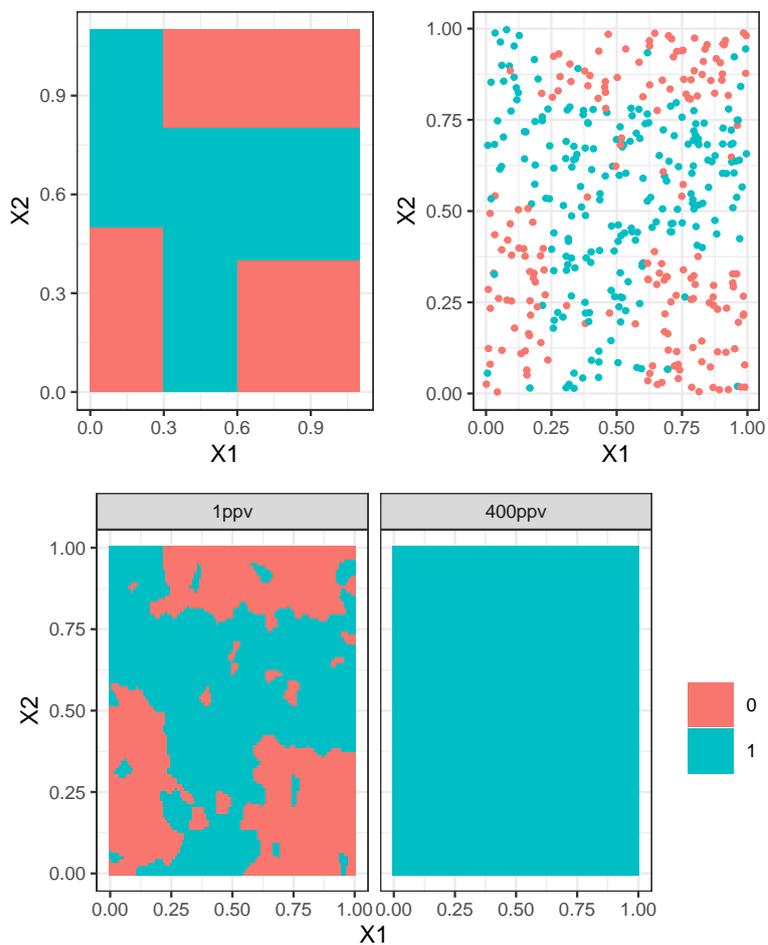


**Overfitting en régression**





## Overfitting en classification supervisée



*Application shiny*

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

## 7 Le package tidymodels

### Présentation du package

- Successeur de `caret` pour conduire des projets machine learning sur R.
- Meta package qui inclut

- `rsample` : pour ré-échantillonner
- `yardstick` : pour les fonctions de perte
- `recipe` : pour les recettes de préparation... des données
- `tune` : pour calibrer les algorithmes
- ...
- Tutoriel : <https://www.tidymodels.org>

## Calibrer des paramètres

- Tous les algorithmes dépendent de *paramètres*  $\theta$  que l'utilisateur doit sélectionner.
- Le procédé est *toujours le même* et peut se résumer dans l'algorithme suivant.

### Choix de paramètres par minimisation du risque (grid search)

#### Entrées :

- Une grille `grille.theta` de valeurs pour  $\theta$ ;
- Un risque de prévision  $\mathcal{R}$ ;
- un algorithme d'estimation du risque.

Pour chaque  $\theta$  dans `grille.theta` :

- Estimer  $\mathcal{R}(f_{n,\theta})$  par l'algorithme choisi  $\implies \widehat{\mathcal{R}}(f_{n,\theta})$

**Retourner** :  $\hat{\theta}$  une valeur de  $\theta$  qui minimise  $\widehat{\mathcal{R}}(f_{n,\theta})$ .

- Ce procédé est *automatisé* dans `tidymodels`.
- Il faut spécifier les différents paramètres :
  - la *méthode* (logistique, ppv, arbre, randomForest...)
  - Une grille pour les *paramètres* (nombre de ppv...)
  - Le *critère de performance* (erreur de classification, AUC, risque quadratique...)
  - La méthode d'*estimation du critère* (apprentissage validation, validation croisée, bootstrap...)
- Nous l'illustrons à travers le *choix du nombre de voisins* de l'algorithme des  $k$ -ppv.

## Les données

- Une variable binaire à expliquer par 2 variables continues

```
> head(don.2D.500)
## # A tibble: 6 x 3
##   X1     X2 Y
##   <dbl> <dbl> <fct>
## 1 0.721 0.209 0
## 2 0.876 0.766 1
## 3 0.761 0.842 1
## 4 0.886 0.934 0
## 5 0.456 0.676 0
## 6 0.166 0.859 1
```

## Le workflow

- On commence par renseigner l'*algorithme* et la manière dont on va *choisir les paramètres*.

```
> library(tidymodels)
> tune_spec <-
+ nearest_neighbor(neighbors=tune(), weight_func="rectangular") %>%
+ set_mode("classification") %>%
+ set_engine("knn")
```

- On crée ensuite la *workflow* :

```
> ppv_wf <- workflow() %>%
+ add_model(tune_spec) %>%
+ add_formula(Y ~ .)
```

## Ré-échantillonnage et grille de paramètres

- On spécifie ensuite la *méthode de ré-échantillonnage*, ici une *validation croisée 10 blocs*

```
> set.seed(12345)
> re_ech_cv <- vfold_cv(don.2D.500,v=10)
> re_ech_cv %>% head()
## # A tibble: 6 x 2
##   splits      id
##   <list>     <chr>
## 1 <split [450/50]> Fold01
## 2 <split [450/50]> Fold02
## 3 <split [450/50]> Fold03
## 4 <split [450/50]> Fold04
## 5 <split [450/50]> Fold05
## 6 <split [450/50]> Fold06
```

- Puis vient la *grille de paramètres*

```
> grille_k <- tibble(neighbors=1:100)
```

⇒ consulter <https://www.tidymodels.org/find/parsnip/> pour trouver les *identifiants* des algorithmes et de leurs paramètres.

## Estimation du risque

- Fonction `tune_grid`

```
> tune_grid(...,resamples=...,grid=...,metrics=...)
```

- Calcul du *risque* pour chaque valeur de la grille :

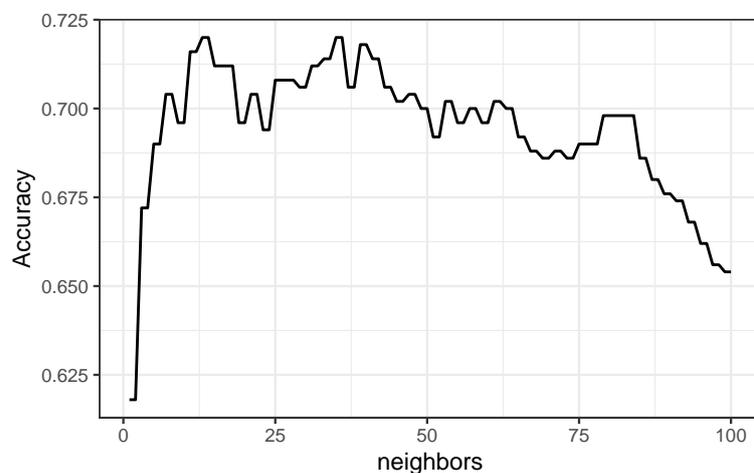
```
> ppv.cv <- ppv_wf %>%
+   tune_grid(
+     resamples = re_ech_cv,
+     grid = grille_k,
+     metrics=metric_set(accuracy))
```

- On lit les résultats avec `collect_metrics` :

```
> ppv.cv %>% collect_metrics() %>% select(1:5) %>% head()
## # A tibble: 6 x 5
##   neighbors .metric .estimator mean    n
##   <int> <chr> <chr> <dbl> <int>
## 1     1 accuracy binary  0.618    10
## 2     2 accuracy binary  0.618    10
## 3     3 accuracy binary  0.672    10
## 4     4 accuracy binary  0.672    10
## 5     5 accuracy binary  0.69     10
## 6     6 accuracy binary  0.69     10
```

## Visualisation des erreurs

```
> tbl <- ppv.cv %>% collect_metrics()
> ggplot(tbl)+aes(x=neighbors,y=mean)+geom_line()+ylab("Accuracy")
```



## Sélection du meilleur paramètre

- On visualise les *meilleures* valeurs de paramètres :

```
> ppv.cv %>% show_best() %>% select(1:6)
## # A tibble: 5 x 6
##   neighbors .metric .estimator mean   n std_err
##   <int> <chr> <chr> <dbl> <int> <dbl>
## 1     13 accuracy binary  0.72    10  0.0255
## 2     14 accuracy binary  0.72    10  0.0255
## 3     35 accuracy binary  0.72    10  0.0207
## 4     36 accuracy binary  0.72    10  0.0207
## 5     39 accuracy binary  0.718   10  0.0199
```

- et on choisit celle qui *maximise l'accuracy* :

```
> best_k <- ppv.cv %>% select_best()
> best_k
## # A tibble: 1 x 2
##   neighbors .config
##   <int> <chr>
## 1     13 Preprocessor1_Model1013
```

## Algorithme final et prévision

- L'*algorithme final* s'obtient en entrainant la méthode sur *toutes les données* pour la *valeur de paramètre sélectionné* :

```
> final_ppv <-
+   ppv_wf %>%
+   finalize_workflow(best_k) %>%
+   fit(data = don.2D.500)
```

- On peut maintenant prédire de nouveaux individus :

```
> newx <- tibble(X1=0.3, X2=0.8)
> predict(final_ppv, new_data=newx)
## # A tibble: 1 x 1
##   .pred_class
##   <fct>
## 1 0
```

## Conclusion

- Les *choix* de l'utilisateur sont des *paramètres* de la procédure.
- $\implies$  facilement *personnalisable*.
- Aisé de changer le critère, la méthode de ré-échantillonnage...

## 8 Annexe : le package caret

### Le package caret

- Il permet d'évaluer la performance de *plus de 230 méthodes* : <http://topepo.github.io/caret/index.html>
- Il suffit d'indiquer :
  - la *méthode* (logistique, ppv, arbre, randomForest...)
  - Une grille pour les *paramètres* (nombre de ppv...)
  - Le *critère de performance* (erreur de classification, AUC, risque quadratique...)
  - La méthode d'*estimation du critère* (apprentissage validation, validation croisée, bootstrap...)

### Apprentissage-validation

```

> library(caret)
> K_cand <- data.frame(k=seq(1,500,by=20))
> library(caret)
> ctrl1 <- trainControl(method="LGOCV",number=1,index=list(1:1500))
> e1 <- train(Y~.,data=donnees,method="knn",trControl=ctrl1,tuneGrid=K_cand)
> e1
## k-Nearest Neighbors
##
## 2000 samples
## 2 predictor
## 2 classes: '0', '1'
##
## No pre-processing
## Resampling: Repeated Train/Test Splits Estimated (1 reps, 75%)
## Summary of sample sizes: 1500
## Resampling results across tuning parameters:
##
## k Accuracy Kappa
## 1 0.620 0.2382571
## 21 0.718 0.4342076
## 41 0.722 0.4418388
## 61 0.718 0.4344073
## 81 0.720 0.4383195
## 101 0.714 0.4263847
## 121 0.716 0.4304965
## 141 0.718 0.4348063
## 161 0.718 0.4348063
## 181 0.718 0.4348063
## 201 0.720 0.4387158
## 221 0.718 0.4350056
## 241 0.718 0.4350056
## 261 0.722 0.4428232
## 281 0.714 0.4267894
## 301 0.714 0.4269915
## 321 0.710 0.4183621
## 341 0.696 0.3893130
## 361 0.696 0.3893130
## 381 0.688 0.3727988
## 401 0.684 0.3645329
## 421 0.686 0.3686666
## 441 0.686 0.3679956
## 461 0.684 0.3638574
## 481 0.680 0.3558050
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 261.

```

## Validation croisée

```

> library(doMC)
> registerDoMC(cores = 3)
> ctrl2 <- trainControl(method="cv",number=10)
> e2 <- train(Y~.,data=dapp,method="knn",trControl=ctrl2,tuneGrid=K_cand)
> e2
## k-Nearest Neighbors
##
## 1500 samples
## 2 predictor
## 2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1350, 1350, 1350, 1350, 1350, 1350, ...
## Resampling results across tuning parameters:
##
## k Accuracy Kappa
## 1 0.6240000 0.2446251
## 21 0.7393333 0.4745290
## 41 0.7306667 0.4570024
## 61 0.7340000 0.4636743

```

```

##      81  0.7333333  0.4632875
##     101  0.7313333  0.4593480
##     121  0.7326667  0.4624249
##     141  0.7333333  0.4640787
##     161  0.7366667  0.4708178
##     181  0.7313333  0.4602309
##     201  0.7326667  0.4626618
##     221  0.7293333  0.4559741
##     241  0.7306667  0.4585960
##     261  0.7353333  0.4676751
##     281  0.7286667  0.4537842
##     301  0.7253333  0.4463516
##     321  0.7173333  0.4294524
##     341  0.7113333  0.4168003
##     361  0.7080000  0.4099303
##     381  0.7140000  0.4213569
##     401  0.7073333  0.4073761
##     421  0.7100000  0.4126434
##     441  0.7066667  0.4054984
##     461  0.6966667  0.3844183
##     481  0.6860000  0.3612515
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 21.

```

## Validation croisée répétée

```

> ctrl3 <- trainControl(method="repeatedcv",repeats=5,number=10)
> e3 <- train(Y~.,data=dapp,method="knn",trControl=ctrl3,tuneGrid=K_cand)
> e3
## k-Nearest Neighbors
##
## 1500 samples
##   2 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 1350, 1350, 1350, 1350, 1350, 1350, ...
## Resampling results across tuning parameters:
##
##   k    Accuracy   Kappa
##   1  0.6232000  0.2438066
##   21  0.7354667  0.4665640
##   41  0.7314667  0.4585144
##   61  0.7317333  0.4592608
##   81  0.7302667  0.4568784
##  101  0.7310667  0.4589567
##
##   121  0.7320000  0.4609326
##   141  0.7322667  0.4616077
##   161  0.7336000  0.4643374
##   181  0.7340000  0.4649895
##   201  0.7332000  0.4632905
##   221  0.7325333  0.4620114
##   241  0.7316000  0.4600484
##   261  0.7305333  0.4578098
##   281  0.7286667  0.4536040
##   301  0.7238667  0.4434101
##   321  0.7189333  0.4330787
##   341  0.7136000  0.4215865
##   361  0.7122667  0.4183400
##   381  0.7098667  0.4131761
##   401  0.7090667  0.4112403
##   421  0.7058667  0.4043164
##   441  0.7001333  0.3920207
##   461  0.6952000  0.3811374
##   481  0.6872000  0.3636126
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 21.

```

## Critère AUC

```
> donnees1 <- donnees
> names(donnees1)[3] <- c("Class")
> levels(donnees1$Class) <- c("G0", "G1")
> ctrl11 <- trainControl(method="LGOCV", number=1, index=list(1:1500),
+                       classProbs=TRUE, summary=twoClassSummary)
> e4 <- train(Class~., data=donnees1, method="knn", trControl=ctrl11,
+            metric="ROC", tuneGrid=K_cand)
> e4
## k-Nearest Neighbors
##
## 2000 samples
## 2 predictor
## 2 classes: 'G0', 'G1'
##
## No pre-processing
## Resampling: Repeated Train/Test Splits Estimated (1 reps, 75%)
## Summary of sample sizes: 1500
## Resampling results across tuning parameters:
##
## k   ROC      Sens      Spec
## 1   0.6190866 0.5983264 0.6398467
## 21  0.7171484 0.6903766 0.7432950
## 41  0.7229757 0.6861925 0.7547893
## 61  0.7200500 0.6945607 0.7394636
## 81  0.7255567 0.6945607 0.7432950
## 101 0.7319450 0.6903766 0.7356322
## 121 0.7382452 0.6945607 0.7356322
## 141 0.7353757 0.7029289 0.7318008
## 161 0.7308549 0.7029289 0.7318008
## 181 0.7351272 0.7029289 0.7318008
## 201 0.7340050 0.7029289 0.7356322
## 221 0.7324099 0.7071130 0.7279693
## 241 0.7349028 0.7071130 0.7279693
## 261 0.7365780 0.7071130 0.7356322
## 281 0.7349749 0.6987448 0.7279693
## 301 0.7356963 0.7029289 0.7241379
## 321 0.7341493 0.6861925 0.7318008
## 341 0.7343898 0.6527197 0.7356322
## 361 0.7306385 0.6527197 0.7356322
## 381 0.7301816 0.6359833 0.7394636
## 401 0.7270957 0.6276151 0.7356322
## 421 0.7255487 0.6317992 0.7356322
##
## 441 0.7258933 0.6192469 0.7471264
## 461 0.7220619 0.6150628 0.7471264
## 481 0.7236330 0.6108787 0.7432950
##
## ROC was used to select the optimal model using the largest value.
## The final value used for the model was k = 121.
```

## 9 Bibliographie

### Références

#### Bibliol

- [Besse, 2018] Besse, P. (2018). *Science des données - Apprentissage Statistique*. INSA - Toulouse. [http://www.math.univ-toulouse.fr/~besse/pub/Appren\\_stat.pdf](http://www.math.univ-toulouse.fr/~besse/pub/Appren_stat.pdf).
- [Bousquet et al., 2003] Bousquet, O., Boucheron, S., and Lugosi, G. (2003). *Introduction to Statistical Learning Theory*, chapter Advanced Lectures on Machine Learning. Springer.
- [Cléménçon et al., 2008] Cléménçon, S., Lugosi, G., and Vayatis, N. (2008). Ranking and empirical minimization of u-statistics. *The Annals of Statistics*, 36(2) :844–874.
- [Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer, second edition.

- [James et al., 2015] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2015). *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer.
- [Vapnik, 2000] Vapnik, V. (2000). *The Nature of Statistical Learning Theory*. Springer, second edition.
- [Wikistat, 2020a] Wikistat (2020a). Apprentissage machine — introduction. <http://wikistat.fr/pdf/st-m-Intro-ApprentStat.pdf>.
- [Wikistat, 2020b] Wikistat (2020b). Qualité de prévision et risque. <http://wikistat.fr/pdf/st-m-app-risque.pdf>.

## Deuxième partie

# Algorithmes linéaires

- *Rappel* : une fonction de prévision  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ .

### Fonction de prévision linéaire

Une fonction de prévision est dite *linéaire* si elle se met sous la forme

$$f(x) = f_\beta(x) = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

### Remarque

- Possibilité d'inclure des *effets non linéaires* :

$$f_\beta(x) = \beta_0 + \beta_{11}x_1 + \beta_{12}x_1^2 + \beta_{21}x_2 + \beta_{22}x_2^2 + \beta_{12}x_1x_2 + \beta_{31}x_3 + \beta_{32}\exp(x_3) \dots$$

- Variables *qualitatives* codées en indicatrices :

$$f_\beta(x) = \beta_0 + \beta_1 \mathbf{1}_{x_1=A} + \beta_2 \mathbf{1}_{x_1=B} + \beta_3 \mathbf{1}_{x_1=C} + \dots$$

muni d'une contrainte identifiante, par exemple  $\beta_1 = 0$ .

### Régression

- $Y$  à valeurs dans  $\mathbb{R}$ .
- On utilise souvent le terme *modèle linéaire* :

$$y_i = \beta_0 + \beta_1 x_{i1} + \dots + \beta_d x_{id} + \varepsilon_i$$

où les  $\varepsilon_i$  sont i.i.d tels que  $\mathbf{E}[\varepsilon_i] = 0$  et  $\mathbf{V}[\varepsilon_i] = \sigma^2$ .

- *Fonction de prévision* :

$$m_\beta(x) = \mathbf{E}[Y|X = x] = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

### Classification binaire

- $Y$  à valeurs dans  $\{0, 1\}$ .
- La classification s'effectue à partir de la probabilité

$$p(x) = \mathbf{P}(Y = 1|X = x).$$

- *Frontière entre les deux classes* :

$$\{x : p(x) = 1 - p(x)\} = \left\{ x : \log \frac{p(x)}{1 - p(x)} = 0 \right\}.$$

- La frontière est *linéaire* si

$$\log \frac{p(x)}{1 - p(x)} = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

- $\implies$  *Modèle logistique*.

### Questions

1. Comment *calculer* (ou plutôt *estimer*) les  $\beta_j$  ?
  - MCO-vraisemblance
  - Approches régularisées  $\implies$  ridge-lasso...
  - Machines à support vecteur (SVM).
2. Comment *choisir* la combinaison linéaire ?
  - Sélection de variables
  - Régression sur composantes  $\implies$  PCR-PLS...
  - Transformation de variables  $\implies$  résidus partiels, modèle additifs...

# 1 Estimation par moindres carrés

## Minimiser les erreurs

- Les *données* :  $(x_1, y_1), \dots, (x_n, y_n)$  à valeurs dans  $\mathbb{R}^d \times \mathbb{R}$ .
- Le *modèle*

$$y_i = \beta_0 + \beta_1 x_{i1} + \dots + \beta_d x_{id} + \varepsilon_i$$

- $\varepsilon_i$  représente l'écart (ou l'erreur) entre la prévision du modèle  $\beta$  et la valeur observée.

## Idée

Choisir  $\beta$  de manière à **minimiser ces erreurs**.

## Estimateurs des moindres carrés

### Définition

On appelle *critère des moindres carrés ordinaires* ou *somme des carrés résiduelles* la fonction de  $\beta$  :

$$\text{SCR}(\beta) = \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_{i1} + \dots + \beta_d x_{id}))^2 = \|\mathbb{Y} - \mathbb{X}\beta\|^2$$

avec

$$\mathbb{Y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \quad \text{et} \quad \mathbb{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1d} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \dots & x_{nd} \end{pmatrix}.$$

### Propriété

Si  $\mathbb{X}$  est de plein rang alors l'**estimateur des MCO**  $\hat{\beta} = (\mathbb{X}^t \mathbb{X})^{-1} \mathbb{X}^t \mathbb{Y}$  minimise  $\text{SCR}(\beta)$ .

## Exemple

- *Données* Hitters, 263 individus, 20 variables

```
> Hitters %>% select(c(1:5,19)) %>% head()
##   AtBat Hits HmRun Runs RBI Salary
## 1   315   81     7   24  38  475.0
## 2   479  130    18   66  72  480.0
## 3   496  141    20   65  78  500.0
## 4   321   87    10   39  42   91.5
## 5   594  169     4   74  51  750.0
## 6   185   37     1   23   8   70.0
```

- *Problème* : Expliquer/prédire le salaire (Salary) par les autres variables.
- Calcul des **estimateurs MCO** avec **lm** :

```
> mod <- lm(Salary~.,data=Hitters)
> coef(mod)[1:5]
## (Intercept)      AtBat      Hits      HmRun      Runs
## 163.103588 -1.979873  7.500768  4.330883 -2.376210
```

- **Prévision** du salaire de nouveaux individus

```
> xnew %>% select(1:5)
##   AtBat Hits HmRun Runs RBI
## 1   585  139   31   93  94
```

avec **predict** :

```
> predict(mod,newdata=xnew)
##           1
## 1129.376
```

## Modèle gaussien

- En *supposant* de plus que les erreurs  $\varepsilon_i$  suivent une **loi Gaussienne**, on obtient la loi des estimateurs

$$\frac{\widehat{\beta}_j - \beta_j}{\widehat{\sigma}_{\widehat{\beta}_j}} \sim \mathcal{T}_{n-(d+1)}.$$

- On en déduit des *procédures de test* :

```
> broom::tidy(mod) %>% head()
## # A tibble: 6 x 5
##   term      estimate std.error statistic p.value
##   <chr>      <dbl>    <dbl>    <dbl>  <dbl>
## 1 (Intercept)  163.      90.8      1.80  0.0736
## 2 AtBat       -1.98     0.634    -3.12  0.00201
## 3 Hits        7.50     2.38     3.15  0.00181
## 4 HmRun        4.33     6.20     0.698  0.486
## 5 Runs       -2.38     2.98    -0.797  0.426
## 6 RBI        -1.04     2.60    -0.402  0.688
```

- Ainsi que des *intervalles de confiance* pour les **paramètres** :

```
> confint(mod) %>% head()
##           2.5 %          97.5 %
## (Intercept) -15.709647 341.9168228
## AtBat       -3.228667  -0.7310792
## Hits        2.817562  12.1839734
## HmRun       -7.884569  16.5463352
## Runs       -8.247625   3.4952055
## RBI        -6.168102   4.0781779
```

- ou pour les **prévisions** :

```
> predict(mod, newdata=xnew, interval="confidence")
##           fit          lwr          upr
## 1 1129.376  889.2244 1369.528
```

## Cas du modèle logistique

- Toutes ces notions se généralisent (assez) rapidement au *modèle logistique*

$$\log \frac{p_\beta(x)}{1 - p_\beta(x)} = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

- Le critère des MCO est remplacé par la *log-vraisemblance* (à maximiser) :

$$\mathcal{L}(y_1, \dots, y_n; \beta) = \sum_{i=1}^n [y_i x_i^t \beta - \log(1 + \exp(x_i^t \beta))].$$

- Pas de solution explicite mais de **(bons) algorithmes qui convergent vers le max.**

## Exemple

- On considère les données **SAheart** :

```
> head(SAheart)
##   sbp tobacco  ldl adiposity famhist typea obesity alcohol age chd
## 1  160   12.00  5.73   23.11 Present    49   25.30   97.20  52  1
## 2  144    0.01  4.41   28.61 Absent    55   28.87    2.06  63  1
## 3  118    0.08  3.48   32.28 Present    52   29.14    3.81  46  0
## 4  170    7.50  6.41   38.03 Present    51   31.99   24.26  58  1
## 5  134   13.60  3.50   27.78 Present    60   25.99   57.34  49  1
## 6  132    6.20  6.47   36.21 Present    62   30.77   14.14  45  0
```

- *Problème* : expliquer/prédire la variable binaire `chd` par les autres variables.
- On obtient les estimateurs avec **glm**

```

> logit <- glm(chd~.,data=SAheart,family="binomial")
> broom::tidy(logit)
## # A tibble: 10 x 5
##   term                estimate std.error statistic    p.value
##   <chr>                <dbl>    <dbl>    <dbl>    <dbl>
## 1 (Intercept)        -6.15     1.31     -4.70  0.0000258
## 2 sbp                 0.00650  0.00573    1.14  0.256
## 3 tobacco            0.0794   0.0266    2.98  0.00285
## 4 ldl                 0.174    0.0597    2.92  0.00355
## 5 adiposity          0.0186   0.0293    0.635 0.526
## 6 famhistPresent     0.925    0.228     4.06  0.0000490
## 7 typea              0.0396   0.0123    3.21  0.00131
## 8 obesity            -0.0629  0.0442   -1.42  0.155
## 9 alcohol            0.000122 0.00448  0.0271 0.978
## 10 age               0.0452   0.0121    3.73  0.000193

```

— Les prévisions de la probabilité de l'évènement  $\{chd=1\}$  pour de nouveaux individus

```

> xnew
##   sbp tobacco ldl adiposity famhist typea obesity alcohol age
## 1 146         0 6.62   25.69 Absent   60   28.07   8.23  63

```

— s'obtiennent avec `predict` :

```

> predict(logit,newdata=xnew,type="response")
##           1
## 0.4719671

```

## Conclusion

### Remarque

La qualité de ces modèles (et donc des prévisions) reposent sur deux postulats :

1. le *modèle est bon* :  $Y$  s'explique bien par une combinaison linéaire des  $X$  ;
2. les *estimateurs sont bons* : ils possèdent de bonnes propriétés statistiques.

— La qualité du modèle est toujours *difficile à vérifier*  $\implies$  ajouter d'autres effets dans la combinaison linéaire (quadratique, interactions...).

— On en sait plus sur la *performance des estimateurs* :

1. *Trop de variables*  $\implies$   $\nearrow$  de la variance (sur-ajustement).
2. *Colinéarités*  $\implies$   $\nearrow$  de la variance (sur-ajustement).

## 2 Sélection de variables

— Une approche naturelle pour répondre aux 2 problèmes évoqués précédemment est de *sélectionner des variables explicatives* parmi  $\{X_1, \dots, X_d\}$ .

### Idée

Supprimer les variables

- qui *n'expliquent pas*  $Y$ .
- dont l'effet est *déjà expliqué* par d'autres variables

$\implies$  ce n'est pas parce qu'une variable n'est *pas sélectionnée* qu'elle n'est *pas liée* à  $Y$  !

### Best subset selection

- $d$  variables explicatives  $\implies$   $2^d$  *modèles concurrents*.
- *Idée* : *construire* les  $2^d$  modèles et les *comparer*.

### Algorithme BSS

**Entrée** : un critère de choix de modèle (AIC, BIC...).

Pour  $j = 0, \dots, d$  :

1. Construire les  $\binom{d}{j}$  modèles linéaires à  $j$  variables ;
2. Choisir parmi ces modèles celui qui a la plus petite SCR. On note  $\mathcal{M}_j$  le modèle sélectionné.

**Retourner** : le meilleur modèle parmi  $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_d$  au sens du critère de choix de modèle.

## Exemples de critères (voir [Cornillon et al., 2019])

- AIC : Akaike Information Criterion

$$-2\mathcal{L}_n(\hat{\beta}) + 2d.$$

- BIC : Bayesian Information Criterion

$$-2\mathcal{L}_n(\hat{\beta}) + \log(n)d.$$

- $R^2$  ajusté :

$$R_a^2 = 1 - \frac{n-1}{n-d+1}(1-R^2) \quad \text{où} \quad R^2 = \frac{SSR}{SST} = \frac{\|\hat{Y} - \bar{Y}\mathbf{1}\|^2}{\|Y - \bar{Y}\mathbf{1}\|^2}.$$

- $C_p$  de Mallow :

$$C_p = \frac{1}{n} \left( \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 + 2d\hat{\sigma}^2 \right).$$

## Ajustement/complexité

- Ces critères sont constitués de deux parties :
  1. une qui mesure la *qualité d'ajustement* du modèle ;
  2. une autre qui mesure sa *complexité*.

### Exemple AIC

- $-2\mathcal{L}_n(\hat{\beta})$  mesure l'ajustement ;
- $2p$  mesure la complexité.

⇒ l'idée est de choisir un modèle de *complexité minimale* qui *ajuste bien* les données.

## Le coin R

- On peut utiliser les packages `leaps` et `bestglm`.
- On propose de présenter `bestglm` qui fait appel à `leaps` pour la régression et fonctionne également pour le *modèle logistique*.

```
> Hitters1 <- Hitters[,c(1:18,20,19)]
> sel.var <- bestglm(Hitters1)
> sel.var$Subsets %>% select(c(1:5,22)) %>% head()
##   (Intercept) AtBat Hits HmRun Runs BIC
## 0      TRUE FALSE FALSE FALSE FALSE 3213.768
## 1      TRUE FALSE FALSE FALSE FALSE 3117.350
## 2      TRUE FALSE TRUE  FALSE FALSE 3079.270
## 3      TRUE FALSE TRUE  FALSE FALSE 3072.569
## 4      TRUE FALSE TRUE  FALSE FALSE 3066.387
## 5      TRUE TRUE  TRUE  FALSE FALSE 3064.125
```

- On obtient le *modèle sélectionné* avec :

```
> sel.var$BestModel %>% broom::tidy()
## # A tibble: 7 x 5
##   term          estimate std.error statistic p.value
##   <chr>          <dbl>     <dbl>     <dbl>   <dbl>
## 1 (Intercept)    91.5      65.0       1.41 1.60e- 1
## 2 AtBat         -1.87     0.527     -3.54 4.70e- 4
## 3 Hits           7.60     1.66       4.57 7.46e- 6
## 4 Walks          3.70     1.21       3.06 2.49e- 3
## 5 CRBI           0.643    0.0644     9.98 5.05e-20
## 6 DivisionW    -123.     39.8      -3.09 2.24e- 3
## 7 PutOuts       0.264    0.0748     3.53 4.84e- 4
```

## Remarque

- L'approche *exhaustive* peut se révéler coûteuse en temps de calcul lorsque  $d > 50$ .
- On utilise généralement des méthodes *pas à pas* dans ce cas.

## Pas à pas ascendant

### Algorithme forward

**Entrée :** un critère de choix de modèle (AIC, BIC...)

1. Construire  $\mathcal{M}_0$  le modèle linéaire qui contient uniquement la constante ;
2. Pour  $j = 0, \dots, d - 1$  :
  - (a) Construire les  $d - j$  modèles linéaires en ajoutant une variable, parmi les variables non utilisées, à  $\mathcal{M}_j$  ;
  - (b) Choisir, parmi ces  $d - j$  modèles, celui qui minimise la SCR  $\rightarrow \mathcal{M}_{j+1}$ .

**Retourner :** le meilleur modèle parmi  $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_d$  au sens du critère de choix de modèle.

### *Le coin R*

Utiliser `method=forward` dans `bestglm`.

## Pas à pas descendant

### Algorithme backward

**Entrée :** un critère de choix de modèle (AIC, BIC...)

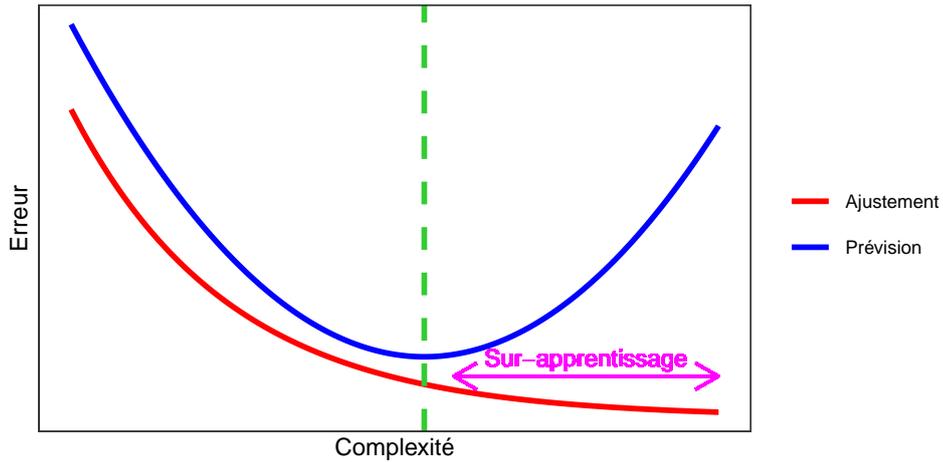
1. Construire  $\mathcal{M}_d$  le modèle linéaire complet (avec toutes les variables explicatives) ;
2. Pour  $j = d, \dots, 1$  :
  - (a) Construire les  $j$  modèles linéaires en supprimant une variable, parmi les variables non utilisées, à  $\mathcal{M}_j$  ;
  - (b) Choisir, parmi ces  $j$  modèles, celui qui minimise la SCR  $\rightarrow \mathcal{M}_{j-1}$ .

**Retourner :** le meilleur modèle parmi  $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_d$  au sens du critère de choix de modèle.

### *Le coin R*

Utiliser `method=backward` dans `bestglm`.

### 3 Régularisation



#### Complexité linéaire

Le nombre de variables est une mesure de la complexité des algorithmes linéaires.

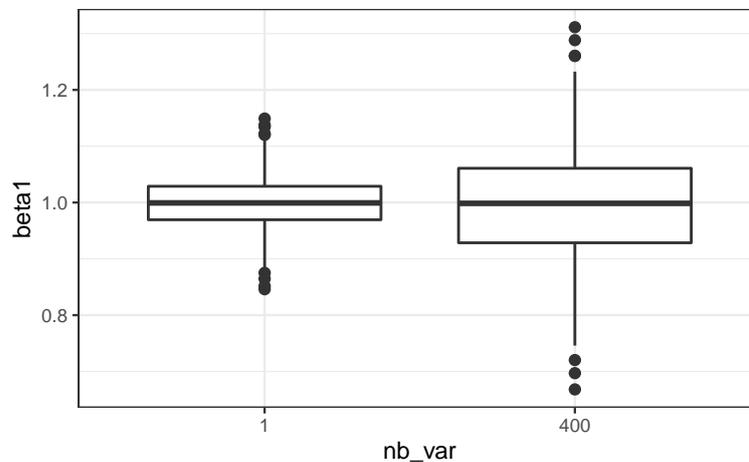
#### Illustration numérique

- On génère des données  $(x_i, y_i), i = 1, \dots, 500$  selon le modèle

$$y_i = 1x_{i1} + 0x_{i2} + \dots + 0x_{iq} + \varepsilon_i$$

où  $x_1, \dots, x_q, \varepsilon$  sont i.i.d. de loi  $\mathcal{N}(0, 1)$ .

- Seule  $X_1$  est explicative, les  $q - 1$  autres variables peuvent être vues comme du bruit.
- On calcule l'estimateur de MCO de  $\beta_1$  sur 1000 répétitions. On trace les boxplot de ces estimateurs pour  $q = 0$  et  $q = 400$ .



#### Conclusion

Plus de variance (donc moins de précision) lorsque le nombre de variables inutiles augmente.

- Lorsque le nombre de variables  $d$  est grand, les estimateurs des moindres carrés du modèle linéaire

$$Y = \beta_1 X_1 + \dots + \beta_d X_d + \varepsilon$$

possèdent généralement une grande variance.

#### Idée des méthodes pénalisés

- Contraindre la valeur des estimateurs des moindres carrés de manière à réduire la variance (quitte à augmenter un peu le biais).

- **Comment ?** En imposant une **contrainte** sur la valeur des estimateurs des moindres carrés :

$$\hat{\beta}^{pen} = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^n \left( y_i - \sum_{j=1}^d x_{ij} \beta_j \right)^2$$

sous la contrainte  $\|\beta\|_1 \leq t$ .

### Questions

- Quelle *norme* utiliser pour la contrainte ?
- *Existence/unicité* des estimateurs ? *Solutions explicites* du problème d'optimisation ?
- Comment *choisir t* ?
  - $t$  petit  $\implies$  estimateurs **contraints** (proche de 0) ;
  - $t$  grand  $\implies$  estimateurs des **moindres carrés** (non pénalisés).

### Remarque/Rappel

- Cas similaire déjà vu pour LDA.
- *Modèle standard LDA* :  $\mathcal{L}(X|Y = k) = \mathcal{N}(\mu_k, \Sigma)$ .
- $\mu_k$  et  $\Sigma$  sont généralement estimés pour les *moyennes et matrice de covariance empiriques*.

### LDA régularisée

On **régularise** la matrice de covariance en **augmentant les valeurs de la diagonale**

$$(1 - \gamma)\hat{\Sigma} + \gamma\hat{\sigma}^2 I_p.$$

## 3.1 Régression ridge

- La *régression ridge* consiste à minimiser le critère des moindres carrés pénalisé par la norme 2 des coefficients.

### Définition

1. Les *estimateurs ridge*  $\hat{\beta}^R$  s'obtiennent en minimisant

$$\sum_{i=1}^n \left( y_i - \beta_0 - \sum_{j=1}^d x_{ij} \beta_j \right)^2 \quad \text{sous la contrainte} \quad \sum_{j=1}^d \beta_j^2 \leq t \quad (1)$$

2. ou de façon *équivalente*

$$\hat{\beta}^R = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^n \left( y_i - \beta_0 - \sum_{j=1}^d x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^d \beta_j^2 \right\}. \quad (2)$$

### Quelques remarques

- Les définitions (1) et (2) sont *équivalentes* dans le sens où pour tout  $t$  il existe un unique  $\lambda$  tels que les solutions aux deux problèmes d'optimisation *coïncident*.
- La *constante*  $\beta_0$  n'entre généralement *pas* dans la *pénalité*.
- L'estimateur *dépend* bien entendu du paramètre  $t$  (ou  $\lambda$ ) :  $\hat{\beta}^R = \hat{\beta}^R(t) = \hat{\beta}^R(\lambda)$ .
- Les variables explicatives sont le plus souvent *réduites* pour *éviter les problèmes d'échelle* dans la pénalité.

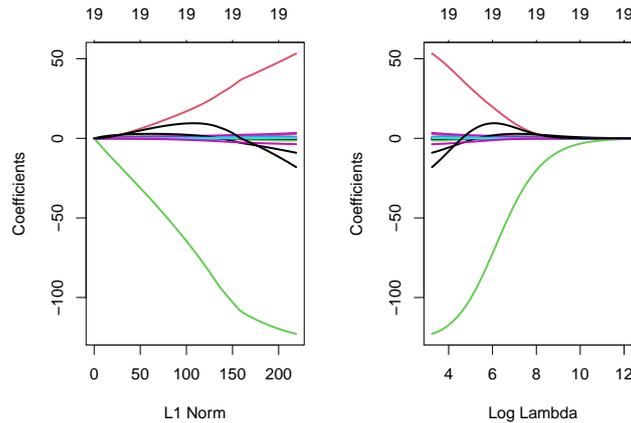
## Exemple avec les données Hitters

- Il existe *plusieurs fonctions et packages* qui permettent de faire de la régression pénalisée sur R. Nous présentons ici `glmnet`.
- `glmnet` n'accepte pas d'objet `formule`. Il faut spécifier la *matrice* des  $X$  et le *vecteur* des  $Y$  :

```
> Hitters.X <- model.matrix(Salary~.,data=Hitters)[,-1]
```

## Ridge avec glmnet

```
> library(glmnet)
> reg.ridge <- glmnet(Hitters.X,Hitters$Salary,alpha=0)
> par(mfrow=c(1,2))
> plot(reg.ridge,lwd=2)
> plot(reg.ridge,lwd=2,xvar="lambda")
```



## Propriétés des estimateurs ridge

### Propriétés

1. Lorsque les variables explicatives sont *centrée-réduites*, l'estimateur Ridge solution de (2) s'écrit

$$\hat{\beta}^R = \hat{\beta}^R(\lambda) = (\mathbb{X}^t \mathbb{X} + \lambda \mathbb{I})^{-1} \mathbb{X}^t \mathbb{Y}.$$

2. On déduit

$$\text{biais}(\hat{\beta}^R) = -\lambda(\mathbb{X}^t \mathbb{X} + \lambda \mathbb{I})^{-1} \beta$$

et

$$\mathbf{V}(\hat{\beta}^R) = \sigma^2(\mathbb{X}^t \mathbb{X} + \lambda \mathbb{I})^{-1} \mathbb{X}^t \mathbb{X} (\mathbb{X}^t \mathbb{X} + \lambda \mathbb{I})^{-1}.$$

### Commentaires

- Si  $\lambda = 0$ , on retrouve le biais et la variance de l'estimateur des **MCO**.
- $\lambda \nearrow \implies$  biais  $\nearrow$  et variance  $\searrow$  et réciproquement lorsque  $\lambda \searrow$ .

### Choix de $\lambda$

- Il est *crucial* : si  $\lambda \approx 0$  alors  $\hat{\beta}^R \approx \hat{\beta}^{MCO}$ , si  $\lambda$  "grand" alors  $\hat{\beta}^R \approx 0$ .
- Le choix de  $\lambda$  se fait le plus souvent de façon "classique" :
  1. **Estimation d'un critère** de choix de modèle pour toutes les valeurs de  $\lambda$  ;
  2. Choix du  $\lambda$  qui **minimise** le critère estimé.
- **Exemple** : la fonction `cv.glmnet` choisit la valeur de  $\lambda$  qui minimise *l'erreur quadratique moyenne*

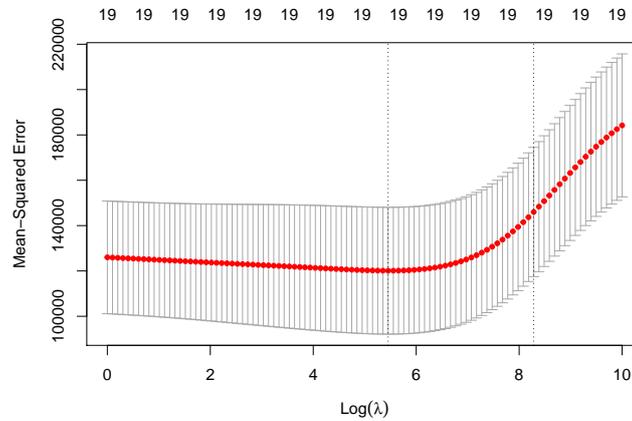
$$\mathbf{E}[(Y - m_{\hat{\beta}^R(\lambda)}(X))^2]$$

estimée par *validation croisée*.

```

> set.seed(321)
> reg.cvridge <- cv.glmnet(Hitters.X,Hitters$Salary,alpha=0,
+                          lambda=exp(seq(0,10,length=100)))
> bestlam <- reg.cvridge$lambda.min
> bestlam
## [1] 233.8186
> plot(reg.cvridge)

```



### 3.2 Régression Lasso

— La *régression lasso* consiste à minimiser le critère des moindres carrés pénalisé par la norme 1 des coefficients.

**Définition [Tibshirani, 1996]**

1. Les *estimateurs lasso*  $\hat{\beta}^L$  s'obtiennent en minimisant

$$\sum_{i=1}^n \left( Y_i - \beta_0 - \sum_{j=1}^d X_{ij} \beta_j \right)^2 \quad \text{sous la contrainte} \quad \sum_{j=1}^d |\beta_j| \leq t \quad (3)$$

2. ou de façon *équivalente*

$$\hat{\beta}^L = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^n \left( Y_i - \beta_0 - \sum_{j=1}^d X_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^d |\beta_j| \right\}. \quad (4)$$

#### Comparaison Ridge-Lasso

— Dans le cas où la matrice  $\mathbb{X}$  est *orthonormée*, on a une *écriture explicite* pour les estimateurs ridge et lasso.

#### Propriété

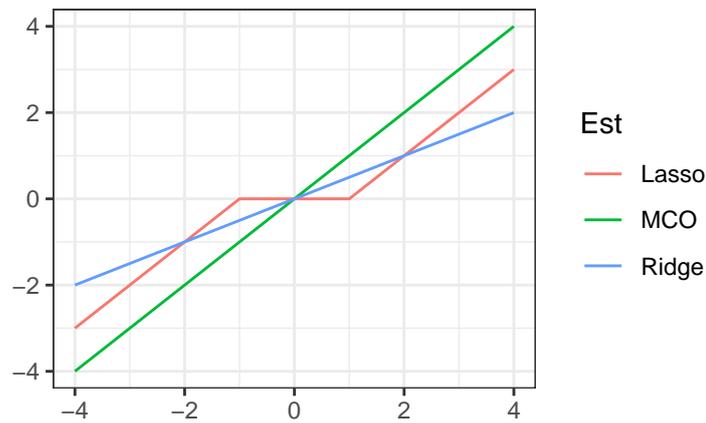
Si la matrice de design  $\mathbb{X}$  est orthonormée, alors

$$\hat{\beta}_j^R = \frac{\hat{\beta}_j}{1 + \lambda} \quad \text{et} \quad \hat{\beta}_j^L = \operatorname{signe}(\hat{\beta}_j) (|\hat{\beta}_j| - \lambda)_+$$

où  $\hat{\beta}_j$  est l'estimateur MCO de  $\beta_j$ .

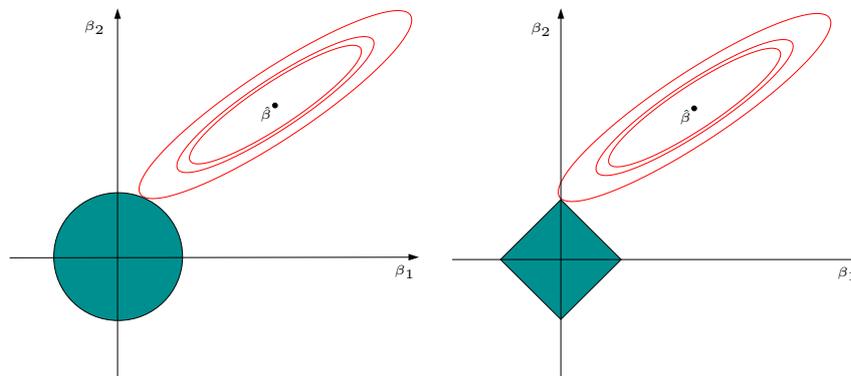
#### Commentaires

- Ridge "diminue" l'estimateur MCO de façon *proportionnelle* ;
- Lasso *translate et tronque* l'estimateur MCO (lorsque ce dernier est petit).



## Conclusion

Le lasso va avoir tendance à "*mettre*" des coefficients à 0 et donc à faire de la *sélection de variables*.



## Remarque

Ces approches reviennent (d'une certaine façon) à *projeter l'estimateur des MCO* sur les boules unités associées à

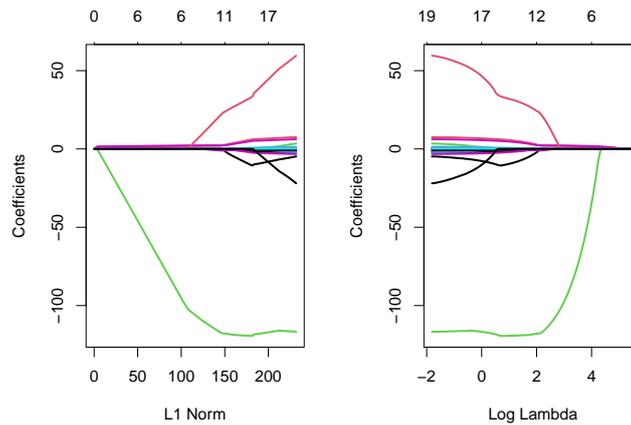
1. la norme 2 pour la régression *ridge* ;
2. la norme 1 pour le *lasso*.

## Quelques remarques

- Comme pour la régression ridge :
  - on préfère souvent *réduire la matrice de design* avant d'effectuer la régression lasso ;
  - Le choix de  $\lambda$  est *crucial* (il est le plus souvent sélectionné en minimisant un critère empirique).
  - $\lambda \nearrow \implies$  biais  $\nearrow$  et variance  $\searrow$  et réciproquement lorsque  $\lambda \searrow$ .
- **MAIS**, contrairement à ridge :  $\lambda \nearrow \implies$  *le nombre de coefficients nuls augmente* ([Bühlmann and van de Geer, 2011]).

## Le coin R

```
> reg.lasso <- glmnet(Hitters.X, Hitters$Salary, alpha=1)
> par(mfrow=c(1,2))
> plot(reg.lasso, lwd=2)
> plot(reg.lasso, lwd=2, xvar="lambda")
```

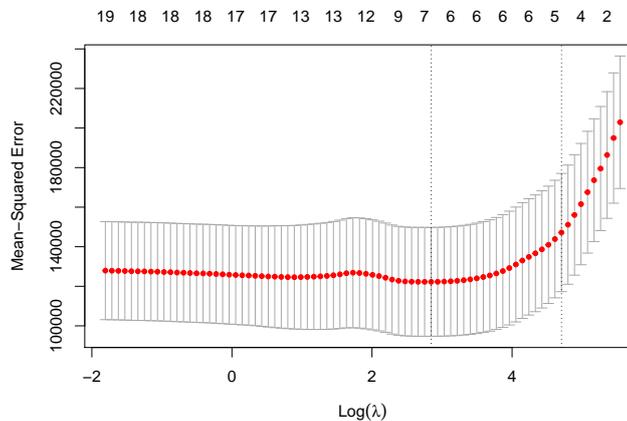


## Sélection de $\lambda$

```

> set.seed(321)
> reg.cvlasso <- cv.glmnet(Hitters.X,Hitters$Salary,alpha=1)
> bestlam <- reg.cvlasso$lambda.min
> bestlam
## [1] 17.19108
> plot(reg.cvlasso)

```



## Résolution numérique

- Il existe plusieurs façons de résoudre le problème numérique d'optimisation lasso (ou ridge).
- Un des plus utilisés est l'algorithme de descente de coordonnées [Hastie et al., 2015].
- On considère le problème lasso

$$\hat{\beta}^L = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^n \left( Y_i - \beta_0 - \sum_{j=1}^d X_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^d |\beta_j| \right\}$$

avec les variables explicatives centrées-réduites (pour simplifier).

## Descente de coordonnées

1. *Initialisation* :  $\hat{\beta}_0 = \bar{y}$ ,  $\hat{\beta}_j = \dots, j = 1, \dots, d$ .
2. Répéter jusqu'à convergence : Pour  $j = 1, \dots, d$  :
  - (a) Calculer les résidus partiels  $r_i^{(j)} = y_i - \sum_{k \neq j} x_{ik} \hat{\beta}_k$  ;
  - (b) Faire la régression simple des  $y_i$  contre  $r_i^{(j)} \implies \tilde{\beta}_j$  ;
  - (c) Mettre à jour  $\hat{\beta}_j = \operatorname{signe}(\tilde{\beta}_j)(|\tilde{\beta}_j| - \lambda)_+$
3. *Retourner* :  $\hat{\beta}_j, j = 1, \dots, d$ .

### 3.3 Variantes de ridge/lasso

#### Différentes pénalités

- Les approches *ridge* et *lasso* diffèrent uniquement au niveau de la **pénalité** ajoutée au critère des moindres carrés.
- *Norme 2* pour *ridge* et *norme 1* pour le *lasso*.
- Il existe tout un tas d'*autres stratégies* de pénalisations.
- Nous en présentons quelques unes dans cette partie.
- On pourra consulter [Hastie et al., 2015] pour plus de détails.

#### Elastic net

- [Zou and Hastie, 2005] ont proposé de *combinaison des approches ridge et lasso* en proposant une pénalité (appelée *elastic net*) de la forme

$$\lambda \sum_{j=1}^d ((1 - \alpha)\beta_j^2 + \alpha|\beta_j|)$$

où  $\alpha \in [0, 1]$ .

- Le paramètre  $\alpha$  définit le **compromis ridge/lasso** :
  - $\alpha = 1 \implies$  Lasso ;
  - $\alpha = 0 \implies$  Ridge ;
  - Ce paramètre correspond (évidemment) à l'argument `alpha` de la fonction `glmnet`.
- *Avantage* : on a plus de flexibilité car la pénalité elastic net propose une gamme de modèles beaucoup plus large que lasso et ridge ;
- *Inconvénient* : en plus du  $\lambda$  il faut *aussi sélectionner le  $\alpha$*  !

#### Group Lasso

- Dans certaines applications, les variables *explicatives* appartiennent à des *groupes de variables* prédéfinis.
- Nécessité de "*shrinker*" ou *sélectionner les variables par groupe*.

#### Exemple : variables qualitatives

- 2 variables explicatives qualitatives  $X_1$  et  $X_2$  et une variable explicative continue  $X_3$ .
- Le *modèle* s'écrit

$$Y = \beta_0 + \beta_1 \mathbf{1}_{X_1=A} + \beta_2 \mathbf{1}_{X_1=B} + \beta_3 \mathbf{1}_{X_1=C} \\ + \beta_4 \mathbf{1}_{X_2=D} + \beta_5 \mathbf{1}_{X_2=E} + \beta_6 \mathbf{1}_{X_2=F} + \beta_7 \mathbf{1}_{X_2=G} + \beta_8 X_3 + \varepsilon$$

muni des contraintes  $\beta_1 = \beta_4 = 0$ .

- 3 groupes :  $\mathbf{X}_1 = (\mathbf{1}_{X_1=B}, \mathbf{1}_{X_1=C})$ ,  $\mathbf{X}_2 = (\mathbf{1}_{X_2=E}, \mathbf{1}_{X_2=F}, \mathbf{1}_{X_2=G})$  et  $\mathbf{X}_3 = X_3$ .

#### Définition

En présence de  $d$  variables réparties en  $L$  groupes  $\mathbf{X}_1, \dots, \mathbf{X}_L$  de cardinal  $d_1, \dots, d_L$ . On note  $\beta_\ell, \ell = 1, \dots, L$  le vecteur des coefficients associé au groupe  $\mathbf{X}_\ell$ . Les *estimateurs group-lasso* s'obtiennent en minimisant le critère

$$\sum_{i=1}^n \left( y_i - \beta_0 - \sum_{\ell=1}^L \mathbf{X}_{i\ell} \beta_\ell \right)^2 + \lambda \sum_{\ell=1}^L \sqrt{d_\ell} \|\beta_\ell\|_2$$

#### Remarque

Puisque  $\|\beta_\ell\|_2 = 0$  ssi  $\beta_{\ell 1} = \dots = \beta_{\ell d_\ell} = 0$ , cette procédure encourage la **mise à zéro** des coefficients d'un **même groupe**.

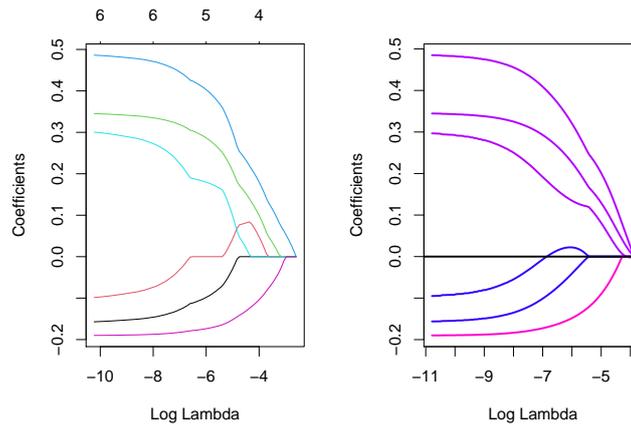
#### Le coin R

- La fonction `gglasso` du package `gglasso` permet de faire du *groupe lasso* sur R.

```

> summary(donnees)
##      X1          X2          X3          Y
## Length:200    Length:200    Min.   :0.009496  Min.   :-3.23315
## Class :character Class :character 1st Qu.:0.237935 1st Qu.: -0.50404
## Mode  :character Mode  :character Median :0.485563 Median : 0.16759
##                                     Mean  :0.483286 Mean  : 0.09792
##                                     3rd Qu.:0.734949 3rd Qu.: 0.66918
##                                     Max.  :0.998741 Max.  : 3.04377
> D <- model.matrix(Y~.,data=donnees)[,-1]
> model <- glmnet(D,Y,alpha=1)
> groupe <- c(1,1,2,2,2,3)
> library(gglasso)
> model1 <- gglasso(D,Y,group=groupe)
> plot(model1)

```



### Remarque

Les coefficients **s'annulent par groupe** lorsque  $\lambda$  augmente (graphe de droite).

### Sparse group lasso

- La **norme 2** de la pénalité group-lasso implique que, généralement, tous les coefficients d'un groupe sont **tous nuls** ou **tous non nuls**.
- Dans certains cas, il peut être intéressant de mettre de la **sparsité** dans les groupes aussi. Comment ?
- En ajoutant la **norme 1** dans la pénalité.

#### Pénalité sparse group lasso

$$\lambda \sum_{\ell=1}^L [(1 - \alpha) \|\beta_{\ell}\|_2 + \alpha \|\beta_{\ell}\|_1].$$

- Sur R : package SGL.

### Fused lasso

- Utile pour prendre en compte la **spatialité des données**.
- **Idée** : deux coefficients successifs doivent être proches.

#### Pénalité fused lasso

$$\lambda_1 \sum_{j=1}^d |\beta_j| + \lambda_2 \sum_{j=2}^d |\beta_{j+1} - \beta_j|$$

qui peut se re-paramétriser en

$$\lambda \sum_{j=2}^d |\beta_{j+1} - \beta_j|.$$

- Sur R : package genlasso.

### 3.4 Discrimination binaire

#### Discrimination binaire

- Les méthodes *ridge et lasso* ont été présentées dans un cadre de régression linéaire.
- Ces techniques d'adaptent directement à la *régression logistique*  $\mathcal{Y} = \{-1, 1\}$ .
- Les *pénalités* sont *identiques*.
- **Seul changement** : le critère moindre carré est remplacé par la déviance  $\implies$  ce qui revient à *minimiser l'opposé de la vraisemblance plus la pénalité*.

#### Lasso et Ridge pour la logistique

##### Définition

On note  $\tilde{y}_i = (y_i + 1)/2$ .

- On appelle *estimateur ridge* en régression logistique l'estimateur

$$\hat{\beta}^R = \operatorname{argmin}_{\beta} \left\{ - \sum_{i=1}^n (\tilde{y}_i x_i^t \beta - \log(1 + \exp(x_i^t \beta))) + \lambda \sum_{j=1}^d \beta_j^2 \right\}.$$

- On appelle *estimateur lasso* en régression logistique l'estimateur

$$\hat{\beta}^L = \operatorname{argmin}_{\beta} \left\{ - \sum_{i=1}^n (\tilde{y}_i x_i^t \beta - \log(1 + \exp(x_i^t \beta))) + \lambda \sum_{j=1}^d |\beta_j| \right\}.$$

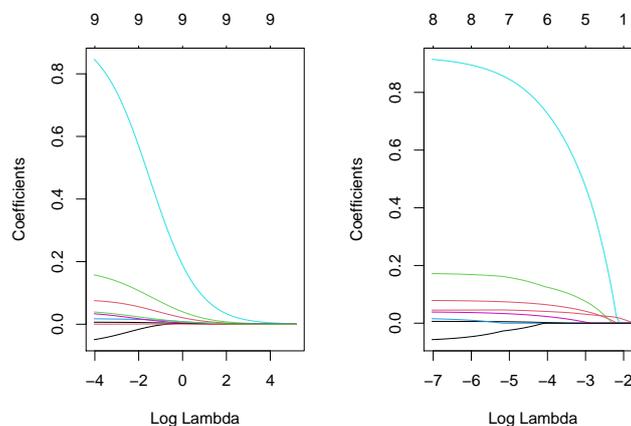
#### Le coin R

- Pour faire du ridge ou lasso en logistique, il suffit d'ajouter l'argument `family=binomial` dans `glmnet`.
- *Tout reste identique* pour le reste (tracé du chemin des coefficients, choix du  $\lambda \dots$ ).
- *Exemple* : données SAheart

```
> head(SAheart)
##   sbp tobacco  ldl adiposity famhist typea obesity alcohol age chd
## 1 160  12.00 5.73   23.11 Present   49  25.30  97.20 52  1
## 2 144   0.01 4.41   28.61 Absent   55  28.87   2.06 63  1
## 3 118   0.08 3.48   32.28 Present  52  29.14   3.81 46  0
## 4 170   7.50 6.41   38.03 Present  51  31.99  24.26 58  1
## 5 134  13.60 3.50   27.78 Present  60  25.99  57.34 49  1
## 6 132   6.20 6.47   36.21 Present  62  30.77  14.14 45  0
```

- On obtient les *chemins de régularisation ridge et lasso* avec les commandes suivantes :

```
> SAheart.X <- model.matrix(chd~., data=SAheart)
> log.ridge <- glmnet(SAheart.X, SAheart$chd, family="binomial", alpha=0)
> log.lasso <- glmnet(SAheart.X, SAheart$chd, family="binomial", alpha=1)
> plot(log.ridge, xvar="lambda")
```



## 4 Support vector machine

### Cadre et notation

- *Discrimination binaire* :  $Y$  à valeurs dans  $\{-1, 1\}$  et  $X = (X_1, \dots, X_d)$  dans  $\mathbb{R}^d$ .
- Les extensions *multi-classes* et *régression* seront présentées à la fin de cette partie.

### Objectif

- Estimer la *fonction de score*  $S(x) = \mathbf{P}(Y = 1|X = x)$  ;
- En déduire une *règle de classification*  $g : \mathbb{R}^d \rightarrow \{-1, 1\}$ .

### Règles linéaires

- Elles consistent à *séparer* l'espace des  $X$  par un *hyperplan*.
- On classe ensuite 1 d'un coté de l'hyperplan, -1 de l'autre coté.

### Mathématiquement

- On cherche une combinaison linéaire des variables  $w_1x_1 + \dots + w_dx_d$ .
- *Règle associée* :

$$g(x) = \begin{cases} 1 & \text{si } w_1x_1 + \dots + w_dx_d \geq 0 \\ -1 & \text{sinon.} \end{cases}$$

### Exemple 1 : régression logistique

- *Modèle* :

$$\text{logit} \frac{p(x)}{1-p(x)} = \beta_0 + \beta_1x_1 + \dots + \beta_dx_d$$

où  $p(x) = \mathbf{P}(Y = 1|X = x)$ .

- *Règle de classification* :

$$g(x) = \begin{cases} 1 & \text{si } p(x) \geq 0.5 \\ -1 & \text{sinon.} \end{cases}$$

- équivalent à

$$g(x) = \begin{cases} 1 & \text{si } \beta_0 + \beta_1x_1 + \dots + \beta_dx_d \geq 0 \\ -1 & \text{sinon.} \end{cases}$$

### Exemple 2 : LDA

- *Modèle* :  $\mathcal{L}(X|Y = k) = \mathcal{N}(\mu_k, \Sigma)$ ,  $k = 0, 1$ .
- *Règle de classification* :

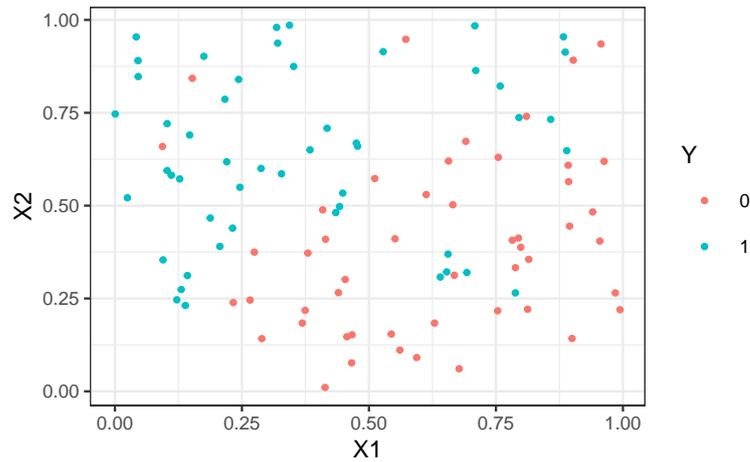
$$g(x) = \begin{cases} 1 & \text{si } p(x) \geq 0.5 \\ -1 & \text{sinon.} \end{cases}$$

- équivalent à

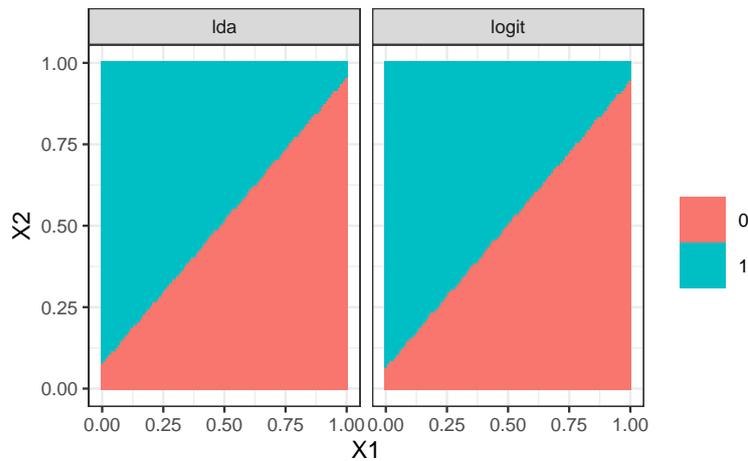
$$g(x) = \begin{cases} 1 & \text{si } c + x^t \Sigma^{-1}(\mu_1 - \mu_0) \geq 0 \\ -1 & \text{sinon.} \end{cases}$$

### Illustration avec $p = 2$

- On considère les données suivantes :



— On compare les prévisions **logistique** et **lda**.



### Remarque

On retrouve bien la *linéarité* (et la *proximité*) de ces deux méthodes.

— Ces approches linéaires s'obtiennent à partir d'un *modèle statistique*

— sur la loi de **Y sachant X** pour la logistique ;

— sur la loi de **X sachant Y** pour la discriminante linéaire.

— L'approche **SVM** repose sur le calcul direct du "**meilleur**" **hyperplan séparateur** qui sera déterminé à partir d'algorithmes d'optimisation.

## 4.1 SVM - cas séparable

### Bibliographie

En plus des documents cités précédemment, cette partie s'appuie sur les diapos de cours de

— *Magalie Fromont*, Apprentissage statistique, Université Rennes 2 ([Fromont, 2015]).

— *Jean-Philippe Vert*, *Support vector machines and applications in computational biology*, disponible à l'url <http://cbio.ensmp.fr/~jvert/svn/kernelcourse/slides/kernel2h/kernel2h.pdf>

### Remarque

Les aspects techniques ne seront pas présentés ici, on pourra en trouver dans la **partie 2.4 du tutoriel**.

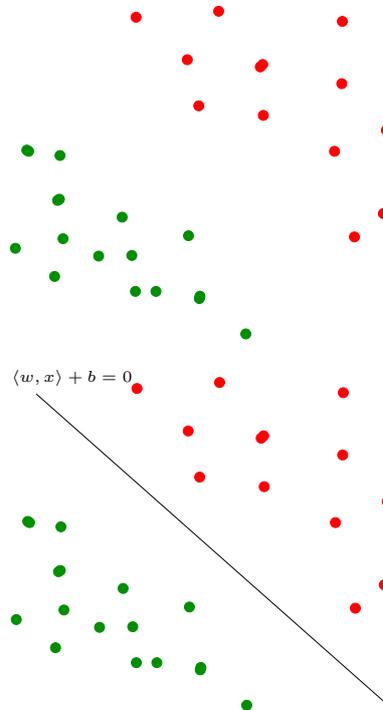
## Présentation

- L'approche SVM [Vapnik, 2000] peut être vue comme une *généralisation* de "recherche d'hyperplan optimal".

### Cas simple

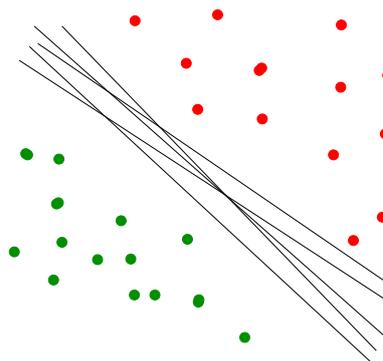
Les données  $(x_1, y_1), \dots, (x_n, y_n)$  sont dites *linéairement séparables* si il existe  $(w, b) \in \mathbb{R}^d \times \mathbb{R}$  tel que pour tout  $i$  :

- $y_i = 1$  si  $\langle w, x_i \rangle + b = w^t x_i + b > 0$  ;
- $y_i = -1$  si  $\langle w, x_i \rangle + b = w^t x_i + b < 0$ .



### Vocabulaire

- L'équation  $\langle w, x \rangle + b$  définit un *hyperplan séparateur* de vecteur normal  $w$ .
- La fonction  $\text{signe}(\langle w, x \rangle + b)$  est une règle de *discrimination* potentielle.

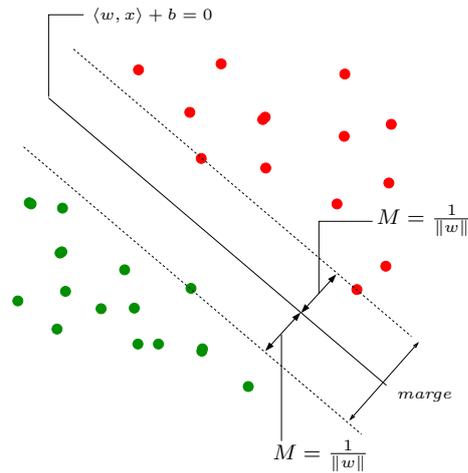


### Problème

Il existe une *infinité d'hyperplans séparateurs* donc une *infinité* de règles de discrimination potentielles.

### Solution

[Vapnik, 2000] propose de choisir l'hyperplan ayant la *marge maximale*.



## Le problème d'optimisation

- On veut trouver l'hyperplan de *marge maximale* qui *sépare* les groupes.

### Hyperplan séparateur optimal

Solution du problème *d'optimisation sous contrainte* :

- **Version 1** :

$$\max_{w, b, \|w\|=1} M$$

sous les contraintes  $y_i(w^t x_i + b) \geq M, i = 1, \dots, n.$

- **Version 2** :

$$\min_{w, b} \frac{1}{2} \|w\|^2$$

sous les contraintes  $y_i(w^t x_i + b) \geq 1, i = 1, \dots, n.$

## Solutions

- On obtient

$$w^* = \sum_{i=1}^n \alpha_i^* y_i x_i.$$

où les  $\alpha_i^*$  sont des constantes positives qui s'obtiennent en résolvant le *dual* du problème précédent.

- De plus,  $b^*$  s'obtient en résolvant

$$\alpha_i^* [y_i(x_i^t w + b) - 1] = 0$$

pour un  $\alpha_i^*$  non nul.

### Remarque

$w^*$  s'écrit comme une *combinaison linéaire* des  $x_i$ .

## Vecteurs supports

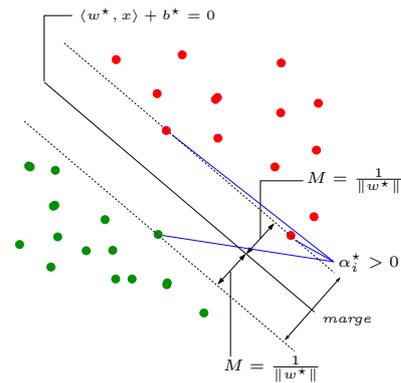
### Propriété (conditions KKT)

$$\alpha_i^* [y_i(x_i^t w^* + b) - 1] = 0, i = 1, \dots, n.$$

### Conséquence (importante)

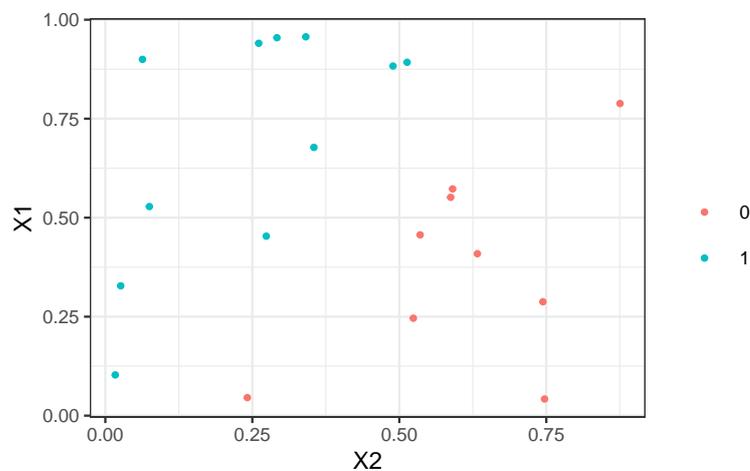
- Si  $\alpha_i^* \neq 0$  alors  $y_i(x_i^t w^* + b) = 1$  et  $x_i$  est *sur la marge*.
- $w^*$  se calcule *uniquement* à partir de ces points là.
- Ces points sont appelés les *vecteurs supports* de la SVM.

## Représentation



## Le coin R

- La fonction `svm` du package `e1071` permet d'ajuster des *SVM*.



```
> library(e1071)
> mod.svm <- svm(Y~., data=df, kernel="linear", cost=10000000000)
```

## La fonction svm

- Les vecteurs supports :

```
> mod.svm$index
## [1] 6 14 12
```

- `mod.svm$coefs` =  $\alpha_i^* y_i$  pour chaque vecteur support

```
> mod.svm$coefs
##           [,1]
## [1,]  1.898982
## [2,]  1.905497
## [3,] -3.804479
```

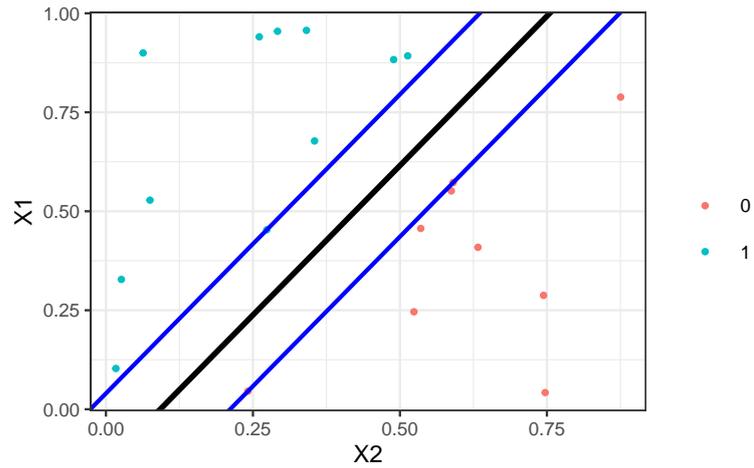
- On peut en déduire l'hyperplan séparateur

```
> w <- apply(mod.svm$coefs*df[mod.svm$index,2:3], 2, sum)
> b <- -mod.svm$rho
> w
##           X1           X2
## -0.5470382  0.5427583
> b
## [1] -0.4035113
```

On peut ainsi visualiser

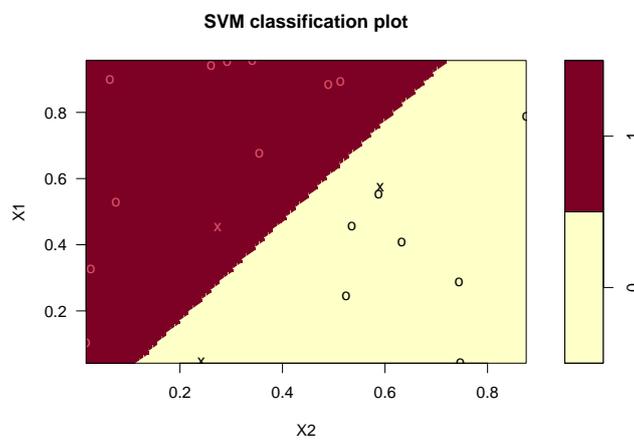
- les vecteurs supports ;
- l'hyperplan séparateur ;

— la marge.



— La fonction `plot` donne aussi une représentation de l'*hyperplan séparateur*.

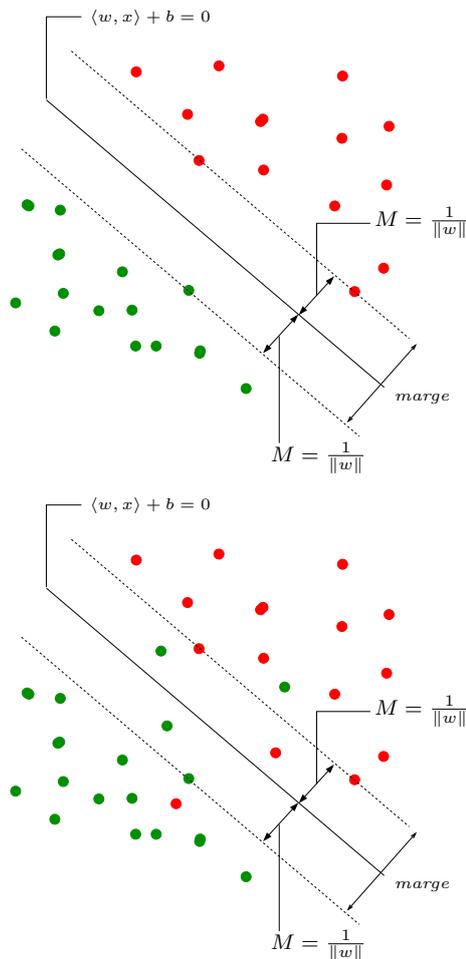
```
> plot(mod.svm,data=df,fill=TRUE,grid=100)
```



## 4.2 SVM : cas non séparable

### *Problème*

Dans la vraie vie, les données ne sont (quasiment) *jamais linéairement séparables*...



### Idée

Autoriser certains points

1. à être *bien classés* mais à l'*intérieur* de la marge;
2. et/ou à être *mal classés*.

### Slack variables

*Rappel : cas séparable*

$$\min_{w,b} \frac{1}{2} \|w\|^2$$

sous les contraintes  $y_i(w^t x_i + b) \geq 1, i = 1, \dots, n.$

- Les contraintes  $y_i(w^t x_i + b) \geq 1$  signifient que tous les points se trouvent en dehors de la frontière définie par la *marge*;
- **Cas non séparable** : le problème ci-dessus n'admet pas de solution!

### Variables ressorts

On introduit des *variables ressorts (slack variables)* positives  $\xi_1, \dots, \xi_n$  telles que  $y_i(w^t x_i + b) \geq 1 - \xi_i$ . 2 cas sont à distinguer :

1.  $\xi_i \in [0, 1] \implies$  bien classé mais *dans* la région définie par la *marge*;
2.  $\xi_i > 1 \implies$  *mal classé*.

- Bien entendu, on souhaite avoir le *maximum* de variables ressorts  $\xi_i$  *nulles*;
- Lorsque  $\xi_i > 0$ , on souhaite que  $\xi_i$  soit le *plus petit possible*.

### Cas non séparable : problème d'optimisation (primal)

- Il s'agit de minimiser en  $(w, b, \xi)$

$$\frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i$$

sous les contraintes  $\begin{cases} y_i(w^t x_i + b) \geq 1 - \xi_i \\ \xi_i \geq 0, i = 1, \dots, n. \end{cases}$

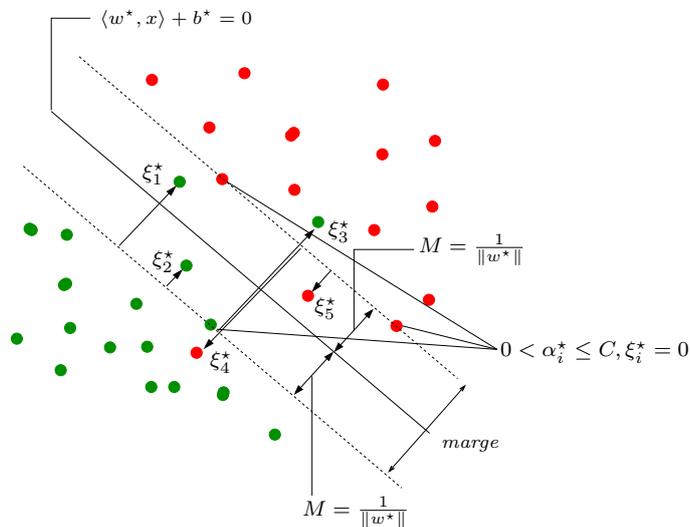
- $C > 0$  est un paramètre à calibrer (**paramètre de coût**).
- Le **cas séparable** correspond à  $C \rightarrow \infty$ .
- Les **solutions** de ce nouveau problème d'optimisation s'obtiennent de la **même façon** que dans le cas séparable (Lagrangien, problème dual...).
- L'**hyperplan optimal** est défini par

$$w^* = \sum_{i=1}^n \alpha_i^* y_i x_i$$

et  $b^*$  est solution de  $y_i(\langle w^*, x_i \rangle + b^*) = 1$  pour tout  $i$  tel que  $0 < \alpha_i^* < C$ .

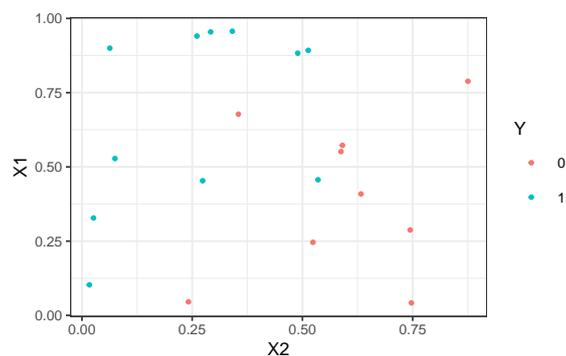
### Vecteurs supports

- Les  $x_i$  tels que  $\alpha_i^* > 0$  sont les vecteurs supports ;
- On en distingue 2 types :
  1. ceux **sur la frontière** définie par la marge :  $\xi_i^* = 0$  ;
  2. ceux **en dehors** :  $\xi_i^* > 0$  et  $\alpha_i^* = C$ .
- Les vecteurs **non supports** vérifient  $\alpha_i^* = 0$  et  $\xi_i^* = 0$ .



### Le coin R

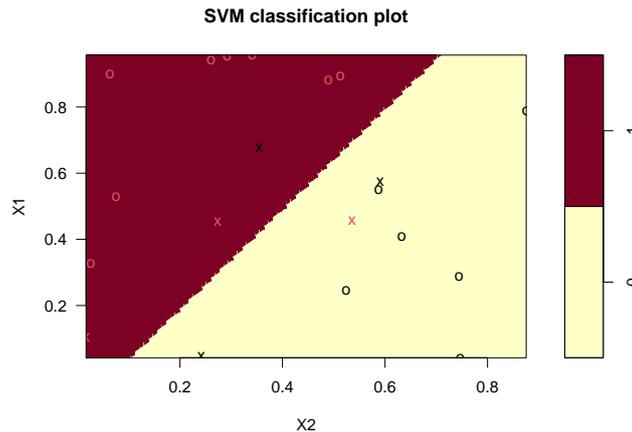
- On utilise la même fonction que dans le **cas séparable** (**svm** du package **e1071**) ;
- L'argument **cost** correspond à la **constante de régularisation C**.



```
> mod.svm1 <- svm(Y~.,data=df1,kernel="linear",cost=1000)
> mod.svm1$index
## [1] 6 13 14 10 12 15
```

### Visualisation de l'hyperplan séparateur

```
> plot(mod.svm1,data=df1,fill=TRUE,grid=100)
```



### Choix de $C$

Ce paramètre règle le *compromis biais/variance* de la svm :

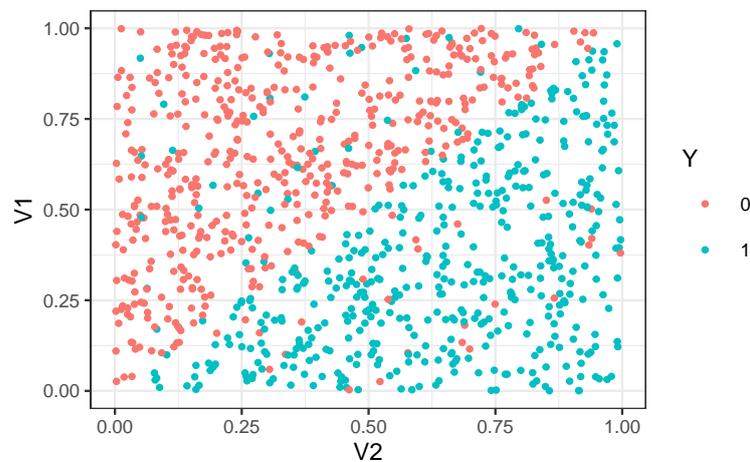
- $C \searrow$  : la marge est privilégiée et les  $\xi_i \nearrow \implies$  beaucoup d'observations dans la marge ou **mal classées** (et donc beaucoup de vecteurs supports).
- $C \nearrow \implies \xi_i \searrow$  donc moins d'observations mal classées  $\implies$  **meilleur ajustement** mais petite marge  $\implies$  risque de **surajustement**.

### Conclusion

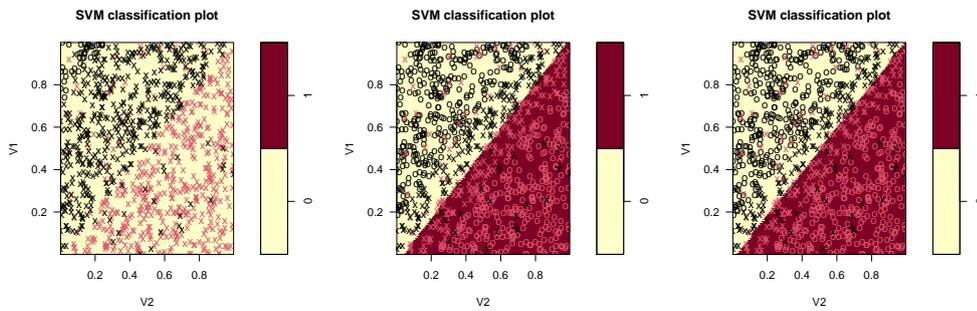
Il est donc très important de bien choisir ce paramètre.

- Le choix est souvent effectué de façon "classique" :
  1. On se donne un *critère de performance* (taux de mal classés par exemple) ;
  2. On *estime la valeur du critère* pour différentes valeurs de  $C$  ;
  3. On choisit la valeur de  $C$  pour laquelle le *critère estimé est minimum*.
- La fonction `tune.svm` permet de choisir  $C$  en estimant le taux de mal classés par *validation croisée*. On peut aussi (bien entendu) utiliser la fonction `train` du package `caret`.

### Un exemple



```
> mod.svm1 <- svm(Y~.,data=df3,kernel="linear",cost=0.000001)
> mod.svm2 <- svm(Y~.,data=df3,kernel="linear",cost=0.1)
> mod.svm3 <- svm(Y~.,data=df3,kernel="linear",cost=5)
```



```
> mod.svm1$nSV
## [1] 480 480
> mod.svm2$nSV
## [1] 190 190
> mod.svm3$nSV
## [1] 166 165
```

## Choix de $C$ avec tune

```
> set.seed(1234)
> tune.out <- tune(svm, Y ~ ., data=df3, kernel="linear",
+                 ranges=list(cost=c(0.001,0.01,1,10,100,1000)))
> summary(tune.out)
## Parameter tuning of 'svm':
## - sampling method: 10-fold cross validation
## - best parameters:
##   cost
##     1
##
## - best performance: 0.075
##
## - Detailed performance results:
##   cost error dispersion
## 1 1e-03 0.127 0.07087548
## 2 1e-02 0.080 0.03944053
## 3 1e+00 0.075 0.03439961
## 4 1e+01 0.075 0.03439961
## 5 1e+02 0.075 0.03439961
## 6 1e+03 0.075 0.03439961
```

```
> bestmod <- tune.out$best.model
> summary(bestmod)
##
## Call:
## best.tune(method = svm, train.x = Y ~ ., data = df3, ranges = list(cost = c(0.001,
## 0.01, 1, 10, 100, 1000)), kernel = "linear")
##
## Parameters:
##   SVM-Type: C-classification
##   SVM-Kernel: linear
##   cost: 1
##
## Number of Support Vectors: 336
##
## ( 168 168 )
##
##
## Number of Classes: 2
##
## Levels:
## 0 1
```

## Approche tune\_grid de tidymodels

1. Initialisation du workflow :

```

> library(tidymodels)
> tune_spec <-
+ svm_poly(cost=tune(),degree=1,scale_factor=1) %>%
+ set_mode("classification") %>%
+ set_engine("kernlab")
> svm_wf <- workflow() %>%
+ add_model(tune_spec) %>%
+ add_formula(Y ~ .)

```

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

```

> set.seed(12345)
> re_ech_cv <- vfold_cv(df3,v=10)
> grille_C <- tibble(cost=c(0.001,0.01,1,10,100,1000))

```

## 3. Calcul des erreurs :

```

> set.seed(123)
> svm_cv <- svm_wf %>%
+ tune_grid(
+   resamples = re_ech_cv,
+   grid = grille_C,
+   metrics=metric_set(accuracy))

```

## 4. Visualisation des résultats :

```

> svm_cv %>% collect_metrics() %>% dplyr::select(-7)
## # A tibble: 6 x 6
##   cost .metric .estimator mean n std_err
##   <dbl> <chr> <chr> <dbl> <int> <dbl>
## 1 0.001 accuracy binary 0.864 10 0.0163
## 2 0.01 accuracy binary 0.915 10 0.00778
## 3 1 accuracy binary 0.927 10 0.00700
## 4 10 accuracy binary 0.927 10 0.00775
## 5 100 accuracy binary 0.929 10 0.00809
## 6 1000 accuracy binary 0.929 10 0.00809

```

## 5. Sélection du meilleur paramètre

```

> best_C <- svm_cv %>% select_best()
> best_C
## # A tibble: 1 x 2
##   cost .config
##   <dbl> <chr>
## 1 100 Preprocessor1_Model5

```

## 6. Ajustement de l'algorithme final :

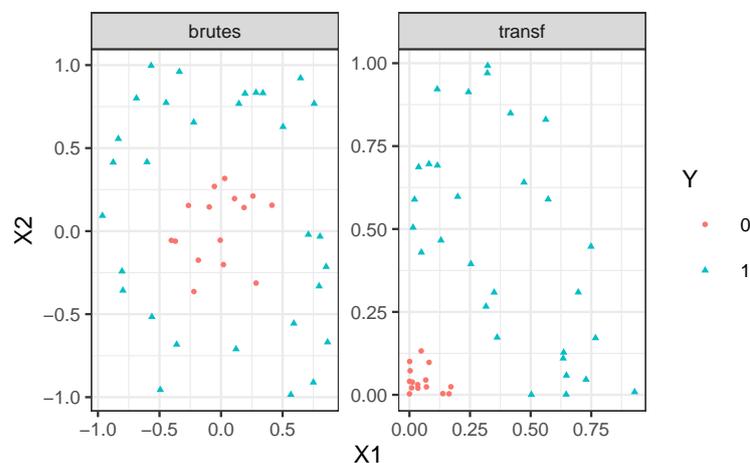
```

> final_svm <-
+ svm_wf %>%
+ finalize_workflow(best_C) %>%
+ fit(data = df3)

```

## 4.3 SVM non linéaire : astuce du noyau

— Les *solutions linéaires* ne sont pas toujours intéressantes.



### Idée

Trouver une transformation des données telle que les **données transformées** soient **linéairement séparables**.

### Noyau

**Définition 4.1.** Soit  $\Phi : \mathcal{X} \rightarrow \mathcal{H}$  une application qui va de l'espace des observations  $\mathcal{X}$  dans un Hilbert  $\mathcal{H}$ . Le **noyau**  $K$  entre  $x$  et  $x'$  associé à  $\Phi$  est le produit scalaire entre  $\Phi(x)$  et  $\Phi(x')$  :

$$K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R} \\ (x, x') \mapsto \langle \Phi(x), \Phi(x') \rangle_{\mathcal{H}}.$$

### Exemple

Si  $\mathcal{X} = \mathcal{H} = \mathbb{R}^2$  et  $\varphi(x_1, x_2) = (x_1^2, x_2^2)$  alors

$$K(x, x') = (x_1 x'_1)^2 + (x_2 x'_2)^2.$$

### L'astuce noyau

- L'**astuce** consiste donc à **envoyer les observations**  $x_i$  dans un espace de Hilbert  $\mathcal{H}$  appelé **espace de représentation** ou **feature space**...
- en espérant que les données  $(\Phi(x_1), y_1), \dots, (\Phi(x_n), y_n)$  soient (presque) **linéairement séparables** de manière à **appliquer une svm sur ces données transformées**.

### Remarque

1. Beaucoup d'**algorithmes linéaires** (en particulier les SVM) peuvent être appliqués sur  $\Phi(x)$  sans calculer explicitement  $\Phi$  ! Il suffit de pouvoir calculer le noyau  $K(x, x')$  ;
2. On n'a **pas besoin** de connaître l'espace  $\mathcal{H}$  ni l'application  $\Phi$ , il suffit de se **donner un noyau**  $K$  !

### SVM dans l'espace original

- Le **problème dual** consiste à maximiser

$$L_D(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{k=1}^n \alpha_i \alpha_k y_i y_k \langle x_i, x_k \rangle$$

$$\text{sous les contraintes } \begin{cases} 0 \leq \alpha_i \leq C, & i = 1, \dots, n \\ \sum_{i=1}^n \alpha_i y_i = 0. \end{cases}$$

- La règle de décision s'obtient en calculant le signe de

$$f(x) = \sum_{i=1}^n \alpha_i^* y_i \langle x_i, x \rangle + b^*.$$

### SVM dans le feature space

- Le **problème dual** consiste à maximiser

$$L_D(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{k=1}^n \alpha_i \alpha_k y_i y_k \langle \Phi(x_i), \Phi(x_k) \rangle$$

$$\text{sous les contraintes } \begin{cases} 0 \leq \alpha_i \leq C, & i = 1, \dots, n \\ \sum_{i=1}^n \alpha_i y_i = 0. \end{cases}$$

- La règle de décision s'obtient en calculant le signe de

$$f(x) = \sum_{i=1}^n \alpha_i^* y_i \langle \Phi(x_i), \Phi(x) \rangle + b^*.$$

## SVM dans le feature space avec un noyau

— Le *problème dual* consiste à maximiser

$$L_D(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{k=1}^n \alpha_i \alpha_k y_i y_k K(x_i, x_k)$$

$$\text{sous les contraintes } \begin{cases} 0 \leq \alpha_i \leq C, & i = 1, \dots, n \\ \sum_{i=1}^n \alpha_i y_i = 0. \end{cases}$$

— La règle de décision s'obtient en calculant le signe de

$$f(x) = \sum_{i=1}^n \alpha_i^* y_i K(x_i, x) + b^*.$$

## Conclusion

— Pour calculer la svm, on n'a *pas besoin de connaître  $\mathcal{H}$  ou  $\Phi$* , il suffit de connaître  $K$ !

## Questions

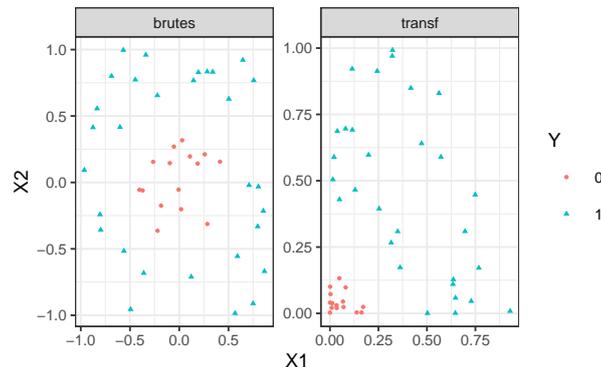
Qu'est-ce qu'un *noyau*? Comment construire un noyau?

**Théorème 4.1** ([Aronszajn, 1950]). Une fonction  $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$  est un *noyau* si et seulement si elle est (symétrique) *définie positive*, c'est-à-dire ssi

1.  $K(x, x') = K(x', x) \forall (x, x') \in \mathcal{X}^2$ ;
2.  $\forall (x_1, \dots, x_N) \in \mathcal{X}^N$  et  $\forall (a_1, \dots, a_N) \in \mathbb{R}^N$

$$\sum_{i=1}^N \sum_{j=1}^N a_i a_j K(x_i, x_j) \geq 0.$$

## Exemple



Si

$$\begin{aligned} \Phi : \quad \mathbb{R}^2 &\rightarrow \mathbb{R}^3 \\ (x_1, x_2) &\mapsto (x_1^2, \sqrt{2}x_1x_2, x_2^2) \end{aligned}$$

alors  $K(x, x') = (x^t x')^2$  (noyau polynomial de degré 2).

## Exemples de noyau

1. *Linéaire* (vanilladot) :  $K(x, x') = \langle x, x' \rangle$ .
2. *Polynomial* (polydot) :  $K(x, x') = (\text{scale} \langle x, x' \rangle + \text{offset})^{\text{degree}}$ .
3. *Gaussien* (Gaussian radial basis function ou RBF - rbfdot)

$$K(x, x') = \exp(-\text{sigma} \|x - x'\|^2).$$

4. *Laplace* (sur  $\mathbb{R}$ ) :  $K(x, x') = \exp(-\text{sigma} \|x - x'\|)$ .
5. ...

## Remarque

Les paramètres correspondent aux noyaux proposés par la fonction `ksvm` de `kernlab` (voir [Karatzoglou et al., 2004]).

## Commentaires

1. En l'absence d'information a priori le *noyau radial* est préconisé.
2. Procédure d'optimisation pour `sigma` proposé dans `ksvm`.

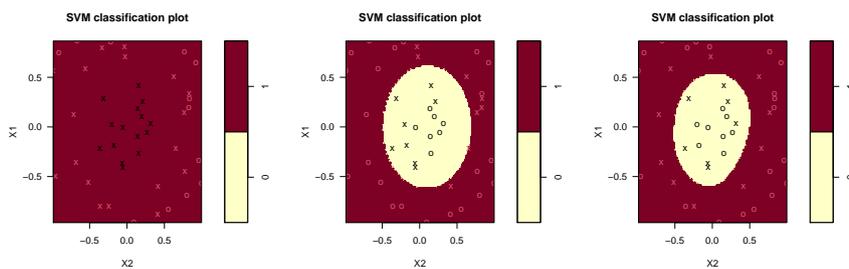
## Remarque

N'importe quelle *fonction définie positive* fait l'affaire... Possibilité de construire des noyaux (et donc de faire des svm) sur des *objets plus complexes* (courbes, images, séquences de lettres...).

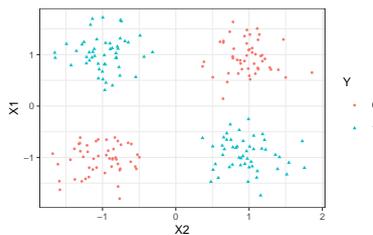
## Le coin R - exemple 1

— Argument `kernel` dans la fonction `svm`.

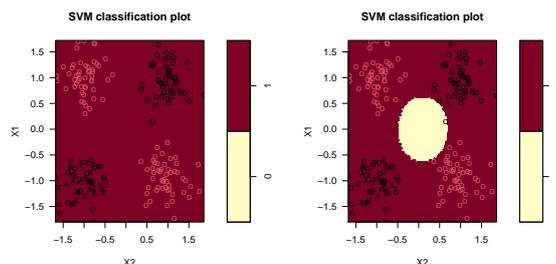
```
> svm(Y~.,data=donnees,cost=1,kernel="linear")
> svm(Y~.,data=donnees,cost=1,kernel="polynomial",degree=2)
> svm(Y~.,data=donnees,cost=1,kernel="radial",gamma=1)
```



## Le coin R - exemple 2



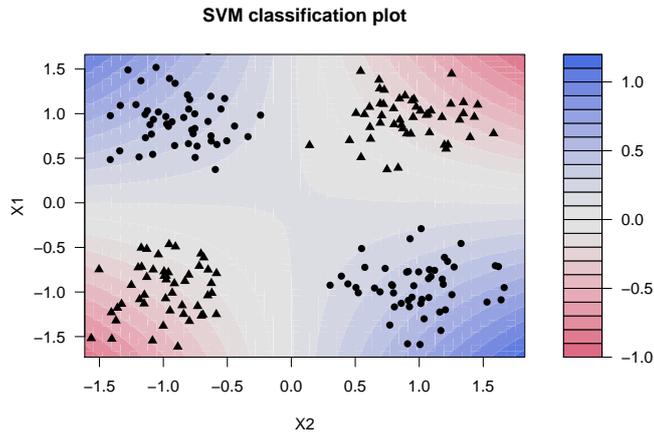
```
> svm(Y~.,data=donnees,kernel="linear",cost=1)
> svm(Y~.,data=donnees,kernel="polynomial",degree=2,cost=1)
```



## Le package kernlab

— Il propose un *choix plus large* de noyau.

```
> library(kernlab)
> mod.ksvm <- ksvm(Y~.,data=donnees,kernel="polydot",
+                 kpar=list(degree=2),C=0.001)
> plot(mod.ksvm)
```



#### 4.4 Scores et probabilités

- Jusqu'à présent nous avons utilisé la SVM uniquement pour *classer* :
  - 1 si on est d'un *coté de l'hyperplan*  $\implies \sum_{i=1}^n \alpha_i^* y_i K(x_i, x) + b^* \geq 0$  ;
  - -1 si on est de l'*autre coté*  $\implies \sum_{i=1}^n \alpha_i^* y_i K(x_i, x) + b^* < 0$ .
- *Rappel* : dans le cas linéaire la fonction

$$f(x) = \sum_{i=1}^n \alpha_i^* y_i K(x_i, x) + b^*$$

mesure la *distance entre  $x$  et l'hyperplan séparateur*.

- *Conclusion* : cette fonction peut être utilisée comme un *score*, puisque sa valeur (absolue) traduit une *confiance* que l'on a dans la prévision.

#### Probabilités

- La valeur de  $f(x)$  est *difficilement interprétable* en tant que telle.
- Il peut être intéressant de la "*ramener*" *entre 0 et 1* pour l'interpréter comme une *estimation de  $P(Y = 1|X = x)$* .

#### Une solution

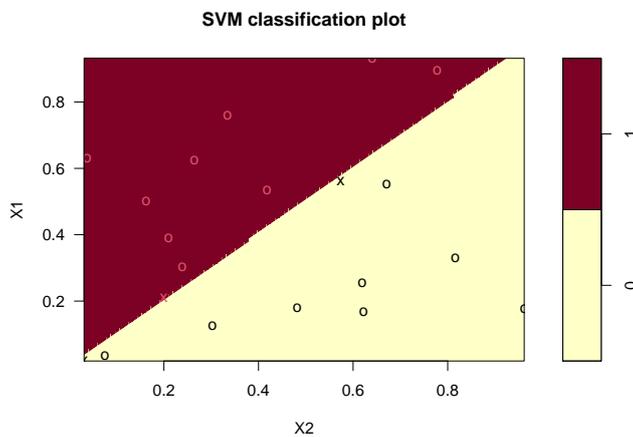
- Considérer un *modèle logistique* :

$$P(Y = 1|X = x) = \frac{1}{1 + \exp(-af(x) + b)}$$

- et d'estimer  $a$  et  $b$  par *maximum de vraisemblance* sur les données  $(f(x_i), y_i), i = 1, \dots, n$ .

#### Le coin R

```
> mod.svm <- svm(Y~., data=df, kernel="linear", cost=10000000000, probability=TRUE)
> plot(mod.svm, data=df, fill=TRUE, grid=100)
```



— Nouvelle observation :

```
> newX <- data.frame(X1=0.2,X2=0.6)
```

— Calcul du *score* et de la *proba* :

```
> predict(mod.svm,newdata=newX,decision.values = TRUE,probability=TRUE)
## 1
## 0
## attr(,"decision.values")
##      0/1
## 1 36.44796
## attr(,"probabilities")
##      0      1
## 1 0.9770206 0.02297939
## Levels: 0 1
```

— On peut retrouver cette proba avec :

```
> a <- mod.svm$probA
> b <- mod.svm$probB
> 1/(1+exp(a*36.44796+b))
## [1] 0.9770206
```

## 4.5 Compléments : SVM multi-classes et SVR

### Cible multi-classes ou quantitative

- On a abordé ici uniquement le problème de la *classification binaire* :  $y_i \in \{-1, 1\}$ .
- Les SVM se généralisent aux cas *multi-classes* :  $y_i \in \{1, \dots, M\}$
- et à la *régression* :  $y_i \in \mathbb{R}$ .

#### 4.5.1 SVM multiclass

- On suppose ici que  $y_i \in \{1, \dots, M\}$
- Il existe plusieurs approches pour généraliser les SVM à ce contexte, notamment :
- **One against one**

##### *Idée*

Faire une SVM binaire sur **toutes les paires**  $(j, k) \in \{1, \dots, M\}^2$  avec  $j \neq k$  et choisir le **groupe qui gagne le plus souvent**.

- **One against all**

##### *Idée*

Faire une SVM binaire de **chaque groupe contre les autres** et choisir le **groupe qui a la "plus belle victoire"**.

### One against one

#### *Algorithme*

1. Pour chaque paire  $(j, k)$ , faire la SVM **binaire** avec uniquement les individus des groupes  $k$  et  $j$  ;
2. On obtient ainsi  $M(M - 1)/2$  **règles "linéaires"**  $f_{j,k}(x)$ .
3. On calcule pour  $j = 1, \dots, M$

$$V(j) = \sum_{k \neq j} \text{signe}(f_{j,k}(x))$$

qui représente le **nombre de fois où on a voté  $j$  contre les autres groupes**.

4. On classe un nouvel individu  $x$  dans le groupe qui a **remporté le plus de suffrage** :

$$f(x) = \underset{j}{\operatorname{argmax}} V(j).$$

## One against all

### Algorithme

1. Faire une SVM binaire avec **tous les individus** de chaque groupe contre les autres.
2. On obtient ainsi  $M$  **règles "linéaires"**  $f_j(x)$  (groupe  $j$  contre les autres).
3. On classe un nouvel individu dans la classe qui a le score le plus élevé :

$$f(x) = \operatorname{argmax}_j f_j(x).$$

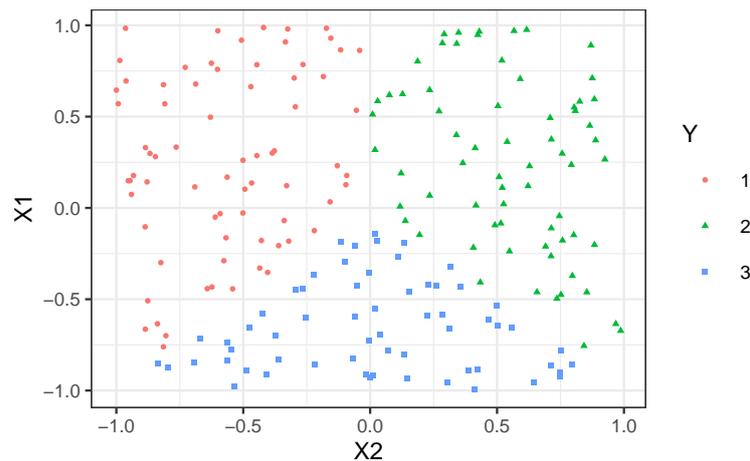
## Comparaison

- $M$  SVM binaire avec l'approche **one against all** contre  $M(M-1)/2$  avec le **one against one** mais...
- **moins** d'individus dans les **one against one**.
- Risque de déséquilibre plus fort avec le **one against all** (mais généralement plus rapide).
- Comme dans le cas binaire, il faut **sélectionner** le **paramètre de complexité**, le **noyau**, les **paramètres du noyau**...

### Le coin R

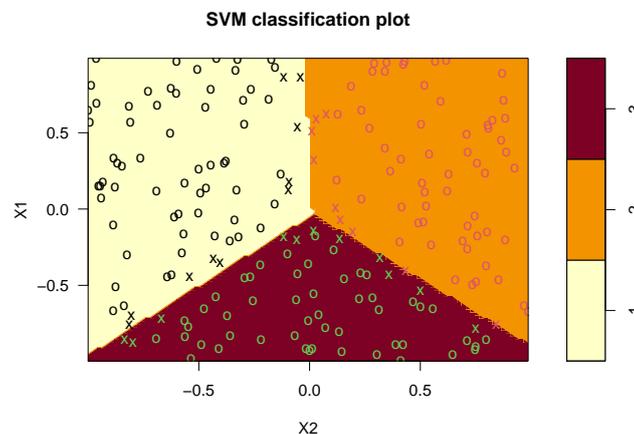
- L'approche **one against one** est plus souvent utilisée.
- C'est le cas par défaut avec **svm** de *e1071* et **ksvm** de *kernelab*.

## Exemple



## SVM linéaire multi classes

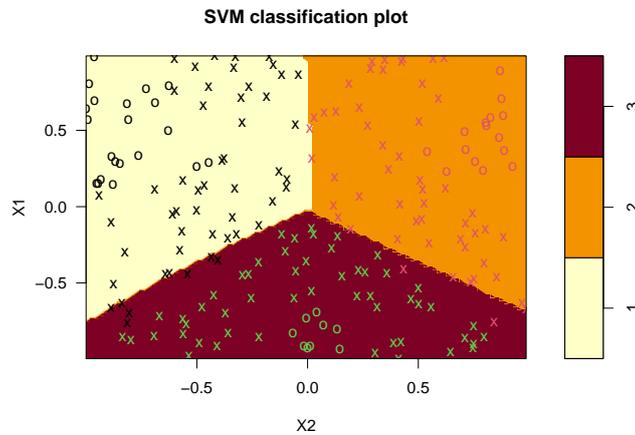
```
> multi1 <- svm(Y~.,data=df,cost=10,kernel="linear")
> plot(multi1,data=df,grid=100)
```



## SVM non linéaire multi classes

— Il "suffit" d'utiliser un *noyau*.

```
> multi2 <- svm(Y~.,data=df,cost=0.1,kernel="sigmoid")
> plot(multi2,data=df,grid=100)
```



### 4.5.2 Support vector regression (SVR)

— On suppose ici que les  $y_i$  sont dans  $\mathbb{R}$ .

— On ne va plus chercher l'hyperplan qui *sépare au mieux les groupes* mais

— l'hyperplan  $(w, b)$  qui *"approche au mieux"* les valeurs  $y_i$

$$|\langle w, x_i \rangle + b - y_i| \text{ petits.}$$

### Comparaison avec les MCO

— *Approche MCO* (rappel) : on cherche  $(w, b)$  qui minimise

$$\sum_{i=1}^n (y_i - \langle w, x_i \rangle - b)^2$$

— *Approche SVR* : on veut

1. tous les points à distance de moins de  $\varepsilon$  de  $(w, b)$  ;
2.  $(w, b)$  de marge maximale ( $\|w\|$  minimale).

### Optimisation SVR

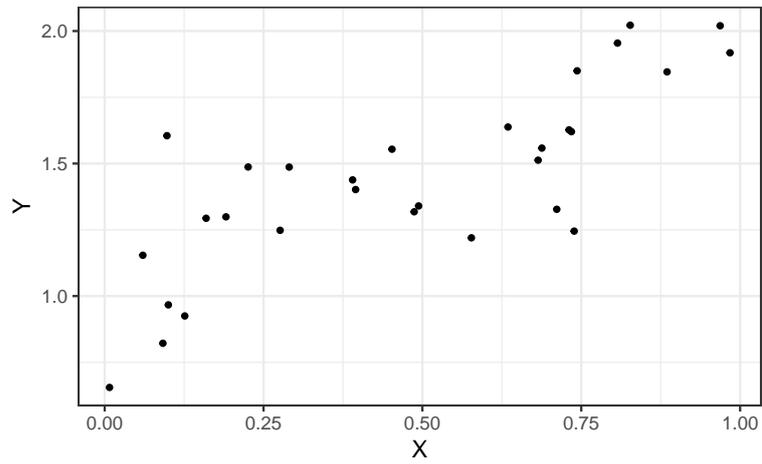
On va chercher à *minimiser la norme de  $w$*  en se fixant comme contrainte que les  $y_i$  ne soient pas "trop loin" de l'hyperplan :

$$\min_{w,b} \frac{1}{2} \|w\|^2$$

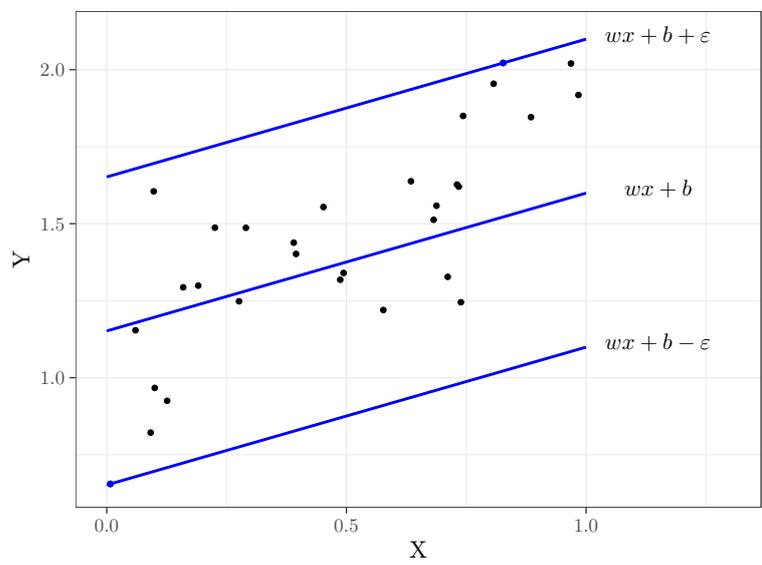
$$\text{sous les contraintes } |y_i - \langle w, x_i \rangle - b| \leq \varepsilon, \quad i = 1, \dots, n,$$

où  $\varepsilon > 0$  est un *paramètre à calibrer par l'utilisateur*.

### Un exemple en dimension 1

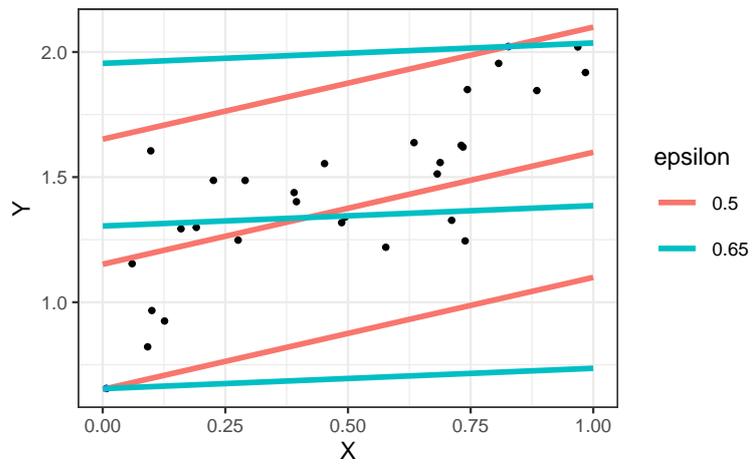


### Un exemple en dimension 1



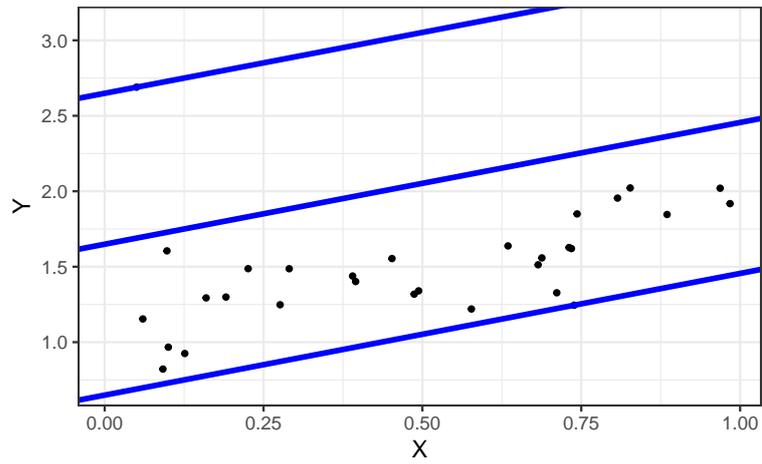
### Influence de $\varepsilon$

— Il contrôle le *niveau de tolérance* que l'on se donne.



### Alléger la contrainte...

— La contrainte nécessite souvent de *prendre des grandes valeurs* de  $\varepsilon$ ...



— Clairement *pas satisfaisant* de prendre  $\varepsilon$  *trop grand*.

### Idée

- Comme pour la **SVM binaire**, autoriser des observations à se situer en dehors de la marge!
- *Comment?* En introduisant des **slack variables!**

### SVR cas général

#### Le problème d'optimisation

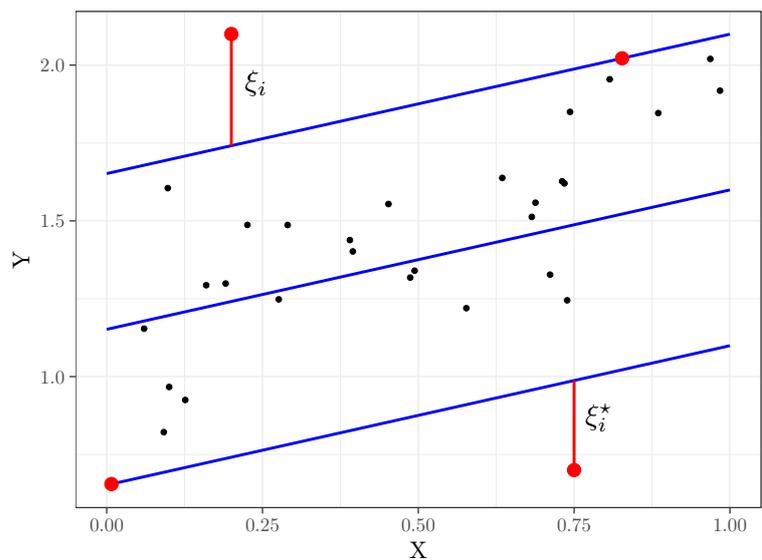
On cherche  $(w, b, \xi, \xi^*)$  qui minimise

$$\frac{1}{2} \|w\|^2 + C \sum_{i=1}^n (\xi_i + \xi_i^*)$$

sous les contraintes

$$\begin{cases} y_i - \langle w, x_i \rangle - b \leq \varepsilon + \xi_i, & i = 1, \dots, n, \\ \langle w, x_i \rangle + b - y_i \leq \varepsilon + \xi_i^*, & i = 1, \dots, n \\ \xi_i \geq 0, \xi_i^* \geq 0, & i = 1, \dots, n \end{cases}$$

### Slack variables en régression



## Rien ne change après...

- Les solutions s'obtiennent en résolvant le *problème dual*  $\implies \alpha_i, \alpha_i^*$ .
- Les données (les  $X$ ) sont généralement *centrées-réduites* pour éviter les problèmes d'échelle.
- Les observations vérifiant  $\alpha_i^* - \alpha_i \neq 0$  sont les *vecteurs supports*.
- L'hyperplan optimal se déduit des *vecteurs supports* :

$$w^* = \sum_{i=1}^n (\alpha_i^* - \alpha_i) x_i.$$

- L'*astuce du noyau* reste d'actualité pour prendre en compte de la *non linéarité*.
- Il faut *sélectionner*  $C$ , le noyau (et ses paramètres) ainsi que  $\varepsilon$ ...

## Le coin R

- Là aussi, pas grand chose ne change.

```
> svm(Y~., data=df, kernel="linear", epsilon=0.5, cost=100)
##
## Call:
## svm(formula = Y ~ ., data = df, kernel = "linear", epsilon = 0.5,
##      cost = 100)
##
##
## Parameters:
##   SVM-Type:  eps-regression
##   SVM-Kernel: linear
##      cost:   100
##   gamma:    1
##   epsilon:  0.5
##
##
## Number of Support Vectors:  11
```

## Conclusion

- Algorithme machine learning pouvant être utilisé en *régression* et en *classification supervisée*.
- Méthode *linéaire* mais prise en compte possible de la *non linéarité* grâce à l'*astuce du noyau*.
- *Calibration difficile* : beaucoup de paramètres
  1. paramètre de cout  $C$
  2. noyau
  3. paramètres du noyau
  4. seuil de tolérance  $\varepsilon$  pour la régression
- et souvent *peu d'information a priori* sur la valeur de ces paramètres...

## 5 Bibliographie

### Références

#### Biblio2

- [Aronszajn, 1950] Aronszajn, N. (1950). Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68 :337–404.
- [Bühlmann and van de Geer, 2011] Bühlmann, P. and van de Geer, S. (2011). *Statistics for high-dimensional data*. Springer.
- [Cornillon et al., 2019] Cornillon, P., Hengartner, N., Matzner-Løber, E., and Rouvière, L. (2019). *Régression avec R*. EDP Sciences.

- [Fromont, 2015] Fromont, M. (2015). Apprentissage statistique. Université Rennes 2, diapos de cours.
- [Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer, second edition.
- [Hastie et al., 2015] Hastie, T., Tibshirani, R., and Wainwright, M. (2015). *Statistical Learning with Sparsity : The Lasso and Generalizations*. CRC Press. [https://web.stanford.edu/~hastie/StatLearnSparsity\\_files/SLS.pdf](https://web.stanford.edu/~hastie/StatLearnSparsity_files/SLS.pdf).
- [Karatzoglou et al., 2004] Karatzoglou, A., Smola, A., Hornik, K., and Zeileis, A. (2004). kernlab – an s4 package for kernel methods in r. *Journal of Statistical Software*, 11(9).
- [Tibshirani, 1996] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58 :267–288.
- [Zou and Hastie, 2005] Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B*, 67 :301–320.

## Troisième partie

# Algorithmes non linéaires

- Algorithmes *linéaires* :

$$f(x) = f_{\beta}(x) = \beta_0 + \beta_1 x_1 + \dots + \beta_d x_d.$$

- *Problème* : tous les problèmes ne sont pas linéaires.
- Possible d'*ajouter de la non linéarité* dans les algorithmes linéaires : effets quadratiques, interaction...
- *Difficile pour l'utilisateur* de trouver quels effets ajouter ! Surtout lorsque  $d$  est grand.

### Dans cette partie

Présentation de quelques algorithmes *non linéaires* :

- Méthodes par *arbres*.
- *Réseaux de neurones*.

## 1 Arbres

### Présentation

- Les arbres sont des algorithmes de prédiction qui fonctionnent en *régression et en discrimination*.
- Il existe *différentes variantes* permettant de construire des prédicteurs par arbres.
- Nous nous focalisons dans cette partie sur la *méthode CART* [Breiman et al., 1984] qui est la plus utilisée.

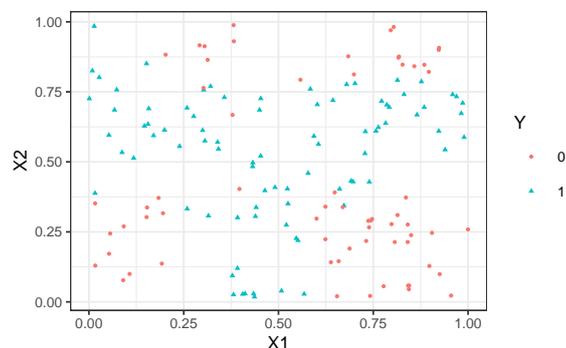
### 1.1 Arbres binaires

#### Notations

- On cherche à *expliquer une variable*  $Y$  par  $d$  *variables explicatives*  $X_d, \dots, X_d$ .
- $Y$  peut admettre un nombre quelconque de modalités et les variables  $X_1, \dots, X_d$  peuvent être *qualitatives et/ou quantitatives*.
- Néanmoins, pour simplifier on se place dans un premier temps en *discrimination binaire* :  $Y$  admet 2 modalités (-1 ou 1). On suppose de plus que l'on a simplement 2 variables explicatives quantitatives.

#### Représentation des données

- On dispose de  $n$  observations  $(x_1, y_1), \dots, (x_n, y_n)$  où  $x_i \in \mathbb{R}^2$  et  $y_i \in \{0, 1\}$ .

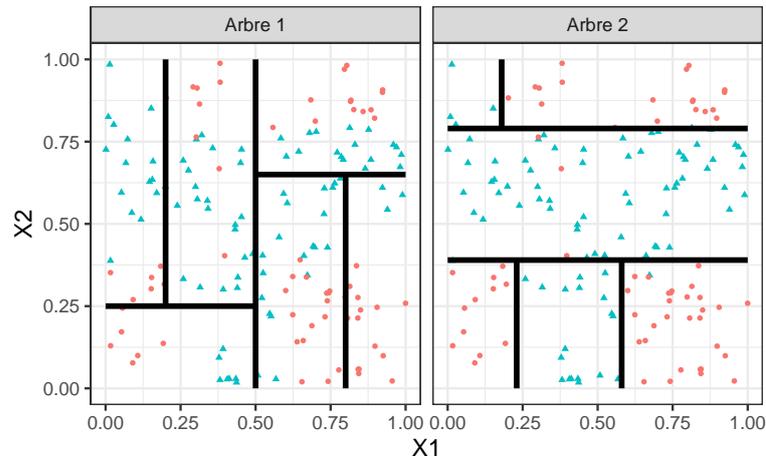


#### Approche par arbres

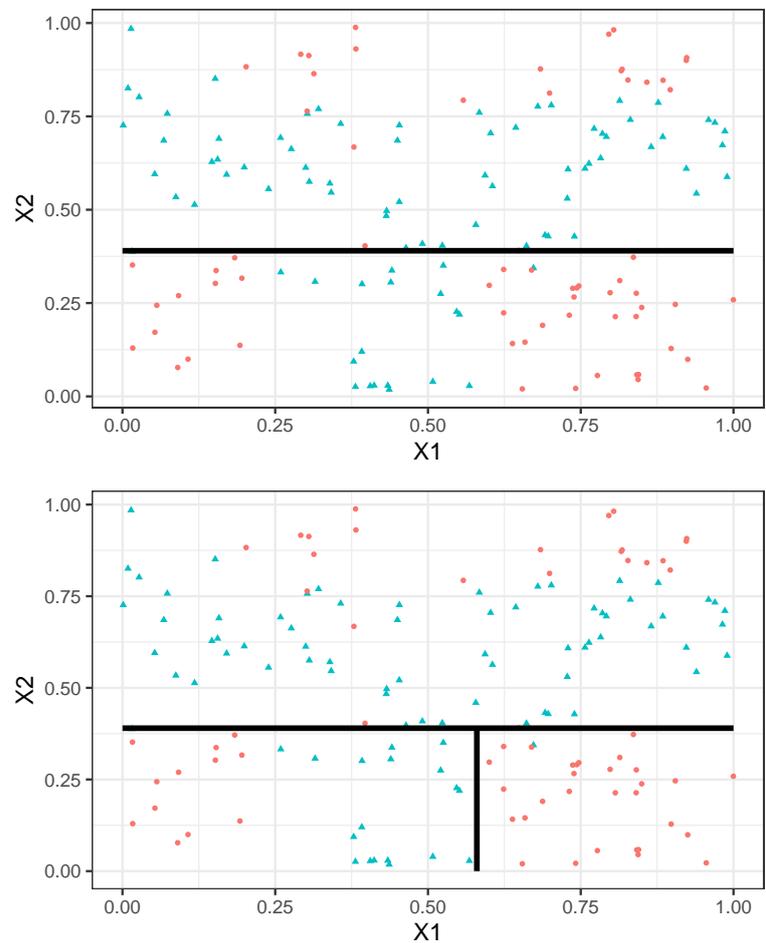
Trouver une *partition* des observations qui *sépare* "au mieux" les points rouges des points bleus.

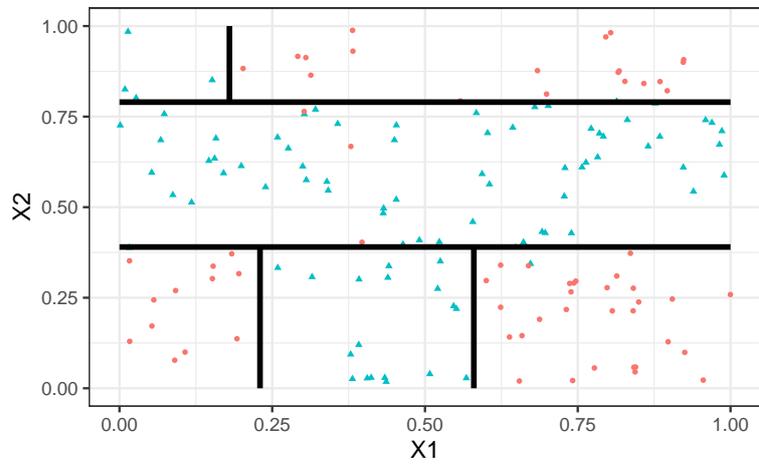
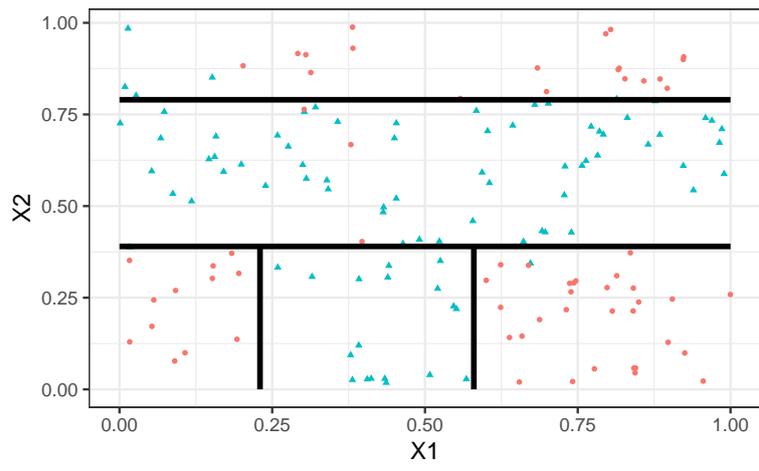
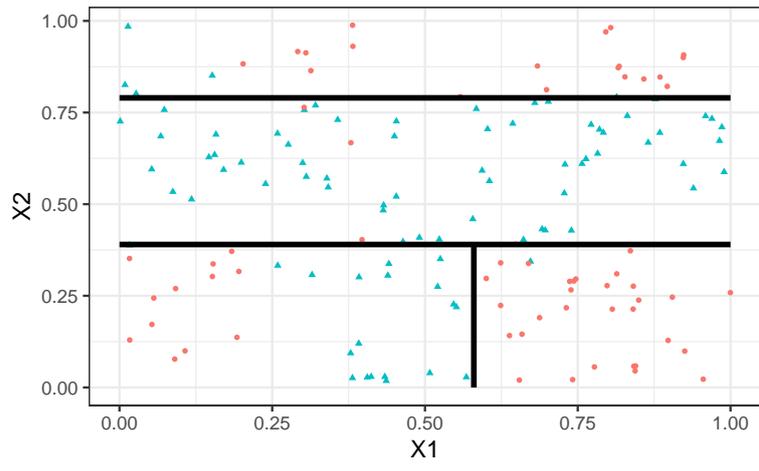
## Arbres binaires

- La *méthode CART* propose de construire une partition basée sur des divisions *successives parallèles aux axes*.
- 2 exemples de partition :

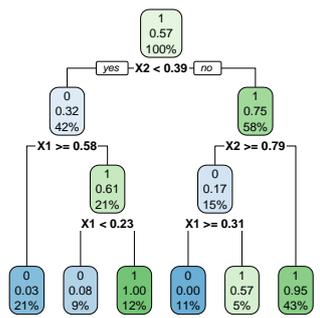
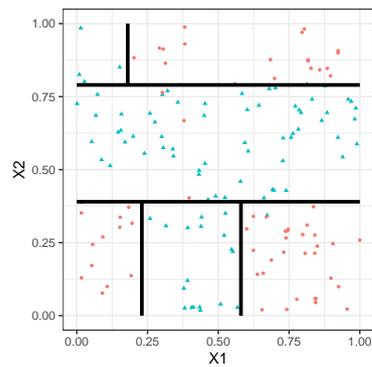


- A chaque étape, la méthode cherche une *nouvelle division* : une *variable* et un *seuil* de coupure.





Représentation de l'arbre



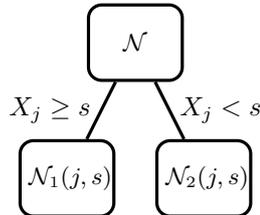
## Remarque

Visuel de *droite* plus pertinent :

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

## Vocabulaire

- Chaque coupure divise une partie de  $\mathbb{R}^d$  en deux parties appelées *nœuds*.
- Le premier nœud, qui contient toutes les observations, est le *nœud racine*.
- Une coupure divise un nœud en deux *nœuds fils* :



- Les nœuds qui ne sont pas découpés (en bas de l'arbre) sont les *nœuds terminaux* ou *feuilles* de l'arbre.

## Arbre et algorithme de prévision

- L'arbre construit, les *prévisions* se déduisent à partir de *moyennes faites dans les feuilles*.
  - On note  $\mathcal{N}(x)$  la feuille de l'arbre qui contient  $x \in \mathbb{R}^d$ , les prévisions s'obtiennent selon :
1. *Régression*  $\implies$  moyenne des  $y_i$  de la feuille

$$m_n(x) = \frac{1}{|\mathcal{N}(x)|} \sum_{i: x_i \in \mathcal{N}(x)} y_i$$

2. *Classification (classe)*  $\implies$  vote à la majorité :

$$g_n(x) = \operatorname{argmax}_k \sum_{i: x_i \in \mathcal{N}(x)} \mathbf{1}_{y_i=k}$$

3. *Classification (proba)*  $\implies$  proportion d'obs. du groupe  $k$  :

$$S_{k,n}(x) = \frac{1}{|\mathcal{N}(x)|} \sum_{i: x_i \in \mathcal{N}(x)} \mathbf{1}_{y_i=k}.$$

## Questions

1. Comment *découper* un nœud ?  
 $\implies$  si on dispose d'un algorithme pour découper un nœud, il suffira de le répéter.
2. Comment choisir la *profondeur de l'arbre* ?
  - Profondeur *maximale* ? (on découpe jusqu'à ne plus pouvoir) *sur-ajustement* ?
  - Critère d'arrêt ?
  - Élagage ? (on construit un arbre profond et on enlève des branches "inutiles"...).

## 1.2 Choix des coupures

- Une *coupure* = un couple  $(j, s) \in \{1, \dots, d\} \times \mathbb{R}$ .
- *Idée* : définir un *critère* mesure la performance d'une coupure et choisir celle qui optimise le critère.
- *Coupure performante*  $\implies$  les deux nœuds fils sont *homogènes* vis-à-vis de  $Y$ .

### Fonction d'impureté

- **Objectif** : mesurer l'homogénéité d'un nœud.
- **Intérêt** : choisir la coupure qui maximise la pureté des nœuds fils.

## Critère de découpe

— L'*impureté*  $\mathcal{I}$  d'un nœud doit être :

1. *faible* lorsque un nœud est homogène : les valeurs de  $Y$  dans le nœud sont *proches*.
2. *élevée* lorsque un nœud est hétérogène : les valeurs de  $Y$  dans le nœud sont *dispersées*.

### L'idée

Une fois  $\mathcal{I}$  définie, on choisira le couple  $(j, s)$  qui *maximise le gain d'impureté* :

$$\Delta(j, s) = p(\mathcal{N})\mathcal{I}(\mathcal{N}) - (p(\mathcal{N}_1(j, s))\mathcal{I}(\mathcal{N}_1(j, s)) + p(\mathcal{N}_2(j, s))\mathcal{I}(\mathcal{N}_2(j, s)))$$

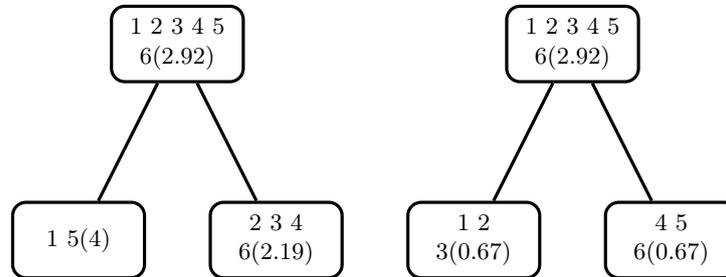
où  $p(\mathcal{N})$  représente la proportion d'observations dans le nœud  $\mathcal{N}$ .

#### 1.2.1 Cas de la régression

— Une mesure naturelle de l'*impureté* d'un nœud  $\mathcal{N}$  en *régression* est la *variance* du nœud :

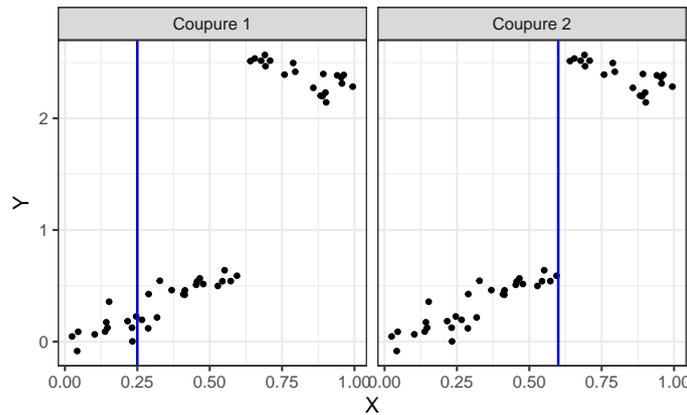
$$\mathcal{I}(\mathcal{N}) = \frac{1}{|\mathcal{N}|} \sum_{i: x_i \in \mathcal{N}} (y_i - \bar{y}_{\mathcal{N}})^2,$$

où  $\bar{y}_{\mathcal{N}}$  désigne la moyenne des  $Y_i$  dans  $\mathcal{N}$ .



⇒ coupure de *droite* plus performante.

### Exemple



	$\mathcal{I}(\mathcal{N})$	$\mathcal{I}(\mathcal{N}_1)$	$\mathcal{I}(\mathcal{N}_2)$	$\Delta$
Gauche	1.05	0.01	0.94	0.34
Droite	1.05	0.04	0.01	<b>1.02</b>

#### 1.2.2 Cas de la classification supervisée

— Les  $Y_i, i = 1, \dots, n$  sont à valeurs dans  $\{1, \dots, K\}$ .

— On cherche une fonction  $\mathcal{I}$  telle que  $\mathcal{I}(\mathcal{N})$  soit

— *petite* si un *label majoritaire* se distingue clairement dans  $\mathcal{N}$  ;

— *grande* sinon.

### Impureté

L'impureté d'un nœud  $\mathcal{N}$  en classification se mesure selon

$$\mathcal{I}(\mathcal{N}) = \sum_{j=1}^K f(p_j(\mathcal{N}))$$

où

- $p_j(\mathcal{N})$  représente la proportion d'observations de la classe  $j$  dans le nœud  $\mathcal{N}$ .
- $f$  est une fonction (concave)  $[0, 1] \rightarrow \mathbb{R}^+$  telle que  $f(0) = f(1) = 0$ .

### Exemples de fonctions $f$

- Si  $\mathcal{N}$  est pur, on veut  $\mathcal{I}(\mathcal{N}) = 0 \implies$  c'est pourquoi  $f$  doit vérifier  $f(0) = f(1) = 0$ .
- Les 2 mesures d'impureté les plus classiques sont :
  1. *Gini* :  $f(p) = p(1 - p)$  ;
  2. *Information* :  $f(p) = -p \log(p)$ .

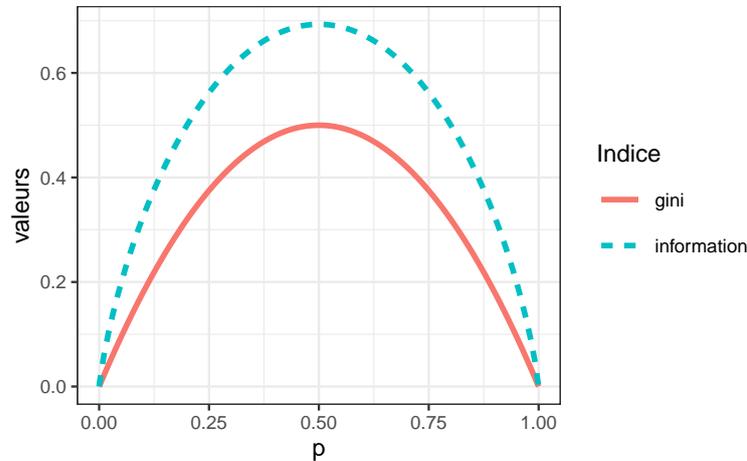
### Cas binaire

Dans ce cas on a

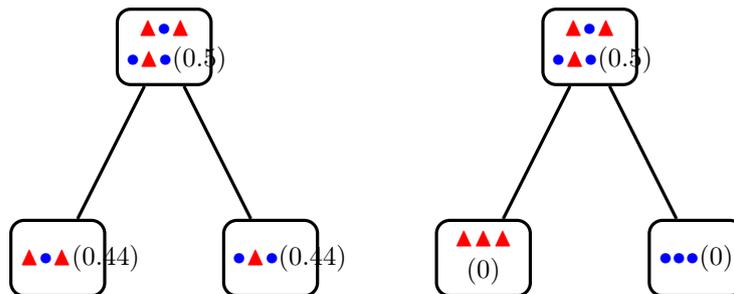
1.  $\mathcal{I}(\mathcal{N}) = 2p(1 - p)$  pour *Gini*
2.  $\mathcal{I}(\mathcal{N}) = -p \log p - (1 - p) \log(1 - p)$  pour *Information*

où  $p$  désigne la proportion de 1 (ou 0) dans  $\mathcal{N}$ .

### Impureté dans le cas binaire

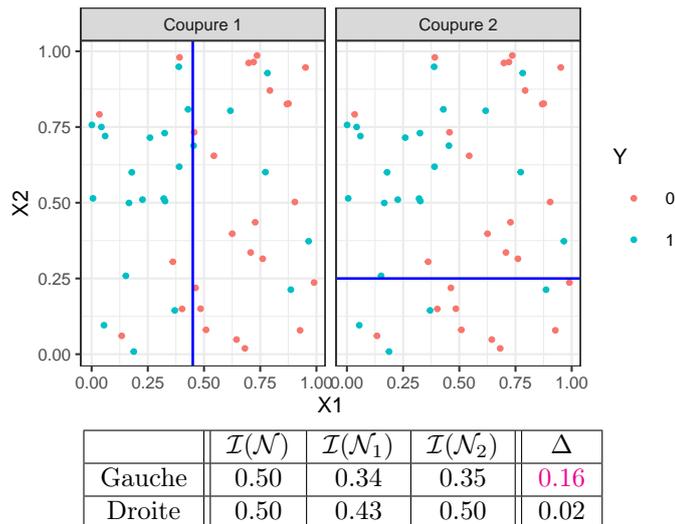


### Exemple 1



$\implies$  coupure de *droite* plus performante.

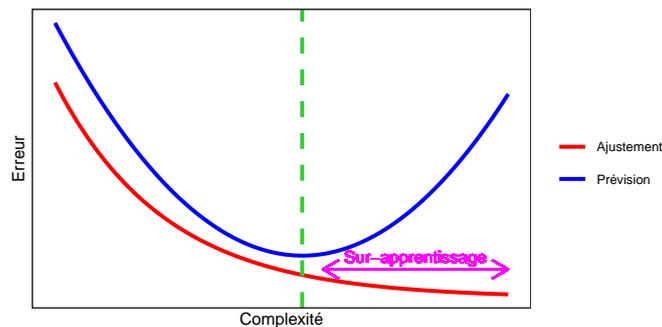
## Exemple 2



## 1.3 Élagage

### Pourquoi élaguer ?

- Les coupures permettent de *séparer les données selon Y*  $\implies$  plus on coupe mieux on ajuste !
- Risque de *sur-ajustement* si on coupe trop !



### Complexité d'un arbre

Représentée par son **nombre de coupures** ou sa **profondeur**.

### Comment faire ?

- *Tester tous les arbres ?*  $\implies$  possible uniquement sur de petits échantillons !
- *Critère d'arrêt* : ne plus découper si une certaine condition est vérifiée.  $\implies$  possible mais... une coupure peut ne pas être pertinente alors que des **coupures plus basses** le seront !

### Élaguer

1. Considérer un **arbre (trop) profond**  $\implies$  qui sur-ajuste ;
2. Supprimer les **branches peu utiles**.

### Élagage CART

- Tester *tous les sous-arbres* d'un arbre très profond se révèlent souvent *trop coûteux* en temps de calcul.
- [Breiman et al., 1984] propose une stratégie d'élagage qui permet de se ramener à *une suite d'arbres emboîtés*

$$\mathcal{T}_{max} = \mathcal{T}_0 \supset \mathcal{T}_1 \supset \dots \supset \mathcal{T}_K.$$

de **taille raisonnable** (plus petite que  $n$ ).

- Il est ensuite possible de *choisir un arbre dans cette suite* par des méthodes traditionnelles :
  1. choix d'un risque ;
  2. optimisation de ce risque (par validation croisée par exemple).

Pour aller plus vite

## Construction de la suite de sous arbres

- Soit  $T$  un arbre à  $|T|$  nœuds terminaux  $\mathcal{N}_1, \dots, \mathcal{N}_{|T|}$ .
- Soit  $R(\mathcal{N})$  un risque (d'ajustement) dans le nœud  $\mathcal{N}$  :
  - *Régression* :

$$R_m(T) = \frac{1}{N_m} \sum_{i: x_i \in \mathcal{N}_m} (y_i - \bar{y}_{\mathcal{N}_m})^2$$

- *Classification* :

$$R_m(T) = \frac{1}{N_m} \sum_{i: x_i \in \mathcal{N}_m} \mathbf{1}_{y_i \neq y_{\mathcal{N}_m}}$$

## Définition

Soit  $\alpha \geq 0$ , le critère *coût/complexité* est défini par :

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m R_m(T) + \alpha |T|.$$

## Idée

- $C_\alpha(T)$  est un critère qui prend en compte l'*adéquation* d'un arbre et sa *complexité*.
- L'*idée* est de chercher un arbre  $T_\alpha$  qui minimise  $C_\alpha(T)$  pour une valeur de  $\alpha$  bien choisie.

## Remarque

- $\alpha = 0 \implies T_\alpha = T_0 = T_{\max}$ .
- $\alpha = +\infty \implies T_\alpha = T_{+\infty} = T_{\text{root}}$  *arbre sans coupure*.

## Question (a priori difficile)

Comment calculer  $T_\alpha$  qui minimise  $C_\alpha(T)$  ?

## Deux lemmes

### Lemme 1

Si  $T_1$  et  $T_2$  sont deux sous-arbres de  $T_{\max}$  avec  $R_\alpha(T_1) = R_\alpha(T_2)$ . Alors  $T_1 \subset T_2$  ou  $T_2 \subset T_1$

$\implies$  garantit une unique solution de *taille minimale*.

### Lemme 2

Si  $\alpha > \alpha'$  alors  $T_\alpha = T_{\alpha'}$  ou  $T_\alpha \subset T_{\alpha'}$ .

$\implies$  garantit une *stabilité des solutions* lorsque  $\alpha$  parcourt  $\mathbb{R}^+$   $\implies$  elles vont être *emboîtées* les unes dans les autres.

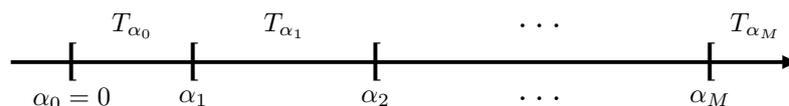
## Théorème [Breiman et al., 1984]

Il existe une suite finie  $\alpha_0 = 0 < \alpha_1 < \dots < \alpha_M$  avec  $M \leq |T_{\max}|$  et une suite associée d'arbres emboîtés  $(T_{\alpha_m})_m$

$$T_{\max} = T_{\alpha_0} \supset T_{\alpha_1} \supset \dots \supset T_{\alpha_M} = T_{\text{root}}$$

telle que  $\forall \alpha \in [\alpha_m, \alpha_{m+1}[$

$$T_m \in \operatorname{argmin}_{T \subseteq T_{\max}} C_\alpha(T).$$



## Commentaires

- Nombre de minimiseurs de  $C_\alpha(T)$  est "*petit*".
- Ils s'obtiennent en *élaguant* : en supprimant des branches.

## Exemple

- On visualise la *suite de sous-arbres* avec la fonction **printcp** ou dans l'objet **rpart** :

```
> library(rpart)
> set.seed(123)
> arbre <- rpart(Y~.,data=don.2D.arbre,cp=0.0001,minsplit=2)
> arbre$cptable
##          CP nsplit  rel error   xerror   xstd
## 1 0.353846154    0 1.00000000 1.00000000 0.09336996
## 2 0.230769231    1 0.64615385 0.7076923 0.08688336
## 3 0.138461538    2 0.41538462 0.5076923 0.07805324
## 4 0.061538462    4 0.13846154 0.2153846 0.05481185
## 5 0.015384615    5 0.07692308 0.1846154 0.05111769
## 6 0.007692308    6 0.06153846 0.2461538 0.05816388
## 7 0.000100000   14 0.00000000 0.2153846 0.05481185
```

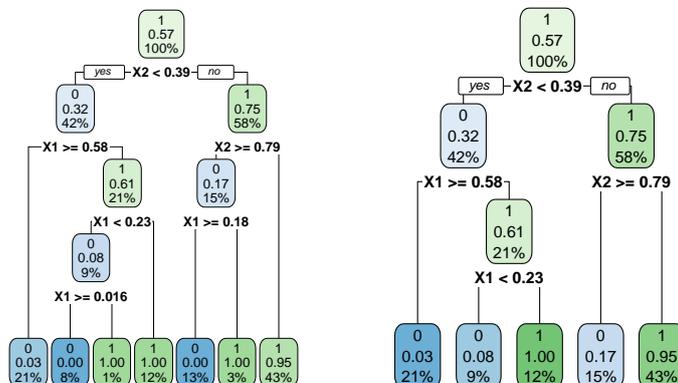
## Sorties printcp

- Suite de 7 *arbres emboîtés*.
- **CP** : complexity parameter, il mesure la complexité de l'arbre :  $CP \searrow \implies$  complexité  $\nearrow$ .
- **nsplit** : nombre de coupures de l'arbre.
- **rel.error** : erreur (normalisée) calculée sur les données d'apprentissage  $\implies$  **erreur d'ajustement**.
- **xerror** : erreur (normalisée) calculée par validation croisée 10 blocs  $\implies$  **erreur de prévision** (voir diapos suivantes).
- **xstd** : écart-type associé à l'erreur de validation croisée.

## Visualisation

- On peut les visualiser en combinant **prune** (extraction) et **rpart.plot** (tracé) :

```
> arbre1 <- prune(arbre,cp=0.01)
> arbre2 <- prune(arbre,cp=0.1)
> library(rpart.plot)
> rpart.plot(arbre1);rpart.plot(arbre2)
```



## Choix de l'arbre final

- Choisir un arbre dans la suite revient à *choisir une valeur de  $\alpha$* .
- Ce choix s'effectue généralement de façon classique :
  1. Choix d'un **risque**.
  2. **Estimation** du risque par **ré-échantillonnage** (CV par exemple) pour tous les  $\alpha_m$ .
  3. **Sélection** du  $\alpha_m$  qui **minimise** le risque estimé.

## Remarque

La fonction **rpart** effectue par défaut une validation croisée 10 blocs en prenant :

- le *risque quadratique* en régression.
- l'*erreur de classification* en classification.

### Validation croisée rpart

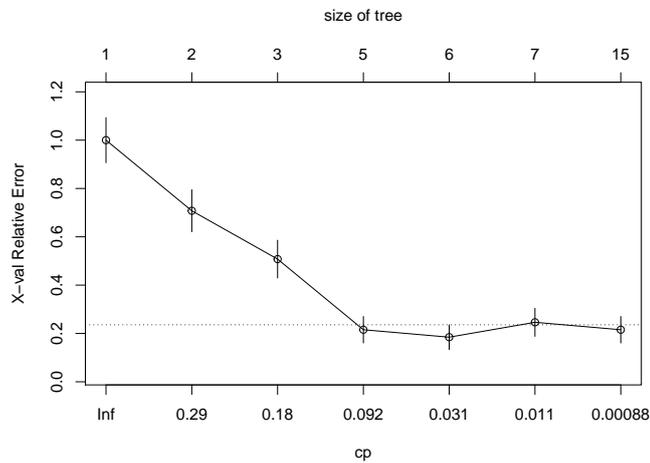
1. Calculer  $\beta_0 = 0$ ,  $\beta_1 = \sqrt{\alpha_1 \alpha_2}$ ,  $\dots$   $\beta_{M-1} = \sqrt{\alpha_{M-1} \alpha_M}$ ,  $\beta_M = +\infty$ .
2. Pour  $k = 1, \dots, K$ 
  - (a) Construire l'arbre maximal sur l'ensemble des données privé du  $k^e$  bloc, c'est-à-dire  $\mathcal{B}^{-k} = \{(x_i, y_i) : i \in \{1, \dots, n\} \setminus B_k\}$ .
  - (b) Appliquer l'algorithme d'élagage à cet arbre maximal, puis extraire les arbres qui correspondent aux valeurs  $\beta_m, m = 0, \dots, M \implies T_{\beta_m}(\cdot, \mathcal{B}^{-k})$ .
  - (c) Calculer les valeurs prédites par chaque arbre sur le bloc  $k$  :  $T_{\beta_m}(x_i, \mathcal{B}^{-k}), i \in B_k$ .
3. En déduire les erreurs pour chaque  $\beta_m$  :

$$\widehat{\mathcal{R}}(\beta_m) = \frac{1}{n} \sum_{k=1}^K \sum_{i \in B_k} \ell(y_i, T_{\beta_m}(x_i, \mathcal{B}^{-k})).$$

**Retourner** : une valeur  $\alpha_m$  telle que  $\widehat{\mathcal{R}}(\beta_m)$  est minimum.

- Les erreurs de validation croisée se trouvent dans la colonne `xerror` de l'élément `cpstable`.
- On peut les visualiser avec `plotcp` :

```
> plotcp(arbre)
```

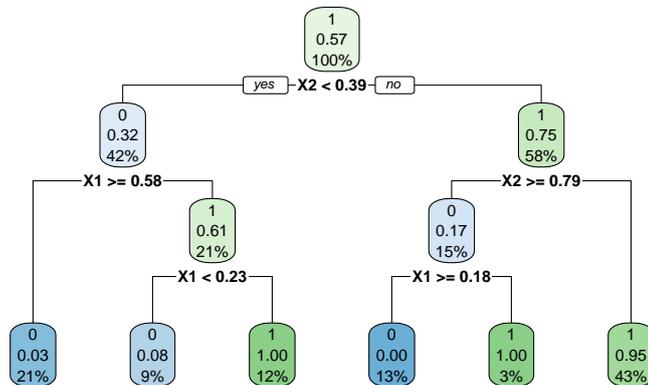


- Il reste à choisir l'arbre qui *minimise l'erreur de prévision* :

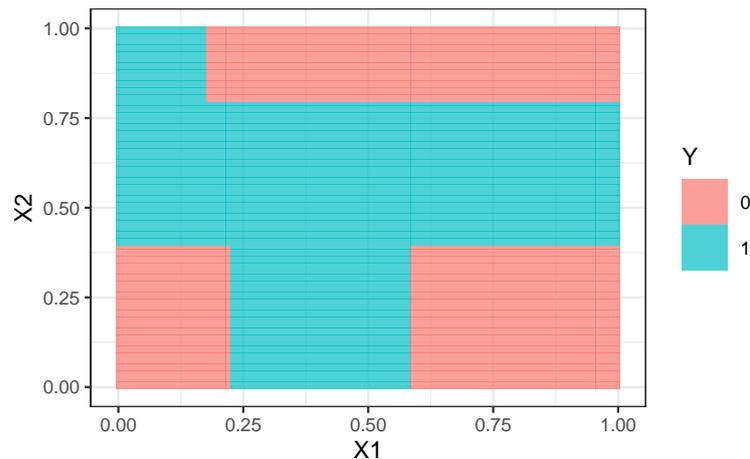
```
> cp_opt <- as_tibble(arbre$cpstable) %>% arrange(xerror) %>%
+ slice(1) %>% select(CP) %>% as.numeric()
> cp_opt
## [1] 0.01538462
```

- et à le visualiser :

```
> arbre_final <- prune(arbre, cp=cp_opt)
> rpart.plot(arbre_final)
```



- 2 variables explicatives  $\implies$  on peut visualiser l'arbre final
- en coloriant le carré  $[0, 1]^2$  en fonction *des valeurs prédites*.



## Prévision

- Nouvel individu :

```
> xnew <- tibble(X1=0.4, X2=0.5)
```

- Prévision de la *classe* :

```
> predict(arbre_final, newdata=xnew, type="class")
## 1
## 1
## Levels: 0 1
```

- Prévision des *probabilités* :

```
> predict(arbre_final, newdata=xnew, type="prob")
##          0          1
## 1 0.046875 0.953125
```

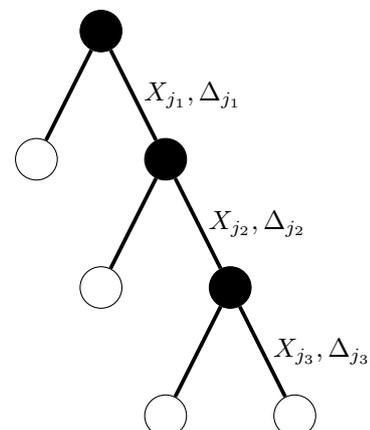
## 1.4 Importance des variables

- La *visualisation de l'arbre* peut donner une idée sur l'*importance des variables* dans l'algorithme.
- *Pas suffisant!* Il se peut en effet que des variables possèdent une grande importance sans pour autant apparaître explicitement dans l'arbre!
  - Difficile de *quantifier l'importance* juste en regardant l'arbre!
  - Il se peut en effet que des variables possèdent une grande importance *sans pour autant apparaître en haut* de l'arbre!

### Mesure d'importance d'un arbre

Basée sur le *gain d'impureté* des nœuds internes.

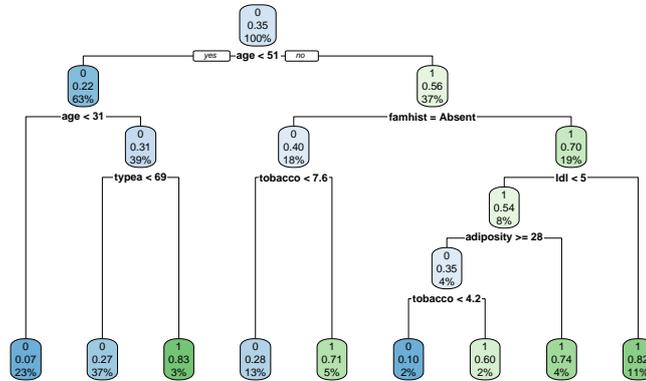
- Nœuds internes  $\implies N_t, t = 1, \dots, J - 1$ ;
- Variables de coupure  $\implies X_{j_t}$ ;
- Gain d'impureté  $\implies i_{j_t}^2$ .



## Mesure d'impureté de la variable $\ell$

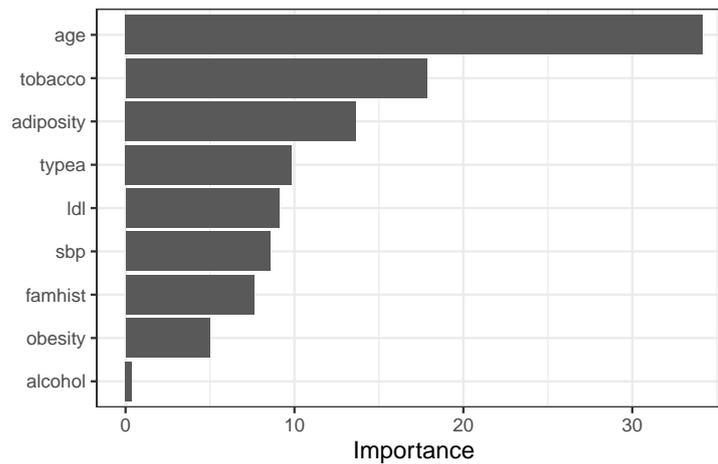
$$\mathcal{I}_\ell(T) = \sum_{t=1}^{|T|-1} \Delta_t \mathbf{1}_{j_t=\ell}.$$

## Exemple



— Visualisation des *importance* à l'aide de **vip** :

```
> library(vip)
> vip(arbre)
```



## Bilan

### 1. *Avantages* :

- Méthode « simple » relativement facile à mettre en œuvre.
- Fonctionne en **régression** et en **classification**.
- Résultats **interprétables** (à condition que l'arbre ne soit pas trop profond).

### 2. *Inconvénients* :

- Performances prédictives **limitées**.
- méthode connue pour être **instable**, sensible à de légères perturbations de l'échantillon.  $\implies$  Cet inconvénient sera un avantage pour des **agrégations bootstrap**  $\implies$  **forêts aléatoires**.

## 2 Réseaux de neurones

### 2.1 Introduction

#### Bibliographie

- *Wikistat* : Neural networks and introduction to deep learning
- *Eric Rakotomalala* : Deep learning : Tensorflow et Keras sous R
- *Rstudio* : R interface to Keras

#### Historique

- Modélisation du *neurone formel* [McCulloch and Pitts, 1943].
- Concept mis en *réseau* avec *une couche d'entrée et une sortie* [Rosenblatt, 1958].
  - Origine du *perceptron*
  - Approche *connexioniste* (atteint ses limites technologiques et théoriques au début des années 70)
- Relance de l'approche connexioniste au début des années 80 avec l'essor technologique et quelques avancées théoriques
- Estimation du *gradient* par *rétro-propagation de l'erreur* [Rumelhart et al., 1986].
- Développement considérable (au début des années 90)
- Remis en veilleuse au milieu des années 90 au profit d'*autres algorithmes d'apprentissage* : boosting, support vector machine...
- Regain d'intérêt dans les années 2010, énorme battage médiatique sous l'appellation d'*apprentissage profond/deep learning*.
- Résultats *spectaculaires* obtenus par ces réseaux en *reconnaissance d'images*, traitement du *langage naturel*...

#### Différentes architectures

Il existe *différents types* de réseaux neuronaux :

- *perceptron multicouches* : les plus anciens et les plus simples ;
- *réseaux de convolution* : particulièrement efficaces pour le traitement d'images ;
- *réseaux récurrents* : adaptés à des données séquentielles (données textuelles, séries temporelles).

#### *Dans cette partie*

nous nous intéresserons uniquement au *perceptron multicouches*.

#### Neurone : vision biologique



### Définition : neurone biologique

Un neurone biologique est une cellule qui se caractérise par

- *des synapses* : les points de connexion avec les autres neurones ;
- *des dendrites* : entrées du neurones ;
- les *axones* ou *sorties* du neurone vers d'autres neurones ;
- le *noyau* qui active les sorties.

### Définition : neurone formel

Un *neurone formel* est un modèle qui se caractérise par

- des entrées  $x_1, \dots, x_p$  ;
- des poids  $w_0, w_1, \dots, w_p$  ;
- une fonction d'activation  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  ;
- une sortie :

$$\hat{y} = \sigma(w_0 + w_1x_1 + \dots + x_px_p).$$

## 2.2 Le perceptron simple

- *Le problème* : expliquer une sortie  $y \in \mathbb{R}$  par des entrées  $x = (x_1, \dots, x_p)$ .

### Définition

Le *perceptron simple* est une fonction  $f$  des entrées  $x$

- pondérées par un vecteur  $w = (w_1, \dots, w_p)$ ,
- complétées par un neurone de biais  $w_0$ ,
- et une fonction d'activation  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$

$$\hat{y} = f(x) = \sigma(w_0 + w_1x_1 + \dots + x_px_p).$$

### Fonction d'activation

Plusieurs fonctions d'activation peuvent être utilisées :

- **Identité** :  $\sigma(x) = x$  ;
- **sigmoïde** ou **logistique** :  $\sigma(x) = 1/(1 + \exp(-x))$  ;
- **seuil** :  $\sigma(x) = \mathbf{1}_{x \geq 0}$  ;
- **ReLU** (Rectified Linear Unit) :  $\sigma(x) = \max(x, 0)$  ;
- **Radiale** :  $\sigma(x) = \sqrt{1/2\pi} \exp(-x^2/2)$ .

### Remarque

Les poids  $w_j$  sont estimés à partir des données (voir plus loin).

## Représentation graphique

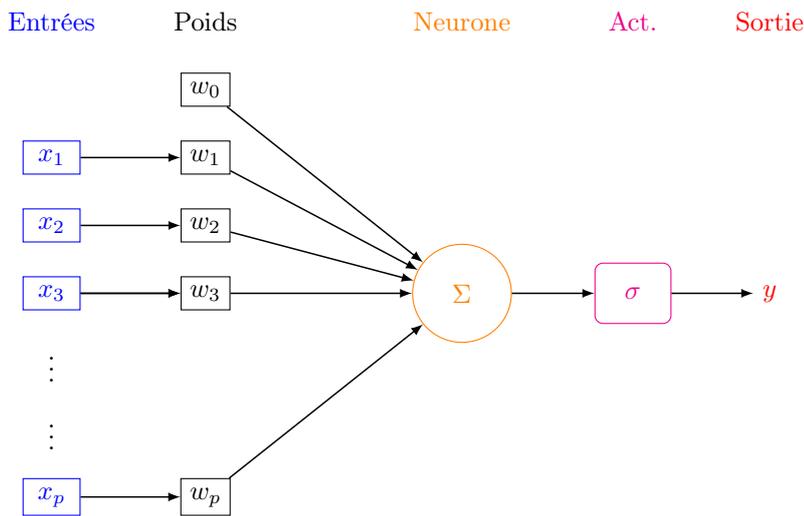
### Le coin R

- Plusieurs *packages R* permettent d'ajuster des réseaux de neurones : *nnet*, *deepnet*...
- Nous présentons ici le package *keras*, initialement programmé en Python et qui a été "traduit" récemment en R.

```
> library(keras)
> install_keras()
```

### Exemple

- On veut expliquer *une variable Y* binaire par 4 variables d'entrées  $X_1, \dots, X_4$ .
- On dispose d'un *échantillon d'apprentissage* de taille 300 :



```
> head(dapp)
##          X1          X2          X3          X4 Y
## 1  0.5855288 -1.4203239  1.67751179 -0.1746226 1
## 2  0.7094660 -2.4669386  0.07947405 -0.6706167 1
## 3 -0.1093033  0.4847158 -0.85642750  0.5074258 0
## 4 -0.4534972 -0.9379723 -0.77877729  1.2474343 0
## 5  0.6058875  3.3307333 -0.38093608 -1.2482755 1
## 6 -1.8179560 -0.1629455 -1.89735834 -1.9347187 1
```

### Définition du modèle

— Elle s'effectue à l'aide des fonctions `keras_model_sequential` et `layer_dense`.

```
> model <- keras_model_sequential()
> model %>% layer_dense(units=1,input_shape=c(4),
+                       activation="sigmoid")
```

- `units` : nombre de neurones souhaités ;
- `activation` : choix de la fonction d'activation.

### Summary

— Un `summary` du modèle permet de visualiser le nombre de paramètres à estimer.

```
> summary(model)
## -----
## Layer (type)                Output Shape          Param #
## -----
## dense (Dense)                (None, 1)              5
## -----
## Total params: 5
## Trainable params: 5
## Non-trainable params: 0
## -----
```

### Estimation des paramètres

— On indique dans la fonction `compile` la fonction de perte pour l'estimation des paramètres du modèle et le critère de performance

```
> model %>% compile(
+   loss="binary_crossentropy",
+   optimizer="adam",
+   metrics="accuracy"
+ )
```

## Estimation

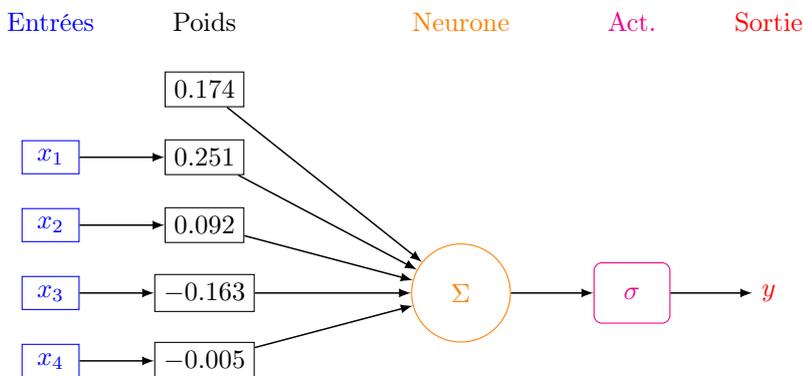
- On utilise la fonction *fit* pour entrainer le modèle

```
> Xtrain <- as.matrix(dapp[,1:4])
> Ytrain <- dapp$Y
> model %>% fit(x=Xtrain,y=Ytrain,epochs=300,batch_size=5)
```

- Et on obtient les poids avec *get\_weights* :

```
> W <- get_weights(model)
> W
## [[1]]
##           [,1]
## [1,]  0.250867128
## [2,]  0.092339918
## [3,] -0.162947521
## [4,] -0.005261241
##
## [[2]]
## [1] 0.1739036
```

## Visualisation du réseau



### Estimation

$$\hat{P}(Y = 1|X = x) = \frac{1}{1 + \exp(-(0.174 + 0.251x_1 + \dots - 0.005x_4))}$$

### Prévision

- On calcule la *prévision de la probabilité*  $P(Y = 1|X = x)$  pour le premier individu de l'échantillon test :

```
> w <- W[[1]]
> w0 <- W[[2]]
> Xtest <- as.matrix(dtest[,1:4])
> sc1 <- w0+sum(w*Xtest[1,])
> 1/(1+exp(-sc1))
## [1] 0.6209704
```

- que l'on retrouve avec *predict\_proba* :

```
> prev <- model %>% predict_proba(Xtest)
> prev[1]
## [1] 0.6209704
```

## 2.3 Perceptron multicouches

### Constat

- *Règle de classification* : le *perceptron simple* affecte un individu dans le groupe 1 si

$$\mathbf{P}(Y = 1|X = x) \geq 0.5 \iff w_0 + w_1x_1 + \dots + w_px_p \geq 0.$$

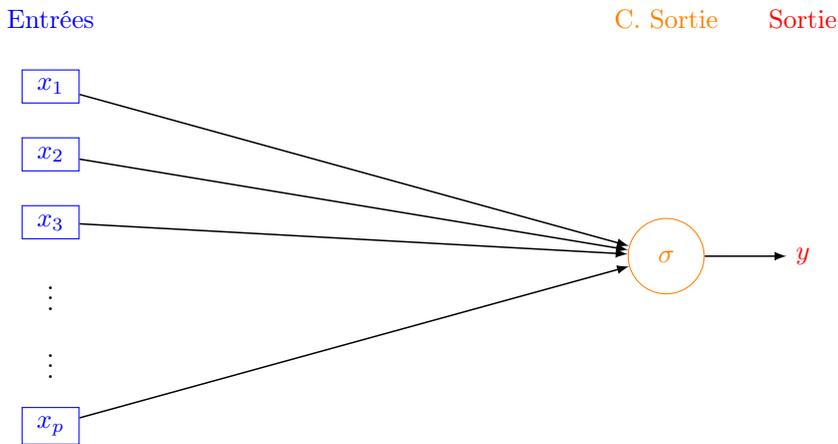
- Il s'agit donc d'une *règle linéaire*.

⇒ *peu efficace* pour représenter des **phénomènes "complexes"**.

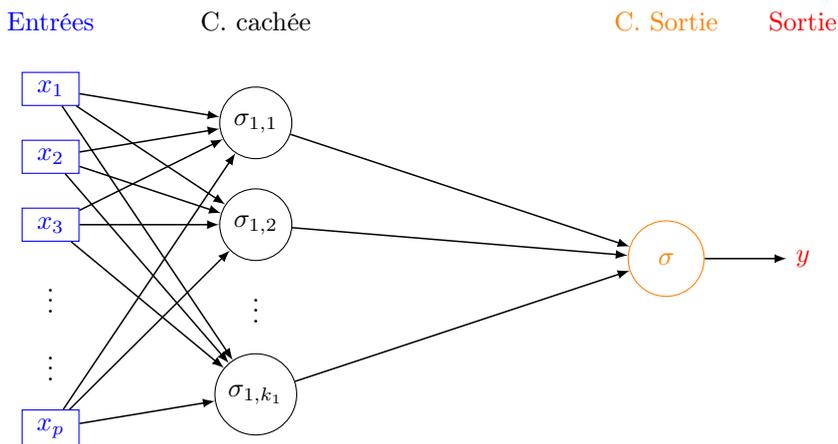
### Idée

Conserver cette structure de réseau en considérant **plusieurs couches** de **plusieurs neurones**.

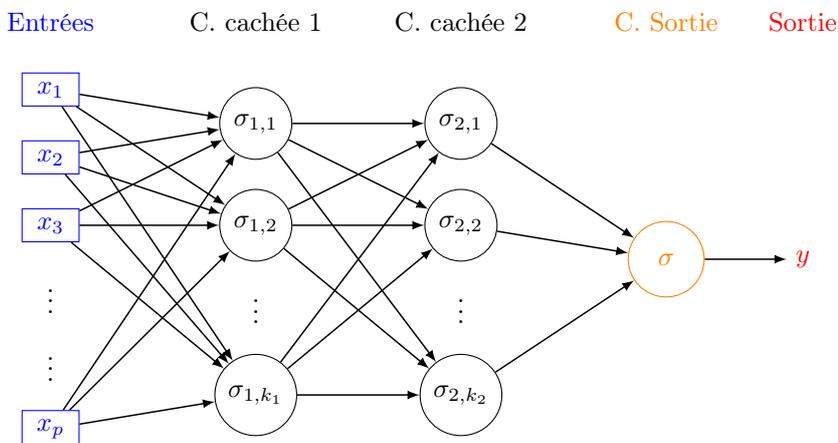
### Perceptron simple



### Une couche cachée



### Deux couches cachées



## Commentaires

- Les neurones de la *première couche (cachée)* calculent des *combinaisons linéaires des entrées*.
- Ces combinaisons linéaires sont ensuite *activées par une fonction d'activation*, produisant *une sortie par neurone*.
- Chaque neurone de la *deuxième couche (cachée)* est une combinaison linéaire des *sorties de la couche précédente*...
- *activées par une fonction d'activation*, produisant *une sortie par neurone*...

## Remarque

Le nombre de neurones dans la *couche finale* est définie par la *dimension de la sortie  $y$*  :

- Régression ou classification binaire  $\implies$  1 neurone.
- Classification multiclasse ( $K$ )  $\implies$   $K$  (ou  $K - 1$ ) neurones.

## Le coin R

- L'ajout de couches cachées dans *keras* est relativement simple.
- Il suffit de définir ces couches au moment de la spécification du modèle.
- Par exemple, pour *deux couches cachées* avec 10 et 5 neurones, on utilisera :

```
> model <- keras_model_sequential()
> model %>% layer_dense(units=10,input_shape=c(4),activation="sigmoid") %>%
+   layer_dense(units=5,activation="sigmoid") %>%
+   layer_dense(units=1,activation="sigmoid")
```

```
> summary(model)
## -----
## Layer (type)                Output Shape          Param #
## -----
## dense_1 (Dense)             (None, 10)            50
## -----
## dense_2 (Dense)             (None, 5)             55
## -----
## dense_3 (Dense)             (None, 1)             6
## -----
## Total params: 111
## Trainable params: 111
## Non-trainable params: 0
## -----
```

## 2.4 Estimation

- L'*utilisateur* doit choisir le *nombre de couches*, le *nombre de neurones par couche*, les *fonctions d'activation* de chaque neurone.
- Une fois ces paramètres choisis, il faut *calculer (estimer)* tous les *vecteurs de poids dans tous les neurones*.

## L'approche

- On désigne par  $\theta$  l'ensemble des *paramètres* à estimer  $\implies f(x, \theta)$  la *règle* associée au réseau.
- *Minimisation de risque empirique* : minimiser

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

où  $\ell$  est une *fonction de perte* (classique).

## Fonctions de perte

- *Erreur quadratique* (régression) :

$$\ell(y, f(x)) = (y - f(x))^2.$$

- *Cross-entropy* ou *log-vraisemblance négative* (classification binaire 0/1) :

$$\ell(y, p(x)) = -(y \log(p(x)) + (1 - y) \log(1 - p(x)))$$

où  $p(x) = \mathbf{P}(Y = 1|X = x)$ .

- *Cross-entropy* ou *log-vraisemblance négative* (classification multi-classes) :

$$\ell(y, p(x)) = - \sum_{k=1}^K \mathbf{1}_{y=k} \log(p_k(x))$$

où  $p_k(x) = \mathbf{P}(Y = k|X = x)$ .

## Descente de gradient

- La solution s'obtient à l'aide de méthodes de type *descente de gradient* :

$$\theta^{\text{new}} = \theta^{\text{old}} - \varepsilon \nabla_{\theta} \mathcal{R}_n(\theta^{\text{old}}).$$

- Le réseau étant *structuré en couches*, la mise à jour des paramètres *n'est pas directe*.

### Algorithme de rétropropagation (voir *ici*)

1. **Etape forward** : calculer tous les poids associés à  $\theta^{\text{old}}$  et stocker toutes les valeurs intermédiaires.
2. **Etape backward** :
  - (a) Calculer le gradient dans la couche de sortie.
  - (b) En déduire les gradients des couches cachées.

## Batch et epoch

- L'algorithme de rétropropagation n'est généralement **pas appliqué sur l'ensemble des données**, mais sur des sous-ensemble de cardinaux  $m$  appelés *batch*.
- Cette approche est classique sur les *gros volumes de données* et permet de prendre en compte des **données séquentielles**.
- Pour prendre en compte *toutes les données* sur une étape de la descente de gradient, on va donc appliquer  **$n/m$  fois l'algorithme de rétropropagation**.
- Une itération sur l'ensemble des données est appelée *epoch*.

## Algorithme de rétropropagation stochastique

### Algorithme

**Entrées** :  $\varepsilon$  (learning rate),  $m$  (taille des batches), nb (nombre d'epochs).

1. Pour  $\ell = 1$  à nb
2. Partitionner aléatoire les données en  $n/m$  batch de taille  $m \implies B_1, \dots, B_{n/m}$ .
  - (a) Pour  $j = 1$  à  $n/m$ 
    - i. Calculer les gradients sur le batch  $j$  avec l'algorithme de *rétropropagation* :  $\nabla_{\theta}$ .
    - ii. Mettre à jour les paramètres

$$\theta^{\text{new}} = \theta^{\text{old}} - \varepsilon \nabla_{\theta^{\text{old}}}.$$

**Sorties** :  $\theta^{\text{new}}$  et  $f(x, \theta^{\text{new}})$ .

## Choix des paramètres

- $\varepsilon$  (pas de la descente de gradient), généralement petit. Existence de *versions améliorées* de l'algorithme précédent moins sensible à ce paramètre (**RMSProp**, **Adam...**).
- $m$  (taille des batch) : généralement petit (pas trop en fonction du temps de calcul). L'utilisateur peut (doit) faire *plusieurs essais*.
- $nb$  (nombre d'époch), proche du nombre d'itérations en boosting  $\implies$  risque de *surapprentissage* si trop grand.

## En pratique

Il est courant de visualiser l'évolution de la *fonction de perte* et/ou d'un *critère de performance* en fonction du *nombre d'époch*.

## Un exemple

- On considère un réseau à *2 couches cachées* comportant *50 nœuds* (2851 paramètres).

```
> model1 <- keras_model_sequential()
> model1 %>% layer_dense(units=50,input_shape=c(4),
+                       activation="sigmoid") %>%
+   layer_dense(units = 50,activation = "sigmoid") %>%
+   layer_dense(units = 1,activation = "sigmoid")
```

- On utilise
  - *crossentropy* comme perte.
  - *Adam* comme algorithme d'optimisation.
  - *accuracy* (taux de bien classés) comme mesure de performance.

```
> model1 %>% compile(
+   loss="binary_crossentropy",
+   optimizer="adam",
+   metrics="accuracy"
+ )
```

- On *estime les paramètres* avec  $m = 5$  et  $nb = 1000$  et utilise 20% des données dans l'échantillon de validation.

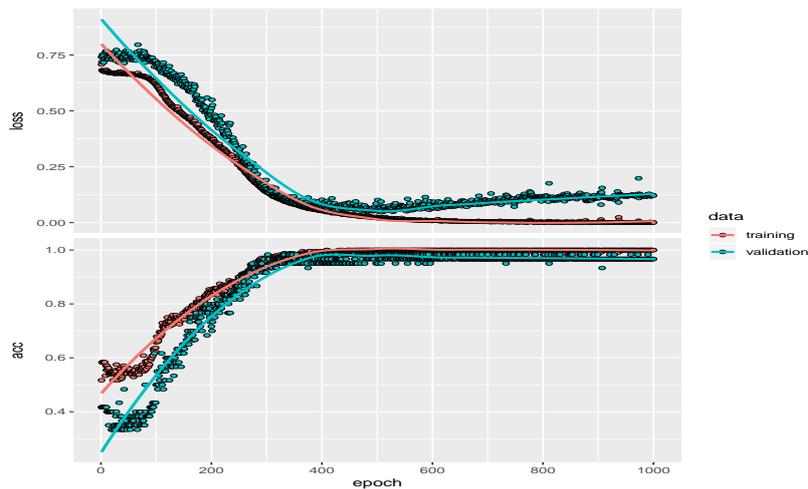
```
> history <- model1 %>% fit(
+   x=Xtrain,
+   y=Ytrain,
+   epochs=1000,
+   batch_size=5,
+   validation_split=0.2
+ )
```

## Erreur et perte

```
> plot(history)
```

- On compare ce *nouveau réseau* avec le *perceptron simple* construit précédemment.

```
> Xtest <- as.matrix(dtest[,1:4])
> Ytest <- dtest$Y
> model %>% evaluate(Xtest,Ytest)
## $loss
## [1] 0.7259337
##
## $acc
## [1] 0.39
> model1 %>% evaluate(Xtest,Ytest)
## $loss
## [1] 0.3290039
##
## $acc
## [1] 0.935
```



### Nombre de couches et de neurones

- A choisir par *l'utilisateur*.
- Il est généralement mieux d'en avoir *trop que pas assez*  $\implies$  plus "facile" de capter des **non linéarités complexes** avec beaucoup de couches et de neurones.
- On fait généralement plusieurs essais que l'on compare (avec *caret* par exemple).
- Voir par exemple l'appli suivante :

<http://playground.tensorflow.org/>

## 2.5 Choix des paramètres et surapprentissage

### Surapprentissage

- *Plusieurs paramètres* peuvent causer du *surapprentissage*, notamment les nombres de **couches cachées**, de **neurones** et **d'époch**.

### Plusieurs solutions

1. *Régularisation* de type ridge/lasso :

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

$\implies$  ajouter *kernel\_regularizer = regularizer\_l2(l = 0.001)* dans la fonction *layer\_dense* par exemple.

2. *Early stopping* : on stoppe l'algorithme lorsque l'ajout d'époch n'améliore pas suffisamment un critère donné.
3. *Dropout* : suppression (aléatoire) de certains neurones dans les couches  $\implies$  souvent la **solution privilégiée**.

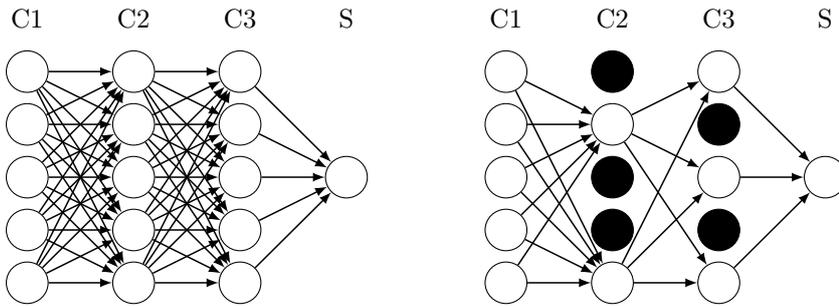
### Dropout

- A chaque étape de la phase d'entraînement, on *supprime un nombre de neurones* (selon une Bernoulli de paramètre  $p$ ).

### Le coin R

- Il suffit d'ajouter *layer\_dropout* après les **couches cachées**.

```
> model3 <- keras_model_sequential()
> model3 %>% layer_dense(units=50,input_shape=c(4),activation="sigmoid") %>%
+   layer_dropout(0.5) %>%
+   layer_dense(units = 50,activation = "sigmoid") %>%
+   layer_dropout(0.5) %>%
+   layer_dense(units = 1,activation = "sigmoid")
```



## Sélection avec caret

- On peut sélectionner la plupart des paramètres avec *caret*.
- On propose par exemple, pour un réseau avec une couche cachée, de choisir
  1. le nombre de **neurones dans la couche cachée** parmi 10, 50, 100
  2. la **fonction d'activation** : sigmoïde ou relu.
- On définit d'abord les *paramètres du modèle*

```
> library(caret)
> dapp1 <- dapp
> dapp1$Y <- as.factor(dapp1$Y)
> param_grid <- expand_grid(size=c(10,50,100),
+                           lambda=0,batch_size=5,lr=0.001,
+                           rho=0.9,decay=0,
+                           activation=c("relu","sigmoid"))
```

- On calcule ensuite les taux de bien classés par *validation croisée 5 blocs* pour chaque combinaison de paramètres.

```
> caret_mlp <- train(Y~.,data=dapp1,method="mlpKerasDecay",
+                   tuneGrid=param_grid,epoch=500,verbose=0,
+                   trControl=trainControl(method="cv",number=5))
```

```
> caret_mlp
## Multilayer Perceptron Network with Weight Decay
## 300 samples
## 4 predictor
## 2 classes: '0', '1'
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 240, 240, 240, 240
## Resampling results across tuning parameters:
## size activation Accuracy Kappa
## 10 relu 0.9200000 0.8394122
## 10 sigmoid 0.8966667 0.7913512
## 50 relu 0.9266667 0.8515286
## 50 sigmoid 0.9066667 0.8127427
## 100 relu 0.9366667 0.8722974
## 100 sigmoid 0.9300000 0.8595025
## Tuning parameter 'lambda' was held constant at a value of 0
## Tuning parameter 'rho' was held constant at a value of 0.9
## Tuning parameter 'decay' was held constant at a value of 0
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were size = 100, lambda =
## 0, batch_size = 5, lr = 0.001, rho = 0.9, decay = 0 and activation = relu.
```

## Conclusion

- *Avantages* :
  - Méthode connue pour être efficace pour (quasiment) tous les problèmes.
  - Plus particulièrement sur des **architectures particulières** : *images*, *données textuelles*.

— *Inconvénients* :

- Gain **plus discutable** sur des problèmes standards.
- (Beaucoup) plus **difficile à calibrer** que les autres algorithmes ML.
- Niveau d'expertise important.

## 3 Bibliographie

### Références

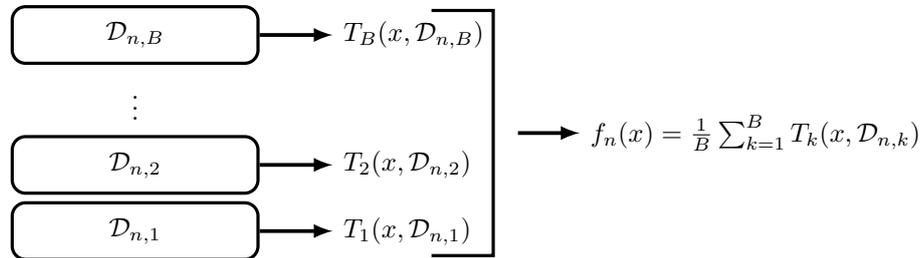
#### Biblio3

- [Breiman et al., 1984] Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and regression trees*. Wadsworth & Brooks.
- [McCulloch and Pitts, 1943] McCulloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5 :115–133.
- [Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron : a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65 :386–408.
- [Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and R. J. Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, pages 533–536.

# Quatrième partie

## 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. ...

## 1 Bagging et forêts aléatoires

### Cadre

- Idem que précédemment, on cherche à *expliquer* une variable  $Y$  par  $d$  variables explicatives  $X_1, \dots, X_d$ .
- Pour simplifier on se place en *régression* :  $Y$  est à valeurs dans  $\mathbb{R}$  mais tout ce qui va être fait s'étant directement à la *classification binaire ou multiclassées*.

### Notations :

- $(X, Y)$  un couple aléatoire à valeurs dans  $\mathbb{R}^d \times \mathbb{R}$ .
- $\mathcal{D}_n = (X_1, Y_1), \dots, (X_n, Y_n)$  un  $n$ -échantillon i.i.d. de même loi que  $(X, Y)$ .
- Un algorithme de la forme :

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

- *Hypothèse* : les  $T_1, \dots, T_b$  sont **identiquement distribuées**.

### Propriété

$$\mathbf{E}[f_n(x)] = \mathbf{E}[T_1(x)] \quad \text{et} \quad \mathbf{V}[f_n(x)] = \rho(x)\mathbf{V}[T_1(x)] + \frac{1 - \rho(x)}{B}\mathbf{V}[T_1(x)]$$

où  $\rho(x) = \text{corr}(T_1(x), T_2(x))$ .

### Conséquence

- **Biais** non modifié.
- **Variance**  $\searrow$  si  $B \nearrow$  et  $\rho(x) \searrow$ .
- Ajuster le *même algorithme* sur les **mêmes données** n'est d'aucun intérêt.
- Ajuster le *même algorithme* sur des **sous-échantillons disjoints** est d'un intérêt limité.
- Utiliser un *grand nombre d'algorithmes différents* différent est compliqué...

### Idée

Ajuster le même algorithme sur des **échantillons bootstraps**.

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

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	$T_B$

## 1.1 Bagging

- 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* :
- Echantillons *bootstrap* : tirage de taille  $n$  avec remise
- A la fin, on *agrège* :

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

### Algorithme bagging

#### Entrées :

- $B$  un entier positif ;
- $T$  un algorithme de prévision.

Pour  $b$  entre 1 et  $B$  :

1. Faire un tirage aléatoire avec remise de taille  $n$  dans  $\{1, \dots, n\}$ . On note  $\theta_b$  l'ensemble des indices sélectionnés et  $\mathcal{D}_{n,b}^* = \{(x_i, y_i), i \in \theta_b\}$  l'échantillon bootstrap associé.
2. Entraîner l'algorithme  $T$  sur  $\mathcal{D}_{n,b}^* \implies T(\cdot, \theta_b, \mathcal{D}_n)$ .

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

### Un algorithme pas forcément aléatoire

- L'*aléa bootstrap* implique que l'algorithme "change" lorsqu'on l'exécute plusieurs fois mais...

$$\lim_{B \rightarrow +\infty} \frac{1}{B} \sum_{b=1}^B T(x, \theta_b, \mathcal{D}_n) = \mathbf{E}_\theta [T(x, \theta, \mathcal{D}_n)] = \bar{f}_n(x, \mathcal{D}_n)$$

### Conséquence

- L'algorithme se *stabilise* (converge) lorsque  $B \nearrow$ .
- Recommandation : choisir  $B$  le *plus grand possible*.

## Choix de $T$

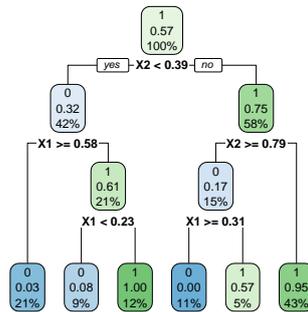
$$\mathbf{E}[f_n(x)] = \mathbf{E}[T_1(x)] \quad \text{et} \quad \mathbf{V}[f_n(x)] = \rho(x)\mathbf{V}[T_1(x)] + \frac{1 - \rho(x)}{B}\mathbf{V}[T_1(x)].$$

### Conclusion

- Bagging ne modifie pas le biais.
- $B$  grand  $\implies \mathbf{V}[f_n(x)] \approx \rho(x)\mathbf{V}[T_1(x)] \implies$  la variance diminue d'autant plus que la corrélation entre les prédicteurs diminue.
- Il est donc nécessaire d'agréger des estimateurs sensibles à de légères perturbations de l'échantillon.
- Les arbres sont connus pour posséder de telles propriétés.

## 1.2 Forêts aléatoires

### Rappels sur les arbres



### Complexité

Profondeur

- petite : biais  $\nearrow$ , variance  $\searrow$
- grande : biais  $\searrow$ , variance  $\nearrow$  (sur-apprentissage).

### Définition

- Comme son nom l'indique, une *forêt aléatoire* est définie à partir d'un ensemble d'arbres.

### Définition

Soit  $T_k(x), k = 1, \dots, B$  des prédicteurs par arbre ( $T_k : \mathbb{R}^d \rightarrow \mathbb{R}$ ). Le prédicteur des *forêts aléatoires* est obtenu par agrégation de cette collection d'arbres :

$$f_n(x) = \frac{1}{B} \sum_{k=1}^B T_k(x).$$

### Forêts aléatoires

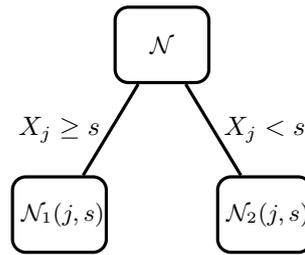
- Forêts aléatoires = *collection d'arbres*.
- Les forêts aléatoires les plus utilisées sont (de loin) celles proposées par *Léo Breiman* (au début des années 2000).
- Elles consistent à *agréger* des arbres construits sur des *échantillons bootstrap*.
- On pourra trouver de la doc à l'url

<http://www.stat.berkeley.edu/~breiman/RandomForests/>

et consulter la thèse de Robin Genuer [Genuer, 2010].

### 1.2.1 Algorithme

#### Coupures "aléatoires"



#### Arbres pour forêt

- Breiman propose de sélectionner la "meilleure" variable dans un ensemble composé **uniquement de mtry 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 :
  - (a) Choisir `mtry` variables au hasard parmi les `d` variables explicatives;
  - (b) Sélectionner la meilleure coupure  $X_j \leq s$  en ne considérant que les `mtry` variables sélectionnées;
  - (c) 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** :  $f_n(x) = \frac{1}{B} \sum_{b=1}^B T(x, \theta_b, \mathcal{D}_n)$ .

#### Type de prévision

La prévision dépend de la **nature de Y** et de ce que l'on souhaite **estimer**

- **Régression** :  $T(x, \theta_b, \mathcal{D}_n) \in \mathbb{R}$  et

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

- **Classification** (classe) :  $T(x, \theta_b, \mathcal{D}_n) \in \{1, \dots, K\}$  et

$$g_n(x) \in \operatorname{argmax}_{k \in \{1, \dots, K\}} \sum_{b=1}^B \mathbf{1}_{T(x, \theta_b, \mathcal{D}_n)=k}, \quad k = 1, \dots, K.$$

- **Classification** (proba) :  $T_k(x, \theta_b, \mathcal{D}_n) \in [0, 1]$  et

$$S_{n,k}(x) = \frac{1}{B} \sum_{b=1}^B T_k(x, \theta_b, \mathcal{D}_n), \quad k = 1, \dots, K.$$

#### 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 %

```

### 1.2.2 Choix des paramètres

- $B$  réglé  $\implies$  le plus grand possible. En pratique on pourra s'assurer que le **courbe d'erreur** en fonction du nombre d'arbres est **stabilisée**.
- Pour les autres paramètres on étudie à nouveau :

$$\mathbf{E}[f_n(x)] = \mathbf{E}[T_1(x)] \quad \text{et} \quad \mathbf{V}[f_n(x)] = \rho(x)\mathbf{V}[T_1(x)] + \frac{1 - \rho(x)}{B}\mathbf{V}[T_1(x)].$$

#### Conséquence

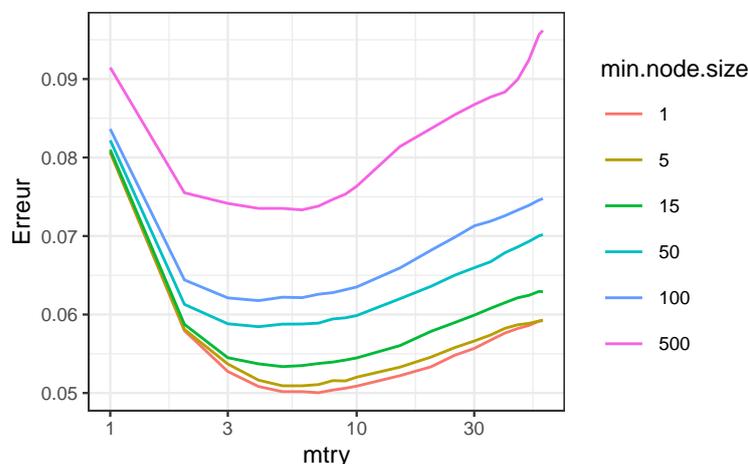
- Le biais n'étant pas amélioré par "l'agrégation bagging", il est recommandé d'agréger des estimateurs qui possèdent un **biais faible** (contrairement au boosting).
- Arbres "*profonds*", *peu d'observations dans les nœuds terminaux*.
- Par défaut dans **randomForest**, `min.node.size = 5` en régression et 1 en classification.

#### Choix de mtry

- Il est en *relation avec la corrélation* entre les arbres  $\rho(x)$ .
- Ce paramètre a une influence sur le *compromis biais/variance* de la forêt.
- $mtry \searrow$ 
  1. tendance à se rapprocher d'un *choix "aléatoire"* des variables de découpe des arbres  $\implies$  les arbres sont de plus en plus différents  $\implies \rho(x) \searrow \implies$  **la variance de la forêt diminue**.
  2. mais... le biais des arbres  $\nearrow \implies$  le **biais de la forêt**  $\nearrow$ .
- Inversement lorsque  $mtry \nearrow$  (risque de **sur-ajustement**).

#### Conclusion

- Il est recommandé de comparer les performances de la forêt pour **plusieurs valeurs de mtry**.
- Par défaut  $mtry = d/3$  en régression et  $\sqrt{d}$  en classification.
- Visualisation d'erreur en fonction de `min.node.size` et `mtry`



## Commentaires

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

## En pratique

- 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).

5. Ajustement de l'algorithme final :

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

### 1.2.3 Erreur OOB et importance des variables

- Comme pour tous les algorithmes de prévision on peut évaluer la *performance des forêts aléatoires* en estimant un *risque par ré-échantillonnage*.
- Les tirages bootstraps permettent de définir une alternative, souvent *moins couteuse en temps de calcul*, au ré-échantillonnage.
- *Idée/astuce* : utiliser les observations non sélectionnées dans les échantillons bootstraps pour estimer le risque.

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	$T_5$
7	10	3	4	9	10	10	8	6	1	$T_6$

### OOB illustration

- Les échantillons 2, 3 et 5 *ne contiennent pas* la première observation, donc

$$\hat{y}_1 = \frac{1}{3}(T_2(x_1) + T_3(x_1) + T_5(x_1)).$$

- On fait de même pour *toutes les observations*  $\implies \hat{y}_2, \dots, \hat{y}_n$ .
- On *calcule l'erreur* selon

$$\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad \text{ou} \quad \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{\hat{y}_i \neq y_i}.$$

### OOB définition

- Pour  $i = 1, \dots, n$  on note

$$\text{OOB}(i) = \{b \leq B : i \notin \mathcal{I}_b\}$$

l'ensemble des tirages bootstrap qui *ne contiennent pas*  $i$  et

$$f_{n, \text{OOB}(i)}(x_i) = \frac{1}{|\text{OOB}(i)|} \sum_{b \in \text{OOB}(i)} T(x_i, \theta_b, \mathcal{D}_n)$$

la prévision de la forêt en ne considérant *que les arbres pour lesquels  $i$  n'est pas dans le tirage bootstrap*.

- L'*erreur OOB* s'obtient en confrontant ces prévisions aux valeurs observées, par exemple

$$\frac{1}{n} \sum_{i=1}^n (y_i - f_{n, \text{OOB}(i)}(x_i))^2 \quad \text{ou} \quad \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{f_{n, \text{OOB}(i)}(x_i) \neq y_i}.$$

$\implies$  erreur renvoyée par défaut dans **ranger** et **randomforest**.

### Importance des variables

Deux mesures sont généralement utilisées.

- *Score d'impureté* : simplement la moyenne des importances de  $X_j$  dans chaque arbre de la forêt :

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

voir chapitre sur les arbres pour la définition de  $\mathcal{I}_j(T_b)$ .

- *Importance par permutation* : comparer les erreurs de chaque arbre sur l'échantillon

1. OOB de l'arbre ;
2. OOB en permutant les valeurs de la variables  $j$ .

$\implies$  *Idée* : Si  $X_j$  est importante ces erreurs doivent être très différentes.

### Importance par permutation

- On présente ce score en régression mais rien ne change pour la classification.
- On note

$$\text{Err}(\text{OOB}_b) = \frac{1}{|\text{OOB}_b|} \sum_{i \in \text{OOB}_b} (y_i - T(x_i, \theta_b, \mathcal{D}_n))^2,$$

avec

$$\text{OOB}_b = \{i \leq n : i \notin \mathcal{I}_b\}.$$

$\implies$  Erreur de l'*arbre  $b$*  calculée sur les *données OOB*.

- On recalcule cette erreur mais sur  $\text{OOB}_b$  où on permute les valeurs de la  $j^{\text{e}}$  colonne.

$$\begin{bmatrix} x_{11} & \dots & x_{1j} & \dots & x_{1d} \\ x_{21} & \dots & x_{2j} & \dots & x_{2d} \\ x_{51} & \dots & x_{3j} & \dots & x_{3d} \\ x_{41} & \dots & x_{4j} & \dots & x_{4d} \\ x_{51} & \dots & x_{5j} & \dots & x_{5d} \end{bmatrix} \implies \begin{bmatrix} x_{11} & \dots & x_{3j} & \dots & x_{1d} \\ x_{21} & \dots & x_{5j} & \dots & x_{2d} \\ x_{51} & \dots & x_{1j} & \dots & x_{3d} \\ x_{41} & \dots & x_{2j} & \dots & x_{4d} \\ x_{51} & \dots & x_{4j} & \dots & x_{5d} \end{bmatrix}$$

- On note  $\tilde{x}_i^j$  les individus de l'échantillon  $\text{OOB}_b$  permuté et on calcule

$$\text{Err}(\text{OOB}_b^j) = \frac{1}{|\text{OOB}_b|} \sum_{i \in \text{OOB}_b} (y_i - T(\tilde{x}_i^j, \theta_b, \mathcal{D}_n))^2.$$

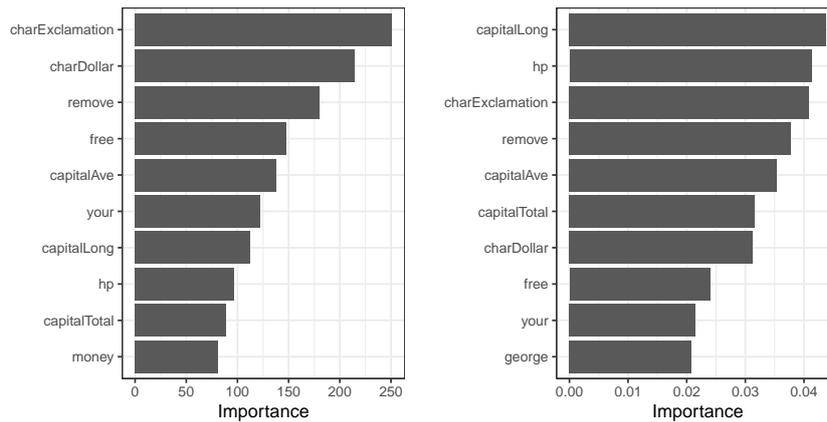
### Importance par permutation

$$\mathcal{I}_j^{\text{perm}} = \frac{1}{B} \sum_{b=1}^B (\text{Err}(\text{OOB}_b^j) - \text{Err}(\text{OOB}_b)).$$

### Le coin R

- On peut *calculer et visualiser* facilement ces importances avec `ranger` :

```
> set.seed(1234)
> foret.imp <- ranger(type~., data=spam, importance="impurity")
> foret.perm <- ranger(type~., data=spam, importance="permutation")
> vip(foret.imp); vip(foret.perm)
```



### 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...)

## 2 Boosting

- 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.

### 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  $\implies \mathcal{R}_n(f) \searrow \implies f(x_i) \approx y_i, i = 1, \dots, n \implies$  *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.

- 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.

## 2.1 Algorithme de gradient boosting

### 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

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

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

### Nouveau problème

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

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

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

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

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

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

- (d) 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` :

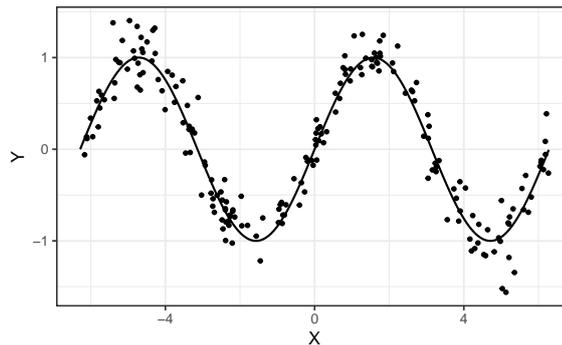
- $\ell$  la fonction de perte  $\implies$  `distribution`
- $B$  nombre d'itérations  $\implies$  `n.tree`
- $J$  le nombre de feuilles des arbres  $\implies$  `interaction.dept` ( $=J - 1$ )
- $\lambda$  le paramètre de rétrécissement  $\implies$  `shrinkage`.

## 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

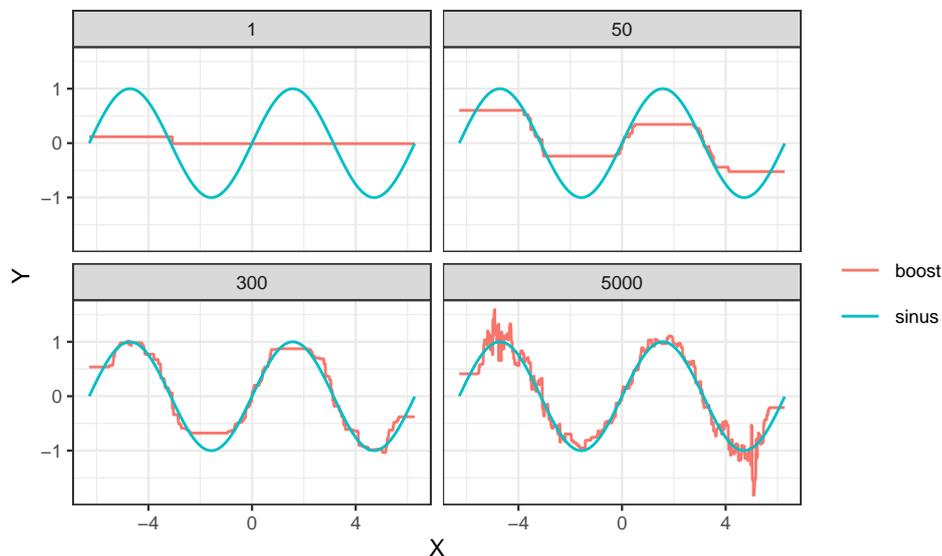
- Données `sinus`



- 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 :



## 2.2 Choix des paramètres

### 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<sub>2</sub>-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<sub>2</sub>-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  $\nearrow$ ).

### **Logitboost**

- *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 :

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

### Remarque

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

### Logitboost

Algorithme de gradient boosting avec la fonction de perte

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

— 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

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

### 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*.

## Profondeur des arbres

- `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.
- 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  $\implies$  si on itère trop, on **sur-ajuste**.
- Nécessité de *calibrer correctement* ce paramètre.

## Comment ?

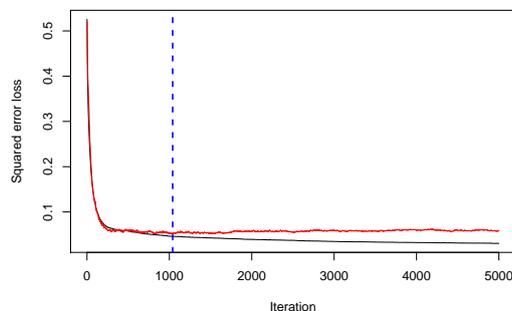
Avec des méthodes classiques d'**estimation du risque**.

## 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**.
- 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
```



$\implies$  **Risque quadratique** estimé par **hold out** avec 75% d'observations dans l'échantillon d'apprentissage.

## Rétrécissement

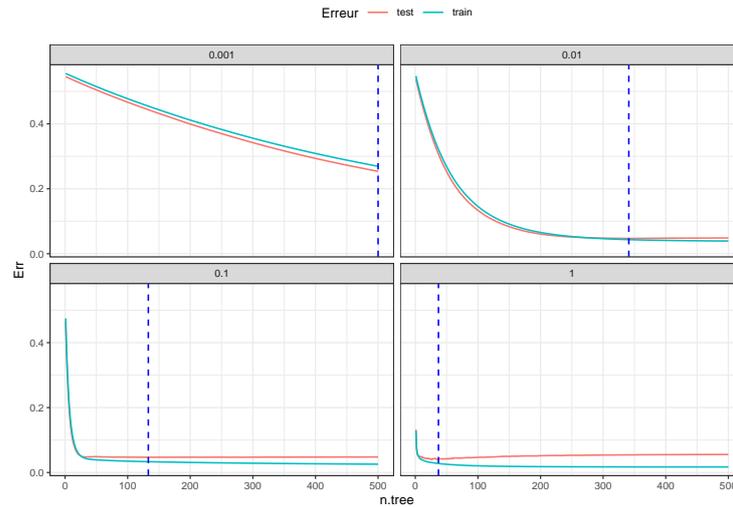
- shrinkage dans **gbm**.
- Correspond au *pas de la descente de gradient* : shrinkage  $\nearrow \implies$  minimisation plus rapide.

### 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.

### Recommandation

- 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*.

## 2.3 Compléments/conclusion

### Importance des variables

- 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  $T_{b-1}$ .

— *Récurtivité* 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"*.

### 3 Bibliographie

**Références**

**Biblio4**

[Breiman, 1996] Breiman, L. (1996). Bagging predictors. *Machine Learning*, 26(2) :123–140.

[Fernández-Delgado et al., 2014] 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 and Schapire, 1996] Freund, Y. and Schapire, R. (1996). Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*.

[Friedman, 2001] Friedman, J. H. (2001). Greedy function approximation : A gradient boosting machine. *Annals of Statistics*, 29 :1189–1232.

[Friedman, 2002] Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 28 :367–378.

[Genuer, 2010] Genuer, R. (2010). *Forêts aléatoires : aspects théoriques, sélection de variables et applications*. PhD thesis, Université Paris XI.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer, second edition.

[Wright and Ziegler, 2017] 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).

**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..)  $\implies$  performance  $\nearrow$  réseaux pré-entraînés  $\implies$  **apprentissage profond/deep learning**.