3ème année ENSAI, 2010 - 2011

Introduction aux méthodes d'agrégation : boosting, bagging et forêts aléatoires. Illustrations avec R.

Laurent Rouvière

ENSAI - Campus de Ker Lann
Rue Blaise Pascal - BP 37203
35172 BRUZ cedex
Tel : 02 99 05 32 63

Mel : laurent.rouviere@ensai.fr

Table des matières

1 Le boosting **5**
1.1 L'algorithme `adaboost` . . . 6
1.1.1 Une justification de l'algorithme `adaboost` . . . 7
1.1.2 Quelques propriétés . . . 9
1.2 Généralisation . . . 11
1.3 Les algorithme `logitboost` et L_2 `boosting` . . . 14
1.3.1 `Logitboost` . . . 14
1.3.2 L_2 `boosting` . . . 15

2 Bagging et forêts aléatoires **17**
2.1 Bagging . . . 17
2.1.1 L'algorithme . . . 17
2.1.2 Tirage de l'échantillon bootstrap . . . 18
2.1.3 Biais et variance . . . 19
2.2 Les forêts aléatoires . . . 20
2.2.1 Les random forests RI . . . 20
2.2.2 Erreur Out Of Bag et importance des variables . . . 22
2.2.3 Un exemple avec R . . . 23

Bibliographie **27**

Chapitre 1

Le boosting

Nous cherchons à expliquer une variable $Y \in \mathcal{Y}$ par p variables $\mathbf{X} = (\mathbf{X}_1, \dots, \mathbf{X}_p) \in \mathbb{R}^p$. Lorsque $\mathcal{Y} = \mathbb{R}$, on parlera de régression et lorsque Y est une variable qualitative nous parlerons de discrimination ou classification supervisée. Pour simplifier nous supposerons que dans ce cas Y ne prend que 2 valeurs (on notera $\mathcal{Y} = \{0, 1\}$ ou $\mathcal{Y} = \{-1, 1\}$ selon le contexte).

Etant donné un n échantillon i.i.d. $\mathcal{D}_n = (\mathbf{X}_1, Y_1), \dots, (\mathbf{X}_n, Y_n)$ de même loi que $(\mathbf{X}, Y)$, nous cherchons à prédire au mieux la sortie y associée à une nouvelle observation $\mathbf{x}$. Pour effectuer cette prévision, il faut constuire une règle de prédiction (règle de régression ou encore de discrimination selon le contexte), c'est-à-dire une fonction mesurable $f : \mathbb{R}^p \to \mathcal{Y}$ qui associe la sortie $f(\mathbf{x})$ à l'entrée $\mathbf{x}$. Naturellement, il existe un grand nombre de façons de construire une telle fonction. Il faut par conséquent se donner un critère qui permette de mesurer la performance des différentes règles de prévision.

Exemple 1.1
Dans un contexte de régression, on pourra par exemple mesurer la performance de f par son erreur quadratique moyenne $\mathbf{E}[(Y - f(\mathbf{X}))^2]$. On est alors amené à chercher une fonction mesurable $f^\star : \mathbb{R}^p \to \mathbb{R}$ telle que

$$\mathbf{E}[(Y - f^\star(\mathbf{X}))^2] = \min_{f:\mathbb{R}^p \to \mathbb{R}} \mathbf{E}[(Y - f(\mathbf{X}))^2].$$

La solution est donnée par la fonction de régression $f^\star(\mathbf{x}) = \mathbf{E}[Y|\mathbf{X} = \mathbf{x}]$. Bien entendu, en pratique la loi de $(\mathbf{X}, Y)$ est inconnue et il est impossible de calculer $f^\star$. Le problème consiste alors à construire un estimateur $f_n(\mathbf{x}) = f_n(\mathbf{x}, \mathcal{D}_n)$ qui soit aussi proche que possible de $f^\star$. Si on désigne par μ la loi de $\mathbf{X}$, on dira que la suite d'estimateurs (f_n) est consistante pour une certaine distribution $(\mathbf{X}, Y)$ si

$$\lim_{n\to\infty} \mathbf{E}\left[\int (f_n(\mathbf{x}) - f^\star(\mathbf{x}))^2 \mu(\mathrm{d}\mathbf{x})\right] = 0.$$

Si la suite (f_n) est consistante pour toutes les distributions de $(\mathbf{X}, Y)$, on parlera de consistance universelle.

Exemple 1.2
Pour la classification binaire (Y prend ses valeurs dans $\{-1, 1\}$), on mesure souvent la performance d'une règle $g : \mathbb{R}^p \to \{-1, 1\}$ par sa probabilité d'erreur $L(g) = \mathbf{P}(g(\mathbf{X}) \neq Y)$. La règle de Bayes définie par

$$g^\star(\mathbf{x}) = \begin{cases} 1 & \text{si } \mathbf{P}(Y = 1|\mathbf{X} = \mathbf{x}) \geq 0.5 \\ -1 & \text{sinon} \end{cases}$$

est la règle optimale pour ce critère, *i.e.*,

$$L^\star = L(g^\star) = \inf_{g:\mathbb{R}^p \to \{-1,1\}} L(g).$$

Comme dans l'exemple précédent, il est bien évidemment impossible de calculer $g^\star$ en pratique. Il nous faut alors construire un estimateur $g_n(\mathbf{x}) = g_n(\mathbf{x}, \mathcal{D}_n)$ tel que $L(g_n)$ soit le plus proche possible de $L^\star$. En ce sens, on dira que la suite (g_n) est convergente pour une certaine distribution $(\mathbf{X}, Y)$ si $\lim_{n\to\infty} \mathbf{E}L(g_n) = L^\star$. Si la suite (g_n) est consistante pour toutes les distributions de $(\mathbf{X}, Y)$, on parlera de consistance universelle.

Que ce soit en régression ou en discrimination les règles des plus proches voisins, du noyau et de l'histogramme sont des exemples de règles universellement convergentes. En pratique, à distance finie, elles se retrouvent cependant souvent confrontées à certaines difficultés : choix de paramètres (nombre de plus proches voisins, taille de la fenêtre, choix de la partition...), fléau de la dimension... Les méthodes d'agrégation permettent de pallier à certaines de ces difficultés.

1.1 L'algorithme `adaboost`

Dans cette section, nous nous plaçons dans un contexte de discrimination. Dans la communauté du *learning*, le boosting (Freund & Schapire (1996)) se révèle être l'une des idées les plus puissantes des 15 dernières et continue de faire l'objet d'une littérature abondante.

Le principe général du boosting consiste à construire une famille de modèles qui sont ensuite agrégés par une moyenne pondérée des estimations ou un vote. Les modèles sont construits de manière récursive : chaque modèle est une version adaptative du précédent en donnant plus de poids aux observations mal ajustées ou mal prédites. Le modèle construit à l'étape k concentrera donc ses efforts sur les observations mal ajustées par le modèle à l'étape $k-1$.

Le terme boosting s'applique à des méthodes générales capables de produire des décisions très précises à partir de règles peu précises (qui font légèrement mieux que le hasard). Nous commençons par décrire l'algorithme `adaboost` développé par Freund & Schapire (1997). Il s'agit de l'algorithme le plus populaire appartenant à la famille des algorithmes boosting. On désigne par $g(\mathbf{x})$ une règle de classification "faible" (weaklearner). Par faible, nous entendons une règle dont le taux d'erreur est légèrement meilleur que celui d'une règle purement aléatoire (penser pile ou face). L'idée consiste à appliquer cette règle plusieurs fois en affectant "judicieusement" un poids différent aux observations à chaque itération (voir Figure 1.1).

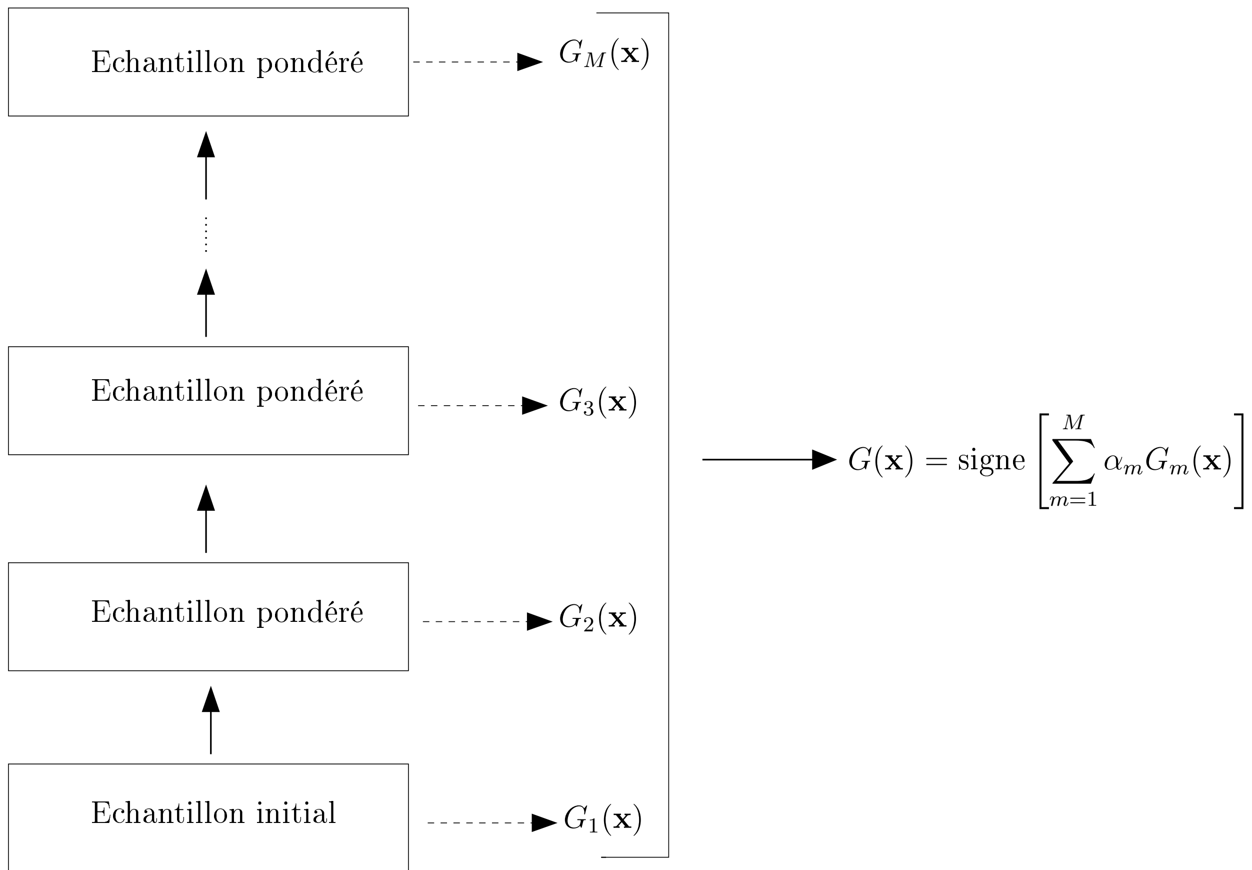

Figure 1.1 – Représentation de l'algorithme adaboost.

`Adaboost` est présenté dans l'algorithme 1. Les poids de chaque observation sont initialisés à $1/n$ pour l'estimation du premier modèle. Ils sont ensuite mis à jour pour chaque itération. L'importance d'une observation w_i est inchangée si l'observation est bien classée, dans le cas inverse elle croit avec la qualité d'ajustement du modèle mesurée par α_m. L'agrégation finale est une combinaison des règles $g_1, \dots, g_M$ pondérée par les qualités d'ajustement de chaque modèle.

Remarque

- Il faut prendre garde que la règle faible ne soit pas trop faible... En effet, pour que les coefficients α_m soient toujours positifs, il faut absolument que les taux d'erreur e_m soient inférieurs à 0.5.
- Se pose bien évidemment la question du choix de la règle faible. L'algorithme suppose que cette règle puisse s'appliquer à un échantillon pondéré. Ce n'est bien évidemment pas le cas de toutes les règles de classification. De nombreux logiciels utilisent par exemple des arbres de classification comme règle faible (CART). Dans ce cas, la règle est ajustée sur un sous échantillon de taille n de d_n dans lequel les observations sont tirées (avec remise) selon les poids $w_1, \dots, w_n$.

1.1.1 Une justification de l'algorithme `adaboost`

Restons dans le contexte de la classification binaire. Un problème classique de l'apprentissage consiste à choisir une règle g à l'intérieur d'une famille donnée $\mathcal{G}$. Ce problème est généralement abordé en minimisant l'espérance d'une fonction de perte $L : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$:

$$g^\star(\mathbf{x}) = \operatorname*{argmin}_{g \in \mathcal{G}} \mathbf{E}[L(Y, g(\mathbf{X}))].$$

Algorithme 1 `AdaBoost`

Entrée :
- $\mathbf{x}$ l'observation à prévoir
- $d_n = (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ l'échantillon
- M le 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 $g_m(\mathbf{x})$ sur l'échantillon d_n pondéré par les poids $w_1, \dots, w_n$
   (b) Calculer le taux d'erreur :
   $$e_m = \frac{\sum_{i=1}^n w_i \mathbf{1}_{y_i \neq g_m(\mathbf{x}_i)}}{\sum_{i=1}^n w_i}.$$
   (c) Calculer : $\alpha_m = \log((1 - e_m)/e_m)$
   (d) Réajuster les poids :
   $$w_i = w_i \exp(\alpha_m \mathbf{1}_{y_i \neq g_m(\mathbf{x}_i)}), \quad i = 1, \dots, n$$
3. **Sortie** : $\hat{g}(\mathbf{x}) = \sum_{m=1}^M \alpha_m g_m(\mathbf{x})$.

Une fonction de perte naturelle pour la classification est $L(g) = \mathbf{1}_{g(\mathbf{X}) \neq Y}$. En pratique, la loi de $(\mathbf{X}, Y)$ étant inconnue, on minimise une version empirique de $\mathbf{E}[L(Y, g(\mathbf{X}))]$:

$$g_n^\star(\mathbf{x}) = \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{n} \sum_{i=1}^n L(Y_i, g(\mathbf{X}_i)) = \operatorname*{argmin}_{g \in \mathcal{G}} \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{Y_i \neq g(\mathbf{X}_i)}.$$

Pour de nombreuses familles de règles, ce problème de minimisation est généralement difficile à résoudre numériquement. Une solution consiste à "convexifier" la fonction de perte de manière à utiliser des algorithmes de minimisation plus efficaces.

Nous allons montrer dans cette partie que l'algorithme `adaboost` répond à ce principe de minimisation pour le risque convexifié :

$$L(y, g(\mathbf{x})) = \exp(-y g(\mathbf{x})).$$

Pour plus de détails, on pourra se référer à Hastie *et al* (2009) (chapitre 10). On se donne $\mathcal{G}$ une famille de règles "faibles", et on cherche à construire un estimateur de manière récursive $\tilde{g}_m(\mathbf{x}) = \tilde{g}_{m-1}(\mathbf{x}) + \beta_m G_m(\mathbf{x})$ tel que β_m et G_m soit choisi de manière a minimiser le risque convexifié ci-dessus :

$$\begin{aligned}
(\beta_m, G_m) &= \operatorname*{argmin}_{\beta, \tilde{g}} \sum_{i=1}^n \exp(-y_i \tilde{g}_m(\mathbf{x}_i)) \\
&= \operatorname*{argmin}_{\beta, \tilde{g}} \sum_{i=1}^n \exp(-y_i \tilde{g}_{m-1}(\mathbf{x}_i)) \exp(-y_i \beta \tilde{g}(\mathbf{x}_i)) \\
&= \operatorname*{argmin}_{\beta, \tilde{g}} \sum_{i=1}^n w_i^m \exp(-y_i \beta \tilde{g}(\mathbf{x}_i))
\end{aligned}$$

Nous remarquons que le critère à minimiser peut se ré-écrire

$$\exp(-\beta)\sum_{y_i=\tilde{g}(\mathbf{x}_i)} w_i^m + \exp(\beta)\sum_{y_i\neq\tilde{g}(\mathbf{x}_i)} w_i^m = (\exp(\beta)-\exp(-\beta))\sum_{i=1}^n w_i^m \mathbf{1}_{Y_i\neq\tilde{g}(\mathbf{x}_i)} + \exp(-\beta)\sum_{i=1}^n w_i^m. \tag{1.1}$$

Ainsi, pour $\beta > 0$ fixé, la solution en $\tilde{g}$ est donnée par

$$G_m = \operatorname*{argmin}_{\tilde{g}} \sum_{i=1}^n w_i^m \mathbf{1}_{y_i\neq\tilde{g}(\mathbf{x}_i)}. \tag{1.2}$$

G_m s'obtient en minimisant l'erreur de prédiction pondérée par les w_i^m.

Calcul de β_m

On obtient β_m en minimisant (1.1) comme une fonction du seul paramètre β (il suffit d'annuler la dérivée) et en pluggant le G_m de (1.2) :

$$\beta_m = \frac{1}{2}\log\left(\frac{1-err_m}{err_m}\right)$$

où

$$err_m = \frac{\sum_{i=1}^n w_i^m \mathbf{1}_{Y_i\neq G_m(\mathbf{x}_i)}}{\sum_{i=1}^n w_i^m}.$$

Mise à jour des poids A l'étape m, on a ainsi $\tilde{g}_m(\mathbf{x}) = \tilde{g}_{m-1}(\mathbf{x}) + \beta_m G_m(\mathbf{x})$, ce qui donne pour les poids

$$w_i^{m+1} = w_i^m \exp(-\beta_m y_i G_m(\mathbf{x}_i)) = w_i^m \exp(2\beta_m)\mathbf{1}_{y_i\neq G_m(\mathbf{x}_i)} \exp(-\beta_m). \tag{1.3}$$

Conclusion :

- à la première itération les poids sont constants et égaux à $1/n$;
- Pour tout m, on remarque que $err_m = e_m$ et donc $\alpha_m = 2\beta_m$. Par conséquent les poids sont mis à jour de la même manière que pour `adaboost` (le facteur $\exp(-\beta_m)$ dans le terme de droite de (1.3) ne dépendant pas de i, il n'a pas d'effet sur l'algorithme) ;
- G_m est obtenu en minimisant une erreur de prévision basée sur les poids w_i^m, ce qui correspond à la première étape d'`adaboost` ($G_m = g_m$) ;
- La prévision $\hat{y}$ obtenue après M itérations s'écrit

$$\hat{y} = \text{signe}\left(\tilde{g}_M(\mathbf{x})\right) = \text{signe}\left(\sum_{m=1}^M \beta_m G_m(\mathbf{x})\right) = \text{signe}\left(\frac{1}{2}\sum_{m=1}^M \alpha_m g_m(\mathbf{x})\right) = \text{signe}\left(\hat{g}(\mathbf{x})\right).$$

On retrouve bien la sortie de `adaboost`.

1.1.2 Quelques propriétés

Freund & Schapire (1999) ont étudié la probabilité d'erreur empirique mesurée sur l'échantillon d'apprentissage ainsi l'erreur de généralisation commises par la règle issue de l'algorithme `adaboost`. Nous présentons les principaux résultats dans cette partie.

On note $\hat{g}_M$ la règle `adaboost` obtenue après M itérations. Avec les notations de la partie précédente, nous avons $\hat{g}_M(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m g_m(\mathbf{x})$. On rappelle que e_m désigne le taux d'erreur calculé sur l'échantillon pondéré de la règle g_m :

$$e_m = \frac{\sum_{i=1}^{n} w_i \mathbf{1}_{y_i \neq g_m(\mathbf{x}_i)}}{\sum_{i=1}^{n} w_i}.$$

Si en entrée, on dispose bien d'une règle "faible" alors sa performance doit être meilleure qu'une règle aléatoire. En ce sens, on désigne par γ_m le gain de la règle g_m par rapport à une règle aléatoire : $e_m = 1/2 - \gamma_m$. On note également $L_n(\hat{g}_M)$ l'erreur empirique de $\hat{g}_M$ calculée sur l'échantillon d'apprentissage :

$$L_n(\hat{g}_M) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{Y_i \neq \hat{g}_M(\mathbf{X}_i)}.$$

Freund & Schapire (1999) ont montré que

$$L_n(\hat{g}_M) \leq \exp\left(-2 \sum_{m=1}^{M} \gamma_m^2\right).$$

Ainsi si les règles faibles sont meilleures que le hasard ($\gamma_m \geq \gamma > 0$), alors l'erreur empirique tend vers 0 exponentiellement avec M.

Intéressons nous maintenant à l'erreur de généralisation $L(\hat{g}_M) = \mathbf{P}(\hat{g}_M(\mathbf{X}) \neq Y)$. On désigne par V la dimension de Vapnik-Chervonenkis de l'espace dans lequel la règle faible est sélectionnée. On a le résultat suivant

$$L(\hat{g}_M) \leq L_n(\hat{g}_M) + \mathrm{O}\left(\sqrt{\frac{MV}{n}}\right).$$

En premier lieu, notons que la dimension V est faible devant T et n pour la plupart des espaces dans lequel la règle faible est sélectionnée. Cette borne suggère une conclusion à laquelle on pouvait s'attendre : `adaboost` va avoir tendance à sur-ajuster l'échantillon d'apprentissage lorsque M devient grand. Cette remarque a donné lieu à de nombreux débats dans la communauté du learning puisqu'on ne remarquait pas ce phénomène de sur-ajustement sur de nombreux exemples (même après plusieurs milliers d'itérations). Aujourd'hui les avis convergent néanmoins sur le fait qu'il y a bien un sur-ajustement. Se pose alors naturellement le problème du choix de M. Les méthodes classiques reposent sur des algorithmes de type validation croisée ou estimateur *out of bag* de l'erreur de généralisation. Nous les présenterons un peu plus tard à l'aide du package `gbm`.

Nous terminons cette partie en énonçant un résultat de convergence pour l'estimateur `adaboost`. Pour simplifier nous présentons ce résultat dans le cas où la classe de règles faibles utilisées pour construire l'algorithme est l'ensemble des arbres binaires à $p+1$ noeuds terminaux ou l'ensemble défini par la classe des indicatrices des demi-espaces engendrés par les hyperplans de $\mathbb{R}^p$.

Théorème 1.1 (Bartlett & Traskin (2007))
On suppose que le nombre d'itération $M = M_n = n^{1-\varepsilon}$ pour $\varepsilon \in]0,1[$. Alors la règle de classification issue de l'algorithme `adaboost` à l'étape M_n est fortement universellement consistante

$$\lim_{n \to \infty} L(g_{M_n}) = L^\star, \quad p.s.$$

1.2 Généralisation

Nous avons vu dans la section 1.1.1 que l'algorithme adaboost peut être vu comme un algorithme itératif minimisant à chaque étape une fonction de coût sur une classe de fonctions donnée. Ce point de vue se généralise à d'autres fonctions de coût. C'est de cette manière que l'on peut présenter les méthodes boosting dans un cadre plus général.

Remarque
La manière de présenter l'algorithme "boosting" peut légèrement différer selon les auteurs ainsi que selon les logiciels (voir même les packages R). Nous présentons ici l'algorithme tel qu'il est utilisé sur le package gbm.

On se donne g une règle faible (par exemple un arbre à deux noeuds terminaux construits via l'algorithme CART) et $L : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ une fonction de perte.

Algorithme 2 Boosting

Entrées :
- $\mathbf{x}$ l'observation à prévoir
- $d_n = (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ l'échantillon
- λ un paramètre de régularisation tel que $0 < \lambda \leq 1$.
- M le nombre d'itérations.

1. Initialisation :
$$g_0(.) = \underset{c}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(y_i, c)$$
2. **Pour** $m = 1$ à M :
   (a) Calculer l'opposé du gradient $-\frac{\partial}{\partial g} L(Y, g)$ et l'évaluer aux points $g_{m-1}(\mathbf{x}_i)$:
$$U_i = -\frac{\partial}{\partial g} L(y_i, g_{m-1}(\mathbf{x}_i)), \quad i = 1, \dots, n.$$
   (b) Ajuster la règle faible sur l'échantillon $(X_1, U_1), \dots, (X_n, U_n)$, on note h_m la règle ainsi définie.
   (c) Mise à jour : $g_m(\mathbf{x}) = g_{m-1}(\mathbf{x}) + \lambda h_m(\mathbf{x})$.
3. **Sortie :** la règle $g_M(\mathbf{x})$.

Remarque
- Cet algorithme est également appelé algorithme FGD (Functional Gradient Descent). Les méthodes de descente de gradient sont généralement utilisées pour minimiser des fonctions convexes. Dans le cas qui est le notre, on cherche à minimiser en g la fonction de coût $\mathbf{E}[L(Y, g(\mathbf{X}))]$. Une méthode de descente de gradient consiste à chercher une suite (g_k) qui converge vers le minimum cherché.

  Nous rappelons cette méthode dans le cas où la fonction à minimiser J va de $\mathbb{R}$ dans $\mathbb{R}$. On note $\tilde{x}$ la solution et on cherche une suite (x_k) qui converge vers $\tilde{x}$. L'algorithme est tout d'abord

initialisé en choisissant une valeur x_0. On cherche alors $x_1 = x_0 + h$ tel que $J'(x_1) \approx 0$. Par un développement limité, on obtient l'approximation

$$J'(x_0 + h) \approx J'(x_0) + hJ''(x_0).$$

Comme $J'(x_0 + h) \approx 0$, il vient $h = -(J''(x_0))^{-1} J'(x_0)$. Si on pose $\lambda = (J''(x_0))^{-1}$, alors $x_1 = x_0 - \lambda J'(x_0)$ et on déduit la formule de récurrence $x_k = x_{k-1} - \lambda J'(x_{k-1})$.

Dans le cas qui est le notre, on cherche à minimiser $J : g \mapsto \mathbf{E}[L(Y, g(\mathbf{X}))]$. En transposant cette approche dans un cadre fonctionnel (voir Bühlmann & Yu (2003)), on obtient la formule de récurrence :

$$g_k = g_{k-1} - \lambda \nabla J(g_{k-1}).$$

Le gradient $\nabla J(g_{k-1})$ au point g_{k-1} est estimé en faisant la régression des U_i sur X_i dans l'algorithme.

- Les descentes de gradient sont connues pour être particulièrement performantes lorsque la fonction à minimiser est convexe. Nous verrons dans la suite que les fonctions de perte classiques utilisées en boosting sont des fonctions convexes.
- Pour le choix $\lambda = 1$ et $L(y, g) = \exp(-yg)$ l'algorithme 2 coïncide avec l'algorithme `adaboost`.
- Le choix du paramètre λ est d'importance mineure. De nombreux auteurs recommandent de le prendre "petit" (de l'ordre de 0.1). Ce choix est lié au choix du nombre d'itérations optimal : une valeur de λ petite nécessitera un grand nombre d'itérations et inversement.

Exemple 1.3
On considère un échantillon d'apprentissage de taille 100 simulé selon le modèle :

$$X_i \sim \mathcal{N}(0,1), \qquad U_i \sim \mathcal{U}[0,1] \quad \text{et} \quad Y_i = \begin{cases} \mathbf{1}_{U_i \leq 0.25} & \text{si } X_i \leq 0 \\ \mathbf{1}_{U_i > 0.25} & \text{si } X_i > 0. \end{cases}$$

On pourra remarquer que l'erreur de Bayes $L^\star$ vaut 0.25. On lance l'algorithme `adaboost` (8 000 itérations) à l'aide du package `gbm` pour $\lambda = 1$ et $\lambda = 0.01$. Le classifieur faible est un arbre à 2 noeuds terminaux (stumps)

```
> B <- 8000
> model <- gbm(Y~.,data=dapp,distribution="adaboost",interaction.depth=1,shrinkage=1,n.trees=B)
> model1 <- gbm(Y~.,data=dapp,distribution="adaboost",interaction.depth=1,shrinkage=0.01,
                                                                                 n.trees=B)
```

On estime ensuite le pourcentage de mal classés de ces modèles en fonction du nombre d'itérations en utilisant un échantillon de validation de taille 400 simulé selon le même modèle. On calcule également le pourcentage de mal classés sur l'échantillon d'apprentissage (voir Figure 1.2).

```
> boucle <- seq(1,B,by=50)
> errapp <- rep(0,length(boucle))
> errtest <- errapp
> k <- 0
> for (i in boucle){
+  k <- k+1
+  prev_app <- predict(model,newdata=dapp,n.trees=i)
+  errapp[k] <- sum(as.numeric(prev_app>0)!=dapp$Y)/nrow(dapp)
```

```
+   prev_test<- predict(model,newdata=dtest,n.trees=i)
+   errtest[k] <- sum(as.numeric(prev_test>0)!=dtest$Y)/nrow(dtest)
+}
> plot(boucle,errapp,type="l",col="blue",ylim=c(0,0.5),xlab="nombre d'iterations",
                                                              ylab="erreur")
> lines(boucle,errtest,col="red")
```

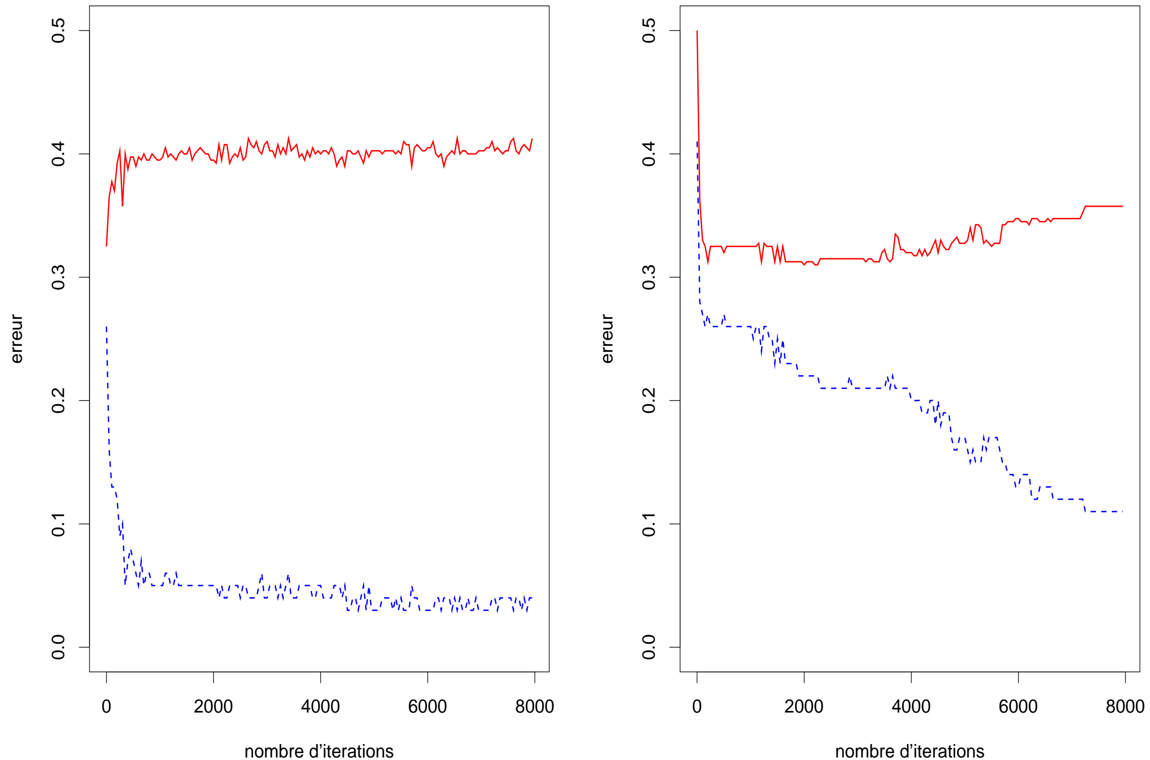

FIGURE 1.2 – Taux de mal classés estimés sur l'échantillon test (rouge, trait plein) et sur l'échantillon d'apprentissage (bleu, tirets) pour $\lambda = 1$ (gauche) et $\lambda = 0.01$ (droite).

A travers cet exemple, nous voyons clairement l'intérêt de prendre λ petit. Pour $\lambda = 1$ la descente de gradient est trop rapide. On retrouve également sur le graphe de droite le problème du sur-apprentissage (overfitting).

Le package gbm propose de sélectionner automatiquement le nombre d'itérations selon 3 critères : apprentissage-validation, validation croisée et méthode Out Of Bag. Par exemple, pour la validation croisée, il suffit d'utiliser l'argument `cv.folds`. Pour une validation croisée découpant l'échantillon en 5, il suffit de lancer la commande :

```
> model1 <- gbm(Y~.,data=dapp,distribution="adaboost",interaction.depth=1,shrinkage=0.01,
                                                                  n.trees=B,cv.folds=5)
```

On obtient le nombre d'itérations sélectionné :

```
> gbm.perf(model1,method="OOB")
[1] 129
```

Il faut prendre garde : le critère utilisé par `gbm` pour sélectionner le nombre d'itération optimal n'est pas le pourcentage de mal classés, c'est la fonction de coût de l'algorithme (ici $\exp(-yf)$). Si on souhaite choisir M qui minimise le pourcentage de mal classés, il faut le programmer soit même.

1.3 Les algorithme `logitboost` et L_2 `boosting`

Avec `adaboost`, ces deux algorithmes figurent parmi les algorithmes boosting les plus utilisés.

1.3.1 `Logitboost`

On se place dans un cadre de discrimination avec Y à valeurs dans $\{0,1\}$. La variable $Y|\mathbf{X}=\mathbf{x}$ suit une loi de Bernoulli de paramètre $p(\mathbf{x})=\mathbf{P}(Y=1|\mathbf{X}=\mathbf{x})$. La vraisemblance pour une observation $(\mathbf{x},y)$ s'écrit

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

Le modèle logistique consiste à poser

$$p(\mathbf{x})=\frac{1}{1+\exp(-\mathbf{x}'\beta)}=\frac{\exp(\mathbf{x}'\beta)}{1+\exp(\mathbf{x}'\beta)}.$$

Le vecteur de paramètres β est généralement estimé en maximisant la vraisemblance.

Comme son nom l'indique, le modèle `logitboost` repose sur une approche similaire. Etant donné $f:\mathbb{R}^p\to\mathbb{R}$, on pose

$$p(\mathbf{x})=\frac{\exp(f(\mathbf{x}))}{\exp(f(\mathbf{x}))+\exp(-f(\mathbf{x}))}=\frac{1}{1+\exp(-2f(\mathbf{x}))},$$

ce qui donne

$$f(\mathbf{x})=\frac{1}{2}\log\left(\frac{p(\mathbf{x})}{1-p(\mathbf{x})}\right).$$

On cherche alors la fonction f qui maximise la log-vraisemblance (ou qui minimise son opposé) :

$$-(y\log(p(\mathbf{x}))+(1-y)\log(1-p(\mathbf{x})))=-2f(\mathbf{x})y+\log(1+\exp(2f(\mathbf{x})))=\log(1+\exp(-2\tilde{y}f))$$

où $\tilde{y}=2y-1\in\{-1,1\}$. L'approche `logitboost` consiste à appliquer l'algorithme 2 en utilisant la fonction de perte

$$L(y,f)=\log(1+\exp(-2\tilde{y}f)).$$

Remarque
Après M itérations, l'algorithme fournit un estimateur f_M de

$$f^\star(.)=\operatorname*{argmin}_f \mathbf{E}[\log(1+\exp(-2\tilde{Y}f(\mathbf{X})))].$$

On peut montrer que $f^\star(\mathbf{x})=\frac{1}{2}\log\left(\frac{p(\mathbf{x})}{1-p(\mathbf{x})}\right)$. On déduit un estimateur $p_M(\mathbf{x})$ de $p(\mathbf{x})$ en posant

$$p_M(\mathbf{x})=\frac{\exp(f_M(\mathbf{x}))}{\exp(f_M(\mathbf{x}))+\exp(-f_M(\mathbf{x}))}=\frac{1}{1+\exp(-2f_M(\mathbf{x}))}.$$

Pour un nouvel individu $\mathbf{x}$, on obtient la règle de classification

$$\hat{y}=\begin{cases}1 & \text{si } f_M(\mathbf{x})\geq 0 \iff p_M(\mathbf{x})\geq 0.5\\ 0 & \text{si } f_M(\mathbf{x})<0 \iff p_M(\mathbf{x})<0.5.\end{cases}$$

1.3.2 L_2 boosting

On se place dans un contexte de régression ($\mathcal{Y} = \mathbb{R}$). On désigne par $f : \mathbb{R}^p \to \mathbb{R}$ un régresseur "faible". Ici par faible, nous entendons un régresseur (fortement) biaisé, par exemple :

- un arbre à deux nœuds terminaux (stumps) ;
- un estimateur à noyau (Nadaraya-Watson) construit avec une "grande" fenêtre.

Le L_2 boosting consiste à appliquer l'algorithme 2 avec la fonction de perte quadratique

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

Remarque

- Après M itérations, l'algorithme fournit un estimateur f_M de

$$f^\star(.) = \operatorname*{argmin}_f \mathbf{E}\left[\frac{1}{2}(Y - f(X))^2\right],$$

  c'est-à-dire de la fonction de régression $f^\star(\mathbf{x}) = \mathbf{E}[Y|\mathbf{X} = \mathbf{x}]$.
- Les variables U_i de l'étape 2.a de l'algorithme 2 s'écrivent $U_i = y_i - f_{m-1}(\mathbf{x}_i)$. L'étape 2.b consiste donc simplement à faire une régression sur les résidus du modèle construit à l'étape $m - 1$.
- Pour certains type de régresseurs faibles (noyaux, splines), on peut montrer qu'à chaque itération (voir Bühlmann & Yu (2003)) le biais de l'estimateur diminue tandis que sa variance augmente. Cependant, la réduction de biais est généralement nettement plus importante que l'augmentation de variance. C'est pourquoi il est préconisé d'utiliser un régresseur biaisé pour cet algorithme (si le régresseur possède déjà une forte variance, on ne pourra pas la diminuer en itérant).

Pour finir, nous récapitulons les trois algorithmes dans le tableau 1.1 ainsi que sur la Figure 1.3. Les fonctions de perte ont été ajustées de sorte qu'elles passent par le point $(0, 1)$.

Espaces	$L(y, f)$	$f^\star(\mathbf{x})$	Algorithme
$y \in \{0, 1\},\ f \in \mathbb{R}$	$\exp(-2\tilde{y}f)$	$\frac{1}{2}\log\left(\frac{p(\mathbf{x})}{1-p(\mathbf{x})}\right)$	Adaboost
$y \in \{0, 1\},\ f \in \mathbb{R}$	$\log(1 + \exp(-2\tilde{y}f))$	$\frac{1}{2}\log\left(\frac{p(\mathbf{x})}{1-p(\mathbf{x})}\right)$	Logitboost
$y \in \mathbb{R},\ f \in \mathbb{R}$	$\frac{1}{2}(y - f)^2$	$\mathbf{E}[Y|\mathbf{X} = \mathbf{x}]$	L_2 boosting

TABLE 1.1 – Algorithmes boosting ($\tilde{y} = 2y - 1$).

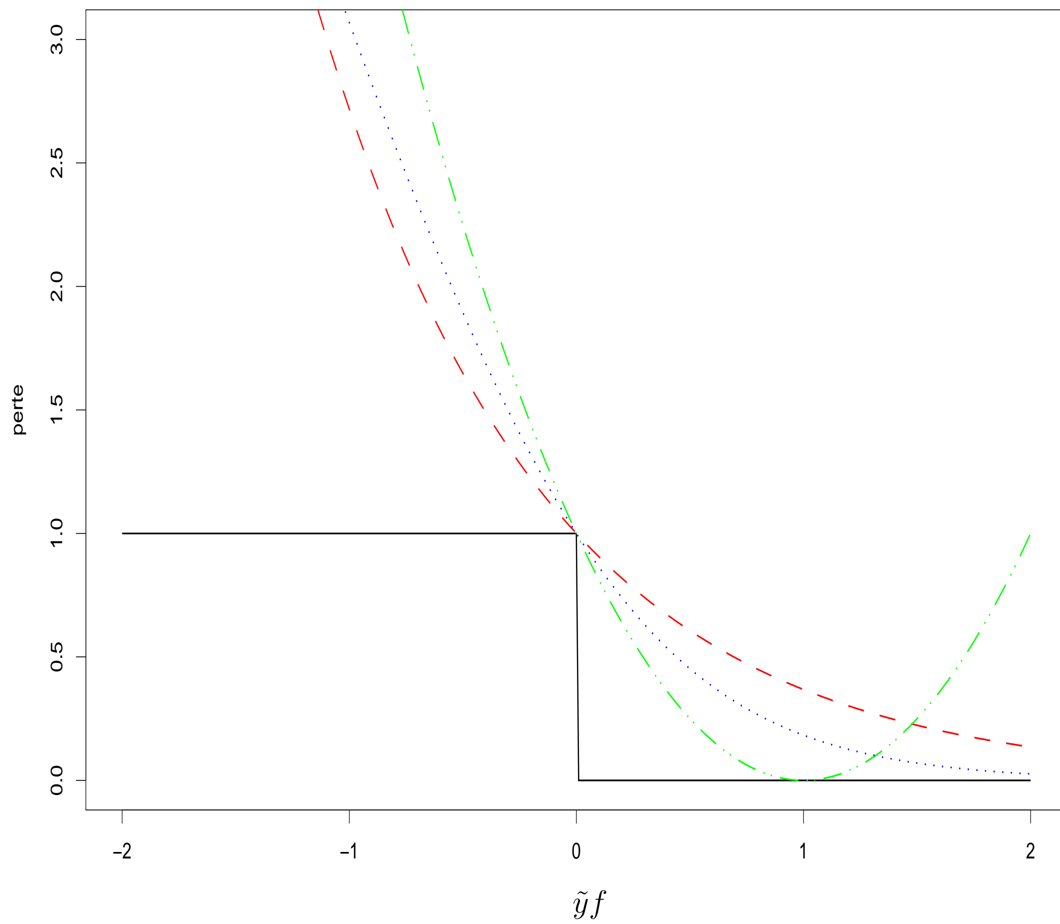

FIGURE 1.3 – Fonctions de perte des algorithmes boosting : `adaboost` (rouge, tirets), `logitboost` (bleu, pointillés), L_2 `boosting` (vert, points-tirets). La fonction de perte $\mathbf{1}_{yf>0}$ est en noir (trait plein).

Chapitre 2

Bagging et forêts aléatoires

2.1 Bagging

Le bagging a été introduit par Léo Breiman en 1996 (voir Breiman (1996)). Le terme bagging vient de la contraction de **B**oostrap **Agg**regat**ing**. Nous présentons cette famille de méthodes dans un contexte de régression. Elles s'étendent aisément à la classification supervisée. On désigne par $(\mathbf{X}, Y)$ un vecteur aléatoire où $\mathbf{X}$ prend ses valeurs dans $\mathbb{R}^p$ et Y dans $\mathbb{R}$. On note $\mathcal{D}_n = (\mathbf{X}_1, Y_1), \dots, (\mathbf{X}_n, Y_n)$ un n-échantillon i.i.d. et de même loi que $(\mathbf{X}, Y)$ et $m(\mathbf{x}) = \mathbf{E}[Y|\mathbf{X} = \mathbf{x}]$ la fonction de régression. Pour $\mathbf{x} \in \mathbb{R}^d$, on considère l'erreur quadratique moyenne d'un estimateur $\hat{m}$ et sa décomposition biais-variance :

$$\mathbf{E}[(\hat{m}(\mathbf{x}) - m(\mathbf{x}))^2] = (\mathbf{E}[\hat{m}(x)] - m(\mathbf{x}))^2 + \mathbf{V}(\hat{m}(\mathbf{x})).$$

Tout comme le boosting, les méthodes bagging sont des méthodes d'agrégation. Elles vont consister à agréger un nombre B d'estimateurs $\hat{m}_1, \dots, \hat{m}_B$: $\hat{m}(\mathbf{x}) = \frac{1}{B}\sum_{k=1}^{B} \hat{m}_k(\mathbf{x})$. Si les régrésseurs $\hat{m}_1, \dots, \hat{m}_B$ sont i.i.d. alors :

$$\mathbf{E}[\hat{m}(\mathbf{x})] = \mathbf{E}[\hat{m}_1(\mathbf{x})] \quad \text{et} \quad \mathbf{V}(\hat{m}(\mathbf{x})) = \frac{1}{B}\mathbf{V}(\hat{m}_1(\mathbf{x})).$$

Le biais de l'estimateur agrégé est donc le même que celui des $\hat{m}_k$ mais la variance diminue. Bien entendu, en pratique il est quasiment impossible de considérer des estimateurs $\hat{m}_k$ indépendants dans la mesure où ils dépendent tous du même échantillon $\mathcal{D}_n$. L'idée du bagging est de créer une "indépendance artificielle" en construisant les estimateurs $\hat{m}_k$ sur des échantillons bootstrap.

2.1.1 L'algorithme

L'algorithme est simple à implémenter : il suffit de construire B estimateurs sur des échantillons bootstrap et de les agréger (voir algorithme 3). Le fait de considérer des échantillons bootstrap introduit un aléa supplémentaire dans l'estimateur. Afin de prendre en compte cette nouvelle source d'aléatoire, on note $\theta_k = \theta_k(\mathcal{D}_n)$ l'échantillon bootstrap de l'étape k et $\hat{m}(., \theta_k)$ l'estimateur construit à l'étape k. On écrira l'estimateur final $\hat{m}_B(\mathbf{x}) = \frac{1}{B}\sum_{k=1}^{B} \hat{m}(\mathbf{x}, \theta_k)$.

Remarque
Les tirages bootstrap sont effectués de la même manière et indépendamment les uns des autres. Ainsi, conditionnellement à $\mathcal{D}_n$, les variables $\theta_1, \dots, \theta_B$ sont i.i.d. et de même loi que θ (qui représentera la loi de la variable de tirage de l'échantillon bootstrap, voir section 2.1.2 pour des exemples).

Algorithme 3 Bagging.

Entrées :
- $\mathbf{x}$ l'observation à prévoir
- un régresseur (arbre CART, 1 plus proche voisin...)
- d_n l'échantillon
- B le nombre d'estimateurs que l'on agrège.

Pour $k = 1, \ldots, B$:

1. Tirer un échantillon bootstrap d_n^k dans d_n
2. Ajuster le régresseur sur cet échantillon bootstrap : $\hat{m}_k$

Sortie : L'estimateur $\hat{m}(\mathbf{x}) = \frac{1}{B}\sum_{k=1}^{B}\hat{m}_k(\mathbf{x})$.

Ainsi, d'après la loi des grands nombres :

$$\lim_{B\to\infty}\hat{m}_B(\mathbf{x}) = \lim_{B\to\infty}\frac{1}{B}\sum_{k=1}^{B}\hat{m}(\mathbf{x},\theta_k) = \mathbf{E}_\theta[\hat{m}(\mathbf{x},\theta)|\mathcal{D}_n] \qquad \text{p.s.}$$

L'espérance est ici calculée par rapport à la loi de θ. On déduit de ce résultat que, contrairement au boosting, prendre B trop grand ne va pas sur-ajuster l'échantillon. Dit brutalement, prendre la limite en B revient à considérer un estimateur "moyen" calculé sur tous les échantillons bootstrap. Le choix de B n'est donc pas crucial pour la performance de l'estimateur, il est recommandé de le prendre le plus grand possible (en fonction du temps de calcul).

2.1.2 Tirage de l'échantillon bootstrap

Deux techniques sont généralement utilisées pour générer les échantillons bootstrap.

Technique A : $\theta_k(\mathcal{D}_n)$ est obtenu en tirant n observations avec remise dans $\mathcal{D}_n$, chaque observation ayant la même probabilité d'être tirée ($1/n$).

Technique B : $\theta_k(\mathcal{D}_n)$ est obtenu en tirant ℓ observations (avec ou sans remise) dans $\mathcal{D}_n$ avec $\ell < n$.

Tout comme le boosting, la bagging permet de rendre performante des règles dites "faibles". Baggons par exemple la règle du 1 plus proche voisin (on rappelle que cette règle n'est pas universellement consistante). On désigne par $\hat{m}(.,\theta_k)$ la règle du 1 plus proche voisin construite sur l'échantillon bootstrap $\theta_k(\mathcal{D}_n)$, le bootstrap ayant été réalisé selon la technique B. On note $\hat{m}(\mathbf{x})$ l'estimateur baggé (une infinité de fois...) :

$$\hat{m}(\mathbf{x}) = \lim_{B\to\infty}\frac{1}{B}\sum_{k=1}^{B}\hat{m}(\mathbf{x},\theta_k) = \mathbf{E}_\theta[\hat{m}(\mathbf{x},\theta)|\mathcal{D}_n].$$

On a le résultat suivant.

Théorème 2.1 (Biau & Devroye (2010))
Si $\ell = \ell_n$ tel que $\lim_{n\to\infty}\ell_n = +\infty$ et $\lim_{n\to\infty}\frac{\ell_n}{n} = 0$ alors l'estimateur $\hat{m}(\mathbf{x})$ est universellement consistant.

2.1.3 Biais et variance

Dans cette partie, nous comparons le biais et la variance de l'estimateur agrégé à ceux des estimateurs que l'on agrège. On note :
- $\hat{m}_B(\mathbf{x}) = \frac{1}{B}\sum_{k=1}^{B}\hat{m}(\mathbf{x},\theta_k)$ et $\hat{m}(\mathbf{x}) = \lim_{B\to\infty}\hat{m}_B(\mathbf{x})$;
- $\sigma^2(x) = \mathbf{V}(\hat{m}(\mathbf{x},\theta_k))$ la variance des estimateurs que l'on agrège ;
- $\rho(\mathbf{x}) = \mathrm{corr}[\hat{m}(\mathbf{x},\theta_1),\hat{m}(\mathbf{x},\theta_2)]$ le coefficient de corrélation entre deux estimateurs que l'on agrège (calculés sur deux échantillons bootstrap).

La variance $\sigma^2(\mathbf{x})$ et la corrélation $\rho(\mathbf{x})$ sont calculées par rapport aux lois de $\mathcal{D}_n$ et de θ. On suppose que les estimateurs $\hat{m}(\mathbf{x},\theta_1),\dots,\hat{m}(\mathbf{x},\theta_B)$ sont identiquement distribués. Cet hypothèse n'est pas contraignante puisque généralement les θ_i sont i.i.d. Il est alors facile de voir que le biais de l'estimateur agrégé est le même que le biais des estimateurs que l'on agrège. Par conséquent, agréger ne modifie pas le biais. Pour la variance, on a le résultat suivant.

Proposition 2.1
On a :

$$\mathbf{V}(\hat{m}_B(\mathbf{x})) = \rho(\mathbf{x})\sigma^2(\mathbf{x}) + \frac{1-\rho(\mathbf{x})}{B}\sigma^2(\mathbf{x}).$$

Par conséquent

$$\mathbf{V}(\hat{m}(\mathbf{x})) = \rho(\mathbf{x})\sigma^2(\mathbf{x}).$$

Preuve
On note $T_k = \hat{m}(\mathbf{x},\theta_k)$. Les T_k étant identiquement distribuées, on a

$$\begin{aligned}\mathbf{V}(\hat{m}_B(\mathbf{x})) =& \mathbf{V}\left[\frac{1}{B}\sum_{k=1}^{B}T_k\right] = \frac{1}{B^2}\left[\sum_{k=1}^{B}\mathbf{V}(T_k) + \sum_{1\leq k\neq k'\leq B}\mathbf{cov}(T_k,T_{k'})\right]\\ =& \frac{1}{B}\sigma^2(\mathbf{x})\frac{1}{B^2}[B^2-B]\rho(\mathbf{x})\sigma^2(\mathbf{x}) = \rho(\mathbf{x})\sigma^2(\mathbf{x}) + \frac{1-\rho(\mathbf{x})}{B}\sigma^2(\mathbf{x}).\end{aligned}$$

Ainsi, si $\rho(\mathbf{x}) < 1$, l'estimateur baggé a une variance plus petite que celle des estimateurs que l'on agrège. A la lueur de ce résultat, on pourrait être tenté de se dire que la bonne stratégie consiste à bagger des estimateurs ayant un biais le plus faible possible. Ceci n'est clairement pas acceptable. En effet, prendre des estimateurs ayant un biais faible implique que leur variance $\sigma^2(\mathbf{x})$ sera forte. Certes, le fait de bagger permettra de réduire dans une certaine mesure cette variance. Cependant, rien ne garantit qu'au final l'estimateur agrégé sera performant (si $\sigma^2(\mathbf{x})$ est très élevée alors $\rho(\mathbf{x})\sigma^2(\mathbf{x})$ sera également élevée !).

On déduit de la proposition 2.1 que c'est la corrélation $\rho(\mathbf{x})$ entre les estimateurs que l'on agrège qui quantifie le gain de la procédure d'agrégation : la variance diminuera d'autant plus que les estimateurs que l'on agrège seront "différents" (décorrélés). Le fait de construire les estimateurs sur des échantillons bootstrap va dans ce sens. Il faut néanmoins prendre garde à ce que les estimateurs que l'on agrège soient sensibles à des perturbations de l'échantillon bootstrap. Si ces estimateurs sont robustes à de telles perturbations, les bagger n'apportera aucune amélioration. Si par exemple $\hat{m}(.,\theta_k)$ est un estimateur B-splines, alors l'estimateur agrégé $\hat{m}_B(.)$ converge presque sûrement vers $\hat{m}(.,\mathcal{D}_n)$ (voir Hastie *et al* (2009), exercice 8.5). Les arbres de régression et de classification sont des estimateurs connus pour être instables. Ce sont des bons candidats pour être baggé. La plupart des logiciels utilisent des arbres dans leurs procédures bagging.

2.2 Les forêts aléatoires

Comme son nom l'indique, une forêt aléatoire consiste à agréger des arbres de discrimination ou de régression.

Définition 2.1
Soit $\{h(\mathbf{x}, \theta_1), \ldots, h(\mathbf{x}, \theta_B)\}$ *une collection de prédicteurs par arbre où* $(\theta_1, \ldots, \theta_B)$ *est une suite de variables aléatoires i.i.d. Le prédicteur des forêts aléatoires est obtenu par agrégation de cette collection d'arbres.*

Une forêt aléatoire n'est donc ni plus ni moins qu'une agrégation d'arbres dépendants de variables aléatoires. Par exemple, bagger des arbres (construire des arbres sur des échantillons bootstrap) définit une forêt aléatoire. Une famille de forêt aléatoire se distingue parmi les autres, notamment de part la qualité de ses performances sur de nombreux jeux de données. Il s'agit des Random Forests-RI (voir Breiman & Cutler (2005)). Dans de nombreux travaux, le terme forêts aléatoires est d'ailleurs souvent employé pour cette famille. C'est ce que nous allons faire dans la suite.

2.2.1 Les random forests RI

Nous présentons dans cette partie la famille des random forests RI. Pour une étude plus approfondie, le lecteur pourra se référer à Genuer (2010). Pour ce type de forêt aléatoire, les arbres sont construits avec l'algorithme CART. Le principe de CART est de partitionner récursivement l'espace engendré par les variables explicatives (ici $\mathbb{R}^p$) de façon dyadique. Plus précisément, à chaque étape du partitionnement, on découpe une partie de l'espace en deux sous parties selon une variable X_j (voir Figure 2.1).

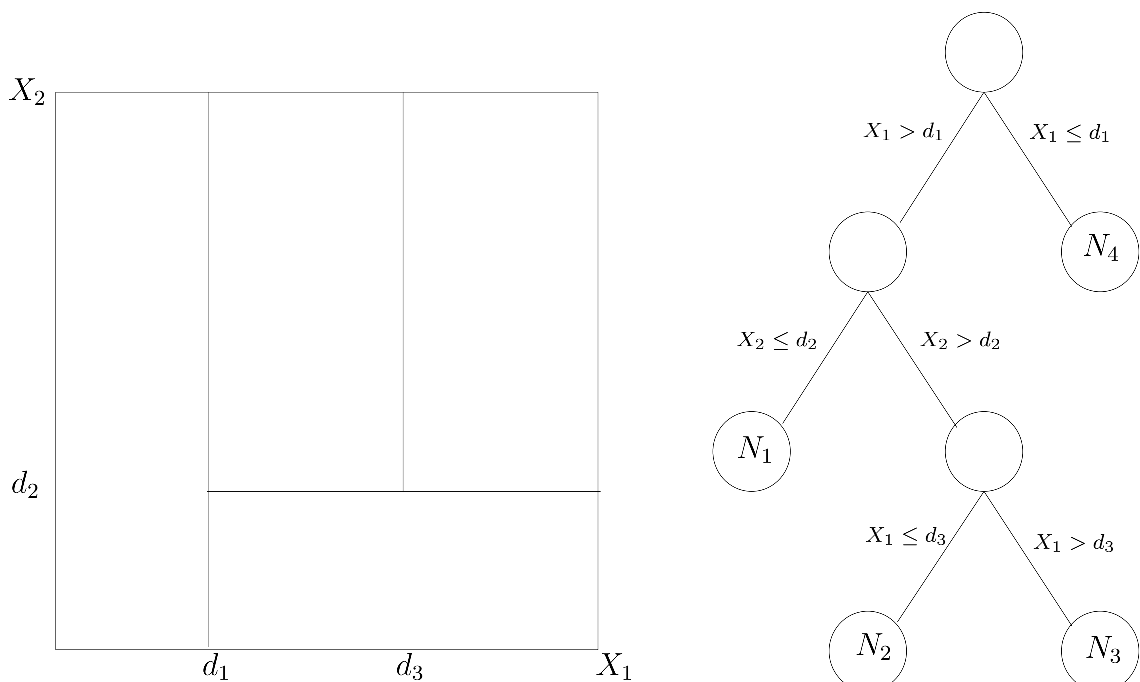

Figure 2.1 – Arbre CART.

Les coupures sont choisies de manière à minimiser une fonction de coût particulière. A chaque étape, on cherche la variable X_j et le réel d qui minimisent :

- la variance des nœuds fils en régression ;
- l'indice de Gini des nœuds fils en classification.

Les arbres sont ainsi construits jusqu'à atteindre une règle d'arrêt. Il en existe plusieurs, nous citons ici simplement celle proposée par le package randomForest : on ne découpe pas un nœud qui contient moins d'observations qu'un nombre fixé au préalable (par défaut randomForest fixe ce nombre à 5 pour la régression et à 1 pour la classification).

Dans la méthode de construction des forêts aléatoires que nous présentons ici, les arbres sont construits selon une variante de CART. Nous avons vu dans la partie précédente que le bagging est d'autant plus performant que la corrélation entre les prédicteurs est faible. Afin de diminuer cette corrélation, Breiman propose de rajouter une couche d'aléa dans la construction des prédicteurs (arbres). Plus précisément, à chaque étape de CART, m variables sont sélectionnées aléatoirement parmi les p et la meilleure coupure est sélectionnée **uniquement** sur ces m variables (voir algorithme 4).

Algorithme 4 Forêts aléatoires

Entrées :

- $\mathbf{x}$ l'observation à prévoir ;
- d_n l'échantillon ;
- B le nombre d'arbres ;
- $m \in \mathbb{N}^\star$ le nombre de variables candidates pour découper un nœud.

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

1. Tirer un échantillon bootstrap dans d_n
2. Construire un arbre CART sur cet échantillon bootstrap, chaque coupure est sélectionnée en minimisant la fonction de coût de CART sur un ensemble de m variables choisies au hasard parmi les p. On note $h(., \theta_k)$ l'arbre construit.

Sortie : L'estimateur $h(\mathbf{x}) = \frac{1}{B}\sum_{k=1}^{B} h(\mathbf{x}, \theta_k)$.

Remarque

- Si nous sommes dans un contexte de discrimination, l'étape finale d'agrégation dans l'algorithme consiste à faire voter les arbres à la majorité.
- On retrouve un compromis biais-variance dans le choix de m :
  - lorsque m diminue, la tendance est à se rapprocher d'un choix "aléatoire" des variables de découpe des arbres. Dans le cas extrême où $m = 1$, les axes de la partition des arbres sont choisies au "hasard", seuls les points de coupure utiliseront l'échantillon. Ainsi, si m diminue, la corrélation entre les arbres va avoir tendance à diminuer également, ce qui entraînera une baisse de la variance de l'estimateur agrégé. En revanche, choisir les axes de découpe des arbres de manière (presque) aléatoire va se traduire par une moins bonne qualité d'ajustement des arbres sur l'échantillon d'apprentissage, d'où une augmentation du biais pour chaque arbre ainsi que pour l'estimateur agrégé.
  - lorsque m augmente, les phénomènes inverses se produisent.

  On déduit de cette remarque que le choix de m est lié aux choix des paramètres de l'arbre, notamment au choix du nombre d'observations dans ses nœuds terminaux. En effet, si ce nombre est petit, chaque arbre aura un biais faible mais une forte variance. Il faudra dans ce cas là s'attacher à diminuer cette variance et on aura donc plutôt tendance à choisir un m relativement faible. A l'inverse, si les arbres ont un grand nombre d'observations dans leurs nœuds terminaux,

ils posséderont moins de variance mais un biais plus élevé. Dans ce cas, la procédure d'agrégation se révélera moins efficace. C'est pourquoi, en pratique, le nombre maximum d'observations dans les nœuds est par défaut pris relativement petit (5 en régression, 1 en classification). Concernant le choix de m, **randomForest** propose par défaut $m = p/3$ en régression et $m = \sqrt{p}$ en classification. Ce paramètre peut également être sélectionné via des procédures apprentissage-validation ou validation croisée.

2.2.2 Erreur Out Of Bag et importance des variables

Parmi les nombreuses sorties proposées par la fonction **randomForest**, deux se révèlent particulièrement intéressantes.

L'erreur Out Of Bag

Il s'agit d'une procédure permettant de fournir un estimateur des erreurs :
- $\mathbf{E}[(\hat{m}(\mathbf{X}) - Y)^2]$ en régression ;
- $\mathbf{P}(\hat{m}(\mathbf{X}) \neq Y)$ en classification.

De tels estimateurs sont souvent construits à l'aide de méthode apprentissage-validation ou validation croisée. L'avantage de la procédure Out Of Bag (OOB) est qu'elle ne nécessite pas de découper l'échantillon. Elle utilise le fait que les arbres sont construits sur des estimateurs baggés et que, par conséquent, ils n'utilisent pas toutes les observations de l'échantillon d'apprentissage. Nous la détaillons dans le cadre des forêts aléatoires mais elle se généralise à l'ensemble des méthodes bagging.

Etant donné une observation $(\mathbf{X}_i, Y_i)$ de $\mathcal{D}_n$, on désigne par $\mathcal{I}_B$ l'ensemble des arbres de la forêt qui ne contiennent pas cette observation dans leur échantillon bootstrap. Pour estimer, la prévision de la forêt sur Y_i on agrège uniquement ces arbres là :

$$\hat{Y}_i = \frac{1}{|\mathcal{I}_B|} \sum_{k \in I_B} h(\mathbf{X}_i, \theta_k).$$

Si on est dans un contexte de classification, la prévision s'obtient en faisant voter ces arbres à la majorité. L'erreur Out Of Bag est alors définie par :
- $\frac{1}{n}\sum_{i=1}^n (\hat{Y}_i - Y_i)^2$ en régression ;
- $\frac{1}{n}\sum_{i=1}^n \mathbf{1}_{\hat{Y}_i \neq Y_i}$ en classification.

Importance des variables

Que ce soit en régression ou discrimination, il est souvent crucial de déterminer les variables explicatives qui jouent un rôle important dans le modèle. L'inconvénient des méthodes d'agrégation est que le modèle construit est difficilement interprétable, on parle souvent d'aspect "boite noire" pour ce type de méthodes. Pour le modèle de forêts aléatoires que nous venons de présenter, Breiman & Cutler (2005) proposent une mesure qui permet de quantifier l'importance des variables $X_j, j = 1, \dots, p$ dans le modèle.

On désigne par OOB_k l'échantillon Out Of Bag associé au $k^{\text{ème}}$ arbre de la forêt. Cet échantillon est formé par les observations qui ne figurent pas dans le $k^{\text{ème}}$ échantillon bootstrap. On note

E_{OOB_k} l'erreur de prédiction de l'arbre $h(.,\theta_k)$ mesurée sur cet échantillon :

$$E_{OOB_k} = \frac{1}{|OOB_k|} \sum_{i \in OOB_k} (h(X_i,\theta_k) - Y_i)^2.$$

On désigne maintenant par OOB_k^j l'échantillon OOB_k dans lequel on a perturbé aléatoirement les valeurs de la variable j et par $E_{OOB_k^j}$ l'erreur de prédiction de l'arbre $h(.,\theta_k)$ mesurée sur cet échantillon :

$$E_{OOB_k}^j = \frac{1}{|OOB_k^j|} \sum_{i \in OOB_k^j} (h(X_i^j,\theta_k) - Y_i)^2,$$

où les X_i^j désignent les observations perturbées de OOB_k^j. Heuristiquement, si la $j^{\text{ème}}$ variable joue un rôle déterminant dans la construction de l'arbre $h(.,\theta_k)$, alors une permutation de ces valeurs dégradera fortement l'erreur. La différence d'erreur $E_{OOB_k}^j - E_{OOB_k}$ sera alors élevée. L'importance de la $j^{\text{ème}}$ variable sur la forêt est mesurée en moyennant ces différences d'erreurs sur tous les arbres :

$$Imp(X_j) = \frac{1}{B}\sum_{k=1}^{B}(E_{OOB_k}^j - E_{OOB_k}).$$

2.2.3 Un exemple avec R

On souhaite comparer différentes méthodes d'apprentissage à la reconnaissance de spam. On dispose d'un échantillon de taille 4 601 comprenant 57 variables explicatives $(V_1 \dots, V_{57})$ et la variable à expliquer Y qui prend pour valeur 1 si le mail est un spam, 0 sinon. Les variables explicatives représentent pour la plupart la fréquence de différents mots utilisés dans le mail. Pour plus d'information sur les variables explicatives utilisées, on pourra se référer à l'URL :

http ://www-stat.stanford.edu/~tibs/ElemStatLearn/

Nous séparons l'échantillon en un échantillon d'apprentissage de taille 2 300 pour construire les modèles et un échantillon test de taille 2 301 pour comparer leurs performances. Nous construisons dans un premier temps une forêt aléatoire en utilisant les paramètres par défaut de la fonction **randomForest** :

```
> library(randomForest)
> mod_RF <- randomForest(Y~.,data=dapp1)
```

On ordonne les variables selon leur mesure d'importance :

```
> imp <- importance(mod_RF)
> order(imp,decreasing=TRUE)
 [1] 52 53  7 55 16 56 21 25 57 24  5 19 23 26 27 46 11 37  8 12  3 50  6 45 18
[26] 17 10  2 28 42 49  1 36 35 13 54  9 30 39 51 33 22 29 14 44 31 43 15 20 48
[51] 41 40  4 32 34 38 47
```

On obtient l'erreur OOB de la forêt via

```
> mod_RF$err.rate[500,1]
       OOB
0.05086957
```

On compare cette erreur avec l'erreur estimée sur l'échantillon de validation :

```
> prev <- predict(mod_RF,newdata=dtest)
> sum(prev!=dtest$Y)/nrow(dtest)
[1] 0.05345502
```

L'estimation OOB est légèrement optimiste sur cet exemple.

Nous allons maintenant comparer les performances des forêts aléatoires avec la règle des k plus proches voisins, les arbres de classification (construits avec **rpart**) et `adaboost`. Pour la règle des k plus proches voisins, on choisit le nombre de voisins qui minimise le taux de mal classés sur l'échantillon d'apprentissage :

```
> library(class)
> K <- seq(1,50,by=1)
> err <- K
> ind <- 0
> for (i in K){
+   ind <- ind+1
+   mod_ppv <- knn(train=dapp[,-58],test=dtest[,-58],cl=dapp$Y,k=K[ind])
+   err[ind] <- sum(mod_ppv!=dtest[,58])/nrow(dtest)
+ }
> min(err)
[1] 0.2007823
```

On obtient près de 20% d'erreur : on retrouve bien que les performances de la règle des plus proches voisins sont faibles en grande dimension. Construisons maintenant un arbre selon **rpart** :

```
> library(rpart)
> dapp1 <- dapp
> dapp1$Y <- as.factor(dapp1$Y)
> arbre <- rpart(Y~.,data=dapp1)
> par(mfrow=c(1,2))
> plot(arbre)
> text(arbre,pretty=0)
> plotcp(arbre)
```

L'arbre est sélectionné en minimisant un critère d'erreur (estimé par validation croisée) en fonction d'un paramètre appelé `cp`. Nous représentons sur la figure 2.2 l'arbre construit ainsi que le critère estimé en fonction de `cp`.

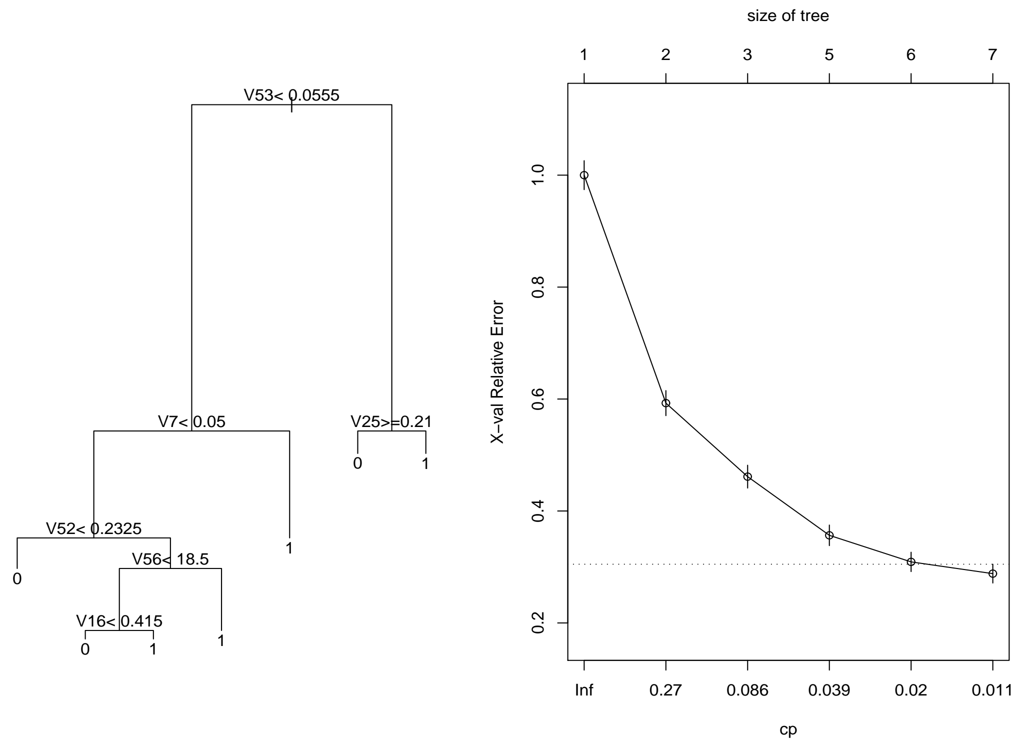

FIGURE 2.2 – Arbre **rpart** (gauche) et estimation du paramètre cp par validation croisée (droite).

On peut remarquer que les 6 variables utilisées pour construire l'arbre se trouvent parmi les 8 variables les plus importantes de la forêt construite précédemment. Nous voyons par ailleurs que les valeurs de `cp` prises par défaut ne permettent pas de minimiser le critère d'erreur. Il faut augmenter la plage de valeurs de `cp` sur laquelle la minimisation est calculée.

```
> prev_arbre <- predict(arbre1,newdata=dtest,type="class")
> err_arbre <- sum(prev_arbre!=dtest$Y)/nrow(dtest)
> err_arbre
[1] 0.08518036
```

On obtient pour cet arbre un taux d'erreur estimé de 0.085.

Pour terminer, on mesure les performances des forêts aléatoires et `adaboost` en utilisant 2 500 itérations.

```
> mod_RF1 <- randomForest(Y~.,data=dapp1,ntree=2500,xtest=dtest[,-58],
                                          ytest=as.factor(dtest[,58]))
> model <- gbm(Y~.,data=dapp,distribution="adaboost",n.trees=2500,
                                          interaction.depth=2,shrinkage=0.05)
> B <- 2500
> errapp <- rep(0,B)
> err_boost <- errapp
> boucle <- seq(1,B,by=50)
> errtest <- rep(0,length(boucle))
> k <- 0
> for (i in boucle){
+   k <- k+1
+   prev_test<- predict(model,newdata=dtest,n.trees=i)
+   err_boost[k] <- sum(as.numeric(prev_test>0)!=dtest$Y)/nrow(dtest)
+ }
```

On représente les taux d'erreurs estimés pour `adaboost` et les forêts aléatoires en fonction du nombre d'itérations. On compare ces taux d'erreurs à ceux de l'arbre **rpart** et de la méthode des k-plus proches voisins (on choisit k qui minimise l'erreur estimée sur l'échantillon de validation).

```
> plot(1:1500,mod_RF1$test$err.rate[1:1500,1],type="l",ylim=c(0.03,0.23),
                                      ylab="erreur",xlab="nombre d'iterations")
> lines(boucle,err_boost,col="red")
> abline(h=min(err),col="blue")
> abline(h=err_arbre,col="green")
```

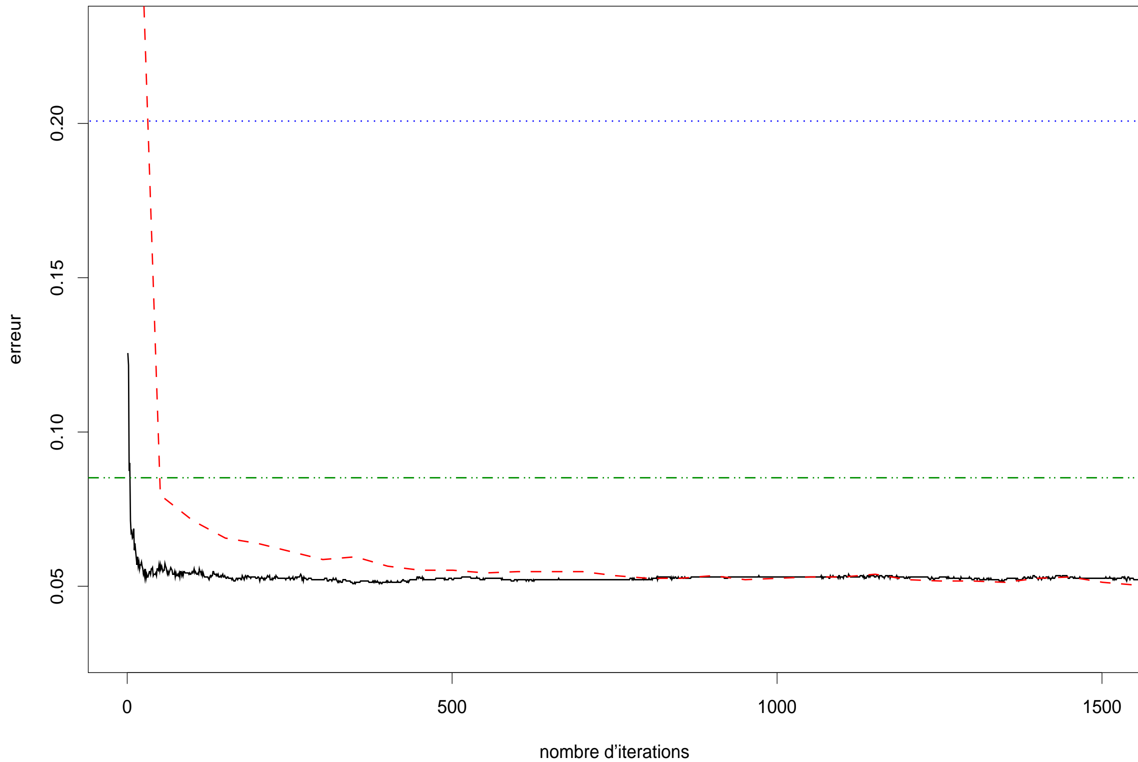

FIGURE 2.3 – Pourcentage de mal classés de la forêt aléatoire (noir, trait plein), de Adaboost (rouge, tirets), de l'arbre **rpart** (vert, points tirets) et de la méthode des k plus proches voisins (bleu, pointillés).

Sur cet exemple, la forêt aléatoire se stabilise rapidement, elle se révèle autant performante que `adaboost` et meilleure que l'arbre **rpart** ainsi que la méthode des k plus proches voisins qui, comme on pouvait s'y attendre, ne se révèle que guère performante en grande dimension. Nous récapitulons dans le tableau 2.1 les erreurs des 4 approches mises en compétition (pour `adaboost` et la forêt aléatoire, nous donnons l'erreur associée au nombre d'itérations qui minimise le taux de mal classés).

Méthode	% de mal classés
k-ppv	0.201
arbre	0.085
adaboost	0.050
forêt	0.051

TABLE 2.1 – Estimation des taux de mal classés pour les 4 approches.

Bibliographie

Bartlett P. & Traskin M. (2007). Adaboost is consistent. *Journal of Machine Learning Research*, **8**, 2347–2368.

Biau G. & Devroye L. (2010). On the layered nearest neigbour estimate, the bagged nearest neigbour estimate and the random forest method in regression and classification. *Journal of Multivariate Analysis*, **101**, 2499–2518.

Breiman L. (1996). Bagging predictors. *Machine Learning*, **26**(2), 123–140.

Breiman L. & Cutler A. (2005). Random forests. Available at http ://stat.berkeley.edu/~breiman/RandomForests/.

Bühlmann P. & Yu B. (2003). Boosting with the l_2 loss : Regression and classification. *Journal of American Statistical Association*, **98**, 324–339.

Freund Y. & Schapire R. (1996). Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning.*

Freund Y. & Schapire R. (1997). A decision-theoritic generalization of online learning and an application to boosting. *Journal of Computer and System Sciences*, **55**, 119–139.

Freund Y. & Schapire R. (1999). A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, **14**(5), 771–780.

Genuer R. (2010). *Forêts aléatoires : aspects théoriques, sélection de variables et applications.* Ph.D. thesis, Université Paris XI.

Hastie T., Tibshirani R. & Friedman J. (2009). *The Elements of Statistical Learning : Data Mining, Inference, and Prediction.* Springer, second edn..