

Chapitre 5

Forêts aléatoires

Les commandes utilisées dans ce chapitre font appel aux packages suivants

```
> #library(randomForest)
> library(ranger)
> library(kernlab)
> library(OOBCurve)
> library(tidymodels)
> library(vip)
> library(rpart)
```

Nous présentons dans cette section un des algorithmes les plus utilisés en apprentissage supervisée : les **forêts aléatoires**. La construction de la méthode sera illustrée à travers le jeu de données **spam** présenté dans la section [1.2.4](#)

```
> data(spam)
> dim(spam)
## [1] 4601 58
> summary(spam$type)
## nonspam spam
## 2788 1813
```

Le problème est de prédire la variable binaire **type** par les 57 autres variables du jeu de données.

5.1 Bagging

Le terme *bagging* ([Breiman \(1996\)](#)) vient de la contraction de *Bootstrap AGGREGATING* et désigne un ensemble de méthodes permettant d'obtenir des algorithmes de prévision en agrégeant d'autres algorithmes. Plusieurs façons d'agréger peuvent

être envisagées, nous considérons ici un algorithme de prévision

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

qui s'écrit comme la moyenne d'autres algorithmes $T_1(x), \dots, T_B(x)$. Nous proposons d'évaluer l'intérêt d'un tel procédé en comparant les performances de f_n (l'algorithme final) à celles des T_b (les algorithmes que l'on agrège). Rappelons qu'un algorithme peut être vu comme l'estimateur d'une fonction inconnue, par exemple la fonction de régression $\mathbf{E}[Y|X=x]$ en régression ou les probabilités a posteriori $\mathbf{P}(Y=j|X=x), j=1, \dots, K$ en classification. La performance d'estimateur se résume souvent à l'étude du compromis biais/variance, nous proposons donc d'étudier ces quantités pour f_n et les T_b . Il est bien entendu possible d'utiliser plusieurs types algorithmes T_b : une régression linéaire pour T_1 , un arbre pour T_2 , une SVM pour T_3 , etc. . . Un tel procédé laisserait beaucoup trop de choix à l'utilisateur pour entraîner les différents algorithmes et rendrait l'analyse de l'estimateur final très complexe. C'est pourquoi nous proposons de construire les T_b de la même façon. Chaque T_b peut par exemple être la règle du 1 plus proche voisin, ou un arbre CART utilisant une unique procédure pour choisir la profondeur. . . Cela revient à supposer que les variables aléatoires T_b sont identiquement distribuées. Sous cette hypothèse, les T_b ont tous le même biais et la même variance et on peut montrer que (voir exercice 5.1).

$$\mathbf{E}[f_n(x)] = \mathbf{E}[T_1(x)] \quad \text{et} \quad \mathbf{V}[f_n(x)] = \rho(x)\mathbf{V}[T_1(x)] + \frac{1-\rho(x)}{B}\mathbf{V}[T_1(x)], \quad (5.1)$$

où $\rho(x) = \text{corr}(T_1(x), T_2(x))$ est le coefficient de corrélation entre les prévisions de 2 algorithmes au même point x . Plusieurs messages importants se déduisent de ces résultats. Du point de vue du biais, la procédure d'agrégation n'est d'aucun intérêt puisque l'espérance de l'algorithme agrégé est la même que celle des algorithmes qu'on agrège. L'éventuel gain se mesure donc à travers la variance. Lorsque B est grand on a $\mathbf{V}[f_n(x)] \approx \rho(x)\mathbf{V}[T_1(x)]$, cela signifie que la variance des $T_b(x)$ est diminué d'un facteur proportionnel à $\rho(x)$. Ce terme est positif et varie entre 0 et 1 (voir exercice 5.1). Supposons, qu'en plus d'être identiquement distribué, les T_b sont indépendants. Rappelons que l'aléa des T_b provient de l'aléa des échantillons sur lesquels ils sont entraînés. L'indépendance suppose donc qu'on entraîne chaque T_b sur des sous-échantillons disjoints de l'échantillon initial. On a dans ce cas une corrélation $\rho(x)$ nulle et donc une variance nulle pour f_n . Néanmoins, une telle situation implique que T_b soient entraînés sur des échantillons de très petites tailles, ils possèdent donc un biais et une variance élevées. Le biais de f_n serait donc également très grand, ce qui n'est pas satisfaisant. On peut également considérer la situation où tous les T_b sont entraînés sur les données initiales. Chaque T_b prédit dans ce cas la même chose et $f_n(x)$ est donc équivalent à $T_1(x)$ et l'agrégation n'est d'aucune utilité.

Le bagging va proposer un compromis entre ces deux situations extrêmes en entraînant chaque algorithme sur des échantillons bootstrap, c'est-à-dire des

échantillons de taille n obtenus à partir de tirages avec remise dans l'échantillon initial. L'idée est de diminuer la corrélation entre les prévisions des T_b en ajoutant de l'aléa issu des tirages bootstrap. La méthode est présentée dans l'algorithme 5.1.

Algorithme 5.1 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 θ_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 $\mathcal{D}_{n,b}^* \Rightarrow T(\cdot, \theta_b, \mathcal{D}_n)$.

Retourner :

- *Régression* : $f_n(x) = \frac{1}{B} \sum_{b=1}^B T(x, \theta_b, \mathcal{D}_n)$.
 - *Classification* : $f_{n,j}(x) = \frac{1}{B} \sum_{b=1}^B \mathbf{1}_{T(x, \theta_b, \mathcal{D}_n)=j}, j = 1, \dots, K$.
-

L'algorithme final renvoie la moyenne des prévisions des T_b en régression. Pour la classification, il estime les probabilités $\mathbf{P}(Y = j|X = x), j = 1, \dots, K$ par les proportions d'algorithmes T_b qui ont classé dans le groupe j . L'écriture $T(\cdot, \theta_b, \mathcal{D}_n)$ permet de distinguer les deux sources d'aléas de l'algorithme : l'aléa traditionnel des données avec \mathcal{D}_n et l'aléa des tirages bootstrap avec θ_b . Ce dernier aléa implique qu'on peut obtenir des prévisions différentes en lançant deux fois cet algorithme sur les mêmes données. La loi des grands nombre permet de nuancer ce constat, en effet

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

où \mathbf{E}_θ désigne l'espérance calculée par rapport à la loi de θ uniquement, c'est-à-dire conditionnellement à \mathcal{D}_n . Ce résultat permet de conclure que, lorsque B est grand, les prévisions de deux algorithmes bagging construits sur des échantillons bootstrap différents convergent vers une même prévision $\bar{f}_n(x, \mathcal{D}_n)$ qui ne dépend plus des tirages bootstrap. Il est ainsi recommandé de choisir B le plus grand possible afin de contrôler l'aléa bootstrap. L'utilisateur doit également choisir l'algorithme à entraîner sur les échantillons bootstrap. Nous avons vu que les prévisions sont d'autant plus performantes que la corrélation $\rho(x) = \text{corr}(T(x, \theta_1, \mathcal{D}_n), T(x, \theta_2, \mathcal{D}_n))$ est petite. La seule différence entre $T(x, \theta_1, \mathcal{D}_n)$ et $T(x, \theta_2, \mathcal{D}_n)$ est le tirage bootstrap. Ces deux prévisions correspondent au même algorithme entraînés sur des échantillons obtenus en dupliquant et supprimant quelques observations dans l'échantillon initial. Utiliser des algorithmes robustes vis-à-vis de légères perturbations de l'échantillon sera donc d'une utilité limitée

puisque les prévisions de tels algorithmes sont peu affectées par les tirages bootstrap. Les régressions linéaires et logistiques sont par exemple connues pour posséder une telle robustesse et il n'est pas courant de les bagger (voir exercice 5.2). Un des reproches souvent fait aux arbres est justement une instabilité par rapport à de légères perturbations de l'échantillon. En effet les arbres sont construits en répétant des coupures binaires de \mathbb{R}^p . Perturber les données peut engendrer des changements de coupure en haut de l'arbre qui vont donc modifier les coupures suivantes et par conséquent toute la structure de l'arbre. Cette instabilité devient un avantage pour le bagging, les arbres sont en effet souvent utilisés pour cette procédure. Les forêts aléatoires présentées dans la section suivante s'inscrivent dans ce cadre.

5.2 Forêts aléatoires

Comme le nom l'indique, les *forêts aléatoires* agrègent des prédicteurs par arbres construits sur des échantillons bootstrap. Il existe différents processus d'agrégation, nous nous focalisons sur les forêts aléatoires proposées par Breiman (2001) qui sont de loin les plus utilisées. On pourra trouver des variantes de cet algorithme dans Poggi et Genuer (2019).

On rappelle qu'un arbre CART (voir chapitre 4) s'obtient en découpant de façon récursive des nœuds selon des règles $X_j \leq s$ où la variable de coupure X_j et le seuil s sont obtenus en maximisant le gain d'impureté entre le nœud père et ses deux nœuds fils sur toutes les variables et toutes les valeurs de seuil. Une variante va être utilisée pour construire les arbres d'un forêt. La variable de coupure ne sera pas choisie parmi toutes les variables $X_j, j = 1, \dots, d$ mais parmi un sous-ensemble de variables tirées au sort. Ce procédé peut paraître étrange à première vue, il est en réalité très astucieux. Il a en effet pour objectif d'augmenter les différences entre les arbres de la forêt et donc de diminuer la corrélation $\rho(x)$ entre deux prévisions d'arbres. La construction de la forêt est décrite dans l'algorithme 5.2.

Afin de ne pas surcharger les notations, nous avons conservé l'écriture $T(x, \theta_b, \mathcal{D}_n)$ mais le paramètre θ_b contient ici tous les paramètres permettant de caractériser le b^e arbre de la forêt : l'échantillon bootstrap, les coupures sélectionnées... Précisons également que les `mtry` variables candidates pour découper un nœud ne sont pas sélectionnées une seule fois : on tire `mtry` variables au hasard avant de découper chaque nœud.

Les packages `randomForest` et `ranger` peuvent être utilisés pour ajuster des forêts aléatoires. `randomForest` est le plus ancien et certainement le plus utilisé. Le package `ranger` codé en C++ est plus efficace au niveau des temps de calcul. Nous proposons d'illustrer la méthode avec ce package.

```
> set.seed(12345)
> foret <- ranger(type~., data=spam)
> foret
```

Algorithme 5.2 Forêt aléatoire.**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 si il contient moins de `min.node.size` observations.
3. On note $T(., \theta_b, \mathcal{D}_n)$ l'arbre obtenu.

Retourner :

- *Régression* : $f_n(x) = \frac{1}{B} \sum_{b=1}^B T(x, \theta_b, \mathcal{D}_n)$.
- *Classification* : $f_{n,j}(x) = \frac{1}{B} \sum_{b=1}^B \mathbf{1}_{T(x, \theta_b, \mathcal{D}_n)=j}, j = 1, \dots, K$.

```
## Ranger result
##
## Call:
## 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:                 4.48 %
```

On retrouve dans l'objet `foret` plusieurs informations sur l'algorithme. Le type de forêt `Classification` car la variable à expliquer (`type`) est qualitative. Si elle avait été de classe `numeric`, on aurait eu une forêt de régression. On peut ensuite lire le nombre d'arbres de la forêt (B) ainsi que la taille d'échantillon (4601) et le nombre de variables explicatives (57). Viennent après les nombres

de variables choisies au hasard pour découper les nœuds (`Mtry` qui vaut 7) et le nombre d'observations minimal dans les nœuds terminaux (`Target node size`). On remarque qu'il vaut 1, cela signifie que les arbres de la forêt sont de profondeur maximale. On lit enfin dans `Splitrule` le critère d'impureté utilisé pour découper les nœuds, l'impureté de `gini` est utilisée par défaut en classification. Une estimation de l'erreur de classification est enfin précisée dans `Oob prediction error`. Cette dernière estimation est calculée par une méthode spécifique aux algorithmes bagging appelée `Out Of Bag`. Nous l'expliquerons dans la section 5.4.1.

5.3 Choix des paramètres

L'algorithme 5.2 dépend de paramètres que l'utilisateur doit choisir. Le premier est le nombre d'arbres B . Nous avons vu que ce paramètre devait être le plus grand possible. En pratique il faudra donc s'assurer que la forêt possède suffisamment d'arbres pour se trouver dans son régime de convergence. Une manière de procéder est de regarder l'évolution des erreurs OOB en fonction du nombre d'arbres. On peut par exemple obtenir l'erreur de classification et l'AUC avec

```
> set.seed(12345)
> foret <- ranger(type~., data=spam, keep.inbag=TRUE)
> spam.task <- mlr::makeClassifTask(data=spam, target="type")
> erreurs <- OOBCurve(foret, measures = list(mmce, auc),
+                   task=spam.task, data=spam)
> erreurs1 <- erreurs %>% as_tibble() %>% mutate(ntrees=1:500) %>%
+   filter(ntrees>=5) %>%
+   pivot_longer(-ntrees, names_to="Erreur", values_to="valeur")
> ggplot(erreurs1)+aes(x=ntrees, y=valeur)+geom_line()+
+   facet_wrap(~Erreur, scales="free")
```

On observe sur la figure 5.1 que les erreurs sont stables, nous pouvons donc considérer que 500 arbres sont suffisants. Les autres paramètres vont influencer le compromis biais variance de la forêt, et donc sa performance. Nous avons représenté sur la figure 5.2 des erreurs de classification estimées par validation hold out pour différentes valeurs de `mtry` et `min.node.size`. Les erreurs ont été calculées en séparant les données en un échantillon d'apprentissage de taille 3000 et un échantillon test de taille 1601. Ce processus a été répété sur 150 coupures différentes pour stabiliser les erreurs.

Les 2 paramètres ont une influence sur la performance de la forêt. En effet l'erreur a tendance à décroître avec la profondeur des arbres, plus les arbres sont profonds (`min.node.size` petit), plus les erreurs sont petites. On peut expliquer cela en revenant au compromis biais variance représenté par (5.1). Les agrégations bagging permettent de réduire la variance des arbres que l'on agrège, en aucun cas le biais. Il est donc nécessaire d'utiliser le bagging avec des algorithmes qui possèdent une grande variance et peu de biais, en l'occurrence des arbres profonds. Cela signifie qu'il n'est pas nécessaire d'élaguer les arbres de

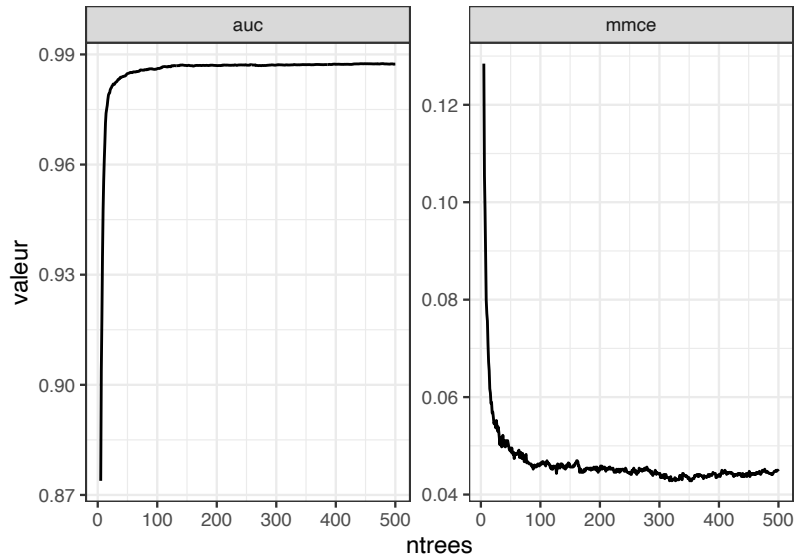


FIGURE 5.1 – Erreurs de classification (gauche) et AUC (droite) en fonction du nombre d’arbres.

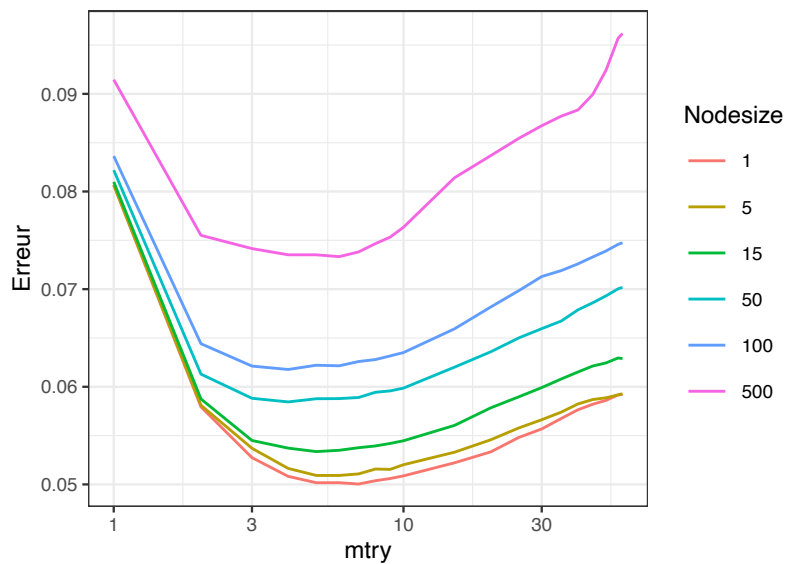


FIGURE 5.2 – Erreurs de classification en fonction de m et de la profondeur des arbres.

la forêt. Il faut même ne pas le faire : la forêt sera plus efficace en agrégeant des arbres peu performants qui sur-ajustent qu'en agrégeant des arbres "optimaux". Des petites valeurs de `min.node.size` sont par conséquent proposées par défaut dans `ranger` : 1 pour la classification et 5 pour la régression.

L'influence de `mtry` peut également se mesurer avec (5.1). Ce paramètre possède une influence influer sur le biais et la variance des arbres de la forêt mais aussi la corrélation $\rho(x)$ entre deux arbres. La figure 5.3 compare les erreurs d'ajustement et de prévision de la forêt en fonction de `mtry`. On visualise des courbes typiques du phénomène de sur-ajustement qui peut s'expliquer en analysant l'influence de ce paramètre sur le biais et la variance de la forêt :

- **mtry petit** signifie que peu de variables sont candidates pour découper les nœuds. La variable de coupure est même choisie au hasard lorsque `mtry=1`. Il est donc plus difficile pour chaque arbre de la forêt de bien ajuster les données, notamment celles qui ne sont pas dans l'échantillon bootstrap. C'est pourquoi l'erreur d'ajustement (voir figure 5.3), et donc le biais, est grande lorsque `mtry` est petit. Au niveau de la variance, on peut faire le constat que $\mathbf{V}[T(x, \theta, \mathcal{D}_n)]$ sera toujours élevée car les arbres sont profonds, quel que soit `mtry`. Néanmoins, prendre des petites valeurs pour `mtry` permet de diminuer la corrélation entre deux arbres de la forêt et par conséquent la variance de la forêt ;
- **mtry grand** signifie à l'inverse qu'un grand nombre de variables sont candidates pour découper les nœuds. Cela permet aux arbres de mieux ajuster les données et donc de diminuer le biais. On a en revanche une corrélation entre deux arbres d'une même forêt plus élevée, ce qui augmente la variance de la forêt.

Le sur-ajustement risque donc d'apparaître lorsque `mtry` est (trop) grand. Les valeurs par défaut sont \sqrt{d} pour la classification et $d/3$ pour la régression mais l'utilisateur doit calibrer ce paramètre. Cela se fait généralement à partir des méthodes classiques d'estimation de risques de prévision par ré-échantillonnage qui ont été présentées dans le chapitre 3. On propose de choisir les paramètres `nodesize` et `mtry` dans la grille suivante :

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

On estime les erreurs de classification et l'AUC par validation croisée répétée 5 fois en utilisant `tidymodels` (voir section ??)

```
> blocs <- vfold_cv(spam, v = 10, repeats = 5)
> 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 ~ .)
> rf_res <- rf_wf %>% tune_grid(resamples = blocs, grid = rf_grid)
```

et on affiche les paramètres sélectionnés

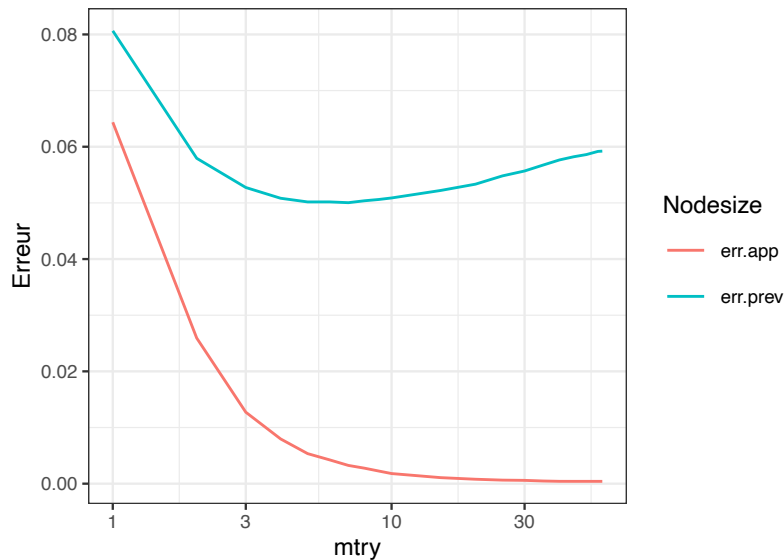


FIGURE 5.3 – Erreur de prévision (calculées sur les données test) et d'ajustement (calculées sur les données d'apprentissage) en fonction de `mtry`.

```
> rf_res %>% show_best("roc_auc")
## # A tibble: 5 x 8
##   mtry min_n .metric .estimator mean      n std_err .config
##   <dbl> <dbl> <chr>   <chr>   <dbl> <int>   <dbl> <chr>
## 1     4     1 roc_auc binary    0.988    50 6.14e-4 Prepro~
## 2     5     1 roc_auc binary    0.988    50 6.23e-4 Prepro~
## 3     6     1 roc_auc binary    0.988    50 6.17e-4 Prepro~
## 4     5     5 roc_auc binary    0.988    50 6.21e-4 Prepro~
## 5     7     1 roc_auc binary    0.988    50 6.45e-4 Prepro~
> rf_res %>% show_best("accuracy")
## # A tibble: 5 x 8
##   mtry min_n .metric .estimator mean      n std_err .config
##   <dbl> <dbl> <chr>   <chr>   <dbl> <int>   <dbl> <chr>
## 1     4     1 accura~ binary    0.954    50 0.00159 Prepro~
## 2     6     1 accura~ binary    0.954    50 0.00141 Prepro~
## 3     7     1 accura~ binary    0.954    50 0.00149 Prepro~
## 4     5     1 accura~ binary    0.954    50 0.00153 Prepro~
## 5     8     1 accura~ binary    0.953    50 0.00146 Prepro~
```

On retrouve bien des petites valeurs pour `nodesize` : il faut des arbres profonds pour que la forêt soit performante. Les valeurs optimales de `mtry` se situent autour de la valeur par défaut (7 ici). On peut donc conserver cette valeur pour ré-ajuster la forêt sur toutes les données.

5.4 Importance des variables et erreur Out Of Bag

5.4.1 Erreur Out Of Bag

Comme pour tous les algorithmes de prévision, il est important d'évaluer la performance d'une forêt aléatoire. Cela peut se faire en utilisant des méthodes comme la validation croisée utilisée dans la section précédente. Le fait d'utiliser des échantillons bootstrap pour ajuster les arbres de la forêt permet de définir une nouvelle méthode : l'estimation *Out Of Bag (OOB)*. Cette technique se base sur les individus qui ne sont pas sélectionnés dans les différents échantillons bootstrap. Plus précisément, on définit pour chaque individu $i = 1, \dots, n$,

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

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. Même si cette prévision, n'est pas calculée à partir de tous les arbres de la forêt, elle présente l'avantage de n'utiliser que des arbres qui n'ont pas été entraînés avec i . On obtient une estimation du risque quadratique en confrontant ces prévisions aux valeurs observées. Par exemple, pour le risque quadratique en régression, l'erreur OOB correspond à :

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

Remarque 5.1. On peut procéder de même pour estimer d'autres risques tels que l'erreur de classification ou la courbe ROC. Il suffit de calculer le groupe ou score prédit $f_{n,\text{OOB}(i)}(x_i)$ par les arbres qui n'ont pas utilisé i et d'en déduire le risque en confrontant ces prévisions aux valeurs observées y_i .

Cette méthode peut être vue comme une approche compétitive aux méthodes de ré-échantillonnage (validation hold out, validation croisée...) présentées dans le chapitre 3. Elle présente l'avantage de ne pas avoir à séparer les données en blocs sur lesquelles on entraîne plusieurs fois l'algorithme et se révèle par conséquent moins coûteuse en temps de calcul. Cette erreur est renvoyée par défaut avec **ranger**, l'erreur de classification OOB de la forêt obtenue dans la section 5.2 est par exemple de 4.59% :

```
> foret$prediction.error
## [1] 0.04477288
```

5.4.2 Importance des variables

On reproche souvent aux forêts aléatoires d'avoir un coté “boîte noire”. Il est en effet difficile d'expliquer comment est calculée la prévision d'une forêt puisqu'elle s'obtient à partir d'un grand nombre d'arbres qui sont de plus très profond. Les scores d'importance permettent d'aider l'utilisateur à interpréter l'algorithme en notant les variables en fonction de leur importance dans la construction de la forêt. Deux scores d'importance sont généralement utilisés. Le premier définit l'importance de la variable X_j par l'importance moyenne de cette variable pour chaque arbre :

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

où $\mathcal{I}_j(T_b)$ est l'importance de X_j pour le b^e arbre de la forêt défini par (4.3) (voir section 4.4.1). Cette dernière importance étant calculée à partir des gains d'impureté des coupures de l'arbre, on appelle ce score *score d'impureté*.

Le second score d'importance, appelé *score par permutation*, fait intervenir l'erreur OOB de la section précédente. Plaçons nous en régression et définissons l'erreur OOB du b^e arbre de la forêt par

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

où

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

$\text{Err}(\text{OOB}_b)$ est donc l'erreur quadratique de l'arbre b calculée sur les individus qui n'ont pas servi à la construction de cet arbre. Afin d'évaluer l'importance de la variable $X_j, j = 1, \dots, d$, on effectue une permutation aléatoire de la j^e colonne des observations de l'échantillon OOB_b comme représentée sur la figure 5.4.

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

FIGURE 5.4 – Permutation de la j^e colonne d'un échantillon OOB.

On note \tilde{x}_i^j les individus de l'échantillon OOB_b permuté (\tilde{x}_1^j correspond par exemple au premier individu de l'échantillon de droite sur la figure 5.4) et on recalcule l'erreur OOB avec ce nouvel échantillon :

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

Si X_j a peu d'importance sur le calcul des prévisions du b^e arbre alors les erreurs $\text{Err}(\text{OOB}_b)$ et $\text{Err}(\text{OOB}_b^j)$ seront proches. Si à l'inverse cette variable est très importante pour prédire, alors la permutation aléatoire va dégrader l'erreur. Le score par permutation mesure ainsi l'importance de X_j par l'écart moyen entre ces deux erreurs sur tous les arbres de la forêt :

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

Remarque 5.2. Pour définir l'importance par permutation en classification, il suffit de remplacer les erreurs quadratiques par des erreurs dans classifications dans E_{OOB_b} et $E_{\text{OOB}_b}^j$.

L'option `importance` dans `ranger` permet d'obtenir ces deux scores :

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

On peut utiliser la fonction `vip` du package `vip` pour visualiser les variables les plus importantes (figure 5.5)

```
> vip(foret.imp)
> vip(foret.perm)
```

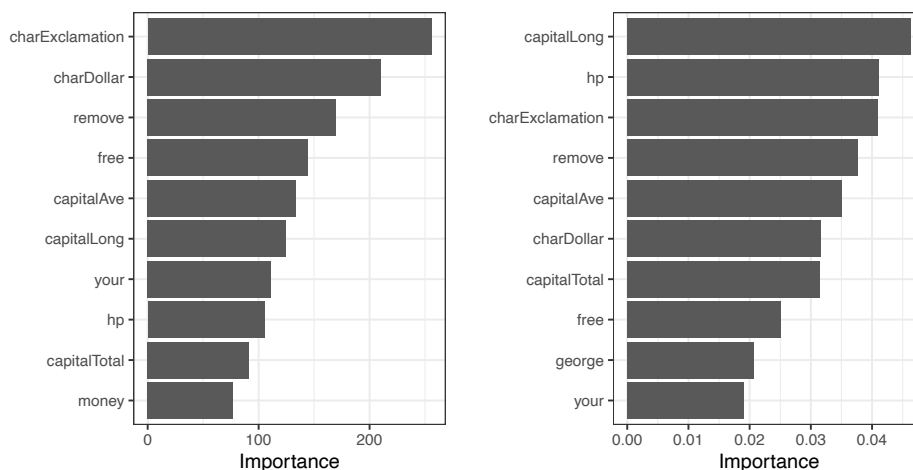


FIGURE 5.5 – Scores d'importance d'impureté (gauche) et par permutation (droite).

Les variables ne sont pas classées dans le même ordre en fonction du score utilisé. On observe néanmoins des tendances similaires puisque 7 variables se retrouvent dans le top 10 des deux scores.

5.5 Exercices

Exercice 5.1 (Biais et variance des algorithmes bagging).

Montrer les égalités (5.1). On prendra également soin de discuter du signe de $\rho(x)$.

Pour simplifier les notations on considère T_1, \dots, T_B B variables aléatoires de même loi et de variance σ^2 . Il est facile de voir que $\mathbf{E}[\bar{T}] = \mathbf{E}[T_1]$. Pour la variance on a

$$\begin{aligned}\mathbf{V}[\bar{T}] &= \frac{1}{B^2} \mathbf{V} \left[\sum_{i=1}^B T_i \right] = \frac{1}{B^2} \left[\sum_{i=1}^B \mathbf{V}[T_i] + \sum_{i \neq j} \mathbf{Cov}(T_i, T_j) \right] \\ &= \frac{1}{B^2} [B\sigma^2 + B(B-1)\rho\sigma^2] = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.\end{aligned}$$

Considérons $\rho \leq 0$. On déduit de l'équation précédente que $B \leq 1 - 1/\rho$. Par exemple si $\rho = -1$, B doit être inférieur ou égal à 2. Il n'est en effet pas possible de considérer 3 variables aléatoires de même loi dont les corrélations 2 à 2 sont égales à -1. De même si $\rho = -1/2$, $B \leq 3$.

Exercice 5.2 (Corrélation bootstrap logistique vs arbre).

On considère le problème de classification binaire à deux dimensions présentés dans la section 1.2.2. On pourra obtenir un échantillon de taille 200 avec :

```
> don <- gen_class_bin2D(n=n,graine=i,bayes=0.25)$donnees
```

On souhaite comparer les corrélations $\rho(T(x, \theta_1, \mathcal{D}_n), T(x, \theta_2, \mathcal{D}_n))$ pour des prévisions par

- modèle logistique avec les paramètres par défaut de la fonction `glm`
- arbre de classification avec comme paramètre `minsplit=3` et `cp=0.00001` pour les arbres.

1. Proposer un algorithme de Monte Carlo permettant d'estimer cette corrélation.

On peut estimer cette corrélation en simulant B (grand) échantillons. Puis pour chaque échantillon on effectue deux tirages bootstrap sur lesquels on entraîne la régression logistique. On calcule ensuite l'estimation de $\mathbf{P}(Y = 1|X = x)$ pour les deux algorithmes ajustés. On estime enfin la corrélation par la corrélation empirique sur les B répétitions.

2. Mettre en œuvre cette procédure et comparer les corrélations entre les prédicteurs logistique et par arbre. On pourra effectuer la comparaison pour la valeur de x suivante :

```
> xnew <- tibble(X1=0.65, X2=0.3)
```

On crée une fonction qui calcule les prévisions des algorithmes sur de nouveaux individus.

```

> calc.cor <- function(B=100,n=200,Xnew){
+   prev.logit <- matrix(0,nrow=B,ncol=2)
+   prev.arbre <- prev.logit
+   prev <- matrix(0,nrow=B,ncol=6) %>% as_tibble()
+   names(prev) <- c("X1","X2","logit1","logit2","arbre1","arbre2")
+   res <- tibble()
+   for (i in 1:B){
+     don <- gen_class_bin2D(n=n,graine=i,bayes=0.25)$donnees
+     theta1 <- sample(n,n,replace=TRUE)
+     theta2 <- sample(n,n,replace=TRUE)
+     D1 <- don[theta1,]
+     D2 <- don[theta2,]
+     logit1 <- glm(Y~.,data=D1,family=binomial)
+     logit2 <- glm(Y~.,data=D2,family=binomial)
+     prev[,3] <- predict(logit1,newdata=Xnew,type="response")
+     prev[,4] <- predict(logit2,newdata=Xnew,type="response")
+     arbre1 <- rpart(Y~.,data=D1,minsplit=3,cp=0.00001)
+     arbre2 <- rpart(Y~.,data=D2,minsplit=3,cp=0.00001)
+     prev[,5] <- predict(arbre1,newdata=Xnew,type="prob")[,2]
+     prev[,6] <- predict(arbre2,newdata=Xnew,type="prob")[,2]
+     prev[,1:2] <- Xnew
+     res <- res %>% bind_rows(prev)
+   }
+   return(res)
+ }

```

On déduit les corrélations avec

```

> aa <- calc.cor(100,200,xnew)
> aa %>% group_by(X1,X2) %>%
+   summarize(logit=cor(logit1,logit2),arbre=cor(arbre1,arbre2))
## # A tibble: 1 x 4
## # Groups:   X1 [1]
##       X1      X2 logit arbre
##   <dbl> <dbl> <dbl> <dbl>
## 1  0.65  0.3 0.461 0.403

```

3. Faire le même travail pour 100 individus x générés aléatoirement sur le carré $[0, 1]$. Comparer les corrélations obtenus à l'aide d'un boxplot.

On génère les individus et calcule toutes les corrélations avec

```

> Xnew <- tibble(X1=runif(100),X2=runif(100))
> bb <- calc.cor(100,200,Xnew)
> bb1 <- bb %>% group_by(X1,X2) %>%
+   summarize(logit=cor(logit1,logit2),arbre=cor(arbre1,arbre2))

```

On peut maintenant les comparer

```

> bb1 %>% pivot_longer(c(logit,arbre),names_to="Algo",
+   values_to="cor") %>%
+   ggplot()+aes(x=Algo,y=cor)+geom_boxplot()

```

On retrouve bien que les corrélations sont plus faibles pour les arbres.

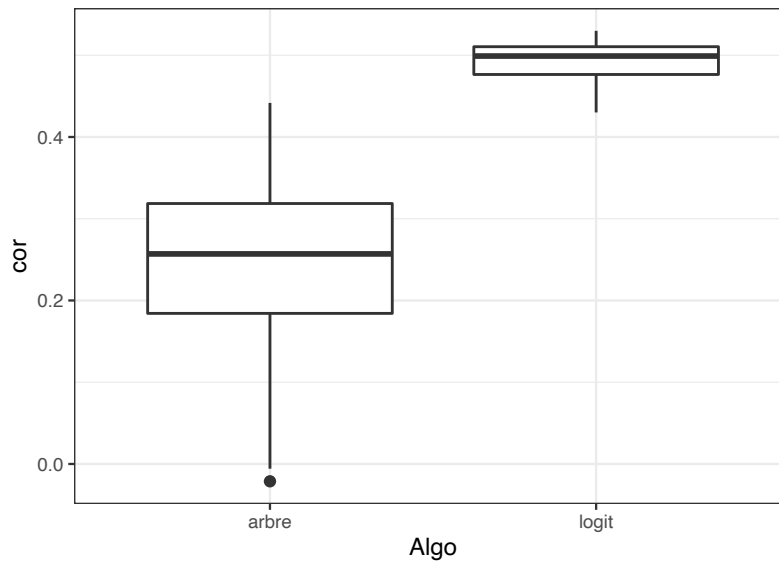


FIGURE 5.6 – Corrélations bootstrap pour algorithmes logistique et arbres.

Exercice 5.3 (Arbre et forêt aléatoire).

```

> nested_cv(spam,
+           outside = vfold_cv(repeats = 1),
+           inside = mc_cv(prop=3/4, times = 2))

> aa <- createDataPartition(1:nrow(spam), times=2)
> bb <- rsample2caret(aa, data = c("analysis", "assessment"))
> ctrl.ho <- trainControl(method="LGOCV", p=2/3, number=2)
> ctrl.ho <- trainControl(method="cv")
> bb <- caret2rsample(ctrl.ho, data = spam)
> bb <- mc_cv(spam, prop = 3/4, times = 2)

```