Méthodes par arbres

L. Rouvière

23 novembre 2022

Table des matières

Présentation

Ce tutoriel présente quelques exercices d'application sur les méthodes par arbres. On pourra trouver

- les supports de cours associés à ce tutoriel ainsi que les données utilisées à l'adresse suivante https://lrouviere.github.io/page_perso/grande_dim.html ;
- le tutoriel sans les corrections à l'url https://lrouviere.github.io/TUTO_ARBRES/
- le tutoriel avec les corrigés (à certains moment) à l'url https://lrouviere.github.io/TUTO_ARBRES/correction.

Il est recommandé d'utiliser **mozilla firefox** pour lire le tutoriel.

Des connaissances de base en R et en statistique (modèles de régression) sont nécessaires. Le tutoriel se structure en 4 parties :

- **Arbres** : construction d'arbres et élagages avec `rpart`
- **Forêts aléatoires** : l'algorithme et le choix des paramètres avec `ranger` et `randomForest`
- **Gradient boosting**: l'algorithme et le choix des paramètres avec `gbm` et `xgboost`

partie I

Arbres

1 Arbres

Les méthodes par arbres sont des algorithmes où la prévision s'effectue à partir de **moyennes locales**. Plus précisément, étant donné un échantillon $(x_1, y_1) \dots, (x_n, y_n)$, l'approche consiste à :

- construire une partition de l'espace de variables explicatives ($\mathbb{R}^p$) ;
- prédire la sortie d'une nouvelle observation x en faisant :
  - la moyenne des y_i pour les x_i qui sont dans la même classe que x si on est en régression ;
  - un vote à la majorité parmi les y_i tels que les x_i qui sont dans la même classe que x si on est en classification.

Bien entendu toute la difficulté est de trouver la "bonne partition" pour le problème d'intérêt. Il existe un grand nombre d'algorithmes qui permettent de trouver une partition. Le plus connu est l'algorithme **CART** (Breiman et al. 1984) où la partition est construite par **divisions successives** au moyen d'hyperplan orthogonaux aux axes de $\mathbb{R}^p$. L'algorithme est récursif : il va à chaque étape séparer un groupe d'observations (**nœuds**) en deux groupes (**nœuds fils**) en cherchant la meilleure variable et le meilleur seuil de coupure. Ce choix s'effectue à partir d'un critère **d'impureté** : la meilleure coupure est celle pour laquelle l'impureté des 2 nœuds fils sera minimale. Nous étudions cet algorithme dans cette partie.

1.1 Coupures CART en fonction de la nature des variables

Une partition CART s'obtient en séparant les observations en 2 selon une coupure parallèle aux axes puis en itérant ce procédé de séparation binaire sur les deux groupes… Par conséquent la première question à se poser est : pour un ensemble de données $(x_1, y_1), \dots, (x_n, y_n)$ fixé, comment obtenir la meilleure coupure ?

Comme souvent ce sont les données qui vont répondre à cette question. La sélection de la meilleur coupure s'effectue en introduisant une **fonction d'impureté** $\mathcal{I}$ qui va mesurer le degrés d'hétérogénéité d'un nœud $\mathcal{N}$. Cette fonction prendra de

- grandes valeurs pour les nœuds hétérogènes (les valeurs de Y diffèrent à l'intérieur du nœud) ;

- faibles valeurs pour les nœuds homogènes (les valeurs de Y sont proches à l'intérieur du nœud).

On utilise souvent comme fonction d'impureté :

- la **variance** en régression

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

  où $\overline{y}_{\mathcal{N}}$ désigne la moyenne des y_i dans $\mathcal{N}$.

- l'impureté de **Gini** en classification binaire

$$\mathcal{I}(\mathcal{N}) = 2p(\mathcal{N})(1 - p(\mathcal{N}))$$

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

Les coupures considérées par l'algorithme CART sont des hyperplans orthogonaux aux axes de $\mathbb{R}^p$, choisir une coupure revient donc à choisir une variable j parmi les p variables explicatives et un seuil s dans $\mathbb{R}$. On peut donc représenter une coupure par un couple (j, s). Une fois l'impureté définie, on choisira la coupure (j, s) qui **maximise le gain d'impureté** entre le noeud père et ses deux noeuds fils :

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

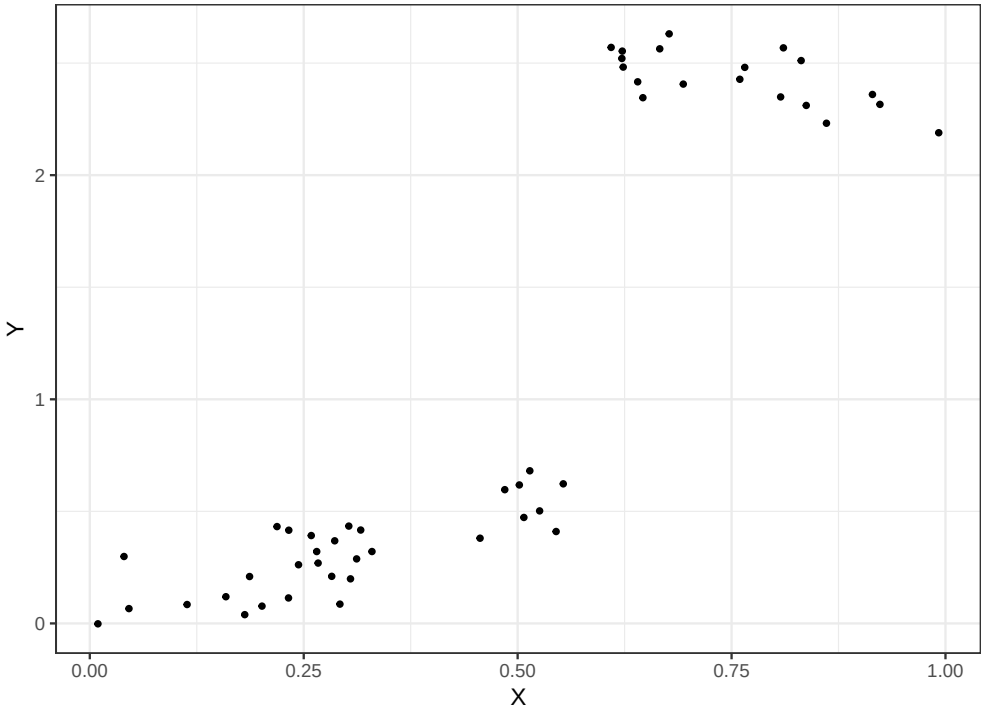
où

- $\mathcal{N}_1(j, s)$ et $\mathcal{N}_2(j, s)$ sont les 2 nœuds fils de $\mathcal{N}$ engendrés par la coupure (j, s) ;
- $\mathbf{P}(\mathcal{N})$ représente la proportion d'observations dans le nœud $\mathcal{N}$.

1.1.1 Arbres de régression

On considère le jeu de données suivant où le problème est d'expliquer la variable quantitative Y par la variable quantitative X.

```
n <- 50
set.seed(1234)
X <- runif(n)
set.seed(5678)
Y <- 1*X*(X<=0.6)+(-1*X+3.2)*(X>0.6)+rnorm(n,sd=0.1)
data1 <- data.frame(X,Y)
ggplot(data1)+aes(x=X,y=Y)+geom_point()
```



1. A l'aide de la fonction **rpart** du package **rpart**, construire un arbre permettant d'expliquer Y par X.

```
library(rpart)
```

2. Visualiser l'arbre à l'aide des fonctions **prp** et **rpart.plot** du package **rpart.plot**.

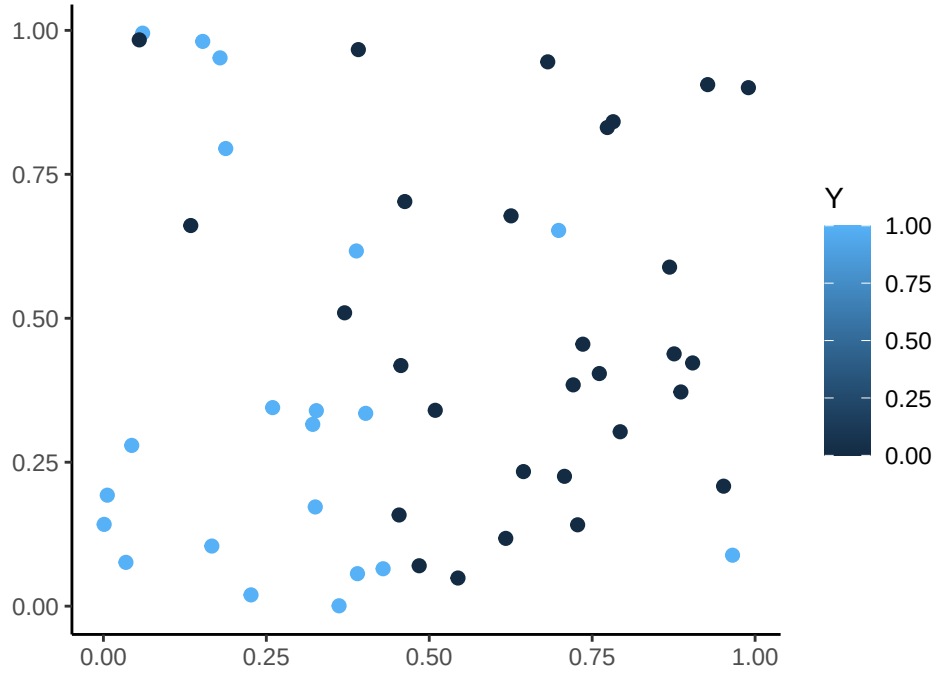
```
library(rpart.plot)
```

3. Écrire l'estimateur associé à l'arbre.
4. Ajouter sur le graphe de la question 1 la partition définie par l'arbre ainsi que les valeurs prédites.

1.1.2 Arbres de classification

On considère les données suivantes où le problème est d'expliquer la variable binaire Y par deux variables quantitatives X_1 et X_2.

```r
n <- 50
set.seed(12345)
X1 <- runif(n)
set.seed(5678)
X2 <- runif(n)
Y <- rep(0,n)
set.seed(54321)
Y[X1<=0.45] <- rbinom(sum(X1<=0.45),1,0.85)
set.seed(52432)
Y[X1>0.45] <- rbinom(sum(X1>0.45),1,0.15)
data2 <- data.frame(X1,X2,Y)
ggplot(data2)+aes(x=X1,y=X2,color=Y)+geom_point(size=2)+scale_x_continuous(name="")+
  scale_y_continuous(name="")+theme_classic()
```



1. Construire un arbre permettant d'expliquer Y par X_1 et X_2. Représenter l'arbre et identifier l'éventuel problème.
2. Écrire la règle de classification ainsi que la fonction de score définies par l'arbre.
3. Ajouter sur le graphe de la question 1 la partition définie par l'arbre.

1.1.3 Entrée qualitative

On considère les données

```
n <- 100
X <- factor(rep(c("A","B","C","D"),n))
set.seed(1234)
Y[X=="A"] <- rbinom(sum(X=="A"),1,0.9)
Y[X=="B"] <- rbinom(sum(X=="B"),1,0.25)
Y[X=="C"] <- rbinom(sum(X=="C"),1,0.8)
Y[X=="D"] <- rbinom(sum(X=="D"),1,0.2)
Y <- as.factor(Y)
data3 <- data.frame(X,Y)
```

1. Construire un arbre permettant d'expliquer Y par X.
2. Expliquer la manière dont l'arbre est construit dans ce cadre là.

1.2 Élagage

Le procédé de coupe présenté précédemment permet de définir un très grand nombre d'arbres à partir d'un jeu de données (arbre sans coupure, avec une coupure, deux coupures…). Se pose alors la question de trouver le **meilleur arbre** parmi tous les arbres possibles. Une première idée serait de choisir parmi tous les arbres possibles celui qui optimise un critère de performance. Cette approche, bien que cohérente, n'est généralement pas possible à mettre en œuvre en pratique car le nombre d'arbres à considérer est souvent trop important.

La méthode CART propose une procédure permettant de choisir automatiquement un arbre en 3 étapes :

- On construit un **arbre maximal** (très profond) $\mathcal{T}_{max}$;
- On sélectionne une **suite d'arbres emboités** :

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

  La sélection s'effectue en optimisant un critère **Cout/complexité** qui permet de réguler le compromis entre **ajustement** et **complexité** de l'arbre.
- On **sélectionne un arbre** dans cette sous-suite en optimisant un critère de performance.

Cette approche revient à choisir un sous-arbre de l'arbre $\mathcal{T}_{\max}$, c'est-à-dire à enlever des branches à $T_{\max}$, c'est pourquoi on parle **d'élagage**.

1.2.1 Élagage pour un problème de régression

On considère les données **Carseats** du package **ISLR**.

```
library(ISLR)
data(Carseats)
summary(Carseats)
```

```
     Sales          CompPrice       Income        Advertising
 Min.   : 0.000   Min.   : 77   Min.   : 21.00   Min.   : 0.000
 1st Qu.: 5.390   1st Qu.:115   1st Qu.: 42.75   1st Qu.: 0.000
 Median : 7.490   Median :125   Median : 69.00   Median : 5.000
 Mean   : 7.496   Mean   :125   Mean   : 68.66   Mean   : 6.635
 3rd Qu.: 9.320   3rd Qu.:135   3rd Qu.: 91.00   3rd Qu.:12.000
 Max.   :16.270   Max.   :175   Max.   :120.00   Max.   :29.000
   Population        Price        ShelveLoc        Age          Education
 Min.   : 10.0   Min.   : 24.0   Bad   : 96   Min.   :25.00   Min.   :10.0
 1st Qu.:139.0   1st Qu.:100.0   Good  : 85   1st Qu.:39.75   1st Qu.:12.0
 Median :272.0   Median :117.0   Medium:219   Median :54.50   Median :14.0
 Mean   :264.8   Mean   :115.8                Mean   :53.32   Mean   :13.9
 3rd Qu.:398.5   3rd Qu.:131.0                3rd Qu.:66.00   3rd Qu.:16.0
 Max.   :509.0   Max.   :191.0                Max.   :80.00   Max.   :18.0
 Urban       US
 No :118   No :142
 Yes:282   Yes:258
```

On cherche ici à expliquer la variable quantitative **Sales** par les autres variables.

1. Construire un arbre permettant de répondre au problème.
2. Expliquer les sorties de la fonction **printcp** appliquée à l'arbre de la question précédente et calculer le dernier terme de la colonne **rel error**.
3. Construire une suite d'arbres plus grandes en jouant sur les paramètres `cp` et `minsplit` de la fonction **rpart**.
4. Expliquer la sortie de la fonction **plotcp** appliquée à l'arbre de la question précédente.
5. Sélectionner le "meilleur" arbre dans la suite construite.

6. Visualiser l'arbre choisi (utiliser la fonction **prune**).
7. On souhaite prédire les valeurs de Y pour de nouveaux individus à partir de l'arbre sélectionné. Pour simplifier on considèrera ces 4 individus :

```
new_ind <- Carseats %>% slice(3,58,185,218) %>% dplyr::select(-Sales)
new_ind
```

```
    CompPrice Income Advertising Population Price ShelveLoc Age Education Urban
3         113     35          10        269    80    Medium  59        12   Yes
58         93     91           0         22   117       Bad  75        11   Yes
185       132     33           7         35    97    Medium  60        11    No
218       106     44           0        481   111    Medium  70        14    No
     US
3   Yes
58   No
185 Yes
218  No
```

   Calculer les valeurs prédites.
8. Séparer les données en un échantillon d'apprentissage de taille 250 et un échantillon test de taille 150.
9. On considère la suite d'arbres définie par

```
set.seed(4321)
tree <- rpart(Sales~.,data=train,cp=0.000001,minsplit=2)
```

   Dans cette suite, sélectionner

   - un arbre très simple (avec 2 ou 3 coupures)
   - un arbre très grand
   - l'arbre optimal (avec la procédure d'élagage classique).
10. Calculer l'erreur quadratique de ces 3 arbres en utilisant l'échantillon test.
11. Refaire la comparaison avec une validation croisée 10 blocs.

1.2.2 Élagage en classification binaire et matrice de coût

On considère ici les mêmes données que précédemment mais on cherche à expliquer une version binaire de la variable **Sales**. Cette nouvelle variable, appelée `High` prend pour valeurs `No` si `Sales` est inférieur ou égal à 8, `Yes` sinon. On travaillera donc avec le jeu `data1` défini ci-dessous.

```
High <- ifelse(Carseats$Sales<=8,"No","Yes")
data1 <- Carseats %>% dplyr::select(-Sales) %>% mutate(High)
```

1. Construire un arbre permettant d'expliquer `High` par les autres variables (sans `Sales` évidemment !) et expliquer les principales différences par rapport à la partie précédente précédente.
2. Expliquer l'option `parms` dans la commande :

   ```
   tree1 <- rpart(High~.,data=data1,parms=list(split="information"))
   tree1$parms
   ```

   ```
   $prior
      1    2
   0.59 0.41

   $loss
        [,1] [,2]
   [1,]    0    1
   [2,]    1    0

   $split
   [1] 2
   ```

3. Expliquer les sorties de la fonction **printcp** sur le premier arbre construit et retrouver la valeur du dernier terme de la colonne **rel error**.
4. Sélectionner un arbre optimal dans la suite.
5. On considère la suite d'arbres

   ```
   tree2 <- rpart(High~.,data=data1,parms=list(loss=matrix(c(0,5,1,0),ncol=2)),
              cp=0.01,minsplit=2)
   ```

   Expliquer les sorties des commandes suivantes. On pourra notamment calculer le dernier terme de la colonne **rel error** de la table **cptable**.

```
tree2$parms
```

```
$prior
   1    2
0.59 0.41

$loss
     [,1] [,2]
[1,]    0    1
[2,]    5    0

$split
[1] 1
```

```
printcp(tree2)
```

```
Classification tree:
rpart(formula = High ~ ., data = data1, parms = list(loss = matrix(c(0,
    5, 1, 0), ncol = 2)), cp = 0.01, minsplit = 2)

Variables actually used in tree construction:
[1] Advertising Age         CompPrice   Education   Income      Population
[7] Price       ShelveLoc

Root node error: 236/400 = 0.59

n= 400

         CP nsplit rel error xerror    xstd
1  0.101695      0   1.00000 5.0000 0.20840
2  0.050847      2   0.79661 3.8136 0.20909
3  0.036017      3   0.74576 3.2034 0.20176
4  0.035311      5   0.67373 3.1271 0.20038
5  0.025424      9   0.50847 2.6144 0.19069
6  0.016949     11   0.45763 2.3475 0.18307
7  0.015537     16   0.37288 2.1992 0.17905
8  0.014831     21   0.28814 2.1992 0.17905
9  0.010593     23   0.25847 2.0466 0.17367
10 0.010000     25   0.23729 2.0297 0.17292
```

6. Comparer les valeurs ajustées par les deux arbres considérés.

1.2.3 Calcul de la sous-suite d'arbres optimaux

Exercice 1.1 (Minimisation du critère coût/complexité)**.** On considère l'algorithme qui permet de calculer les suites $(\alpha_m)_m$ et $(T_{\alpha_m})_m$ du théorème présenté en cours. Pour simplifier on se place en classification binaire et on considère les notations suivantes (en plus de celles présentées dans le chapitre) :

- $R(t)$: erreur de classification dans le nœud t pondérée par la proportion d'individus dans le nœud (nombre d'individus dans t sue le nombre total d'individus).
- T^t : la branche de l'arbre T issue du nœud interne t.
- $R(T^t)$: l'erreur de la branche T^t pondérée par la proportion d'individus dans le nœud.

L'algorithme suivant présente le calcul explicite des suites $(\alpha_m)_m$ et $(T_{\alpha_m})_m$.

Initialisation : on pose $\alpha_0 = 0$ et on calcule l'arbre maximale T_0 qui minimise $C_0(T)$. On fixe $m = 0$. Répéter jusqu'à obtenir l'arbre racine

1. Calculer pour tous les nœuds t internes de T_{α_m}
$$g(t) = \frac{R(t) - R(T^t_{\alpha_m})}{|T^t_{\alpha_m}| - 1}$$
2. Choisir le nœud interne t_m qui minimise $g(t)$.
3. On pose
$$\alpha_{m+1} = g(t_m) \quad \text{et} \quad T_{\alpha_{m+1}} = T_{\alpha_m} - T^{t_m}_{\alpha_m}.$$
4. Mise à jour : $m := m + 1$.

Retourner : les suites finies $(\alpha_m)_m$ et $(T_{\alpha_m})_m$.

On propose d'utiliser cet algorithme sur l'arbre construit suivant

```
gen_class_bin2D <- function(n=100,graine=1234,bayes=0.1){
  set.seed(graine)
  grille <- 0.1
  X1 <- runif(n)
  X2 <- runif(n)
  Y <- rep(0,n)
  cond0 <- (X1>0.2 & X2>=0.8) | (X1>0.6 & X2<0.4) | (X1<0.25 & X2<0.5)
  cond1 <- !cond0
  Y[cond0] <- rbinom(sum(cond0),1,bayes)
  Y[cond1] <- rbinom(sum(cond1),1,1-bayes)
```

```
  donnees <- tibble(X1,X2,Y=as.factor(Y))
 # indapp <- 1:napp
#  dapp <- donnees[indapp,]
#  dtest <- donnees[-indapp,]

  px1 <- seq(0,1,by=grille)
  px2 <- seq(0,1,by=grille)
  px <- expand.grid(X1=px1,X2=px2)
  py <- rep(0,nrow(px))
  cond0 <- (px[,1]>0.2 & px[,2]>=0.8) | (px[,1]>0.6 & px[,2]<0.4) | (px[,1]<0.25 & px[,2]<
  cond1 <- !cond0
  py[cond0] <- 0
  py[cond1] <- 1
  df <- px %>% as_tibble() %>% mutate(Y=as.factor(py))
  p <- ggplot(df)+aes(x=X1,y=X2,fill=Y)+geom_raster(hjust=1,vjust=1)#+theme(legend.positio
  return(list(donnees=donnees,graphe=p))
}
don.2D.arbre <- gen_class_bin2D(n=150,graine=3210,bayes=0.05)$donnees
set.seed(123)
T0 <- rpart(Y~.,data=don.2D.arbre)
rpart.plot(T0,extra = 1)
```

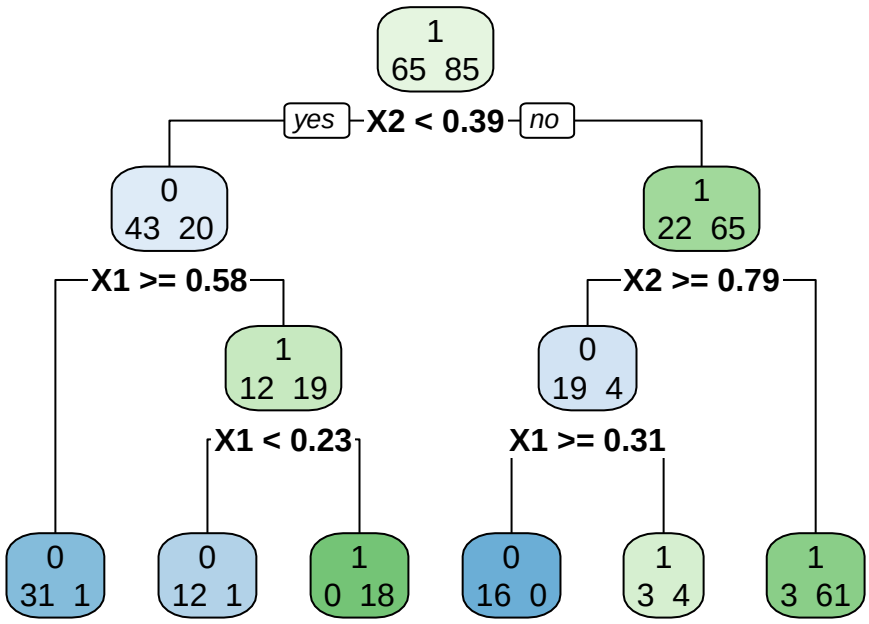


Figure 1.1: L'arbre T_0.

Cet arbre n'est pas l'arbre maximal mais la manière d'élaguer est identique.

1. Calculer pour les 5 nœuds internes de T_0 la fonction $g(t)$.
2. En déduire la valeur de α_1 ainsi que l'arbre T_{α_1}.
3. Retrouver cette valeur en utilisant la fonction `printcp` et représenter l'arbre T_1 en utilisant `prune`.
4. Faire le même travail pour calculer α_2 et T_{α_2}.

partie II

Agrégation

2 Forêts aléatoires

Les méthodes par arbres présentées précédemment sont des algorithmes qui possèdent tout un tas de qualités (facile à mettre en œuvre, interprétable...). Ce sont néanmoins rarement les algorithmes qui se révèlent les plus performants. Les méthodes d'agrégation d'arbres présentées dans cette partie sont souvent beaucoup plus pertinentes, notamment en terme de qualité de prédiction. Elles consistent à construire un très grand nombre d'arbres "simples" : $g_1, \dots, g_B$ et à les agréger en faisant la moyenne :

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

Les forêts aléatoires (Breiman 2001) et le gradient boosting (Friedman 2001) utilisent ce procédé d'agrégation. Dans ce chapitre on étudiera ces algorithmes sur le je de données `spam` :

```
library(kernlab)
data(spam)
set.seed(1234)
spam <- spam[sample(nrow(spam)),]
```

Le problème est d'expliquer la variable binaire **type** par les autres.

L'algorithme des forêts aléatoires consiste à construire des arbres sur des échantillons bootstrap et à les agréger. Il peut s'écrire de la façon suivante :

Entrées :

- $x \in \mathbb{R}^d$ l'observation à prévoir, $\mathcal{D}_n$ l'échantillon ;
- B nombre d'arbres ; n_{max} nombre max d'observations par nœud
- $m \in \{1, \dots, d\}$ le nombre de variables candidates pour découper un nœud.

Algorithme : pour $k = 1, \dots, B$:

1. Tirer un échantillon *bootstrap* dans $\mathcal{D}_n$
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 d. On note $T(., \theta_k, \mathcal{D}_n)$ l'arbre construit.

Sortie : l'estimateur $T_B(x) = \frac{1}{B}\sum_{k=1}^{B} T(x, \theta_k, \mathcal{D}_n)$.

Cet algorithme peut être utilisé sur **R** avec la fonction **randomForest** du package **randomForest** ou la fonction **ranger** du package **ranger**.

Exercice 2.1 (Biais et variance des algorithmes bagging)**.** Comparer le biais et la variance de la forêt $T_B(x)$ au biais et à la variance d'un arbre de la forêt $T(x, \theta_k, \mathcal{D}_n)$. On pourra utiliser $\rho(x) = \text{corr}(T(x, \theta_1, \mathcal{D}_n), T(x, \theta_2, \mathcal{D}_n))$ pour comparer les variances.

Exercice 2.2 (RandomForest versus ranger)**.** On sépare le jeu de données `spam` en un échantillon d'apprentissage et un échantillon test :

```
set.seed(1234)
library(tidymodels)
data_split <- initial_split(spam, prop = 3/4)
spam.app <- training(data_split)
spam.test  <- testing(data_split)
```

1. Entraîner une forêt aléatoire sur les données d'apprentissage uniquement en utilisant les paramètres par défaut de la fonction **randomForest**. Commenter.
2. Calculer les groupes prédits pour les individus de l'échantillon test et en déduire une estimation de l'erreur de classification.
3. Calculer les estimations de la probabilité de spam pour les individus de l'échantillon test.
4. Refaire les questions précédentes avec la fonction **ranger** du package **ranger** (voir https://arxiv.org/pdf/1508.04409.pdf).

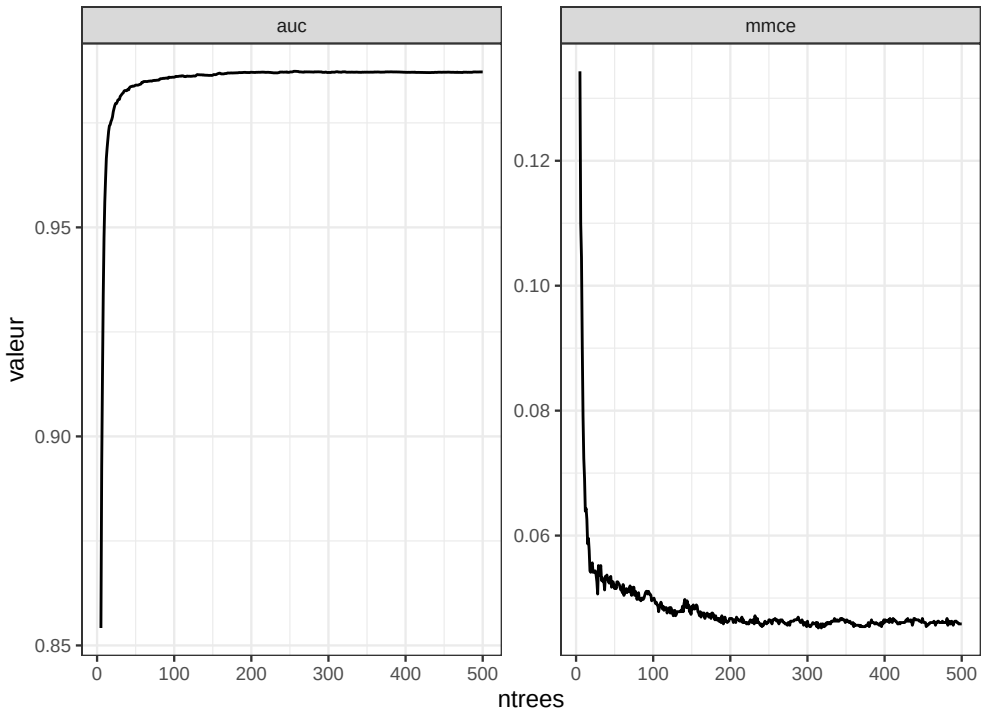
   ```
   library(ranger)
   ```

5. Comparer les temps de calcul de **randomForest** et **ranger**.

Exercice 2.3 (Sélection des paramètres)**.** Nous nous intéressons ici au choix des paramètres de la forêt aléatoire.

1. Expliquer la sortie suivante.

```
set.seed(12345)
library(OOBCurve)
foret1 <- ranger(type~.,data=spam,keep.inbag=TRUE)
spam.task <- mlr::makeClassifTask(data=spam,target="type")
erreurs <- OOBCurve(foret1,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")
```



2. Construire la forêt avec `mtry=1` et comparer ses performances avec celle construite précédemment.

3. A l'aide des outils `tidymodels` sélectionner les paramètres `mtry` et `min_n` dans les grilles `c(1,6,seq(10,50,by=10),57)` et `c(1,5,100,500)`. On pourra notamment visualiser les critères en fonction des valeurs de paramètres.

4. Visualiser l'importance des variables pour les scores d'impureté et de permutations.

Exercice 2.4 (Arbre vs forêt aléatoire)**.** Proposer et mettre en œuvre une procédure permettant de comparer les performances (courbes ROC, AUC et accuracy) d'un arbre CART utilisant la procédure d'élagage proposée dans la Section 1.2 avec une forêt aléatoire.

3 Gradient boosting

Les algorithmes de gradient boosting permettent de minimiser des pertes empiriques de la forme

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

où $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ est une fonction de coût convexe en son second argument. Il existe plusieurs type d'algorithmes boosting. Un des plus connus et utilisés a été proposé par Friedman (2001), c'est la version que nous étudions dans cette partie.

Cette approche propose de chercher la meilleure combinaison linéaire d'arbres binaires, c'est-à-dire que l'on recherche $g(x) = \sum_{m=1}^{M} \alpha_m h_m(x)$ qui minimise

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

Optimiser sur toutes les combinaisons d'arbres binaires se révélant souvent trop compliqué, Friedman (2001) utilise une descente de gradient pour construire la combinaison d'arbres de façon récursive. L'algorithme est le suivant :

Entrées :

- $d_n = (x_1, y_1), \dots, (x_n, y_n)$ l'échantillon, λ un paramètre de régularisation tel que $0 < \lambda \leq 1$.
- $M \in \mathbb{N}$ le nombre d'itérations.
- paramètres de l'arbre (nombre de coupures…)

Itérations :

1. Initialisation : $g_0(.) = \text{argmin}_c \frac{1}{n}\sum_{i=1}^{n}\ell(y_i, c)$
2. Pour $m = 1$ à M :
   a. Calculer l'opposé du gradient $-\frac{\partial}{\partial g(x_i)}\ell(y_i, g(x_i))$ et l'évaluer aux points $g_{m-1}(x_i)$:

$$U_i = -\frac{\partial}{\partial g(x_i)}\ell(y_i, g(x_i))\Big|_{g(x_i)=g_{m-1}(x_i)}, \quad i = 1, \dots, n.$$

   b. Ajuster un arbre sur l'échantillon $(x_1, U_1), \dots, (x_n, U_n)$, on le note h_m.

c. Mise à jour : $g_m(x) = g_{m-1}(x) + \lambda h_m(x)$.

Sortie : la suite $(g_m(x))_m$.

Sur **R** On peut utiliser différents packages pour faire du gradient boosting. Nous utilisons ici le package **gbm** (Ridgeway 2006).

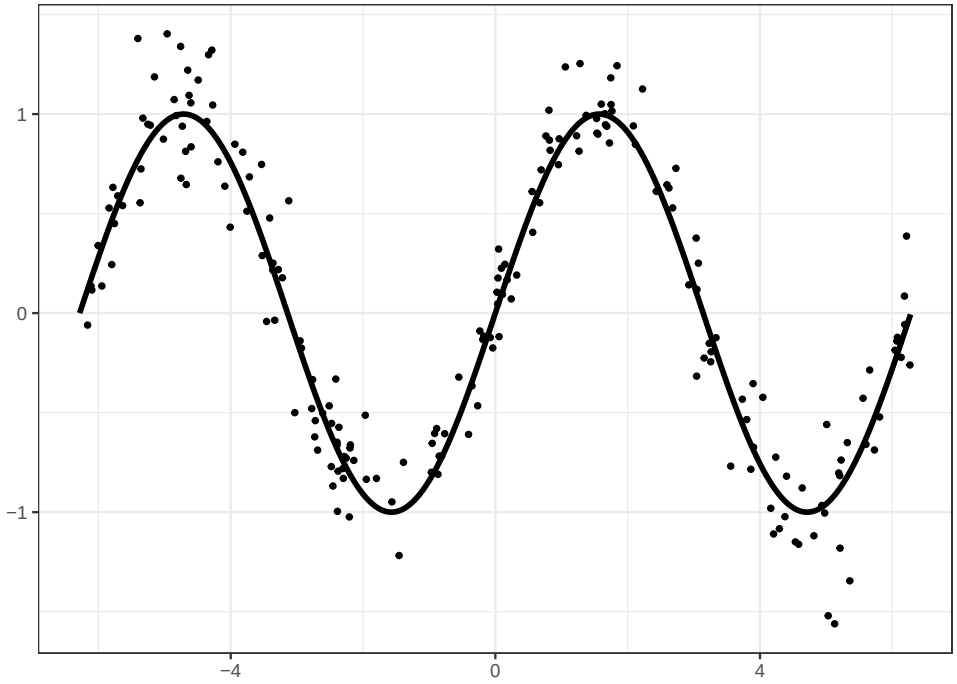
3.1 Un exemple simple en régression

On considère un jeu de données $(x_i, y_i), i = 1, \dots, 200$ issu d'un modèle de régression

$$y_i = m(x_i) + \varepsilon_i$$

où la vraie fonction de régression est la fonction **sinus** (mais on va faire comme si on ne le savait pas).

```
x <- seq(-2*pi,2*pi,by=0.01)
y <- sin(x)
set.seed(1234)
X <- runif(200,-2*pi,2*pi)
Y <- sin(X)+rnorm(200,sd=0.2)
df1 <- data.frame(X,Y)
df2 <- data.frame(X=x,Y=y)
p1 <- ggplot(df1)+aes(x=X,y=Y)+geom_point()+geom_line(data=df2,size=1)+xlab("")+ylab("")
p1
```



1. Rappeler ce que siginifie le L_2-boosting.
2. A l'aide de la fonction **gbm** du package **gbm** construire un algorithme de L_2-boosting. On utilisera 500000 itérations et gardera les autres valeurs par défaut de paramètres, à l'exception de `bag.fraction` qu'on prendra égal à 1.
3. Visualiser l'estimateur à la première itération. On pourra faire un **predict** avec l'option `n.trees` ou utiliser directement la fonction **plot.gbm** avec l'option `n.trees`.
4. Faire de même pour les itérations 1000 et 500000.
5. Sélectionner le nombre d'itérations par la procédure de votre choix.
6. Représenter l'estimateur sélectionné.

3.2 Adaboost et logitboost pour la classification binaire.

On considère le jeu de données **spam** du package **kernlab**.

```
library(kernlab)
data(spam)
set.seed(1234)
spam <- spam[sample(nrow(spam)),]
```

1. Exécuter la commande et commenter la sortie.

```
model_ada1 <- gbm(type~.,data=spam,distribution="adaboost",interaction.depth=2,
                  shrinkage=0.05,n.trees=500)
```

2. Proposer une correction permettant de faire fonctionner l'algorithme.
3. Expliciter le modèle ajusté par la commande précédente.
4. Effectuer un **summary** du modèle ajusté. Expliquer la sortie.
5. Utiliser la fonction **vip** du package **vip** pour retrouver ce sorties.
6. Sélectionner le nombre d'itérations pour l'algorithme adaboost en faisant de la validation croisée 5 blocs.
7. Faire la même procédure en changeant la valeur du paramètre **shrinkage**. Interpréter.
8. Expliquer la différence entre **adaboost** et **logitboost** et précisez comment on peut mettre en œuvre ce dernier algorithme.

3.3 Comparaison de méthodes

On reprend les données `spam` de l'exercice précédent et on les coupe en un échantillon d'apprentissage pour entraîner les algorithmes et un échantillon test pour les comparer :

```
set.seed(1234)
perm <- sample(1:nrow(spam))
app <- spam[perm[1:3000],]
test <- spam[-perm[1:3000],]
```

1. Sur les données d'apprentissage uniquement, entraîner
   - l'algorithme **adaboost** en sélectionnant le nombre d'itérations par validation croisée
   - l'algorithme **logitboost** en sélectionnant le nombre d'itérations par validation croisée
   - une **forêt aléatoire** avec les paramètres par défaut
2. Pour les 3 algorithmes, calculer, pour tous les individus de l'échantillon `test`, la probabilité que ce soit un spam. On pourra stocker toutes ces probabilités dans un même `tibble`.
3. Comparer les 3 algorithmes avec la courbe ROC, l'AUC et l'erreur de classification.

4. Comment aurait-on pu faire pour obtenir des résultats plus précis.

3.4 Xgboost

L'algorithme **xgboost** Chen et Guestrin (2016) va plusloin que le **gradient boosting** en minimisant une approximation à l'odre 2 de la fonction de perte et en ajoutant un terme de régularisation dans la fonction objectif. On cherche toujours des combinaisons d'arbres

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

est un arbre à T feuilles : $w \in \mathbb{R}^T$ et $q : \mathbb{R}^d \to \{1, 2, ..., T\}$. À l'étape b, on cherche l'arbre qui minimise la **fonction objectif** de la forme

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

où $\Omega(h_j)$ est un terme de **régularisation** qui va pénaliser h_j en fonction de son nombre de feuilles T et des valeurs prédites w. Un développement limité à l'ordre 2 permet d'approcher cette fonction par

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

avec

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

Pour les arbres, la fonction de régularisation a la forme suivante :

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

où γ et λ contrôlent le **poids** que l'on donne aux paramètres de l'arbre. On obtient au final l'algorithme suivant

1. Initialisation $f_0 = h_0$.
2. Pour $b = 1, \dots, B$
   a) Ajuster un arbre h_b à T feuilles qui minimise
   $$\sum_{i=1}^{n}[\ell_i^{(1)} h_b(x_i) + \frac{1}{2}\ell_i^{(2)} h_b^2(x_i)] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j.$$
   b) Mettre à jour
   $$f_b(x) = f_{b-1}(x) + h_b(x).$$
3. Sortie : la suite d'algorithmes $(f_b)_b$.

On pourra trouver plus de précisions ici : https://xgboost.readthedocs.io/en/stable/tutorials/index.html

Exercice 3.1 (Prise en main des principales fonction de xgboost)**.** On commence par charger le package

```
library(xgboost)
```

et on reprend les données sur le **sinus** de la Section 3.1 :

```
x <- seq(-2*pi,2*pi,by=0.01)
y <- sin(x)
set.seed(1234)
X <- runif(200,-2*pi,2*pi)
Y <- sin(X)+rnorm(200,sd=0.2)
df1 <- data.frame(X,Y)
df2 <- data.frame(X=x,Y=y)
```

La fonction `xgboost` requiert que les données possèdent la classe `xgb.DMatrix`, on peut l'obtenir avec

```
X_mat <- xgb.DMatrix(as.matrix(df1[,1]),label=df1$Y)
```

1. Expliquer la sortie

```
boost1 <- xgboost(data=X_mat,nrounds = 5,params=list(objective="reg:squarederror"))
```

```
[1] train-rmse:0.633250
[2] train-rmse:0.468674
[3] train-rmse:0.354599
[4] train-rmse:0.275842
[5] train-rmse:0.224803
```

```
boost1
```

```
##### xgb.Booster
raw: 16.4 Kb
call:
  xgb.train(params = params, data = dtrain, nrounds = nrounds,
    watchlist = watchlist, verbose = verbose, print_every_n = print_every_n,
    early_stopping_rounds = early_stopping_rounds, maximize = maximize,
    save_period = save_period, save_name = save_name, xgb_model = xgb_model,
    callbacks = callbacks)
params (as set within xgb.train):
  objective = "reg:squarederror", validate_parameters = "TRUE"
xgb.attributes:
  niter
callbacks:
  cb.print.evaluation(period = print_every_n)
  cb.evaluation.log()
niter: 5
nfeatures : 1
evaluation_log:
 iter train_rmse
    1  0.6332497
    2  0.4686737
    3  0.3545990
    4  0.2758424
    5  0.2248031
```

2. Faire la même chose avec 100 itération et un `learning rate` de 0.1.
3. On peut obtenir les valeurs prédites entre -2π et 2π pour 100 itérations avec

```
Xtest <- as.matrix(df2$X)
prev100 <- predict(boost2,newdata=Xtest,iterationrange = c(1,101))
```

Tracer les estimateurs **xgboost** pour 1 itération, 20 itérations et 100 itérations. Commenter.

4. Commenter la sortie

```
set.seed(123)
sel.xgb <- xgb.cv(data = X_mat,
                  nrounds = 100, objective = "reg:squarederror",
                  eval_metric = "rmse",
                  nfold=5,eta=0.1,
                  early_stopping_rounds=10,
                  verbose=FALSE)
sel.xgb$evaluation_log |> head()
```

```
   iter train_rmse_mean train_rmse_std test_rmse_mean test_rmse_std
1:    1       0.7894653    0.012973111      0.7906004    0.05513590
2:    2       0.7189852    0.011889426      0.7228767    0.05364182
3:    3       0.6556685    0.010877136      0.6607388    0.05284791
4:    4       0.5985454    0.010104324      0.6055111    0.05240736
5:    5       0.5471973    0.009379501      0.5567722    0.05117564
6:    6       0.5007942    0.008971210      0.5129458    0.04971156
```

```
(ite.opt.xgb <- sel.xgb$best_iteration)
```

```
[1] 27
```

```
sel.xgb$niter
```

```
[1] 37
```

5. Tracer les prévisions pour l'algorithme sélectionné.

Exercice 3.2 (Xgboost sur les données spam)**.** On reprend les données spam de la Section 3.2. Entraîner un algorithme **xgboost** avec la fonction de perte `binary:logistic` et en sélectionnant la nombre d'itérations par validation croisée en optimisant l'AUC. **Attention** cette fonction de perte requiert que la variable à expliquer prenne pour valeurs 0 ou 1 en classe `numeric`.

Exercice 3.3 (Sélection avec tidymodels)**.** Refaire l'exercice précédent avec la syntaxe `tidymodels`. On choisira notamment :

- la profondeur des arbres dans le vecteur

```
c(1,3,8,10)
```

- le nombre d'itérations entre 1 et 500 avec un `early_stopping` toujours égal à 10 et un learning rate à 0.05.

On pourra consulter la page https://www.tidymodels.org/find/parsnip/ pour trouver les noms de paramètre du `worfklow` et sur le tutoriel https://juliasilge.com/blog/shelter-animals/ pour la stratégie. Elle est ici de fixer le nombre d'itérations à 500 puisqu'on utilise le **early stopping** en séparant les données en 2. On initialisera donc le workflow avec

```
library(tidymodels)
tune_spec <-
  boost_tree(tree_depth=tune(),trees=500,learn_rate=0.05,
             stop_iter=10) |>
  set_mode("classification") |>
  set_engine("xgboost",validation=0.2)

xgb_wf <- workflow() |>
  add_model(tune_spec) |>
  add_formula(type ~ .)
```

Références

Breiman, L. 2001. « Random forests ». *Machine learning* 45: 5-32.

Breiman, L., J. Friedman, R. Olshen, et C. Stone. 1984. *Classification and regression trees*. Wadsworth & Brooks.

Chen, T., et C. Guestrin. 2016. « XGBoost: A Scalable Tree Boosting System ». In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785-94. KDD '16. New York, NY, USA: ACM. https://doi.org/10.1145/2939672.2939785.

Friedman, J. H. 2001. « Greedy Function Approximation: A Gradient Boosting Machine ». *Annals of Statistics* 29: 1189-1232.

Ridgeway, G. 2006. « Generalized boosted models: A guide to the gbm package ».