Arbres et Forêts

R. Paegelow - MF Beclin

DO4

Décembre 2024

Arbres

Pour aller plus loin : Méthodes ensemblistes

Introduction :

Les arbres sont des outils d'exploitation des données et d'aide à la décision qui permettent d'expliquer et de prédire une variable quantitative (arbre de régression) ou qualitative (arbre de classification) à partir de variables explicatives quantitatives et/ou qualitatives.

En théorie des graphes, un arbre est un graphe non orienté, acyclique et connexe. L'ensemble des nœuds se divise en trois catégories :

- Nœud racine (l'accès à l'arbre se fait par ce nœud),
- Nœuds internes : les nœuds qui ont des descendants (ou enfants), qui sont à leur tour des nœuds,
- Nœuds terminaux (ou feuilles) : nœuds qui n'ont pas de descendant.

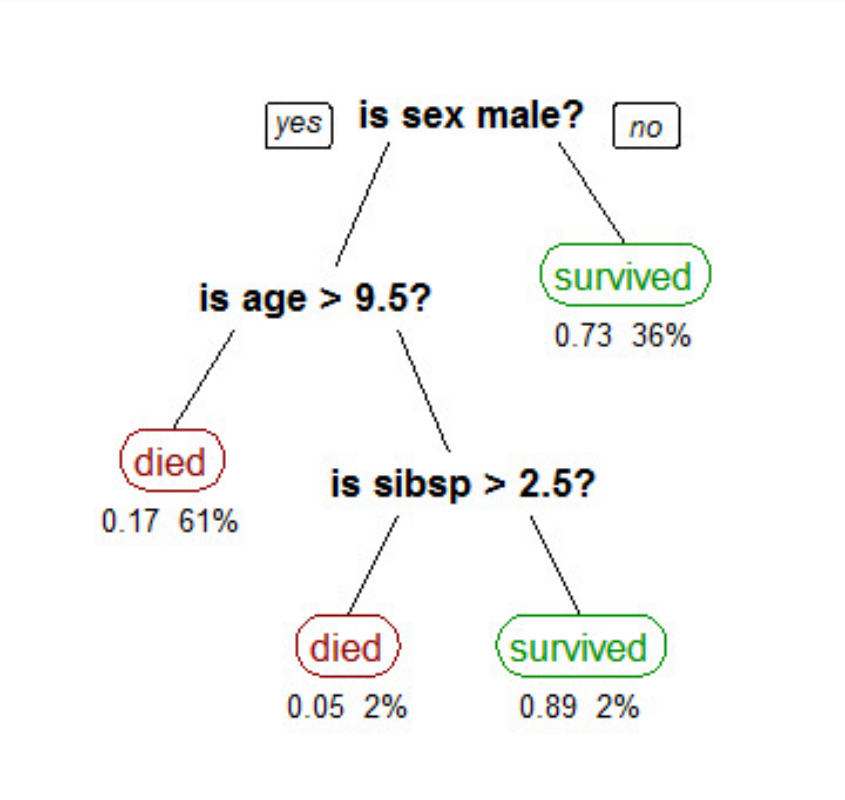


Figure 1: Classement sur la base de données des survivants du Titanic. (sibsp :

Le but des arbres est d'apprendre un arbre pour estimer une classe (en classification) ou une valeur (en régression).

En classification binaire par exemple, on peut se demander quelle variable discrimine le mieux les deux classes et quel est le meilleur seuil. A chaque nouveau noeud qu'on souhaite créer, il nous faut un critère qui permettra de quantifier le caractère discriminant du noeud, afin de choisir le meilleur.

La construction d'un arbre de discrimination binaire consiste à déterminer une séquence de nœuds.

Pour construire un noeud, il faut :

- Le choix conjoint d'une variable parmi les explicatives et d'une division qui induit une partition en deux classes
- Une division est elle-même définie par une valeur seuil de la variable quantitative sélectionnée ou un partage en deux groupes des modalités si la variable est qualitative.

À la racine correspond l'ensemble de l'échantillon ; la procédure est ensuite itérée sur chacun des sous-ensembles. L'algorithme considéré nécessite :

1. un critère permettant de sélectionner la meilleure division parmi toutes celles admissibles pour les différentes variables ;
2. une règle permettant de décider qu'un nœud est terminal : il devient ainsi une feuille ;
3. l'affectation de chaque feuille à l'une des classes ou à une valeur de la variable à expliquer.

Algorithme :

CART : Classification And Regression Tree (*Breiman et al. 1984*) : la méthode de référence.

$\Rightarrow$ permet d'obtenir des classes d'individus construites à partir des variables explicatives de manière à ce que les individus d'une même classe soient le plus homogène possible du point de vue de la variable d'intérêt.

Avantages :

- La simplicité et la facilité d'interprétation des résultats grâce à la représentation sous forme d'arbre.
- Aide à comprendre le phénomène par un étiquetage des nœuds en produisant des règles de décision.
- Modèle non paramétrique, permet de décrire des situations non-linéaires.

Inconveniant :

- Si la taille maximale de l'arbre autorisée est trop grande, il y a des risques de surajustement. L'arbre est trop adapté aux données et ne pourra pas prédire correctement une nouvelle data.

Packages :

Deux packages, rpart et tree, proposent la construction d'arbres basés sur CART.

Exemple :

Le même exemple de marketing bancaire.

L'objectif reste le même : détecter les clients intéressés par le produit proposé en fonction d'informations sur le client (âge, emploi,...). La variable à expliquer est qualitative

$\Rightarrow$ construire un arbre de classification.

Les étapes :

1) Importer les données
2) Construire et analyser l'arbre de classification
3) Choisir la taille de l'arbre
4) Prévision
5) Évaluer la performance de l'arbre

Traitement de données :

1) Importer les données :

On importe et on résume les données du fichier *bank-additional.csv*.
Il y a 451 clients ont souscrit au produit parmi 4119 clients

2) Construire et analyser l'arbre de classification :

On sépare tout d'abord aléatoirement la base de données en :

- Un échantillon d'apprentissage de taille 3000 qui sera utilisé pour construire les arbres.
- Un échantillon test de taille 1119 qui sera utilisé pour mesurer la performance des arbres.

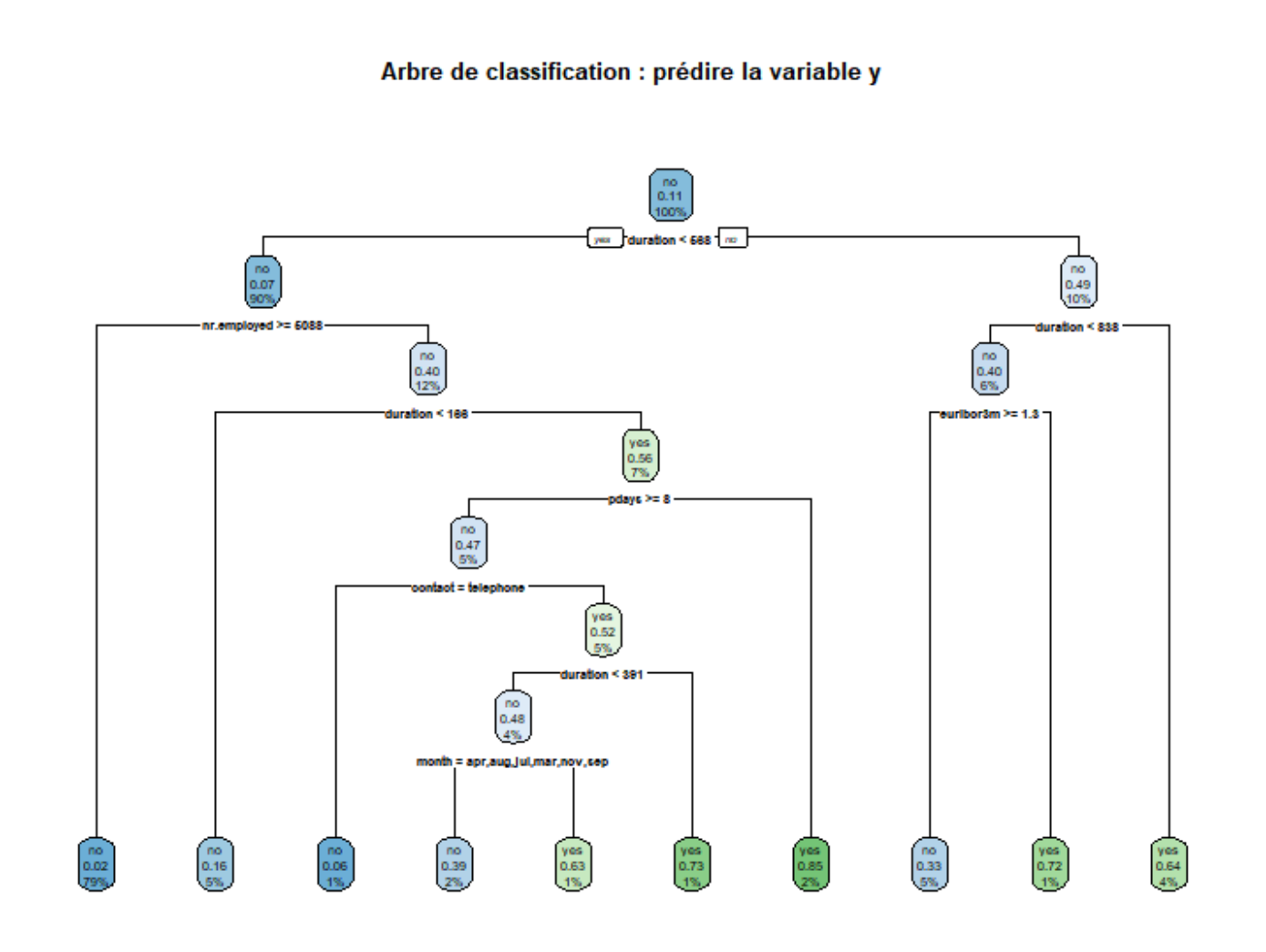
On implique la fonction rpart(comme lm).
Le paramètre cp permet de contrôler la profondeur de l'arbre : plus ce paramètre est petit, plus l'arbre est profond. Nous abordons le problème de choix de ce paramètre dans la partie 3.

```
> arbre1 <- rpart(y~., data= app, cp=0.02)
> print(arbre1)
n= 3000

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 3000 327 no (0.89100000 0.10900000)
   2) nr.employed>=5087.65 2620 157 no (0.94007634 0.05992366)
     4) duration< 616.5 2403  57 no (0.97627965 0.02372035) *
     5) duration>=616.5 217 100 no (0.53917051 0.46082949)
      10) duration< 837.5 121  41 no (0.66115702 0.33884298) *
      11) duration>=837.5 96  37 yes (0.38541667 0.61458333) *
   3) nr.employed< 5087.65 380 170 no (0.55263158 0.44736842)
     6) duration< 165.5 135  23 no (0.82962963 0.17037037) *
     7) duration>=165.5 245  98 yes (0.40000000 0.60000000)
      14) pdays>=15.5 175  85 no (0.51428571 0.48571429)
        28) duration< 401 128  50 no (0.60937500 0.39062500) *
        29) duration>=401 47  12 yes (0.25531915 0.74468085) *
      15) pdays< 15.5 70   8 yes (0.11428571 0.88571429) *
```

R0.6 Chaque nœud de l'arbre correspond un groupe d'individus :

- Le numéro du nœud : node , exp : 4)
- La règle de coupure : split, exp :**duration**$<$**616.5**
- Le nombre d'individus dans le groupe : n, exp : **2403**
- Le nombre d'individus mal classés : loss, exp :**57**
- La valeur prédite :yval, exp : **no**
- La probabilité d'appartenance à chaque classe : yprob, exp : **0.97627965 0.02372035**

La sortie print n'est pas facile à analyser. Il est plus simple de visualiser l'arbre pour comprendre sa construction.



La fonction rpart.plot du package **rpart.plot** permet d'obtenir la présentation graphique de cette segmentation $\Rightarrow$ résume l'information contenue dans print.

arbre2 $<-$ rpart(y$\sim$. , data $=$ app, maxcompete $=2$, maxsurrogate $=1$)
summary(arbre2)

Les arbres permettent aussi de traiter les valeurs manquantes. Pour ce faire, l'arbre propose pour chaque variable sélectionnée (à chaque nœud), une variable suppléante (surrogate) qui classe les individus quasiment de la même façon.

▷ L'argument maxsurrogate permet de préciser le nombre de variables suppliantes.

▷ Au niveau du 1^{er} nœud, les deux variables compétitrices sont **duration** et **nr.employed** et la variable suppléante est **euribor3m** : si on souhaite classer un nouvel individu pour lequel on n'a pas observé la variable **nr.employed**, on pourra ainsi utiliser la variable suppléante **euribor3m** pour découper le premier nœud.

3)Choisir la taille de l'arbre :

Produire des groupes homogènes $\rightarrow$ choisir l'arbre ayant le maximum de feuilles pures.
Cependant un arbre très profond risque d'avoir une mauvaise capacité de généralisation à de nouveaux individus : on parle de surapprentissage.
$\Rightarrow$ Il est crucial de choisir la taille de l'arbre.

- La fonction rpart permet de construire une suite d'arbres emboîtés qui optimise un critère prenant en compte à la fois l'ajustement et la complexité de l'arbre.
- La fonction printcp permet d'obtenir des informations sur les arbres.

▶ 6 arbres emboîtés.

▶ La colonne **CP** : un paramètre associé à la complexité de l'arbre : plus ce paramètre est petit, plus l'arbre est profond.

▶ La colonne **nsplit** : le nombre de coupures de l'arbre.

▶ **rel.error** : l'erreur de classification calculée sur les données d'apprentissage.

▶ La colonne **xerror** renvoie l'erreur de classification estimée par validation croisé sur 10 blocs. Ces erreurs sont normalisées par rapport à l'erreur de l'arbre racine qui vaut ici 0.11133.

▶ La dernière colonne : l'estimateur de l'ecart-type de l'erreur estimée dans la colonne **xerror**.

La sélection de l'arbre final s'effectue généralement en choisissant l'arbre dont l'erreur de classification est minimale, on choisit donc l'arbre qui minimise la colonne **xerror** (la ligne 6).

▶ L'erreur **xerror** ne cesse de décroitre lorsque la profondeur augmente. Il est donc possible qu'un arbre plus profond puisse posséder une erreur plus petite et donc plus performant.

⇒ Un arbre maximal plus profond : diminuer les valeurs des paramètres **cp** et **minsplit** de la fonction rpart.

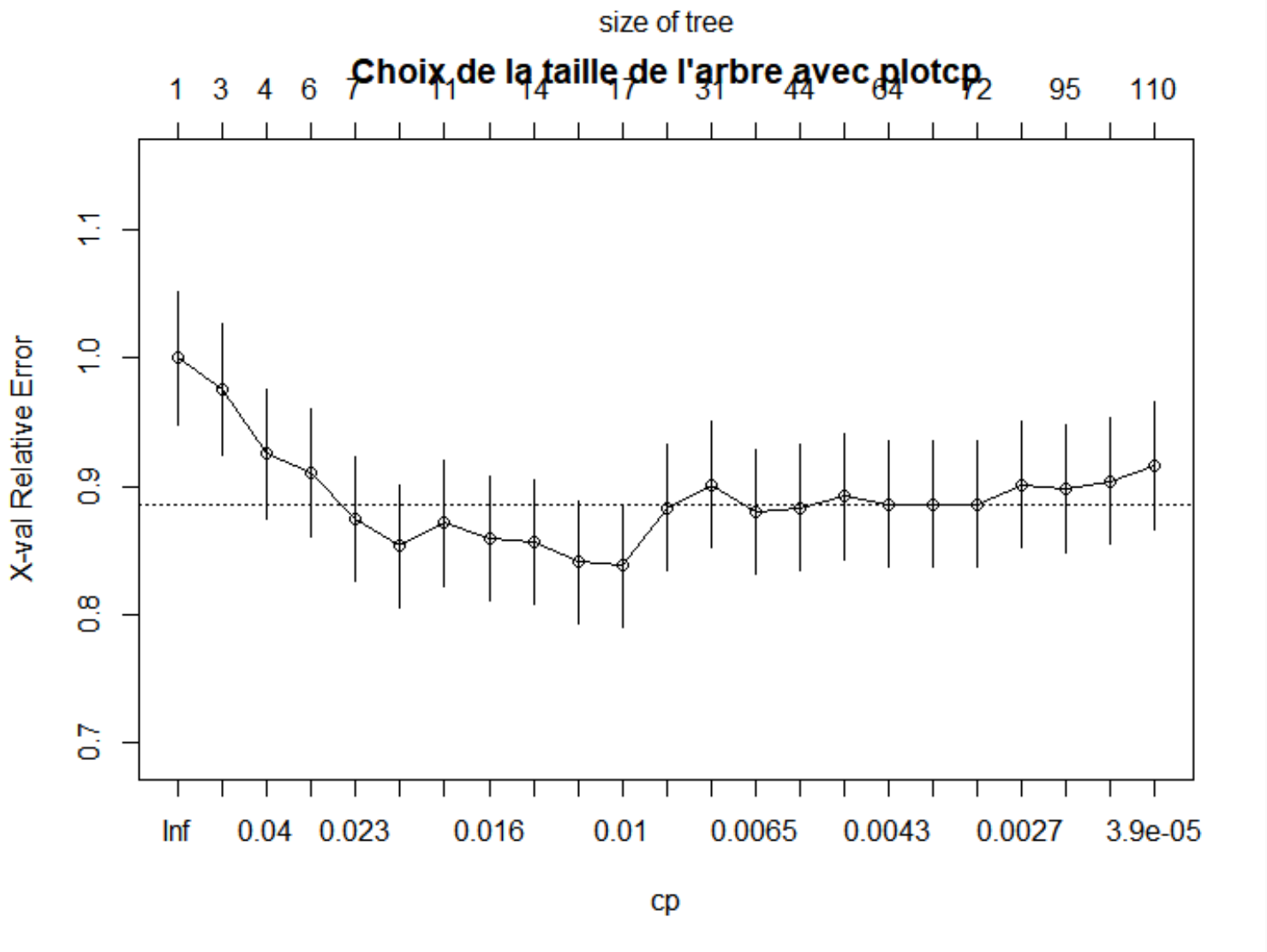
```
> #selection de l'arbre
> set.seed(34)
> arbre3 <- rpart(formula = y~. , data = app, cp= 0.000001, minsplit= 5)
> printcp(arbre3)
```

```
Classification tree:
rpart(formula = y ~ ., data = app, cp = 1e-06, minsplit = 5)

Variables actually used in tree construction:
 [1] age            campaign       cons.conf.idx  cons.price.idx contact        day_of_week    default
 [8] duration       education      emp.var.rate   euribor3m      housing        job            loan
[15] marital        month          nr.employed    pdays          previous

Root node error: 334/3000 = 0.11133

n= 3000

          CP nsplit rel error  xerror     xstd
1  0.0479042      0   1.00000 1.00000 0.051582
2  0.0419162      2   0.90419 0.97605 0.051037
3  0.0389222      3   0.86228 0.92515 0.049846
4  0.0239521      5   0.78443 0.91018 0.049487
5  0.0229541      6   0.76048 0.87425 0.048608
6  0.0179641      9   0.69162 0.85329 0.048084
7  0.0164671     10   0.67365 0.87126 0.048534
8  0.0149701     12   0.64072 0.85928 0.048235
9  0.0134731     13   0.62575 0.85629 0.048159
10 0.0119760     15   0.59880 0.84132 0.047780
11 0.0089820     16   0.58683 0.83832 0.047704
12 0.0074850     26   0.49701 0.88323 0.048830
13 0.0069860     30   0.46707 0.90120 0.049269
14 0.0059880     33   0.44611 0.88024 0.048756
15 0.0049900     43   0.38623 0.88323 0.048830
16 0.0044910     50   0.35030 0.89222 0.049051
17 0.0041916     63   0.29042 0.88623 0.048904
18 0.0039920     68   0.26946 0.88623 0.048904
19 0.0029940     71   0.25749 0.88623 0.048904
20 0.0023952     85   0.21557 0.90120 0.049269
21 0.0019960     94   0.18862 0.89820 0.049197
22 0.0014970    105   0.16168 0.90419 0.049342
23 0.0000010    109   0.15569 0.91617 0.049631
```

L'arbre maximal est ici beaucoup plus profond (109 coupures).

★ L'arbre optimal se trouve à la ligne 11.

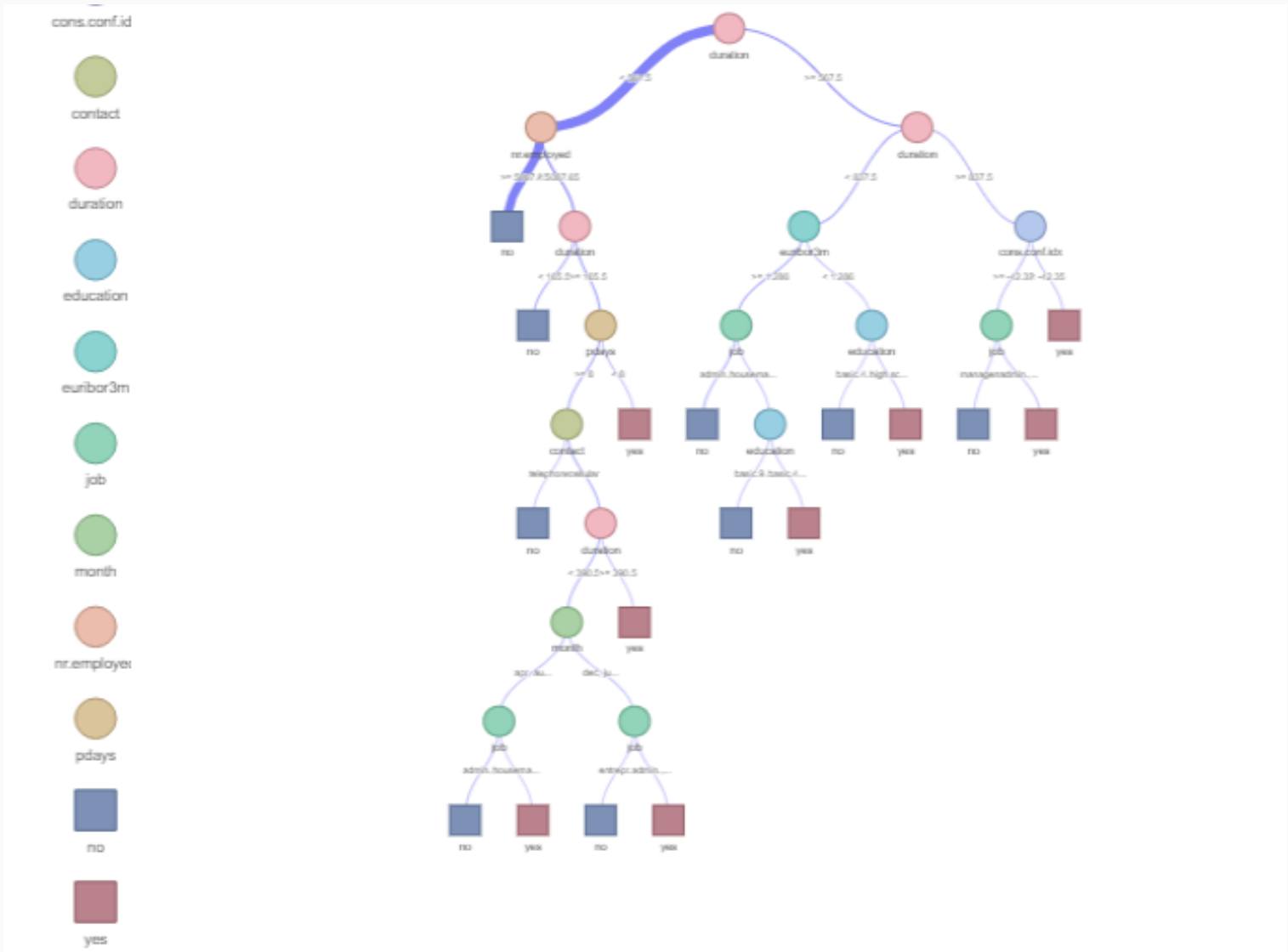
On peut visualiser graphiquement les erreurs des 23 arbres de la suite à l'aide de la commande **plotcp**.



Pour construire l'arbre final, on sélectionne dans **l'arbre3** le sous-arbre qui correspond à la valeur **cp= 0.0089820** à l'aide de la fonction prune.

```
#sous-arbre (cp optimal)
library(tidyverse)
cp_opt <- arbre3$cptable %>% as.data.frame() %>% filter(xerror == min(xerror)) %>% select(CP) %>% max()
            %>% as.numeric()
arbre.fin <- prune(arbre3, cp =cp_opt)
rpart.plot(arbre.fin)

#visualisation
library(visNetwork)
visTree(arbre.fin, main = "Arbre final obtenu par validation croisée")
```

La fonction visTree du package **visNetwork** propose une visualisation dynamique de l'arbre.


cons.conf.id
contact
duration
education
euribor3m
job
month
nr.employed
pdays
no
yes
duration
duration
no
duration
no
pdays
job
education
job
yes
contact
no
yes
no
education
no
yes
no
yes
no
duration
no
yes
month
job
job
no
yes
no
yes

Prévision :

Les arbres construits précédemment peuvent être utilisés dans un contexte de prévision.

La fonction predict.rpart (on peut utiliser simplement predict) permet d'obtenir les probabilités $\mathbb{P}(Y = yes \mid X = x)$ et $\mathbb{P}(Y = no \mid X = x)$ prédites par l'arbre **arbre.fin** pour les individus de l'échantillon test.

On obtient les classes prédites en ajoutant l'argument **type = "classe"**.

```
##4) Prévision :
predict(arbre.fin, newdata = test) %>% head(n=3)

predict(arbre.fin, newdata = test, type = "class") %>% head(n=3)
```

5) Évaluer la performance de l'arbre :

▶ On souhaite évaluer les performances des arbres **arbre3** et **arbre.fin** en estimant les taux de mal classés et les courbes ROC de ces deux modèles sur l'échantillon test.

▶ On récupère d'abord les probabilités estimées d'être **yes** par les deux modèles pour les individus de l'échantillon test.

▶ On obtient les erreurs de classification estimées des deux arbres en confrontant les valeurs prédites aux valeurs observées.

▶ L'arbre final possède une erreur plus petite.

▶ L'arbre final est également meilleur au sens de la courbe ROC. Cela se confirme en calculant les AUC (L'aire sous la courbe ROC (ou Area Under the Curve, AUC) peut être interprétée comme la probabilité)

```r
##5) Évaluer la performance de l'arbre :
prev.class <- data.frame(large = predict(arbre3, newdata = test, type = "class"),
                         fin= predict(arbre.fin, newdata = test, type = "class"), obs= test$y )
head(prev.class ,n=3)

#erreur de classification estimés des 2 arbres
prev.class %>% summarise_all(funs(err=mean(obs!=.))) %>% select(-obs_err) %>% round(3)

# courbes ROC
score <- data.frame(large=predict(arbre3,newdata= test)[,2], fin=predict(arbre.fin,newdata= test)[,2],obs=test$y)
library(plotROC)
df.roc <- score %>% gather(key = Methode,value=score, large,fin)
ggplot(df.roc)+aes(d= obs,m=score,color=Methode)+geom_roc()+theme_classic()


# verifier AUC
library(pROC)
df.roc %>% group_by(Methode) %>% summarize(AUC=pROC::auc(obs,score))
```

Pour aller plus loin : Méthodes ensemblistes

On va chercher à agréger un grand nombre de modèles tout en évitant le su rajustement (overfitting) :

Deux grandes stratégies : le bagging et le boosting.

Définition 2.1 : Bootstrap

Le bootstrap est une méthode de rééchantillonnage. Le but est de créer une nouvelle échantillon uniquement à partir du jeu de données. On peut par exemple réaliser un rééchantillonnage avec remise : On génère un grand nombre d'échantillons de taille en tirant aléatoirement avec remise dans l'échantillon initial. Chaque échantillon bootstrap peut contenir plusieurs fois certaines observations et en exclure d'autres.

Bagging

- $\boldsymbol{z} = (x_1, y_1)...(x_n, y_n)$ échantillon d'apprentissage, x_i décrit par p variables explicatives.
- On a B échantillons bootstrap $z_1, ...z_B$ d'individus. Ces échantillons sont issus de $\boldsymbol{z}$ par tirage aléatoire avec remise.
- Sur chacun de ces échantillons on apprend un modèle d'arbre (CART par exemple) qu'on appelle ϕ_i L'agrégation de ces B modèles se fait différemment en régression et en classification. En régression, on moyenne :

$$\hat{\phi}(x) = \frac{1}{B} \sum_{i=1}^{B} \phi_i(x)$$

En classification, on procède par vote.

Random Forest

Pour $b = 1 \dots B$

- Tirer un échantillon aléatoire z_b avec remise parmi $\boldsymbol{z}$
- Estimer un arbre sur z_b avec randomisation des variables :
  - Pour la construction de chaque noeud i de chaque arbre, on tire uniformement q_i variables parmi p pour former la décision associée au noeud. (Le but est de rendre les arbres plus indépendant entre eux)
- Puis on agrège les arbres comme pour le Bagging

L'intéret est de rendre chaque arbre non corrélé avec les autres.

Pour aller plus loin : le boosting Le boosting est une technique d'apprentissage automatique utilisée pour améliorer la performance des modèles en combinant plusieurs modèles faibles pour en créer un plus performant.

L'idée est d'entraîner plusieurs modèles successivement, en mettant plus de poids sur les erreurs du modèle précédent afin de corriger ses faiblesses.

Le boosting est largement utilisé en machine learning pour les tâches de classification et de régression, car il permet d'obtenir des modèles précis et robustes.

Comment fonctionne le Boosting ?

1. Initialisation
   - On entraîne un premier modèle faible / un arbre de décision peu profond.
   - On évalue les erreurs sur l'ensemble d'entraînement.
2. Pondération
   - Les échantillons mal classés ou avec de grandes erreurs sont pondérés davantage
   - On évalue les erreurs sur l'ensemble d'entraînement.
3. On répète le processus plusieurs fois, chaque nouvel arbre apprend en essayant de corriger les erreurs de l'arbre précédent.
4. La décision finale est obtenue en combinant tous les modèles faibles selon une règle (moyenne pondérée, somme, votes, etc.).

Quelques algorithmes populaires de boosting :

- AdaBoost (Adaptive Boosting) (AdaBoost python,AdaBoost R)
- Gradient Boosting : (Gradient boosting scikit-learn)
- XGBoost :Extreme Gradient Boosting (XGBoost R ; XGBoost python)

Avantages

- Excellente performance sur des tâches complexes.
- Fonctionne bien avec des jeux de données déséquilibrés.

Inconvénients

- Plus lent que d'autres méthodes
- Attention aux sur-aprentissages

Conclusion : Parmi les techniques que nous venons d'étudier : Arbres, Bagging d'arbre, Random Forest et Gradient Boosting ; CART et random Forest sont les plus populaires. CART offre une interprétation et Random Forest prévient le surajustement en ajoutant de l'aléa par l'échantillonnage du bootstrap et du nombre de variables considérées à chaque noeuds.

Bibliographie :

Breiman, L., Friedman, J. H., Olshen, R. A., Stone, C. J. (1984). Classification and Regression Trees. CRC Press.

https ://scikit-learn.org/stable/modules/tree.html

https ://www.math.univ-toulouse.fr/ gadat/Ens/M2SID/11-m2-Random$_F orests.pdf$