Tutoriel machine learning

Laurent Rouvière

31 mai 2021

Table des matières

Le tutoriel utilise le packages suivants :

```
library(tidyverse)
library(tidymodels)
library(glmnet)
library(kernlab)
library(kknn)
library(doParallel)
library(rpart)
library(rpart.plot)
library(ranger)
library(gbm)
```

On propose dans ce tutoriel de comparer quelques algorithmes d'apprentissage supervisé sur le jeu de données `spam`

```
data(spam)
dim(spam)
## [1] 4601   58
```

qui contient 4601 individus et 58 variables. Le problème est d'expliquer la variable binaire `type` par les 57 autres variables :

```
summary(spam$type)
## nonspam    spam
##    2788    1813
```

Risque et calibration avec tidymodels

On s'intéresse dans cette section à l'algorithme des k plus proches voisins. Une manière simple d'évaluer la performance de cet algorithme est de séparer l'échantillon en un échantillon d'apprentissage et un échantillon test :

```
set.seed(123)
spam_split <- initial_split(spam,prop=2/3)
dapp <- training(spam_split)
dtest <- testing(spam_split)
```

On entraîne ensuite l'algorithme des k ppv sur la partie apprentissage et on prédit l'échantillon test :

```
knn20 <- kknn(type~.,train=dapp,test=dtest,k=25,
              kernel="rectangular")$fitted.values
head(knn20)
## [1] spam    spam    spam    spam    nonspam spam
## Levels: nonspam spam
```

Un **risque** peut ensuite être estimé en confrontant les valeurs observées aux valeurs prédites. Pour l'**erreur de classification** on a par exemple

```
mean(knn20!=dtest$type)
## [1] 0.09197652
```

que l'on peut retrouver avec

```
tibble(prev=knn20,obs=dtest$type) %>% accuracy(truth=obs,estimate=prev)
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.908
```

Cette procédure est la plus simple. On peut obtenir des estimations plus précises en répétant les **ré-échantillonnages** : validation croisée, bootstrap... Le package `tidymodels` propose une syntaxe générale qui permet d'estimer le risque d'un grand nombre d'algorithmes. À titre d'exemple, nous proposons de calculer l'erreur de classification par validation croisée 10 blocs pour une grille de paramètres de k :

1. Définition de l'algorithme et de ses paramètres
   ```
   tune_spec <-
     nearest_neighbor(neighbors=tune(),weight_func="rectangular") %>%
     set_mode("classification") %>%
     set_engine("kknn")
   ```
2. Création du **workflow**
   ```
   ppv_wf <- workflow() %>%
     add_model(tune_spec) %>%
     add_formula(type ~ .)
   ```
3. Choix de la méthode de ré-échantillonnage
   ```
   set.seed(123)
   re_ech_cv <- vfold_cv(spam,v=10)
   ```
4. Choix de la grille
   ```
   grille_k <- tibble(neighbors=c(1,5,11,101,1001))
   ```
5. Calcul du risque avec `tune_grid`

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

On visualise les résultats avec :

```
ppv.cv %>% collect_metrics()
## # A tibble: 5 x 7
##   neighbors .metric  .estimator  mean     n std_err .config
##       <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1         1 accuracy binary     0.918    10 0.00515 Preprocessor1_Model1
## 2         5 accuracy binary     0.907    10 0.00389 Preprocessor1_Model2
## 3        11 accuracy binary     0.907    10 0.00450 Preprocessor1_Model3
## 4       101 accuracy binary     0.876    10 0.00395 Preprocessor1_Model4
## 5      1001 accuracy binary     0.744    10 0.00732 Preprocessor1_Model5
```

On peut ensuite choisir la meilleure valeur de `k` :

```
(best_k <- ppv.cv %>% select_best())
## # A tibble: 1 x 2
##   neighbors .config
##       <dbl> <chr>
## 1         1 Preprocessor1_Model1
```

et ré-entraîner l'algorithme sur toutes les données :

```
final_ppv <-
  ppv_wf %>%
  finalize_workflow(best_k) %>%
  fit(data = spam)
```

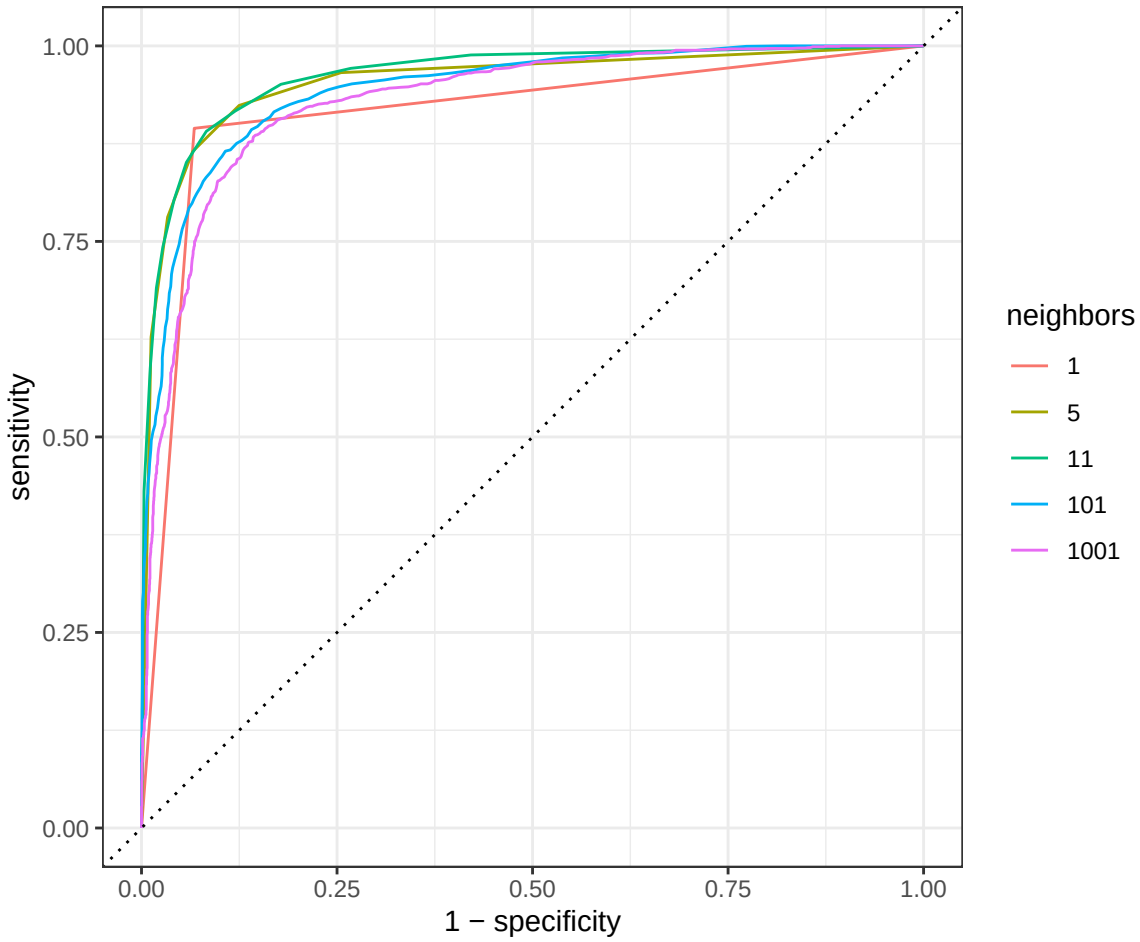
Exercice 1

1. Refaire le même travail en utilisant comme critère l'**aire sous la courbe ROC**. **Indication** : utiliser `control=control_resamples(save_pred = TRUE)` dans `tune_grid`.

```
auc.cv <- ppv_wf %>%
  tune_grid(resamples = re_ech_cv,
        grid = grille_k,
        control=control_resamples(save_pred = TRUE),
        metrics=metric_set(roc_auc))
auc.cv %>% collect_metrics()
## # A tibble: 5 x 7
##   neighbors .metric .estimator  mean     n std_err .config
##       <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1         1 roc_auc binary     0.914    10 0.00525 Preprocessor1_Model1
## 2         5 roc_auc binary     0.954    10 0.00347 Preprocessor1_Model2
## 3        11 roc_auc binary     0.962    10 0.00271 Preprocessor1_Model3
## 4       101 roc_auc binary     0.944    10 0.00299 Preprocessor1_Model4
## 5      1001 roc_auc binary     0.931    10 0.00309 Preprocessor1_Model5
```

2. Tracer les courbes **ROC** pour chaque valeur de `k`.

```
score <- collect_predictions(auc.cv)
score %>% group_by(neighbors) %>%
  roc_curve(type, .pred_spam,event_level="second") %>%
  autoplot()
```



Quelques algorithmes

On propose dans cette section de comparer les algorithmes :

- ridge
- lasso
- svm avec noyau gaussien
- arbres
- forêts aléatoires
- boosting

en estimant l'**accuracy** et l'**AUC** sur l'échantillon `dtest`. Pour ce faire on crée un `tibble` où on stockera les estimations des probabilités $\mathbf{P}(Y = spam|X = x)$ de chaque méthode :

```
tbl.prev <- matrix(0,ncol=6,nrow=nrow(dtest)) %>% as_tibble()
names(tbl.prev) <- c("Ridge","Lasso","SVM","Arbre","Foret","Boosting")
```

Ridge et lasso

On rappelle que `glmnet` n'admet pas de formule, il faut expliciter la matrice des `X` et le vecteur des `Y`.

```
X.app <- model.matrix(type~.,data=dapp)[,-1]
Y.app <- dapp$type
```

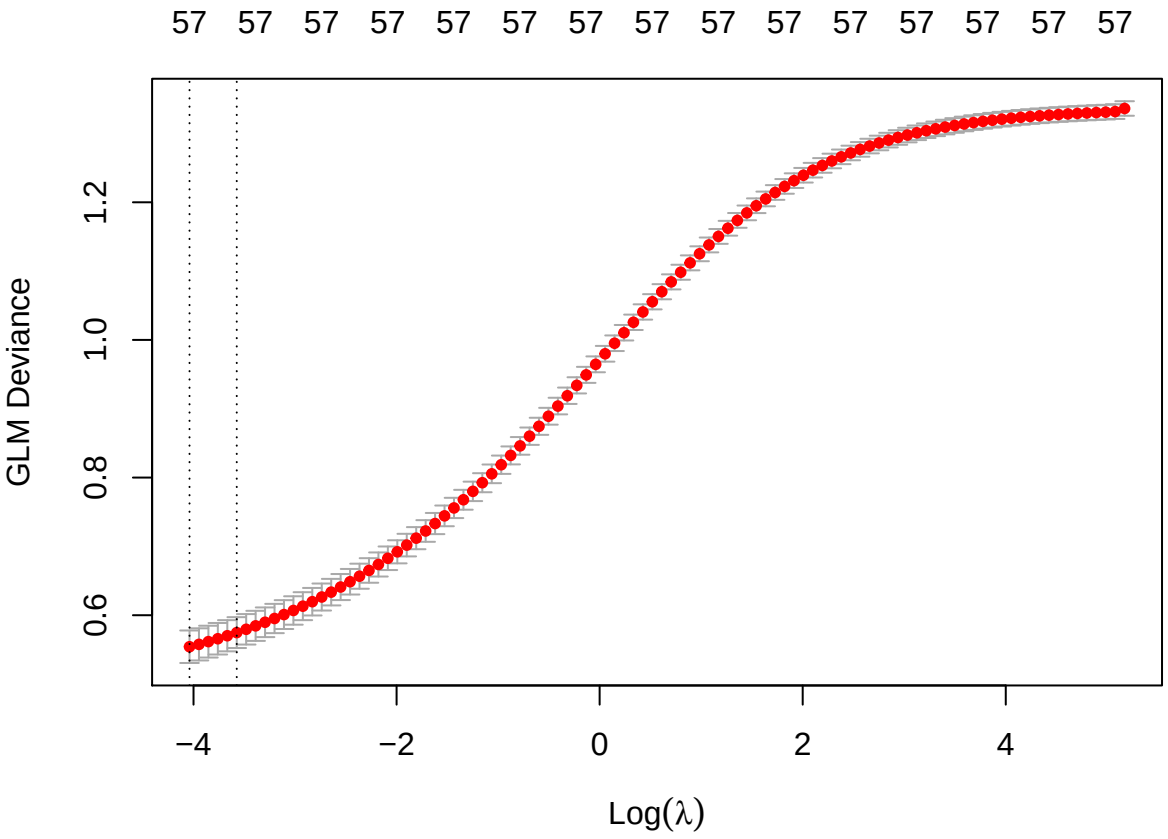
```
X.test <- model.matrix(type~.,data=dtest)[,-1]
Y.test <- dtest$type
```

Exercice 2

Entraîner l'algorithme `ridge` sur `dapp` et compléter la colonne `Ridge` de `tbl.prev`.

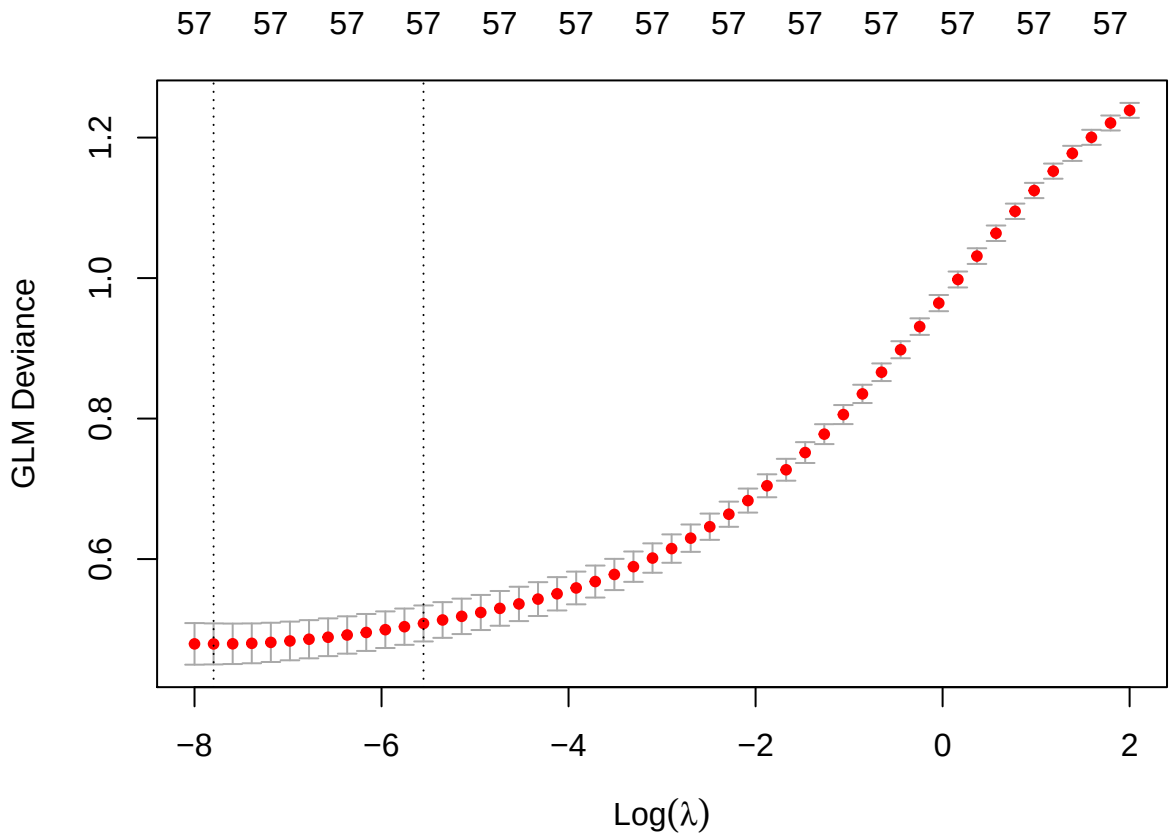
On sélectionne `lambda` par validation croisée.

```
set.seed(123)
ridge.cv <- cv.glmnet(X.app,Y.app,alpha=0,family=binomial)
plot(ridge.cv)
```



La meilleure valeur se trouve à l'extrémité de la grille : il faut changer les valeurs par défaut :

```
set.seed(123)
ridge.cv <- cv.glmnet(X.app,Y.app,alpha=0,family=binomial,
                      lambda=exp(seq(-8,2,length=50)))
plot(ridge.cv)
```



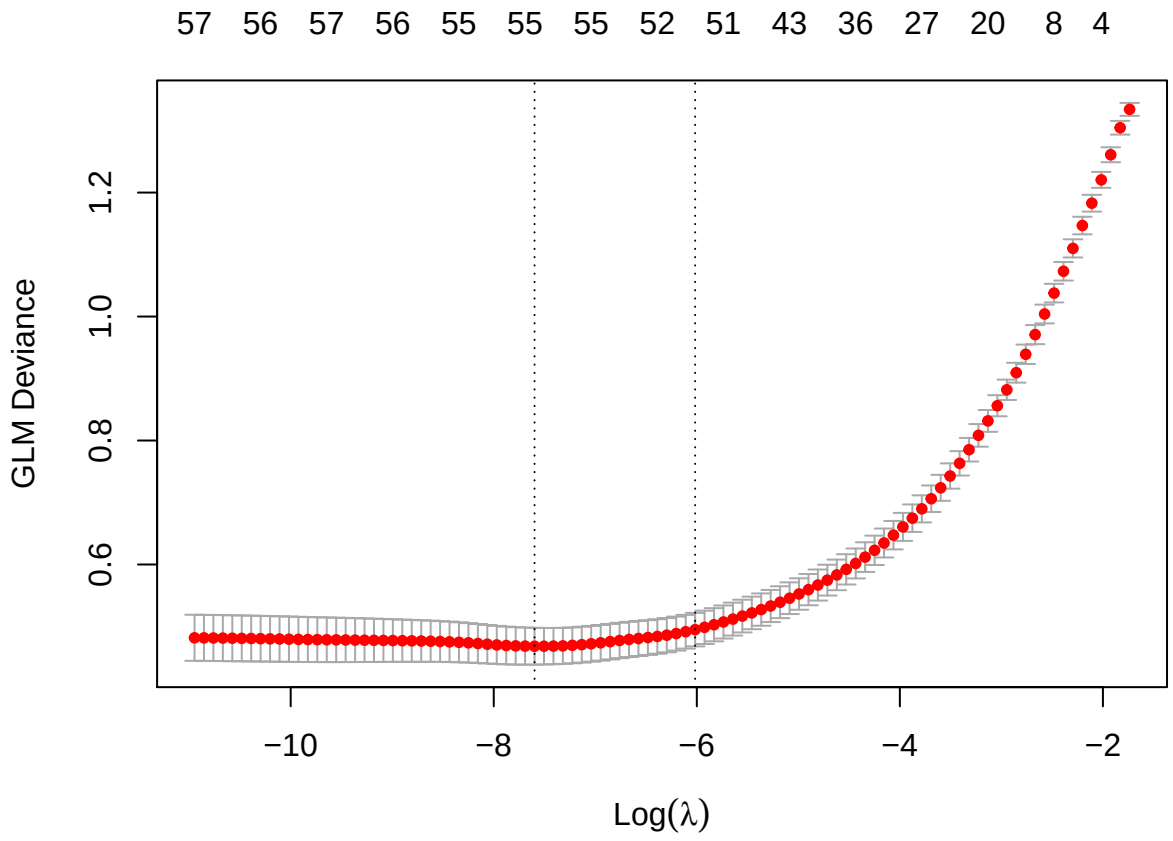

```
prev.ridge <- predict(ridge.cv,newx=X.test,type="response") %>% as.numeric()
tbl.prev$Ridge <- prev.ridge
```

Exercice 3

Faire la même chose pour le lasso.

On sélectionne `lambda` par validation croisée.

```
set.seed(123)
lasso.cv <- cv.glmnet(X.app,Y.app,alpha=1,family=binomial)
plot(lasso.cv)
```



Ici tout est OK, on peut calculer les prévisions :

```
prev.lasso <- predict(lasso.cv,newx=X.test,type="response") %>% as.numeric()
tbl.prev$Lasso <- prev.lasso
```

On peut déjà comparer les **AUC** de ces deux algorithmes :

```
tbl.prev %>% mutate(obs=dtest$type) %>%
  summarize_at(1:2,~roc_auc_vec(truth=obs,estimate=.,event_level="second"))
## # A tibble: 1 x 2
##   Ridge Lasso
##   <dbl> <dbl>
## 1 0.973 0.975
```

Support vector machine

On propose ici de considérer une **SVM** à noyau gaussien. Pour gagner un peu de temps on fixe le `sigma` du noyau à 0.1 et on sélectionnera uniquement le paramètre `cost` :

```
tune_spec_svm <-
  svm_rbf(cost=tune(),rbf_sigma = 0.1) %>%
  set_mode("classification") %>%
  set_engine("kernlab")
svm_wf <- workflow() %>%
  add_model(tune_spec_svm) %>%
  add_formula(type ~ .)
```

On effectue une validation croisée 5 blocs avec la grille de valeurs suivante :

```
set.seed(12345)
re_ech_cv <- vfold_cv(spam,v=5)
grille_C <- tibble(cost=c(0.1,1,5,10))
```

Exercice 4

Sélectionner le paramètre `cost` qui maximise l'**AUC**. On pourra lancer les calculs en parallèle en utilisant le package `doParallel`.

```
cl <- makePSOCKcluster(4)
registerDoParallel(cl)
svm.cv <- svm_wf %>%
  tune_grid(resamples = re_ech_cv,
            grid = grille_C,
            control=control_resamples(save_pred = TRUE),
            metrics=metric_set(roc_auc))
stopCluster(cl)
```

On visualise les résultats

```
svm.cv %>% collect_metrics()
## # A tibble: 4 x 7
##    cost .metric .estimator  mean     n std_err .config
##   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1   0.1 roc_auc binary     0.949     5 0.00379 Preprocessor1_Model1
## 2   1   roc_auc binary     0.967     5 0.00398 Preprocessor1_Model2
## 3   5   roc_auc binary     0.965     5 0.00387 Preprocessor1_Model3
## 4  10   roc_auc binary     0.965     5 0.00389 Preprocessor1_Model4
```

On sélectionne la meilleure valeur et on ré-entraîne sur toutes les données :

```
best_C <- svm.cv %>% select_best()
final_svm <- svm_wf %>%
  finalize_workflow(best_C) %>%
  fit(data = dapp)
```

On peut enfin prédire les individus de l'échantillon test :

```
prev_svm <- predict(final_svm,new_data=dtest,type="prob") %>%
  select(.pred_spam) %>% unlist() %>% as.numeric()
tbl.prev$SVM <- prev_svm
```

et calculer l'**AUC**

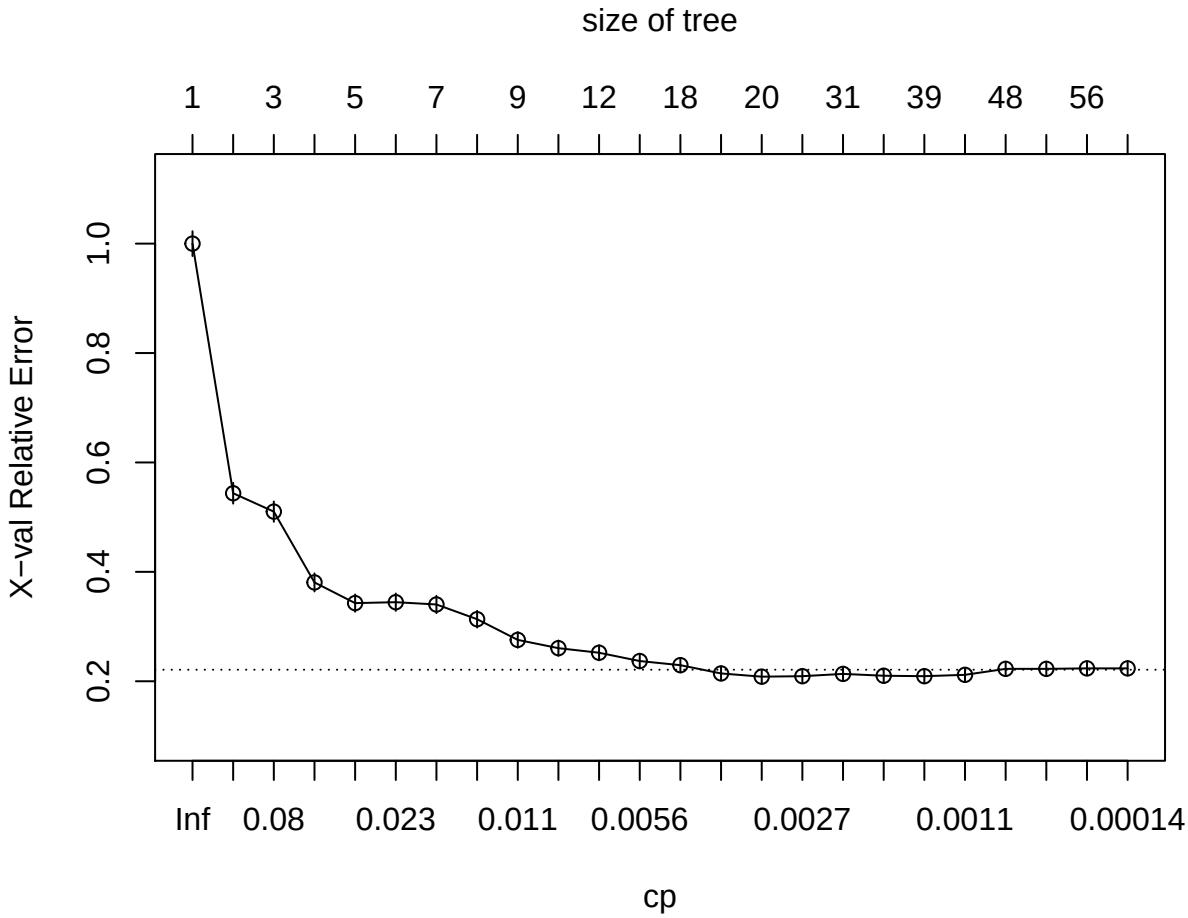
```
tbl.prev %>% mutate(obs=dtest$type) %>%
  summarize_at(1:3,~roc_auc_vec(truth=obs,estimate=.,event_level="second"))
## # A tibble: 1 x 3
##   Ridge Lasso   SVM
##   <dbl> <dbl> <dbl>
## 1 0.973 0.975 0.970
```

Arbres

Exercice 5

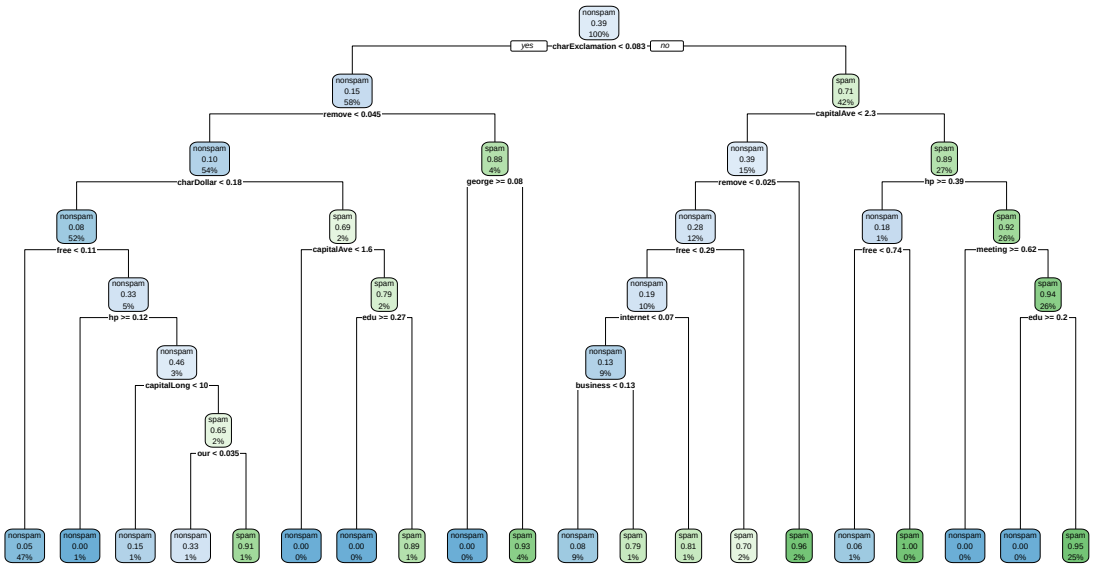
Ajuster un arbre CART sur les données d'apprentissage en utilisant la méthode d'élagage présentée en cours.

```
set.seed(123)
arbre <- rpart(type~.,data=dapp,cp=0.0001,minsplit=15)
plotcp(arbre)
```



Les erreurs de prévision ont l'air de se stabiliser à partir de cp=0.0035 :

```
arbre_final <- prune(arbre,cp=0.0035)
rpart.plot(arbre_final)
```



On calcule les prévisions

```
prev_arbre <- predict(arbre_final,newdata=dtest)[,2]
tbl.prev$Arbre <- prev_arbre
```

et déduit l'**AUC**

```
tbl.prev %>% mutate(obs=dtest$type) %>%
  summarize_at(1:4,~roc_auc_vec(truth=obs,estimate=.,event_level="second"))
## # A tibble: 1 x 4
##   Ridge Lasso   SVM Arbre
##   <dbl> <dbl> <dbl> <dbl>
## 1 0.973 0.975 0.970 0.921
```

Forêts aléatoires

On effectue une forêt aléatoire en conservant les paramètres par défaut :

```
foret.prob <- ranger(type~.,data=dapp,probability=TRUE)
```

Les prévisions sur les données `dtest` s'obtiennent avec

```
prev_foret <- predict(foret.prob,data=dtest)$predictions[,2]
tbl.prev$Foret <- prev_foret
```

Les valeurs d'**AUC** sont maintenant de

```
tbl.prev %>% mutate(obs=dtest$type) %>%
  summarize_at(1:5,~roc_auc_vec(truth=obs,estimate=.,event_level="second"))
## # A tibble: 1 x 5
##   Ridge Lasso   SVM Arbre Foret
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 0.973 0.975 0.970 0.921 0.988
```

Boosting

On utilise le package `gbm` pour l'algorithme `logitboost` :

```
logit1 <- gbm(type~.,data=dapp,distribution="bernoulli",
              interaction.depth=4,
              shrinkage=0.05,n.trees=500)
## Error in gbm.fit(x = x, y = y, offset = offset, distribution = distribution, : Bernoulli requires th
```

La fonction `gbm` nécessite que les variables binaires soient codées en `0/1` :

```
dapp1 <- dapp
dapp1$type <- as.numeric(dapp1$type)-1
set.seed(1234)
boost1 <- gbm(type~.,data=dapp1,distribution="bernoulli",
              interaction.depth=4,
              shrinkage=0.05,n.trees=500)
```

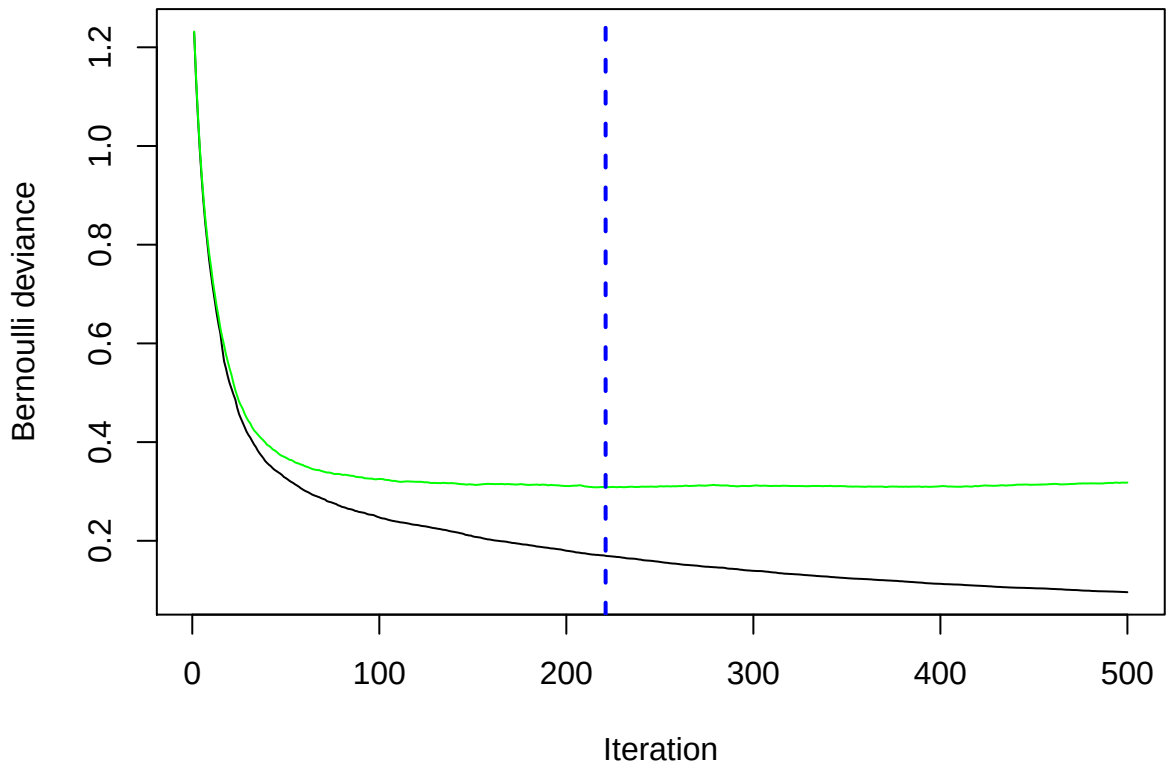
Exercice 6

1. Sélectionner le nombre d'itérations en faisant une validation croisée 5 blocs.

```
set.seed(321)
boost <- gbm(type~.,data=dapp1,distribution="bernoulli",
             interaction.depth=4,
             cv.folds=5,
             shrinkage=0.1,n.trees=500)
```

On obtient le nombre d'itérations avec `gbm.perf` :

```
(ntrees_opt <- gbm.perf(boost))
```




```
## [1] 221
```

2. Calculer les prévision sur les individus de l'échantillon test.

```
prev_boost <- predict(boost,newdata=dtest,
                      type="response",n.trees=ntrees_opt)
tbl.prev$Boosting <- prev_boost
```

Comparaison des algorithmes

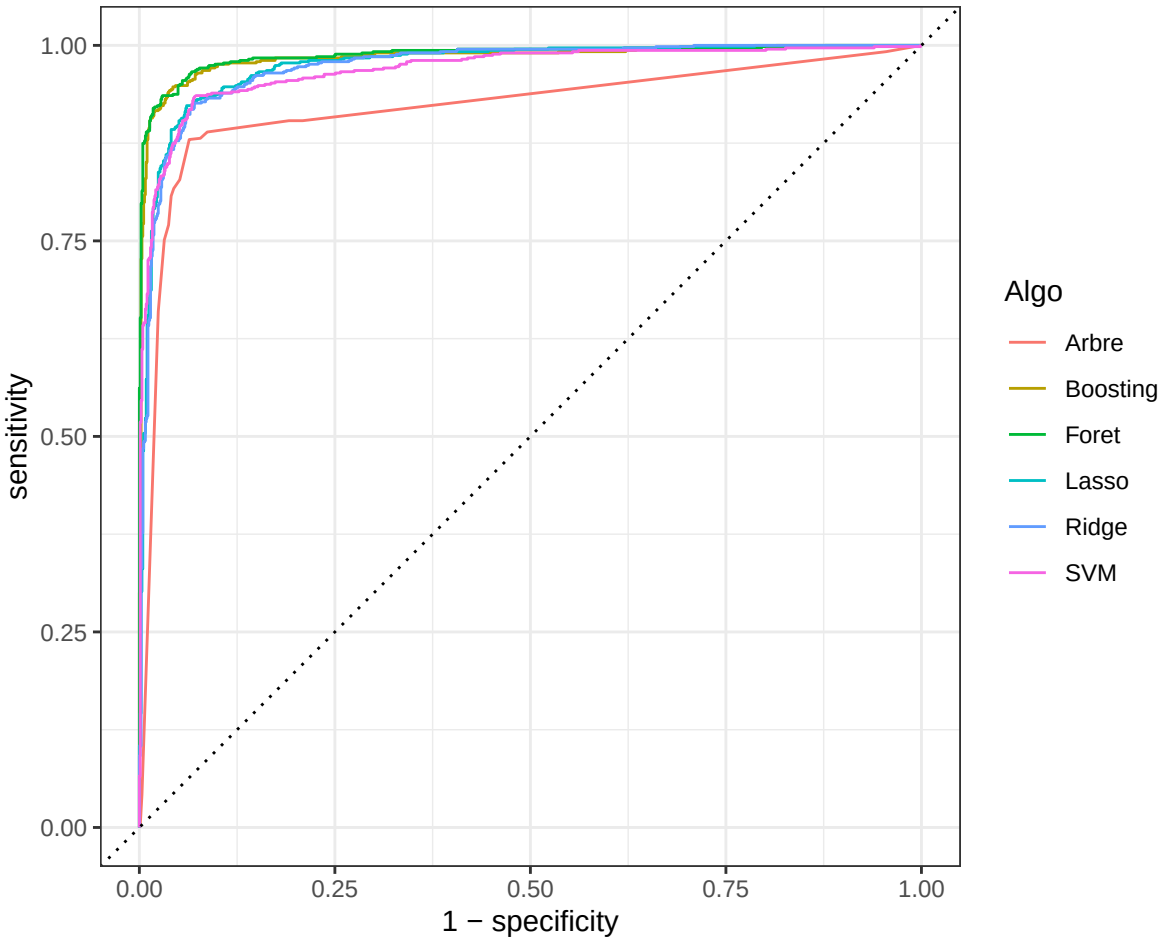
Critère ROC et AUC

Le tibble `tbl.prev` contient les estimations des probabilités $\mathbf{P}(Y = \text{`}spam\text{`}|X = x)$. On obtient les **AUC** avec

```
tbl.prev %>% mutate(obs=dtest$type) %>%
  summarize_at(1:6,~roc_auc_vec(truth=obs,estimate=.,event_level="second")) %>%
  round(3)
## # A tibble: 1 x 6
##   Ridge Lasso   SVM Arbre Foret Boosting
##   <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>
## 1 0.973 0.975  0.97 0.921 0.988    0.985
```

Les méthodes d'agrégation se trouvent en tête pour ce critère. On peut également tracer les courbes **ROC** :

```
tbl.prev %>% mutate(obs=dtest$type) %>%
  pivot_longer(-obs,names_to="Algo",values_to = "score") %>%
  group_by(Algo) %>%
  roc_curve(truth=obs,estimate=score,event_level="second") %>%
  autoplot()
```



Critères basés sur les classes

De nombreux critères comme l'**accuracy**, le **F1-score**, le **kappa de Cohen** sont basés sur la prévision des classes. Cette prévision s'obtient en comparant la probabilité estimée $\mathbf{P}(Y = spam|X = x)$ à un seuil $s \in [0, 1]$. Par exemple avec le seuil 0.5 :

```
prev.class <- round(tbl.prev) %>%
  mutate_all(~recode(.,"0"="nonspam","1"="spam")) %>%
  bind_cols(obs=dtest$type)
head(prev.class)
## # A tibble: 6 x 7
##   Ridge Lasso SVM   Arbre Foret Boosting obs
##   <chr> <chr> <chr> <chr> <chr> <chr>    <fct>
## 1 spam  spam  spam  spam  spam  spam     spam
## 2 spam  spam  spam  spam  spam  spam     spam
## 3 spam  spam  spam  spam  spam  spam     spam
## 4 spam  spam  spam  spam  spam  spam     spam
## 5 spam  spam  spam  spam  spam  spam     spam
## 6 spam  spam  spam  spam  spam  spam     spam
```

Les valeurs des différents critères peuvent s'obtenir à l'aide des fonctions du package `yardstick` :

```
multi_metric <- metric_set(accuracy,bal_accuracy,f_meas,kap)
prev.class %>%
  pivot_longer(-obs,names_to = "Algo",values_to = "classe") %>%
```

```
  mutate(classe=as.factor(classe)) %>%
  group_by(Algo) %>%
  multi_metric(truth=obs,estimate=classe,event_level = "second") %>%
  mutate(.estimate=round(.estimate,3)) %>%
  pivot_wider(-.estimator,names_from=.metric,values_from = .estimate)
## # A tibble: 6 x 5
##   Algo     accuracy bal_accuracy f_meas   kap
##   <chr>       <dbl>        <dbl>  <dbl> <dbl>
## 1 Arbre       0.913        0.908  0.892 0.819
## 2 Boosting    0.952        0.949  0.94  0.9
## 3 Foret       0.954        0.95   0.943 0.905
## 4 Lasso       0.922        0.913  0.9   0.837
## 5 Ridge       0.92         0.912  0.899 0.833
## 6 SVM         0.917        0.905  0.892 0.824
```

Les méthodes d'agrégation sont toujours en tête. Les performance de la SVM sont très faibles, il est fort possible que cela vienne du choix du seuil : la valeur de 0.5 n'est peut être pas bien appropriée...

Les méthodes ont été comparées par une procédure de validation hold out. Elle présente l'avantage d'être simple mais l'inconvénient de manquer de précision au niveau de l'estimation des critères. Il est en effet préférable d'utiliser des validations croisées, voire même de les répéter. On pourra consulter https://lrouviere.github.io/TUTO_ML/correction/comp-algo.html où une validation croisée est effectuée pour estimer les critères.