Scénario: Données QSAR et criblage virtuel de molécules

Résumé

*Différente méthodes de discrimination et de régression sont utilisées et leur qualités prédictives comparées sur deux jeux de données QSAR (*quantitative structure-activity relationship*) de criblage virtuel de molécule. La librairie* `caret` *est utilisée pour "industrialiser" la stratégie de choix de modèle et de méthode. .*

1 Introduction

1.1 Problématique

Avant de lancer de coûteuses synthèses de nouvelles molécules, l'industrie pharmaceutique procède à un criblage (*screening*) de molécules théoriques susceptibles de présenter des applications thérapeutiques intéressantes. Le principe est de caractériser un ensemble de molécules par toute une série de variables physico-chimiques évaluées par des logiciels spécifiques. Connaissant par ailleurs les propriétés thérapeutiques de ces molécules, la question est la suivante : est-il possible de prévoir ces propriétés à partir des variables physico-chimiques ? Le cas échéant il est alors possible d'appliquer le modèle ainsi estimé sur de nouvelles formulations de molécules afin de ne synthétiser que celles suffisamment prometteuses.

Deux jeux de données sont étudiés, l'un associé à un problème de régression (*blood-brain barrier data*), l'autre (*Multidrug Resistance Reversal*) à un problème de classification supervisée ou discrimination. Ces deux jeux de données de même que sept autres sont systématiquement utilisés par Svetnik et al. (2005)[3] pour comparer les performances de huit méthodes récentes de modélisation statistique ou d'apprentissage (boosting, random forest, CART, k-nn, SVM avec noyau linéaire et gaussien, classifieur bayésien naïf) associées à deux stratégies d'optimisation des paramètres (coûteuse en temps de calcul ou non coûteuse). La démarche proposée utilise systématiquement les ressources de la librairie `caret` afin de

1. tester et sélectionner les méthodes de modélisation pertinentes pour chaque jeu de données,
2. optimiser le choix en comparant les distributions des erreurs de prévision évaluées sur un ensemble d'échantillons aléatoires de test.

1.2 Données

Les données sont accessibles dans R au sein de la librairie *caret*.

Multidrug resistance Reversal (MDRR) Agent Data

La variables réponse originale est un rapport mesurant la capacité d'un composant à inverser la résistance de cellules cancéreuses (leucémie) à l'adriamycin (doxorubicine)[1]. Le problème est ici traité comme un cas de discrimination : les composants ou molécules de rapports supérieur à 4,2 sont considérés actifs, ceux de rapport inférieur à 2,0, sont considérés inactifs. Les composants de rapport intermédiaire et jugés à effet "modéré" ont été retirés des données. Il reste ainsi $n = 528$ composants (298 actifs et 230 inactifs). Un logiciel spécifique (*Dragon software*) a produit $p = 342$ descripteurs moléculaires ; `mdrrDescr` est le *data frame* contenant les variables explicatives X tandis que le vecteur de type facteur `mdrrClass` contient les classes d'activité.

Blood-brain barrier data

L'objectif est de modéliser le logarithme du rapport de la concentration d'un composant (molécule) dans le cerveau sur sa concentration dans le sang. Ce rapport mesure donc la capacité de cette molécule à traverser les barrières de protection du cerveau. Chacune des $n = 208$ molécules de domaine public sont caractérisées par trois groupes de descripteurs moléculaires : *MMOE 2D, rule-of-five and Charge Polar Surface or CPSA* ; en tout $p = 134$ variables. Le vecteur `logBBB` contient la variable Y à expliquer (logarithme du rapport de concentration) tandis que le *data frame* `bbbDescr` contient les variables X explicatives.

1. La doxorubicine prévient la croissance des cellules cancéreuses en parasitant l'action du matériel génétique (A.D.N.) qui est nécessaire à leur reproduction.

2 Discrimination et MDRR agent data

2.1 Prise en charge des données

```
library(caret)
data(mdrr)
Ymdrr=mdrrClass
Xmdrr=mdrrDescr
summary(Xmdrr)
```

2.2 Statistiques élémentaires

```
# les diagrammes boîtes des variables explicatives
boxplot(data.frame(Xmdrr))
# montrent une valeur très atypique
hist(Xmdrr[,"VRA1"])
hist(Xmdrr[,"Xt"])
var(Xmdrr[,"Xt"]);var(Xmdrr[,"VRA1"])
boxplot(data.frame(Xmdrr[,-210]))
boxplot(data.frame(scale(Xmdrr)))
var(Xmdrr[,"Xt"])
var(Xmdrr[,"VRA1"])
# Une ACP pour voir
acp=prcomp(Xmdrr, scale=TRUE)
plot(acp)
biplot(acp)
```

Commenter ces quelques résultats. Curieusement, Svetnik et al. (2005) ne proposent rien de particulier pour les valeurs atypiques de même que pour les variables quasi-constantes à zéro ; elles sont donc laissées en l'état. Un prétraitement plus approfondi serait sans doute nécessaire.

2.3 Tests de différents modèles

L'étude se limite à une utilisation de la librairie `caret` (Kunh, 2008 [2]) pour quelques unes des méthodes de modélisation proposées.

Préparation des données

Extraction des échantillons d'apprentissage et de test

```
# indices de l'échantillon d'apprentissage
set.seed(11)
inTrain = createDataPartition(Xmdrr[,1],
   p = 60/100, list = FALSE)
# Extraction des échantillons
trainDescr=Xmdrr[inTrain,]
testDescr=Xmdrr[-inTrain,]
trainY=Ymdrr[inTrain]
testY=Ymdrr[-inTrain]
```

Il est recommandé de centrer et réduire les variables dans plusieurs méthodes. C'est fait systématiquement et simplement en utilisant évidemment les mêmes transformations sur l'échantillon test que celles mises en place sur l'échantillon d'apprentissage. D'autre part, l'erreur de prévision est estimée par validation croisée. Un paramètre de la validation croisée spécifie que des probabilités sont prédites.

```
xTrans=preProcess(trainDescr)
trainDescr=predict(xTrans,trainDescr)
testDescr=predict(xTrans,testDescr)
cvControl=trainControl(method="cv",number=10,
   classProbs =  TRUE)
```

Il peut arriver, qu'après éclatement de l'échantillon, certaines variables très creuses deviennent strictement constantes à zéro. Ceci pose des problèmes à certaines méthodes comme l'analyse discriminante car la variance est nulle.

Estimation et optimisation des modèles

La librairie intègre beaucoup plus de méthodes mais celles sélectionnées ci-dessous semblent les plus pertinentes. Beaucoup des méthodes sont d'autre part inutilisables à cause du nombre de variables (`mars, nnet`) ou encore à cause de la présence de variables identiquement nulles (analyse discriminante) à la suite de la construction de l'échantillon d'apprentissage (variance nulle). Exécuter successivement les blocs de commandes pour tracer séparément cha-

cun des graphes afin de contrôler le bon comportement de l'optimisation du paramètre de complexité de chaque modèle.

```
#1 Support vector machine noyau linéaire
set.seed(2)
svmFit = train(trainDescr, trainY,
   method = "svmLinear", tuneLength = 8,
   trControl = cvControl)
svmFit
plot(svmFit)
#2 Support vector machine noyau gaussien
set.seed(2)
svmgFit = train(trainDescr, trainY,
   method = "svmRadial", tuneLength = 8,
   trControl = cvControl)
svmgFit
plot(svmgFit)
#3 K-nn
set.seed(2)
knnFit = train(trainDescr, trainY,
   method = "knn", tuneLength = 10,
   trControl = cvControl)
knnFit
plot(knnFit)
#4 Random forest
set.seed(2)
rfFit = train(trainDescr, trainY,
   method = "rf", tuneLength = 4,
   trControl = cvControl)
rfFit
plot(rfFit)
#5 boosting
set.seed(2)
gbmFit = train(trainDescr, trainY,
   method = "gbm", tuneLength = 8,
   trControl = cvControl)
gbmFit
plot(gbmFit)
```

Remarque : plusieurs méthodes comme les réseaux de neurones ou GAM (generalised additive model) sont rejetées à cause du nombre élevé de variables ou à cause de la présence de variables constantes (variance nulle) sur l'échantillon sélectionné pour l'apprentissage.

Prévisions et erreurs en test

Les méthodes sélectionnées et optimisées sont ensuite appliquées à la prévision de l'échantillon test. Estimation du taux de bien classés :

```
models=list(gbm=gbmFit,svm=svmFit,svmg=svmgFit,
   knn=knnFit,rf=rfFit)
testPred=predict(models, newdata = testDescr)
# taux de bien classés
lapply(testPred,function(x)mean(x==testY))
```

Tracer les courbes ROC pour analyser spécificité et sensibilité.

```
# Courbes ROC
library(ROCR)
testProb=predict(models, newdata = testDescr,
   type="prob")
predroc=lapply(testProb,
   function(x)prediction(x[,1],testY=="Active"))
perfroc=lapply(predroc,
   function(x)performance(x, "tpr", "fpr"))

plot(perfroc$gbm,col=1)
plot(perfroc$svm,col=2,add=TRUE)
plot(perfroc$svmg,col=3,add=TRUE)
plot(perfroc$knn,col=4,add=TRUE)
plot(perfroc$rf,col=5,add=TRUE)
legend("bottomright",legend=c("boost","svmL",
   "svmG", "knn","RF"),col=c(1:5),pch="_")
```

Importance des variables

Les meilleures méthodes (forêts aléatoires, svm) sont très peu explicites, car de véritables “boîtes noires”, quant au rôle et impact des variables sur la prévision des observations. Néanmoins, des indicateurs d’importance sont proposés pour les forêts aléatoires.

```
rfFit2=randomForest(trainDescr,trainY,
   importance=T)
imp.mdrr=sort(rfFit2$importance[,3],
   decreasing=T)[1:20]
par(xaxt="n")
plot(imp.mdrr,type="h",ylab="Importance",
   xlab="Variables")
points(imp.mdrr,pch=20)
par(xaxt="s")
axis(1,at=1:20,labels=names(imp.mdrr),cex.axis=0.8,
   las=3)
```

2.4 Automatisation

L’échantillon est de faible taille, les estimations des taux de bien classés comme le tracé des courbes ROC sont très dépendants de l’échantillon test ; on peut s’interroger sur l’identité du modèle le plus performant comme sur la réalité des différences entre les méthodes. Il est donc important d’itérer le processus sur plusieurs échantillons tests.

Exécuter la fonction en annexe en choisissant les méthodes semblant les plus performantes :

```
pred.mdrr=pred.autom(Xmdrr,Ymdrr,methodes=c("gbm",
   "svmLinear","svmRadial","knn","rf"),N=50,
   size=c(10,10,10,10,4),type="prob")
```

Puis calculer :

```
# Calcul des taux de bien classés
obs=pred.mdrr$obs
prev.mdrr=pred.mdrr$pred
res.mdrr=lapply(prev.mdrr,function(x)apply((x>0.5)
   ==(obs==1),2,mean))
# Moyennes des taux de bien classés par méthode
lapply(res.mdrr,mean)
# distributions des taux de bien classés
boxplot(data.frame(res.mdrr))
# tracer des courbes ROC moyennes

predroc.mdrr=lapply(prev.mdrr,
   function(x)prediction(x,obs==1))
perfroc.mdrr=lapply(predroc.mdrr,
   function(x)performance(x,"tpr","fpr"))
plot(perfroc.mdrr$gbm,col=1,lwd=1.5,
   avg="vertical")
plot(perfroc.mdrr$svmLinear,col=2,add=TRUE,
   lwd=1.5,avg="vertical")
plot(perfroc.mdrr$svmRadial,col=3,add=TRUE,
   lwd=1.5,avg="vertical")
plot(perfroc.mdrr$knn,add=TRUE,col=4,lwd=1.5,
   avg="vertical")
plot(perfroc.mdrr$rf,add=TRUE,col=5,lwd=1.5,
   avg="vertical")
legend("bottomright",legend=c("boost","svmL",
   "svmG", "knn","RF"),col=c(1:5),pch="_")
```

3 Régression et data BBB

3.1 Prise en charge des données

```
library(caret)
data(BloodBrain)
Ybbb=logBBB
Xbbb=bbbDescr
```

3.2 Statistiques élémentaires

Attention, certaines variables ont très peu de valeurs différentes de 0. Ceci pose un problème lors du tirage de l’échantillon d’apprentissage, elle peut ap-

paraître comme constante et nulle.

```
# la variable à expliquer
hist(Ybbb)
# variables triées par nombre de valeurs nulles
nzero=apply(Xbbb==0,2,sum)
sort(nzero,decreasing =TRUE)[1:10]
# suppression des variables trop problématiques
Xbbb=Xbbb[,nzero<180]
```

Certaines variables apparaissent ainsi plus comme des facteurs que comme des variables quantitatives mais il est difficile de de faire le tri *a postriori*. C'est tout le problème des données publiques pour lesquelles on ne dispose pas de toutes les informations "métier". Certaines variables vont poser des soucis principalement aux méthodes linéaires de modélisation.

```
# les diagrammes boîtes des variables explicatives
boxplot(data.frame(Xbbb))
# la réduction s'impose
boxplot(data.frame(scale(Xbbb)))
# Une ACP pour voir
acp=prcomp(Xbbb, scale=TRUE)
plot(acp)
biplot(acp)
```

3.3 Tests de différents modèles

L'étude se limite à une utilisation de la librairie `caret` (Kunh, 2008 [2]) pour quelques unes des méthodes de modélisation proposées.

Préparation des données

Extraction des échantillons d'apprentissage et de test

```
set.seed(111)
# indices de l'échantillon d'apprentissage
inTrain = createDataPartition(Xbbb[,1],
   p = 80/100, list = FALSE)
# Extraction des échantillons
trainDescr=Xbbb[inTrain,]
testDescr=Xbbb[-inTrain,]
trainY=Ybbb[inTrain]
testY=Ybbb[-inTrain]
```

Il est recommandé de centrer et réduire les variables dans plusieurs méthodes. C'est fait systématiquement et simplement en utilisant évidemment les mêmes transformations sur l'échantillon test que celles calculées sur l'échantillon d'apprentissage. D'autre part, l'erreur de prévision est estimée par la suite par validation croisée.

```
xTrans=preProcess(trainDescr)
trainDescr=predict(xTrans,trainDescr)
testDescr=predict(xTrans,testDescr)
XbbbDes=predict(xTrans,Xbbb) # toutes les données
cvControl=trainControl(method="cv",number=10)
```

Estimation et optimisation des modèles

La librairie intègre beaucoup plus de méthodes mais celles sélectionnées ci-dessous semblent les plus pertinentes. Exécuter successivement les blocs de commandes pour tracer séparément chacun des graphes afin de contrôler le bon comportement de l'optimisation du paramètre de complexité de chaque modèle.

```
#1 Régression linéaire leapBackward
set.seed(2)
llmFit = train(trainDescr, trainY,
   method = "leapBackward", tuneLength = 30,
   trControl = cvControl)
llmFit
plot(llmFit)
#2 régression pls
set.seed(2)
plsFit = train(trainDescr, trainY,
   method = "pls", tuneLength = 10,
   trControl = cvControl)
plsFit
```

```
plot(plsFit)
#3 Support vector machine noyau linéaire
set.seed(2)
svmFit = train(trainDescr, trainY,
   method = "svmLinear", tuneLength = 8,
   trControl = cvControl)
svmFit
plot(svmFit)
#4 Support vector machine noyau gaussien
set.seed(2)
svmgFit = train(trainDescr, trainY,
   method = "svmRadial", tuneLength = 10,
   trControl = cvControl)
svmgFit
plot(svmgFit)
#5 Multivariate Adaptive Regression Spline
set.seed(2)
marsFit = train(trainDescr, trainY,
   method = "earth", tuneLength = 10,
   trControl = cvControl)
marsFit
plot(marsFit)
#6 Kernel-based Regularized Least Squares (KRLS)
set.seed(2)
krlsFit = train(trainDescr, trainY,
   method = "krlsRadial", tuneLength = 6,
   trControl = cvControl)
krlsFit
plot(krlsFit)
#7 random forest
set.seed(2)
rfFit = train(trainDescr, trainY,
   method = "rf", tuneLength = 10,
   trControl = cvControl)
rfFit
plot(rfFit)
```

```
#8 k-nn
set.seed(2)
knnFit = train(trainDescr, trainY,
   method = "knn", tuneLength = 10,
   trControl = cvControl)
knnFit
plot(knnFit)
#9 boosting
set.seed(2)
gbmFit = train(trainDescr, trainY,
   method = "gbm", tuneLength = 6,
   trControl = cvControl)
gbmFit
plot(gbmFit)
```

Prévisions, graphes et erreurs

La librairie offre la possibilité de gérer directement une liste des modèles et donc une liste des résultats.

```
models=list(leaplm=llmFit,pls=plsFit,gbm=gbmFit,
   svm=svmFit,svmg=svmgFit,knn=knnFit,
   mars=marsFit,krls=krlsFit,rf=rfFit)
testPred=predict(models, newdata = testDescr)
# MSE
lapply(testPred,function(x)mean((x-testY)^2))
# graphes
resPlot=extractPrediction(models,
   testX=testDescr, testY=testY)
plotObsVsPred(resPlot)
```

Commenter les résultats.

Importance des variables

Certaines méthodes (forêts aléatoires) sont très peu explicites (boîte noire) quant au rôle des variables sur la prévision des observations. Néanmoins, des indicateurs d'importance sont proposés.

```
rfFit2=randomForest(trainDescr,trainY,
   importance=T)
imp.bbb=sort(rfFit2$importance[,2],
   decreasing=T)[1:20]
par(xaxt="n")
plot(imp.bbb,type="h",ylab="Importance",
   xlab="Variables")
points(imp.bbb,pch=20)
par(xaxt="s")
axis(1,at=1:20,labels=names(imp.bbb),cex.axis=0.8,
   las=3)
```

3.4 Automatisation

L'échantillon est de faible taille, les estimations des erreurs très dépendantes de l'échantillon test sont sujettes à caution et on peut s'interroger sur la réalité des différences entre les différentes méthodes. En regardant les graphiques, on observe qu'il suffit d'une observation pour influencer les résultats. Il est donc important d'itérer le processus sur plusieurs échantillons tests. Il suffit d'intégrer les instructions précédentes dans une boucle comme dans la fonction listée en annexe. Exécuter cette fonction en annexe puis les lignes ci-dessous en choisissant les méthodes semblant les plus performantes :

```
pred.bbb=pred.autom(Xbbb,Ybbb,methodes=c("gbm",
   "pls","krlsRadial","rf","svmRadial"),N=50,
   size=c(6,10,6,10,10))
```

Puis calculer :

```
# Calcul des risques
prev.bbb=pred.bbb$pred
obs=pred.bbb$obs
# carrés des différences
dif=lapply(prev.bbb,function(x)(x-obs)^2)
# risque pour chaque échantillon
moy=lapply(dif,function(x)apply(x,2,mean))
# distributions des risques
boxplot(data.frame(moy))
# moyennes des risques
lapply(moy,mean)
```

Commenter les résultats.

3.5 Collaboration entre "machines"

Avec un objectif non "explicatif" mais de prévision "bute", plutôt que de rechercher la "meilleure" méthode de prévision ou régression comme précédemment, une solution alternative consiste à rechercher une meilleure combinaison de celles-ci.

COBRA

Biau et al. (2013)[1] proposent de combiner une collection de m fonctions de régression $\widehat{f_k}(k = 1, m)$ en tenant compte de la proximité entre les données d'apprentissage avec l'observation à prévoir. Plus précisément, la prévision en $\widehat{y}\mathbf{x}$ est obtenue à partir de m prévisions comme la *moyenne non pondérée des observations* y_i dont les prévisions par $\alpha * m$ machines (α entre 0 et 1) sont dans les boules de rayon ε centrées en chaque $\widehat{f_k}(\mathbf{x}_i)$. Ils montrent que, asymptotiquement, cette combinaison d'estimateurs fait au moins aussi bien, au sens du risque quadratique ou erreur L^2 de prévision, que la meilleure des fonctions de régression de la collection. On se propose de tester cette approche en combinant les modèles qui viennent d'êtres estimés sur les données QSAR ; la librairie R COBRA implémente cette méthode en proposant une procédure d'optimisation des paramètres α et ε de la méthode.

Comme toujours, le point délicat dans l'utilisation d'une librairie est de mettre les données sous le bon format. Les "machines" sont en colonnes. Il s'agit de la collection des 9 modèles précédemment estimés. Chaque ligne contient la prévision ou l'ajustement de chaque observation de l'échantillon d'apprentissage puis la prévision de chaque observation de l'échantillon test.

```
library(COBRA)
# initialisation des paramètres
nappr=nrow(trainDescr)
n=nrow(XbbbDes)
machines.names=c("leaplm","pls","gbm","svmL",
    "svmG","knn","mars","krls","rf")
```

```
machines=matrix(nr=nrow(XbbbDes),
    nc=length(machines.names),data=0)
# prévision de l'apprentissage (ajustement)
# et du test pour toutes les machines
AllPred=as.data.frame(predict(models,
    newdata = XbbbDes))
AllPred=as.matrix(AllPred)
# séparation des ajustements (apprentissage)
# suivis des prévisions (test)
machines[1:nappr,]=AllPred[inTrain,]
machines[(nappr+1):n,]=AllPred[-inTrain,]
res = COBRA(train.design = trainDescr,
            train.responses = trainY,
            test = testDescr,
            machines = machines,
            machines.names = machines.names,
            grid=1000,
            plot=TRUE)
```

Attention, les résultats numériques de COBRA concernent les risques apparents sur l'échantillon d'apprentissage. Voici ceux sur l'échantillon test.

```
# toutes les machines
apply((AllPred[-inTrain,]-Ybbb[-inTrain])^2,
    2,mean)
# COBRA
mean((res$predict-Ybbb[-inTrain])^2)
```

Les résultats obtenus semblent raisonnables mais il faudrait introduire cette approche dans la procédure automatique sur plusieurs échantillons comme ci-dessus afin de rendre la comparaison plus fiable.

SuperLearner

Le principe de l'approche proposée par van der Laan et al. (2007) [4] est simple, il s'agit de calculer une combinaison convexe ou moyenne pondérée de plusieurs prévisions obtenues par exemple avec les précédents modèles. Les paramètres de la combinaison sont optimisés en minimisant un critère de validation croisée. La librairie R `SuperLearner` fournit les fonctions adéquates. Toutes les combinaisons de méthodes ne sont pas possibles, seules une liste prédéfinie est implémentée.

```
library(SuperLearner)
# liste des méthodes disponibles
listWrappers(what = "SL")
SL.library =c("SL.glm","SL.randomForest","SL.gbm",
     "SL.earth", "SL.svm")
res=SuperLearner(trainY, trainDescr,
   newX = testDescr, SL.library=SL.library)
# calcul du risque
mean((res$SL.predict-testY)^2)
```

Commenter les différents résultats. La prévision "brute" plutôt qu"explicative" étant l'objectif recherché sur ce type de données, quelle stratégie semble la plus efficace. Remarque, le *Super Learner* est aussi applicable en discrimination.

Références

[1] G. Biau, A. Ficher, B. Guedj et J. D. Malley, *COBRA : A Nonlinear Aggregation Strategy*, (2013), http://arxiv.org/abs/1303.2236.

[2] Max Kuhn, *Building Predictive Models in R Using the caret Package*, Journal of Statistical Software **28** (2008), n° 5.

[3] V. Svetnik, T. Wang, C. Tong, A. Liaw, R. Sheridan et Q Song, *Boosting : An Ensemble Learning Tool for Compound Classification and QSAR Modeling*, J. Chem. Inf. Model. **45** (2005), 786–799.

[4] M. J. van der Laan, E. C. Polley et A. E. Hubbard, *Super learner*, Statistical Applications in Genetics and Molecular Biology **6 :1** (2007).

Annexe : prévision de N échantilllons

```
pred.autom=function(X,Y,p=1/2,methodes=c("knn",
   "rf"),size=c(10,2),xinit=11,N=10,typerr="cv",
   number=4,type="raw")
# Fonction de prévision de N échantillons tests
```

```
# par une liste de méthodes de régression
# ou classification (uniquement 2 classes)
# Optimisation des paramètres par validation
# croisée (défaut) ou bootstrap ou... (cf. caret)
# X : matrice ou frame des variables explicatives
# Y : variable cible quantitative ou qualitative
# p : proportion entre apprentissage et test
# methodes : liste des méthodes de rdiscrimination
# size : e grille des paramètres à optimiser
# xinit : générateur de nombres aléatoires
# N : nombre de réplications apprentissage / test
# typerr : "cv" ou "boo" ou "oob"
# number : nombre de répétitions CV ou bootstrap
# pred : liste des matrices de prévision
{
# type d'erreur
Control=trainControl(method=typerr,number=number)
# initialisation du générateur
set.seed(xinit)
# liste de matrices stockant les prévisions
# une par méthode, une ligne par observation,
# une colonne par échantillon test
inTrain=createDataPartition(Y,p=p,list=FALSE)
ntest=length(Y[-inTrain])
pred=vector("list",length(methodes))
names(pred)=methodes
pred=lapply(pred,function(x)x=matrix(0,
   nrow=ntest,ncol=N))
# variable cible des échantillons test
obs=matrix(0,ntest,N)

set.seed(xinit)
# N itérations, une par échantillon test
for(i in 1:N)
{
# indices de l'échantillon d'apprentissage
inTrain=createDataPartition(Y,p=p,list=FALSE)
# Extraction des échantillons
trainDescr=X[inTrain,]
testDescr=X[-inTrain,]
trainY=Y[inTrain]
testY=Y[-inTrain]
# stockage des observés de testY
obs[,i]=testY
# centrage et réduction des variables
xTrans=preProcess(trainDescr)
trainDescr=predict(xTrans,trainDescr)
testDescr=predict(xTrans,testDescr)
# estimation et optimisation des modèles
# pour chaque méthode de la liste
for(j in 1:length(methodes))
{
# modélisation
modFit = train(trainDescr, trainY,
   method = methodes[j], tuneLength = size[j],
   trControl = Control)
# prévisions de l'échantillon test
if (type=="prob")  pred[[j]][,i]=predict(modFit,
    newdata = testDescr,type=type)[,1]
else pred[[j]][,i]=predict(modFit,
   newdata = testDescr)
}}
list(pred=pred,obs=obs) # résultats
}
```