TP ozone : algorithmes d'agrégation de modèles

Résumé

Prévision du pic d'ozone par bagging, boosting, random forest.

1 Introduction

Des sections précédentes ont permis d'expérimenter les techniques maintenant classiques de construction d'un modèle de prévision assorties du problème récurrent liés à l'optimisation de la complexité du modèle. Cette section aborde d'autres stratégies dont l'objectif est de s'affranchir de ce problème de choix, par des méthodes se montrant pas ou très peu sensibles au sur-apprentissage ; c'est le cas des algorithmes d'agrégation de modèles.

Sur le plan logiciel, R montre dans cette situation tout son intérêt. La plupart des techniques récentes sont en effet expérimentées avec cet outil et le plus souvent mises à disposition de la communauté scientifique sous la forme d'une librairie afin d'en assurer la "promotion". Pour les techniques d'agrégation de modèles, nous pouvons utiliser les librairies `gbm` et `randomForest` respectivement réalisées par Greg Ridgeway et Leo Breiman. Ce n'est pas systématique, ainsi J. Friedman a retiré l'accès libre à ses fonctions (MART) et créé son entreprise (Salford).

Les objectifs de cette section sont de

1. tester le *bagging* et le choix des ensembles de variables ainsi que le nombre d'échantillons considérés,
2. étudier l'influence des paramètres (profondeur d'arbre, nombre d'itérations, *shrinkage*) sur la qualité de la prévision par *boosting* ;
3. même chose pour les forêts aléatoires (nb de variables tirées (`mtry`à, `nodesize`).
4. Expérimenter les critères de Breiman qui permettent de mesurer l'influence des variables au sein d'une famille agrégée de modèles. Les références bibliographiques sont accessibles sur le site de l'auteur : `www.stat.Berkeley.edu/users/breiman`

2 Bagging

En utilisant la fonction `sample` de R, il est très facile d'écrire un algorithme de *bagging*. Il existe aussi une librairie qui propose des exécutions plus efficaces. Par défaut, l'algorithme construit une famille d'arbres complets (`cp=0`) et donc de faible biais mais de grande variance. L'erreur *out-of-bag* permet de contrôler le nombre d'arbres ; un nombre raisonnable semble suffire ici.

2.1 Régression

L'utilisation est immédiate :

```
library(ipred)
bagging(O3obs~., nbag=50, data=datappr,coob=TRUE)
bagging(O3obs~., nbag=25, data=datappr,coob=TRUE)
```

2.2 Discrimination

```
bagging(DepSeuil~., nbag=50, data=datappq,coob=TRUE)
bagging(DepSeuil~., nbag=25, data=datappq,coob=TRUE)
```

Les résultats sont aléatoires et différents pour chaque exécution. Néanmoins, ceux-ci semblent relativement stables et peu sensibles au nombre d'arbres. Enfin, il est possible de modifier certaines paramètres de `rpart` pour juger de leur influence : ci-dessous, l'élagage de l'arbre.

```
bagging(O3obs~., nbag=50,control=rpart.control(cp=0.1),
   data=datappr,coob=TRUE)
```

Cela nécessite une optimisation du choix du paramètre. Remarquer néanmoins que le nombre d'arbres (`nbag`) n'est pas un paramètre "sensible" et qu'il suffit de se contenter d'arbres "entiers".

Enfin, considérer le choix optimal de la régression au sens de l'erreur `oob` comme une estimation par validation croisée puis tracer le graphe des résidus.

```
bag.reg=bagging(O3obs~., nbag=50, data=datappr,
```

```
   coob=TRUE)
# calcul et graphe des résidus
fit.bagr=predict(bag.reg)
res.bagr=fit.bagr-datappr[,"O3obs"]
plot.res(fit.bagr,res.bagr)
```

Avec le choix optimal pour la discrimination, calculer l'erreur apparente ou par re-substitution mesurant la qualité de l'ajustement.

```
bag.dis=bagging(DepSeuil~., nbag=50, data=datappq,
   coob=TRUE)
fit.bagq=predict(bag.dis,type="class")
#matrice de confusion
table(fit.bagq,datappq$DepSeuil)
```

2.3 Prévision de l'échantillon test

Avec les meilleures combinaisons de paramètres précédentes, estimer les erreurs sur l'échantillon test.

```
bag.reg=bagging(O3obs~., nbag=50, data=datappr)
bag.dis=bagging(DepSeuil~., nbag=50, data=datappq)
pred.bagr=predict(bag.reg,newdata=datestr)
pred.bagq=predict(bag.dis,newdata=datestq,
   type="class")
# Erreur quadratique moyenne de prévision
sum((pred.bagr-datestr[,"O3obs"])^2)/nrow(datestr)
# Matrice de confusion pour la prévision
#  du dépassement de seuil (régression)
table(pred.bagr>150,datestr[,"O3obs"]>150)
# Même chose pour la discrimination
table(pred.bagq,datestq[,"DepSeuil"])
```

Noter les taux d'erreur.

3 Forêt aléatoire

Le programme est disponible dans la librairie `randomForest`. Il est écrit en fortran, donc efficace en terme de rapidité d'exécution, et facile à utiliser grâce à une interface avec R. Les paramètres et sorties sont explicités dans l'aide en ligne.

3.1 Régression

L'utilisation est encore immédiate :

```
library(randomForest)
rf.reg=randomForest(O3obs~., data=datappr,
   xtest=datestr[,-2],ytest=datestr[,"O3obs"],
   ntree=500,do.trace=50,importance=TRUE)
```

3.2 Discrimination

```
rf.dis=randomForest(DepSeuil~.,
   data=datappq,xtest=datestq[,-10],ytest=datestq[,
   "DepSeuil"],ntree=500,do.trace=50,importance=TRUE)
```

Il peut être tenté une optimisation des paramètres `mtry` (nombre de variables tirés à chaque nœud mais cela n'apparaît pas comme indispensable, la technique est suffisamment robuste.

```
# calcul et graphe des résidus
fit.rfr=rf.reg$predicted
res.rfr=fit.rfr-datappr[,"O3obs"]
plot.res(fit.rfr,res.rfr)
```

Calculer l'erreur apparente ou par re-substitution mesurant la qualité de l'ajustement.

```
fit.rfq=rf.dis$predicted
table(fit.rfq,datappq$DepSeuil) #matrice de confusion
```

3.3 Prévision de l'échantillon test

Estimer les erreurs sur l'échantillon test.

```
pred.rfr=rf.reg$test$predicted
pred.rfq=rf.dis$test$predicted
# Erreur quadratique moyenne de prévision
```

```
sum((pred.rfr-datestr[,"O3obs"])^2)/nrow(datestr)
# Matrice de confusion pour la prévision du
# dépassement de seuil (régression)
table(pred.rfr>150,datestr[,"O3obs"]>150)
# Même chose pour la discrimination
table(pred.rfq,datestq[,"DepSeuil"])
```

Comparer les erreurs.

3.4 Importances des variables

Le modèle obtenu est ininterprétable mais des coefficients estiment les contributions des variables dans leur participation à la discrimination.

```
sort(round(importance(rf.reg), 2)[,1])
sort(round(importance(rf.dis), 2)[,4])
```

Comparer avec les variables sélectionnées par les autres modèles.

4 Boosting

Deux librairies proposent des versions relativement sophistiquées des algorithmes de *boosting* dans R. La librairie `boost` propose 4 approches : `adaboost, bagboost` et deux `logitboost`. Développées pour une problématique particulière : l'analyse des données d'expression génomique, elle n'est peut-être pas complètement adaptée aux données étudiées ; elles se limitent à des prédicteurs quantitatifs et peut fournir des résultats étranges. La librairie `gbm` lui est préférée ; elle offre aussi plusieurs versions dépendant de la fonction coût choisie.

La variable à prévoir doit être codée numériquement (0,1) pour cette implémentation. Le nombre d'itérations, ou nombre d'arbres, est paramétré ainsi qu'un coefficient de rétrécissement (*shrinkage*) contrôlant le taux ou pas d'apprentissage. Attention, par défaut, ce paramètre a une valeur très faible (0.001) et il faut un nombre important d'itérations (d'arbres) pour atteindre une estimation raisonnable. La qualité est visualisée par un graphe représentant l'évolution de l'erreur d'apprentissage. D'autre part, une procédure de validation croisée est incorporée ; elle fournit un nombre optimal d'itérations à considérer. Des détails sur cette procédures sont disponibles dans le fichier :

```
library(gbm)
vignette(gbm) # descriptif détaillé de la librairie
```

4.1 Régression

```
boost.reg=gbm(O3obs~., data=datappr,
  distribution="gaussian",n.trees=500, cv.folds=10,
  n.minobsinnode = 5,shrinkage=0.03,verbose=TRUE)
# fixer verbose à FALSE pour éviter trop de sorties
plot(boost.reg$cv.error)
# nombre optimal d'itérations
best.iter=gbm.perf(boost.reg,method="cv")
best.iter
```

On peut s'assurer de l'absence d'un phénomène de sur-apprentissage critique en calculant puis traçant l'évolution de l'erreur sur l'échantillon test en fonction du nombre d'arbre dans le modèle :

```
test=numeric()
for (i in 10:500){
pred.test=predict(boost.reg,newdata=datestr,n.trees=i)
err=sum((pred.test-datestr[,"O3obs"])^2)/nrow(datestr)
test=c(test,err)}
plot(10:500,test,type="l")
abline(v=best.iter)
```

La ligne verticale précise le nombre d'itérations sélectionné. Tester ces fonctions en faisant varier le coefficient de rétrécissement.

4.2 Discrimination

Attention, la variable à modéliser doit être codée $(0, 1)$ et il fait préciser un autre paramètre de distribution pour considérer le bon terme d'erreur.

```
boost.dis=gbm(as.numeric(DepSeuil)-1~.,data=datappq,
   distribution="adaboost",n.trees=500, cv.folds=10,
   n.minobsinnode = 5,shrinkage=0.03,verbose=FALSE)
plot(boost.dis$cv.error)
# nombre optimal d'itérations
```

```
best.ited=gbm.perf(boost.dis,method="cv")
best.ited
```

Comme pour la régression, il est possible de faire varier le coefficient de rétrécissement en l'associant au nombre d'arbres dans le modèle.

```
# calcul et graphe des résidus
fit.boostr=boost.reg$fit
res.boostr=fit.boostr-datappr[,"O3obs"]
plot.res(fit.boostr,res.boostr)
```

Calculer l'erreur apparente ou par re-substitution mesurant la qualité de l'ajustement.

```
fit.boostq=boost.dis$fit>0.5
#matrice de confusion
table(fit.boostq,datappq$DepSeuil)
```

4.3 Echantillon test

La prévision de l'échantillon test et de la matrice de confusion associée sont obtenus par les commandes :

```
boost.reg=gbm(O3obs~., data=datappr,
   distribution="gaussian",n.trees=500, cv.folds=10,
   n.minobsinnode = 5,shrinkage=0.03,verbose=FALSE)
best.iter=gbm.perf(boost.reg,method="cv")
pred.boostr=predict(boost.reg,newdata=datestr,
   n.trees=best.iter)
# Erreur quadratique moyenne de prévision
sum((pred.boostr-datestr[,"O3obs"])^2)/nrow(datestr)
# Matrice de confusion pour la prévision
# du dépassement de seuil (régression)
table(pred.boostr>150,datestr[,"O3obs"]>150)
# Même chose pour la discrimination
pred.boostd=predict(boost.dis,newdata=datestq,
   n.trees=best.ited)
table(as.factor(sign(pred.boostd)),
   datestq[,"DepSeuil"])
```

Quelle stratégie d'agrégation de modèles vous semble fournir le meilleur résultat de prévision ? Est-elle, sur ce jeu de données, plus efficace que les modèles classiques expérimentés auparavant ?

5 Comparaison des courbes ROC

Il y a en tout 6 courbes à comparer ; par souci de lisibilité, elles sont séparées en deux groupes et toujours comparées avec le modèle de covariance quadratique initial.

5.1 Modèles de régression

```
library(ROCR)
rocbagr=pred.bagr/300
predbagr=prediction(rocbagr,datestq$DepSeuil)
perfbagr=performance(predbagr,"tpr","fpr")

rocrfr=pred.rfr/300
predrfr=prediction(rocrfr,datestq$DepSeuil)
perfrfr=performance(predrfr,"tpr","fpr")

rocbstr=pred.boostr/300
predbstr=prediction(rocbstr,datestq$DepSeuil)
perfbstr=performance(predbstr,"tpr","fpr")
# tracer les courbes ROC en les superposant
# pour mieux comparer
plot(perflogit,col=1)
plot(perfbagr,col=2,add=TRUE)
plot(perfrfr,col=3,add=TRUE)
plot(perfbstr,col=4,add=TRUE)
```

5.2 Modèles de discrimination

```
ROCbag=predict(bag.dis,newdata=datestq,type="prob")[,2]
predbag=prediction(ROCbag,datestq$DepSeuil)
perfbag=performance(predbag,"tpr","fpr")
```

```
ROCrf=rf.dis$test$vote[,2]
predrf=prediction(ROCrf,datestq$DepSeuil)
perfrf=performance(predrf,"tpr","fpr")

ROCboost=predict(boost.reg,newdata=datestr,
   n.trees=best.ited)
predboost=prediction(ROCboost,datestq$DepSeuil)
perfboost=performance(predboost,"tpr","fpr")

plot(perflogit,col=1)
plot(perfbag,col=2,add=TRUE)
plot(perfrf,col=3,add=TRUE)
plot(perfboost,col=4,add=TRUE)
legend("bottomright",legend=c("acova","bag",
   "randF","boost"),col=c(1:4),pch="_")
```

Une méthode de prévision d'occurence de dépassement du pic de pollution est-elle globalement meilleure ?