Scénario: Calibration (1-cookies) de spectres *NIR*

Résumé

Modélisation (ou calibration) en très grande dimension $p >> n$ où les variables explicatives sont les discrétisations de spectres en proche infra rouge (NIR) et la variable à expliquer la part de sucre dans une pâte à gâteau. Différente méthodes de régression (ridge, lasso, lars, elasticnet, svm, pls, sur composantes principales...) adaptées à cette situation sont utilisés et leur qualités prédictives comparées. La librairie `caret` *est ensuite utilisée pour "industrialiser" la stratégie de choix de modèle et de méthode.*

Un deuxième scénario sur des données NIR (Tecator) aborde des méthodes non linéaires et l'usage des splines pour lisser et calculer les dérivées.

1 Introduction

1.1 Problématique

Le travail présenté s'intéresse à un problème de contrôle de qualité sur une chaîne de fabrication de biscuits (cookies). Il est nécessaire de contrôler le mélange des ingrédients avant cuisson afin de s'assurer que les proportions en lipides, sucre, farine, eau, sont bien respectées c'est-à-dire proches des valeurs nominales de la recette qui a fait la réputation de l'entreprise avant l'industrialisation de la production. Il s'agit de savoir s'il est possible de dépister au plus tôt une dérive afin d'intervenir sur les équipements concernés. Les mesures et analyses, faites dans un laboratoire classique de chimie, sont relativement longues et coûteuses ; elles ne peuvent être entreprises pour un suivi régulier ou même en continue de la production. C'est pour cela qu'il à été décidé l'utilisation d'un spectromètre en proche infrarouge (NIR). L'appareil envisagé mesure l'absorbance c'est-à-dire les spectres dans les longuers d'ondes du proche infra-rouge (BIR).

Ce type de problème est très classique en agro-alimentaire et plus généralement dans l'industrie et correspond à une étude dite de calibration en chimiométrie. Une très abondante littérature lui est consacrée.

1.2 Données

Les données originales sont dues à Osbone et al. (1984) [4] et ont été souvent utilisées pour la comparaison de méthodes (Stone et al. 1990 [5], Brown et al. 2001 [1], Krämer et al. 2008 [2]). Elles sont accessibles dans R au sein du package *ppls*. Les mesures ont été faites sur deux échantillons, l'un de taille 40 prévu pour l'apprentissage, l'autre de taille 32 pour les tests. Pour chacun de ces 72 biscuits, les compositions en lipides, sucre, farine, eau, sont mesurées par une approche classique tandis que le spectre est observé sur toutes les longueurs d'ondes entre 1100 et 2498 nanomètres, régulièrement espacés de 2 nanomètres. Nous avons donc 700 valeurs observées, ou variables potentiellement explicatives, par échantillon de pâte à biscuit.

Typiquement, cette étude se déroule dans un contexte de très grande dimension avec $p >> n$.

1.3 Objectif

L'objectif principal est de répondre à la question suivante : est-il possible de prévoir ces quantités à partir des spectres ? En cas de réponse positive, le gain de temps, donc d'argent, serait immédiat. L'étude est restreinte à la seule modélisation du taux de sucre pour la recherche d'un meilleur modèle de prévision. Il s'agit donc d'évaluer, comparer différentes stratégies et méthodes pour aboutir à celle la plus efficace. Le travail est découpé en deux parties, dans la première sont testées différentes méthodes en construisant les modèles "à la main" ou pas à pas tandis que la 2ème partie fait appel à une librairie (`caret`) qui propose une comparaison automatique d'optimisaiton des paramètres pour un vaste ensemble de méthodes.

2 Approche exploratoire

2.1 Prise en charge des données

```
library(ppls)
```

```
data(cookie)
# extraire le taux de sucre et les spectres
cook = data.frame(cookie[,702],cookie[,1:700])
names(cook)= c("sucre",paste("X",1:700,sep=""))
```

2.2 Statistiques élémentaires

```
# la variable à expliquer
hist(cook[,"sucre"])
# les diagrammes boîtes puis variances des
# variables explicatives
boxplot(cook[,-1])
hist(var(cook[,-1]))
# les spectres colorés par l'intensité en sucre
coul=rainbow(20)[as.integer(as.factor
    (as.integer(cook[,1]-10)))]
ts.plot(t(cook[,-1]),col=coul)
# Une ACP pour voir
acp=prcomp(cook[,-1])
plot(acp)
biplot(acp)
plot(acp$x,col=coul)
plot.ts(acp$rotation[,1:10])
```

Une fonction sera utilisée pour une représentation homogène des résidus des modèles :

```
plot.res=function(x,y,titre="titre")
{
plot(x,y,col="blue",ylab="Résidus",
  xlab="Valeurs predites",main=titre)
abline(h=0,col="green")
}
```

3 Modélisation artisanale

La modélisation est construite méthode par méthode pour mieux en assimiler le déroulement. Chaque méthode de modélisation possède des spécificités, notamment dans la manière d'optimiser plus ou moins facilement les valeurs des paramètres. Il est indispensable de bien se familiariser avec ces différents éléments.

3.1 Les échantillons

Les 40 premières lignes sont usuellement considérées pour l'apprentissage tandis que les 32 dernières sont l'échantillon test. Ce sera fait ici pour éventuellement comparer avec des résultats de la littérature mais les données seront ensuite regroupées pour itérer la répartition aléatoire entre apprentissage et test.

```
# extraire apprentissage et test
cook.app=cook[1:40,]
cook.test=cook[41:72,]
```

L'optimisation de paramètres est calculée en minimisant une erreur estimée par "4-fold" validation croisée.

```
# Sélection des indices de validation croisée
library(pls)
set.seed(87)
cvseg=cvsegments(nrow(cook.app),k=4,
    type="random")
```

3.2 Régression ridge

```
library(MASS)
# paramètres en fonction de la pénalisation
plot(lm.ridge(sucre ~ ., data=cook.app,
   lambda = seq(0,1,0.01)))
plot(lm.ridge(sucre ~ ., data=cook.app,
   lambda = seq(0,0.2,0.001)))
cook.ridge=lm.ridge(sucre ~ ., data=cook.app,
   lambda = seq(0,0.2,0.001))
```

Le choix visuel n'est pas possible, une fonction de calcul de la validation croisée est utilisée :

```
# fonction de 4-fold validatin croisée
choix.kappa=function(kappamax,cvseg,nbe=100){
press=rep(0,nbe)
for (i in 1:length(cvseg)){
valid=cook.app[unlist(cvseg[i]),]
modele=lm.ridge(sucre~.,
   data=cook.app[unlist(cvseg[-i]),],
   lambda=seq(0,kappamax,length=nbe))
coeff=coef(modele)
prediction=matrix(coeff[,1],nrow(coeff),
   nrow(valid))+coeff[,-1]%*%
   t(data.matrix(valid[,-1]))
press=press+rowSums((matrix(valid[,1],
   nrow(coeff),nrow(valid),byrow=T)-
   prediction)^2)
      }
kappaet=seq(0,kappamax,length=
   nbe)[which.min(press)]
return(list(kappaet=kappaet,press=press))
   }
# exécution
res=choix.kappa(0.5,cvseg,nbe=1000)
plot(seq(0,0.5,length=1000),res$press)
# valeur optimale
kappaet=res$kappaet
cook.ridgeo=lm.ridge(sucre~.,data=cook.app,
   lambda=kappaet)
coeff=coef(cook.ridgeo)
# Calcul des valeurs ajustées et des résidus
fit.rid=rep(coeff[1],nrow(cook.app))+
    as.vector(coeff[-1]%*%
    t(data.matrix(cook.app[,-1])))
plot(fit.rid,cook.app[,"sucre"])
res.rid=fit.rid-cook.app[,"sucre"]
plot.res(fit.rid,res.rid,titre="")
```

Commenter la qualité d'ajustement du modèle.

Prévision de l'échantillon test et calcul de l'erreur de prévision

```
ychap=rep(coeff[1],nrow(cook.test))+
   as.vector(coeff[-1]%*%
   t(data.matrix(cook.test[,-1])))
plot(ychap,cook.test[,1])
mean((cook.test[,1]-ychap)^2)
```

Noter l'erreur trouvée pour comparer avec les autres méthodes.

3.3 Régression PLS

Estimation de la régression PLS sur au plus 28 composantes orthogonales et graphe des résidus avec 28 composantes.

```
cook.pls=plsr(sucre~.,ncomp=28,
 data=cook.app,validation="CV",segments=cvseg)
print(cook.pls)
plot(cook.pls)
```

Le nombre de composantes et donc la complexité du modèle est optimisé par validation croisée.

```
msepcv.pls=MSEP(cook.pls,estimate=
   c("train","CV"))
plot(msepcv.pls,type="l")
```

Choisir le nombre optimal de composantes puis réestimer le modèle, prévoir l'échantillon test et calculer l'erreur sur cet échantillon.

```
ncompo=which.min(msepcv.pls$val["CV",,])-1
# modèle optimal
cook.plso=plsr(sucre~.,ncomp=ncompo,
   data=cook.app)
# résidus
fit.pls=predict(cook.plso,ncomp=
   1:ncompo)[,,ncompo]
plot(fit.pls,cook.app[,"sucre"])
res.pls=fit.pls-cook.app[,"sucre"]
plot.res(fit.pls,res.pls)
```

```
# erreur quadratique moyenne
ychap=predict(cook.plso,newdata=
   cook.test)[,1,ncompo]
plot(ychap,cook.test[,1])
mean((cook.test[,"sucre"]-ychap)^2)
```

Comparer les qualités d'ajustement des régression ridge et PLS ainsi que les erreurs de prévision.

Attention, une observation (la 23ème) semble se démarquer de l'échantillon. Ce n'est pas "flagrant" mais elle est signalée comme atypique (*outlier*) dans la littérature. La repérer sur le graphique ; elle sera prise en compte dans la 2ème partie.

Trois graphiques analogues à ceux de l'ACP viennent compléter les résultats mais ceux-ci sont moins utiles ou pertinents qu'en version 2 de la PLS quand la variable à expliquer Y est multidimensionnelle.

```
scoreplot(cook.pls)
corrplot(cook.pls)
loadingplot(cook.pls)
```

3.4 Régression sur composantes principales

Comme précédemment, le nombre de composantes est optimisé par validation croisée.

```
cook.pcr=pcr(sucre~.,ncomp=28,data=cook.app,
    validation="CV",segments=cvseg)
msepcv.pcr=MSEP(cook.pcr,estimate=
   c("train","CV"))
plot(msepcv.pcr,type="l")
ncompo=which.min(msepcv.pcr$val["CV",,])-1
cook.pcro=pcr(sucre~.,ncomp=ncompo,
   data=cook.app)
```

Puis calcul et graphe des résidus avant d'estimer l'erreur de prévision sur l'échantillon test.

```
fit.pcr=predict(cook.pcro,ncomp=1:6)[,,6]
plot(fit.pcr,cook.app[,"sucre"])
res.pcr=fit.pcr-cook.app[,"sucre"]
plot.res(fit.pcr,res.pcr)
ychap=predict(cook.pcro,newdata=
   cook.test)[,1,ncompo]
mean((cook.test[,"sucre"]-ychap)^2)
```

Comme précédemment comparer qualités d'ajustement et de prévision.

3.5 Random forest

Sur ce jeu de données, les agrégations de modèles d'arbres se montrent peu performantes.

```
library(randomForest)
# estimation
cook.rf=randomForest(sucre~.,data=cook.app,
   xtest=cook.test[,-1],ytest=
   cook.test[,"sucre"],do.trace=50,
   importance=TRUE,mtry=50,corr.bias=TRUE)
# résidus
fit.rfr=cook.rf$predicted
plot(fit.rfr,cook.app[,"sucre"])
res.rfr=fit.rfr-cook.app[,"sucre"]
plot.res(fit.rfr,res.rfr)
# estimation de l'erreur
pred.rfr=cook.rf$test$predicted
mean((pred.rfr-cook.test[,"sucre"])^2)
```

A noter qu'en choisissant `mtry=700`, revenant à faire du *bagging*, l'erreur est légèrement améliorée mais il est clair que l'approche non linéaire par arbre n'est pas pertinente sur cet exemple pour lequel un modèle linéaire fournit un bon ajustement.

Pour mémoire, voici la façon d'éditer les parties importantes du spectre.

```
sort(round(importance(cook.rf), 2)[,1])[1:10]
summary(importance(cook.rf))
plot(importance(cook.rf)[,2])
```

3.6 Régression Lasso par algorithme lars

Le déroulement suit les mêmes étapes.

```
library(lars)
frac.delta=seq(0,1,length=1000)
mse.cv=cv.lars(data.matrix(cook.app[,-1]),
   cook.app[,1],K=4,se=F,index=frac.delta,
   use.Gram=F)
plot(frac.delta,mse.cv$cv,xlab="delta",
   ylab="MSEP")
frac.delta.o=frac.delta[which.min(mse.cv$cv)]

cook.lasso=lars(data.matrix(cook.app[,-1]),
   cook.app[,1],use.Gram=F)

fit.lasso=predict(cook.lasso,
   data.matrix(cook.app[,-1]),s=frac.delta.o,
   mode="fraction")$fit
plot(fit.lasso,cook.app[,"sucre"])
res.lasso=fit.lasso-cook.app[,"sucre"]
plot.res(fit.lasso,res.lasso)

pred.lasso=predict(cook.lasso,
   data.matrix(cook.test[,-1]),
   s=frac.delta.o,mode="fraction")$fit
mean((pred.lasso-cook.test[,"sucre"])^2)
```

Pas très utiles mais jolis : les chemins de régularisation des coefficients.

```
plot(cook.lasso,breaks=F)
```

4 Modélisation industrielle

Kunh (2008)[3] a développé tout un environnement (librairie `caret`) qui facilite modélisation et optimisation des paramètres pour quasiment toutes les méthodes de classification et/ou régression disponibles dans des librairies R et, il y en a beaucoup. Chacune possède une logique ou une stratégie particulière de gestion des paramètres ; cet outil permet d'en uniformiser la syntaxe. Cela concerne la transmission des données, du ou des paramètres de complexité à optimiser lors de l'appel des fonctions et le calcul des prévisions.

La librairie possède plusieurs fonctionnalités qui sont mises en œuvre et illustrées successivement dans le cas présent de la régression :

- extraction des échantillons d'apprentissage et test,
- transformation et sélection préliminaire des variables,
- estimation des modèles,
- choix du mode d'estimation de l'erreur de prévision (validation croisée, bootstrap, .632 bootstrap, *out of bag*...)
- optimisation des paramètres (complexité, pénalisation),
- calcul des prévisions de l'échantillon test ou d'un autre jeu de données,
- graphes de diagnostic (résidus).

Les mêmes fonctionnalités sont proposées en situation de classification supervisée avec des spécificités pour le type de l'erreur (mauvais classement, courbe ROC).

De façon générale, le défi pour la suite, est d'arriver à faire mieux en terme de qualité de prévision que la régression PLS.

4.1 Restriction des données

Osbone et al. (1984) [4] et leur successeurs s'accordent sur le fait que l'observation 23 est atypique. Ils font remarquer de plus que les extrémités des spectres sont bruitées ; en les retirant, les erreurs de prévision sont réduites. Enfin, Goutils et Fearn (1996) suggèrent également que la discrétisation est trop "fine'. Que devient la prévision avec quatre fois moins de variables ?

Il s'agit donc de reprendre les calculs des erreurs de prévisions avec l'extraction ci-dessous qui supprime la 23ème observation, les 50 premières et dernières valeurs du spectre tout en ne considérant qu'une valeur sur 4 :

```
cook2=cook[-23,c(1,seq(50,650,4))]
```

4.2 Préparation des données

Extraction des échantillons d'apprentissage et de test

```
# Mêmes indices de l'échantillon d'apprentissage
inTrain = 1:39
```

```
trainDescr=cook2[inTrain,-1]
testDescr=cook2[-inTrain,-1]
trainY=cook2[inTrain,1]
testY=cook2[-inTrain,1]
```

Il est recommander 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.

```
library(caret)
xTrans=preProcess(trainDescr)
trainDescr=predict(xTrans,trainDescr)
testDescr=predict(xTrans,testDescr)
```

Plusieurs estimations sans biais des erreurs de prévision sont proposées comme par exemple :

```
btControl=trainControl(method="boot",
   number=100)
cvControl=trainControl(method="cv",number=4)
```

4.3 Estimation et optimisation des modèles

La librairie intègre beaucoup plus de méthodes mais celles séelctionnées ci-dessous semblent les plus pertinentes. Exécuter successivement les blocs de commandes pour tracer séparamment chacun des graphes.

```
# régression ridge
set.seed(2)
ridgeFit = train(trainDescr, trainY,
   method = "ridge", tuneLength = 8,
   trControl = cvControl)
ridgeFit
plot(ridgeFit)
# régression pls
set.seed(2)
plsFit = train(trainDescr, trainY,
   method = "pls", tuneLength = 12,
   trControl = cvControl)
plsFit
plot(plsFit)
# régression lasso
set.seed(2)
lassoFit = train(trainDescr, trainY,
   method = "lasso", tuneLength = 8,
   trControl = cvControl)
lassoFit
plot(lassoFit)
# régression lars
set.seed(2)
larsFit = train(trainDescr, trainY,
   method = "lars", tuneLength = 8,
   trControl = cvControl)
larsFit
plot(larsFit)
# elasticnet
set.seed(2)
enetFit = train(trainDescr, trainY,
   method = "enet", tuneLength = 8,
   trControl = cvControl)
enetFit
plot(enetFit)
plot(predict(enetFit),trainY)
# support vector machine
set.seed(2)
svmFit = train(trainDescr, trainY,
   method = "svmLinear", tuneLength = 5,
   trControl = cvControl)
svmFit
plot(svmFit)
```

4.4 Prévisions, graphes et erreurs

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

```
models=list(ridge=ridgeFit,pls=plsFit,
    lasso=lassoFit,lars=larsFit,
    elasticnet=enetFit,svm=svmFit)
testPred=predict(models, newdata = testDescr)
lapply(testPred, function(x) mean((x-testY)^2))
resPlot=extractPrediction(models,
   testX=testDescr, testY=testY)
plotObsVsPred(resPlot)
```

4.5 Automatisation

L'échantillon est de faible taille, les estimations des erreurs, même par bootstrap 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.

Evidemment le temps de calcul s'en ressent mais, le cas échéant, c'est-à-dire en cas d'accès à un cluster ou une machine à plusieurs processeurs, la librairie fournit un accès directe à la parallèlisation des calculs en interfaçant les outils de NetWorkSpaces (Scientific Computing Associates, Inc. 2007) qui sont également en accès libre.

```
# fixer N le nombre d'itérations
xx=50
# fixer l'initialisation du générateur
xxx=11
#exécuter le programme en annexe

# patienter ...

# Moyennes des erreurs par méthode
colMeans(res.reg)
# distributions des erreurs
boxplot(data.frame((res.reg)))
```

Tester l'autre estimation par bootstrap des erreurs quadratique de prévision, c'est plus long ! Commenter, conclure...

Références

[1] P.J. Brown, T. Fearn et M. Vannucci, *Bayesian Wavelet Regression on Curves with Applications to a Spectroscopic Calibration Problem*, Journal of the American Statistical Society **96** (2001), 398–408.

[2] Nicole Krämer, Anne Laure Boulesteix et Gerhard Tutz, *Penalized Partial Least Squares with applications to B-spline transformations and functional data*, Chemometrics and Intelligent Laboratory Systems **94** (2008), 60–69.

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

[4] B. G. Osborne, T. Fearn, A. R. Miller et S. Douglas, *Application of Near Infrared Reflectance spectroscopy to the compositional analysis of biscuits and biscuit doughs*, J. Sci. Food Agric. **35** (1984), 99–105.

[5] M. Stone et R. J. Brooks, *Continuum regression : cross-validated sequentially constructed prediction embracing ordinary least squares, partial least squares and principal components regression*, Journal of The Royal Statistical Society B **52** (1990), 237–269.

Annexe : Programme

```
#######
# changer l'initialisation xxx du générateur
set.seed(xxx)
# fixer le nombre xx d'iterations ou nombre
# de tirages "apprentissage / test"
N = xx
################
# Librairies
# les autres sont chargées par la commande
# "train" de "caret"
library(ppls)
library(caret)
```

```
# Gestion des données
data(cookie)
# extraire le taux de sucre et les spectres
cook = data.frame(cookie[,702],cookie[,1:700])
names(cook)= c("sucre",paste("X",1:700,sep=""))
# extraction des données "pertinentes"
cook2=cook[-23,c(1,seq(50,650,4))]

methodes= c("ridge","pls","lasso","lars",
   "e.net","svm")

# initialisation des matrices stockant
# les erreurs quadratiques de prévision
res.reg = matrix(0,nrow=N,ncol=length(methodes))
colnames(res.reg) = methodes


for(i in 1:N)
{
# indices de l'échantillon d'apprentissage
inTrain = createDataPartition(cook2[,1],
     p = 55/100, list = FALSE)
# Extraction des échantillons
trainDescr=cook2[inTrain,-1]
testDescr=cook2[-inTrain,-1]
trainY=cook2[inTrain,1]
testY=cook2[-inTrain,1]
# centrage et réduction des variables
xTrans=preProcess(trainDescr)
trainDescr=predict(xTrans,trainDescr)
testDescr=predict(xTrans,testDescr)
# estimation de l'erreur par validation croisée
# ou par bootstrap
# btControl=trainControl(method="boot",number=100)
cvControl=trainControl(method="cv",number=4)
# estimation et optimisation des modèles
# régression ridge
ridgeFit = train(trainDescr, trainY,
   method = "ridge", tuneLength = 10,
   trControl = cvControl)
# régression pls
plsFit = train(trainDescr, trainY,
   method = "pls", tuneLength = 12,
   trControl = cvControl)
# régression lasso
lassoFit = train(trainDescr, trainY,
   method = "lasso", tuneLength = 10,
   trControl = cvControl)
# régression lars
larsFit = train(trainDescr, trainY,
   method = "lars", tuneLength = 10,
   trControl = cvControl)
# elasticnet
enetFit = train(trainDescr, trainY,
   method = "enet", tuneLength = 10,
   trControl = cvControl)
# support vector machine
svmFit = train(trainDescr, trainY,
   method = "svmLinear", tuneLength = 5,
   trControl = cvControl)
####
# Calcul des erreurs
models=list(ridge=ridgeFit,pls=plsFit,
    lasso=lassoFit,lars=larsFit,
    elasticnet=enetFit,svm=svmFit)
testPred=predict(models, newdata = testDescr)
res.reg[i,]=do.call(c,lapply(testPred,
     function(x) mean((x-testY)^2)))
}
```