Scénario: Calibration (2-Tecator) de spectres *NIR*

Résumé

Modélisation (ou calibration) en grande dimension. Les variables explicatives sont les discrétisations de spectres en proche infra rouge (NIR) et la variable à expliquer la part de matière grasse dans des échantillons de viande. Différente méthodes de régression (lasso, krls, pls, svm, mars...) sont utilisées et leur qualités prédictives comparées. L'étude de la forme des spectres par l'introduction des dérivées est utile sur ce jeu de données. La librairie `caret` *est directement utilisée pour "industrialiser" la stratégie de choix de modèle et de méthode. Ce scénario vient compléter celui des données de pâte à biscuit.*

1 Introduction

1.1 Problématique

These data are recorded on a Tecator Infratec Food and Feed Analyzer working in the wavelength range 850 - 1050 nm by the Near Infrared Transmission (NIT) principle. Each sample contains finely chopped pure meat with different moisture, fat and protein contents. [1]

Chaque échantillon de viande est caractérisé par des mesures d'absorption pour 100 longueurs d'ondes échantillonnées dans le proche infra rouge ainsi que par des mesures chimiques de la proportion (pourcentage) en eau, graisse et protéine. Les mesures d'absorption ont déjà été transformées ($\log_{10}$). Nous nous attacherons à modéliser le pourcentage de matière grasse.

Ces données ont été largement étudiées dans la littérature car elles servent de jeu de donnée test (*benchmark*) pour comparer des prétraitements des données (centrage, réduction sur données brutes ou après pca, pls, décomposition sur base de splines et dérivations, réduction de dimension par ACP ou PLS) des méthodes de modélisation (régression linéaire, svm, réseaux de neurones, cart...), des techniques de sélection de variables (stepwise, par critère bayésien, informaiton mutuelle, par algorithme stochastique...) et toutes les combinaisons de ces différentes approches...

Contrairement à la plupart des données NIR du domaine public, ces données présentent une composante non linéaire certes légère mais présente. De plus, la prise en compte des dérivées ou tout du moins des différences d'ordre 2 des spectres est aussi pertinente, d'où l'intérêt qu'elles ont sucité et suscitent toujours dans la littérature du métier, mais aussi statistique ou de *m*achine learning, pour justifier l'utilisation de nouvelles méthodes : non-linéaires et/ou fonctionnelles.

1.2 Données

Les données sont accessibles dans R au sein du package *caret*. Elles sont contenues dans deux matrices : `absorp` de $n = 215$ lignes et $p = 100$ colonnes, les absorptions par longueur d'onde, et `endpoints` de $n = 215$ lignes et $p = 3$ colonnes, les pourcentages d'eau, graisses, et protéines. Les 129 premières observations sont généralement utilisées comme échantillon d'apprentissage.

1.3 Objectif

L'objectif principal est la recherche d'un meilleur modèle de prévision du pourcentage de matière grasse. Il s'agit d'évaluer, comparer, différentes stratégies et méthodes pour aboutir à celle la plus efficace. Le travail est découpé en trois parties. La première explore les spectres, leur lissage et de leur dérivées ; la deuxième modélise les spectres initiaux tandis que la suivante prend en compte les dérivées des spectre après lissage spline avant finalement d'automatiser les traitements pour optimiser les choix.

2 Approche exploratoire

2.1 Prise en charge des données

```
library(caret)
```

1. The data are available in the public domain with no responsability from the original data source. The data can be redistributed as long as this permission note is attached.

```
data(tecator)
# corrélation des variables à expliquer
splom(endpoints)
## tracer un échantillon aléatoire de 10 spectres
# sélection des observations
set.seed(1)
inSubset <- sample(1:dim(endpoints)[1], 10)
absorpSubset <- absorp[inSubset,]
endpointSubset <- endpoints[inSubset, 2]
# tri des spectres selon la première valeure
newOrder <- order(absorpSubset[,1])
absorpSubset <- absorpSubset[newOrder,]
endpointSubset <- endpointSubset[newOrder]
# définition des couleurs
plotColors <- rainbow(10)
# tracé des échelles
plot(absorpSubset[1,],
   type = "n",
   ylim = range(absorpSubset),
   xlim = c(0, 105),
   xlab = "Wavelength Index",
   ylab = "Absorption")
# tracé des spectres et du taux de graisse
for(i in 1:10){
points(absorpSubset[i,],type = "l",
   col = plotColors[i], lwd = 2)
text(105,absorpSubset[i,100], endpointSubset[i],
   col = plotColors[i])
}
title("Predictor Profiles for 10 Random Samples")
gras=endpoints[,2]
```

La moyenne de chaque spectre ne semble pas être une information pertinente pour l'estimation du taux de graisse. En revanche, ils présentent des formes remarquable dans certaines zones, d'où l'intérêt éventuel d'étudier la dérivée de ces courbes.

2.2 Statistiques élémentaires

```
# la variable à expliquer
hist(gras)
# les diagrammes boîtes des variables explicatives
boxplot(data.frame(absorp))
# Une ACP pour voir
acp=prcomp(absorp)
plot(acp)
biplot(acp)
plot(acp$x)
plot.ts(acp$rotation[,1:10])
```

Très grande importance de la première composante principale. Elle a pour forme celle moyenne des courbes dont les décalages génèrent une variance importante.

2.3 Lissage spline et dérivées des spectres

Choix du paramètre de lissage ou degré de liberté qui est la trace de la "*hat matrix*" ou matrice de lissage. Le premier spectre, son lissage et ses dérivées sont tracés pour différentes valeurs du paramètre `df`. Si celui-ci n'est pas fourni (dernier cas), il est optimisé par validation croisée afin de reconstruire au mieux la courbe.

```
library(splines)
par(mfcol=c(3,3))
degre=c(3,10,20)
for (i in 1:3)
        {
model=smooth.spline(x=1:100,y=absorp[1,],df=degre[i])
lisse=predict(model)$y
deriv1=predict(model,deriv=1)$y
deriv2=predict(model,deriv=2)$y
ts.plot(cbind(lisse,absorp[1,]))
ts.plot(deriv1)
ts.plot(deriv2)
}
par(mfcol=c(1,1))
```

```
model=smooth.spline(x=1:100,y=absorp[1,])
lisse=predict(model)$y
deriv1=predict(model,deriv=1)$y
deriv2=predict(model,deriv=2)$y
ts.plot(cbind(lisse,absorp[1,]))
ts.plot(deriv1)
ts.plot(deriv2)
```

Il semblerait raisonnable de lisser un peu plus que ce que suggère la validation croisée afin d'obtenir des dérivées plus lisses. L'idéal serait évidemment d'optimiser ce paramètre en cherchant à prévoir au mieux le pourcentage de matières grasses.

Génération des bases de B-splines cubiques et tracer de quelques unes d'entre elles.

```
base=bs(1:100,df=model$fit$nk)
ts.plot(base[,c(1,20,30,64)])
```

Pour chaque spectre, les commandes ci-dessous calculent un lissage des spectres par B-splines cubiques : noeuds équirépartis, degré fixé à 15. Le programme fournit les dérivées première et seconde aux points de discrétisation, les coefficients dans la base de base de B-Splines.

```
degre=15
# initialisation des matrices
coeff=matrix(ncol=model$fit$nk,nrow=215)
lisse=matrix(ncol=100,nrow=215)
deriv1=matrix(ncol=100,nrow=215)
deriv2=matrix(ncol=100,nrow=215)

# itération sur tous les spectres
for (obs in 1:215){
   model=smooth.spline(x=1:100,y=absorp[obs,],
      df=degre)
        coeff[obs,]=model$fit$coef
        lisse[obs,]=predict(model)$y
        deriv1[obs,]=predict(model,deriv=1)$y
        deriv2[obs,]=predict(model,deriv=2)$y
        }
```

3 Modélisation des spectres

L'étude se limite à une utilisation de la librairie `caret` (Kunh, 2008 [1]) pour quelques unes des méthodes de modélisation proposées.

3.1 Préparation des données

Extraction des échantillons d'apprentissage et de test

```
# Indices usuelles des échantillons
# d'apprentissage et de test
inTrain = 1:129
trainDescr=absorp[inTrain,]
testDescr=absorp[-inTrain,]
trainY=gras[inTrain]
testY=gras[-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'autres part, l'erreur de prévision est estimée par validation croisée.

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

3.2 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.

```
#1 Régression linéaire leapBackward
set.seed(2)
llmFit = train(trainDescr, trainY,
```

```
   method = "leapBackward", tuneLength = 20,
   trControl = cvControl)
llmFit
plot(llmFit)
#2 régression pls
set.seed(2)
plsFit = train(trainDescr, trainY,
   method = "pls", tuneLength = 30,
   trControl = cvControl)
plsFit
plot(plsFit)
#3 Ridge regression
set.seed(2)
ridgeFit = train(trainDescr, trainY,
   method = "ridge", tuneLength = 8,
   trControl = cvControl)
ridgeFit
plot(ridgeFit)
#4 Regression elastic net
set.seed(2)
enetFit = train(trainDescr, trainY,
   method = "enet", tuneLength = 8,
   trControl = cvControl)
enetFit
plot(enetFit)
#5 Support vector machine noyau linéaire
set.seed(2)
svmFit = train(trainDescr, trainY,
   method = "svmLinear", tuneLength = 8,
   trControl = cvControl)
svmFit
plot(svmFit)
#6 Support vector machine noyau gaussien
set.seed(2)
svmgFit = train(trainDescr, trainY,
   method = "svmRadial", tuneLength = 8,
   trControl = cvControl)
svmgFit
plot(svmgFit)
#7 Multivariate Adaptive Regression Spline
set.seed(2)
marsFit = train(trainDescr, trainY,
   method = "earth", tuneLength = 10,
   trControl = cvControl)
marsFit
plot(marsFit)
#8 Kernel-based Regularized Least Squares (KRLS)
set.seed(2)
krlsFit = train(trainDescr, trainY,
   method = "krlsRadial", tuneLength = 10,
   trControl = cvControl)
krlsFit
plot(krlsFit)
```

3.3 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,
   elasticnet=enetFit,svm=svmFit,svmg=svmgFit,
   mars=marsFit,krls=krlsFit)
testPred=predict(models, newdata = testDescr)
lapply(testPred,function(x)sqrt(mean((x-testY)^2)))
resPlot=extractPrediction(models,
   testX=testDescr, testY=testY)
plotObsVsPred(resPlot)
```

remarquer quels osnt les modèle prenant en compte la non linéarité des données, ceux acceptant des valeurs tests relativements atypiques.

4 Utilisation des dérivées

4.1 Extraction des échantillons

Extraction des échantillons d'apprentissage et de test

```
# Mêmes indices de l'échantillon d'apprentissage
inTrain = 1:129
trainDescr1=deriv1[inTrain, ]
testDescr1 =deriv1[-inTrain,]
# pré-process
xTrans1=preProcess(trainDescr1)
trainDescr1=predict(xTrans1,trainDescr1)
testDescr1 =predict(xTrans1,testDescr1 )
```

Le calcul est restreint au 3 meilleures méthodes trouvées précédemment.

```
# Régression PLS
set.seed(2)
plsFit1 = train(trainDescr1, trainY,
   method = "pls", tuneLength = 30,
   trControl = cvControl)
plsFit1
plot(plsFit1)
# Multivariate Adaptive Regression Spline
set.seed(2)
marsFit1 = train(trainDescr1, trainY,
   method = "earth", tuneLength = 10,
   trControl = cvControl)
marsFit1
plot(marsFit1)
# Kernel-based Regularized Least Squares (KRLS)
set.seed(2)
krlsFit1 = train(trainDescr1, trainY,
   method = "krlsRadial", tuneLength = 10,
   trControl = cvControl)
krlsFit1
plot(krlsFit1)
# Caclul des erreurs et tracé des graphes
models=list(pls1=plsFit1,mars1=marsFit1,
   krls1=krlsFit1)
testPred1=predict(models, newdata = testDescr1)
lapply(testPred1,function(x)sqrt(mean((x-testY)^2)))
resPlot=extractPrediction(models,
   testX=testDescr1, testY=testY)
plotObsVsPred(resPlot)
```

5 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.

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.

Exécuter le programme en annexe en fixant par exemple :

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

Puis calculer :

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

Modifier le programme pour faire le calcul en utilisant les dérivées des spectres, commenter, conclure...

Références

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

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
library(caret)
# Gestion des données
data(tecator)
Yvar=endpoints[,2]
# Xvar=deriv1 # calcul sur dérivées
 Xvar=absorp
methodes= c("pls","reg-sel","svm","krls")
# 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(Xvar[,1],p=60/100,
   list = FALSE)
# Extraction des échantillons
trainDescr=Xvar[inTrain,]
testDescr=Xvar[-inTrain,]
trainY=Yvar[inTrain]
testY=Yvar[-inTrain]
# 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 pls
plsFit = train(trainDescr, trainY,
   method = "pls", tuneLength = 30,
   trControl = cvControl)
# régression leap backward
regselFit = train(trainDescr, trainY,
   method = "leapBackward", tuneLength = 30,
   trControl = cvControl)
# support vector machine
svmFit = train(trainDescr, trainY,
   method = "svmRadial", tuneLength = 20,
   trControl = cvControl)
# Kernel-based Regularized Least Squares
krlsFit = train(trainDescr, trainY,
   method = "krlsRadial", tuneLength = 20,
   trControl = cvControl)
####
# Calcul des erreurs
models=list(pls=plsFit,regsel=regselFit,svm=svmFit,
   krls=krlsFit)
testPred=predict(models, newdata = testDescr)
res.reg[i,]=do.call(c,lapply(testPred, function(x)
   sqrt(mean((x-testY)^2))))
}
```