# Scénario: *Text mining*, filrage d'un corpus de courriels

## Résumé

*Après l'exploration et la classification des contenus d'une base de message afin de catégoriser les pourriels, il s'agit de contruire des* scores *ou modèles prévoyant la nature d'un message : spam ou non, en fonction de son contenu ou plutôt de la présence ou fréquence de certains mots et caractères. Les principales méthodes de modélisation et apprentissage statistique sont testées et comparées.*

## 1 Introduction

### 1.1 Objectif

Cette étude est un exemple d'analyse textuelle d'un corpus de documents, ici des courriels. Une telle analyse est classiquement basée sur la fréquence d'une sélection de mots. Après l'analyse exploratoire, l'objectif général est de pouvoir discriminer les courriels pertinents ou encore de définir un modèle personnalisé de "détection des pourriels" (spams), c'est-à-dire adapté au contenu spécifique de la boîte d'un internaute. Il s'agit donc d'un modèle susceptible de prévoir la "qualité" d'un message reçu en fonction de son contenu. Le déroulement de cette étude est évidemment marqué par le type particulier des données mais celle-ci peut facilement se transposer à d'autres types de données textuelles ou analyse du contenu de livres, pages web, discours politiques, réponses ouvertes à des questionnaires... les exemples sont nombreux en sciences humaines, marketing lorsqu'il s'agit d'estimer / prévoir des scores, par exemple, de satisfaction de clientèle. L'objectif du présent travail est d'évaluer, comparer différentes stratégies et méthodes pour aboutir à celle la plus efficace de filtrage des courriers indésirables. Il s'agit de répondre aux questions suivantes :

- Comment estimer les erreurs de prévisions pour optimiser les modèles ?
- Quelle méthode de modélisation parmi celles concurrentes : logit, arbre, réseau de neurones, random forest, boosting ?
- Quelle meilleure stratégie adaptative adoptée ?

Pour des raisons évidentes de temps, toutes les solutions ne seront pas testées.

### 1.2 Données

George, ingénieur chez HP dans le département *Computer Science* a recueilli un échantillon de messages électroniques dans chacun desquels il a évalué le nombre d'occurrences d'une sélection de mots et caractères. Les variables considérées sont, dans une première partie, des rapports : nombre d'occurrences d'un mot spécifique sur le nombre total de mots ou nombre d'occurrences d'un caractère sur le nombre de caractères du message. L'approche exploratoire a nécessité de recoder en classes ces variables, le plus souvent de simples classes présence : absence des mots ou caractères cocernés. Les variables d'occurrences sont décrites dans le tableau 1, celles associées à la casse dans le tableau 2. Ces données sont publiques, elles servent régulièrement de *b*enchmark pour la comparaison de méthodes d'apprentissage machine :

Frank, A. ; Asuncion, A. (2010). UCI Machine Learning Repository. Irvine, CA : University of California, School of Information and Computer Science. `http://archive.ics.uci.edu/ml`

Le tableau 1 liste 54 variables exprimant soit :

- le rapport du nombre d'occurrence d'un mot (resp. de caractères) sur le nombre total de mots (de caractères) dans un message,
- soit la présence ou non de ce mot (resp. caractère) dans le message,
- des numéros (85...) qui sont ceux de bureau, téléphone, code postal de George.

La tableau 2 liste 4 variables dont celle dénombrant le bombre de lettres majuscules.

## 2 Travail à réaliser

Le déroulement de cette étude va nécessité de nombreuses étapes afin de mettre en oeuvre les comparaisons recherchées.

1. lecture des données et extraction des échantillons,
2. comparaison de différentes stratégie utilisant la régression logistique : échantillon de validation ou validation croisée, variables codées en classe ou variables brutes réelles ?

TABLE 1 – *Les colonnes contiennent successivement le libellé de la variable, le mot ou ensemble de caractères concernés, le libellé des modalités Présence / Absence utilisées après recodage.*

| Variable | Mot ou Carac. | Modalités P/A | Variable | Mot ou Carac. | Modalités |
|---|---|---|---|---|---|
| make | make | make / Nmk | X650 | 650 | 650 / N65 |
| address | address | addr / Nad | lab | lab | lab / Nlb |
| all | all | all / Nal | labs | labs | labs / Nls |
| X3d | 3d | 3d / N3d | telnet | telnet | teln / Ntl |
| our | our | our / Nou | X857 | 857 | 857 / N87 |
| over | over | over / Nov | data | data | data / Nda |
| remove | remove | remo / Nrm | X415 | 415 | 415 / N41 |
| internet | internet | inte / Nin | X85 | 85 | 85 / N85 |
| order | order | orde / Nor | technology | technology | tech / Ntc |
| mail | mail | mail / Nma | X1999 | 1999 | 1999/ N19 |
| receive | receive | rece / Nrc | parts | parts | part / Npr |
| will | will | will / Nwi | pm | pm | pm / Npm |
| people | people | peop / Npp | direct | direct | dire / Ndr |
| report | report | repo / Nrp | cs | cs | cs / Ncs |
| addresses | addresses | adds / Nas | meeting | meeting | meet/Nmt |
| free | free | free / Nfr | original | original | orig / or |
| business | business | busi / Nbs | project | project | proj / Npj |
| email | email | emai / Nem | re | re | re / Nre |
| you | you | you / Nyo | edu | edu | edu / Ned |
| credit | credit | cred / Ncr | table | table | tabl / Ntb |
| your | your | your / Nyr | conference | conferenc | e conf / Ncf |
| font | order | font / Nft | CsemiCol | ; | Cscl / NCs |
| X000 | 000 | 000 / N00 | Cpar | ( | Cpar / NCp |
| money | money | mone/ Nmn | Ccroch | [ | Ccro / NCc |
| hp | hp | hp / Nhp | Cexclam | ! | Cexc / NCe |
| hpl | hpl | hpl / Nhl | Cdollar | $ | Cdol / NCd |
| george | george | geor / Nge | Cdiese | # | Cdie / NCi |

TABLE 2 – *Liste de 4 variables, de leur libellé et des modalités après recodage.*

| Code variable | Libellé | Modalités |
|---|---|---|
| Spam | Type de message pourriel ou non | Spam / Nsp |
| CapLM | Nombre moyen de capitales par mot | Mm1 / Mm2 / Mm3 |
| CapLsup | Nombre max de capitales par mot | Ms1 / Ms2 / Ms3 |
| CapLtot | Nombre totale de lettres capitales | Mt1 / Mt2 / Mt3 |

3. Ces choix étant opérés, les modèles issus des différentes méthodes sont ensuite optimisés :
4. analyse discriminante ou knn, arbre de classification, réseau de neurones, random forest, boosting.
5. La procédure d'extraction aléatoire d'un échantillon test est itérée de façon à optenir plusieurs estimations des erreurs de prévision. Comparaison des disctributions des erreurs et tracé des courbes ROC.
6. Synthèse et conclusion.

# 3 Gestion des données

Les données sont disponibles sous deux formes dans le répertoire "data" : fréquences quantitatives dans le fichier `spam.dat` et recodées en classes dans le fichier `spamq.dat`. Les lire avec les commandes suivantes :

```
spam.quali=read.table("spamq.dat")
summary(spam.quali)
spamr=read.table("spam.dat")
summary(spamr)
```

Extrations des échantillons apprentissage, validation et tests dans le cas qualitatif.

```
# Dans le cas qualitatif :
# Spam.appt1, validatin croisée et  Spam.test
# Spam.appt2, échantillon Spam.valid et Spam.test
# choisir une valeur pour``xx''
set.seed(xx)
npop=nrow(spam.quali)
test=sample(1:npop,1000)
Spam.test=spam.quali[test,]
appt1=setdiff(1:npop,test)
Spam.appt1=spam.quali[appt1,]
npop=nrow(Spam.appt1)
valid=sample(1:npop,1000)
Spam.valid=Spam.appt1[valid,]
appt2=setdiff(1:npop,valid)
```

```
Spam.appt2=Spam.appt1[appt2,]
```

```
# spamr.appt et spamr.test dans le cas quantitatif
# choisir une valeur pour ''xx''
set.seed(11)
npop=nrow(spamr)
test=sample(1:npop,1000)
spamr.test=spamr[test,]
appt=setdiff(1:npop,test)
spamr.appt=spamr[appt,]
```

# 4 Régression logistique

## 4.1 Choix de la stratégie d'apprentissage

Le volume suffisant des données a permis d'extraire deux sous échantillons, un de validation pour optimiser les modèles et un de test pour les comparer. On s'interroge de savoir si cette atratégie prévaut sur celle adaptée à de plus petits échantillons pour lesquels l'erreur de prévisions est estimée par validation croisée pour être minimisée. Les deux stratégies sont comparées dans le cas de la régression logistique et pour deux sélection de modèles basés sur une pénalisation AIC (backward et stepwise).

### *Validation croisée*

```
spam1.logit1=glm(spamf~.,data=Spam.appt1,
   family=binomial)
spam1.log1=step(spam1.logit1,direction='backward')

spam1.logit2=glm(spamf~1,data=Spam.appt1,
   family=binomial)
spam1.log2=step(spam1.logit2,direction='both',
   scope=list(lower=~1,upper=~make + address +
   all + X3d + our + over + remove + internet +
   order + mail + receive + will + people +
   report + addresses + free + business +
   email + you + credit + your + font + X000 +
   money + hp + hpl + george + X650 + lab +
   labs + telnet +  X857 + data + X415 + X85 +
   technology + X1999 + parts + pm + direct +
   cs + meeting + original + project + re +
   edu + table + conference + CsemiCol +
   Cpar + Ccroch + Cexclam + Cdollar +
   Cdiese + CapLMq + CapLsupq + CapLtotq))
```

Comparer les performances des deux modèles :

```
library(boot)
cv.glm(Spam.appt1,spam1.log1,K=10)$delta[1]
cv.glm(Spam.appt1,spam1.log2,K=10)$delta[1]
```

Retenir le meilleur modèle disons `spam1.log2` mais cela dépend de l'échantillon d'apprentissage tiré. La commande

```
anova(spam2.log1,test="Chisq")
```

permet d'identifier la variable la moins significative du modèle. Retirer celle-ci et, après ré-estimation du modèle, vérifier si l'erreur de prévision sur l'échantillon de validation est améliorée ; itérer ou non.

### *Echantillon de validation*

```
spam2.logit1=glm(spamf~.,data=Spam.appt2,
   family=binomial)
spam2.log1=step(spam2.logit1,
   direction='backward')

spam2.logit2=glm(spamf~1,data=Spam.appt2,
   family=binomial)
spam2.log2=step(spam2.logit2,direction='both',
   scope=list(lower=~1,
upper=~make + address + all + X3d + our + over +
  remove + internet + order + mail + receive +
  will + people + report + addresses + free +
  business + email + you + credit + your + font +
  X000 + money + hp + hpl + george + X650 + lab +
```

```
  labs + telnet +  X857 + data + X415 + X85 +
  technology + X1999 + parts + pm + direct + cs +
  meeting + original + project + re + edu +
  table + conference + CsemiCol + Cpar + Ccroch +
  Cexclam + Cdollar + Cdiese + CapLMq +
  CapLsupq + CapLtotq))
```

```
valid.log1=predict(spam2.log1,newdata=Spam.valid)
table(valid.log1>0.5,Spam.valid[,"spamf"])

valid.log2=predict(spam2.log2,newdata=Spam.valid)
table(valid.log1>0.5,Spam.valid[,"spamf"])
```

Retenir le meilleur modèle disons `spam2.log2` mais cela dépend de l'échantillon d'apprentissage tiré. La commande

```
anova(spam2.log1,test="Chisq")
```

permet d'identifier la variable la moins significative du modèle. Retirer celle-ci et, après ré-estimation du modèle, vérifier si l'erreur de prévision sur l'échantillon de validation est améliorée ; itérer ou non.

### *Echantillon test*

Il s'agit de comparer alors les deux meilleurs modèles : celui optimisé par validation croisée (noté `spam1.log21` et celui optimisé par échantillon de validation noté `spam2.log21` mais, attention, pour être "équitable", il faut ré-estimer le modèle 2 sur l'échantillon complet afin qu'il bénéficie du même échantillon (de la même taille) que le modèle 1.

```
test1.log=predict(spam1.log21,newdata=Spam.test)
table(test1.log>0.5,Spam.test[,"spamf"])

spam2.log21=glm(spamf ~ liste des variables,
        data=Spam.appt1,family=binomial)
test21.log=predict(spam2.log21,newdata=Spam.test)
table(test21.log>0.5,Spam.test[,"spamf"]) % 0.075
```

Comparer les résultats, quelle est le "meilleure" stratégie d'apprentissage.

La comparaison de ces deux stratégies sur les autres méthodes conduit aux mêmes conclusions. Elles ne sont pas reproduites ici et le choix est fixé pour la suite sur l'utilisation de la validation croisée.

## 4.2 Données recodées qualitatives ou réelles

```
spamr.logit1=glm(spam~1,data=spamr.appt,
   family=binomial)
spamr.log1=step(spamr.logit1,
   direction='backward')

spamr.logit2=glm(spam~1,data=spamr.appt,
   family=binomial)
spamr.log2=step(spamr.logit,direction='both',
   scope=list(lower=~1,upper=~make + address +
   all + X3d + our + over + remove + internet +
    order + mail + receive + will + people +
    report + addresses + free + business +
    email + you + credit + your + font + X000 +
    money + hp + hpl + george + X650 + lab +
    labs + telnet + X857 + data + X415 + X85 +
    technology + X1999 + parts + pm + direct +
    cs + meeting + original + project + re +
    edu + table + conference + CsemiCol +
    Cpar + Ccroch + Cexclam + Cdollar +
    Cdiese + CapLM + CapLsup + CapLtot))
```

Comparer les modèles.

```
library(boot)
cv.glm(spamr.appt,spamr.log1,K=10)$delta[1]
cv.glm(spamr.appt,spamr.log2,K=10)$delta[1]
```

Optimiser éventuellement le choix opéré en tâchant de réduire l'erreur par validaiton croisée par le retrait des variables les moins significatives du modèle.

Appliquer le meilleur modèle obtenu à l'achantillon test.

```
pred.log=predict(spamr.log3,newdata=spamr.test)
table(pred.log>0.5,spamr.test[,"spam"]) # 0.065
```

Conclusion.

Question : pourquoi le choix des variables brutes s'impose-t-il pour les autres méthodes ?

# 5 Comparaison systématique des méthodes

La suite du travail se propose d'utiliser la librairie `caret`(Kuhn, 2008)[1] afin de faciliter et "industrialiser" les traitements. Attention, cette librairie fait appel à de nombreuses autres librairies qui doivent être installées à la demande. La version "artisanale" des commandes, sans l'utilisation de `caret` est proposée en annexe.

## 5.1 Préparation des données

Les données sont celles brutes c'est-à-dire quantitatives et la stratégie adoptée pour optimiser les modèles est la validation croisée. D'autres choix sont possibles (bootstrap). La librairie `caret` intègre des fonctions d'échantillonnage et de normalisation des données.

```
library(caret)
# extraction des données
# remplacer les niveaux "0" et "1"
# par "nospam" et "spam"
Y=rep("spam",nrow(spamr))
Y[spamr[,58]==0]="nospam"
Y=as.factor(Y)
summary(Y)
X=spamr[,-58]
summary(X)
# indices de l'échantillon d'apprentissage
set.seed(xx)  # Replacer xx par un entier
              # pour initialiser
inTrain = createDataPartition(X[,1],
   p = 78/100, list = FALSE)
# Extraction des échantillons
trainDescr=X[inTrain,]
testDescr=X[-inTrain,]
trainY=Y[inTrain]
testY=Y[-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.

```
# Normalisation
xTrans=preProcess(trainDescr)
trainDescr=predict(xTrans,trainDescr)
testDescr=predict(xTrans,testDescr)
# Choix de la validation croisée
cvControl=trainControl(method="cv",number=10)
```

Il peut arriver sur certains jeux de données qu'après partitionnement de l'échantillon par validation croisée ou bootstrap, certaines variables deviennent strictement constantes sur une classe. Ceci pose des problèmes à certaines méthodes comme l'analyse discriminante ou les classifieurs bayésien naïf car la variance est nulle pour ces variables au sein d'une classe.

La librairie `caret` inclut une fonction permettant d'itentifier ces variables en vue de leur suppression.

C onsulter la documentation ?nearZeroVar nearZeroVar(spamr, saveMetrics= TRUE) Le problème de ces données textuelles très creuses : beaucoup de mots ne sont pas présents dans la plupart des messages, est que la grande majorité des variables est considérée comme *near zero variance*. Le seuil de décision peut être remonté mais ela n'élimine alors plus le risque de vraince constante !

## 5.2 Estimation et optimisation des modèles

La librairie intègre beaucoup plus de méthodes et celles sélectionnées ci-dessous semblent les plus pertinentes. Certaines méthodes ne sont pas utilisables à cause du nombre de variables. Il pourrait être utile et intéressant d'en tester encore d'autres. Exécuter successivement les blocs de commandes pour tracer séparamment chacun des graphes afin de contrôler le bon comportement de l'optimisation du paramètre de complexité de chaque modèle.

Consulter la liste des méthodes disponibles dans l'aide de la fonction : `?train` et tester les principales. L'utilisation de certaines comme la

régression logistique est sans doute moins flexible qu'en utilisation plus "manuelle" en particulier dans le choix de l'algorithme de sélection de variables. Il faut se montrer (très) patient pour certaines optimisations alors que d'autres sont immédiates, voire inutiles. Le paramètre `tuneLength` caractérise un "effort" d'optimisation, c'est en gros le nombre de valeurs de paramètres testées sur une grille fixée automatiquement. En prenant plus de soin et aussi plus de temps, il est possible de fixer précisément des grilles pour les valeurs du ou des paramètres optimisés pour chaque méthode.

```
#1 Régression logistique
set.seed(2)
rlogFit = train(trainDescr, trainY,
   method = "glmStepAIC", tuneLength = 10,
   trControl = cvControl)
rlogFit
#2 analyse discriminante linéaire
set.seed(2)
ldaFit = train(trainDescr, trainY,
   method = "lda2", tuneLength = 10,
   trControl = cvControl)
ldaFit
#3 K plus proches voisins
set.seed(2)
knnFit = train(trainDescr, trainY,
   method = "knn", tuneLength = 10,
   trControl = cvControl)
knnFit
plot(knnFit)
#4 Arbre de décision
set.seed(2)
rpartFit = train(trainDescr, trainY,
   method = "rpart", tuneLength = 10,
   trControl = cvControl)
rpartFit
plot(rpartFit)
#5 Réseau de neurones
set.seed(2)
nnetFit = train(trainDescr, trainY,
   method = "nnet", tuneLength = 6,
   trControl = cvControl)
nnetFit
plot(nnetFit)
#6 Random forest
set.seed(2)
rfFit = train(trainDescr, trainY,
   method = "rf", tuneLength = 10,
   trControl = cvControl)
rfFit
plot(rfFit)
#7 Boosting d'arbres
set.seed(2)
gbmFit = train(trainDescr, trainY,
   method = "gbm", tuneLength = 8,
   trControl = cvControl)
gbmFit
plot(gbmFit)
#8 Support vector machine avec noyau linéaire
set.seed(2)
svmFit = train(trainDescr, trainY,
   method = "svmLinear", tuneLength = 8,
   trControl = cvControl)
svmFit
plot(svmFit)
#9 Support vector machine avec noyau gaussien
set.seed(2)
svmgFit = train(trainDescr, trainY,
    method = "svmRadial", tuneLength = 8,
   trControl = cvControl)
svmgFit
plot(svmgFit)
# 10 Classifieur bayésien naïf
# Quel problème pose ce "classifieur" ?
set.seed(2)
```

```
nbFit = train(trainDescr, trainY,
    method = "nb", tuneLength = 8,
   trControl = cvControl)
nbFit
plot(nbFit)
```

## 5.3 Prévision et erreur 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(logit=rlogFit,add=ldaFit,knn=knnFit,
  cart=rpartFit,nnet=nnetFit,rf=rfFit,gbm=gbmFit,
  svm=svmFit,svmg=svmgFit)
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 des différentes méthodes. L'analyse discriminante qui ne fournit pas de probabilités d'occurence de la classe "spam" n'est pas adaptée au tracé d'une courbe ROC. Comme elle fait par ailleurs partie des moins performantes sur l'échantillon test, elles est retirée de la liste des modèles.

```
# Courbes ROC
library(ROCR)
models=list(logit=rlogFit,knn=knnFit,
  cart=rpartFit,nnet=nnetFit,rf=rfFit,gbm=gbmFit,
  svm=svmFit,svmg=svmgFit)
testProb=predict(models, newdata = testDescr,
   type="prob")
predroc=lapply(testProb,
   function(x)prediction(x[,1],testY=="nospam"))
perfroc=lapply(predroc,
   function(x)performance(x, "tpr", "fpr"))
plot(perfroc$logit,col=1)
plot(perfroc$knn,col=2,add=TRUE)
plot(perfroc$cart,col=3,add=TRUE)
plot(perfroc$nnet,col=4,add=TRUE)
plot(perfroc$rf,col=5,add=TRUE)
plot(perfroc$gbm,col=6,add=TRUE)
plot(perfroc$svm,col=7,add=TRUE)
plot(perfroc$svmg,col=8,add=TRUE)
legend("bottomright",legend=c("logit","knn",
   "CART","nnet","RF","boost","svmL",
   "svmG"),col=c(1:8),pch="_")
```

## 5.4 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)
```

## 5.5 Automatisation

L'échantillon est de talle raisonnable, mais 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. Attention au temps de calcul !

```
# Choisir la liste des méthodes
# l'effort d'optimisation
# Initialiser le générateur et
# fixer le nombre d'itérations
models=c("gbm","svmRadial","rf","nnet",
   "glmStepAIC")
noptim=c(6,6,6,6,6)
Niter=xxx ; Init=xx
pred.spam=pred.autom(X,Y,methodes=models,
   N=Niter,xinit=Init,size=noptim,type="prob")
```

Puis calculer et représenter les erreurs pour les méthodes considérées.

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

Les commandes suivandes concerent les courbes ROC.

```
# Apprécier la faible dispersion des
# courbes ROC associées aux forêts aléatoires
plot(perfroc.spam$rf,col=2)
plot(perfroc.spam$rf,col=1,add=TRUE,
   lwd=2,avg="vertical")
# Comparaison des méthodes par le
# tracer des courbes ROC moyennes
# Problème pas identifié avec rlogit !
predroc.spam=lapply(prev.spam,
   function(x)prediction(x,obs==1))
perfroc.spam=lapply(predroc.spam,
   function(x)performance(x,"tpr","fpr"))
plot(perfroc.spam$gbm,col=1,lwd=2,
   avg="vertical")
plot(perfroc.spam$rf,col=2,add=TRUE,
   lwd=2,avg="vertical")
plot(perfroc.spam$svmRadial,col=3,add=TRUE,
   lwd=2,avg="vertical")
plot(perfroc.spam$nnet,add=TRUE,col=4,lwd=1.5,
   avg="vertical")
%plot(perfroc.spam$rlogit,add=TRUE,col=5,lwd=1.5,
%   avg="vertical")
legend("bottomright",legend=c("boost","RF",
   "svmG", "nnet"),col=c(1:4),pch="_")
```

Que suggérer à Georges pour améliorer son détecteur de pourriel ? Comment éviter que vos messages ne soient "mangés" par les anti-spams ?

La plupart des détecteurs de spams sont des classifieurs bayésiens "améliorés". Consulter le site Wikipedia à ce sujet. Quelle est la raison principale de ce choix de classifieur ? Quel est le problème rencontré ici, comment est-il résolu par ces détecteurs.

## Références

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

## Annexe

### $N$ réplications des estimations / prévisions

```
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
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))
obs=matrix(0,ntest,N)
set.seed(xinit)
for(i in 1:N) {  # N itérations
# 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
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
}
```

## Traitements "artisanaux"

Les commandes suivantes permettent de lancer les estimations et optimisations sans faire appel à la librairie `caret`. Une pratique manuelle des optimisations conduirait-elle à de meilleurs résultats ?

### *Arbre de classification*

```
tree=rpart(spam~.,data=spamr.appt,
    parms=list(split='information'),cp=0)
plot(tree)
text(tree)
xmat = xpred.rpart(tree,xval=10)
# Comparaison des valeurs prédite et observée
xerr=(spamr.appt$spam=="1") != (xmat>1.5)
# Calcul et affichage des estimations
# des taux d'erreur
apply(xerr, 2, sum)/nrow(xerr) #
spamtree=rpart(spam~.,data=spamr.appt,
    parms=list(split='information'),cp=0.0008570)
```

```
pred.tree=predict(spamtree,newdata=spamr.test,
    type="class")
table(pred.tree,spamr.test[,"spam"]) #
pred.tree=predict(spamtree,newdata=spamr.test,
    type="prob")[,2]
```

### *Réseau de neurones*

Définir la fonction utilisée pour estimer l'erreur par validation croisée.

```
CVnn=function(formula, data, size, niter = 1,
   nplis = 10, decay = 0, maxit = 100)
{
   n = nrow(data)
   tmc=0
   un = rep(1, n)
   ri = sample(nplis, n, replace = T)
   cat(" k= ")
   for(i in sort(unique(ri))) {
     cat(" ", i, sep = "")
     for(rep in 1:niter) {
       learn = nnet(formula, data[ri != i,  ],
         size = size,trace = F, decay = decay,
         maxit = maxit)
       tmc = tmc + sum(un[(data$spam[ri == i]
        == "1") != (predict(learn, data[ri == i,
         ]) > 0.5)])
       }
   }
   cat("\n", "Taux de mal classes")
   tmc/(niter * length(unique(ri)) * n)
}
```

Estimation du modèle avec un nombre grand de neurones. Selon l'installation, la taille maximale est limitée.

```
net=nnet(spam~.,data=spamr.appt,size=15)
summary(net)
CVnn(spam~.,data=spamr.appt,size=15,decay=0,
    maxit=500)
```

Optimiser le paramètre "decay" de pénalisation avant d'estimer l'erreur sur l'échantillon test.

```
net=nnet(spam~.,data=spamr.appt,size=15,decay=1,
    maxit=500)
pred.rn=predict(net,spamr.test) > 0.5
t=table(pred.rn,spamr.test[,"spam"])
```

### *Forêtes aléatoires*

Estimer le modèle avec des valeurs de paramètre (nombre d'arbres) relativement élevées et optimiser le paramètre le plus sensible : le nombre de variables tirées à chaque recherche de division optimale pour un nœud.

Application de la valeur "optimale" c'est-à-dire minimisant l'erreur out-of-bag à l'échantillon test et interprétation des variables les plus importantes.

```
library(randomForest)
rf=randomForest(spam~.,data=spamr.appt,
   xtest=spamr.test[,-58],ytest=spamr.test[,"spam"],
   ntree=500,do.trace=50, mtry=10,importance=TRUE)
sort(round(importance(rf1), 2)[,4])
# échantillon test
pred.rf=rf$test$predicted
table(pred.rf,spamr.test[,"spam"])
```

### *Boosting*

Même démarche pour le boosting avec optimisation du paramètre de shrinkage.

```
boost=gbm(as.numeric(spamr.appt$spam)-1~.,
   data=spamr.appt,distribution="adaboost",
   n.trees=500, cv.folds=10,n.minobsinnode = 5,
   shrinkage=0.1,verbose=FALSE)
plot(boost$cv.error)
# nombre optimal d'itérations
best.iter=gbm.perf(boost,method="cv")
```

```
best.iter # 260
pred.boost=predict(boost,newdata=spamr.test,
   n.trees=best.iter)
table(as.factor(sign(pred.boost)),spamr.test$spam)
mean(sign(pred.boost) !=
   (2*as.numeric(spamr.test$spam)-3))
```