Data Mining - TP classification

Julien JACQUES

06/03/2019

Données MNIST

Chargeons les images

```
source("mnist.R")
```

On extrait 5000 images d'apprentissage et 1000 de test

```
index=1:6000
app_x=train$x[index[1:5000],]
app_y=train$y[index[1:5000]]
test_x=train$x[index[5001:6000],]
test_y=train$y[index[5001:6000]]
```

K-plus proches voisins

On réalise un kNN avec k=10

```
library(class)
res=knn(app_x,test_x,app_y,k=10)
```

Comparons les prédictions obtenues pour l'échantillon test aux vraies étiquettes

```
table(res,test_y)
```

```
##    test_y
## res   0   1   2   3   4   5   6   7   8   9
##   0 110   0   2   1   0   0   0   0   1   1
##   1   0 106   6   2   4   3   0   4   2   0
##   2   0   0  79   0   0   0   0   0   1   0
##   3   0   0   3 107   0   0   0   0   2   2
##   4   1   0   0   0  80   1   1   0   0   3
##   5   1   0   0   3   0  74   1   0   1   0
##   6   1   0   1   0   1   0 105   0   3   0
##   7   0   1   2   1   0   0   0  90   1   2
##   8   0   0   0   0   0   0   0   0  78   0
##   9   0   1   0   1   3   2   0   7   0  98
```

```
cat('taux de bons classements :',mean(res==test_y))
```

```
## taux de bons classements : 0.927
```

arbre de décision

Effectuons maintenant la prédiction à l'aide d'un arbre de décision. Pour cela nous commençons par charger les packages nécessaires,

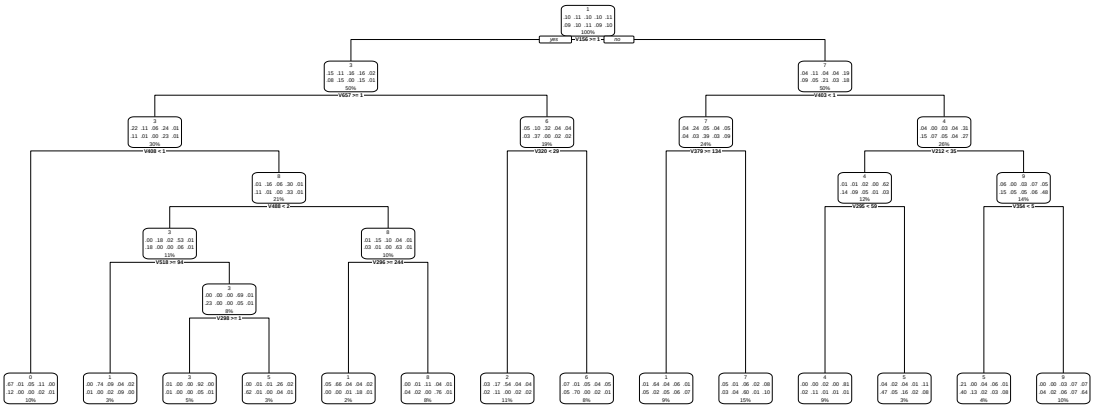
```
library('rpart')
library('rpart.plot')
```

puis à créer deux data.frame, contenant les données d'apprentissage et de test, en renomant la variable contenant les étiquettes afin de bien l'identifier, et en la définissant comme un facteur

```
app=data.frame(cbind(app_x,app_y))
names(app)[785]="y"
app$y=as.factor(app$y)
test=data.frame(cbind(test_x,test_y))
names(test)[785]="y"
test$y=as.factor(test$y)
```

On peut maintenant estimer l'arbre de décision

```
arbre=rpart(y~.,data=app)
rpart.plot(arbre)
```



On constate que l'arbre est très petit, très peu de pixels sont utilisés

Effectuons maintenant la prédiction sur l'echantillon test, et comparons la partition estimée à la partition réelle

```
res=predict(arbre,test,type="class")
table(res,test$y)
```

```
##
## res   0   1   2   3   4   5   6   7   8   9
##   0  80   1   6  19   0   6   0   0   0   2
##   1   0  83   6   8   2   2   1   4  12   5
##   2   3  19  43   6   0   1   9   0   3   3
##   3   2   1   3  47   0   1   0   0   6   0
##   4   0   0   1   0  52   3   9   0   0   3
##   5  14   0   2  14  16  53   7   9   2   3
##   6   2   1  10   6   5   5  73   0   4   0
##   7   9   1   5   4   8   1   7  77   3  25
##   8   2   0  13   2   1   1   1   0  49   0
##   9   1   2   4   9   4   7   0  11  10  65
```

```
cat('taux de bons classements :',mean(res==test_y))
```

```
## taux de bons classements : 0.622
```

Par contre cela ne fonctionne pas très bien, mais il est connu qu'un seul arbre a tendance a sur-apprendre l'échantillon d'apprentissage et à ne pas être très bon en prédiciton.

Forêts aléatoires

On va désormais réaliser la prédiction avec une forêt aléatoire de 200 arbres, effectuer la prédiction sur l'échantillon test et comparer les partitions réelles et estimées.

```
library(randomForest)
model <- randomForest(y~.,data=app,ntree=200)
pred<-predict(model,newdata=test,type="class")
table(pred,test$y)
```

```
##
## pred   0   1   2   3   4   5   6   7   8   9
##    0 109   0   2   1   0   0   0   0   1   1
##    1   0 105   1   2   1   0   0   0   0   0
##    2   0   1  83   0   0   0   0   1   0   0
##    3   0   0   2 107   0   1   0   0   2   2
##    4   1   0   1   0  84   2   0   0   0   5
##    5   0   0   0   3   0  74   2   0   1   0
##    6   1   0   1   0   1   1 104   0   2   0
##    7   0   1   1   0   0   1   0  93   1   2
##    8   2   0   2   2   0   0   1   0  81   1
##    9   0   1   0   0   2   1   0   7   1  95
```

```
cat('taux de bons classements :',mean(pred==test_y))
```

```
## taux de bons classements : 0.935
```

Les forêts aléatoires fonctionnent très bien.

Regression logistique multinomial

La régression logistique est très longue... Commençons par supprimer les variables inutiles (constantes=0 sur toutes les images)

```
app_x_reduit=app_x[,which(! colSums(app_x)==0)]
test_x_reduit=test_x[,which(! colSums(app_x)==0)]
```

On a déjà bien réduit la dimension mais pas sur que cela suffise. Cherchons les variables qui (en univariée) ne serait pas différentes suivants les classes

```
alpha=0.05/ncol(app_x) # seuil de significativité avec correction de Bonferroni
var_signif=NULL
for (j in 1:ncol(app_x_reduit)){
  if ((anova(aov(app_x_reduit[,j]~app_y))$`Pr(>F)`[1])<alpha) var_signif=c(var_signif,j)
}
```

```
app_x_reduit=app_x_reduit[,var_signif]
test_x_reduit=test_x_reduit[,var_signif]
```

```
app_reduit=data.frame(cbind(app_x_reduit,app_y))
names(app_reduit)[ncol(app_reduit)]="y"
app_reduit$y=as.factor(app_reduit$y)
test_reduit=data.frame(cbind(test_x_reduit,test_y))
names(test_reduit)[ncol(test_reduit)]="y"
test_reduit$y=as.factor(test_reduit$y)
```

Mais même là encore c'est très long...

```
library(VGAM)
reglog=vglm(y~.,data=app_reduit,family=multinomial)
```

Par contre une régression logistique pénalisée (LASSO) tourne un peu plus rapidement

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-1
```

```
fit=glmnet(app_x_reduit,app_y,alpha=1,family="multinomial",lambda=0.1)
p <- predict(fit, newx = test_x_reduit,type='class')
table(p,test_y)
```

```
##    test_y
## p     0   1   2   3   4   5   6   7   8   9
##   0  73   0   3   0   3   0  14   5  10   5
##   1   5 107  22 104  17  41  13  10  42  17
##   4  30   0  28   3  52  11  18   1   2  25
##   6   1   0  27   0   0   4  55   1  12   1
##   7   4   1  13   8  16  24   7  84  23  58
```

```
cat('taux de bons classements :',sum(p==test_y)/length(p))
```

```
## taux de bons classements : 0.371
```

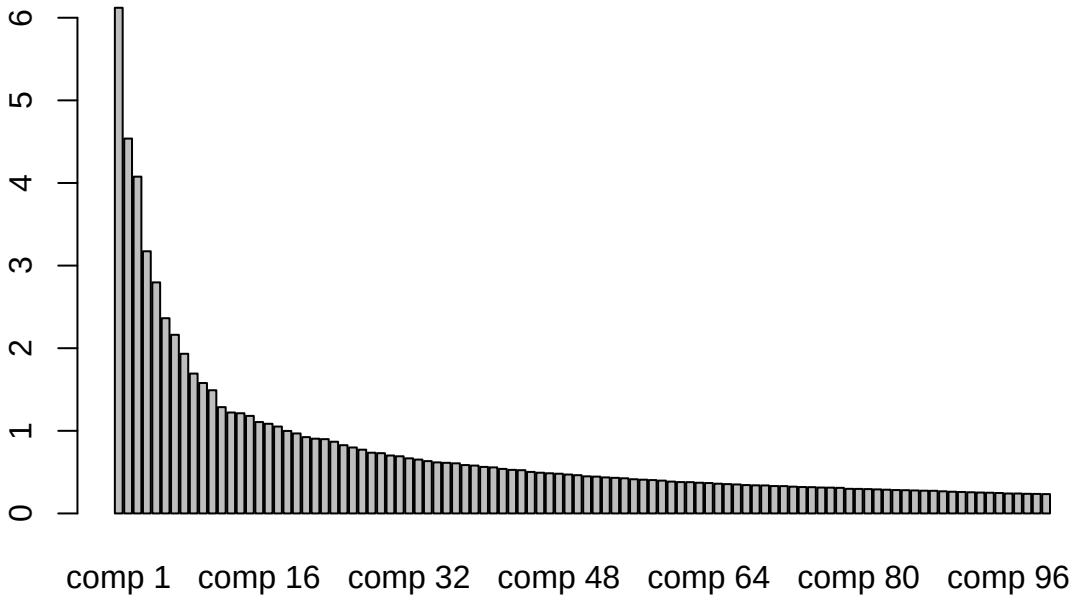
mais en fixant le lambda arbitrairement la performance est mauvaise.

Mais si on voulait faire de la validation croisée pour choisir le lambda, ce serait beaucoup plus long...

```
library(glmnet)
fit=cv.glmnet(app_x_reduit,app_y,alpha=1,family="multinomial",parallel=TRUE)
best_lam <- cv$lambda.min
lasso_best <- glmnet(app_x_reduit,app_y,alpha=1,family="multinomial",lambda = best_lam)
p <- predict(fit, newx = test_x_reduit)
table(p,test$y)
```

Une solution alternative est de passer par une ACP pour réduire la dimension. On réalise donc une ACP sur l'ensemble des données, mais en mettant en supplémentaire les individus de la base de test :

```
library(FactoMineR)
acp=PCA(rbind(app_x,test_x),graph=F,ncp = 100,ind.sup = 5001:6000)
barplot(acp$eig[1:100,2])
```



Une analyse visuelle rapide nous indique de choisir 11 composantes. Mais là encore, ce nombre pourrait être optimiser en fonction de notre objectif de prédiction. Quelques tests non concluant nous font augmenter ce nombre de composantes

```
ncomp=50
app_x_reduit=acp$ind$coord[,1:ncomp]
test_x_reduit=acp$ind.sup$coord[,1:ncomp]
app_reduit=data.frame(cbind(app_x_reduit,app_y))
names(app_reduit)[ncol(app_reduit)]="y"
app_reduit$y=as.factor(app_reduit$y)
test_reduit=data.frame(cbind(test_x_reduit,test_y))
names(test_reduit)[ncol(test_reduit)]="y"
test_reduit$y=as.factor(test_reduit$y)
```

Mais là encore, la performance du modèle est très mauvaises

```
library(glmnet)
fit=glmnet(app_x_reduit,app_y,alpha=1,family="multinomial",lambda=0)
p <- predict(fit, newx = test_x_reduit,type='class')
table(p,test_y)
```

```
##    test_y
## p     0   1   2   3   4   5   6   7   8   9
##   0 113 108  93 114  84  79 107 101  89 100
##   1   0   0   0   1   0   0   0   0   0   0
##   3   0   0   0   0   0   0   0   0   0   1
##   4   0   0   0   0   1   0   0   0   0   4
##   6   0   0   0   0   0   1   0   0   0   0
##   7   0   0   0   0   0   0   0   0   0   1
##   9   0   0   0   0   3   0   0   0   0   0
```

```
cat('taux de bons classements :',sum(p==test_y)/length(p))
```

```
## taux de bons classements : 0.114
```

Si on compare aux forêts aléatoires sur les mêmes données de dimension réduite, on comprend que le soucis n'est ici pas la réduction de dimension mais le modèle de régression logistique qui n'est pas bien adapté du

tout à ces données.

```
model <- randomForest(y~.,data=app_reduit,ntree=200)
pred<-predict(model,newdata=test_reduit,type="class")
table(pred,test$y)
```

```
##
## pred   0   1   2   3   4   5   6   7   8   9
##    0 110   0   1   0   0   1   0   1   1   2
##    1   0 105   0   1   1   0   0   1   0   0
##    2   0   1  83   2   2   0   0   2   2   0
##    3   0   1   4 107   0   3   0   0   3   1
##    4   0   0   0   0  81   1   1   2   0   4
##    5   0   0   0   2   0  75   4   0   1   0
##    6   2   0   2   0   1   0 101   0   1   0
##    7   0   0   0   0   0   0   0  88   1   5
##    8   1   0   2   1   1   0   1   0  80   0
##    9   0   1   1   2   2   0   0   7   0  94
```

```
cat('taux de bons classements :',sum(pred==test_y)/length(res))
```

```
## taux de bons classements : 0.924
```

SVM

On peut utiliser le package suivant pour faire de la classification en 10 classes. On laisse ici le kernel par défaut, mais il faudrait en tester plusieurs (c'est un hyperparamètre de la méthode)

```
library(kernlab)
svm1=ksvm(y ~ ., data = app, scaled = FALSE, kernel = "rbfdot")
pred_svm1 <- predict(svm1, newdata = test_x, type = "response")
table(pred_svm1, test_y)
```

```
##           test_y
## pred_svm1   0   1   2   3   4   5   6   7   8   9
##         0 111   0   2   1   0   0   0   0   0   0
##         1   0 105   1   1   0   0   0   0   0   0
##         2   0   1  84   1   0   0   2   0   2   0
##         3   0   1   3 110   0   2   0   0   0   2
##         4   1   0   1   0  86   2   0   1   0   4
##         5   0   0   0   1   0  76   2   0   2   0
##         6   0   0   0   0   0   0 103   0   2   0
##         7   0   0   2   0   0   0   0  95   1   1
##         8   1   0   0   1   0   0   0   0  81   1
##         9   0   1   0   0   2   0   0   5   1  98
```

```
cat('taux de bons classements :',sum(pred_svm1==test_y)/length(test_y))
```

```
## taux de bons classements : 0.949
```

Les résultats sont très bons.

Réseaux de neuronnes

Commençons avec la librairie nnet.

```
library(nnet)
nn2=nnet(y~.,data=app,size=1,maxit=100,trace=F)
pred=predict(nn2,test_x,type = 'class')
table(pred,test_y)
```

Ca ne marche pas du tout avec 1 neuronne, et si on en veut plus il dit qu'il a trop de paramètres...

Essayons avec neuralnet. Après plusieurs essais, il semble que le réseau marche mieux si on centré et réduit les données :

```
library(caret)
normParam <- preProcess(app)
app=predict(normParam, app)
test=predict(normParam, test)
```

Appliquons un réseau à une couche et 10 neuronnes.

```
library(neuralnet)
nn <- neuralnet(y~.,data=app,hidden=c(10),linear.output=FALSE,stepmax = 20000)
pred=predict(nn,test[,1:784])
classe_pred=apply(pred, 1, which.max)-1
table(classe_pred,test_y)
cat('taux de bons classements :',sum(classe_pred==test_y)/nrow(pred))
```

Les résultats sont corrects (81.2% de bons classements), mais bien moins bons qu'avec les random forest ou encore les SVM. Cela montre que les réseaux de neuronnes, bien que très puissants, demande une grande expertise pour bien les "régler". On trouvera ici une façon de bien configurer un réseau sur les données MNIST https://stackoverflow.com/questions/21827195/unexpected-output-while-using-neuralnet-in-r