Data Mining - TP classification

Julien JACQUES

14 avril, 2023

Données MNIST

Chargeons les images

```
#source("mnist.R")
load('MNIST.Rdata')
show_digit <- function(x, col=gray(12:1/12), title='image') {
image(matrix(x, nrow=28)[,28:1], col=col, main=title)
}
```

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  78   0   0   0   0   0   1   0
##   3   0   0   3 109   0   1   0   0   2   2
##   4   1   0   0   0  80   1   1   0   0   2
##   5   1   0   0   3   0  73   1   0   1   0
##   6   1   0   1   0   1   0 105   0   3   0
##   7   0   1   3   0   0   0   0  90   1   2
##   8   0   0   0   0   0   0   0   0  78   0
##   9   0   1   0   0   3   2   0   7   0  99
```

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

```
## taux de bons classements : 0.928
```

arbre de décision

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

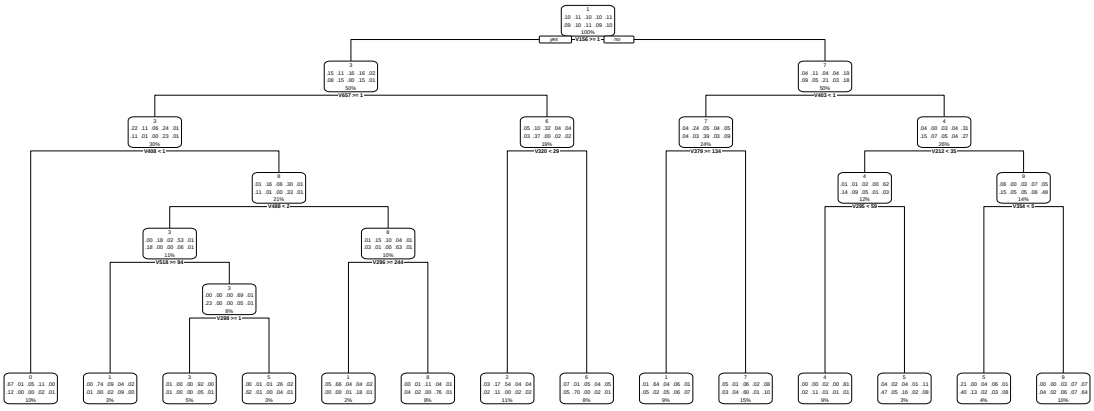
```
library('rpart')
library('rpart.plot')
```

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
##    0 110   0   2   0   0   0   0   0   1   1
##    1   0 104   0   1   0   0   0   0   0   1
##    2   0   1  84   0   0   0   0   0   1   1
##    3   0   0   3 110   0   1   0   0   1   2
##    4   1   1   1   0  84   1   1   1   0   4
##    5   0   0   0   2   0  76   1   0   1   0
##    6   0   0   1   0   2   1 104   0   4   0
##    7   0   1   1   0   0   0   0  92   1   0
##    8   2   0   1   2   0   0   1   0  80   0
##    9   0   1   0   0   2   1   0   8   0  97
```

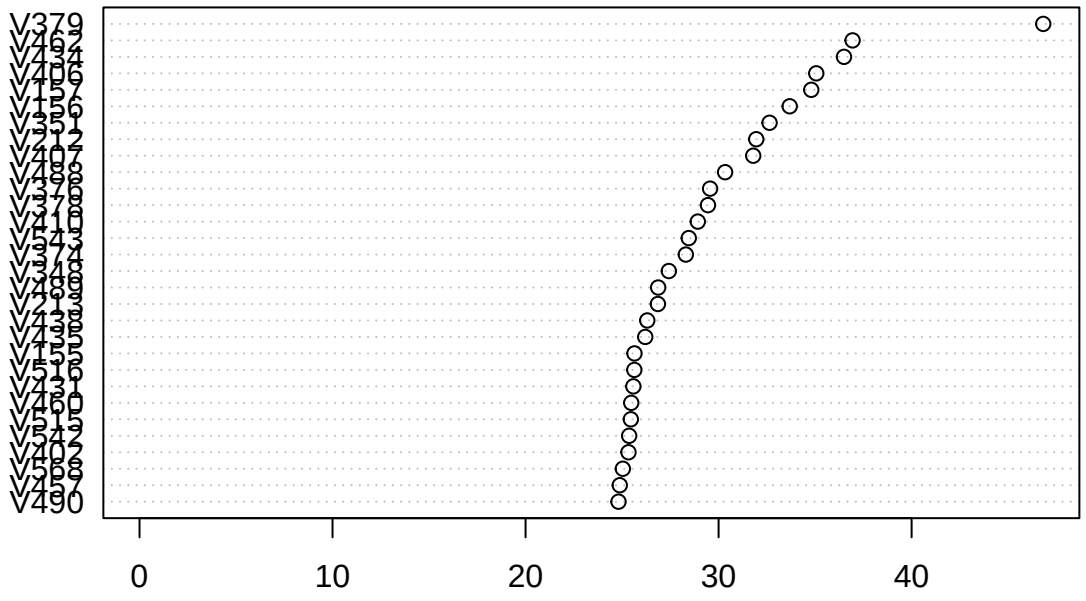
```
cat('taux de bons classements :',mean(pred==test_y))
```

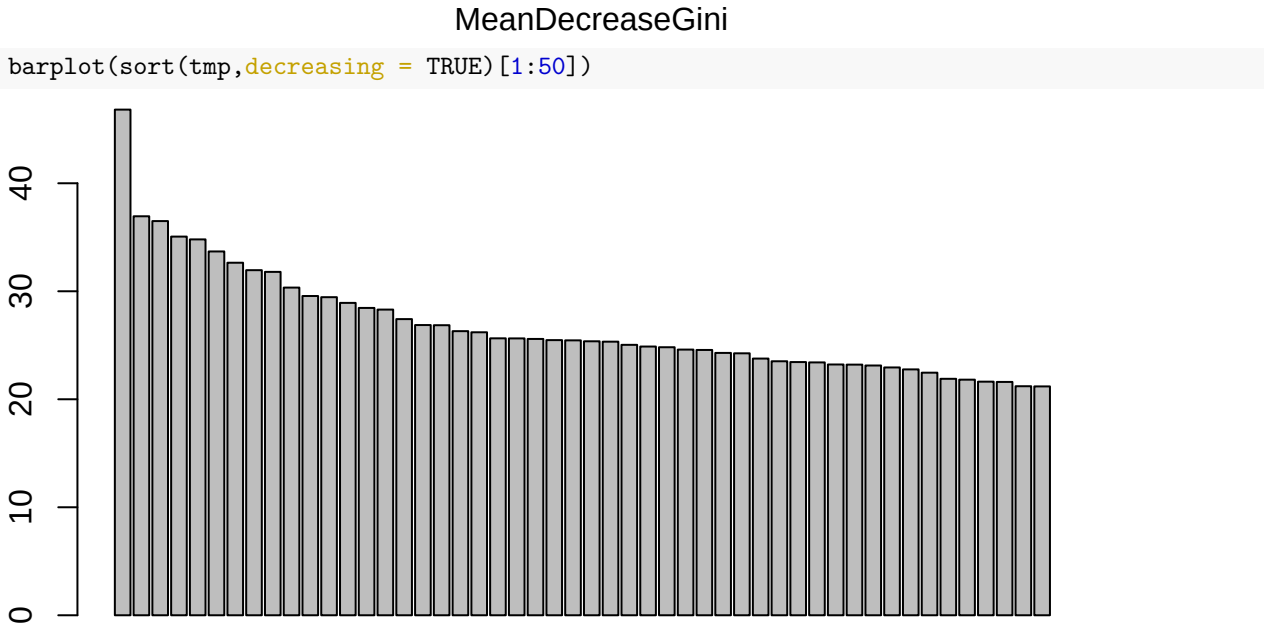
```
## taux de bons classements : 0.941
```

Les forêts aléatoires fonctionnent très bien.

On peut chercher à sélectionner les variables importantes en prenant les meilleurs suivants l'indicateur d'importance.

```
tmp=varImpPlot(model)
```






```
barplot(sort(tmp,decreasing = TRUE)[1:50])
```

```
which(tmp>sort(tmp,decreasing = TRUE)[8])
```

```
## [1] 156 157 351 379 406 434 462
```

On choisit ici les 8 variables les plus importantes, car il semble y avoir une cassure dans l'ébouli d'importance des variables

```
model2 <- randomForest(y~V156+V212+V357+V379+V404 +V406+V462,data=app,ntree=200)
pred<-predict(model2,newdata=test,type="class")
table(pred,test$y)
```

```
##
## pred   0   1   2   3   4   5   6   7   8   9
```

```
##    0  90   0   6   9   0   3   6   7   0   2
##    1   0 100   4   1   1   0   1   3   5   0
##    2   0   2  38   4   3   1  14   0   6   2
##    3   1   1   7  55   1   8   7   1  10   4
##    4   0   1   7   1  66  12   7   2   0   5
##    5   2   0   2  16   8  34   7   1   3   7
##    6   0   0   8  10   2   4  51   3   5   1
##    7  16   2   3   3   3   6   4  76   0  13
##    8   2   2  14   8   0   0   8   0  49   2
##    9   2   0   4   8   4  12   2   8  11  70
```

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

```
## taux de bons classements : 0.629
```

On perd pas mal en performance de prédiction.

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)]
```

```
library(glmnet)
```

```
## Le chargement a nécessité le package : Matrix
```

```
## Loaded glmnet 4.1-6
```

```
fit=cv.glmnet(app_x_reduit,app_y,alpha=1,family="multinomial",parallel=TRUE)
```

```
## Warning: executing %dopar% sequentially: no parallel backend registered
```

```
p <- predict(fit, newx = test_x_reduit,s='lambda.1se',type='class')
table(p,test$y)
```

```
##
## p     0   1   2   3   4   5   6   7   8   9
##   0 109   0   2   0   0   0   0   1   1   3
##   1   0 101   1   3   2   1   0   0   1   0
##   2   0   1  77   0   0   0   3   2   2   0
##   3   1   0   5 105   0   4   0   0   3   1
##   4   1   1   2   1  80   6   0   1   1   6
##   5   0   1   0   5   1  65   2   0   3   0
##   6   0   0   2   0   1   1 101   0   4   0
##   7   0   1   2   0   0   0   0  94   1   4
##   8   2   2   2   1   0   1   1   0  73   0
##   9   0   1   0   0   4   2   0   3   0  92
```

Recommençons sur les données de départ pour comparer

```
require(doMC)
```

```
## Le chargement a nécessité le package : doMC
```

```
## Le chargement a nécessité le package : foreach
```

```
## Le chargement a nécessité le package : iterators
```

```
## Le chargement a nécessité le package : parallel
```

```
registerDoMC(cores=2)
library(glmnet)
fit=cv.glmnet(app_x,app_y,alpha=1,family="multinomial",parallel=TRUE)
p <- predict(fit, newx = test_x,s='lambda.1se',type='class')
table(p,test_y)
```

```
##    test_y
## p     0   1   2   3   4   5   6   7   8   9
##   0 110   0   2   0   0   0   0   1   1   3
##   1   0 101   1   3   2   1   0   0   1   0
##   2   0   1  77   0   0   0   3   2   2   0
##   3   0   0   5 105   0   4   0   0   3   1
##   4   1   1   2   1  80   6   0   1   1   6
##   5   0   1   0   5   1  66   2   0   3   0
##   6   0   0   2   0   1   0 101   0   4   0
##   7   0   1   2   0   0   0   0  94   1   4
##   8   2   2   2   1   0   1   1   0  72   0
##   9   0   1   0   0   4   2   0   3   1  92
```

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)
```

```
##     test_y
## pred   0   1   2   3   4   5   6   7   8   9
##    1 105 101  41 113  12  68  23   6  83  10
##    4   8   7  52   2  76  12  84  95   6  96
```

Ca ne marche pas du tout avec 1 neuronne, et si on en veut plus on obtient une erreur indiquant qu'il y 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)
```

```
## Le chargement a nécessité le package : ggplot2
```

```
##
## Attachement du package : 'ggplot2'
```

```
## L'objet suivant est masqué depuis 'package:kernlab':
##
##     alpha
```

```
## L'objet suivant est masqué depuis 'package:randomForest':
##
##     margin
```

```
## Le chargement a nécessité le package : lattice
```

```
normParam <- preProcess(app)
```

```
## Warning in preProcess.default(app): These variables have zero variances: V1,
## V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17, V18,
## V19, V20, V21, V22, V23, V24, V25, V26, V27, V28, V29, V30, V31, V32, V33, V34,
## V35, V52, V53, V54, V55, V56, V57, V58, V59, V60, V82, V83, V84, V85, V86, V87,
## V88, V110, V111, V112, V113, V114, V115, V139, V140, V141, V142, V169, V170,
## V281, V308, V309, V337, V365, V392, V393, V420, V421, V449, V450, V477, V478,
## V505, V506, V533, V534, V561, V562, V588, V589, V590, V616, V617, V618, V644,
## V645, V646, V672, V673, V674, V675, V698, V699, V700, V701, V702, V703, V726,
## V727, V728, V729, V730, V731, V753, V754, V755, V756, V757, V758, V759, V760,
## V761, V780, V781, V782, V783, V784
```

```
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)
```

```
##            test_y
## classe_pred   0   1   2   3   4   5   6   7   8   9
##           0  97   0   3   3   2   0   0   1   2   3
##           1   0 100   1   1   2   0   0   0   1   0
##           2   1   0  66   7   1   0   4   2   3   2
```

```
##           3   2   0  11  88   0   5   1   1   4   1
##           4   1   2   0   1  73   5   5   2   3   8
##           5   6   2   0   8   1  63   4   0   4   4
##           6   3   1   6   0   3   2  89   0   0   1
##           7   0   1   4   5   1   1   3  90   2   6
##           8   3   1   2   2   2   3   1   2  68   4
##           9   0   1   0   0   3   1   0   3   2  77
```

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

```
## taux de bons classements : 0.811
```

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