1 Objectif

Un cas d'étude pour montrer comment optimiser un programme sous R. Outils de benchmarking et de profiling de code. Programmation de la procédure leave-one-out pour l'évaluation des modèles en analyse prédictive.

Les étudiants m'interrogent parfois sur mon langage de programmation favori. Je réponds invariablement Delphi. En réalité, plus que du langage en lui-même, il s'agit plutôt de technologie. J'ai énormément pratiqué le compilateur Delphi 6, je le connais tellement bien que je rédigeais mon code en anticipant sur son comportement. Je suis moins dans le développement aujourd'hui. Mon activité favorite consiste à découvrir, étudier, et faire découvrir en écrivant des supports de cours et des tutoriels, sous R et sous Python principalement. Mais il reste un petit fond de geek en moi et je ne mégote pas mon plaisir quand il s'agit de grapiller des millisecondes sur une petite procédure de derrière les fagots.

Mon attention a été attirée récemment par l'excellent ouvrage ''Efficient R Programming'' de Gillepsie et Lovelace, accessible en ligne (https://csgillespie.github.io/efficientR/). Les auteurs exposent les tenants et aboutissants de la programmation efficace sous R. Au-delà des trucs et astuces, ils discutent des principes de l'écriture de programmes performants (en occupation mémoire et en temps d'exécution) et présentent – entres autres – les outils de benchmarking et de profiling (profilage en français) de code. Je me suis dit que ce serait une bonne chose d'illustrer leur utilisation dans le cadre de la programmation d'une procédure type de machine learning.

Mon choix s'est porté sur la programmation du *leave-one-out*, une procédure de rééchantillonnage pour l'évaluation des modèles prédictifs. L'optimisation du code reposera sur une analyse fine des étapes grâce à l'outil de profiling. Nous pourrons gratter du temps d'exécution en jouant sur les spécificités des primitives de calcul utilisées, le choix des structures de données, la parallélisation des calculs. Tout cela en conservant la lisibilité du code source, gage indispensable de pérennité des applications sur le long terme.

2 Procédure ''leave-one-out''

Le ''leave-one-out'' (**LOOCV**, leave-one-out cross-validation) est une procédure de rééchantillonnage qui permet d'estimer les performances prédictives d'un modèle élaboré sur un dataset de taille n. Cette technique est un cas particulier de la validation croisée, qui s'impose lorsque nous disposons d'une base de données de taille réduite et qu'il n'est pas réaliste de la subdiviser en échantillons d'apprentissage et de test.

L'algorithme est le suivant :

```
v = vecteur des prédictions
Pour i allant de 1 à n
    Construire le modèle sur toutes les observations sauf le n°i (n-1 observations)
    Prédire sur l'individu n°i, collecter la valeur dans v
Croiser l'attribut cible et le vecteur des prédictions v
En déduire la proportion de mauvaises prédictions, il s'agit du taux d'erreur en LOOCV
```

On va construire n fois le modèle prédictif, mieux vaut avoir une bonne machine et/ou savoir programmer astucieusement, c'est ce que nous allons essayer de faire dans ce tutoriel, en nous aidant des outils de benchmarking et de profiling.

Nous utiliserons l'analyse discriminante (lda du package MASS) en guise d'algorithme de machine learning. L'information n'est pas anodine. En effet, elle est stable (faible variance), peu propice au surapprentissage, l'utilisation de la LOOCV pour mesurer l'erreur en généralisation se justifie pleinement dans son contexte.

3 Données

Nous utilisons une variante de la base ''waveform'' avec 1000 observations. La variable cible ''onde'' est à 3 modalités, nous disposons de 21 descripteurs (V1…V21).

De fait, nous aurons à réaliser n = 1000 cycles d'apprentissage sur (n – 1) = 999 observations durant la LOOCV. Il faut espérer que la fonction `lda()` soit bien programmée, sinon nous risquons d'attendre un moment devant notre PC, avec un ventilateur s'époumonant

bruyamment, essayant tant bien que mal d'évacuer les calories générées par les sollicitations des ressources de calcul.

Voici le code pour importer et vérifier l'intégrité des données.

```
#changer de répertoire
setwd("... votre dossier de travail ...")

#charger les données
D <- read.csv("wave1000.txt",header=TRUE,sep="\t",dec=".")
print(str(D))
```

```
'data.frame':1000 obs. of  22 variables:
 $ onde: Factor w/ 3 levels "A","B","C": 1 3 1 2 3 3 2 2 3 1 ...
 $ V01 : num  0.31 -1.03 -1.08 -0.37 -1.02 -2 -1.41 -0.72 -0.47 1.42 ...
 $ V02 : num  1.8 -0.45 1.59 0.21 -1.29 -0.44 -1.19 1.37 -0.02 -0.14 ...
 $ V03 : num  0.86 -0.34 0.91 0.57 -1.24 0.13 0.22 1.3 0.59 0.61 ...
 $ V04 : num  2.34 2.08 2.95 1.67 -1.96 -1.41 0.68 3.6 -0.44 1.17 ...
 $ V05 : num  1.25 -0.81 5.01 3.06 0.6 0.27 3.01 3.63 0.84 2.7 ...
 $ V06 : num  4.33 0.12 5.42 2.32 0.18 -2.76 3.1 3.4 0.37 1.35 ...
 $ V07 : num  4.76 3.35 4.79 4.36 1.28 1.17 2.45 6.88 1.3 3.66 ...
 $ V08 : num  3.46 2.78 5.32 5.02 4.06 2.28 3.29 4.99 0.31 2.17 ...
 $ V09 : num  3.88 3.53 3.07 4.37 3.24 1.16 2.33 2.71 0.87 1.65 ...
 $ V10 : num  1.28 5.27 1.35 5.19 3.23 3.78 3.41 2.75 1.05 3 ...
 $ V11 : num  3.94 5.78 1.89 4.09 5.02 1.48 5.43 3.79 3.2 3.7 ...
 $ V12 : num  2.28 5.88 1.85 3.62 7.4 2.97 3.83 0.52 3.79 3.19 ...
 $ V13 : num  0.81 2.35 -0.61 1.1 3.94 5.3 2.58 0.65 3.43 0.66 ...
 $ V14 : num  2.64 3.46 2.12 2.92 4.27 3.84 0.34 1.18 5.37 1.95 ...
 $ V15 : num  1.3 2.84 -1.07 1.08 2.44 3.85 1.74 0.83 4.64 4.26 ...
 $ V16 : num  -0.49 2.43 2.2 0.75 1.75 4.27 -0.39 1.61 2.71 -0.35 ...
 $ V17 : num  0.8 1.66 1.31 -1.43 -0.66 4.5 1.38 -1.63 2.59 1.6 ...
 $ V18 : num  0.59 -0.39 1.24 0.87 -0.38 4.24 0.8 -0.49 2.92 1.95 ...
 $ V19 : num  -1.25 0.03 0.66 0.85 0.14 2.17 0.97 1.2 2.13 -0.7 ...
 $ V20 : num  -2.38 0.6 0.86 -0.3 0.24 -0.1 0.09 0.12 1.56 1.17 ...
 $ V21 : num  -0.59 -1.71 0.47 0.56 -0.2 1.87 1.22 -1.49 0.58 -1.39 ...
```

4 LOOCV – Version 1

4.1 Programmation de la fonction pour la LOOCV

Notre première version de la fonction R pour le LOOCV retranscrit l'algorithme ci-dessus de la manière la plus simple possible.

```
#library MASS
library(MASS)

#leave-one-out - version 1
```

```
lvo_v1 <- function()
{
  #vecteur de résultat
  v <- c()
  #boucler
  for (i in 1:nrow(D)){
    #données de test
    DTest <- D[i,]
    #données d'apprentissage
    DTrain <- D[-i,]
    #modélisation
    modele <- lda(formula=onde ~ ., data=DTrain)
    #prédiction
    pred <- predict(modele, newdata=DTest)
    #récupère la prédiction
    v <- c(v,pred$class[1])
  }
  #matrice de confusion
  mc <- table(D$onde,v)
  #taux d'erreur
  err <- 1.0-sum(diag(mc))/sum(mc)
}
```

Il y a plusieurs éléments clés dans cette interprétation de l'algorithme qui manipule la variable D représentant l'ensemble de données initial en tant que variable globale (ce n'est pas très conventionnel, mais ce choix permettra de faciliter l'étude du code avec l'outil de profiling plus loin) :

1. Nous itérons sur chaque individu du data frame D, c'est le rôle de la boucle *for*.
2. Pour chaque itération, nous créons un sous-data frame DTest avec le seul individu n°i auquel sera appliqué la fonction *predict()*.
3. Et un autre data frame DTrain avec tous les individus sauf le n°i (de taille n-1 donc), auquel nous appliquerons l'analyse discriminante linéaire *lda()*.
4. Chaque prédiction est ajoutée au vecteur v qui sera de longueur n (le nombre d'observations dans la base) finalement.

Les étapes 2 à 4 sont exécutées 1000 fois dans notre cas. Un gain même faible sur une de ces parties engendrera une amélioration sensible du temps d'exécution à n'en pas douter.

La confrontation de la cible observée et prédite avec *table()*, puis le calcul du taux d'erreur err n'appelle pas de remarques particulières en revanche. Optimiser sur ces deux

dernières étapes n'apporterait pas grand-chose puisque les lignes de code correspondantes ne sont appelées qu'une seule fois.

Voyons le résultat et la durée d'exécution de notre fonction.

```
#vérification
system.time(print(lvo_v1()))
```

```
[1] 0.156
utilisateur     système      écoulé
       7.55        0.00        7.63
```

Le taux d'erreur en LOOCV est de 15.6%. Le temps de calcul est de 7.63 secondes.

4.2 Benchmarking

La mesure du temps écoulé peut être perturbé par de nombreux facteurs, en particulier la sollicitation de la machine durant l'expérimentation. Pour disposer d'une mesure fiable, il est conseillé de répéter plusieurs fois l'opération. Nous pourrions écrire une boucle pour cela, mais nous préférons utiliser le package ''microbenchmark'' qui se charge de tout (lancer les répétitions, collecter les résultats), et permettra de réaliser des comparaisons par la suite.

```
#package pour mesurer le temps de calcul
library(microbenchmark)

#mesure avec 5 répétitions
print(microbenchmark(times=5,lvo_v1()))
```

Avec 5 itérations, nous avons des indications sur les durées min et max des traitements, et la médiane. Nous constatons qu'elles sont assez stables finalement.

```
Unit: seconds
     expr      min       lq     mean   median       uq      max neval
 lvo_v1() 7.100747 7.202584 7.265844 7.227133 7.387621 7.411134     5
```

4.3 Profiling

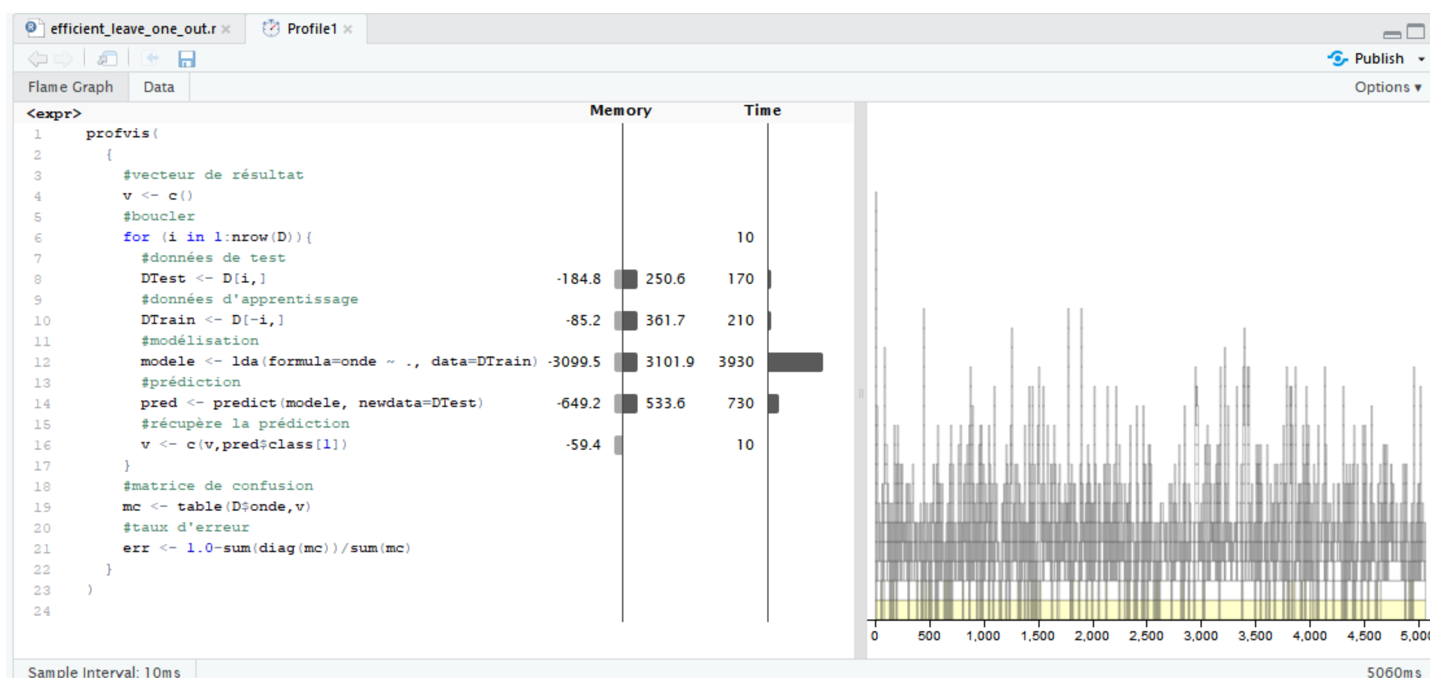
Maintenant arrive l'optimisation. Pour ce faire, il faut identifier les étapes les plus gourmandes en calcul dans notre première interprétation de la LOOCV, et voir si elles sont perfectibles. Nous utilisons le package ''profvis''. Elle se charge de lancer l'outil RProf() de R (package ''utils'') qui mesure le temps d'exécution de chaque ligne de code, de collecter les résultats et de les présenter de manière avenante.

Dans mes premiers tests, il suffisait de passer en paramètre de profvis() les fonctions à tester. Par la suite, avec la même approche, je n'avais plus le détail du contenu de la fonction, je n'ai jamais su pourquoi. Il m'a donc fallu recopier explicitement le code à tester dans l'appel de profvis(), c'est pour cela que j'ai du passer le data frame D en variable globale dans la fonction.

```
#package pour profilage de code
library(profvis)

#profiler le code
profvis(
  {
    #vecteur de résultat
    v <- c()
    #boucler
    for (i in 1:nrow(D)){
      #données de test
      DTest <- D[i,]
      #données d'apprentissage
      DTrain <- D[-i,]
      #modélisation
      modele <- lda(formula=onde ~ ., data=DTrain)
      #prédiction
      pred <- predict(modele, newdata=DTest)
      #récupère la prédiction
      v <- c(v,pred$class[1])
    }
    #matrice de confusion
    mc <- table(D$onde,v)
    #taux d'erreur
    err <- 1.0-sum(diag(mc))/sum(mc)
  }
)
```

Une fenêtre apparaît dans RStudio, c'est la partie gauche qui nous intéresse au plus haut point, en particulier la partie ''**Time**'' qui retrace la durée d'exécution de chaque ligne de code (Remarque : ''Memory'' correspond à la mémoire allouée et désallouée, cette information est importante si nous rencontrons des problèmes d'occupation mémoire, ce qui n'est pas notre cas dans cette expérimentation).



Nous constatons que :

- La partie la plus gourmande est l'appel de la fonction lda() (3930 millisecondes).
- Puis vient l'appel de predict() (730 ms).
- Les créations des data frame intermédiaires DTest et DTrain sont également voraces, mais dans une moindre mesure. Nous y reviendrons dans la section 6.

5 LOOCV – Version 2

Que peut-on faire pour réduire le temps de calcul sur lda() et predict() ?

1. Nous faisons usage de la formule "formula = onde ~ . " lors de l'appel de lda(). Cela oblige l'outil à parser l'expression et à la mettre en relation avec le data frame passé en paramètre de data. Or à la lecture de la documentation, on se rend compte que lda() propose une autre signature `lda(x, grouping, ...)`. Nous passons directement la matrice des descripteurs x et le vecteur de la variable cible y. Ce faisant, nous zappons l'étape répétitive d'interprétation de la formule.
2. Corollaire à cela, il ne sera plus nécessaire pour predict() de faire correspondre les noms de variables de formula avec ceux du data frame passé en paramètre de newdata.

Dans les deux cas, il y a moins d'interprétations à effectuer préalablement à la réalisation des calculs : moins d'opérations devrait impliquer une réduction du temps de calcul.

```
#seconde version - modifier l'appel de lda et predict
lvo_v2 <- function()
{
  #vecteur de résultat
  v <- c()
  #boucler
  for (i in 1:nrow(D)){
    #données de test
    DTest <- D[i,]
    #données d'apprentissage
    DTrain <- D[-i,]
    #modélisation
    modele <- lda(x=DTrain[,-1],grouping=DTrain$onde)
    #prédiction
    pred <- predict(modele, newdata=DTest[,-1])
    #récupère la prédiction
    v <- c(v,pred$class[1])
  }
  #matrice de confusion
  mc <- table(D$onde,v)
  #taux d'erreur
  err <- 1.0-sum(diag(mc))/sum(mc)
}

#vérification
system.time(print(lvo_v2()))
```

On a bien le même taux d'erreur, cette vérification était très importante. Et le temps de calcul connaît une réduction assez spectaculaire.

```
[1] 0.156
utilisateur     système      écoulé
       4.83        0.00        4.86
```

Comparons les deux approches de la LOOCV en réitérant 5 fois les traitements.

```
#benchmarking
print(microbenchmark(times=5,lvo_v1(),lvo_v2()))
```

```
Unit: seconds
     expr      min       lq     mean   median       uq      max neval
 lvo_v1() 7.250704 7.288669 7.302932 7.299069 7.310987 7.365231     5
 lvo_v2() 4.691743 4.756461 4.763046 4.769976 4.787479 4.809573     5
```

La seconde implémentation est plus efficace, la progression est significative.

De nouveau, nous passons le code à l'épreuve de l'outil de profiling `profvis()`.

```
#profiler le code - version 2
profvis(
  {
    #vecteur de résultat
    v <- c()
    #boucler
    for (i in 1:nrow(D)){
      #données de test
      DTest <- D[i,]
      #données d'apprentissage
      DTrain <- D[-i,]
      #modélisation
      modele <- lda(x=DTrain[,-1],grouping=DTrain$onde)
      #prédiction
      pred <- predict(modele, newdata=DTest[,-1])
      #récupère la prédiction
      v <- c(v,pred$class[1])
    }
    #matrice de confusion
    mc <- table(D$onde,v)
    #taux d'erreur
    err <- 1.0-sum(diag(mc))/sum(mc)
  }
)
```

Les gains sont effectivement substantiels sur ces deux étapes clés de l'implémentation. IL suffisait de lire la documentation pour obtenir ce résultat.

6 LOOCV – Version 3

Pour gagner encore du temps, attardons-nous un peu sur les structures de données. Les data frame sont des listes de vecteurs, qui peuvent être de type quelconque. L'accès par colonne doit être certainement efficace, par ligne c'est une autre affaire. Je suppute que les deux étapes où nous sélectionnons et excluons la ligne n°i pour créer les objets intermédiaires DTest et DTrain ne sont pas gérées efficacement.

Essayons de passer par une structure matricielle. Pour les descripteurs, ce sera un avantage certain, les types de données sont harmonisées et l'espace mémoire occupé est contigu. On peut espérer que les manipulations des lignes seront plus efficaces.

Voici le nouveau code.

```
#troisième version - passer par des matrices et vecteurs
lvo_v3 <- function()
{
  #matrice des descripteurs
  X <- as.matrix(D[,-1])
  y <- D[,1]
  #vecteur de résultat
  v <- c()
  #boucler
  for (i in 1:nrow(D)){
    #données de test
    XTest <- X[i,]
    #données d'apprentissage
    XTrain <- X[-i,]
    yTrain <- y[-i]
    #modélisation
    modele <- lda(x=XTrain,grouping=yTrain)
    #prédiction
    pred <- predict(modele, newdata=XTest)
    #récupère la prédiction
    v <- c(v,pred$class[1])
  }
  #matrice de confusion
  mc <- table(D$onde,v)
  #taux d'erreur
  err <- 1.0-sum(diag(mc))/sum(mc)
}

#vérification
system.time(print(lvo_v3()))
```

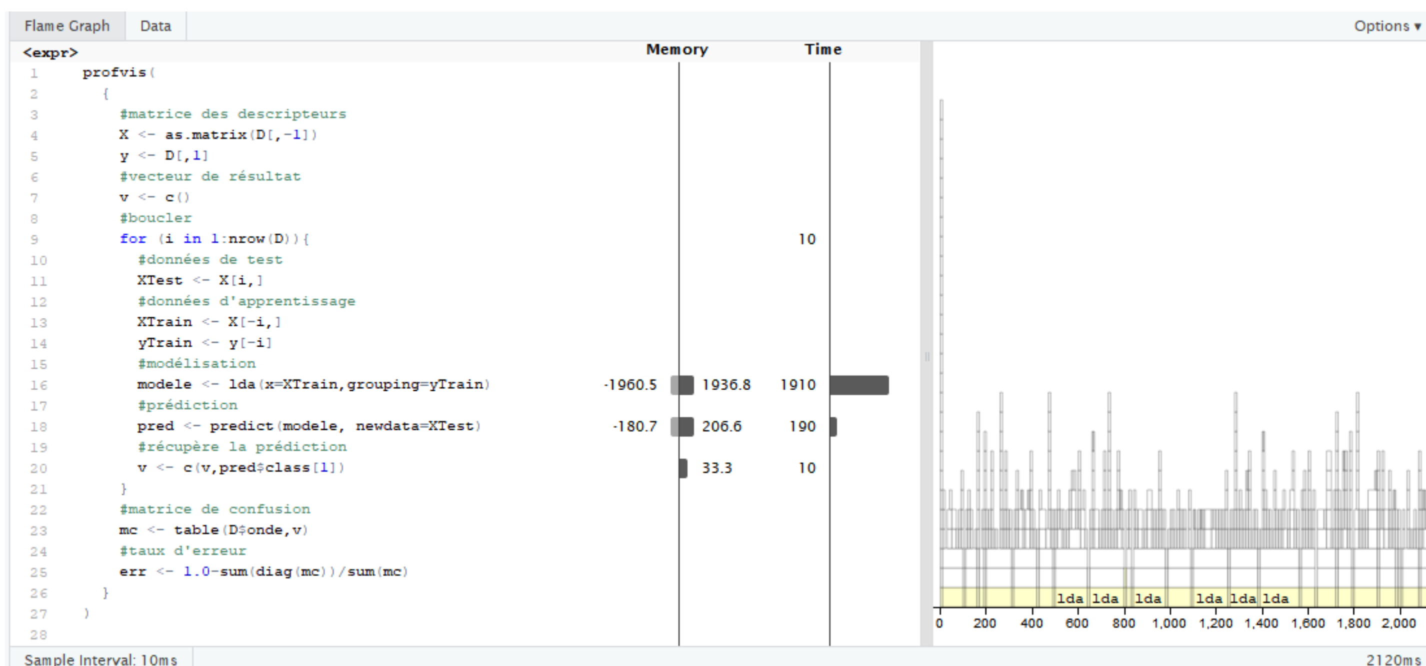
```
[1] 0.156
utilisateur      système       écoulé
          4            0            4
```

Nous sommes passés à 4 secondes. Confirmons cela avec l'outil de benchmarking.

```
#benchmarking
print(microbenchmark(times=5,lvo_v2(),lvo_v3()))
```

```
 Unit: seconds
     expr      min       lq     mean   median       uq      max neval
 lvo_v2() 4.938198 4.949924 4.999829 4.981190 5.028926 5.100908     5
 lvo_v3() 3.868182 3.932292 3.948329 3.938576 3.975252 4.027345     5
```

Nous avons gagné une seconde pleine en créant et en exploitant les variables intermédiaires X et y, sans entacher la lisibilité du code. L'outil de profiling confirme l'amélioration.



Nous constatons par la même occasion que lda() et predict() bénéficient également de cette modification du type des données à manipuler.

Certes cette modification était moins évidente que la première, il fallait un peu connaître R et ses structures internes pour avoir cette idée. Mais à la sortie, le gain est appréciable.

7 LOOCV – Programmation parallèle

Puisque je dispose d'une machine avec un processeur multicœurs, pourquoi ne pas exploiter la programmation parallèle ? Je m'étais déjà essayé à la chose sous R il y a quelques années

(''Programmation parallèle sous R'', juin 2013). Notre ouvrage de référence donne également des indications assez précises sur l'utilisation du package ''parallel'' (Section 7.5).

Mon processeur étant un quad-core, je vais diviser en 4 les calculs. Pour ce faire, nous dispatchons le vecteur de numéros des individus dans une liste à 4 vecteurs. Nous pouvons ensuite appliquer la méthode parSapply() sur cette liste où chaque bloc (*chunk*) sera traité séparément sur un cluster (selon la terminologie de ''parallel'').

```
#librairie pour prog. parallèle
library(parallel)

#programmation parallèle de la LOOCV
lvo_parallel <- function(){
  #fonction comptage des erreurs pour un bloc
  chunk_lvo <- function(index,descriptors,y){
    v <- c()
    for (i in index){
      #données de test
      XTest <- descriptors[i,]
      #données d'apprentissage
      XTrain <- descriptors[-i,]
      yTrain <- y[-i]
      #modélisation
      modele <- MASS::lda(x=XTrain,grouping=yTrain)
      #prédiction
      pred <- predict(modele, newdata=XTest)
      #récupère la prédiction
      v <- c(v,pred$class)
    }
    #décompte des erreurs
    nb_errs <- sum(as.numeric(unclass(y[index]) != v))
    return(nb_errs)
  }
  #matrice des descripteurs
  XD <- D[,-1]
  #vecteur de la variable cible
  yD <- D[,1]
  #spliter le vecteur désignant les individus en 4 listes
  id <- 1:nrow(D)
  lst_id <- split(id,id%/%(nrow(D)%/%4+1))
  #préparer les 4 clusters
  cl = makeCluster(4)
  #stopper les clusters à la sortie de la fonction
  on.exit(stopCluster(cl))
  #lancer les calculs - on a vecteurs des erreurs par chunk
```

```
  erreurs <- parSapply(cl,X=lst_id,chunk_lvo,descriptors=XD,y=yD)
  #taux d'erreur
  return(sum(erreurs)/nrow(D))
}
```

Quelques commentaires :

- Dans la fonction **lvo_parallel()**, nous préparons la matrice des descripteurs XD et le vecteur yD représentant la variable cible.
- Puis nous dispatchons l'identifiant des individus dans une liste à 4 vecteurs, c'est le rôle de la fonction split().
- Nous démarrons un cluster de 4 machines avec makeCluster().
- Pour être sûr qu'elles seront bien stoppées à la sortie de la fonction, nous appelons stopCluster() dans la clause on.exit().
- Nous lançons alors le calcul avec parSapply(). La procédure procède à 4 appels (parce qu'on a 4 vecteurs dans la liste lst_id) en parallèle de la fonction locale **chunk_lvo()**, laquelle tient une place centrale dans le dispositif.
- Etant appelé en parallèle dans des environnements différents, il est hors de question de manipuler des variables globales dans chunk_lvo(), c'est pour cela que nous lui passons explicitement en paramètres : les identifiants des individus à traiter (index), la matrice complète des descripteurs (descriptors), et le vecteur cible (y).
- Nous n'avons pas la garantie que les blocs finaliseront le calcul dans l'ordre des appels. De fait, récupérer un vecteur global des prédictions à confronter avec la variable ''onde'' est compliqué ici (il faudrait un identifiant de bloc pour savoir dans quel ordre concaténer les vecteurs de prédiction).
- Je suis passé par une autre voie : chunk_lvo() renvoie le nombre d'erreurs par bloc, il suffit de les additionner pour avoir le total des erreurs, lequel sera divisé par l'effectif pour obtenir le taux d'erreur en LOOCV. Cette astuce nous affranchit du suivi de l'ordre de réception des résultats.

Voyons ce que cela donne :

```
#appel de la fonction
system.time(print(lvo_parallel()))
```

```
[1] 0.156
```

```
utilisateur     système       écoulé
       0.02        0.00         2.33
```

Le calcul est correct (taux d'erreur = 0.156), et nous obtenons un gain assez intéressant en termes de rapidité des traitements. Confirmons cela en la comparant avec la version précédente lvo_v3() qui était la plus rapide jusqu'à présent.

```
#comparaisons
print(microbenchmark(times=5,lvo_v3(),lvo_parallel()))
```

```
Unit: seconds
           expr      min       lq     mean   median       uq      max neval
       lvo_v3() 3.654640 3.662621 3.691752 3.684537 3.708540 3.748423     5
 lvo_parallel() 2.148017 2.177586 2.216542 2.215299 2.244804 2.297002     5
```

Nous ne divisons pas par 4 la durée de traitements parce qu'il y a un temps nécessaire de préparation des données et de synchronisation (cf. ''Programmation parallèle sous R'', juin 2013 ; section 3). Mais il n'en reste pas moins que le gain est appréciable. Une parallélisation bien menée est toujours bénéfique, nous remarquerons également que la lisibilité du programme n'est pas dégradée. Seule la partie consacrée à la subdivision du vecteur d'identifiants en une liste de 4 vecteurs avec `split()` peut paraître un peu abscons.

8 LOOCV native de lda()

J'en étais à ce stade, assez content de moi j'avoue, lorsque je me suis replongé dans la documentation de lda() par acquit de conscience. Et j'ai découvert avec stupéfaction que l'option (CV = TRUE) revenait à produire les résultats de la validation croisée : ''*If true, returns results (classes and posterior probabilities) for leave-one-out cross-validation*''. J'étais vert (un comble pour un lyonnais amoureux de foot), mais je ne pouvais pas passer à côté.

8.1 L'option CV de lda()

J'ai donc implémenté cette option pour la confronter à mes implémentations.

```
#utiliser l'implémentation native de lda
lvo_lda <- function()
{
  #matrice des descripteurs
  X <- as.matrix(D[,-1])
  y <- D[,1]
  #lda avec LVO
  p <- lda(x=X,grouping=y,CV=TRUE)
```

```
  #matrice de confusion
  mc <- table(D$onde,p$class)
  #taux d'erreur
  err <- 1.0-sum(diag(mc))/sum(mc)
}

#vérification
system.time(print(lvo_lda()))
```

```
[1] 0.154
utilisateur     système      écoulé
       0.00        0.00        0.03
```

Argh, j'ai eu une double coup au cœur (j'exagère, j'exagère ….) :

- Le temps de calcul est sans commune mesure avec ce que nous avions réalisé jusqu'à présent, nous sommes très loin du compte.
- Le taux d'erreur présenté (0.154) est différent du nôtre (0.156). Aurions-nous laissé passer une erreur dans notre implémentation ?

C'est le temps de calcul qui m'a mis sur la bonne piste. Clairement, la procédure ne réitère pas 1000 fois le schéma apprentissage-test. Elle doit calculer une seule fois le modèle sur la totalité des données, et utiliser par la suite un artifice similaire au mécanisme du levier en régression pour obtenir les prévisions en tant qu'observations supplémentaires en LOOCV (voir ''Pratique de la régression linéaire multiple – Diagnostic et sélection de variables'', 2015 ; chapitre 1). Mais j'ai beau cherché, je n'ai pas trouvé les formules utilisées sur le net.

8.2 Vérification du taux d'erreur

Et cette stratégie de calcul explique les différences entre les taux d'erreurs. La documentation de lda() nous annonce, toujours concernant l'option (CV = TRUE) : ''*Note that if the prior is estimated, the proportions in the whole dataset are used.*''.

J'ai donc essayé de réimplémenter la LOOCV en estimant au préalable un vecteur des fréquences relatives des classes sur la totalité des données, stockées dans la variable proba. Il sera systématiquement utilisé pour chaque appel de `lda()`.

```
#distribution des classes sur la totalité des données
proba <- as.numeric(table(D$onde)/nrow(D))

#programmation du LVO
```

```
#pour retrouver les résultats de lda(CV=TRUE)
lvo_my_lda <- function()
{
  #matrice des descripteurs
  X <- as.matrix(D[,-1])
  y <- D[,1]
  #vecteur de résultat
  v <- c()
  #boucler
  for (i in 1:nrow(D)){
    #données de test
    XTest <- X[i,]
    #données d'apprentissage
    XTrain <- X[-i,]
    yTrain <- y[-i]
    #modélisation
    modele <- lda(x=XTrain,grouping=yTrain,prior=proba)
    #prédiction
    pred <- predict(modele, newdata=XTest)
    #récupère la prédiction
    v <- c(v,pred$class[1])
  }
  #matrice de confusion
  mc <- table(D$onde,v)
  #taux d'erreur
  err <- 1.0-sum(diag(mc))/sum(mc)
}

#vérification
print(lvo_my_lda())
```

```
[1] 0.154
```

Nous retrouvons bien le résultat de lda() avec l'option CV = TRUE (taux d'erreur = 0.154). La moralité de tout ceci est qu'il faut toujours lire attentivement la documentation pour éviter les incompréhensions, voire les désillusions.

9 Conclusion

L'interpréteur R a quand-même bien évolué depuis les premières versions. Je sais qu'il y a eu un gap notable avec la version 3.5.0, j'utilise la 3.5.2 dans ce tutoriel. La boucle *for* itérant sur les individus de la base aurait eu des conséquences catastrophiques il y a quelques années. Il en est de même en ce qui concerne le vecteur de prédictions dont j'augmente la

taille graduellement (petites allocations mémoires successives, argh). Là R avale tout ça sans moufeter. Je suis assez ébahi.

On sait que R compile à la volée le code préalablement à l'exécution maintenant, efficacement visiblement. Mais cela ne nous empêche d'essayer d'optimiser notre code. L'exemple du ''leave-one-out'' qui nous a servi de fil conducteur dans ce tutoriel montre qu'il est possible de gagner aisément en rapidité avec une bonne connaissance des primitives de calcul utilisées, un choix judicieux des structures de données, et l'exploitation de la programmation parallèle.

Enfin, *last but not least*, nous avons travaillé sous R, mais ces principes se généralisent facilement à d'autres langages et outils de développement.

10 Références

C. Gillespie, R. Lovelace, ''Efficient R Programming'', 2017 ; https://csgillespie.github.io/efficientR/

Tutoriel Tanagra, ''Validation croisée, Bootstrap – Diapos'', février 2015 ; http://tutoriels-data-mining.blogspot.com/2015/02/validation-croisee-bootstrap-diapos.html