Analyse Discriminante Linéaire sous R

Tutoriel Tanagra

1 Introduction

En rédigeant mon précédent tutoriel consacré à l'analyse discriminante sous Python (TUTO 1), je me suis rendu compte que je n'avais jamais écrit d'équivalent pour R, où l'on passerait en revue les fonctionnalités de la fonction lda() du package "MASS" qui fait référence sous cet environnement. C'est étonnant, surtout que je la pratique depuis un bon moment déjà. C'est même une des premières méthodes de R que j'ai explorées avec glm() et rpart(). Nous allons essayer d'y remédier dans ce document.

Les requêtes Google faisant foi, lda() semble être la fonction R la plus populaire en matière d'analyse discriminante. Pourtant, de prime abord, elle n'est pas facile à appréhender. A l'instar de ce que l'on peut trouver ici ou là dans les présentations que l'on en fait dans la littérature, elle mixe les visées descriptives et prédictives. J'avais pu montrer cette particularité succinctement dans un comparatif des implémentations de la méthode (TUTO 2). Là où SAS propose deux procédures distinctes (proc candisc et proc discrim), "MASS / R" affiche les coefficients des fonctions canoniques avec lda(), pour une projection dans l'espace factoriel, mais produit les prédictions et les probabilités d'affectation avec predict(). Le lien entre les deux n'est pas toujours évident et, à dire vrai, déroute un peu souvent mes étudiants.

Dans ce document, nous reprenons la trame de la présentation pour Python (TUTO 1) en prenant appui toujours sur notre support de cours dédié (COURS 1), mais en l'adaptant bien sûr aux spécificités de lda(). Notre référence reste SAS, avec les deux procédures précitées. Un des enjeux fort sera de faire le lien entre les parties descriptives et prédictives en dérivant les fonctions de classement (classification functions) à partir des fonctions canoniques discriminantes (canonical discrimant functions).

2 Analyse discriminante linéaire (ADL) avec lda()

2.1 Importation des données

Nous utilisons les données de notre précédent tutoriel (TUTO 1, section 2.1). Il s'agit d'identifier la nature d'eaux de vie {KIRSCH, MIRAB, POIRE} à partir de leur composition (butanol, méthanol, etc. ; 8 variables). Nous disposons de 52 observations en apprentissage, 50 en test.

Nous chargeons la première feuille “TRAIN” du classeur “**Eau_de_vie_LDA_R.xlsx**”.

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

#chargement et inspection des données d'apprentissage (TRAIN)
library(xlsx)
DTrain <- read.xlsx("Eau_de_vie_LDA_R.xlsx",header = TRUE, sheetIndex = 1)
str(DTrain)

## 'data.frame':    52 obs. of  9 variables:
##  $ TYPE: Factor w/ 3 levels "KIRSCH","MIRAB",..: 1 1 1 1 1 1 1 1 1 ...
##  $ MEOH: num  336 442 373 418 84 ...
##  $ ACET: num  225 338 356 62 65 ...
##  $ BU1 : num  1 1.9 0 0.8 2 2.2 1.9 0.4 0.9 0.8 ...
##  $ BU2 : num  1 10 29 0 2 52.1 46 3 36 12 ...
##  $ ISOP: num  92 91 83 89 2 123 85 6 84 7 ...
##  $ MEPR: num  37 30 27 24 0 38.2 33 9 36 9 ...
##  $ PRO1: num  177 552 814 342 288 ...
##  $ ACAL: num  0 31 11 7 6 13.3 35 4 4.8 2 ...
```

Nous préparons les structures pour la suite de l’étude.

```
#X
XTrain <- DTrain[-1]
print(str(XTrain))

## 'data.frame':    52 obs. of  8 variables:
##  $ MEOH: num  336 442 373 418 84 ...
##  $ ACET: num  225 338 356 62 65 ...
##  $ BU1 : num  1 1.9 0 0.8 2 2.2 1.9 0.4 0.9 0.8 ...
##  $ BU2 : num  1 10 29 0 2 52.1 46 3 36 12 ...
##  $ ISOP: num  92 91 83 89 2 123 85 6 84 7 ...
##  $ MEPR: num  37 30 27 24 0 38.2 33 9 36 9 ...
##  $ PRO1: num  177 552 814 342 288 ...
##  $ ACAL: num  0 31 11 7 6 13.3 35 4 4.8 2 ...
## NULL

#y
yTrain <- DTrain$TYPE

#nombre de variables
p <- ncol(XTrain)

#nombre d'observations
n <- nrow(DTrain)

#nombre de classes
K <- nlevels(yTrain)
```

Dans ce qui suit, **n = 52** correspond au nombre d’observations disponibles pour les calculs, **p = 8** est le nombre de variables explicatives ; **K = 3** est le nombre de classes {KIRSCH, MIRAB, POIRE}.

Nous affichons les fréquences relatives des classes.

```
#distribution
print(prop.table(table(yTrain)))

## yTrain
##    KIRSCH     MIRAB     POIRE
## 0.3269231 0.2884615 0.3846154
```

Elles sont relativement équilibrées.

2.2 Modélisation - Lecture des résultats

Pour lancer l'analyse discriminante, nous chargeons le package "MASS" qui est installé par défaut. Puis nous faisons appel à la fonction lda : `(TYPE ~ .)` signifie que nous expliquons "TYPE" à partir des autres variables disponibles dans la base. Nous affichons les résultats.

```
#lda - analyse discriminante linéaire
library(MASS)
mLda <- lda(TYPE ~ ., data = DTrain)
print(mLda)

## Call:
## lda(TYPE ~ ., data = DTrain)
##
## Prior probabilities of groups:
##    KIRSCH     MIRAB     POIRE
## 0.3269231 0.2884615 0.3846154
##
## Group means:
##             MEOH     ACET   BU1      BU2      ISOP     MEPR     PRO1     ACAL
## KIRSCH  371.6765 203.0176  1.20 21.01765  81.58824 28.89412 790.7706 12.01176
## MIRAB   934.2000 235.0667 20.20 13.56667  90.93333 29.40000 195.2667 12.35333
## POIRE  1084.3500 185.2500 21.33 49.38000 118.05000 50.00000 317.4000 14.49500
##
## Coefficients of linear discriminants:
##                LD1           LD2
## MEOH  0.0050393379 -0.0002002615
## ACET  0.0006684062 -0.0045426042
## BU1   0.0717648107 -0.0740663683
## BU2   0.0180727292  0.0126223695
## ISOP -0.0035288557 -0.0166297343
## MEPR -0.0058194132  0.0935069344
## PRO1 -0.0016560787 -0.0006355886
## ACAL -0.0396500855  0.0637142639
##
## Proportion of trace:
##    LD1    LD2
## 0.9037 0.0963
```

Ce n'est pas vraiment ce que l'on a l'habitude de voir dans une analyse discriminante prédictive (pour SAS par ex., TUTO 2, section 4.1 ; pour TANAGRA, TUTO 2, section 3.2). Nous distinguons dans les sorties ci-dessus :

- Les distributions relatives des classes (Prior Probabilities of groups).
- Les moyennes conditionnelles des variables (Group Means).
- Les coefficients des fonctions canoniques discriminantes (Coefficients of linear discriminants). Contrairement aux fonctions de classement (TUTO 1, Figure 1), leur nombre n'est pas égal au nombre de classes K, mais à min(K-1,p) = min(3-1,8) = 2.
- La part d'explication portée par les axes (Proportion of trace).

En réalité, nous sommes plutôt dans une optique d'analyse factorielle discriminante (COURS 2), descriptive donc (pour TANAGRA, voir par exemple TUTO 2, secion 3.4). Nous reprenons ci-dessous les sorties de la PROC CANDISC de SAS sur les mêmes données, via les commandes suivantes :

```
PROC CANDISC DATA = EAU_DE_VIE;
      class TYPE;
      VAR MEOH ACET BU1 BU2 ISOP MEPR PRO1 ACAL;
RUN;
```

Le tableau "Coefficients of linear discriminants" correspond aux "Coefficients canoniques bruts" dans les sorties de SAS (Figure 1) :

Coefficients canoniques bruts			
Variable	Libellé	Can1	Can2
MEOH	MEOH	0.0050393379	-.0002002615
ACET	ACET	0.0006684062	-.0045426042
BU1	BU1	0.0717648107	-.0740663683
BU2	BU2	0.0180727292	0.0126223695
ISOP	ISOP	-.0035288557	-.0166297343
MEPR	MEPR	-.0058194132	0.0935069344
PRO1	PRO1	-.0016560787	-.0006355886
ACAL	ACAL	-.0396500855	0.0637142639

Figure 1 - Coefficients canoniques bruts - SAS

Il manque deux informations importantes dans les sorties de lda(). Nous essaierons de les reconstruire plus loin :

- Puisque les fonctions discriminantes s'appliquent sur les variables originelles (non-standardisées), les équations devraient comporter une constante. Il faut les calculer pour pouvoir réellement utiliser ces fonctions pour projeter les individus dans le repère factoriel.

- Les moyennes conditionnelles des classes dans le repère factoriel jouent un rôle important en déploiement. Il faut pouvoir en disposer, si possible en s'épargnant le passage par le calcul des coordonnées des individus de l'échantillon d'apprentissage dans le nouveau repère.

Auparavant, regardons en détail les propriétés de l'objet. Peut-être que certaines répondent directement à ces questions.

2.3 Inspection de l'objet lda()

Nous affichons la liste des propriétés de l'objet avec la commande attributes().

```
#affichage des propriétés
print(attributes(mLda))

## $names
##  [1] "prior"   "counts"  "means"   "scaling" "lev"     "svd"     "N"
##  [8] "call"    "terms"   "xlevels"
##
## $class
## [1] "lda"
```

Voyons celles qui sont en rapport avec les calculs que nous effectuerons par la suite :

- $prior correspond à la distribution relative des classes.

```
#distribution des classes
print(mLda$prior)

##    KIRSCH     MIRAB     POIRE
## 0.3269231 0.2884615 0.3846154
```

- $counts à la distribution absolue.

```
#distribution absolue des classes
print(mLda$counts)

## KIRSCH  MIRAB  POIRE
##     17     15     20
```

- $means indique les moyennes des variables conditionnellement aux classes.

```
#moyennes conditionnes des variables
print(mLda$means)

##             MEOH     ACET   BU1      BU2      ISOP     MEPR     PRO1     ACAL
## KIRSCH  371.6765 203.0176  1.20 21.01765  81.58824 28.89412 790.7706 12.01176
## MIRAB   934.2000 235.0667 20.20 13.56667  90.93333 29.40000 195.2667 12.35333
## POIRE  1084.3500 185.2500 21.33 49.38000 118.05000 50.00000 317.4000 14.49500
```

- $scaling est la matrice des coefficients des fonctions discriminantes.

```
#coefficients des fonctions discriminantes
print(mLda$scaling)

##                LD1           LD2
## MEOH  0.0050393379 -0.0002002615
## ACET  0.0006684062 -0.0045426042
## BU1   0.0717648107 -0.0740663683
## BU2   0.0180727292  0.0126223695
## ISOP -0.0035288557 -0.0166297343
## MEPR -0.0058194132  0.0935069344
## PRO1 -0.0016560787 -0.0006355886
## ACAL -0.0396500855  0.0637142639
```

- $lev énumère les libéllés des classes.

```
#liste des classes
mLda$lev

## [1] "KIRSCH" "MIRAB"  "POIRE"
```

2.4 Constantes des fonctions canoniques

Pour obtenir les constantes de chaque fonction canonique, nous devons tout d'abord calculer les moyennes (marginales) des variables.

```
#moyennes des variables
xb <- colMeans(DTrain[-1])
print(xb)

##      MEOH      ACET       BU1       BU2      ISOP      MEPR      PRO1      ACAL
## 808.04808 205.42885  14.42308  29.77692  98.30769  37.15769 436.92500  13.06538
```

Puis nous appliquons les coefficients sur ces moyennes (COURS 2, page 15). Nous obtenons :

```
#constantes des fonctions canoniques
const_ <- apply(mLda$scaling,2,function(v){-sum(v*xb)})
print(const_)

##        LD1        LD2
## -3.9777814 -0.6070055
```

Nous constaterons plus loin (section 4.3.2) qu'il est dès lors possible d'obtenir les coordonnées factorielles des individus en appliquant les coefficients sur la description brute (variables non-transformées, ni centrées, encore moins réduites) d'un individu et en y rajoutant ces constantes.

2.5 Calcul des moyennes conditionnelles

Appliquer ces coefficients sur les moyennes conditionnelles dans l'espace originel permet d'obtenir les coordonnées factorielles des barycentres conditionnels.

```
#avec les fonctions canoniques
#on retrouve les moyennes conditionnelles
```

```
cond_Xb <- sapply(1:ncol(mLda$scaling),function(j)
  {sapply(mLda$lev,
          function(niveau){sum(mLda$scaling[,j]*mLda$means[niveau,]) + const_[j]})
  })
colnames(cond_Xb) <- paste("LD",1:ncol(mLda$scaling),sep="")
rownames(cond_Xb) <- mLda$lev

#affichage
print(cond_Xb)

##               LD1        LD2
## KIRSCH -3.745022  0.1804718
## MIRAB   1.276755 -1.2869207
## POIRE   2.225702  0.8117896
```

Nous constatons que l'adjonction de la constante était essentielle. Nous retrouvons bien les valeurs du tableau "Moyennes des classes sur les variables canoniques" de SAS (Figure 2) :

Moyennes de classes sur les variables canoniques		
TYPE	Can1	Can2
KIRSCH	-3.745021600	0.180471766
MIRAB	1.276754873	-1.286920748
POIRE	2.225702205	0.811789560

Figure 2 - Moyennes des classes dans le repère factoriel - SAS

2.6 Construction de la fonction de classement

Tout ça est très joli, mais nous sommes loin de la notion de fonction de classement, essentielle en analyse prédictive c.-à-d. un modèle qui, appliqué aux données, permet d'obtenir directement une valeur égale ou proportionnelle à la probabilité d'appartenance aux classes (COURS 1, page 6). Nous avions vu que "scikit-learn" pour Python savait les générer directement, il en est de même pour SAS (TUTO 1, section 2.3). Est-ce qu'il est possible de dériver quelque chose d'approchant à partir des éléments de lda() ?

J'aborde la question dans mon support de cours sur l'analyse factorielle discriminante (COURS 2, pages 29 et 30). Une formule était avancée dans le tutoriel consacré à la comparaison des outils (TUTO 2, sections 3.4.7 et 3.4.8). Essayons de mettre en œuvre ces idées sous R.

2.6.1 Coefficients associés aux variables

Nous nous servons de deux outils principalement : les coefficients des fonctions discrimantes, et les moyennes conditionnelles dans le repère factoriel. Nous devons disposer d'une fonction de classement (Classification function) par classe à la sortie.

```
#coefficients des fonctions de classement
#déduits des fonctions canoniques
```

```
coef_ <- sapply(mLda$lev,function(niveau)
  {rowSums(sapply(1:ncol(mLda$scaling),
          function(j){mLda$scaling[,j]*cond_Xb[niveau,j]}))
  })

#affichage
print(coef_)

##            KIRSCH        MIRAB        POIRE
## MEOH -0.018908571  0.006691720  0.011053495
## ACET -0.003323008  0.006699362 -0.002199965
## BU1  -0.282127654  0.186943618  0.099600793
## BU2  -0.065404780  0.006830456  0.050471221
## ISOP  0.010214443  0.016895666 -0.021354027
## MEPR  0.038669190 -0.127765978  0.062955672
## PRO1  0.006087345 -0.001296454 -0.004201902
## ACAL  0.159989053 -0.132618648 -0.036526708
```

2.6.2 Constante de la fonction de classement

Le calcul de l'intercept nécessite les constantes des fonctions canoniques, des moyennes conditionnelles, et des probabilités a priori d'appartenance aux classes.

```
#intercept des fonctions de classement
intercept_ <- sapply(mLda$lev,function(niveau)
  {sum(mapply(prod,const_,cond_Xb[niveau,]))-0.5*sum(cond_Xb[niveau,]^2)+log(mLda$prior[niveau])})
names(intercept_) <- levels(yTrain)
print(intercept_)

##     KIRSCH      MIRAB      POIRE
##   6.640421  -7.183811 -13.108005
```

2.6.3 Remarque sur les résultats

Comme nous l'avions déjà noté dans la précédente étude, les paramètres estimés sont différents de ceux de SAS ou de TANAGRA (TUTO 1, Figure 1 et Figure 6). Mais les écarts entre les coefficients d'une variable sont les mêmes d'une fonction à l'autre (TUTO 2, section 3.4.7). En définitive, après transformation softmax, nous obtenons bien les mêmes probabilités d'affectation en déploiement sur les nouveaux individus (section 4.4.3). Les systèmes classent à l'identique avec la même confiance. Ils sont totalement équivalents.

3 Evaluation statistique de l'ADL

L'évaluation statistique du modèle manque totalement dans lda(). Heureusement, le langage R est suffisamment puissant pour que nous puissions écrire en quelques lignes le code destiné à évaluer facilement la pertinence globale du modèle via le Lambda de Wilks d'une part, et la pertinence individuelle des variables dans le modèle d'autre part.

3.1 Evaluation globale - Lambda de Wilks

Nous calculons tout à tour les matrices de covariance totale…

```
#matrice de convariance totale
TOT <- cov(XTrain)
print(TOT)

##              MEOH        ACET         BU1          BU2         ISOP       MEPR
## MEOH 136738.7084  6617.58388  3173.90671   372.960935  7655.13198 3593.54913
## ACET   6617.5839 14921.54366   -99.42107  -146.492066    74.83605   51.90673
## BU1    3173.9067   -99.42107   121.60769    85.720347   182.12021   79.10727
## BU2     372.9609  -146.49207    85.72035  2968.665732  -178.39864   71.99508
## ISOP   7655.1320    74.83605   182.12021  -178.398643  2333.86425  754.02896
## MEPR   3593.5491    51.90673    79.10727    71.995083   754.02896  341.19896
## PRO1 -65773.7620 12047.59417 -2032.14176 22370.955686 -3366.73137 -731.99343
## ACAL   1082.7291   427.86337    10.52827    -4.595913    32.95988   14.03772
##              PRO1        ACAL
## MEOH -65773.7620 1082.729148
## ACET  12047.5942  427.863371
## BU1   -2032.1418   10.528265
## BU2   22370.9557   -4.595913
## ISOP  -3366.7314   32.959879
## MEPR   -731.9934   14.037722
## PRO1 392100.2023  626.809510
## ACAL    626.8095   64.872504
```

… et intra-classe (TUTO 1, section 2.4.1).

```
#WITHIN - covariance intra-classe
WIT <- (1.0/(n-
K))*Reduce("+",lapply(levels(yTrain),function(niveau){(sum(yTrain==niveau)-
1)*cov(XTrain[yTrain==niveau,])}))
print(WIT)

##              MEOH       ACET         BU1          BU2         ISOP       MEPR
## MEOH 40223.7025  7653.7918  299.526327 -2522.67577  3494.67521 1340.44526
## ACET  7653.7918 15093.4728 -110.066327   148.72976   293.41307  223.26922
## BU1    299.5263  -110.0663   36.221265    22.43800    70.23612   21.94082
## BU2  -2522.6758   148.7298   22.438000  2825.92714  -431.04673  -91.42956
## ISOP  3494.6752   293.4131   70.236122  -431.04673  2156.40818  615.87467
## MEPR  1340.4453   223.2692   21.940816   -91.42956   615.87467  245.69489
## PRO1  7923.7079 14043.4170  272.533878 24116.53208 -1034.01305  305.21851
## ACAL   833.6814   462.6808    3.353327   -22.95753    15.06614    2.40533
##              PRO1       ACAL
## MEOH   7923.7079 833.681423
## ACET  14043.4170 462.680778
## BU1     272.5339   3.353327
## BU2   24116.5321 -22.957528
## ISOP  -1034.0130  15.066136
## MEPR    305.2185   2.405330
## PRO1 340956.9520 798.808215
## ACAL    798.8082  66.145806
```

Nous déduisons le lambda de Wilks par le rapport des déterminants (TUTO 1, section 2.4.2).

```
#lambda de Wilks
#matrices intermédiaires
WITprim <- (n-K)/n*WIT
TOTprim <- (n-1)/n*TOT

#rapport des déterminants -- Lambda de Wilks
LW <- det(WITprim)/det(TOTprim)
print(LW)

## [1] 0.06671324
```

Nous pouvons aisément (la formule est alambiquée, mais le programme est simple) déduire la statistique de Rao pour tester la significativité globale du modèle. Je laisse aux férrus de programmation R le soin de transposer le code Python (TUTO 1, section 2.4.3).

3.2 Evaluation des variables - Test F

Pour évaluer la pertinence de chaque variable (TUTO 1, section 2.5), le code R est encore plus simple que celui de Python, et surtout nous nous en inspirerons pour la sélection backward de variables plus loin (section 5.2).

```
#
# *** contribution des variables ***
#

#ddl numérateur
ddlSuppNum <- K - 1

#ddl dénominateur
ddlSuppDenom <- n - K - p + 1

#vecteur des F
FTest <- numeric(p)

#p-value
pvalueFTest <- numeric(p)

#boucle
for (j in 1:p){
  #Lambda corresp.
  LWvar <- det(WITprim[-j,-j])/det(TOTprim[-j,-j])
  #print(LWvar)
  #F
  FTest[j] <- ddlSuppDenom / ddlSuppNum * (LWvar/LW - 1)
  #p-value
  pvalueFTest[j] <- pf(FTest[j],ddlSuppNum,ddlSuppDenom,lower.tail=FALSE)
}

#affichage
temp <- data.frame(var=colnames(XTrain),FValue=FTest,pvalue=round(pvalueFTest,6))
print(temp)
```

```
##    var    FValue   pvalue
## 1 MEOH 16.136067 0.000006
## 2 ACET  2.341965 0.108572
## 3  BU1  5.499262 0.007563
## 4  BU2  9.122996 0.000513
## 5 ISOP  1.761764 0.184196
## 6 MEPR  6.636945 0.003128
## 7 PRO1  8.084336 0.001071
## 8 ACAL  2.886700 0.066885
```

Les résultats sont en tous points conformes à ceux de TANAGRA et SAS (TUTO 1, Figures 6 et 7).

4 Traitements sur l'échantillon test

Nous mettons à contribution l'échantillon test dans cette section pour inspecter les propriétés des sorties de predict() appliquée à un objet lda().

4.1 Chargement des données

Nous chargeons la feuille TEST de notre classeur Excel. Nous préparons les structures idoines.

```
#chargement de l'échantillon test
DTest <- read.xlsx("Eau_de_vie_LDA_R.xlsx",header = TRUE, sheetIndex = 2)
str(DTest)

## 'data.frame':    50 obs. of  9 variables:
##  $ TYPE: Factor w/ 3 levels "KIRSCH","MIRAB",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ MEOH: num  3 475 186 371 583 0 421 557 167 523 ...
##  $ ACET: num  15 172 101 414 226 25 142 447 86 367 ...
##  $ BU1 : num  0.2 1.9 0 1.2 2.3 0.1 1.6 0 0 2.6 ...
##  $ BU2 : num  30 7 1.6 0 19 8 8 34 0 30 ...
##  $ ISOP: num  9 113 36 97 120 0 75 107 32 116 ...
##  $ MEPR: num  9 33 11 39 46 6 24 39 10 45 ...
##  $ PRO1: num  350 546 128 502 656 253 128 162 114 787 ...
##  $ ACAL: num  9 14 8 9 11 7 31 94 8 25 ...

#X - matrice des explicatives en test
XTest <- DTest[-1]

#y - vecteur cible en test
yTest <- DTest$TYPE
```

4.2 Utilisation de la fonction predict()

Nous appliquons predict() sur ces nouvelles données. L'objet – c'est une liste – qui en est issu possède plusieurs propriétés.

```
#prediction en test
predTest <- predict(mLda,newdata=XTest)
```

```
#propriétés de l'objet
print(attributes(predTest))

## $names
## [1] "class"     "posterior" "x"
```

4.3 Projection dans l'espace factoriel

4.3.1 Valeurs restituées par l'objet

La propriété $x correspond aux coordonnées factorielles des individus en test. Nous affichons les premières observations.

```
#coordonnées factorielles
head(predTest$x, 5)

##         LD1        LD2
## 1 -4.416715  0.7309797
## 2 -3.256389  0.2157101
## 3 -3.664270 -0.2245967
## 4 -3.502808 -0.3627711
## 5 -2.594041  0.9087476
```

La projection dans le plan est naturelle puisque nous avons 2 dimensions. Nous veillons à ce que les échelles en abscisse et ordonnée soient identiques (eqsplot).

```
#graphique - même échelle en abscisse et ordonnée
eqscplot(predTest$x[,1],predTest$x[,2],col=c("red","green","blue")[yTest],xlab="LD1",ylab="LD2")
legend("topright",legend=levels(yTest),text.col=c("red","green","blue"))
abline(h=0,col="gray")
abline(v=0,col="gray")
```

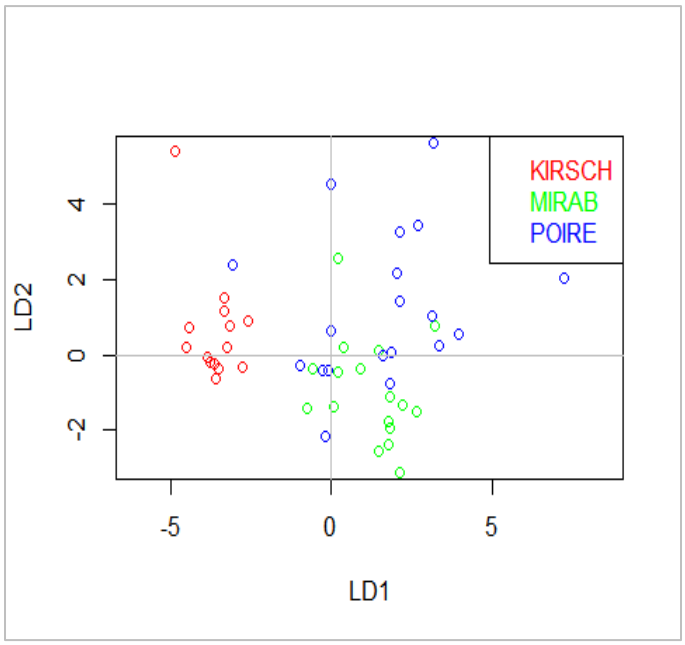

Figure 3 - Projection des individus en test dans le repère factoriel

KIRSCH est assez bien discriminé. Il y a un empiètement en revanche entre MIRAB et POIRE. On sait d'où viendra l'erreur en classement.

4.3.2 Calcul des coordonnées à partir des fonctions canoniques

Puisque nous disposons des descriptions des individus (XTest) et des coefficients des fonctions canoniques de projection. Nous devrions pouvoir reconstituer facilement ces coordonnées factorielles avec un produit matriciel. Vérifions cela.

```
#produit matriciel
coordFact <- t(t(mLda$scaling) %*% t(as.matrix(XTest)))

#sans oublier d'ajouter les constantes !
coordFact <- sapply(1:ncol(mLda$scaling),function(j){coordFact[,j] + const_[j]})

#affichage
head(coordFact, 5)

##           [,1]       [,2]
## [1,] -4.416715  0.7309797
## [2,] -3.256389  0.2157101
## [3,] -3.664270 -0.2245967
## [4,] -3.502808 -0.3627711
## [5,] -2.594041  0.9087476
```

Et nous avons bien les mêmes valeurs ! Nous constatons au passage qu'il fallait absolument disposer de - et utiliser - la constante pour produire les coordonnées correctes.

4.4 Calcul des probabilités d'appartenance aux groupes

4.4.1 Valeurs restituées par l'objet

La propriété $posterior restitue les probabilités d'appartenance aux groupes des individus.

```
#proba a posteriori - proba d'affectation
head(predTest$posterior)

##      KIRSCH        MIRAB        POIRE
## 1 1.0000000 1.535523e-08 4.487814e-10
## 2 0.9999886 1.109717e-05 3.307808e-07
## 3 0.9999972 2.730580e-06 2.193638e-08
## 4 0.9999924 7.523981e-06 5.271811e-08
## 5 0.9998616 1.116948e-04 2.673034e-05
## 6 1.0000000 1.745132e-08 1.549001e-10
```

Elles sont utiles pour le classement mais aussi pour le scoring. La manière de les obtenir à partir des résultats de lda() n'est pas évidente de prime abord. Je suis allé voir le code source du package pour m'en assurer.

4.4.2 Valeurs à partir des coordonnées factorielles

Il apparaît (dans le fichier "**lda.R**" du package) que MASS utilise les distances aux barycentres conditionnels pour calculer cette probabilité, en s'appuyant sur une formule que j'avais moi-même identifié en lisant la documentation de SAS (TUTO 2, section 3.4.5). L'outil utilise une distance généralisée c.-à-d. corrigée par les probabilités a priori d'appartenance aux groupes.

```
#distance carrée généralisée aux centres de classes dans le repère factoriel
d2G <- sapply(mLda$lev,
              function(niveau)
              {rowSums(scale(predTest$x,center=cond_Xb[niveau,],scale=FALSE)^2)-2*log(mLda$prior[niveau])})
head(d2G)

##      KIRSCH    MIRAB    POIRE
## 1 2.990292 38.97391 46.03926
## 2 2.476064 25.29368 32.31966
## 3 2.406662 28.02865 37.67690
## 4 2.589841 26.18466 36.10644
## 5 4.091203 22.29041 25.15035
## 6 2.870776 38.59848 48.04726
```

Pour les traduire en probabilités, nous appliquons une transformation *inspirée* de softmax.

```
#variante softmax avec la correction -- cf. doc de SAS
pClassD2G <- t(apply(d2G,1,function(ligne){exp(-0.5*ligne)/sum(exp(-0.5*ligne))}))
head(pClassD2G)

##      KIRSCH        MIRAB        POIRE
## 1 1.0000000 1.535523e-08 4.487814e-10
## 2 0.9999886 1.109717e-05 3.307808e-07
## 3 0.9999972 2.730580e-06 2.193638e-08
## 4 0.9999924 7.523981e-06 5.271811e-08
## 5 0.9998616 1.116948e-04 2.673034e-05
## 6 1.0000000 1.745132e-08 1.549001e-10
```

Et nous retrouvons les valeurs de la propriété $posterior de la prédiction lda(). Nous pouvons procéder à une comparaison globale pour nous en assurer.

```
#vérification globale
#somme écart absolu entre les probabilités en 2 façons
print(sum(abs(predTest$posterior-pClassD2G)))

## [1] 6.355273e-15
```

4.4.3 Valeurs à partir des fonctions de classement

Bien, nous avons pu reconstituer les calculs de lda(). Mais n'y avait-il pas une autre manière de faire ? Nous avons dérivé des fonctions de classement à partir des fonctions canoniques plus haut (section 2.6). Nous savons qu'elles sont (devraient être) équivalentes en prédiction. C'est le moment de le vérifier.

Nous calculons les “scores” de classement des individus via les fonctions de classement.

```
#application fonction de classement sur les individus en test
#avec la description originelle, sans transformation préalable
#ni projection dans l'espace factoriel
vfClass <- t(t(coef_) %*% t(as.matrix(XTest)))

#rajouter la constante
vfClass <- sapply(mLda$lev,function(niveau){vfClass[,niveau]+intercept_[niveau]})
head(vfClass)

##        KIRSCH     MIRAB      POIRE
## [1,] 8.525706 -9.466103 -12.998779
## [2,] 4.087269 -7.321540 -10.834529
## [3,] 5.535329 -7.275664 -12.099788
## [4,] 4.905711 -6.891696 -11.852588
## [5,] 1.731834 -7.367768  -8.797739
## [6,] 8.897932 -8.965919 -13.690309
```

Nous les traduisons en probabilités avec la transformation softmax usuelle.

```
#softmax
pClass <- t(apply(vfClass,1,function(ligne){exp(ligne)/sum(exp(ligne))}))
head(pClass)

##         KIRSCH        MIRAB        POIRE
## [1,] 1.0000000 1.535523e-08 4.487814e-10
## [2,] 0.9999886 1.109717e-05 3.307808e-07
## [3,] 0.9999972 2.730580e-06 2.193638e-08
## [4,] 0.9999924 7.523981e-06 5.271811e-08
## [5,] 0.9998616 1.116948e-04 2.673034e-05
## [6,] 1.0000000 1.745132e-08 1.549001e-10
```

Vérifions la concordance sur l’ensemble des individus…

```
#vérification globale
print(sum(abs(predTest$posterior-pClass)))

## [1] 1.489855e-14
```

Les résultats sont cohérents. Moralité : nous avons extraits des fonctions de classement à partir des résultats de la lda(), équivalent à ce que l’on peut trouver dans d’autres outils tels que SAS ; et elles ont un comportement identique en prédiction. C’est tout bonus.

Je ne l’ai pas fait parce que ce n’est pas notre propos dans ce tutoriel. Mais j’imagine qu’en déploiement sur de très grandes volumétries, un produit matriciel devrait être nettement plus rapide que des calculs de distances. A vérifier.

4.5 Prédiction brute et mesure des performances

Le champ $class correspond aux classes prédites en déploiement.

```
#prédiction brute
print(predTest$class)

##  [1] KIRSCH KIRSCH KIRSCH KIRSCH KIRSCH KIRSCH KIRSCH KIRSCH KIRSCH KIRSCH
## [11] KIRSCH KIRSCH KIRSCH KIRSCH POIRE  MIRAB  MIRAB  MIRAB  MIRAB  MIRAB
## [21] MIRAB  MIRAB  MIRAB  MIRAB  MIRAB  MIRAB  MIRAB  POIRE  MIRAB  MIRAB
## [31] POIRE  MIRAB  MIRAB  MIRAB  POIRE  KIRSCH POIRE  POIRE  MIRAB  POIRE
## [41] POIRE  POIRE  POIRE  POIRE  POIRE  POIRE  POIRE  MIRAB  POIRE  POIRE
## Levels: KIRSCH MIRAB POIRE

#distribution des prédiction
print(table(predTest$class))

##
## KIRSCH  MIRAB  POIRE
##     15     19     16
```

Sur notre échnatillon test, 15 observations ont été classées KIRSCH, 19 MIRAB et 16 POIRE.

Via la confrontation entre les classes observées et prédites, nous avons la matrice de confusion.

```
#matrice de confusion
mc <- table(yTest,predTest$class)
print(mc)

##
## yTest    KIRSCH MIRAB POIRE
##   KIRSCH     14     0     0
##   MIRAB       0    14     3
##   POIRE       1     5    13
```

A partir de laquelle nous pouvons dériver le taux de reconnaissance (taux de succès), …

```
#taux de reconnaissance (accuracy)
accTest <- sum(diag(mc))/sum(mc)
print(accTest)

## [1] 0.82
```

… le taux d'erreur, ….

```
#taux d'erreur
errTest <- 1 - accTest
print(errTest)

## [1] 0.18
```

…, la sensibilité (rappel) par classe, ….

```
#rappel (sensibilité par classe)
sensTest <- diag(mc)/rowSums(mc)
print(sensTest)

##    KIRSCH     MIRAB     POIRE
## 1.0000000 0.8235294 0.6842105
```

…, la précision par classe.

```
#précision par classe
precisionTest <- diag(mc)/colSums(mc)
print(precisionTest)

##    KIRSCH     MIRAB     POIRE
## 0.9333333 0.7368421 0.8125000
```

Pour les impatients, la fonction confusion_matrix() du package "caret" nous fait le tout en une seule commande.

```
#on a la même chose avec caret
library(caret)

## Loading required package: lattice

## Loading required package: ggplot2

#appel de confusion matrix
caret::confusionMatrix(reference=yTest,data=predTest$class)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction KIRSCH MIRAB POIRE
##     KIRSCH     14     0     1
##     MIRAB       0    14     5
##     POIRE       0     3    13
##
## Overall Statistics
##
##                Accuracy : 0.82
##                  95% CI : (0.6856, 0.9142)
##     No Information Rate : 0.38
##     P-Value [Acc > NIR] : 2.298e-10
##
##                   Kappa : 0.7294
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: KIRSCH Class: MIRAB Class: POIRE
## Sensitivity                 1.0000       0.8235       0.6842
## Specificity                 0.9722       0.8485       0.9032
## Pos Pred Value              0.9333       0.7368       0.8125
## Neg Pred Value              1.0000       0.9032       0.8235
## Prevalence                  0.2800       0.3400       0.3800
## Detection Rate              0.2800       0.2800       0.2600
## Detection Prevalence        0.3000       0.3800       0.3200
## Balanced Accuracy           0.9861       0.8360       0.7937
```

5 Sélection de variables

Limiter le modèle aux variables explicatives pertinentes est primordial pour l'interprétation et le déploiement des modèles. Voyons ce que l'on peut faire en relation directe avec l'analyse discriminante linéaire sous R.

5.1 Sélection forward avec le package "klaR"

Le premier réflexe (je deviens comme les étudiants) est de chercher le package qui va bien sur le web. Ils ne sont pas légions ou ne sont pas convaincants pour l'analyse discriminante. Le seul qui produit des résultats conformes à ceux de SAS est "klaR". Mais il s'en tient seulement à la sélection "forward" avec la fonction greedy.wilks() (TUTO 2, section 5.5). Voyons son comportement sur nos données d'apprentissage avec un seuil d'acceptation de 1% (COURS 1, page 14).

```
library(klaR)
greedy.wilks(TYPE ~ ., data = DTrain, niveau = 0.01)

## Formula containing included variables:
##
## TYPE ~ MEOH + BU1 + MEPR
## <environment: 0x00000000709f30b0>
##
##
## Values calculated in each step of the selection procedure:
##
##   vars Wilks.lambda F.statistics.overall p.value.overall F.statistics.diff
## 1 MEOH    0.2826288             62.18613    3.587597e-14         62.186129
## 2  BU1    0.1925469             30.69441    1.883938e-16         11.228252
## 3 MEPR    0.1477860             25.08637    1.401659e-17          7.117602
##   p.value.diff
## 1 3.587597e-14
## 2 9.677890e-05
## 3 1.963012e-03
```

Wilks.lambda est la valeur du lambda de Wilks du modèle incluant les variables sélectionnées. La valeur est forcément décroissante. F.statistics.overall et p.value.overrall sont les transformations de Rao et les p-value correspondantes.

F.statistics.diff est la statistique du test de pertinence lorsqu'on rajoute la variable dans le modèle. On peut le voir comme une contribution à la décroissance du lambda global. Nous avons la p-value associée avec p.value.diff.

Le processus de sélection est stoppé lorsque la meilleure variable à une étape donnée présente une p.value.diff supérieure au seuil que l'on s'est choisi, qui est de 1% dans notre étude.

A titre de comparaison, voici (Figure 4) le tableau fourni par STEPDISC sous SAS avec les commandes suivantes :

```
PROC STEPDISC DATA = EAU_DE_VIE METHOD = forward SLENTRY= 0.01;
      class TYPE;
      VAR MEOH ACET BU1 BU2 ISOP MEPR PRO1 ACAL;
RUN;
```

Synthèse de la sélection ascendante

Etape	Nombre dans	Saisi	Libellé	R carré partiel	Valeur F	Pr > F	Lambda de Wilk	Pr < Lambda	Corrélation canonique moyenne au carré	Pr > ASCC
1	1	MEOH	MEOH	0.7174	62.19	<.0001	0.28262884	<.0001	0.35868558	<.0001
2	2	BU1	BU1	0.3187	11.23	<.0001	0.19254694	<.0001	0.41786644	<.0001
3	3	MEPR	MEPR	0.2325	7.12	0.0020	0.14778601	<.0001	0.51868897	<.0001

Figure 4 - Sélection forward - SAS

SAS fournit les détails des opérations. A l'étape 4, nous disposons du tableau suivant (Figure 5) :

La procédure STEPDISC
Sélection ascendante : Etape 4

Statistiques pour l'entrée, DDL = 2, 46

Variable	Libellé	R carré partiel	Valeur F	Pr > F	Tolérance
ACET	ACET	0.0661	1.63	0.2076	0.2987
BU2	BU2	0.1103	2.85	0.0679	0.3256
ISOP	ISOP	0.1216	3.18	0.0507	0.2511
PRO1	PRO1	0.0759	1.89	0.1628	0.3300
ACAL	ACAL	0.1382	3.69	0.0327	0.2672

Figure 5 - Sélection forward - Etape 4 - SAS

La meilleure variable est ISOP parce qu'elle présente la p-value (Pr > F) la plus faible. Mais elle n'est pas intégrée parce que 0.0507 > 0.01 qui était notre seuil.

5.2 Sélection backward

Pour la sélection backward, après une recherche rapide, je n'ai pas trouvé de package convaincant. Je me suis dit qu'il était plus rapide de le programmer moi-même plutôt que de perdre son temps à inspecter des outils plus ou moins aboutis.

L'idée est de généraliser le code destiné à éprouver la pertinence des variables (test.retrait), en l'englobant dans une boucle permettant de les retirer petit à petit (sel.backward). Bon, j'ai écrit un code simple en allant à l'essentiel. Il manque certainement de nombreux contrôles, indispensables lorsqu'il s'agit de diffuser un programme.

```
#fonction pour le test du retrait d'une variable numéro j
#SW : matrice de covariance intra
#ST : matrice de covariance totale
#N : nombre d'observations
#K : nombre de classes
#precLW : valeur du lambda avant le retrait
test.retrait <- function(j,SW,ST,N,K,precLW){
  #nombre de variables à cette étape
  nbVar <- ncol(SW)
  #lamdba
  lambada <- det(SW[-j,-j])/det(ST[-j,-j])
  #ddl
  ddlNum <- K - 1
  ddlDenom <- N - K - nbVar + 1
  #f
  f <- ddlDenom / ddlNum * (lambada / precLW - 1)
  #p-value
  pvalue <- pf(f,ddlNum,ddlDenom,lower.tail=FALSE)
  #renvoyer sous forme de vecteur
  return(c(lambada,f,pvalue))
}

#fonction de sélection backward
#DW : matrice initiale de covariance intra
#DT : matrice initiale de covariance totale
#lambdaInit : valeur initiale du lambda de wilks
#alpha : risque pour piloter la sélection
sel.backward <- function(DW,DT,lstInit,lambdaInit,alpha=0.05){
  #matrice pour récupérer les résultats à chaque étape
  mResult <- matrix(0,nrow=0,ncol=3)
  colnames(mResult) <- c('Lambda','F-Test','p-value')
  #liste des variables retirées
  lstvarDel <- c()
  #******************
  #boucle de recherche
  repeat{
    #resultats d'une étape
    res <- t(sapply(1:ncol(DW),test.retrait,SW=DW,ST=DT,N=n,K=K,precLW=lambdaInit))
    #trouver la ligne qui maximise la p-value
    id <- which.max(res[,3])
    #vérifier si retrait (comparer pvalue à alpha)
    if (res[id,3] > alpha){
      #nom de la variable à retirer
      lstvarDel <- c(lstvarDel,lstInit[id])
      #rajouter la ligne de résultats (lambda,F-Test,p-value)
      mResult <- rbind(mResult,res[id,])
      #retirer
      lstInit <- lstInit[-id]
      #si plus de variables
      if (length(lstInit) == 0){
        #arrêt puisque plus de variables à retirer
        break
      } else {
        #retirer les colonnes des matrices
        DW <- DW[-id,-id]
```

```
        DT <- DT[-id,-id]
        #màj du lambda
        lambdaInit <- res[id,1]
      }
    } else {
      #plus de retrait parce que condition alpha
      break
    }
  }
  #resultats sous la forme d'un data frame
  resultats <- data.frame(var=lstvarDel,mResult)
  return(resultats)
}
```

Nous prenons comme point de départ les matrices de covariance intra-classe et totale, les noms de variables, la valeur initiale du lambda de Wilks lorsque toutes les variables sont utilisées. La fonction est paramétrée par le seuil (alpha) de retrait des variables.

L'idée de l'algorithme est de retirer au fur et à mesure les variables tant que la pire d'entre elles à chaque étape n'est pas pertinente c.-à-d. présente une p-value supérieure à alpha.

Nous lançons le processus.

```
#lancer la sélection
removedVar <- sel.backward(WITprim,TOTprim,colnames(XTrain),LW,alpha=0.01)
print(removedVar)

##    var     Lambda   F.Test   p.value
## 1 ISOP 0.07231005 1.761764 0.1841962
## 2 ACET 0.07982573 2.234643 0.1193160
## 3 ACAL 0.08648809 1.836148 0.1714373
```

Les variables ISOP, ACET et ACAL ont été retirées du modèle initial incluant la totalité des variables. Le lambda est forcément croissant ici.

A titre de comparaison, voici (Figure 6) le tableau produit par la commande STEPDISC suivante :

```
PROC STEPDISC DATA = EAU_DE_VIE METHOD = backward SLSTAY= 0.01;
       class TYPE;
       VAR MEOH ACET BU1 BU2 ISOP MEPR PRO1 ACAL;
RUN;
```

Synthèse de l'élimination descendante										
Etape	Nombre dans	Supprimé	Libellé	R carré partiel	Valeur F	Pr > F	Lambda de Wilk	Pr < Lambda	Corrélation canonique moyenne au carré	Pr > ASCC
0	8			.	.	.	0.06671324	<.0001	0.66062639	<.0001
1	7	ISOP	ISOP	0.0774	1.76	0.1842	0.07231005	<.0001	0.63868213	<.0001
2	6	ACET	ACET	0.0942	2.23	0.1193	0.07982573	<.0001	0.60879343	<.0001
3	5	ACAL	ACAL	0.0770	1.84	0.1714	0.08648809	<.0001	0.59635580	<.0001

Figure 6 - Sélection backward - SAS

Il est toujours plaisant de voir que quelques lignes de code sous R permettent de reproduire les résultats de SAS.

Avec les variables restantes (sélectionnées), nous pouvons lancer une nouvelle modélisation.

```
#variables restantes
resteVar <- setdiff(colnames(XTrain),removedVar$var)
print(resteVar)

## [1] "MEOH" "BU1"  "BU2"  "MEPR" "PRO1"

#créer une nouvelle formule
formule <- reformulate(resteVar,"TYPE")
print(formule)

## TYPE ~ MEOH + BU1 + BU2 + MEPR + PRO1

#lancer la lda
mLdaBack <- lda(formule,data=DTrain)
print(mLdaBack)

## Call:
## lda(formule, data = DTrain)
##
## Prior probabilities of groups:
##    KIRSCH     MIRAB     POIRE
## 0.3269231 0.2884615 0.3846154
##
## Group means:
##             MEOH   BU1      BU2      MEPR     PRO1
## KIRSCH  371.6765  1.20 21.01765 28.89412 790.7706
## MIRAB   934.2000 20.20 13.56667 29.40000 195.2667
## POIRE  1084.3500 21.33 49.38000 50.00000 317.4000
##
## Coefficients of linear discriminants:
##                LD1           LD2
## MEOH  0.004426822  0.0002284161
## BU1   0.070051119 -0.0858977210
## BU2   0.019825650  0.0160829498
## MEPR -0.009459813  0.0579211568
## PRO1 -0.001823728 -0.0007308973
##
## Proportion of trace:
```

```
##    LD1    LD2
## 0.9359 0.0641
```

6 Conclusion

L'objectif de ce tutoriel était de renouveller l'exploration de la fonction lda() du package MASS pour R. Il y avait un petit enjeu parce qu'on trouve énormément de documents sur le même thème sur le web. Répeter ce qui a été déjà bien fait par ailleurs a peu d'intérêt.

Mon idée, tout comme dans le tutoriel pour Python d'ailleurs (TUTO 1), était de prendre comme référence l'historique PROC DISCRIM de SAS, et de montrer dans quelle mesure il était possible d'obtenir des résultats de même nature en prenant comme point de départ les outputs natifs de la fonction lda(). Nous avons pu constater notamment qu'il était possible de dériver des fonctions de classement à partir des fonctions canoniques, et de pouvoir les utiliser directement pour le déploiement.

7 Références

(COURS 1) R. Rakotomalala, "Analyse Discriminante Linéaire".

(COURS 2) R. Rakotomalala, "Analyse Factorielle Discriminante", janvier 2011.

(TUTO 1) Tutoriel Tanagra, "Analyse Discriminante Linéaire sous Python", avril 2020.

(TUTO 2) Tutoriel Tanagra, "Analyse Discriminante Linéaire - Comparaisons de logiciels", juillet 2012.