1 Introduction

Pratique de l'analyse discriminante sous Python avec le package « *scikit-learn* ». Comparaison des résultats avec ceux de SAS (PROC DISCRIM) et TANAGRA.

L'analyse discriminante linéaire est une méthode prédictive où le modèle s'exprime sous la forme d'un système d'équations linéaires des variables explicatives. On parle de « fonctions de classement ». Je l'affectionne particulièrement : parce qu'elle est simple et robuste ; parce qu'elle sait directement appréhender les problèmes multi-classes (nombre de modalités de la variable cible > 2) ; parce qu'elle se prête à des variétés d'interprétations qui éclairent sous autant d'angles les mécanismes de l'apprentissage supervisé (probabiliste, géométrique, …), en ce sens elle fait le pont entre les cultures statistiques, machine learning et pattern recognition (reconnaissance de formes) ; parce qu'il existe de nombreuses bibliothèques de calcul efficaces (voir *TUTO*, juillet 2012), j'ai beaucoup moi-même travaillé sur l'optimisation de son implémentation (ex. « *Multithreading équilibré pour la discriminante* », juin 2013), etc. Le temps que je lui consacre dans mes enseignements fait partie de mes séances favorites. Je tiendrai très facilement des heures avec un nombre pourtant réduit de slides (voir *COURS*).

Dans ce tutoriel, nous étudierons le comportement de la classe de calcul *LinearDiscriminantAnalysis* du package « *scikit-learn* » pour Python. En faisant le tour de ses propriétés et méthodes, nous passerons en revue les différents aspects de la technique. Pour mieux situer la teneur des résultats, nous mettrons en parallèle les sorties de la procédure PROC DISCRIM du logiciel SAS qui fait figure de référence dans le domaine.

2 LDA avec scikit-learn

Le site de « scikit-learn » consacre une *notice méthodologique* à l'analyse discriminante. On y apprend notamment que la librairie propose plusieurs approches (estimation algorithms) pour l'estimation des paramètres (des coefficients de la fonction de classement) et qu'il est possible d'introduire une régularisation (shrinkage) lorsque le ratio entre les nombres d'observations et

de variables explicatives est défavorable. Pour rapprocher au mieux nos résultats avec ceux de SAS, nous avons choisi l'algorithme « solver = eigen » qui implique le calcul explicite de la matrice de variance covariance dont nous aurons besoin dans les calculs subséquents.

2.1 Importation et inspection des données d'apprentissage

Nous utilisons une variante des données de notre précédent tutoriel (TUTO, section 2). Il s'agit de prédire de TYPE d'alcool (KIRSCH, MIRAB, POIRE) à partir de ses composants (butanol, méthanol, etc. ; 8 variables). Dans le classeur « **Eau_de_vie_LDA_python.xlsx** », la première feuille « TRAIN » correspond à l'ensemble d'apprentissage (52 observations), la seconde « TEST » au test (50 obs.).

```
#changement de dossier
import os
os.chdir("... votre dossier de travail ...")

#importation de l'échantillon train
import pandas
DTrain = pandas.read_excel("Eau_de_vie_LDA_python.xlsx",sheet_name="TRAIN")
print(DTrain.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 52 entries, 0 to 51
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   TYPE    52 non-null     object
 1   MEOH    52 non-null     float64
 2   ACET    52 non-null     float64
 3   BU1     52 non-null     float64
 4   BU2     52 non-null     float64
 5   ISOP    52 non-null     int64
 6   MEPR    52 non-null     float64
 7   PRO1    52 non-null     float64
 8   ACAL    52 non-null     float64
dtypes: float64(7), int64(1), object(1)
memory usage: 3.8+ KB
```

Nous créons les structures intermédiaires qui nous serviront dans nos calculs par la suite.

```
#préparation des structures – vecteur cible
yTrain = DTrain.iloc[:,0]

#préparation des structures – matrice des explicatives
XTrain = DTrain.iloc[:,1:]

#effectif -- 52
n = DTrain.shape[0]
```

```
#nombre de descripteurs -- 8
p = XTrain.shape[1]

#nombre de classes -- 3
K = len(yTrain.value_counts())
```

Nous calculons la distribution relative des classes, qui sont assez équilibrées.

```
#distribution des classes
print(yTrain.value_counts(normalize=True))
```

```
POIRE     0.384615
KIRSCH    0.326923
MIRAB     0.288462
Name: TYPE, dtype: float64
```

2.2 La PROC DISCRIM de SAS

Par sa richesse et son antériorité, la PROC DISCRIM de SAS fait figure de référence en matière d'analyse discriminante prédictive. Nous y avons fait appel avec les paramètres suivants :

```
PROC DISCRIM DATA = EAU_DE_VIE PCOV TCOV MANOVA;
      class TYPE;
      VAR MEOH ACET BU1 BU2 ISOP MEPR PRO1 ACAL;
      PRIORS proportional;
RUN;
```

« class » et « VAR » spécifient les variables cibles et explicatives. Nous demandons la production des matrices de variance covariance intra-classes (PCOV) et totales (TCOV). Le test de significativité globale est calculé et affiché (*MANOVA*). Enfin, les prévalences des classes sont estimées à partir des données (PRIORS). Nous reprendrons les résultats de la PROC DISCRIM au fur et à mesure que nous présenterons les sorties de « scikit-learn ».

2.3 Modélisation avec LinearDiscriminantAnalysis

Sage précaution avec les packages pour Python, nous affichons le numéro de la version de « scikit-learn » utilisée durant ce tutoriel. Nous fonctionnons avec la « **0.22.2.post1** ».

```
#version
import sklearn
print(sklearn.__version__)
```

```
0.22.2.post1
```

Comme usuellement avec les objets de « scikit-learn », nous instancions la classe de calcul puis nous faisons appel à la méthode fit() en lui passant les données à traiter. Nous énumérons les membres de l'objet.

```
#importation
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

#instanciation
lda = LinearDiscriminantAnalysis(solver="eigen",store_covariance=True)

#apprentissage
lda.fit(XTrain,yTrain)

#liste des champs
print(dir(lda))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_estimator_type',
'_get_param_names', '_get_tags', '_max_components', '_more_tags',
'_predict_proba_lr', '_solve_eigen', '_solve_lsqr', '_solve_svd', 'classes_',
'coef_', 'covariance_', 'decision_function', 'explained_variance_ratio_', 'fit',
'fit_transform', 'get_params', 'intercept_', 'means_', 'n_components', 'predict',
'predict_log_proba', 'predict_proba', 'priors', 'priors_', 'scalings_', 'score',
'set_params', 'shrinkage', 'solver', 'store_covariance', 'tol', 'transform']
```

Le champ « coef_ » nous intéresse particulièrement. Il correspond aux coefficients des fonctions de classement (COURS, page 6).

```
#affichage brut des coefficients
print(lda.coef_)
```

```
[[ 0.00363806  0.0067817  -0.06758017 -0.00093726  0.02449539  0.03978902
   0.00209182  0.07023599]
 [ 0.03080572  0.01741768  0.43020975  0.07572054  0.03158566 -0.13683605
  -0.00574405 -0.24028647]
 [ 0.03543454  0.0079735   0.3375194   0.12203319 -0.00900585  0.06556243
  -0.00882738 -0.13831135]]
```

La matrice est de dimension (3, 8) puisque nous avons un problème à (K = 3) classes (le nombre de modalités de la variable cible TYPE) et 8 descripteurs.

```
#dimensions
print(lda.coef_.shape)
```

```
(3, 8)
```

Assurons-nous de l'ordre de prise en compte des modalités de la cible.

```
#liste des modalités
print(lda.classes_)
```

```
['KIRSCH' 'MIRAB' 'POIRE']
```

Nous pouvons dès lors adopter une présentation plus sympathique des fonctions de classement en y associant les noms des variables explicatives et des modalités de TYPE.

```
#structure temporaire pour affichage des coefficients
tmp= pandas.DataFrame(lda.coef_.transpose(),columns=lda.classes_,index=XTrain.columns)
print(tmp)
```

```
        KIRSCH     MIRAB     POIRE
MEOH  0.003638  0.030806  0.035435
ACET  0.006782  0.017418  0.007973
BU1  -0.067580  0.430210  0.337519
BU2  -0.000937  0.075721  0.122033
ISOP  0.024495  0.031586 -0.009006
MEPR  0.039789 -0.136836  0.065562
PRO1  0.002092 -0.005744 -0.008827
ACAL  0.070236 -0.240286 -0.138311
```

Il ne faut pas oublier les constantes (intercept) des fonctions linéaires.

```
#et les constantes pour chaque classe
print(lda.intercept_)
```

```
[ -5.25513156 -19.91808283 -26.22259567]
```

A titre de référence, voici le tableau de sortie de SAS (Figure 1).

Fonction discriminante linéaire pour TYPE				
Variable	Libellé	KIRSCH	MIRAB	POIRE
Constante		-5.01645	-18.84069	-24.76488
MEOH	MEOH	0.00343	0.02903	0.03339
ACET	ACET	0.00639	0.01641	0.00751
BU1	BU1	-0.06368	0.40539	0.31805
BU2	BU2	-0.0008832	0.07135	0.11499
ISOP	ISOP	0.02308	0.02976	-0.00849
MEPR	MEPR	0.03749	-0.12894	0.06178
PRO1	PRO1	0.00197	-0.00541	-0.00832
ACAL	ACAL	0.06618	-0.22642	-0.13033

Figure 1 - Fonctions de classement - PROC DISCRIM de SAS

Les coefficients semblent différents, et c'est vrai quelle que soit la méthode (solver) utilisée. Cela m'a un peu dérouté initialement. J'ai constaté après coup que les coefficients étaient en réalité identiques à un facteur près $\left(\frac{n-K}{n}\right)$ pour toutes les variables et modalités. Dès lors, il n'y

a pas lieu de s'inquiéter. Les interprétations et les comportements en prédiction seront exactement les mêmes. C'est ce qui importe en définitive.

2.4 Evaluation globale du modèle

La méthode étant d'obédience statistique, les outils fournissent généralement un test de significativité globale du modèle. Il est basé sur l'écartement entre les barycentres conditionnels pour l'analyse discriminante. SAS affiche le tableau « Statistiques multivariées et Approximations F » (Figure 2). Nous nous intéressons en particulier à la ligne relative à « Wilks' Lambda ».

Statistiques multivariées et Approximations F					
S=2 M=2.5 N=20					
Statistique	Valeur	Valeur F	DDL num.	DDL den.	Pr > F
Wilks' Lambda	0.06671324	15.08	16	84	<.0001
Pillai's Trace	1.32125279	10.46	16	86	<.0001
Hotelling-Lawley Trace	8.17410078	21.08	16	65.231	<.0001
Roy's Greatest Root	7.38683115	39.70	8	43	<.0001
NOTE: la statistique F pour la plus grande racine de Roy est une borne supérieure.					
NOTE: la statistique F pour Lambda de Wilks est exacte.					

Figure 2 - Test de significativité globale - PROC DISCRIM

« Scikit-learn » n'intègre pas nativement ces statistiques. Mais nous pouvons les retrouver facilement (plus ou moins) à partir des informations fournies par l'objet.

2.4.1 Matrices de covariance

Nous avons besoin des matrices de covariance intra-classes et totales pour le calcul du Lambda (Λ) de Wilks (COURS, page 9).

Matrice de covariance intra-classe. Elle semble être directement fournie par l'objet Python.

```
#modifier le mode d'affichage des matrices numpy
#pas de notation scientifique
import numpy
numpy.set_printoptions(suppress=True,linewidth=120,precision=4)

#covariance
print(lda.covariance_)
```

```
[[ 37903.1042   7212.2269    282.246   -2377.1368   3293.0593   1263.1119   7466.5709    785.5844]
 [  7212.2269  14222.6955   -103.7163    140.1492    276.4854    210.3883  13233.2198    435.9877]
 [    282.246    -103.7163     34.1316     21.1435     66.184      20.675     256.8108      3.1599]
 [ -2377.1368    140.1492     21.1435   2662.8929   -406.1787    -86.1548  22725.1937    -21.6331]
```

```
[  3293.0593    276.4854     66.184    -406.1787    2032.         580.3434    -974.3584      14.1969]
[  1263.1119    210.3883     20.675     -86.1548     580.3434     231.5202    287.6098       2.2666]
[  7466.5709  13233.2198    256.8108  22725.1937    -974.3584     287.6098 321286.3586     752.7231]
[   785.5844    435.9877      3.1599    -21.6331      14.1969       2.2666    752.7231      62.3297]]
```

Mais, patatras… elle n'est pas identique à celle de SAS (Figure 3).

Matrice de covariance groupée intra-classe, DDL = 49									
Variable	Libellé	MEOH	ACET	BU1	BU2	ISOP	MEPR	PRO1	ACAL
MEOH	MEOH	40223.7025	7653.7918	299.5263	-2522.6758	3494.6752	1340.4453	7923.7079	833.6814
ACET	ACET	7653.7918	15093.4728	-110.0663	148.7298	293.4131	223.2692	14043.4170	462.6808
BU1	BU1	299.5263	-110.0663	36.2213	22.4380	70.2361	21.9408	272.5339	3.3533
BU2	BU2	-2522.6758	148.7298	22.4380	2825.9271	-431.0467	-91.4296	24116.5321	-22.9575
ISOP	ISOP	3494.6752	293.4131	70.2361	-431.0467	2156.4082	615.8747	-1034.0130	15.0661
MEPR	MEPR	1340.4453	223.2692	21.9408	-91.4296	615.8747	245.6949	305.2185	2.4053
PRO1	PRO1	7923.7079	14043.4170	272.5339	24116.5321	-1034.0130	305.2185	340956.9520	798.8082
ACAL	ACAL	833.6814	462.6808	3.3533	-22.9575	15.0661	2.4053	798.8082	66.1458

Figure 3 - Matrice de variance-covariance intra-classes - SAS

En réalité, il faut corriger la matrice de « scikit-learn » par les degrés de liberté (n est le nombre d'observations de l'échantillon d'apprentissage, K est le nombre de modalités de la variable cible, cf. Section 2.1).

```
#pour retrouver la matrice W de SAS
W = n/(n-K)*lda.covariance_
print(W)
```

```
[[ 40223.7025    7653.7918    299.5263  -2522.6758    3494.6752    1340.4453    7923.7079     833.6814]
 [  7653.7918  15093.4728   -110.0663    148.7298     293.4131     223.2692   14043.417      462.6808]
 [   299.5263   -110.0663     36.2213     22.438       70.2361      21.9408     272.5339       3.3533]
 [ -2522.6758    148.7298     22.438    2825.9271    -431.0467     -91.4296   24116.5321     -22.9575]
 [  3494.6752    293.4131     70.2361   -431.0467    2156.4082     615.8747   -1034.013       15.0661]
 [  1340.4453    223.2692     21.9408    -91.4296     615.8747     245.6949     305.2185       2.4053]
 [  7923.7079  14043.417     272.5339  24116.5321   -1034.013      305.2185  340956.952      798.8082]
 [   833.6814    462.6808      3.3533    -22.9575      15.0661       2.4053     798.8082      66.1458]]
```

A partir de cette matrice est calculée le tableau "Informations sur la matrice de covariance combinée" proposée par SAS (Figure 4).

Informations sur la matrice de covariance combinée	
Rang de la matrice de covariance	Log. naturel du déterminant de la matrice de covariance
8	58.32674

Figure 4 - Informations sur la matrice de covariance combinées – SAS

Qu'il nous est facile de reproduire.

```
#calculons son déterminant
detW = numpy.linalg.det(W)
print("Déterminant de la matrice : %.4f" % (detW))

#et son logarithme
logDetW = numpy.log(detW)
print("Logarithme du déterminant : %.4f" % (logDetW))
```

```
Déterminant de la matrice : 2142797158899651008947814 4.0000
Logarithme du déterminant : 58.3267
```

Matrice de covariance totale. La matrice de covariance totale n'est pas proposée par l'objet « scikit-learn ». Mais nous pouvons la calculer aisément avec la librairie « numpy ». La correspondance avec SAS est exacte (Figure 5)

Matrice de covariance sur la totalité de l'échantillon, DDL = 51									
Variable	Libellé	MEOH	ACET	BU1	BU2	ISOP	MEPR	PRO1	ACAL
MEOH	MEOH	136738.7084	6617.5839	3173.9067	372.9609	7655.1320	3593.5491	-65773.7620	1082.7291
ACET	ACET	6617.5839	14921.5437	-99.4211	-146.4921	74.8360	51.9067	12047.5942	427.8634
BU1	BU1	3173.9067	-99.4211	121.6077	85.7203	182.1202	79.1073	-2032.1418	10.5283
BU2	BU2	372.9609	-146.4921	85.7203	2968.6657	-178.3986	71.9951	22370.9557	-4.5959
ISOP	ISOP	7655.1320	74.8360	182.1202	-178.3986	2333.8643	754.0290	-3366.7314	32.9599
MEPR	MEPR	3593.5491	51.9067	79.1073	71.9951	754.0290	341.1990	-731.9934	14.0377
PRO1	PRO1	-65773.7620	12047.5942	-2032.1418	22370.9557	-3366.7314	-731.9934	392100.2023	626.8095
ACAL	ACAL	1082.7291	427.8634	10.5283	-4.5959	32.9599	14.0377	626.8095	64.8725

Figure 5 - Matrice de covariance totale - SAS

```
#covariance totale avec Numpy
T = numpy.cov(XTrain.values,rowvar=False)
print(T)
```

```
[[136738.7084   6617.5839   3173.9067    372.9609    7655.132     3593.5491  -65773.762     1082.7291]
 [  6617.5839  14921.5437    -99.4211   -146.4921      74.836       51.9067   12047.5942     427.8634]
 [  3173.9067    -99.4211    121.6077     85.7203     182.1202      79.1073   -2032.1418      10.5283]
 [   372.9609   -146.4921     85.7203   2968.6657    -178.3986      71.9951   22370.9557      -4.5959]
 [  7655.132       74.836     182.1202   -178.3986    2333.8643     754.029    -3366.7314      32.9599]
 [  3593.5491     51.9067     79.1073     71.9951     754.029      341.199     -731.9934      14.0377]
 [-65773.762    12047.5942  -2032.1418  22370.9557   -3366.7314    -731.9934  392100.2023     626.8095]
 [  1082.7291    427.8634     10.5283     -4.5959      32.9599      14.0377     626.8095      64.8725]]
```

2.4.2 Calcul du Lambda de Wilks

Avec ces matrices, nous pouvons calculer le Lambda de Wilks (COURS, page 9). De menues corrections sont nécessaires pour retrouver la valeur produite par SAS (Figure 2).

```
#lambda de Wilks
LW = numpy.linalg.det(lda.covariance_)/numpy.linalg.det((n-1)/n*T)
print(LW)
```

```
0.06671323859740352
```

2.4.3 Transformation de RAO

Le Lambda n'est pas tabulé. Il faut passer par une transformation – dite de Rao – un peu ésotérique pour obtenir une statistique qui suit une loi connue, celle de Fisher en l'occurrence. Les formules sont accessibles dans la documentation en ligne de SAS. On peut dès lors rendre compte de la significativité globale du modèle.

```
#calcul du F de Rao et des degrés de liberté
#Voir
https://support.sas.com/documentation/cdl/en/statug/63962/HTML/default/viewer.htm#statug_introreg_sect012.htm

#Lambda de Wilks
print("Lambda de Wilks : %.4f" % (LW))

#ddl numérateur
ddlNum = p * (K-1)
print("DDL numérateur : %.d" % (ddlNum))

#valeur intermédiaire pour calcul du ddl dénominateur
temp = p**2 + (K-1)**2 - 5
temp = numpy.where(temp > 0,numpy.sqrt(((p**2) * ((K-1)**2) - 4)/temp),1)

#ddl dénominateur
ddlDenom = (2*n-p-K-2)/2 * temp - (ddlNum - 2)/2
print("DDL dénominateur : %.d" % (ddlDenom))

#stat de test
FRao = LW**(1/temp)
FRao = ((1-FRao)/FRao)*(ddlDenom/ddlNum)
print("Statistique F de Rao : %.4f" % (FRao))

#p-value
import scipy
print("P-value du test : %.4f" % (1-scipy.stats.f.cdf(FRao,ddlNum,ddlDenom)))
```

```
Lambda de Wilks : 0.0667
DDL numérateur : 16
DDL dénominateur : 84
Statistique F de Rao : 15.0761
P-value du test : 0.0000
```

Argh. Les calculs ne sont pas des plus aisés, loin de là. Mais nous retrouvons bien les valeurs de référence (Figure 2). L'écartement entre les barycentres conditionnels est significatif à 5%. L'analyse discriminante est viable dans ce contexte.

2.5 Evaluation des contributions des variables

Mesurer l'impact des variables est crucial pour l'interprétation du mécanisme d'affectation. J'y ai moi-même consacré plusieurs supports (« *Importance des variables dans les modèles* », février 2019 ; « *Interpréter un classement en analyse prédictive* » ; avril 2019 ; etc.). Pour l'analyse discriminante, il est possible de produire une mesure d'importance des variables basée sur leurs contributions à la discrimination. Concrètement, il s'agit simplement d'opposer les lambda de Wilks avec ou sans la variable à évaluer.

2.5.1 Affichage des contributions sous TANAGRA

Ces résultats sont fournis directement par certains logiciels tels que TANAGRA (Figure 6).

MANOVA

Stat	Value	p-value
Wilks' Lambda	0.0667	-
Bartlett -- C(16)	123.1845	0.0000
Rao -- F(16, 84)	15.0761	0.0000

LDA Summary

	Classification functions			Statistical Evaluation			
Attribute	KIRSCH	MIRAB	POIRE	Wilks L.	Partial L.	F(2,42)	p-value
MEOH	0.003428	0.029028	0.033390	0.117975	0.565488	16.13607	0.000006
ACET	0.006390	0.016413	0.007513	0.074153	0.899667	2.34196	0.108572
BU1	-0.063681	0.405390	0.318047	0.084183	0.792475	5.49926	0.007563
BU2	-0.000883	0.071352	0.114993	0.095695	0.697142	9.12300	0.000513
ISOP	0.023082	0.029763	-0.008486	0.072310	0.922600	1.76176	0.184196
MEPR	0.037494	-0.128942	0.061780	0.087798	0.759852	6.63695	0.003128
PRO1	0.001971	-0.005413	-0.008318	0.092396	0.722038	8.08434	0.001071
ACAL	0.066184	-0.226424	-0.130332	0.075884	0.879150	2.88670	0.066885
constant	-5.016453	-18.840686	-24.764879	-			

Figure 6 - Analyse discriminante linéaire sous TANAGRA

Dans le tableau des coefficients, nous distinguons le lambda de Wilks lorsque la variable est retirée (Wilks L. ; 0.117975 si on retire la variable MEOH par exemple), le ratio entre le lambda global (0.0667) et ce dernier (Partial L. ; pour MEOH : 0.0667 / 0.117975 = 0.565488), et la

statistique de test de significativité de l'écart (F = 16.13607) qui suit une distribution de Fisher à (K – 1 = 2) et (N – K – p + 1 = 52 – 3 – 8 + 1 = 42) (COURS, page 10).

2.5.2 Obtention des contributions sous SAS

Il faut un peu ruser pour obtenir le même tableau sous SAS. Nous initions une sélection de variables « backward » avec STEPDISC.

```
PROC STEPDISC DATA = EAU_DE_VIE METHOD = backward SLSTAY= 0.5;
      class TYPE;
      VAR MEOH ACET BU1 BU2 ISOP MEPR PRO1 ACAL;
RUN;
```

La première grille renvoie les mêmes informations (Figure 7).

Statistiques pour la suppression, DDL = 2, 42				
Variable	Libellé	R carré partiel	Valeur F	Pr > F
MEOH	MEOH	0.4345	16.14	<.0001
ACET	ACET	0.1003	2.34	0.1086
BU1	BU1	0.2075	5.50	0.0076
BU2	BU2	0.3029	9.12	0.0005
ISOP	ISOP	0.0774	1.76	0.1842
MEPR	MEPR	0.2401	6.64	0.0031
PRO1	PRO1	0.2780	8.08	0.0011
ACAL	ACAL	0.1208	2.89	0.0669

Figure 7 - Importance des variables – SAS

Le « R carré partiel » de SAS correspond au (1 – Lambda partiel) de TANAGRA. Pour MEOH par exemple, nous avons 0.4345 = 1 – 0.565488.

2.5.3 Calculs à partir des matrices de covariances

Comment obtenir ces résultats, essentiels pour l'interprétation du modèle, sous Python ? L'idée est d'évaluer la contribution de chaque variable dans l'écartement entre les barycentres conditionnels en opposant les Lambda de Wilks, avec et sans la variable incriminée. Nous réalisons les opérations en manipulant judicieusement les matrices de covariance intra-classes et totales sans avoir à réitérer entièrement les calculs de l'analyse discriminante. Avec une simple boucle, il est possible de traiter rapidement une base comprenant beaucoup de variables.

```
#degrés de liberté - numérateur
ddlSuppNum = K-1

#degré de liberté dénominateur
ddlSuppDenom = n - K - p + 1

#FTest - préparation
FTest = numpy.zeros(p)

#p-value pour FTest
pvalueFTest = numpy.zeros(p)

#Tester chaque variable
for j in range(p):
    #matricesintermédiaires numérateur
    tempNum = lda.covariance_.copy()
    #supprimer la référence de la variable à traiter
    tempNum = numpy.delete(tempNum,j,axis=0)
    tempNum = numpy.delete(tempNum,j,axis=1)
    #même chose pour dénominateur
    tempDenom = (n-1)/n*T
    tempDenom = numpy.delete(tempDenom,j,axis=0)
    tempDenom = numpy.delete(tempDenom,j,axis=1)
    #lambda sans la variable
    LWVar = numpy.linalg.det(tempNum)/numpy.linalg.det(tempDenom)
    #print(LWVar)
    #FValue
    FValue = ddlSuppDenom/ddlSuppNum*(LWVar/LW-1)
    #récupération des valeurs
    FTest[j] = FValue
    pvalueFTest[j] = 1 - scipy.stats.f.cdf(FValue,ddlSuppNum,ddlSuppDenom)

#affichage
temp = {'var':XTrain.columns,'F':FTest,'pvalue':pvalueFTest}
print(pandas.DataFrame(temp))
```

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

Les résultats sont tout à fait conformes à ceux de TANAGRA et SAS.

3 Evaluation en test

L'évaluation sur l'échantillon test est une approche privilégiée pour mesurer et comparer les performances des modèles de nature et de complexité différente. Dans cette section, nous traitons la seconde feuille « TEST » comportant 50 observations de notre classeur Excel.

3.1 Importation et inspection des données

Nous chargeons la feuille « TEST ». Nous préparons les structures. Nous affichons pour vérification la distribution des classes. Elle est similaire à celle de l'échantillon « TRAIN ».

```
#chargement échantillon test
DTest = pandas.read_excel("Eau_de_vie_LDA_python.xlsx",sheet_name="TEST")
print(DTest.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50 entries, 0 to 49
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   TYPE    50 non-null     object
 1   MEOH    50 non-null     int64
 2   ACET    50 non-null     int64
 3   BU1     50 non-null     float64
 4   BU2     50 non-null     float64
 5   ISOP    50 non-null     int64
 6   MEPR    50 non-null     int64
 7   PRO1    50 non-null     int64
 8   ACAL    50 non-null     float64
dtypes: float64(3), int64(5), object(1)
```

```
#récupérer la matrice des X en test
XTest = DTest.iloc[:,1:]

#distribution des classes
print(DTest.TYPE.value_counts(normalize=True))
```

```
POIRE     0.38
MIRAB     0.34
KIRSCH    0.28
Name: TYPE, dtype: float64
```

3.2 Prédiction des classes et mesures des performances

Il y a deux étapes dans l'évaluation : effectuer la prédiction à partir de la matrice des explicatives de l'échantillon test, la confronter avec les classes observées. La fonction score() fait tout le travail en un seul appel de la méthode. Mais les résultats sont des plus laconiques, nous avons

uniquement accès au taux de reconnaissance (taux de succès), il est de 82% c.-à-d. 0.82 x 50 = 41 observations ont été correctement classées.

```
#la fonction "score" fournit le taux de succès
print(lda.score(XTest,DTest.TYPE))
```

```
0.82
```

Explicitons la démarche dans ce qui suit, pour plus de clarté, et aussi pour obtenir des indicateurs de performances plus détaillées. La fonction predict() permet de produire les prédictions à partir de la matrice des explicatives en test.

```
#prediction simple
pred = lda.predict(XTest)

#distribution des classes prédites
print(numpy.unique(pred,return_counts=True))
```

```
(array(['KIRSCH', 'MIRAB', 'POIRE'], dtype='<U6'), array([15, 19, 16], dtype=int64))
```

15 observations ont été prédites KIRSCH, 19 MIRAB et 16 POIRE. La matrice de confusion est issue de la confrontation entre ces prédictions et les classes observées. Nous faisons appel au module « *metrics* » de la librairie « scikit-learn ».

```
#matrice de confusion
from sklearn import metrics
print(metrics.confusion_matrix(DTest.TYPE,pred))
```

```
[[14  0  0]
 [ 0 14  3]
 [ 1  5 13]]
```

La fonction classification_report() génère un rapport sur les performances globales, mais aussi sur les reconnaissances par classe (rappel, précision et *F-Measure* [F1-Score]).

```
#rapport
print(metrics.classification_report(DTest.TYPE,pred))
```

```
              precision    recall  f1-score   support

      KIRSCH       0.93      1.00      0.97        14
       MIRAB       0.74      0.82      0.78        17
       POIRE       0.81      0.68      0.74        19

    accuracy                           0.82        50
   macro avg       0.83      0.84      0.83        50
weighted avg       0.82      0.82      0.82        50
```

Nous retrouvons, entres autres, le taux de succès de 82%.

3.3 Probabilités d'affectation

« Scikit-learn » peut aussi calculer les probabilités d'affectation aux classes avec predict_proba(). Elles permettent une analyse plus fine de la qualité du modèle, via la construction de la courbe ROC par exemple, dont le principe reste valable pour les problèmes multi classes.

```
#probabilités d'affectation des 15 premiers (individus en test n°0 à n°14)
proba = lda.predict_proba(XTest)
print(proba[:15,:])
```

```
[[1.     0.     0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [0.9999 0.0001 0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [0.9998 0.0002 0.    ]
 [1.     0.     0.    ]
 [1.     0.     0.    ]
 [0.     0.3023 0.6977]]
```

Les colonnes sont dans l'ordre des classes.

```
#ordre des classes
print(lda.classes_)
```

```
['KIRSCH' 'MIRAB' 'POIRE']
```

La classe prédite correspond à celle qui maximise la probabilité d'affectation. Les voici par exemple pour l'individu n°14.

```
#proba individu n°14
print(proba[14,:])
```

```
[0.     0.3023 0.6977]
```

La classe prédite correspond à la probabilité maximum.

```
#affectation pour individu n°14
print(pred[14])
```

```
POIRE
```

Nous reproduisons le mécanisme de calcul de ces probabilités dans cette section. Nous prendrons comme référence les résultats pour l'individu n°14.

La description de l'individu est le point de départ.

```
#description de l'individu n°14
print(XTest.iloc[14,:])
```

```
MEOH    1090.0
ACET     106.0
BU1       18.0
BU2        0.0
ISOP      60.0
MEPR      26.0
PRO1      99.0
ACAL      22.0
Name: 14, dtype: float64
```

Nous lui appliquons la fonction de classement KIRSCH (COURS, page 6).

```
#application de la fonction de classement
#KIRSCH
fKIRSCH = numpy.sum(DTest.iloc[14,1:] * lda.coef_[0,:]) + lda.intercept_[0]
print(fKIRSCH)
```

```
2.469291271246247
```

Nous faisons de même pour les autres classes.

```
#MIRAB
fMIRAB = numpy.sum(DTest.iloc[14,1:] * lda.coef_[1,:]) + lda.intercept_[1]

#POIRE
fPOIRE = numpy.sum(DTest.iloc[14,1:] * lda.coef_[2,:]) + lda.intercept_[2]
```

Ces valeurs ne correspondent pas à des probabilités. Nous leur appliquons la transformation sotfmax pour obtenir quelque chose qui y ressemble c.-à-d. des valeurs qui varient entre 0 et 1, et dont la somme est égale à 1.

```
#dénominateur
denomSoftMax = numpy.exp(fKIRSCH)+numpy.exp(fMIRAB)+numpy.exp(fPOIRE)

#softmax KIRSCH
pKIRSCH = numpy.exp(fKIRSCH)/denomSoftMax

#softmax MIRAB
pMIRAB = numpy.exp(fMIRAB)/denomSoftMax

#softmax POIRE
pPOIRE = numpy.exp(fPOIRE)/denomSoftMax
```

```
#affichage des probas d'affectation estimées
print("(KIRSCH : %.8f ; MIRAB : %.8f : POIRE : %.8f)" % (pKIRSCH,pMIRAB,pPOIRE))
```

```
(KIRSCH : 0.00000053 ; MIRAB : 0.30227923 : POIRE : 0.69772024)
```

4 Plus loin avec la classe LinearDiscriminantAnalysis

La classe *LinearDiscriminantAnalysis* de « scikit-learn » nous gratifie d'outils supplémentaires que nous explorons dans cette section.

4.1 Analyse factorielle discriminante

L'*analyse factorielle discriminante* (AFD) est une technique descriptive qui vise à produire un espace de dimension réduite – un repère factoriel – permettant de discerner au mieux les barycentres conditionnels des classes. La finalité est différente de l'analyse discriminante linéaire, prédictive, mais les deux approches se rejoignent. Il est même possible de déduire un mécanisme de classement de l'AFD (TUTO, section 3.4). Des outils qui ont pignon sur rue – la fonction *lda*() du package *MASS* de R par exemple – n'affichent que les équations des axes factoriels.

La fonction transform() de « scikit-learn » calcule la représentation factorielle des individus. Le nombre de facteurs est égal à min(K – 1, p) = min (3 – 1, 8) = 2.

```
#projection dans l'espace facoriel
Fact = lda.transform(XTrain)
print(Fact[:5,:])
```

```
[[-1.1335  0.6866]
 [-0.1353  1.3163]
 [-0.4198  0.0022]
 [-0.935   0.5853]
 [ 0.0778 -0.2771]]
```

Avec un repère factoriel à 2 dimensions, nous pouvons projeter les individus dans le plan.

```
%matplotlib inline

#data frame pour représentation graphique
dfFact = pandas.DataFrame(Fact,columns=['F1','F2'])
dfFact['classe'] = DTrain.TYPE

#représentation graphique
import seaborn as sns
sns.scatterplot(x="F1",y="F2",hue="classe",data=dfFact)
```

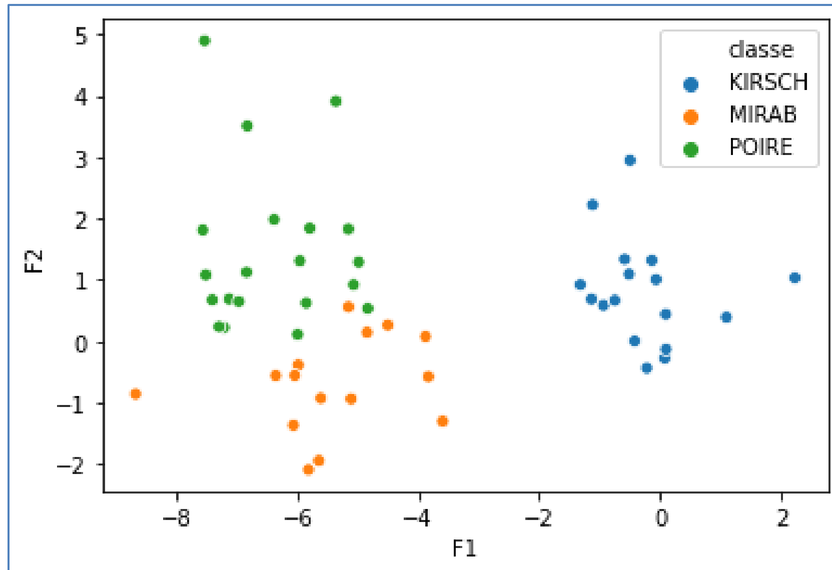

Figure 8 - Représentation factorielle des observations

Nous distinguons assez bien les 3 catégories {KIRSCH, MIRAB, POIRE} dans le repère factoriel. KIRSCH est vraiment à part, ce qui explique la valeur de F1-Score élevé. Les erreurs proviennent avant de tout de l'empiètement entre les classes MIRAB et POIRE.

4.2 Frontière de séparation

L'analyse discriminante linéaire est un classifieur linéaire. Dans un problème à deux variables explicatives, nous pouvons représenter la frontière entre les classes à l'aide d'une droite. Pour cette section, nous utilisons un nouveau jeu de données (Tableau 1, page 29) issu de l'excellent ouvrage de Tomassone et al. (1988) qui a fait une bonne part de ma culture en apprentissage statistique durant mes années d'études. Nous chargeons la nouvelle base.

```
#chargement des données
DTom = pandas.read_excel("Tomassone_et_al.xlsx",sheet_name=0)
print(DTom.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Groupe   10 non-null     object
 1   X1       10 non-null     int64
 2   X2       10 non-null     int64
dtypes: int64(2), object(1)
memory usage: 368.0+ bytes
None
```

Puisque nous avons 2 variables, nous pouvons projeter les individus dans le plan.

```
#graphique
sns.scatterplot(x="X1",y="X2",hue="Groupe",data=DTom)
```

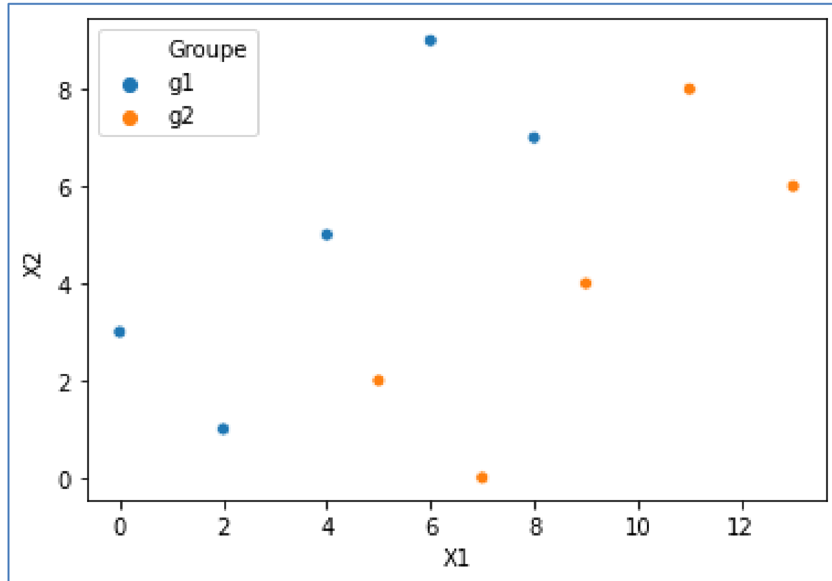

Figure 9 - Données Tomassone et al., 1988

Nous avons un problème binaire, la variable cible « Groupe » possède deux modalités {g1, g2}. Elles semblent linéairement séparables dans l'espace de représentation.

Nous lançons l'analyse discriminante sur ce nouveau jeu de données. Malgré que nous ayons deux classes, nous avons une fonction score unique [a * x1 + b * x2 + c] qui résulte – c'est une manière simple de le voir – d'une différence termes à termes des coefficients des fonctions de classement. Je l'appelle « fonction score » dans mes enseignements (COURS, page 11).

```
#lda
ldaTom = LinearDiscriminantAnalysis(solver="eigen")

#apprentissage
ldaTom.fit(DTom.iloc[:,1:],DTom.Groupe)

#fonction de classement
print("Coef. de la fonction de classement :")
print(ldaTom.coef_)

#constante
print("Intercept de la fonction de classement :")
print(ldaTom.intercept_)
```

```
Coef. de la fonction de classement :
[[ 2.0139 -1.7361]]
Intercept de la fonction de classement :
[-5.2778]
```

La règle d'affectation est modifiée. Nous comparons le score à la valeur seuil 0 pour affecter à la classe « g1 » ou « g2 ». De fait, la frontière séparant les classes est définie par l'équation :

$$a * x1 + b * x2 + c = 0$$

Nous traçons cette frontière dans notre graphique nuage de points (Figure 10).

```
#valeurs de x1 pour générer la frontière
x1 = numpy.arange(0,14,step=1)

#coordonnées x2 poir la frontière
x2 = (-ldaTom.intercept_ - ldaTom.coef_[0][0] * x1) / ldaTom.coef_[0][1]

#data frame
dfTemp = pandas.DataFrame({'x1':x1,'x2':x2})

#graphique des points
ax = sns.scatterplot(x="X1",y="X2",hue="Groupe",data=DTom)

#rajouter la frontière de séparation
sns.lineplot(x='x1',y='x2',data=dfTemp)
```

Nous obtenons une discrimination parfaite. Aucun individu n'est du mauvais côté de la frontière. Le taux d'erreur, en resubstitution tout du moins, est nul.

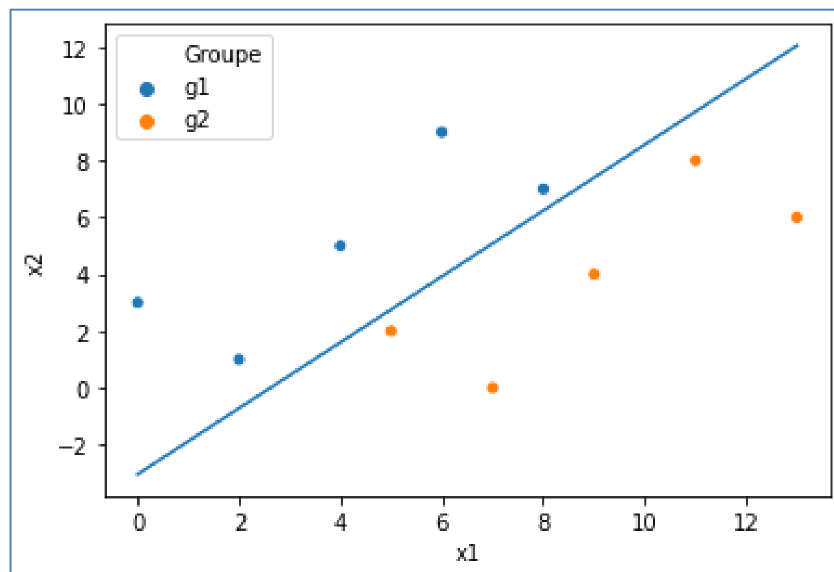

Figure 10 - Frontière de séparation des classes - Tomassone et al.

4.3 Distance à la frontière de séparation

Les individus sont certes tous bien classés, mais pas avec la même intensité. Pour certains, la décision est plus tranchée (ceux qui sont éloignés de la frontière) que d'autres (ceux qui sont proches). On peut ainsi produire un indicateur de fiabilité de la décision à partir de la distance à la frontière, ou tout du moins d'une valeur qui lui est proportionnelle. C'est ce que fournit la méthode decision_function() de « scikit-learn ».

```
#distance à la frontière
dist = ldaTom.decision_function(DTom.iloc[:,1:])
print(pandas.DataFrame(dist,index=DTom.index))
```

```
           0
0 -10.486111
1  -2.986111
2  -5.902778
3  -8.819444
4  -1.319444
5   1.319444
6   8.819444
7   5.902778
8   2.986111
9  10.486111
```

Nous avons repris notre graphique en y ajoutant les numéros des individus et la « distance » à la frontière pour certains d'entre eux (elles ont été rajoutées à la main sous Powerpoint, sous Python ça doit être possible mais je ne le sentais pas vraiment…).

```
#graphique des points
ax = sns.scatterplot(x="X1",y="X2",hue="Groupe",data=DTom)

#ajouter les numéros d'individus
for i in DTom.index:
    ax.text(DTom.X1[i]+0.2,DTom.X2[i],str(DTom.index[i]))

#rajouter la frontière de séparation
sns.lineplot(x='x1',y='x2',data=dfTemp)
```

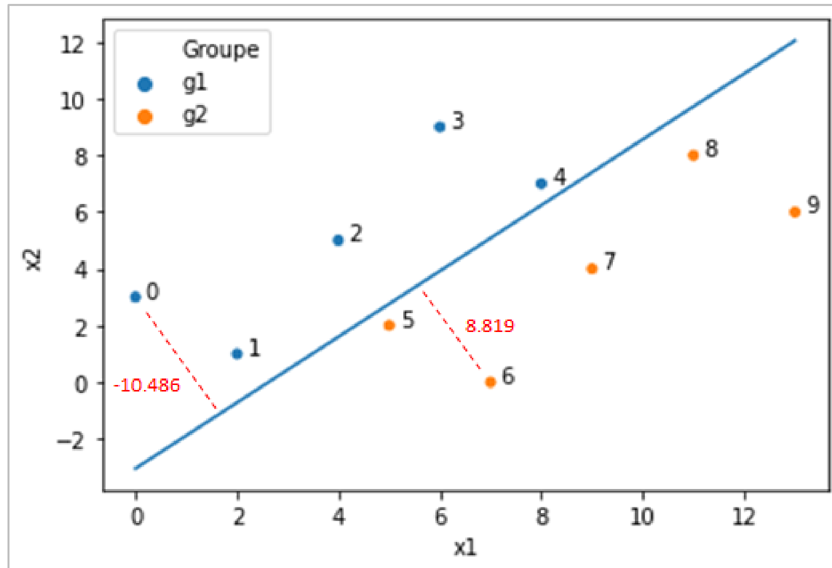

Figure 11 - Distance à la frontière des individus

Curieux de comprendre la nature de ces valeurs, j'ai entrepris de calculer la distance à la droite (a x1 + b x2 + c = 0) de chaque point. La *formule* est simple. Pour un individu (ω) :

$$d(\omega) = \frac{|a.x_1(\omega) + b.x_2(\omega) + c|}{\sqrt{a^2 + b^2}}$$

Je me suis rendu compte que decision_function() n'en reprenait que le numérateur, et que la valeur était signée selon la position par rapport à la frontière. Quelques commandes Python suffisent à les retrouver.

```
#vérifions pour les points:
# => la distance n'est pas normée - on n'a que le numérateur de la formule
# => elle est signée
d = ldaTom.coef_[0][0]*DTom.X1+ldaTom.coef_[0][1]*DTom.X2+ldaTom.intercept_
print(d)
0   -10.486111
1    -2.986111
2    -5.902778
3    -8.819444
4    -1.319444
5     1.319444
6     8.819444
7     5.902778
8     2.986111
9    10.486111
dtype: float64
```

Comme nous pouvons le constater ci-dessous, ces valeurs de fiabilité de la décision basées sur la distance à la frontière ne sont pas sans lien avec les probabilités d'appartenances aux classes, qui varient dans le même sens mais avec une valeur seuil de 0.5 pour l'attribution des modalités. Cette propriété est utilisée par les techniques d'apprentissage qui ne savent pas produire de prime abord la probabilité d'affectation aux classes (ex. *SVM*, pages 30 et suivantes).

```
#et la probabilité d'affectation (à g2) est proportionnelle à cette distance
print(pandas.DataFrame(ldaTom.predict_proba(DTom.iloc[:,1:]),index=DTom.index)[1])
```

```
0    0.000028
1    0.048057
2    0.002724
3    0.000148
4    0.210911
5    0.789089
6    0.999852
7    0.997276
8    0.951943
9    0.999972
Name: 1, dtype: float64
```

5 Conclusion

« Qu'importe le flacon pourvu qu'on ait l'ivresse », je me répète souvent à ce sujet. Dans ce tutoriel, nous avons étudié le fonctionnement de la classe *LinearDiscriminantAnalysis* de « scikit-learn » pour Python. Ce n'est pas l'outil qui nous viendrait immédiatement à l'esprit en matière d'analyse discriminante linéaire. Pourtant, force est de constater que nous avons pu facilement retrouver les résultats des logiciels qui font référence dans le domaine.

6 Références

(COURS) R. Rakotomalala, « *Analyse discriminante linéaire* ».

(TUTO) Tutoriel Tanagra, « *Analyse discriminante linéaire – Comparaison* », juillet 2012.

Tomassone R., Danzart M., Daudin J.J., Masson J.P., « Discrimination et classement », Masson, 1988.