TP mixte UP2 : Apprentissage statistique

El Amrani-Zerrifi Hamza, El Yaakoubi Oussama, Jhabli Ayoub

November 13, 2022

Introduction

Dans ce rapport, nous étudions un échantillon de données intitulé **"Ionosphere"**. Ce dernier contient des données radar qui ont été recueillies par un système à Goose Bay, au Labrador Canada. Ce système est constitué d'un réseau phasé de plusieurs antennes haute fréquence avec une puissance totale émise de l'ordre de 6,4 kilowatts. Les cibles étaient des électrons libres dans l'ionosphère. Les **bons** (classifiés «g» pour **good**) retours radar sont ceux qui mettent en évidence un certain type de structure dans l'ionosphère. Les **mauvais** («b» pour **bad**) retours sont ceux qui ne le font pas ; leurs signaux traversent l'ionosphère.

L'échantillon contient 351 observations et **33 variables**, dont la variable **"class"** qui est celle que l'on cherche à expliquer. Les 32 variables quantitatives sont supposées continues. Nous sommes ainsi confrontés à un problème de **classification binaire**.

Le rapport se structure de la façon qui suit: dans un premier temps, il s'agit de faire une **analyste statistique** de notre échantillon de données. Ensuite, nous séparerons l'échantillon en deux groupes (*train* et *test*) et construirons puis testerons des modèles de **forêt aléatoire** pour la classification. Après celà, il s'agira de construire et de tester sur ces données un modèle de **forêt d'isolement** de détection d'anomalies. Par la suite, nous réaliserons l'**analyse par composante principale (ACP)** des données et étudierons son utilisation dans la réduction du nombre de variables ainsi que dans la détection d'anomalie. Enfin, nous mettrons en oeuvre l'**AFD descriptive et prédictive** sur ces données dans le but de discriminer les deux classes "g" et "b" d'individus.

Tout au long du rapport, nous comparerons et interpréterons les différences entre les résultats issus de ces diverses techniques.

1 Analyse statistique

On récupère dans un premier temps nos données:

[1]:
```
setwd('D:/ICM 2A/Data science/UP2 - Apprentissage statistique')
rm(list=ls())
data <- read.csv(file='Ionosphere.csv', header=TRUE, sep=";")
```

Visualisons les différentes variables:

[2]:
```
summary(data)
```

```
      a01                a02                 a03                a04          
 Min.   :-1.0000   Min.   :-1.00000   Min.   :-1.0000   Min.   :-1.0000  
```

```
1st Qu.: 0.4721   1st Qu.:-0.06474   1st Qu.: 0.4127   1st Qu.:-0.0248
Median : 0.8711   Median : 0.01631   Median : 0.8092   Median : 0.0228
Mean   : 0.6413   Mean   : 0.04437   Mean   : 0.6011   Mean   : 0.1159
3rd Qu.: 1.0000   3rd Qu.: 0.19418   3rd Qu.: 1.0000   3rd Qu.: 0.3347
Max.   : 1.0000   Max.   : 1.00000   Max.   : 1.0000   Max.   : 1.0000
     a05               a06               a07               a08
Min.   :-1.0000   Min.   :-1.00000   Min.   :-1.00000   Min.   :-1.00000
1st Qu.: 0.2113   1st Qu.:-0.05484   1st Qu.: 0.08711   1st Qu.:-0.04807
Median : 0.7287   Median : 0.01471   Median : 0.68421   Median : 0.01829
Mean   : 0.5501   Mean   : 0.11936   Mean   : 0.51185   Mean   : 0.18135
3rd Qu.: 0.9692   3rd Qu.: 0.44567   3rd Qu.: 0.95324   3rd Qu.: 0.53419
Max.   : 1.0000   Max.   : 1.00000   Max.   : 1.00000   Max.   : 1.00000
     a09               a10               a11               a12
Min.   :-1.00000   Min.   :-1.00000   Min.   :-1.0000   Min.   :-1.00000
1st Qu.: 0.02112   1st Qu.:-0.06527   1st Qu.: 0.0000   1st Qu.:-0.07372
Median : 0.66798   Median : 0.02825   Median : 0.6441   Median : 0.03027
Mean   : 0.47618   Mean   : 0.15504   Mean   : 0.4008   Mean   : 0.09341
3rd Qu.: 0.95790   3rd Qu.: 0.48237   3rd Qu.: 0.9555   3rd Qu.: 0.37486
Max.   : 1.00000   Max.   : 1.00000   Max.   : 1.0000   Max.   : 1.00000
     a13               a14               a15               a16
Min.   :-1.0000   Min.   :-1.00000   Min.   :-1.0000   Min.   :-1.000000
1st Qu.: 0.0000   1st Qu.:-0.08170   1st Qu.: 0.0000   1st Qu.:-0.225690
Median : 0.6019   Median : 0.00000   Median : 0.5909   Median : 0.000000
Mean   : 0.3442   Mean   : 0.07113   Mean   : 0.3819   Mean   :-0.003617
3rd Qu.: 0.9193   3rd Qu.: 0.30897   3rd Qu.: 0.9357   3rd Qu.: 0.195285
Max.   : 1.0000   Max.   : 1.00000   Max.   : 1.0000   Max.   : 1.000000
     a17               a18               a19               a20
Min.   :-1.0000   Min.   :-1.00000   Min.   :-1.0000   Min.   :-1.000000
1st Qu.: 0.0000   1st Qu.:-0.23467   1st Qu.: 0.0000   1st Qu.:-0.243870
Median : 0.5762   Median : 0.00000   Median : 0.4991   Median : 0.000000
Mean   : 0.3594   Mean   :-0.02402   Mean   : 0.3367   Mean   : 0.008296
3rd Qu.: 0.8993   3rd Qu.: 0.13437   3rd Qu.: 0.8949   3rd Qu.: 0.188760
Max.   : 1.0000   Max.   : 1.00000   Max.   : 1.0000   Max.   : 1.000000
     a21               a22               a23               a24
Min.   :-1.0000   Min.   :-1.00000   Min.   :-1.0000   Min.   :-1.00000
1st Qu.: 0.0000   1st Qu.:-0.36689   1st Qu.: 0.0000   1st Qu.:-0.33239
Median : 0.5318   Median : 0.00000   Median : 0.5539   Median :-0.01505
Mean   : 0.3625   Mean   :-0.05741   Mean   : 0.3961   Mean   :-0.07119
3rd Qu.: 0.9112   3rd Qu.: 0.16463   3rd Qu.: 0.9052   3rd Qu.: 0.15676
Max.   : 1.0000   Max.   : 1.00000   Max.   : 1.0000   Max.   : 1.00000
     a25               a26               a27               a28
Min.   :-1.0000   Min.   :-1.00000   Min.   :-1.0000   Min.   :-1.00000
1st Qu.: 0.2864   1st Qu.:-0.44316   1st Qu.: 0.0000   1st Qu.:-0.23689
Median : 0.7082   Median :-0.01769   Median : 0.4966   Median : 0.00000
Mean   : 0.5416   Mean   :-0.06954   Mean   : 0.3784   Mean   :-0.02791
3rd Qu.: 0.9999   3rd Qu.: 0.15354   3rd Qu.: 0.8835   3rd Qu.: 0.15407
Max.   : 1.0000   Max.   : 1.00000   Max.   : 1.0000   Max.   : 1.00000
     a29               a30               a31               a32
```

```
Min.   :-1.0000   Min.   :-1.000000   Min.   :-1.0000   Min.   :-1.00000
1st Qu.: 0.0000   1st Qu.:-0.242595   1st Qu.: 0.0000   1st Qu.:-0.16535
Median : 0.4428   Median : 0.000000   Median : 0.4096   Median : 0.00000
Mean   : 0.3525   Mean   :-0.003794   Mean   : 0.3494   Mean   : 0.01448
3rd Qu.: 0.8576   3rd Qu.: 0.200120   3rd Qu.: 0.8138   3rd Qu.: 0.17166
Max.   : 1.0000   Max.   : 1.000000   Max.   : 1.0000   Max.   : 1.00000
   class
Length:351
Class :character
Mode  :character
```

On a 33 variables descriptives de 351 individus. 32 variables sont numériques et la variable classe est binaire ('g' ou 'b'). Quelques statistiques descriptives (min, mean, max, median) des variabes numériques sont données par la fonction summary ci-dessus. Quelques remarques :

Les 32 variables numériques sont toutes **à valeur dans l'intervalle** $[-1, 1]$, ce qui laisserait peut-être penser qu'elles possèdent la même unité (ou du moins le même ordre de grandeur). Certaines semblent **centrées** (moyenne nulle), mais d'autres non.

Etudions l'écart-type des 32 variables numériques:

[3]:
```
sapply(data[,-33], sd, na.rm = TRUE)
```

a01 0.497708202520014 **a02** 0.44143477986208 **a03** 0.519861513411958 **a04** 0.460810128995047
a05 0.492653771140902 **a06** 0.520749895758722 **a07** 0.507065526859402 **a08** 0.483850888265982
a09 0.563496361914066 **a10** 0.494817449158568 **a11** 0.622186124042585 **a12** 0.494872640828379
a13 0.652827832222125 **a14** 0.45837067230505 **a15** 0.618020354205151 **a16** 0.496761983253337
a17 0.626266809736973 **a18** 0.519076091804175 **a19** 0.609828322665831 **a20** 0.518165886822502
a21 0.603767489643601 **a22** 0.527456318702114 **a23** 0.578450887520743 **a24** 0.508494500358945
a25 0.516204665375986 **a26** 0.550025242775508 **a27** 0.575885558649988 **a28** 0.507974098829109
a29 0.571483371941104 **a30** 0.51357436073864 **a31** 0.522663372822405 **a32** 0.46833722312208

Les 32 variables descriptives possèdent toutes un **écart-type proche de 0.5**, du même ordre de grandeur.

En conclusion, les variables semblent partager la même **échelle** et ont une **variance comparable**, elles ne sont néanmoins pas toutes centrée.

Ces similarités entre les différentes variables descriptives nous seront utiles plus tard, notamment lors de l'ACP, pour décider du type de prétraitement nécessaire des données avant analyse.

2 Classification avec une forêt aléatoire

Le but est de construire une forêt aléatoire de haute performance qui a pour but de bien classer les observations de la variable "class" entre "g" ou "b".

Dans un premier temps, on divise notre base de données en 70% train pour l'apprentissage et 30% test (méthode hold-out pour la validation), en fixant le seed (génération des suites pseudo-aléatoires)

à 1234:

[4]:
```
set.seed(1234)
index <- sample(1:nrow(data),round(0.70*nrow(data)))
train <- data[index,]
test <- data[-index,]
```

On importe la librairie randomForest pour créer notre modèle de forêt:

[7]:
```
library(randomForest)
library(Metrics)
```

2.1 Forêt aléatoire non optimisée

On commence par une forêt aléatoire avec les hyperparamètres par défaut:

[7]:
```
set.seed(1234)
rf <- randomForest(as.factor(class) ~ .,data=train)
```

Réalisons désormais la prédiction sur nos données test et mesurons la précision des prédictions:

[8]:
```
p1=predict(rf,test)
accuracy(test$class, p1)
```

0.952380952380952

Cela signifie qu'avec ce modèle, 95% des individus de l'ensemble test ont bien été classifiés.

Ceci nous fournit un score de référence pour la suite de notre étude.

Construisons la matrice de confusion:

[9]:
```
mc1 <- table(Predicted = p1, Actual = test$class)
mc1
```

```
         Actual
Predicted  b  g
        b 40  3
        g  2 60
```

2.2 Optimisation des hyperparamètres de la forêt

Afin d'optimiser notre modèle de forêt aléatoire, il est nécessaire de bien comprendre comment elles sont générées et les paramètres qui influent sur cette génération. Intéressons-nous désormais de plus près à notre modèle de forêt:

Les forêts aléatoires sont composées (comme le terme "forêt" l'indique) d'un ensemble d'arbres décisionnels. Ces arbres se distinguent les uns des autres par le sous-échantillon de données sur lequel ils sont entraînés. Ces sous-échantillons sont tirés au hasard (d'où le terme "aléatoire") dans le jeu de données initial. Il y a deux niveaux de hasard : un Bootstrapping avec remise au niveau de l'echantillon d'apprentissage, et un tirage aléatoire des variables explicative par les variables qui interviennent dans le modele.

Par conséquent, il est logique de remarquer que la fonction randomForest peut prendre en entrée plusieurs hyperparamètres qui influencent le modèle en sortie. Parmi eux:

ntree : Nombre d'arbres décisionnels à considérer. Il ne doit pas être trop petit pour s'assurer que chaque ligne (individu) est prédite au moins plusieurs fois.

mtry : Nombre de variables tirées aléatoirement à chaque coupe.

nodesize : Taille minimum des noeuds terminaux. Une valeur élevée implique des arbres créés plus petits (et donc une vitesse de calcul plus rapide).

Un remarque importante pour la suite (optimisation des hyperparamètres): il est possible de prouver, en se basant sur la loi faible des grands nombres, que l'erreur moyenne décroit stochastiquement avec le nombre **ntree** d'arbres décisionnels dans la forêt.

En effet, intuitivement: les arbres d'une forêt aléatoire sont indépendants et identiquement distribués. Les arbres sont distribués de manière identique car chaque arbre est créé à l'aide d'une stratégie de randomisation répétée identiquement pour chaque arbre : Boot-strap sur les données d'apprentissage, puis création de chaque arbre en choisissant la meilleure coupe pour une variable parmi les m variables sélectionnées pour le noeud. Dans l'algorithme de forêt aléatoire, les arbres sont créés sur leur propre sous-échantillon bootstrap sans tenir compte des autres arbres, ils peuvent donc être considérés comme indépendants. (C'est d'ailleurs en ce sens que l'algorithme de la forêt aléatoire est "embarassingly parallel": on peut paralléliser la construction des arbres car chaque arbre est créé indépendamment, et c'est bien grâce au calcul parallèle que les algorithmes que l'on utilise permettent de retourner des résultats en temps raisonnable).

Dans le cas d'une classification binaire (notre cas), la loi faible des grands nombres s'applique car les arbres sont indépendants et identiquement distribués et que la variable à prédire (**class**) possède une variance finie (en temps que variable binaire à valeur dans {0,1}).

L'application de la loi faible des grands nombres dans ce cas implique que, pour chaque échantillon, l'ensemble va tendre vers une valeur moyenne particulière de prédiction pour cet échantillon lorsque le nombre d'arbres **ntree** tend vers l'infini. De plus, pour un ensemble d'échantillons donnés, une statistique qui nous intéresse sur ces échantillons convergera vers une valeur moyenne lorsque le nombre d'arbres **ntree** tend vers l'infini.

On pourrait penser qu'augmenter **ntree** pourrait causer le sur-apprentissage, mais en réalité il est prouvé que ce n'est pas le cas. Par contre, les autres hyperparamètres (**mtry** et **nodesize**) jouent sur l'over-fitting.

Dès lors, il est inutile d'essayer d'optimiser le paramètre **ntree**, il faut simplement le prendre suffisamment grand pour assurer la convergence de l'erreur.

Afin d'optimiser notre modèle, nous allons prendre un **ntree** assez grand, puis jouer sur les autres hyperparamètres (**mtry** et **nodesize**) pour obtenir une forêt haute performance. Pour cela, nous nous appuyons sur les librairies **mlr** et **caret** :

[9]:
```
library(caret)
library(mlr)
```

La façon de procéder pour sélectionner les hyperparamètres optimaux repose sur la validation croisée. Pour chaque set d'hyperparamètres, on réalise une cross-validation sur nos données train pour avoir une idée du caractère prédictif de la forêt sur des données nouvelles. Nous avons

notamment utilisé cette méthode pour trouver les hyperparamètres optimaux pour la fonction **rpart** dans le TP Arbres de Décisions.

Seulement, nous sommes ici dans un cas plus volumineux avec les forêts aléatoires. En effet, les plages de valeurs prises par les hyperparamètres sur lesquels nous voulons jouer sont larges, et le temps de calcul pour générer chaque forêt est plus conséquence, par conséquent une recherche exhaustive de type **grid-search** peut prendre beaucoup de temps. A la place, on peut considérer une méthode stochastique de type **random-search**: nous allons tirer au hasard un certain nombre fixé de sets d'hyperparamètres à tester puis sélectionner le meilleur d'entre eux à la suite des résultat de la validation croisée.

Nous proposons les deux implémentations ci-dessous, nous avons aupravant utilisé la méthode exhaustive **grid-search** sur le TP Arbres de Décisions, mais ici nous utilisons la méthode stochastique **random-search** pour alléger le temps de calcul.

[449]:
```
set.seed(1234)
d.tree.mlr <- makeClassifTask(
  data=train,
  target="class")

# Méthode de controle: random-search avec 300 triplets d'hyperparamètres tirés␣
 ↪aléatoirement parmi les plages possibles
control_random=makeTuneControlRandom(maxit = 300L)

# Méthode de controle exhaustive: grid-search avec tous les triplets␣
 ↪d'hyperparamètres dans les plages possibles
# Donnée en commentaire mais non utilisée ici
#control <- trainControl(method="repeatedcv", number=10, repeats=3)
#control_grid = makeTuneControlGrid()

# Méthode de validation: cross-validation à 7 blocs
resample = makeResampleDesc("CV", iters = 7L)

# Mesure à considérer: la précision des prédictions
measure = acc

# Hyperparamètres à considérer et leur plages de valeurs
param_grid_multi <- makeParamSet(
  makeDiscreteParam("mtry", values=1:30),
  makeDiscreteParam("nodesize", values=1:40),
  makeDiscreteParam("ntree", values=500:500)

)
```

[137]:
```
dt_tuneparam_multi <- tuneParams(learner='classif.randomForest',
                                 task=d.tree.mlr,
                                 resampling = resample,
                                 measures = measure,
```

```
                                    par.set=param_grid_multi,
                                    control=control_random,
                                    show.info = FALSE)
dt_tuneparam_multi
```

```
Tune result:
Op. pars: mtry=5; nodesize=14; ntree=500
acc.test.mean=0.9227891
```

Testons notre modèle avec les hyperparamètres optimisés:

[428]:
```
rf2 <- randomForest(as.factor(class) ~ .
 ↪,ntree=500,mtry=5,nodesize=14,data=train)
p2=predict(rf2,test)
accuracy(test$class, p2)
```

0.980952380952381

Cela signifie qu'avec ce modèle, 98% des individus de l'ensemble test ont bien été classifiés. Construisons la matrice de confusion :

[429]:
```
mc2 <- table(Predicted = p2, Actual = test$class)
mc2
```

```
         Actual
Predicted  b  g
        b 42  2
        g  0 61
```

2 individus tests on été mal prédits. Les deux sont en réalité "good" et ont été prédits comme étant "bad".

Remarque: Afin de construire ce modèle, nous nous sommes basées sur la cross-validation car c'est une méthode de validation plus robustes vis-à-vis de la fluctuation dues au hasard et aux échantillons. Si, dans une optique "challenge", l'on souhaite uniquement augmenter l'accuracy sur notre ensemble test sans nous intéresser à la robustesse du modèle, le hasard peut jouer en notre faveur. En prenant un nombre **ntree** petit, les variations dûes au hasard autour de la valeur de convergence sont élevées, et on peut arriver à un score d'accuracy meilleur en cas de fluctuation favorable.

Mais cela n'a pas réellement de fondement autre que le hasard, il faut préférer un modèle avec **ntree** grand pour assurer la convergence de l'erreur et ne pas reposer sur un cas particulier si l'on souhaite généraliser le modèle.

On présente un tel modèle à erreur fluctuante qui, par hasard, fournit une précision supérieure du fait de l'aléatoire (l'erreur est fluctuente et le modèle peut donner des précisions variables, puisque **ntree** est petit il n'y a pas convergence comme expliqué plus haut):

[431]:
```
rf3 <- randomForest(as.factor(class) ~ .,mtry=5,ntree=12,nodesize=1,data=train)
p3=predict(rf3,test)
accuracy(test$class, p3)
```

0.990476190476191

Construisons la matrice de confusion associée:

[432]:
```
mc3 <- table(Predicted = p3, Actual = test$class)
mc3
```

```
         Actual
Predicted  b  g
        b 42  1
        g  0 62
```

Un unique individu test a été mal prédit (réalité: good, prédit: bad).

3 Détection des anomalies avec une forêt d'isolement

L'algorithme des forêts d'isolement permet de calculer un score d'anomalie, c'est-à-dire une mesure qui reflète à quel point une donnée peut être considérer comme atypique. Afin de calculer ce score, l'algorithme isole la donnée en question de manière récursive : il choisit aléatoirement une variable puis sélectionne un "seuil de coupure", puis il évalue si cela permet d'isoler la donnée en question ; si tel est le cas, l'algorithme s'arrête, sinon il choisit une autre variable et un autre seuil de coupure, et ainsi de suite jusqu'à ce que la donnée soit isolée du reste.

On s'appuie sur la librairie IsolationForest:

[10]:
```
library(IsolationForest)
```

```
IsolationForest 0.0-26
```

3.1 Forêt d'isolement non paramétrée

On construit une forêt d'isolement avec les hyperparamètres par défaut:

num_trees=100 : nombre d'arbres construits dans la forêt.

nRowSamp=nrow(train) : taille du sous-échantillon, doit être inférieure ou égale à la taille de l'échantillon train.

rFactor=1 : facteur de randomisation, variant de 0 (déterminisme total) à 1 (aléatoire total).

hlim=(ceiling(log2(nrow(train))) : limite de la hauteur maximale de la forêt.

nmin= 1 : nombre minimum d'échantillons pour former un noeud terminal.

[11]:
```
set.seed(1234)

iso <- IsolationTrees(train)

# calcul des scores d'anomalie des données test
anomaly_scores <- AnomalyScore(test, iso)$outF
```

Les indices des observations avec les 5 plus hauts scores d'anomalie sont les suivants:

[12]:
```
hauts_scores_indices <- order(anomaly_scores, decreasing=T)
hauts_scores_indices[1:5]
```

1. 10 2. 26 3. 76 4. 20 5. 21

Ces indices sont les indices des individus en tant qu'éléments de la liste **test**, retrouvons les indices de base (en tant qu'élément du dataset d'origine) de ces individus:

[14]:
```
library(rlist)
```

[16]:
```
ind <- 1:351
index_test=ind[!ind %in% index]
li=list()
for (j in hauts_scores_indices[1:5]) {
  li[length(li)+1]=index_test[j]
}
t(li)
```

A matrix: 1×5 18 78 233 54 56

Les scores des anomalies associées sont respectivement :

[17]:
```
anomaly_scores[hauts_scores_indices[1:5]]
```

1. 0.690986387736716 2. 0.677239204205587 3. 0.660860545942482 4. 0.659016724721649 5. 0.658379724467345

Les indices des observations avec les 5 plus bas scores d'anomalie sont les suivants:

[18]:
```
bas_scores_indices <- order(anomaly_scores, decreasing=F)
bas_scores_indices[1:5]
```

1. 48 2. 89 3. 98 4. 99 5. 77

Ces indices sont les indices des individus en tant qu'éléments de la liste **test**, retrouvons les indices de base (en tant qu'élément du dataset d'origine) de ces individus:

[19]:
```
li=list()
for (j in bas_scores_indices[1:5]) {
  li[length(li)+1]=index_test[j]
}
t(li)
```

A matrix: 1×5 162 294 331 332 238

Les scores des anomalies associées sont respectivement :

[20]:
```
anomaly_scores[bas_scores_indices[1:5]]
```

1. 0.347659958916313 2. 0.347659958916313 3. 0.353009451140308 4. 0.354174374012211 5. 0.355698637942661

Seulement, une fois en possession de ces résultats, on ne peut comme dans les parties précédentes calculer un taux d'erreur ou de précision. La raison est simple: il s'agit ici d'un **modèle non supervisé**. En effet, nos données ne sont pas labelisées: on ne sait pas lesquels parmi elles sont effectivement des outliers.

On peut cependant tout de même réaliser une sorte de vérification, proposons plusieurs méthodes:

Une manière de faire serait d'extraire échantillon de taille raisonnable, calculer les scores anomalies sur cet échantillon, et demander à un expert de confirmer les résultats en labelisant cet échantillon par des moyens autres que statistiques: on tombe alors sur un modèle dit **semi-supervisé**. Malheureusement, nous ne disposons pas d'experts pour réaliser cette méthode dans notre cas.

Une seconde méthode serait d'étudier la différence de la variance, de certaines variables, avec et sans les anomalies et choisir la combinaison d'hyperparamètres qui donne le plus grand nombre de variables avec une différence de variance statistiquement significative. En effet, le retrait des anomalies a forcément pour effet une diminution de la variance (écarts à la moyenne en général importants pour les outliers). Afin de mesurer si une différence de variance est statistiquement significative on se base sur une test d'ANOVA appelé **Fligner Killeen** (basé sur le test de **Levene**), qui teste l'hypothèse H_0 supposant que les variances de deux échantillons sont égales et fournit une **p-valeur**: si la p-valeur est inférieure à 0.05, on peut conclure que le test est significatif et que les variances sont différentes. Essayons cette méthode de validation :

[38]:
```
set.seed(1234)
iso1 <- IsolationTrees(train,ntree=500,hlim=200,nmin=5,rFactor=0,nRowSamp=3)
anomaly_scores1 <- AnomalyScore(test, iso1)$outF
hauts_scores_indices1 <- order(anomaly_scores1, decreasing=T)
bas_scores_indices1 <- order(anomaly_scores1, decreasing=F)
```

Observons les p-valeur du test sur les variables avant et après avoir retiré les individus au plus haut score d'anomalie:

[39]:
```
for (i in 1:32){
  sample1=test[,i]
  sample2=test[-hauts_scores_indices1[1:10],i]
  y <- c(sample1, sample2)
  group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
  print(fligner.test(y,group)$p.value)
}
```

```
[1] 0.1497913
[1] 0.2811646
[1] 0.2266602
[1] 0.9243086
[1] 0.4969533
[1] 0.7474381
[1] 0.172346
[1] 0.6553336
[1] 0.6429735
[1] 0.3454602
[1] 0.7574907
```

```
[1] 0.7618899
[1] 0.4997442
[1] 0.8555822
[1] 0.7142917
[1] 0.372936
[1] 0.591967
[1] 0.7071677
[1] 0.8659419
[1] 0.9617319
[1] 0.6819971
[1] 0.9443135
[1] 0.2523647
[1] 0.2213158
[1] 0.7440451
[1] 0.5130427
[1] 0.4763579
[1] 0.3917741
[1] 0.3831919
[1] 0.295928
[1] 0.8731856
[1] 0.9578232
```

Aucune des p-valeur ne permet de conclure à une différence de variance significative.

Essayons une autre combinaison des hyperparamétres d'isolationTree qui est ntree=500,hlim=100,nmin=1,rFactor=1,nRowSamp=3 :

[44]:
```
set.seed(1234)
iso2 <- IsolationTrees(train,ntree=500,hlim=100,nmin=5,rFactor=1,nRowSamp=3)
anomaly_scores2 <- AnomalyScore(test, iso2)$outF
hauts_scores_indices2 <- order(anomaly_scores2, decreasing=T)
```

Au lieu de regarder les p-valeurs une par une, on écrit une fonction qui compte le nombre de variables observant une différence significative de variance après retrait des outliers:

[45]:
```
compteur<-function(){
  cpt=0
  for (i in 1:32)
  {
    sample1=test[-hauts_scores_indices2[1:10],i]
    sample2=test[-hauts_scores_indices2[1:10],i]
    y <- c(sample1, sample2)
    group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
    if (fligner.test(y,group)$p.value<0.05)
    {cpt<-cpt+1}
  }
  print(cpt)}
compteur()
```

```
[1] 0
```

Encore une fois, le test statistique de différence de variance n'est pas significatif pour cette combinaison d'hyperparametres.

On a essayé une cinquantaine de combinaison d'hyperparamètres diversifiées, et le test de flinger ne devient siginficatif que lorsqu'on enleve un nombre important d'observations (à peu prés 20%) ce qui n'est pas concluant car nous retirons alors plus que les possibles outliers.

La forêt d'isolement retire effectivement des individus et diminue la variance, pour s'en convaincre, observons la somme des differences des variances des variables aves et sans les observations aberrantes:

[46]:
```
sum(sapply(test[1:32], var)-sapply(test[1:32][-hauts_scores_indices1[1:10],],␣
↪var))
sum(sapply(test[1:32], var)-sapply(test[1:32][-hauts_scores_indices2[1:10],],␣
↪var))
```

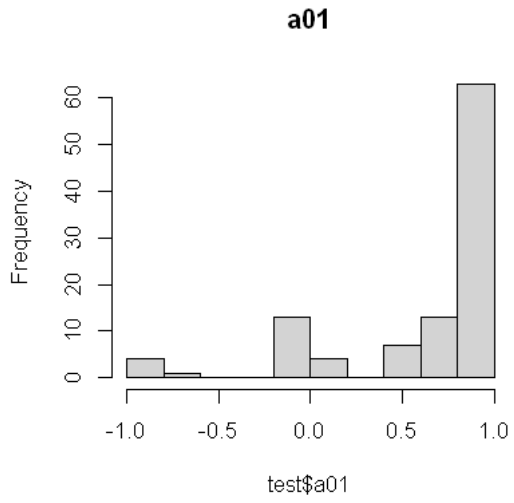
1.3331077900871

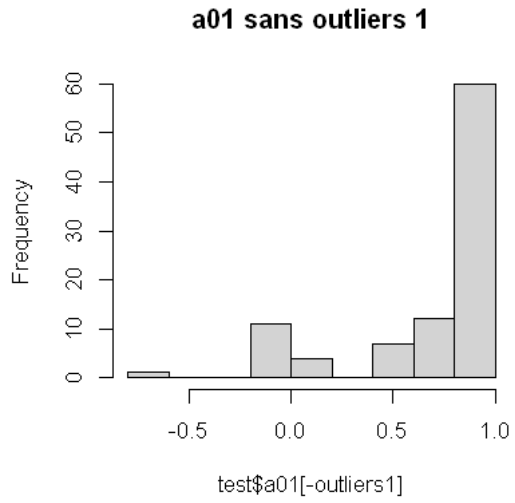
1.84226583138508

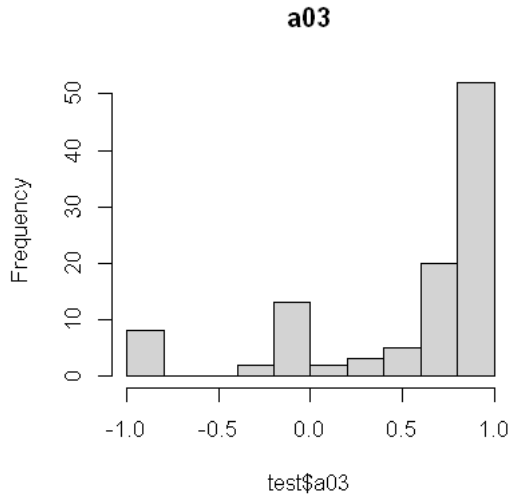
On voit bien une amélaioration, mais cette amélioration n'est pas significative et visible pour chaque variable une par une avec nos tests ANOVA (que ce soit le test de Flinger ou de Levene, issu de la librairie **car**).

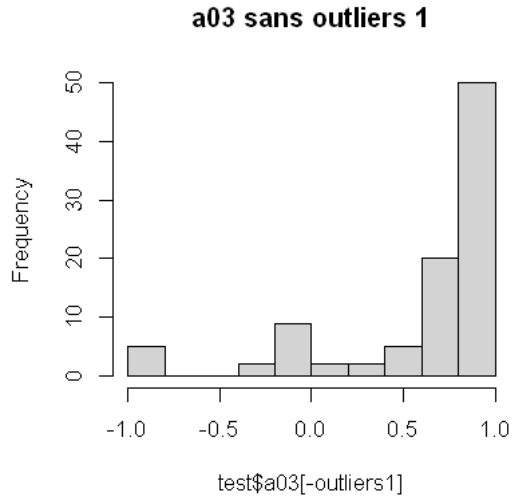
Une autre approche serait de comparer visuelement les distributions des variables qui nous semblent impactées par le retrait des outliers, en visualisant leur répartition avec et sans les observations aberrantes :

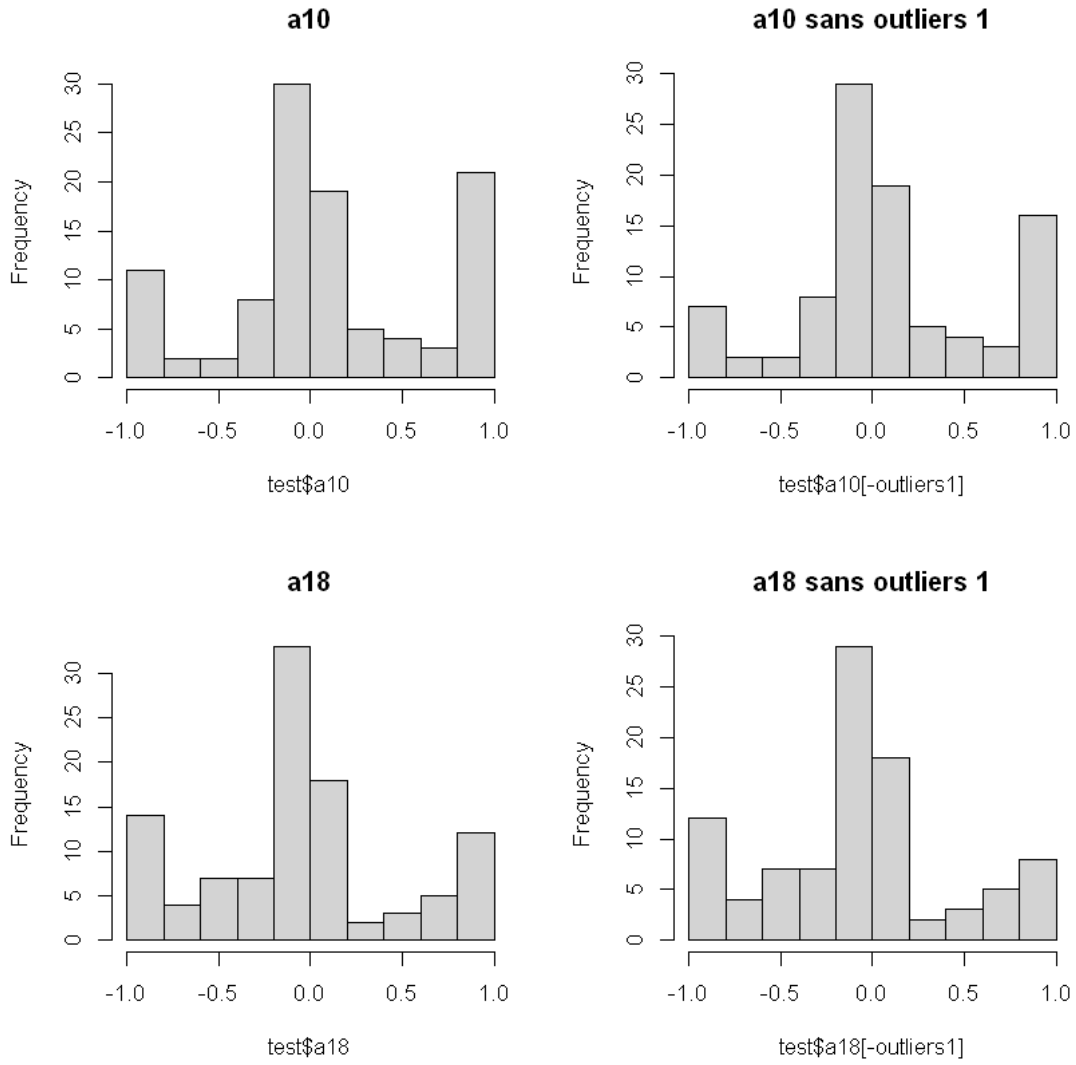
[47]:
```
outliers1=hauts_scores_indices1[1:10]
outliers2=hauts_scores_indices2[1:10]


par(mfrow=c(2,2))
hist(test$a01,main='a01')
hist(test$a01[-outliers1],main='a01 sans outliers 1')
hist(test$a03,main='a03')
hist(test$a03[-outliers1],main='a03 sans outliers 1')
par(mfrow=c(2,2))
hist(test$a10,main='a10')
hist(test$a10[-outliers1],main='a10 sans outliers 1')
hist(test$a18,main='a18')
hist(test$a18[-outliers1],main='a18 sans outliers 1')
```


a01
Frequency
0
10
20
30
40
50
60
-1.0
-0.5
0.0
0.5
1.0
test$a01


a01 sans outliers 1
Frequency
0
10
20
30
40
50
60
-0.5
0.0
0.5
1.0
test$a01[-outliers1]


a03
Frequency
0
10
20
30
40
50
-1.0
-0.5
0.0
0.5
1.0
test$a03

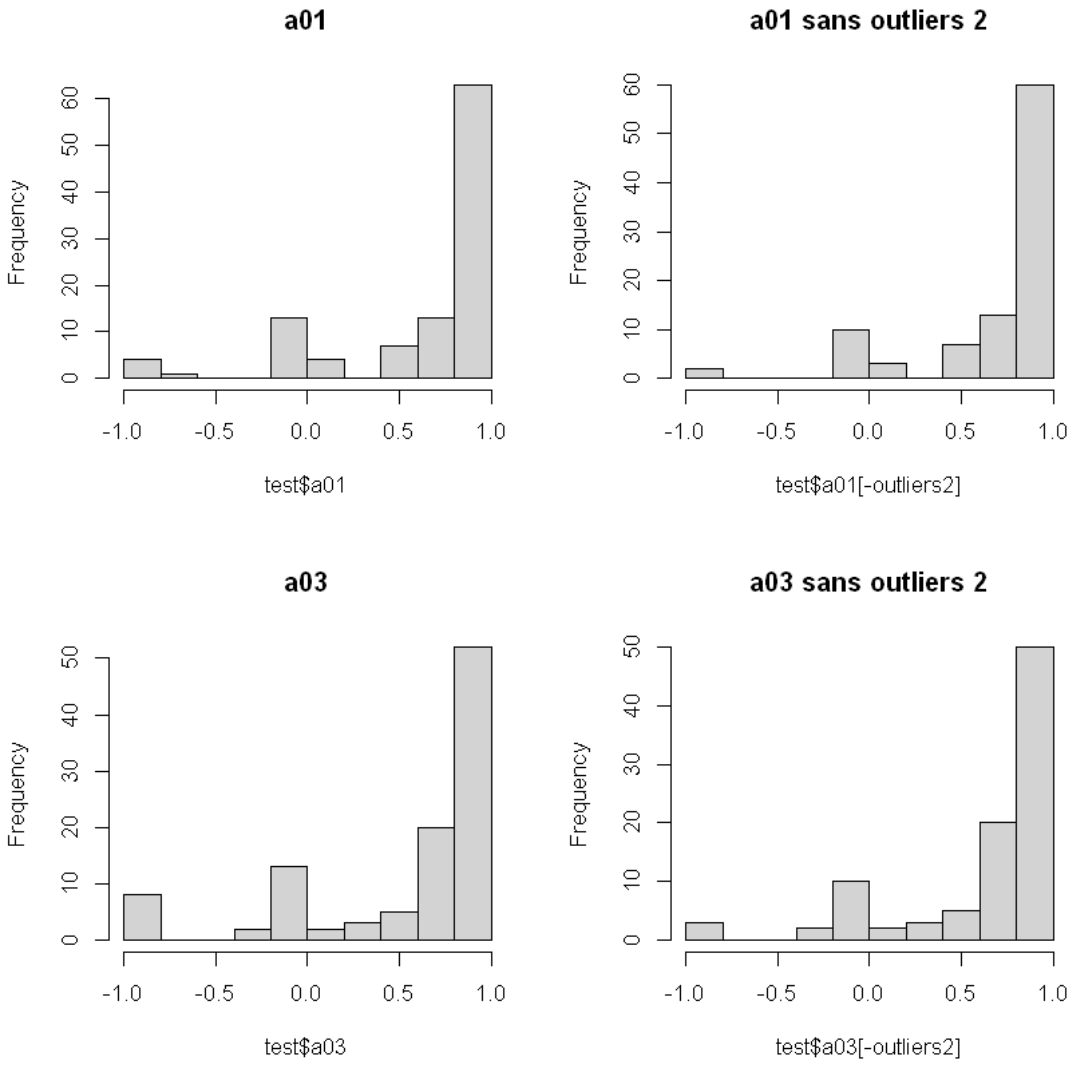

a03 sans outliers 1
Frequency
0
10
20
30
40
50
-1.0
-0.5
0.0
0.5
1.0
test$a03[-outliers1]

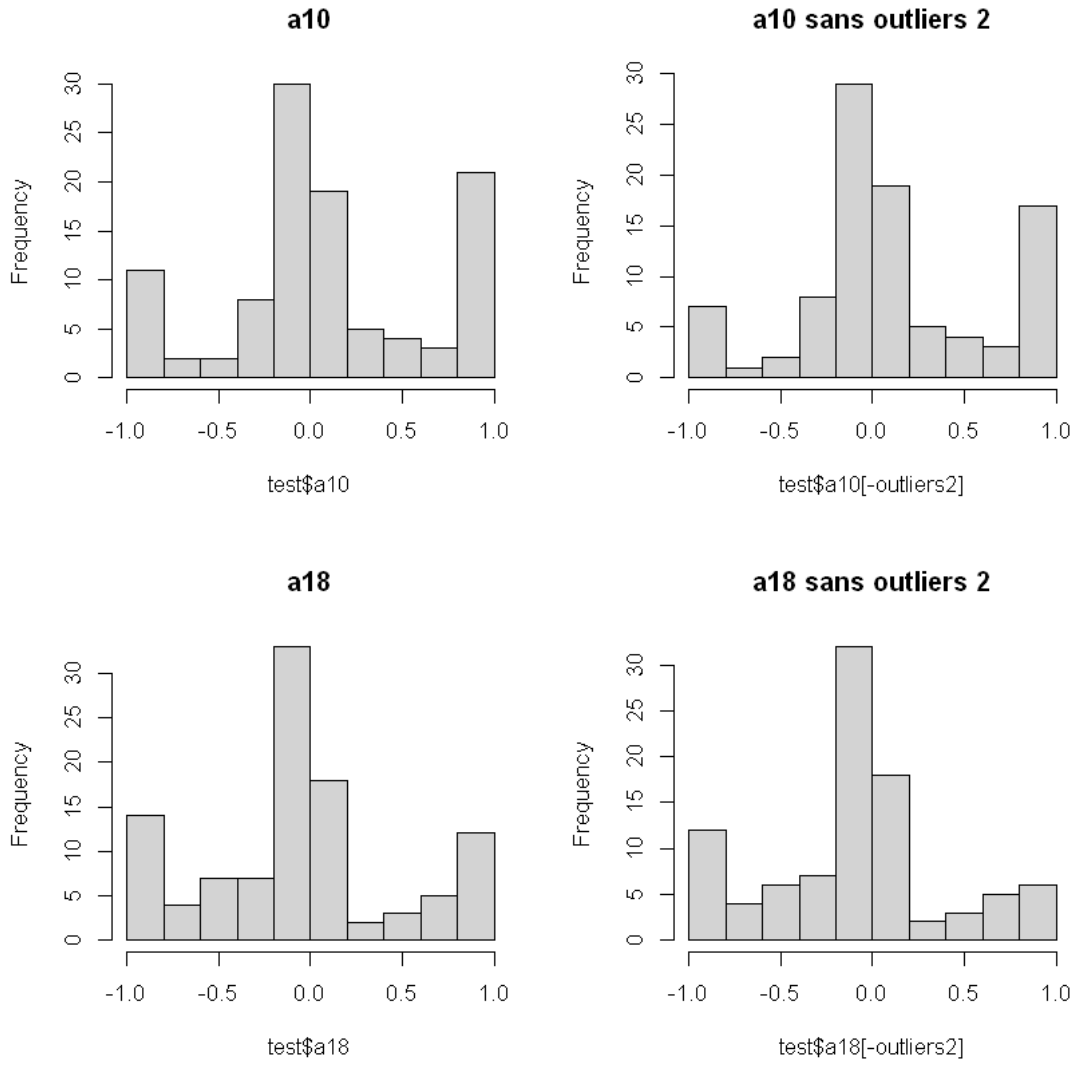


On voit bien que le retrait des outliers permet de légèrement diminuer la distribution sur les extrêmes ainsi qu'au centre, c'est cohérent du fait que les outliers sont souvent introduits par erreur et ont des valeurs bien souvent absurdes (maximale, minimale ou nulle). Observons cela pour les outliers issues du second set d'hyper-paramètres présenté:

[48]:
```
par(mfrow=c(2,2))
hist(test$a01,main='a01')
hist(test$a01[-outliers2],main='a01 sans outliers 2')
hist(test$a03,main='a03')
hist(test$a03[-outliers2],main='a03 sans outliers 2')


par(mfrow=c(2,2))
```

```
hist(test$a10,main='a10')
hist(test$a10[-outliers2],main='a10 sans outliers 2')
hist(test$a18,main='a18')
hist(test$a18[-outliers2],main='a18 sans outliers 2')
```





Même constat que précédemment, on remarque que qu'il ya une difference legere entre les distributions de ces 4 variables avec et sans outliers au centre et aux extrêmes. Mais cela reste une comparaison visuelle, qui ne suffit pas réellement pour valider le modele lorsque les test statistiques ne sont pas significatifs dans ce contexte et permet encore moins de juger quels hyperparamètres sont les plus efficaces et de sélectionner les hyperparamètres optimaux.

Finalement, on se rend compte de la difficulté de superviser un modèle qui par définition est **non supervisé**, surtout lorsque les tests statistiques ne sont pas significatifs. Enfin, tentons une dernière approche pour motiver le choix des hyperparamètres de notre forêt.

Une autre manière de faire repose sur l'ACP. En effet, la projection sur le premier plan factoriel de nos données permet souvent de remarquer à vue d'oeil les données dites "aberrantes". On désigne alors ici par données "aberrantes" des données qui, bien que correctement représentées par les deux premières composantes principales (elles suivent donc le même "modèle" que le reste des individus),

possèdent des valeurs dans les extrêmes. Pour comprendre ce concept simplement, on peut donner l'exemple simple d'un set d'individus décrits par deux variables "poids" et "taille". En supposant dans le cas le plus simple que le poids est une fonction affine de la taille, tous les être humains seront sur une droite qui sera décrite par le premier axe factoriel. Un homme extrêmement grand sera alors correctement projeté sur ce plan, mais apparaîtra dans les valeurs extrêmes pour cette composante. C'est ce que l'on désigne par des données aberrantes **extrêmes**.

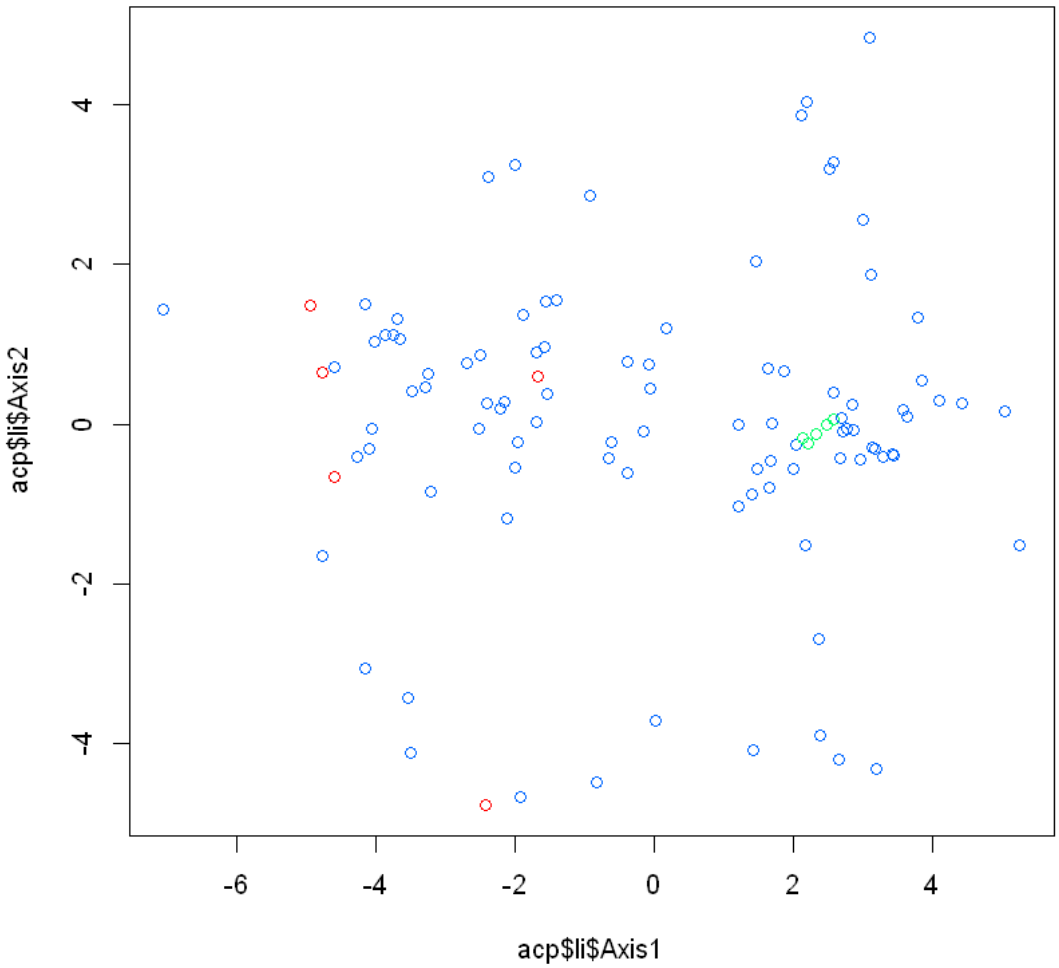
On verra dans la suite que cette définition des données aberrantes est limitée, car en réalité, il y a un autre type de données que l'on pourrait considérer comme aberrantes: les données qui ne suivent pas du tout le même modèle que les autres, et qui seront donc par conséquent mal représentés par les premières composantes principales de l'ACP. Si l'on reprend notre exemple simple du paragraphe précédent, ce serait alors le cas si l'on avait un set de valeurs issues de mesures faites sur un animal (par exemple une souris), alors que les autres individus sont humains. Il sera donc mal projeté sur les premières composantes principales, et aller chercher ces individus mal projeté est une manière de faire pour détecter ce type d'outliers "non conformes au modèle" via l'ACP. On étudiera cette méthode dans la partie suivante.

Pour l'instant, on se limitera à représenter dans le premier plan factoriel les outliers détectés par notre forêt d'isolement, en espérant qu'ils soient effectivement des outliers "aux valeurs extrêmes" (et qui seront donc isolés sur le premier plan factoriel).

Réalisons donc l'ACP:

[51]:
```
library(ade4)
library(rgl)
acp <- dudi.pca(test[,-33], center = T,scannf=FALSE, nf=3)
```

[81]:
```
couleurs <- rep(rainbow(5)[4], nrow(test))
couleurs[hauts_scores_indices[1:5]] <- rainbow(5)[1]
couleurs[bas_scores_indices[1:5]] <- rainbow(5)[3]
plot(acp$li$Axis1, acp$li$Axis2, col = couleurs)
```



En rouge sont représentées les 5 observations avec les plus hauts scores d'anomalie, que l'on considère comme outliers. On remarque que 3 des observations parmi les 5 sélectionnées comme outliers semblent effectivement isolées sur le 1er plan factoriel. Le fait que des supposés outliers soit situé plus au centre que les autres peut signifier qu'ils présentent un comportement isolé sur d'autres composantes factoriels que les deux premières, ou que notre modèle de forêt d'isolement n'est pas correctement paramétré pour nos données.

En vert sont représentées les 5 observations avec les plus bas scores d'anomalie, on remarque qu'elles sont regroupées avec une grande partie des observation dans un cluster.

Remarquons que le choix du seuil des 5 observations pour les considérés comme outlier est arbitraire, c'est là à nouveau un paramètre à modifier pour l'adapter à nos données. Souvent, on utilise un taux appelé **contamination rate** entre 0 et 1 qui donne le pourcentage des observations ayant les plus hauts scores d'anomalies que l'on choisit de considérer comme outliers.

Une autre façon de conforter nos résultats repose sur l'analyse et la comparaison de la variance des observations sur les composantes principales avant et après avoir retirer les outliers.

Avant de retirer les outliers, on a les variances suivantes sur les deux premières composantes principales:

[52]:
```
var(acp$li$Axis1)
var(acp$li$Axis2)
```

8.50719290149094

3.42185672978941

Après avoir retiré les outliers, on a les variances suivantes sur les deux premières composantes principales:

[53]:
```
var(acp$li$Axis1[-hauts_scores_indices[1:5]])
var(acp$li$Axis2[-hauts_scores_indices[1:5]])
```

7.9622278664635

3.31841242410023

[55]:
```
sample1=acp$li$Axis1[-hauts_scores_indices[1:5]]
sample2=acp$li$Axis1
y <- c(sample1, sample2)
group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
fligner.test(y,group)
```

```
        Fligner-Killeen test of homogeneity of variances

data:  y and group
Fligner-Killeen:med chi-squared = 0.014963, df = 1, p-value = 0.9026
```

[56]:
```
sample1=acp$li$Axis2[-hauts_scores_indices[1:5]]
sample2=acp$li$Axis2
y <- c(sample1, sample2)
group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
fligner.test(y,group)
```

```
        Fligner-Killeen test of homogeneity of variances

data:  y and group
Fligner-Killeen:med chi-squared = 0.081151, df = 1, p-value = 0.7757
```

On observe donc que la variance diminue après avoir retiré les outliers. Par exemple, la baisse en pourcentage sur la 2ème composante est due au fait que l'on a considéré comme outlier le point le plus extrémal sur l'axe y dans le premier plan factoriel comme on a vu sur la projection. Néanmoins,

bien que les p-valeurs soient plus représentatives que dans l'étude sans ACP, elles ne sont toujours pas sous le seuil significatif des $0,5$.

Cela conforte notre résultat du fait que souvent, les données dites "aberrantes" augmentent la variance en se situant sur des extrêmes. Néanmoins, ce modèle ne semble pas assez adapté à nos données du fait de la p-valeur, essayons d'optimiser les hyperparamètres avec cette méthode.

3.2 Modification des hyperparamètres pour s'adapter à nos données

Bien qu'il s'agisse d'un modèle non supervisé, les différentes méthodes de validations décrites plus tôt nous pousse à nous intéresser aux hyperparamètres que l'algorithme des forêts d'isolement prend en entrée pour adapter le modèle au mieux à nos données.

Rappelons les différents hyperparamètres et leur role dans la construction de la forêt:

num_trees: nombre d'arbres construits dans la forêt.

nRowSamp=: taille du sous-échantillon, doit être inférieure ou égale à la taille de l'échantillon train.

rFactor=1 : facteur de randomisation, variant de 0 (déterminisme total) à 1 (aléatoire total).

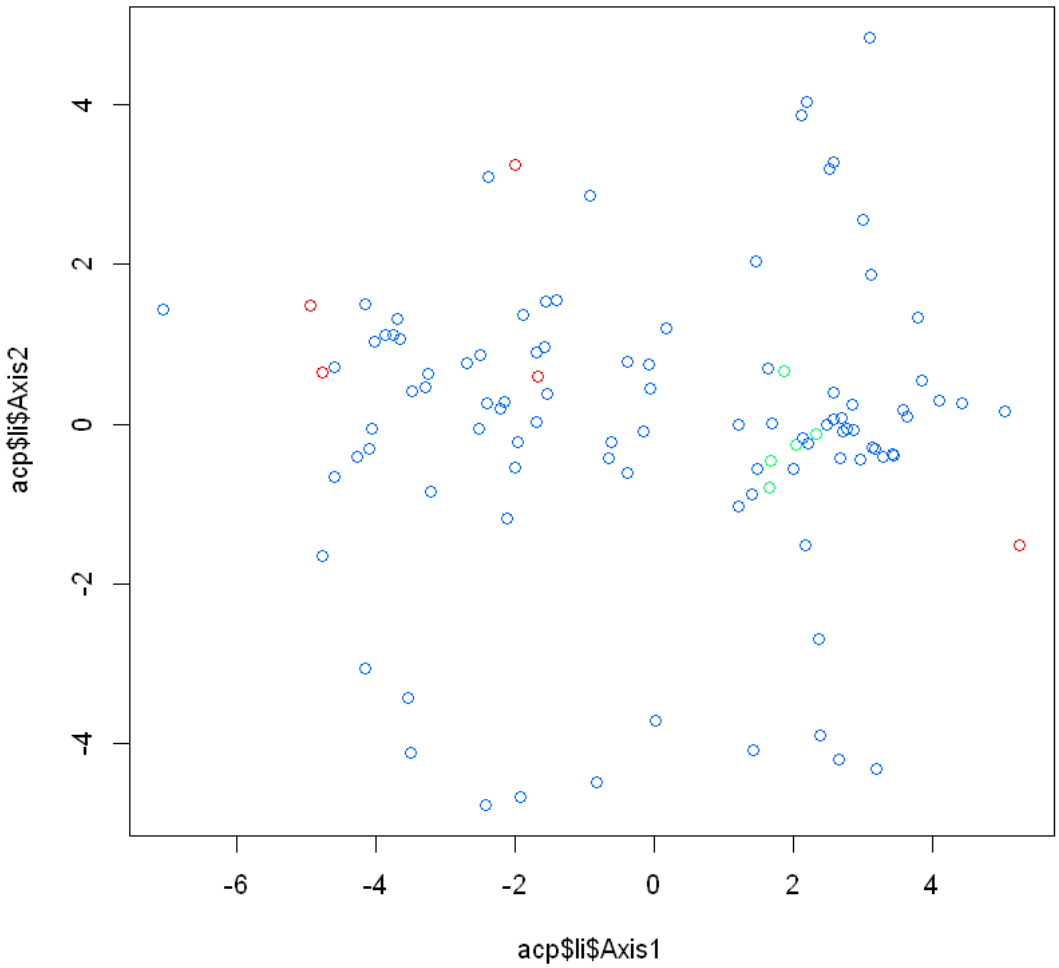
hlim : limite de la hauteur maximale de la forêt.

nmin : nombre minimum d'échantillons pour former un noeud terminal.

Observons le comportement lorsque nous réduisons **num_tree** et **hlim**:

[58]:
```
set.seed(1234)

iso <- IsolationTrees(x=train,ntree=10,hlim=3)

# calcul des scores d'anomalie des données test
anomaly_scores <- AnomalyScore(test, iso)$outF

# observations avec les plus hauts scores d'anomalies et les plus bas scores
hauts_scores_indices <- order(anomaly_scores, decreasing=T)
bas_scores_indices <- order(anomaly_scores, decreasing=F)
# ACP et projection sur le 1er plan factoriel
acp <- dudi.pca(test[,-33], center = T,scannf=FALSE, nf=3)
```

[85]:
```
couleurs <- rep(rainbow(5)[4], nrow(test))
couleurs[hauts_scores_indices[1:5]] <- rainbow(5)[1]
couleurs[bas_scores_indices[1:5]] <- rainbow(5)[3]
plot(acp$li$Axis1, acp$li$Axis2, col = couleurs)
```



On observe à vue d'oeil sur le 1er plan factoriel que ce modèle semble moins bien décrire les outliers dits "extremaux" que le 1er. En réduisant la taille et la hauteur de la forêt, celle-ci semble être moins précise et adaptée à nos données.

Réalisons le test d'égalité de variance sur les deux premières composantes:

[59]:
```
sample1=acp$li$Axis1[-hauts_scores_indices[1:5]]
sample2=acp$li$Axis1
y <- c(sample1, sample2)
group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
fligner.test(y,group)
```

```
        Fligner-Killeen test of homogeneity of variances
```

```
data:  y and group
Fligner-Killeen:med chi-squared = 0.071412, df = 1, p-value = 0.7893
```

[60]:
```r
sample1=acp$li$Axis2[-hauts_scores_indices[1:5]]
sample2=acp$li$Axis2
y <- c(sample1, sample2)
group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
fligner.test(y,group)
```

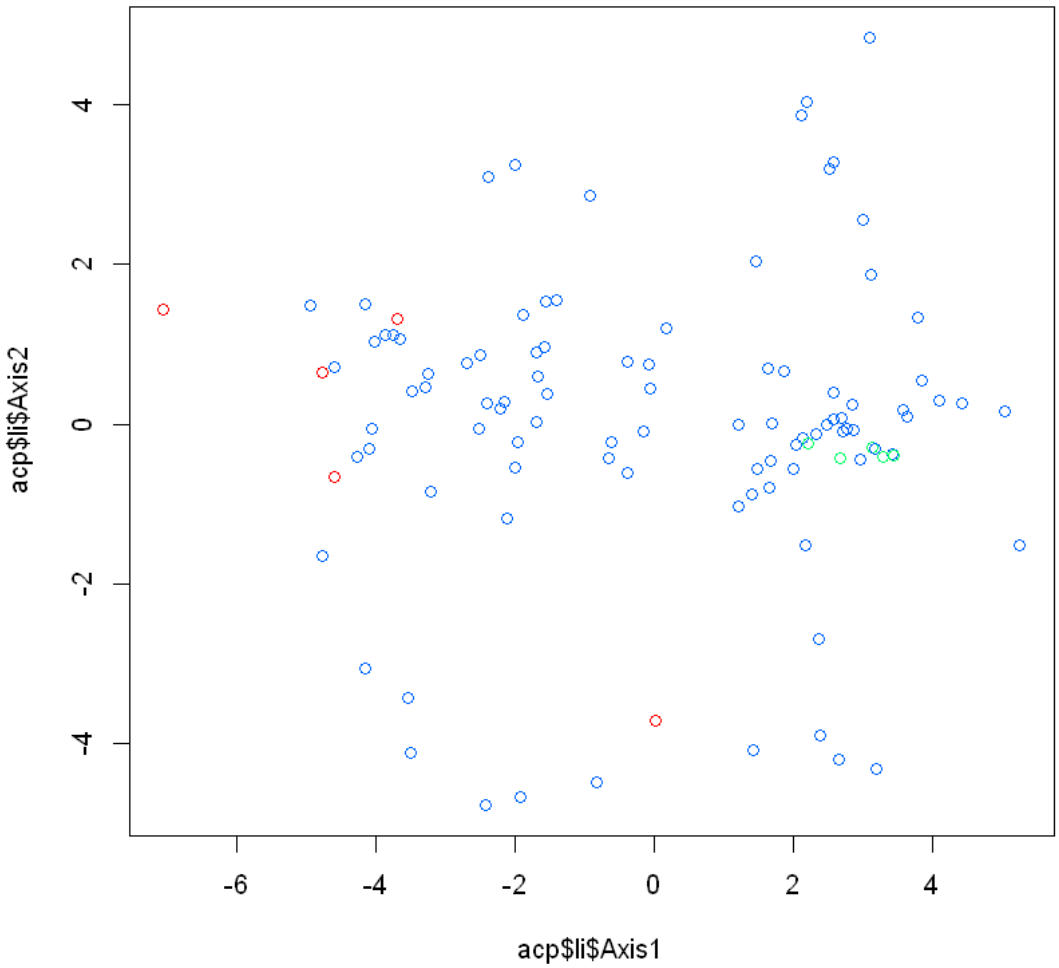
```
        Fligner-Killeen test of homogeneity of variances

data:  y and group
Fligner-Killeen:med chi-squared = 0.029655, df = 1, p-value = 0.8633
```

Les p-valeurs confirment que le modèle n'est pas aussi bien adapté.

Observons le comportement lorsque nous augmentons **nmin**:

[61]:
```r
set.seed(1234)

iso <- IsolationTrees(x=train,nmin=10)

# calcul des scores d'anomalie des données test
anomaly_scores <- AnomalyScore(test, iso)$outF

# observations avec les plus hauts scores d'anomalies et les plus bas scores
hauts_scores_indices <- order(anomaly_scores, decreasing=T)
bas_scores_indices <- order(anomaly_scores, decreasing=F)

# ACP et projection sur le 1er plan factoriel
acp <- dudi.pca(test[,-33], center = T,scannf=FALSE, nf=3)
```

[87]:
```r
couleurs <- rep(rainbow(5)[4], nrow(test))
couleurs[hauts_scores_indices[1:5]] <- rainbow(5)[1]
couleurs[bas_scores_indices[1:5]] <- rainbow(5)[3]
plot(acp$li$Axis1, acp$li$Axis2, col = couleurs)
```



On observe qu'en augmentant **nmin** le nombre minimum d'échantillons pour former un noeud terminal, certains des outliers les plus isolés sont désormais détectés par la forêt. A nouveau, il est important de remarquer qu'il ne s'agit pas d'une règle générale mais d'une observation spécifique à nos données et à ce modèle.

[64]:
```
sample1=acp$li$Axis1[-hauts_scores_indices[1:5]]
sample2=acp$li$Axis1
y <- c(sample1, sample2)
group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
fligner.test(y,group)
```

```
	Fligner-Killeen test of homogeneity of variances
```

```
data:  y and group
Fligner-Killeen:med chi-squared = 0.016956, df = 1, p-value = 0.8964
```

[63]:
```r
sample1=acp$li$Axis2[-hauts_scores_indices[1:5]]
sample2=acp$li$Axis2
y <- c(sample1, sample2)
group <- as.factor(c(rep(1, length(sample1)), rep(2, length(sample2))))
fligner.test(y,group)
```

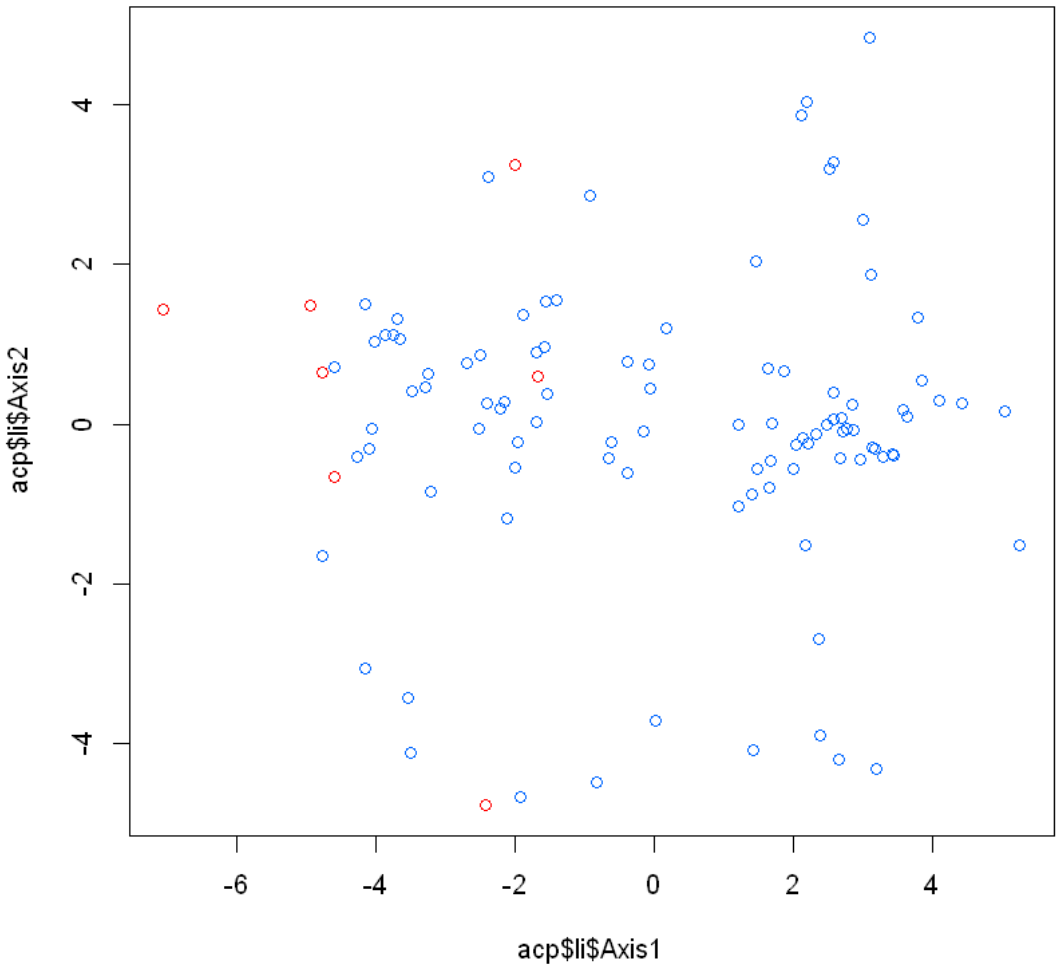
```
        Fligner-Killeen test of homogeneity of variances

data:  y and group
Fligner-Killeen:med chi-squared = 0.028497, df = 1, p-value = 0.8659
```

Les p-valeur ne nous permettent toujours pas de valider le modèle. On se rend compte (après une cinquantaine de tests) que cette méthode quantitative de validation ne nous a jamais permis d'obtenir une p-valeur inférieure à 5% en retirant simplement les 5 individus avec le plus grand score d'anomalie. On se concentre sur l'aspect visuel des outliers dans le premier plan factoriel pour optimiser notre modèle.

Finalement, en jouant sur les différents types d'hyperparamètres, on tente d'obtenir un modèle qui s'adapte bien à nos données:

[71]:
```r
set.seed(1234)
iso <- IsolationTrees(x=train,ntree=1000,hlim=5,nmin=1,rFactor=1,nRowSamp=1)

# calcul des scores d'anomalie des données test
anomaly_scores <- AnomalyScore(test, iso)$outF

# observations avec les plus hauts scores d'anomalies et les plus bas scores
hauts_scores_indices <- order(anomaly_scores, decreasing=T)
bas_scores_indices <- order(anomaly_scores, decreasing=F)

# ACP et projection sur le 1er plan factoriel
acp <- dudi.pca(test[,-33], center = T,scannf=FALSE, nf=3)
contamination_rate <- 0.07
```

[89]:
```r
couleurs <- rep(rainbow(5)[4], nrow(test))
couleurs[hauts_scores_indices[1:round(nrow(test)*contamination_rate)]] <-␣
 ↪rainbow(5)[1]
plot(acp$li$Axis1, acp$li$Axis2, col = couleurs)
compteur()
```



Les indices des individus outliers sont les suivants, en tant qu'éléments de **test**:

[91]:
```
hauts_scores_indices[1:7]
```

1. 10 2. 20 3. 68 4. 51 5. 26 6. 21 7. 53

Et en tant qu'éléments du dataset d'origine, ils ont pour indice:

[92]:
```
li=list()
for (j in hauts_scores_indices[1:7]) {
  li[length(li)+1]=index_test[j]
}
t(li)
```

A matrix: 1 × 7 18 54 207 167 78 56 171

Finalement, avec un **contamination rate** (pourcentage fixant seuil de sélection des outliers) de 0.07 et les hyperparamètres sélectionnés, on observe une description assez convenable des outliers extremaux. A nouveau, le fait qu'un point qui semble central soit détecté comme outlier peut être dû au fait qu'il présente des caractéristiques abberrantes sur d'autres composantes principales que les deux premières (par exemple si c'est un autre type d'outlier que ceux décrits plus haut).

Avant de retirer les outliers, on a les variances suivantes sur les trois premières composantes principales:

[72]:
```
var(acp$li$Axis1)
var(acp$li$Axis2)
var(acp$li$Axis3)
```

8.50719290149094

3.42185672978941

2.97201732786015

Après avoir retiré les outliers, on a les variances suivantes:

[73]:
```
var(acp$li$Axis1[-hauts_scores_indices[1:round(nrow(test)*contamination_rate)]])
var(acp$li$Axis2[-hauts_scores_indices[1:round(nrow(test)*contamination_rate)]])
var(acp$li$Axis3[-hauts_scores_indices[1:round(nrow(test)*contamination_rate)]])
```

7.70021515963329

3.26859850079081

2.84253584824507

Finalement, modifier les hyperparamètres et le taux de contamination a permis de réduire la variance des composantes principales, et la projection sur le premier plan factoriel montre des outliers isolés visuellement. Ceci donne une idée de l'efficacité du modèle pour la détection d'anomalie en l'absence de labelisation.

4 ACP sur $\mathbb{R}^p$

Pour réaliser l'ACP, nous passons sur Python car nous avions réalisé dans le TP ACP précédent les scripts sur ce langage. Pour cela, nous importons les fichiers **train** et **test** qui représentent les individus train et test séparés avec le seed 1234 sur R, pour pouvoir travailleur sur les exacts même jeux de données que les parties précédentes.

4.1 Import des modules

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
import pandas as pd
import time
import warnings
```

```
warnings.filterwarnings("ignore")
```

4.2 Import et prétraitement des données

[3]:
```
data = pd.read_csv('Ionosphere.csv', delimiter=';', decimal=".", ␣
 ↪error_bad_lines=False)
```

[4]:
```
train = pd.read_csv('train.csv', delimiter=';',decimal=",", ␣
 ↪error_bad_lines=False)
test = pd.read_csv('test.csv', delimiter=';',decimal=",", ␣
 ↪error_bad_lines=False)
```

On supprime les colonnes inutiles (**unnamed** désigne l'indice et **class** la classe de l'individus, elles ne nous sont pas utiles pour l'ACP):

[5]:
```
X_train=train.drop('class', axis=1)
del X_train['Unnamed: 0']
y_train=train["class"]
X_test=test.drop('class', axis=1)
del X_test['Unnamed: 0']
y_test=test["class"]
```

4.3 Définition des fonctions utiles à l'ACP issues du TP précédent

Les fonctions de centrage et de normalisation des données:

[6]:
```
def centrage(data) :
    data_centree=data.copy()
    for col in data.columns :
        data_centree[col]=(data_centree[col]-data.mean()[col])
    return data_centree

def normalise(data) :
    data_norm=data.copy()
    for col in data.columns :
        data_norm[col]=(data_norm[col]-data.mean()[col])/(np.std(data[col]))
    return data_norm
```

La fonction fournissant les valeurs propres et vecteurs propres des axes factoriels, dans l'ordre décroissant de variance expliqué:

[7]:
```
def hyperplans(data,k=10**3) :
    data_transposee=np.transpose(data)
    p,n=data_transposee.shape
    cov_data=np.dot(data_transposee,data)/n
    ValeursPropres,VecteursPropres = np.linalg.eig(cov_data)

    # ordonner les valeurs propres
```

```
    ordre=ValeursPropres.argsort()[::-1]
    ValeursPropres=ValeursPropres[ordre]
    VecteursPropres=VecteursPropres[:,ordre]
    return ValeursPropres[:k] , VecteursPropres[:k]
```

4.4 ACP sur nos données train

Nous décidons de centrer les données. Et pour cause: dans la 1ère partie, nous avons fait une analyse statistique des différentes variables. Certaines variables ne sont pas de moyenne nulle, et les données doivent nécessairement être au minimum centrée pour réaliser l'ACP afin que la direction privilégiée soit celle augmentant la variance. Néanmoins, les variables sont toutes à valeur dans le même intervalle $[-1, 1]$, avec des écart-types similaires autour de $0,5$, ce qui fait qu'aucune ne prédominera dans l'ACP à cause d'un problème d'échelle ou d'unité par rapport aux autres. Donc il n'est pas nécessaire de normaliser les données dans notre cas.

De plus, normaliser les données poserait un problème lorsqu'il s'agit de détecter les outliers dits "extremaux", c'est-à-dire les individus qui possèdent des valeurs extrêmes sur certaines variables, et qui peuvent être considérés comme aberrantes. En normalisant les données, nous lisserions ces variations extrêmes sur l'ensemble des individus.

Finalement, nous décidons de simplement centrer nos données pour ces raisons.

[9]:
```
data_centree=centrage(X_train)
val_propres_data_centree,vec_propres_data_centree=hyperplans(data_centree)
```

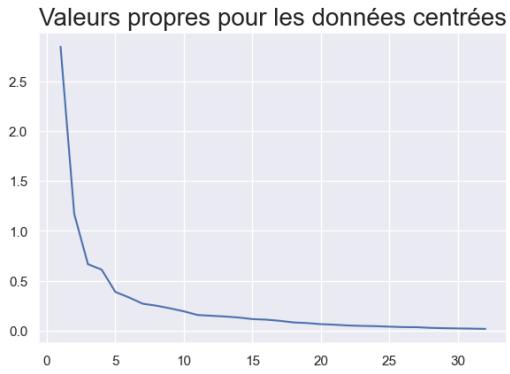
4.4.1 Détermination du nombre de composantes k à retenir

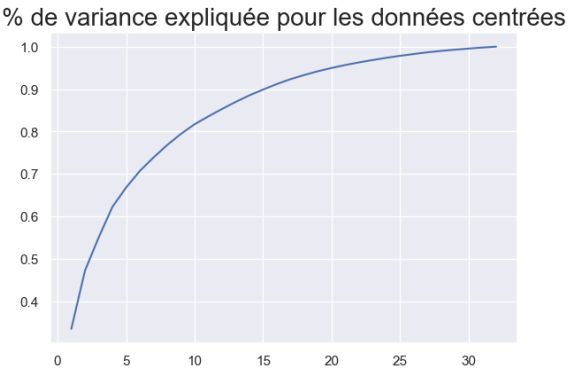
Pour déterminer le nombre k de composantes principales à retenir dans l'ACP, il existe plusieurs règle.

Une première méthode serait de disposer d'un seuil prédéfini de pourcentage de variance expliquée, ce qui n'est pas le cas ici.

Une autre façon de faire s'appelle la règle de **Catelli**. Il s'agit d'observer la courbe de décroissance des valeurs propres et de croissance de la variance expliquée et de fixer k en fonction des cassures:

[10]:
```
plt.figure(figsize=(15,10))
plt.subplot(221)
plt.plot(range(1,len(val_propres_data_centree)+1),val_propres_data_centree)
plt.title("Valeurs propres pour les données centrées",fontsize=20)
plt.subplot(222)
plt.plot(range(1,len(val_propres_data_centree)+1),np.
 ↪cumsum(val_propres_data_centree/(sum(val_propres_data_centree))))
plt.title("% de variance expliquée pour les données centrées",fontsize=20)
plt.show()
```





Nous observons une cassure pour par exemple $k = 3$ et pour $k = 5$, cette délimitation reste subjective.

Considérons alors une autre règle, plus directe: la règle de **Kaiser-Guttman**. Il s'agit de ne retenir que les composantes principales associées à des valeurs propres supérieures à la moyenne des valeurs propres:

[11]:
```
def Kaiser_Guttman(val_propres_data):
    k=0
    val_propre_mean=np.mean(val_propres_data)
    for elt in val_propres_data:
        if elt >= val_propre_mean :
            k+=1
    return k
Kaiser_Guttman(val_propres_data_centree)
```

[11]: 7

La règle de Kaiser-Guttman suggère $k = 7$.

Enfin, étudions une dernière règle, dite **règle de l'éboulis**. Il s'agit de retenir les 2 premiers axes factoriels au moins, puis de "couper" l'éboulis (ou scree plot) des valeurs propres entre les deux valeurs propres successives dont la différence est maximale.

[12]:
```
def eboulis(val_propres_data):
    n=len(val_propres_data)
    if n<=2:
        return n
    else :
        k=2
        diff_max=0
        for j in range(3,n):
            if val_propres_data[j]-val_propres_data[j-1]>=diff_max:
                k=j-1
                diff_max=val_propres_data[j]-val_propres_data[j-1]
```

```
        return k

eboulis(val_propres_data_centree)
```

[12]: 2

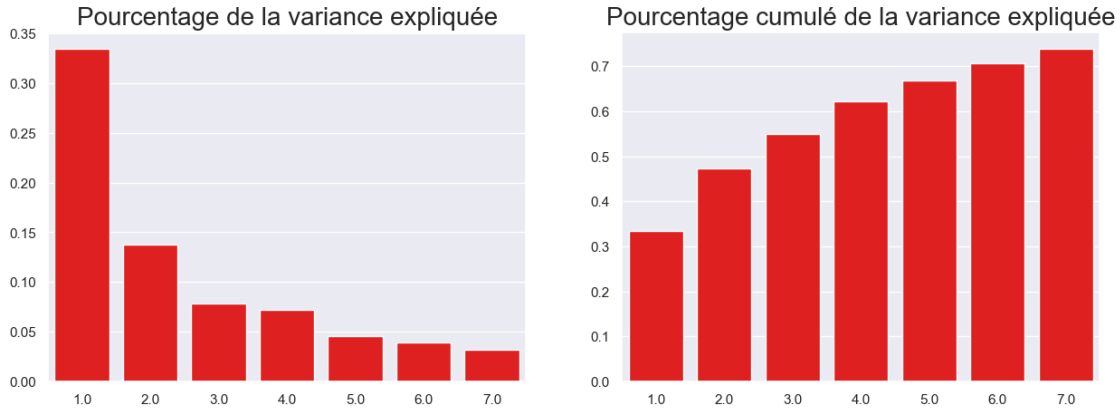
Nous décidons finalement, pour choisir le nombre de composantes principales k à considérer, de combiner la règle de Kaiser-Guttman avec la règle de l'éboulis: on commence par regarder combien de valeurs propres sont supérieures à la moyenne (règle de Kaiser-Guttman), puis on regarde si la dernière valeur propre retenue (supérieure à la moyenne) est suffisamment éloignée de celle qui la suit (inférieure à la moyenne). Si oui, on reste sur la décision de la règle de Kaiser, si non, on choisir le k correspondant au saut le plus important qui suit.

Dans notre cas, l'application de cette méthode aboutit finalement au choix de $k = 7$.

4.4.2 ACP sur les 7 premières composantes

[13]:
```
val_propres_data_centree7,vec_propres_data_centree7=hyperplans(data_centree,7)
```

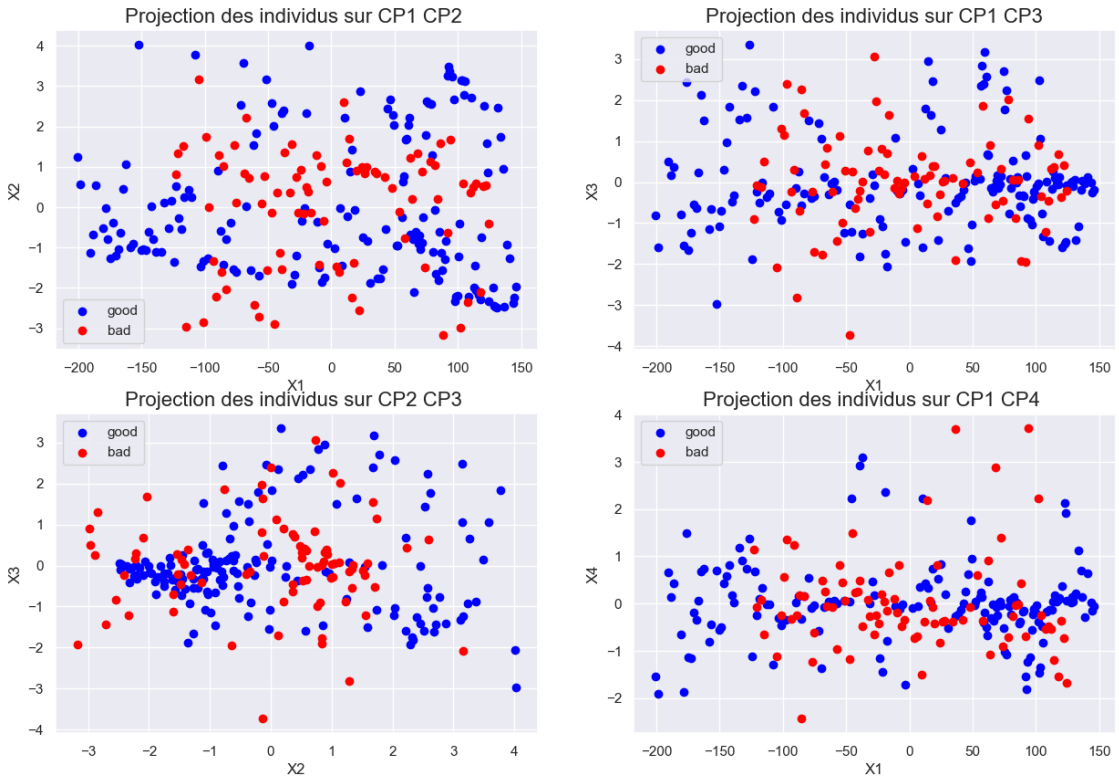
[14]:
```
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

sns.barplot(x=np.linspace(start=1, stop=7, num=7),y=val_propres_data_centree7/
 ↪(sum(val_propres_data_centree)),color ="red",
ax=ax1).set_title("Pourcentage de la variance expliquée", fontsize=20)

sns.barplot(x=np.linspace(start=1, stop=7, num=7),y=np.
 ↪cumsum(val_propres_data_centree7/(sum(val_propres_data_centree))),color =␣
 ↪'red',
ax=ax2).set_title('Pourcentage cumulé de la variance expliquée', fontsize=20)

plt.show()
```



On voit qu'avec $k = 7$, on obtient un pourcentage de variance expliquée supérieur à 70%

Visualisons le nuage des n points dans les premiers plans factoriels retenus en mettant une couleur associée à chacune des classes:

[19]:
```
train_g = train.loc[y_train == "g"]
train_b = train.loc[y_train == "b"]
train_b=train_b.drop('class', axis=1)
train_g=train_g.drop('class', axis=1)

train_g_centree=centrage(train_g)
val_propres_g_centree,vec_propres_g_centree=hyperplans(train_g_centree)
g_new_coord=(np.dot(np.array(train_g_centree), vec_propres_g_centree))

train_b_centree=centrage(train_b)
val_propres_b_centree,vec_propres_b_centree=hyperplans(train_b_centree)
b_new_coord=(np.dot(np.array(train_b_centree), vec_propres_b_centree))

plt.figure(figsize=(15,10))
plt.subplot(221)
plt.scatter(g_new_coord[:,0],g_new_coord[:,1],color='blue')
plt.scatter(b_new_coord[:,0],b_new_coord[:,1],color='red')
plt.xlabel('X1')
plt.ylabel('X2')
plt.legend(["good","bad"])
plt.title('Projection des individus sur CP1 CP2 ', fontsize=16)
plt.subplot(222)
plt.scatter(g_new_coord[:,0],g_new_coord[:,2],color='blue')
plt.scatter(b_new_coord[:,0],b_new_coord[:,2],color='red')
plt.xlabel('X1')
plt.ylabel('X3')
plt.legend(["good","bad"])
plt.title('Projection des individus sur CP1 CP3', fontsize=16)
plt.subplot(223)
plt.scatter(g_new_coord[:,1],g_new_coord[:,2],color='blue')
plt.scatter(b_new_coord[:,1],b_new_coord[:,2],color='red')
plt.xlabel('X2')
plt.ylabel('X3')
plt.legend(["good","bad"])
plt.title('Projection des individus sur CP2 CP3 ', fontsize=16)
plt.subplot(224)
plt.scatter(g_new_coord[:,0],g_new_coord[:,3],color='blue')
plt.scatter(b_new_coord[:,0],b_new_coord[:,3],color='red')
plt.xlabel('X1')
plt.ylabel('X4')
plt.legend(["good","bad"])
plt.title('Projection des individus sur CP1 CP4 ', fontsize=16)
plt.show()
```



L'ACP, si elle permet de représenter les données de façon compacte et réduite en fonction des directions de grande variance, n'est pas forcément adaptée pour visualiser distinctivement les deux types de données (rouge et bleu). Les deux classes ne seront distinctes sur les plans factoriels que si les individus associés présentent des tendances opposées sur ces même direction. L'ACP vise à représenter correctement la plupart des informations contenues dans les données sur un nombre de composante réduit, sans pour autant se focaliser sur la classification.

Au contraire, optimiser la direction où les classes sont les mieux séparée est le but de l'AFD, que l'on réalisera dans la partie suivante.

4.4.3 Qualité de projections des individus test

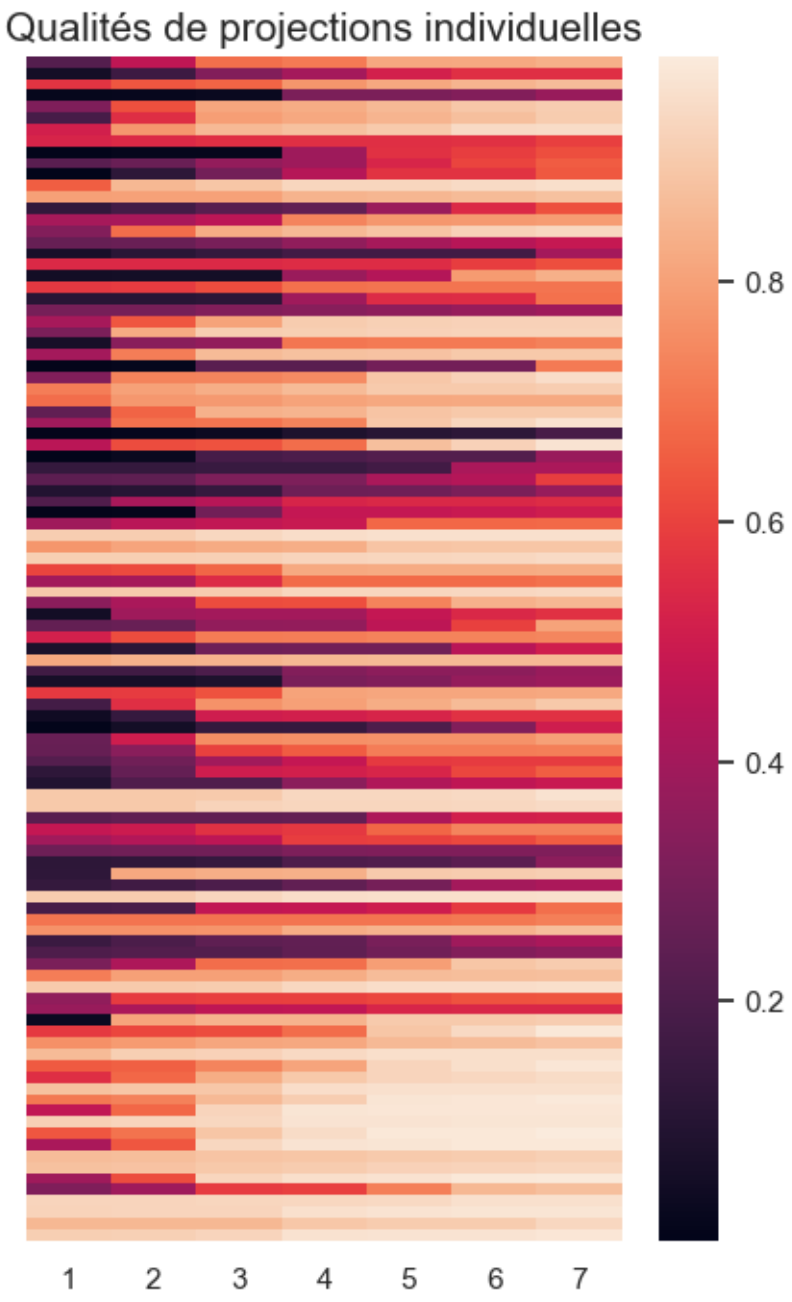
On définit la fonction qui calcule les qualités de projections individuelles $q(k,i) = \frac{\sum_{j=1}^{k}(C_i^j)^2}{\sum_{j=1}^{p}(C_i^j)^2}$, où C_i^j désigne la nouvelle coordonnée de l'individu i sur la composante j.

[20]:
```
def qpi(k,i,c):
    n,p=np.shape(c)
    s1,s2=0,0
    for j in range(k):
        s1+=c[i][j]**2
    for j in range(p):
        s2+=c[i][j]**2
    return s1/s2
```

Représentons les qualités de projections des individus tests:

[21]:
```
centree_test=centrage(X_test)
val_propres_test_centree,vec_propres_test_centree=hyperplans(centree_test)
new_coord_centré_test=(np.dot(np.array(centree_test), vec_propres_test_centree))
```

[23]:
```
n,k=105,7
Q_centré=np.transpose(np.array([[qpi(j,i,new_coord_centré_test) for i in␣
 ↪range(n)] for j in range(1,k+1)]))
fig = plt.figure(figsize=(5, 8))
ax1 = fig.add_subplot()
sns.heatmap(data=Q_centré,ax = ax1,xticklabels=range(1,8),
            yticklabels=False).set_title('Qualités de projections␣
 ↪individuelles',fontsize=15)
plt.show()
```



Les qualités de projections deviennent logiquement plus importantes lorsque k augmente puisque l'on considère d'autant plus de composantes.

Notons que les qualités de projections par axe représentent le carré du cosinus de l'angle entre l'axe de projection et le vecteur individu. Les qualités de projections sur k composantes additionnent ces cosinus. Par conséquent, ce critère reposant sur une mesure d'angle n'a pas de signification pour les individus proches du centre de gravité. Il faut bien faire attention à ces exceptions avant d'interpréter ce critère.

La plupart des individus ont une qualité de projection supérieure à 80% avec les 7 composantes, mais certains semblent mal projetés. Intéressons-nous à ceux-là.

Trouvons les 5 individus tests les moins bien projetés sur les 7 premières composantes principales:

[24]:
```
Q_proj_test=[qpi(7,i,new_coord_centré_test) for i in␣
 ↪range(len(new_coord_centré_test))]
K=5
res = sorted(range(len(Q_proj_test)), key = lambda sub: Q_proj_test[sub])[:K]
res_indice_data=[]
for i in range(len(res)):
    res_indice_data.append(test.iloc[res[i]]["Unnamed: 0"]-1)
print("Les indices en tant qu'éléments de l'ensemble test:")
print([x+1 for x in res])
print("\nLes indices en tant qu'éléments du dataset d'origine:")
print([x+1 for x in res_indice_data])
```

```
Les indices en tant qu'éléments de l'ensemble test:
[34, 71, 72, 80, 39]

Les indices en tant qu'éléments du dataset d'origine:
[113, 223, 225, 253, 129]
```

On obtient donc la liste des indices (dans le dataset d'origine) des 5 individus les moins bien projetés, dont voici les qualités de projections associées:

[27]:
```
np.transpose([Q_proj_test[x] for x in res])
```

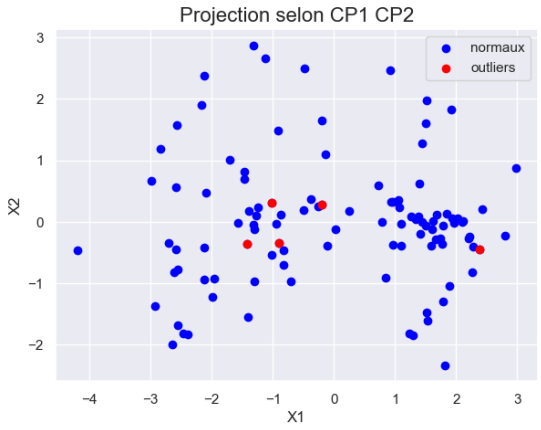
[27]:
```
array([0.18878219, 0.3203658 , 0.34500677, 0.34505739, 0.37204927])
```

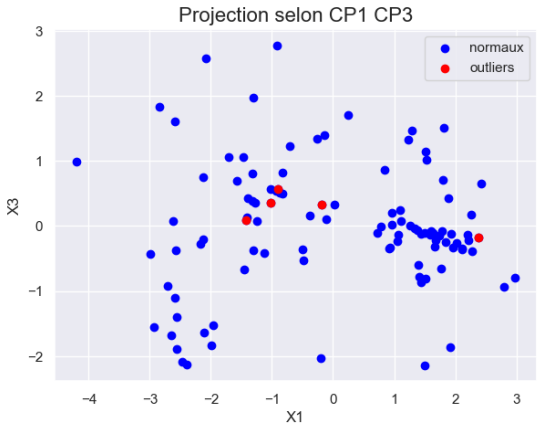
Observons le comportement de ces individus sur les premiers plans factoriels:

[28]:
```
x=[]
y=[]
z=[]
for i in res :
    x.append(new_coord_centré_test[i,0])
    y.append(new_coord_centré_test[i,1])
    z.append(new_coord_centré_test[i,2])
```

[29]:
```
plt.figure(figsize=(15,5))
plt.subplot(121)
plt.scatter(new_coord_centré_test[:,0],new_coord_centré_test[:,1],color='blue')
plt.scatter(x, y,color='red')
plt.xlabel('X1')
plt.ylabel('X2')
plt.legend(["normaux","outliers"])
plt.title('Projection selon CP1 CP2', fontsize=16)
plt.subplot(122)
plt.scatter(new_coord_centré_test[:,0],new_coord_centré_test[:,2],color='blue')
```

```
plt.scatter(x, z,color='red')
plt.xlabel('X1')
plt.ylabel('X3')
plt.legend(["normaux","outliers"])
plt.title('Projection selon CP1 CP3', fontsize=16)
plt.show()
```





Ils n'ont pas de comportement particulier sur ces premières composantes, ils ont tendance à être proche de l'origine $(0, 0)$. C'était attendu, du fait qu'ils ont la plus faible qualité de projection sur les 7ère composantes principales.

Mais alors, qu'est-ce qui caractérise ces individus ? Une faible qualité de projection sur les premières composantes principales signifie que ces individus sont mal décrits par les direction qui présentent la plus grande variance. Cela signifie qu'ils sont susceptibles de ne pas suivre le même modèle que les autres individus, car l'éloignement de leurs coordonnées à la moyenne sont les plus importants sur des variables à faible variance. Ils sont donc

Intéressons-nous aux valeurs associées à ces individus "outliers" sur les différentes variables de départ:

[30]:
```
mean=np.mean(data).to_numpy()
std=np.std(data).to_numpy()
new_data= np.transpose(X_test.iloc[res])
new_data['mean']=mean.tolist()
new_data['std']=std.tolist()
new_data
```

[30]:

	33	70	71	79	38	mean	std
a01	0.42	1.0	0.0	0.00000	0.00000	0.641342	0.496999
a02	-0.61	-1.0	0.0	0.00000	0.00000	0.044372	0.440806
a03	0.00	1.0	1.0	0.00000	-0.33672	0.601068	0.519120
a04	0.00	1.0	1.0	0.00000	0.85388	0.115889	0.460153
a05	1.00	1.0	0.0	0.00000	0.00000	0.550095	0.491951

```
a06 -1.00 -1.0  0.0  0.00000  0.00000  0.119360  0.520008
a07  0.90  1.0  1.0 -1.00000  0.68869  0.511848  0.506343
a08  1.00  1.0  1.0  1.00000 -1.00000  0.181345  0.483161
a09  0.43  1.0  0.0  0.00000  0.97078  0.476183  0.562693
a10  0.64 -1.0  0.0  0.00000  0.31385  0.155040  0.494112
a11  0.00  1.0  1.0  1.00000 -0.26048  0.400801  0.621299
a12  0.00  1.0 -1.0  0.37333 -0.59212  0.093414  0.494167
a13  0.00  1.0  0.0 -0.12000 -0.30241  0.344159  0.651897
a14  0.00 -1.0  0.0 -0.12000  0.65565  0.071132  0.457717
a15  0.67  1.0  0.0  0.00000  0.94155  0.381949  0.617139
a16 -0.29 -1.0  0.0  0.00000  0.16391 -0.003617  0.496054
a17  0.84  1.0  0.0 -1.00000  0.00000  0.359390  0.625374
a18 -1.00  1.0  0.0 -1.00000  0.00000 -0.024025  0.518336
a19  0.00  1.0  0.0  0.00000  0.00000  0.336695  0.608959
a20  0.00  1.0  0.0  0.00000  0.00000  0.008296  0.517427
a21  0.00  1.0  0.0  1.00000 -0.18043  0.362475  0.602907
a22  0.00 -1.0  0.0 -1.00000 -1.00000 -0.057406  0.526704
a23  0.21  1.0  0.0  0.00000  0.00000  0.396135  0.577626
a24  0.68 -1.0  0.0  0.00000  0.00000 -0.071187  0.507770
a25  1.00  1.0  1.0  1.00000  1.00000  0.541641  0.515469
a26  0.22 -1.0 -1.0  0.22667 -1.00000 -0.069538  0.549241
a27  0.00  1.0  0.0  0.00000  0.00000  0.378445  0.575065
a28  0.00  1.0  0.0  0.00000  0.00000 -0.027907  0.507250
a29  0.00  1.0 -1.0  0.00000  0.04447  0.352514  0.570669
a30  0.00 -1.0  1.0  0.00000  0.61881 -0.003794  0.512842
a31  0.00  1.0  0.0  0.00000  0.00000  0.349364  0.521918
a32  0.00  1.0  0.0  0.00000  0.00000  0.014480  0.467670
```

On remarque que la plupart de ces individus ont des valeurs extrêmes $(1, -1)$ ou nulles sur énormément de variables. Ils sont souvent très éloignés de la moyenne et leur écart à la moyenne est bien souvent plus élevé que l'écart-type. On retrouve là le comportement typique de valeurs aberrantes issues d'erreurs de mesure ou de saisie par exemple: ils ne suivent pas le même modèle que les reste des individus.

Les indices des outliers ici, en tant qu'élément du dataset d'origine, sont:

[31]:
```
[x+1 for x in res_indice_data]
```

[31]:
```
[113, 223, 225, 253, 129]
```

Rappelons que pour le dernier modèle de forêt d'isolement, on avait trouvé pour individus outliers les élements d'indices $18, 54, 207, 167, 78, 56, 171$. Ce ne sont donc pas les même individus. Observons désormais le comportement des outliers de la forêt d'isolement pour comprendre les différences entre ce deux types de valeurs "aberrantes":

[32]:
```
res2=[9,19,67,50,25,20,52]  #Indices des outliers en tant qu'éléments de␣
 ↪"test", ce sont les individus dont les indices sont ci-dessus en tant␣
 ↪qu'éléments du dataset d'origine
new_data2= np.transpose(X_test.iloc[res2])
```

```
new_data2['mean']=mean.tolist()
new_data2['std']=std.tolist()
new_data2
```

[32]:

	9	19	67	50	25	20	52	mean	std
a01	0.0	-0.67935	-1.00000	0.0000	1.0	0.08333	-1.0	0.641342	0.496999
a02	0.0	-1.00000	1.00000	0.0000	1.0	-0.20685	1.0	0.044372	0.440806
a03	-1.0	-1.00000	-1.00000	-1.0000	1.0	-1.00000	1.0	0.601068	0.519120
a04	-1.0	1.00000	0.15244	-1.0000	-1.0	1.00000	1.0	0.115889	0.460153
a05	1.0	1.00000	0.28354	-1.0000	1.0	-1.00000	0.0	0.550095	0.491951
a06	1.0	0.63317	1.00000	1.0000	1.0	1.00000	0.0	0.119360	0.520008
a07	-1.0	0.03515	-1.00000	0.0000	1.0	0.71875	1.0	0.511848	0.506343
a08	1.0	-1.00000	1.00000	0.0000	1.0	0.47173	-1.0	0.181345	0.483161
a09	-1.0	-1.00000	-1.00000	-1.0000	1.0	-0.82143	1.0	0.476183	0.562693
a10	1.0	-1.00000	-1.00000	1.0000	1.0	-0.62723	-1.0	0.155040	0.494112
a11	1.0	1.00000	1.00000	1.0000	1.0	-1.00000	1.0	0.400801	0.621299
a12	-1.0	1.00000	1.00000	1.0000	1.0	-1.00000	-1.0	0.093414	0.494167
a13	1.0	0.88683	-1.00000	1.0000	1.0	-1.00000	-1.0	0.344159	0.651897
a14	1.0	-1.00000	-0.23476	-1.0000	1.0	1.00000	-1.0	0.071132	0.457717
a15	-1.0	-1.00000	0.28301	0.0000	1.0	-0.02753	0.0	0.381949	0.617139
a16	-1.0	1.00000	-1.00000	0.0000	-1.0	0.59152	0.0	-0.003617	0.496054
a17	-1.0	0.83840	1.00000	0.0000	-1.0	-0.42113	-1.0	0.359390	0.625374
a18	1.0	1.00000	1.00000	0.0000	1.0	-0.42113	-1.0	-0.024025	0.518336
a19	1.0	1.00000	-0.31402	-1.0000	-1.0	-0.74628	0.0	0.336695	0.608959
a20	-1.0	-1.00000	-1.00000	-1.0000	1.0	-1.00000	0.0	0.008296	0.517427
a21	-1.0	-1.00000	-1.00000	-1.0000	-1.0	-1.00000	0.0	0.362475	0.602907
a22	1.0	-1.00000	-1.00000	1.0000	1.0	-0.46801	0.0	-0.057406	0.526704
a23	-1.0	-0.18856	1.00000	1.0000	1.0	-1.00000	-1.0	0.396135	0.577626
a24	1.0	1.00000	-1.00000	0.4375	-1.0	0.23810	-1.0	-0.071187	0.507770
a25	1.0	1.00000	-1.00000	1.0000	1.0	1.00000	1.0	0.541641	0.515469
a26	-1.0	-1.00000	-0.03578	-1.0000	1.0	-1.00000	-1.0	-0.069538	0.549241
a27	-1.0	-1.00000	1.00000	0.0000	-1.0	-1.00000	1.0	0.378445	0.575065
a28	1.0	-1.00000	-1.00000	0.0000	1.0	-0.38914	1.0	-0.027907	0.507250
a29	-1.0	-1.00000	-1.00000	-1.0000	-1.0	-1.00000	-1.0	0.352514	0.570669
a30	-1.0	1.00000	-0.32317	-1.0000	-1.0	-1.00000	-1.0	-0.003794	0.512842
a31	1.0	1.00000	0.14939	-1.0000	-1.0	-1.00000	0.0	0.349364	0.521918
a32	-1.0	0.33611	1.00000	1.0000	1.0	0.61458	0.0	0.014480	0.467670

On remarque que ces derniers, contrairements aux outliers de l'ACP, sont très rarement nuls sur les variables. Ils possèdent quasiment tous des valeurs extrêmes (-1 ou 1) sur énormément de variables. Et pour cause, on a tuné nos hyperparamètres du modèle de forêt d'isolement de telle sorte qu'ils correspondent à des outliers situés aux extrêmes des plan factoriels. On confirme cela en observant leur qualité de projection, souvent importante:

[35]:
```
Q_proj_outliers=[qpi(7,i,new_coord_centré_test) for i in res2]
np.transpose(Q_proj_outliers)
```

[35]:
```
array([0.64865331, 0.83524051, 0.52080094, 0.80579286, 0.72886295,
       0.70152667, 0.50895734])
```

Les deux modèles mettent donc en évidence deux types d'outliers. Selon nos besoins, on paramètrera la forêt d'isolement ou on utilisera l'ACP pour détecter les valeurs que l'on considère comme aberrantes. Comme nous manquons d'information qualitatives sur les données que nous manipulons, nous travaillons à l'aveugle et donc bien que nous pouvons mettre en évidence plusieurs types de données suspectes, nous ne sommes pas réellement capables de dire lesquels correspondent en réalité à des fausses valeurs.

Remarque: Un autre problème que l'on a pour utiliser l'ACP comme méthode de validation de la forêt d'isolement est que l'ACP ne prend en compte que les variables quantitatives, elle ne considère pas la classe des individus contrairement à la forêt d'isolement. Or, les individus de la classe "g" peuvent suivre un modèle différent que les individus de la classe "b", donc il est peut être plus pertinent de plutôt faire l'ACP pour chaque classe séparément pour espérer obtenir des résultats d'anomalie similaires à ceux de la forêt d'isolement. Réalisons donc l'ACP par classe:

[36]:
```
test_g=test.loc[test["class"]=="g"]
te_g=test_g.copy()
test_b=test.loc[test["class"]=="b"]
te_b=test_b.copy()
```

[37]:
```
del test_g['Unnamed: 0']
del test_g['class']
del test_b['Unnamed: 0']
del test_b['class']
```

Observons la répartition des individus test suivant leur classe:

[38]:
```
np.shape(test_g)
```

[38]:
```
(63, 32)
```

[39]:
```
np.shape(test_b)
```

[39]:
```
(42, 32)
```

Réalisons l'ACP pour chacune des classes et tirons les individus les moins bien projetés pour ceux de la classe **"g"**:

[42]:
```
centree_test_g=centrage(test_g)
val_propres_test_g_centree,vec_propres_test_g_centree=hyperplans(centree_test_g)
new_coord_centré_test_g=(np.dot(np.array(centree_test_g),␣
 ↪vec_propres_test_g_centree))
n,k=63,7
Q_centré_g=np.transpose(np.array([[qpi(j,i,new_coord_centré_test_g) for i in␣
 ↪range(n)] for j in range(1,k+1)]))
Q_proj_test_g=[qpi(7,i,new_coord_centré_test_g) for i in␣
 ↪range(len(new_coord_centré_test_g))]
```

```
K=5
res_g = sorted(range(len(Q_proj_test_g)), key = lambda sub:␣
 ↪Q_proj_test_g[sub])[:K]
res_indice_data_g=[]
for i in range(len(res_g)):
    res_indice_data_g.append(te_g.iloc[res_g[i]]["Unnamed: 0"]-1)
print("Les indices en tant qu'éléments de l'ensemble test_g:")
print([x+1 for x in res_g])
print("\nLes indices en tant qu'éléments du dataset d'origine:")
print([x+1 for x in res_indice_data_g])
```

```
Les indices en tant qu'éléments de l'ensemble test_g:
[21, 42, 19, 27, 15]

Les indices en tant qu'éléments du dataset d'origine:
[140, 269, 124, 170, 106]
```

De même pour ceux de la classe **“b”**:

[43]:
```
centree_test_b=centrage(test_b)
val_propres_test_b_centree,vec_propres_test_b_centree=hyperplans(centree_test_b)
new_coord_centré_test_b=(np.dot(np.array(centree_test_b),␣
 ↪vec_propres_test_b_centree))
n,k=42,7
Q_centré_b=np.transpose(np.array([[qpi(j,i,new_coord_centré_test_b) for i in␣
 ↪range(n)] for j in range(1,k+1)]))
Q_proj_test_b=[qpi(7,i,new_coord_centré_test_b) for i in␣
 ↪range(len(new_coord_centré_test_b))]
K=5
res_b = sorted(range(len(Q_proj_test_b)), key = lambda sub:␣
 ↪Q_proj_test_b[sub])[:K]
res_indice_data_b=[]
for i in range(len(res_b)):
    res_indice_data_b.append(te_b.iloc[res_b[i]]["Unnamed: 0"]-1)
print("Les indices en tant qu'éléments de l'ensemble test_b:")
print([x+1 for x in res_b])
print("\nLes indices en tant qu'éléments du dataset d'origine:")
print([x+1 for x in res_indice_data_b])
```

```
Les indices en tant qu'éléments de l'ensemble test_b:
[42, 38, 23, 28, 20]

Les indices en tant qu'éléments du dataset d'origine:
[253, 225, 159, 177, 129]
```

Cette ACP par classe est dès lors déjà plus apte à éventuellement décrire des outliers similaires à ceux issues de l’algorithme de forêt d’isolement, sous hypothèse que l’on choisisse bien les hyper-paramètres pour aller chercher les même types d’outliers que l’ACP. Cela dépendra de nos données et de ce que l’on considère réellement comme “valeur aberrante”.

5 AFD sur $\mathbb{R}^p$

Nous allons réaliser l'**analyse factorielle discriminante (AFD)** sur nos données des parties précédentes. Il s'agit d'une technique statistique qui vise à décrire, expliquer et prédire l'appartenance à des classes prédéfinies d'un ensemble d'observations (nos individus) à partir d'une série de variables prédictives, en se basant sur la décomposition de Huygens de l'inertie totale comme somme de l'**inertie inter-classe** et de l'**inertie intra-classe**.

L'objectif est donc de trouver des facteurs, combinaisons linéaires des variables quantitatives de départ, qui séparent au mieux les différentes classes.

5.1 Import des modules

[287]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
```

5.2 Import et prétraitement des données

Nous importons les données des parties précédentes:

[288]:
```
data = pd.read_csv('Ionosphere.csv', delimiter=';', decimal=".", ␣
 ↪error_bad_lines=False)
train = pd.read_csv('train.csv', delimiter=';',decimal=",", ␣
 ↪error_bad_lines=False)
test = pd.read_csv('test.csv', delimiter=';',decimal=",", ␣
 ↪error_bad_lines=False)
tr = train.copy()
te = test.copy()
del test["Unnamed: 0"]
del train["Unnamed: 0"]
```

Explicitons les différentes classes:

[289]:
```
groupes = list(set(data["class"]))
groupes
```

[289]:
```
['g', 'b']
```

Rassemblons les données suivant leurs classes: "b" et "g".

[290]:
```
data_groupe_g = train.loc[train["class"] == "g"]
data_groupe_b = train.loc[train["class"] == "b"]
```

[291]:
```
del data_groupe_b["class"]
del data_groupe_g["class"]
```

Une fois cela fait, calculons désormais les matrices de variances inter et intra-classe:

5.3 Calcul de la matrice de variance intra-classe W

Centrons nos données et calculons les deux matrices de variances de chaque classe W_g et W_b.

[292]:
```
n_g = len(data_groupe_g)
n_b = len(data_groupe_b)
```

[293]:
```
def centrage(data):
    data_centree = data.copy()
    for col in data.columns :
        data_centree[col] = (data_centree[col]-data.mean()[col])
    return data_centree

def matrice_cov(data):
    data_transposee = np.transpose(data)
    p, n = data_transposee.shape
    cov_data = np.dot(data_transposee,data)/n
    return cov_data
```

[294]:
```
W_g = matrice_cov(centrage(data_groupe_g))
W_b = matrice_cov(centrage(data_groupe_b))
```

Nous pouvons en déduire la matrice de variance intra-classe donnée par la formule $W = \frac{1}{n}\sum_k n_k \times W_k$.

[295]:
```
n = len(train)
W = (n_g * W_g + n_b * W_b)/n
W
```

[295]:
```
array([[ 0.18531541,  0.04748837,  0.05221866, …, -0.00270268,
         0.03304089,  0.01414866],
       [ 0.04748837,  0.18102275, -0.00411242, …, -0.0193937 ,
        -0.0234831 ,  0.00318927],
       [ 0.05221866, -0.00411242,  0.1816105 , …,  0.00655465,
         0.06452714, -0.01616549],
       …,
       [-0.00270268, -0.0193937 ,  0.00655465, …,  0.22009248,
        -0.02570962,  0.13014747],
       [ 0.03304089, -0.0234831 ,  0.06452714, …, -0.02570962,
         0.25650154, -0.045066  ],
       [ 0.01414866,  0.00318927, -0.01616549, …,  0.13014747,
        -0.045066  ,  0.20947581]])
```

5.4 Calcul de la matrice de variance inter-classe B

La matrice de variance inter-classe donne la dispertion des centroïdes des classes autour du centre de gravité global, et est défine par $B = \frac{1}{n}\sum_k n_k({}^t(\mu_k - \mu))(\mu_k - \mu)$, où μ est le centre de gravité du nuage de points global et μ_k le centre de gravité de la classe k.

Calculons μ_g et μ_b (nous travaillons avec deux classes uniquement ici):

[296]:
```
del train["class"]
mu_g = []
for col in data_groupe_g.columns :
    mu_g.append(data_groupe_g.mean()[col])
mu_b = []
for col in data_groupe_b.columns :
    mu_b.append(data_groupe_b.mean()[col])
mu = []
for col in train.columns :
    mu.append(train.mean()[col])
```

[297]:
```
mu_g = pd.DataFrame(mu_g)
mu_b = pd.DataFrame(mu_b)
mu = pd.DataFrame(mu)
```

On peut donc finalement calculer notre matrice B:

[298]:
```
B_g = np.dot(mu_g - mu, pd.DataFrame.transpose(mu_g - mu))
B_b = np.dot(mu_b - mu, pd.DataFrame.transpose(mu_b - mu))
B = (n_g * B_g + n_b * B_b)/n
B
```

[298]:
```
array([[ 0.06345239,  0.01595062,  0.06203811, …,  0.00327825,
         0.02925308, -0.00319334],
       [ 0.01595062,  0.00400966,  0.0155951 , …,  0.00082408,
         0.00735362, -0.00080274],
       [ 0.06203811,  0.0155951 ,  0.06065535, …,  0.00320518,
         0.02860107, -0.00312217],
       …,
       [ 0.00327825,  0.00082408,  0.00320518, …,  0.00016937,
         0.00151135, -0.00016498],
       [ 0.02925308,  0.00735362,  0.02860107, …,  0.00151135,
         0.01348638, -0.00147221],
       [-0.00319334, -0.00080274, -0.00312217, …, -0.00016498,
        -0.00147221,  0.00016071]])
```

5.5 Calcul de la matrice de variance totale V

On démontre de variance totale V se décompose simplement comme somme de la variance inter-classe et de la variance intra-classe: $V = W + B$.

[299]:
```
V = B + W
V
```

[299]:
```
array([[ 0.2487678 ,  0.06343899,  0.11425677, …,  0.00057557,
         0.06229397,  0.01095532],
       [ 0.06343899,  0.18503241,  0.01148268, …, -0.01856962,
        -0.01612948,  0.00238653],
```

```
       [ 0.11425677,  0.01148268,  0.24226585, …,  0.00975983,
         0.0931282 , -0.01928765],
       …,
       [ 0.00057557, -0.01856962,  0.00975983, …,  0.22026185,
        -0.02419827,  0.12998248],
       [ 0.06229397, -0.01612948,  0.0931282 , …, -0.02419827,
         0.26998792, -0.04653821],
       [ 0.01095532,  0.00238653, -0.01928765, …,  0.12998248,
        -0.04653821,  0.20963652]])
```

Calculons désormais la matrice de variance totale Σ de la manière classique, et vérifions que l'on a bien égalité comme énoncé par le théorème de Huygens:

[300]:
```
Sigma = matrice_cov(centrage(train.iloc[:,:32]))
Sigma
```

[300]:
```
array([[ 0.2487678 ,  0.06343899,  0.11425677, …,  0.00057557,
         0.06229397,  0.01095532],
       [ 0.06343899,  0.18503241,  0.01148268, …, -0.01856962,
        -0.01612948,  0.00238653],
       [ 0.11425677,  0.01148268,  0.24226585, …,  0.00975983,
         0.0931282 , -0.01928765],
       …,
       [ 0.00057557, -0.01856962,  0.00975983, …,  0.22026185,
        -0.02419827,  0.12998248],
       [ 0.06229397, -0.01612948,  0.0931282 , …, -0.02419827,
         0.26998792, -0.04653821],
       [ 0.01095532,  0.00238653, -0.01928765, …,  0.12998248,
        -0.04653821,  0.20963652]])
```

[301]:
```
np.linalg.norm(Sigma-V)
```

[301]:
```
2.0365533782034384e-15
```

On remarque que $\|\Sigma - V\|$ est quasi-nul. En réalité, cette quantité est nulle, le résidu est simplement dû à l'accumulation d'erreur du calcul approché des flottant qui n'est pas exact.

Ceci est normal en vertu du théorème d'Huyghens (qui est la généralisation mutlidimensionnelle de la formule de décomposition de la variance), qui affirme $\Sigma = V = B + W$.

5.6 Recherche d'axes et de variables discriminantes

L'objectif désormais est de trouver les axes factoriels, décrits par des variables discriminantes qui sont combinaisons linéaires des variables quantitatives de départ, qui séparent au mieux les différentes classes.

Le critère sur le premier axe principal, dirigé par un certain vecteur u, est donc à la fois minimiser la dispertion intraclasse ($u'Wu$) et de maximiser la dispertion interclasse ($u'Bu$). La résolution de ce problème avec ces deux contraintes simultanées est impossible.

On montre qu'un bon compromis est atteint en maximisant le rapport de la variance interclasse sur la variance totale, c'est-à-dire de trouver le vecteur v vérifiant : $\partial_v(\frac{v'Bv}{v'\Sigma v}) = 0$.

En posant $0 < \lambda = \frac{v'Bv}{v'\Sigma v} < 1$, on se ramène après quelques calculs à résoudre le problème aux valeurs propres $\Sigma^{-1}Bv = \lambda v$.

On sait également montrer de plus que $\Sigma^{-1}B$ possède au plus $q-1$ valeurs propres non nulles, avec q le nombre de classes. On trouve donc $q-1$ axes factoriels de projection.

Dans notre cas, nous avons 2 classes uniquement ($q = 2$), on trouve donc un unique axe factoriel sur lequel projeter nos données.

Réalisons donc la réduction:

[302]:
```
def val_vec_pro(data, k = 10**3) :
    ValeursPropres, VecteursPropres = np.linalg.eig(data)
    # ordonner les valeurs propres
    ordre = ValeursPropres.argsort()[::-1]
    ValeursPropres = ValeursPropres[ordre]
    VecteursPropres = VecteursPropres[:,ordre]
    return ValeursPropres[:k], VecteursPropres[:k]
```

On détermine les valeurs et vecteurs propres de la matrice $\Sigma^{-1}B$:

[303]:
```
val_prop, vect_prop = val_vec_pro(np.dot(np.linalg.inv(Sigma), B))
```

[304]:
```
vect_prop.astype("float")[:,0]
```

[304]:
```
array([ 0.20199987,  0.07282878,  0.14139644,  0.13953027,  0.15793396,
        0.18649204,  0.43806908,  0.10538869, -0.11233482, -0.14837356,
        0.06702495,  0.04158587, -0.07767611, -0.01821725, -0.03244738,
       -0.16571747, -0.19083055,  0.13786548, -0.0566079 , -0.35530645,
        0.11208504,  0.07538155,  0.07971692,  0.28364556, -0.29732504,
       -0.04297663,  0.15469899,  0.16522062,  0.123553  ,  0.18997549,
        0.0116793 , -0.30506388])
```

[305]:
```
val_prop.astype("float")
```

[305]:
```
array([ 6.00206990e-01,  4.10101134e-16,  2.65210490e-16,  7.87010957e-17,
        7.87010957e-17,  6.80196466e-17,  3.06598994e-17,  2.46530224e-17,
        2.46530224e-17,  1.54294844e-17,  1.54294844e-17,  1.26422776e-17,
        1.26422776e-17,  1.37225883e-18,  1.37225883e-18, -5.49480444e-19,
       -5.49480444e-19, -7.37865912e-18, -7.37865912e-18, -1.57427523e-17,
       -2.07092181e-17, -2.07092181e-17, -3.43768608e-17, -3.43768608e-17,
       -5.02471411e-17, -5.02471411e-17, -8.09566643e-17, -8.09566643e-17,
       -9.48845453e-17, -9.48845453e-17, -1.97335386e-16, -5.60157250e-16])
```

La diagonalisation de la matrice $\Sigma^{-1}B$ donne théoriquement des valeurs propres entre 0 et 1, ce qui facilite leur interprétation, valeur propre = pouvoir discriminant de l'axe. Sur R la fonction "lda" dans la librairie "MASS" donne la diagonalisation de la matrice $W^{-1}B$, les valeurs propres ne sont pas les mêmes, mais elles sont reliées par la formule: $\lambda_{\Sigma_{-1}B} = \frac{\mu_{W_{-1}B}}{\mu_{W_{-1}B}+1}$.

Pour chaque valeur propre λ_i on calcule: $\frac{\lambda_i}{\sum \lambda_k}$

[306]:
```
sum_eig = sum(val_prop.astype("float"))
l=[]
for i in val_prop.astype("float"):
   l.append(i/sum_eig)
print(l)
```

```
[1.0000000000000004, 6.832661752381911e-16, 4.4186504754735983e-16,
1.3112325764651203e-16, 1.3112325764651203e-16, 1.13326981867828e-16,
5.1082209776356626e-17, 4.107420074181144e-17, 4.107420074181144e-17,
2.5706938950259397e-17, 2.5706938950259397e-17, 2.1063196288013695e-17,
2.1063196288013695e-17, 2.2863093168836547e-18, 2.2863093168836547e-18,
-9.15484913452482e-19, -9.15484913452482e-19, -1.2293524148753847e-17,
-1.2293524148753847e-17, -2.622887201865067e-17, -3.450346038029198e-17,
-3.450346038029198e-17, -5.727500918392291e-17, -5.727500918392291e-17,
-8.371635450453105e-17, -8.371635450453105e-17, -1.3488124215402105e-16,
-1.3488124215402105e-16, -1.580863717606874e-16, -1.580863717606874e-16,
-3.2877888772245946e-16, -9.33273452791918e-16]
```

On remarque que la plus grande valeur propre est la seule significative ce qui est normale car le nombre de composantes sur lesquelles on projette = K - 1, avec K le nombre de classes dans ce cas K = 2.

Les nouvelles coordonnées sur cet axe sont ensuite obtenues par un produit scalaire entre les points et notre vecteur propre (puisque c'est une base orthonormée à 1 vecteur):

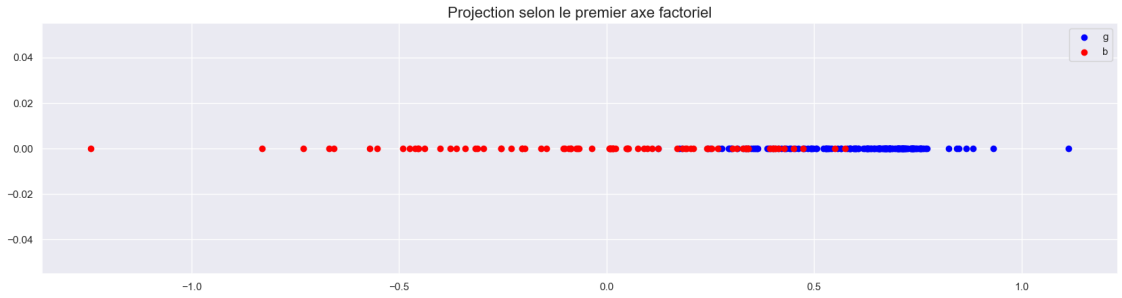
[307]:
```
new_coord = np.dot(np.array(train),vect_prop.astype(float)[:,0])
```

On différencie pour chaque classe:

[308]:
```
new_coord_train_g = np.array(data_groupe_g.dot(vect_prop.astype(float)[:,0]))
new_coord_train_b = np.array(data_groupe_b.dot(vect_prop.astype(float)[:,0]))
```

On visualise graphiquement la classification des individus par rapport à notre vecteur propre:

[309]:
```
fig = plt.figure()
fig.set_figwidth(20.5)
plt.scatter(new_coord_train_g[:], np.zeros(len(new_coord_train_g)),color='blue')
plt.scatter(new_coord_train_b[:], np.zeros(len(new_coord_train_b)),color='red')
plt.legend(["g","b"])
plt.title('Projection selon le premier axe factoriel', fontsize=16)
plt.show()
```



La projection sur cete axe de l'AFD permet de mettre en évidence la séparation entre les deux classes **g** et **b**. C'est bien là l'intérêt de l'AFD descriptive. On observe néanmoins, au milieu, que les deux classes s'entremêlent: c'est ces individus qui sont le plus "difficile" à classfier avec ce modèle. Ont-il un rôle particulier ? C'est ce que nous étudions dans la suite:

5.6.1 Qualités de projections individuelles

On projette sur un seul axe factoriel, la qualité de projection dans ce cas se définit comme suit: $Q_i = \frac{1}{n}\frac{C_i^2}{\lambda}$, avec C_i la coordonnée du point i sur notre axe factoriel, et λ notre valeur propre unique.

[310]:
```
del test["class"]
def qpi(i,c):
    return c[i]**2/(val_prop[0]*n)
```

[311]:
```
centree_test = centrage(test)
new_coord_centré_test = np.dot(np.array(centree_test),vect_prop.astype(float)[:
 ↪,0])

Q_proj_test = [qpi(i,new_coord_centré_test) for i in␣
 ↪range(len(new_coord_centré_test))]
```

Une fois que l'on a calculé nos qualités de projections individuelle, intéressons-nous aux 5 individus les moins bien projetés sur cet axe:

[312]:
```
K = 5
res = sorted(range(len(Q_proj_test)), key = lambda sub: Q_proj_test[sub])[:K]
res_indice_data = []
for i in range(len(res)):
    res_indice_data.append(te.iloc[res[i]]["Unnamed: 0"]-1)
print("Les indices en tant qu'éléments de l'ensemble test:")
print([x+1 for x in res])
print("\nLes indices en tant qu'éléments du dataset d'origine:")
print([x+1 for x in res_indice_data])
```

```
Les indices en tant qu'éléments de l'ensemble test:
[32, 77, 22, 4, 10]
```

```
Les indices en tant qu'éléments du dataset d'origine:
[110, 238, 64, 8, 18]
```

Ce ne sont pas les même individus que les moins bien projetés de l'ACP. C'était prévisible et attendu: l'ACP n'a rien avoir avec la classification, elle ne considère que les variables quantitatives et les individus mal projetés sont ceux qui ont des coordonnées importantes sur les composantes à faible variance expliquée. Les individus mal projetés en AFD sont ceux qui sont difficile à classifier avec notre modèle: ils présentent des caractéristiques qui peuvent être associées aux deux classes (ou à aucune des classes).

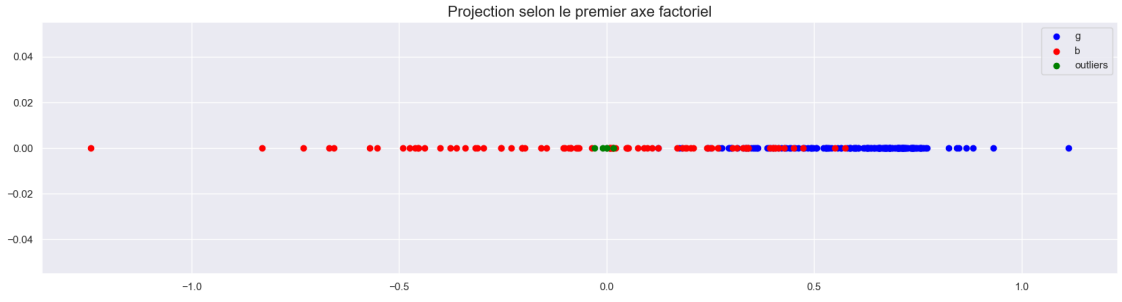
Ce ne sont pas non plus les même individus que les points avec le plus haut score d'anomalie issue de l'algorithme des forêts d'isolement. La forêt d'isolement est une boite noire, elle considère des milliers de paramètres pour décrire les individus à forte anomalies. L'AFD ne considère que leur qualité de projection sur l'axe optimal vis à vis de la classification, ce qui est très restrictif.

[313]:
```
x=[]
for i in res:
    x.append(new_coord_centré_test[i])
```

[314]:
```
data_test_g = te.loc[te["class"] == "g"]
data_test_b = te.loc[te["class"] == "b"]
del data_test_g["class"]
del data_test_b["class"]
del data_test_g["Unnamed: 0"]
del data_test_b["Unnamed: 0"]
```

[315]:
```
new_coord_test_g = np.array(data_test_g.dot(vect_prop.astype(float)[:,0]))
new_coord_test_b = np.array(data_test_g.dot(vect_prop.astype(float)[:,0]))
```

[316]:
```
fig = plt.figure()
fig.set_figwidth(20.5)
plt.scatter(new_coord_train_g[:], np.zeros(len(new_coord_train_g)),color='blue')
plt.scatter(new_coord_train_b[:], np.zeros(len(new_coord_train_b)),color='red')
plt.scatter(x,np.zeros(5),color='green')
plt.legend(["g","b","outliers"])
plt.title('Projection selon le premier axe factoriel', fontsize=16)
plt.show()
```



On voit, comme prévue avec la formule données pour la qualité de projection en AFD, que les individus les plus mal projetés sont situés au centre de l'axe. Mais on se rend compte, en étudiant la projection, que ce choix n'est peut-être pas optimal: il est clair que le "seuil" séparant les deux classes et situé plus à droite que l'origine. Ceci peut être une conséquence du fait que sur notre échantillon (**test**), les classes **g** et **b** ne sont pas uniformément réparties (il n'y a pas égalité des cardinaux entre les deux classes). Notre définition de la qualité de projection ici est à revoir.

6 Mise en oeuvre de l'AFD prédictive

6.1 Fonction discriminante et règle de Fisher

Le score de Fisher est défini comme suit: $f(x) = (x - \mu_g)^T W^{-1}(x - \mu_g) - (x - \mu_b)^T W^{-1}(x - \mu_b)$, avec x un individu et μ_g, μ_b comme définies avant.

L'interprétation: si f(x) $>$ 0, l'individu est classé dans le groupe "b", et vice versa.

[332]:
```
def score_fisher(x):
    x = np.array(x)
    f_g = np.linalg.multi_dot([x - np.array(np.transpose(mu_g)), np.linalg.
 ↪inv(W), np.transpose(x - np.array(np.transpose(mu_g)))])
    f_b = np.linalg.multi_dot([x - np.array(np.transpose(mu_b)), np.linalg.
 ↪inv(W), np.transpose(x - np.array(np.transpose(mu_b)))])
    f = f_g - f_b
    if f > 0:
        return "b"
    else:
        return "g"
```

[333]:
```
score_fisher(test[:][0:1])
```

[333]:
```
'g'
```

On va tester sur les données test pour comparer avec les résultats déjà existantes.

[334]:
```
donnees = np.array((X_test).copy())
scores = []
```

[335]:
```
test[:][0:1]
```

[335]:
```
       a01      a02      a03      a04      a05      a06  a07     a08      a09  \
0  0.99539 -0.05889  0.85243  0.02306  0.83398 -0.37708  1.0  0.0376  0.85243

       a10  …      a23      a24      a25      a26      a27     a28      a29  \
0 -0.17755  …  0.56811 -0.51171  0.41078 -0.46168  0.21266 -0.3409  0.42267

       a30      a31    a32
0 -0.54487  0.18641 -0.453

[1 rows x 32 columns]
```

[336]:
```
coord_acp
```

[336]:
```
            0         1         2         3         4         5         6  \
0    0.845346 -0.904100  0.860177  0.277647  0.576287 -0.129882 -0.240280
1   -0.702356 -0.976913  1.230992 -0.836346  1.004040  0.606400  0.123228
2    1.104155 -0.384394  0.238528  0.480572  0.301979 -0.258157 -0.205639
3   -0.497469  0.195276 -0.364778 -2.096464 -0.135063  0.485935  0.897938
4    1.508906 -1.479972  1.144598  0.311900  0.462807 -0.506573 -0.311559
..        …         …         …         …         …         …         …
100 -1.960465 -0.927820 -1.521135 -0.319831 -1.257892  1.252897  0.459297
101  1.501060 -0.063715 -0.102028  0.076002 -0.072368 -0.224772  0.045515
102  1.378390  0.081221 -0.060778  0.247812 -0.123406 -0.087705  0.037842
103  1.777579 -0.059035 -0.083886  0.379550 -0.196302 -0.051681  0.332042
104  2.024380  0.060791 -0.258063  0.372952 -0.135921 -0.030875  0.213886

            7         8         9        10
0    0.020483 -0.451026  0.043456  0.031595
1    0.001717 -1.208398  0.075685 -0.729596
2    0.090898 -0.163315 -0.061304  0.013977
3    0.879155 -0.005744  0.970383 -0.635662
4    0.110703 -0.396230 -0.179601 -0.020847
..        …         …         …         …
100  0.256810 -0.516791  0.230575 -0.094118
101  0.030102  0.170770 -0.080040 -0.055271
102  0.153704 -0.001341 -0.034301  0.034470
103 -0.001168  0.205403 -0.018128  0.115304
104  0.108458  0.218556 -0.085311  0.006295

[105 rows x 11 columns]
```

[337]:
```
n = np.shape(test)[0]
class_pred = []
for i in range(n):
    class_pred.append(score_fisher(test[:][i:i+1]))
```

[338]:
```
score = 0
for i in range(n):
    if te.iloc[i]['class'] == class_pred[i]:
        score += 1
```

[339]:
```
score/n
```

[339]:
```
0.819047619047619
```

[340]:
```
class_reel = list(te.iloc[:]['class'])
```

[341]:
```
from sklearn.metrics import confusion_matrix
cm_AFD = confusion_matrix(class_reel ,class_pred)
```

[342]:
```
cm_AFD
```

[342]:
```
array([[27, 15],
       [ 4, 59]], dtype=int64)
```

On calcule le taux de précision de l'AFD:

[343]:
```
(cm_AFD[0][0]+cm_AFD[1][1])/(sum(cm_AFD[0]) + sum(cm_AFD[1]))
```

[343]:
```
0.819047619047619
```

On a déjà trouvé précédemment la matrice de confusion des forets aléatoires:

[344]:
```
cm_IF = np.array([[42, 2],[0, 62]])
```

[345]:
```
cm_IF
```

[345]:
```
array([[42,  2],
       [ 0, 62]])
```

On calcule le taux de précision des forets aléatoires:

[346]:
```
(cm_IF[0][0]+cm_IF[1][1])/(sum(cm_IF[0]) + sum(cm_IF[1]))
```

[346]:
```
0.9811320754716981
```

Le taux de précision des forets aléatoires est très supérieur à celui de l'AFD, ceci est du à la grande sensibilité de cette dernière aux Outliers. De plus l'algorithme des forets aléatoires est très poussé par rapport à l'AFD simple.

6.2 Effet de la réduction de dimension

Dans cette question, on fera une AFD suivie d'une ACP, on importe donc nos données réduites après ACP (projetées sur 7 composantes).

[348]:
```
new_coord_centré_test
```

[348]:
```
array([[ 0.84534564, -0.90410026,  0.86017653, …, -0.09525242,
        -0.04498434,  0.06931671],
       [-0.70235563, -0.97691329,  1.23099229, …,  0.14946615,
        -0.25421144,  0.02587376],
       [ 1.10415454, -0.38439426,  0.23852835, …,  0.03936799,
         0.10954913,  0.07934381],
       …,
       [ 1.37838974,  0.0812212 , -0.06077765, …, -0.02014815,
        -0.0173995 ,  0.02942371],
       [ 1.77757883, -0.05903519, -0.08388584, …, -0.00358707,
```

```
        -0.07428663,  0.0377585 ],
       [ 2.02438024,  0.06079109, -0.25806326, …, -0.01954556,
        -0.02830192,  0.03507752]])
```

[349]:
```
coord_acp = pd.DataFrame(new_coord_centré_test[:,0:11])
```

[350]:
```
coord_acp["class"] = test["class"]
nca = coord_acp.copy()
```

[351]:
```
coord_acp
```

[351]:

	0	1	2	3	4	5	6	\
0	0.845346	-0.904100	0.860177	0.277647	0.576287	-0.129882	-0.240280	
1	-0.702356	-0.976913	1.230992	-0.836346	1.004040	0.606400	0.123228	
2	1.104155	-0.384394	0.238528	0.480572	0.301979	-0.258157	-0.205639	
3	-0.497469	0.195276	-0.364778	-2.096464	-0.135063	0.485935	0.897938	
4	1.508906	-1.479972	1.144598	0.311900	0.462807	-0.506573	-0.311559	
..	…	…	…	…	…	…	…	
100	-1.960465	-0.927820	-1.521135	-0.319831	-1.257892	1.252897	0.459297	
101	1.501060	-0.063715	-0.102028	0.076002	-0.072368	-0.224772	0.045515	
102	1.378390	0.081221	-0.060778	0.247812	-0.123406	-0.087705	0.037842	
103	1.777579	-0.059035	-0.083886	0.379550	-0.196302	-0.051681	0.332042	
104	2.024380	0.060791	-0.258063	0.372952	-0.135921	-0.030875	0.213886	

	7	8	9	10	class
0	0.020483	-0.451026	0.043456	0.031595	g
1	0.001717	-1.208398	0.075685	-0.729596	b
2	0.090898	-0.163315	-0.061304	0.013977	g
3	0.879155	-0.005744	0.970383	-0.635662	b
4	0.110703	-0.396230	-0.179601	-0.020847	g
..	…	…	…	…	…
100	0.256810	-0.516791	0.230575	-0.094118	g
101	0.030102	0.170770	-0.080040	-0.055271	g
102	0.153704	-0.001341	-0.034301	0.034470	g
103	-0.001168	0.205403	-0.018128	0.115304	g
104	0.108458	0.218556	-0.085311	0.006295	g

[105 rows x 12 columns]

Maintenant on fera l'AFD, on calcule B et Σ pour diagonaliser $\Sigma^{-1}B$.

[352]:
```
coord_acp_g = coord_acp.loc[coord_acp["class"] == "g"]
coord_acp_b = coord_acp.loc[coord_acp["class"] == "b"]
```

[353]:
```
del coord_acp_g["class"]
del coord_acp_b["class"]
```

[354]:
```
m_g = len(coord_acp_g)
m_b = len(coord_acp_b)
m = len(coord_acp)
```

[355]:
```
del coord_acp["class"]
mu_g_acp = []
for col in coord_acp_g.columns :
    mu_g_acp.append(coord_acp_g.mean()[col])
mu_b_acp = []
for col in coord_acp_b.columns :
    mu_b_acp.append(coord_acp_b.mean()[col])
mu_acp = []
for col in coord_acp.columns :
    mu_acp.append(coord_acp.mean()[col])
```

[356]:
```
mu_g_acp = pd.DataFrame(mu_g_acp)
mu_b_acp = pd.DataFrame(mu_b_acp)
mu_acp = pd.DataFrame(mu_acp)
```

[357]:
```
B_g_acp = np.dot(mu_g_acp - mu_acp, pd.DataFrame.transpose(mu_g_acp - mu_acp))
B_b_acp = np.dot(mu_b_acp - mu_acp, pd.DataFrame.transpose(mu_b_acp - mu_acp))
B_acp = (m_g * B_g_acp + m_b * B_b_acp)/m
```

[358]:
```
Sigma_acp = matrice_cov(centrage(coord_acp))
```

[359]:
```
val, vec = val_vec_pro(np.dot(np.linalg.inv(Sigma_acp), B_acp))
```

On trouve notre vecteur propre unique (un seul axe factoriel K = 1).

[360]:
```
vec.astype(float)[:,0]
```

[360]:
```
array([ 0.20059832, -0.21271544, -0.4515953 ,  0.56921379,  0.29785977,
       -0.229055  , -0.12718317,  0.43055265, -0.05126263, -0.19721913,
       -0.04804972])
```

On détermine les nouvelles coordonnées sur notre axe factoriel.

[361]:
```
new_coord_acp = np.dot(np.array(coord_acp),vec.astype(float)[:,0])
```

On fait la prédiction à l'aide de la règle de Fisher comme fait précédemment.

[362]:
```
def score_fisher_acp(x):
    x = np.array(x)
    f_g = np.linalg.multi_dot([x - np.array(np.transpose(mu_g_acp)), np.linalg.
 ↪inv(Sigma_acp - B_acp), np.transpose(x - np.array(np.transpose(mu_g_acp)))])
    f_b = np.linalg.multi_dot([x - np.array(np.transpose(mu_b_acp)), np.linalg.
 ↪inv(Sigma_acp - B_acp), np.transpose(x - np.array(np.transpose(mu_b_acp)))])
    f = f_g - f_b
    if f > 0:
```

```
        return "b"
    else:
        return "g"
```

[366]:
```
class_pred_acp = []
for i in range(m):
    class_pred_acp.append(score_fisher_acp(coord_acp[:][i:i+1]))
```

[367]:
```
score_acp = 0
for i in range(m):
    if nca.iloc[i]['class'] == class_pred_acp[i]:
        score_acp += 1
```

Le taux de précision pour une ACP (7 composantes) suivie d'une AFD est:

[368]:
```
score_acp/m
```

[368]: 0.8952380952380953

On remarque que la réduction par ACP a un effet positif sur la discrimination par AFD. Ceci est compréhensible du fait que se limiter aux composantes principales permet d'atténuer l'effet des individus les moins bien projetés (les "outliers" de l'ACP) qui étaient succeptible d'également poser problème pour la classification (bien que l'on a vu que les 5 individus les moins bien projetés n'étaient pas forcément exactement les même dans les deux analyses factorielles). L'AFD est également connu pour avoir des problèmes de type calculatoire en haute dimension, du fait de l'instabilité et de la singularité de la matrice de variance-covariance. Réduire la dimension aux composantes principales semble donc être un point important avant de réaliser l'AFD.

On pourrait maintenant se poser la question si cette réduction de la dimension impacte de la même manière les forêts aléatoires, reprenons notre modèle de forêt et appliquons le aux composantes principales uniquement:

On repasse sur R et on extrait la projection des individus tests de Python sur les 7ères composantes de l'ACP en important le fichier des nouvelles coordonnées:

[33]:
```
data <- read.csv(file='new_coord_acp.csv', header=TRUE, sep=",")[,c(1:7,33)]
```

[34]:
```
set.seed(1234)
index <- sample(1:nrow(data),round(0.70*nrow(data)))
train <- data[index,]
test <- data[-index,]
```

On réalise ensuite le même modèle que précédemment, avec nos hyperparamètres optimaux:

[56]:
```
rf2 <- randomForest(as.factor(class) ~ .
 ↪,ntree=500,mtry=5,nodesize=14,data=train)
p2=predict(rf2,test)
accuracy(test$class, p2)
```

0.980952380952381

Le score de précision des prédiction est le même en se limitant aux 7ère composantes. Ainsi, bien que l'ACP ne permet pas d'améliorer la prédiction avec notre forêt, il est inutile de garder toutes les variables et il est très pratique d'effectuer la réduction par ACP avant de construire les forêts aléatoires également. Le fait que la prédiction est améliorée pour l'AFD mais pas pour les forêts est compréhensible: les forêts aléatoires réalisent une importante série de tirages aléatoires et indépendants entre les différentes variables lors de la construction des arbres, et dès lors l'algorithme sélectionne en lui-même les composantes utiles à la classification lors de la phase d'apprentissage. En effectuant la réduction en amont, on allège les données et on améliore le temps de calcul.

Conclusion

Dans ce rapport, nous avons étudiés des méthodes de classifications (**forêts aléatoires** et **AFD prédictive**) ainsi que des méthodes factorielles de descriptions des données (suivant les composantes à hautes variances avec l'**ACP**, et suivant l'axe optimal de discrimination des classes avec l'**AFD descriptive**). Nous avons également vu comment ces méthodes factorielles peuvent donner des renseignements sur des valeurs potentiellement aberrantes, en nous intéressant aux individus les moins bien projetées, et nous avons également étudié un algorithme de détection d'anomalie plus complet et moins spécifique que sont les **forêts d'isolements**.

Nous avons ainsi compris l'intérêt de comprendre le fonctionnement des différentes méthodes afin d'interpréter leurs résultats (que ce soit en terme de projection ou d'individus isolés des autres) qui peuvent différer, et que l'on doit choisir la méthode adaptée à nos données et à ce que l'on souhaite en faire pour obtenir des résultats cohérents et spécifiques. Nous avons également vu l'importance de sélectionner les bons hyperparamètres dans les algorithmes de type "black box" que sont les forêts aléatoires et forêts d'isolement afin d'aboutir à des résultats adaptées à notre utilisation.

Enfin, nous avons à nouveau souligné l'importance et l'utilité de la méthode phare de l'analyse des données, qui est l'ACP, qui peut alléger les données et même dans certains cas améliorer les résultats des autres méthodes descriptives et prédictives en se concentrant uniquement sur les composantes principales.