Université de Technologie de Compiègne
SY09 : Analyse des données et Apprentissage automatique

SY09 : TP 05 : Classification automatique

Julien Jerphanion

Printemps 2018

Table des matières

1 Visualisation des données

Dans cette partie, on va principalement s'intéresser au positionnement multidimensionnel des jeux de données que l'on a pu étudier les précédentes séances.

Rappelons rapidement l'intérêt du positionnement multidimensionnel : dans certains cas, on peut se trouver en face non pas d'un *tableau individus variables* mais plutôt en face d'un *tableau de dissimilarité* entre individus. Dans l'analyse de ces données, on préferait se ramener à un positionnement des individus les uns par rapport aux autres pour étudier les variations ou comprendre leurs relations. Le *positionnement multidimensionel* vise à offrir une représentation d'individus dans un espace euclidien munie d'une distance d à partir d'une mesure de dissimilarité δ. L'analyse factorielle en tableau de distance est la méthode la plus utilisé pour le positionnement multidimensionel.

Commençons avec le jeu de données `iris`

1.1 Analyse en composantes principales sur les données Iris

On charge le jeu de données comme à notre habitude :

```
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

On effectue une analyse en composante principale, pour cela on centre et réduit les données en colonne sur les données quantitatives pour obtenir une représentation euclidienne centrée `X`:

```
X = scale(iris[,-c(5)], scale = TRUE, center = TRUE)
covMat = cov(iris[,-c(5)])
ACP = eigen(covMat)
vp = ACP$values
U = ACP$vectors
XACP = X %*% U
```

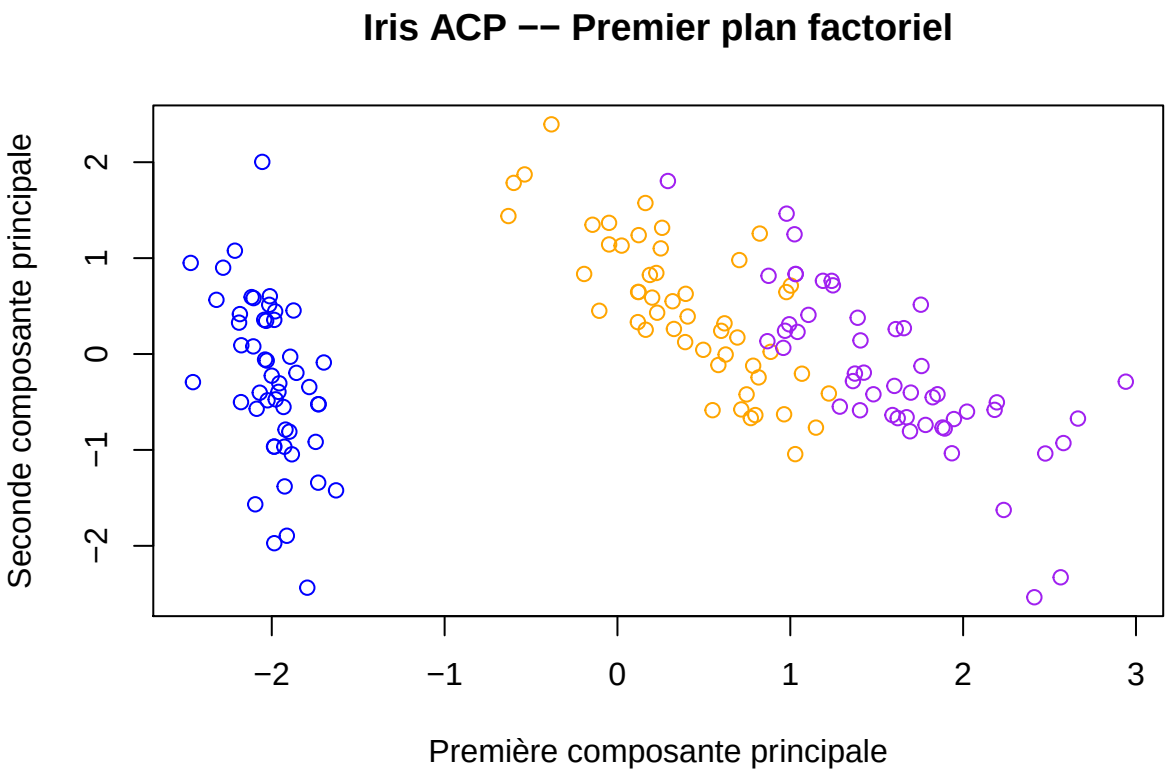
On peut se remémorer le nombre de classe du jeu de données :

```
length(unique(iris$Species))
```

```
## [1] 3
```

On peut projeter les données sur le premier plan factoriel :

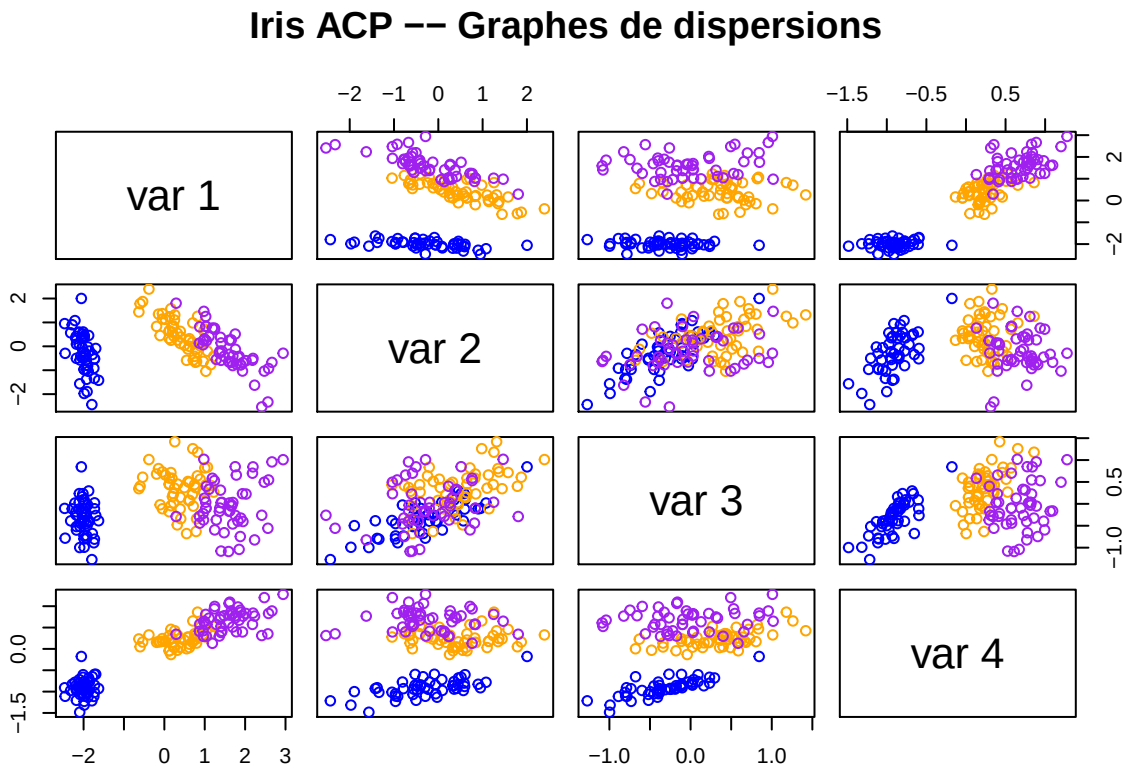
```
plot(XACP[,1],
     XACP[,2],
     xlab = "Première composante principale",
     ylab = "Seconde composante principale",
     main = "Iris ACP -- Premier plan factoriel",
     col = c("blue","orange","purple")[iris$Species]
)
```



Visuellement deux groupes de points semblent se distinguer si on ne regarde pas les couleurs. Il y en a en réalité 3 qui ont été projetés et cela est probant en rajoutant les couleurs.

On peut voir l'intégralité des graphes de dispersions ainsi:

```
pairs(XACP,
     main = "Iris ACP -- Graphes de dispersions",
     col = c("blue","orange","purple")[iris$Species]
)
```



Selon les autres plan factoriels, on peut aussi voir qu'une espèce se distingue des deux autres et est plus facilement identifiable.

Si on recherche une partition de données on peut s'attendre à ce qu'il y ait 3 composantes si cette partition est correctement réalisée.

Intéressons nous maintenant au positionnement multidimensionnel d'un autre jeu de données.

1.2 Analyse factorielle en tableau de distances sur les données Mutations

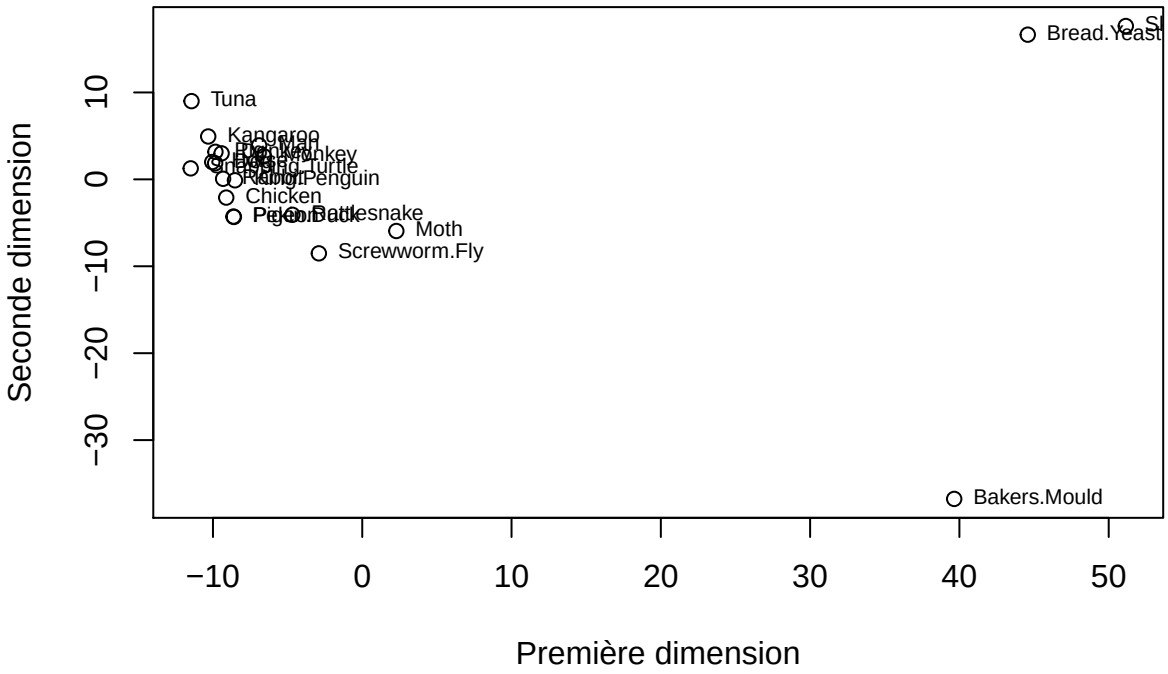
On charge les données et on construit ici un tableau de distance `mut` entre individus.

```
mutOriginal = read.csv("donnees/mutations2.csv",
                       header = TRUE,
                       row.names = 1)

mut = as.dist(mutOriginal, diag = TRUE, upper = TRUE)
```

Et on réalise un positonnement mutlidimensionnel dans un espace bidimensionnel. En `R`, on peut utiliser la `cmdscale` qui s'occupe de cela. On peut regarder comme les individus se positionne dans le premier plan ; ici la dernière instruction utilisée avec `text` permet de rajouter les étiquettes de chaque individus des données.

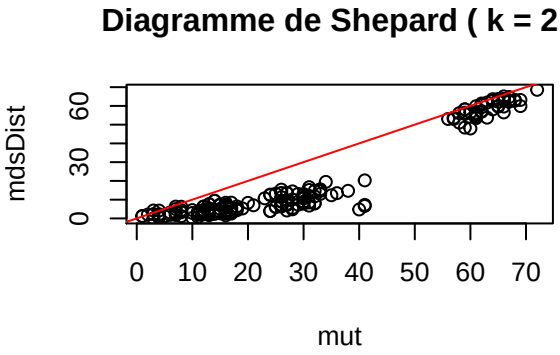
```
mds = cmdscale(mut,
               eig = TRUE, # pour retourner valeurs et vecteurs propres
               k = 2 # nombre de dimensions de l'espace
               )
positionnement = mds$points
```

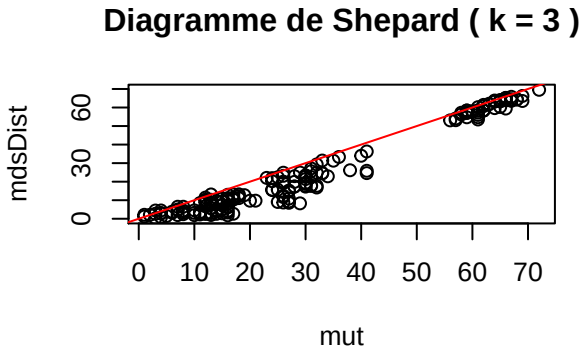
```
plot(positionnement,
     xlab = "Première dimension",
     ylab = "Seconde dimension")

text(positionnement,
     labels = names(mutOriginal),
     cex = 0.7,
     pos = 4)
```

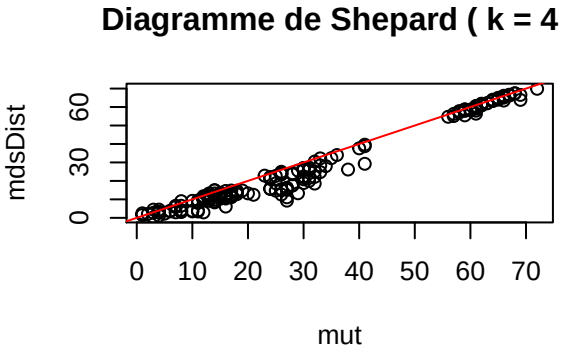


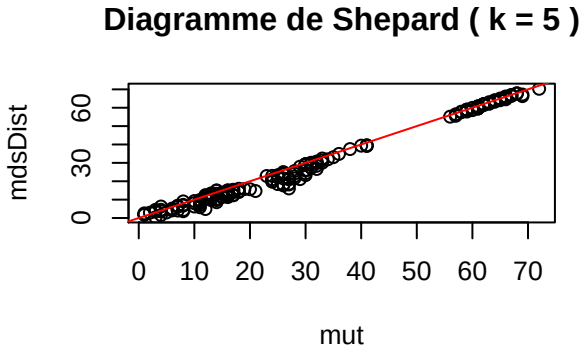
Pour savoir si le positionnement multidimensionel est pertinent, on peut utiliser un *diagramme de Shepard*. Sur ce diagramme sont représentées les distances $d_{ij} = d(x_i, x_j)$ entre les représentations des individus x_i et x_j déterminées par l'AFTD en fonction de la dissimilarité initiale δij entre ceux-ci dans le jeu de données initiales. Une représentation est bonne si ces nouvelles distances d_{ij} sont proches des dissimilarités δ_{ij} ; graphiquement, cela se traduit par des points proche de la première birectrice (tracée ici en rouge).

```
def.par <- par(no.readonly=T)
par(mfrow=c(2,2))
for (l in 2:5) {
  mds = cmdscale(mut, eig = TRUE, k = l)
  mdsDist = dist(mds$points, method = "euclidian")
  plot(x = mut,
       y = mdsDist,
       main = paste("Diagramme de Shepard ( k =", l,")"))
  abline(0,1,col = "red")
}
```









On peut voir que l'écart entre distances et dissimilarités se réduisent au fur et à mesure que la dimension de l'espace augmente. Un espace de plus grande dimensions offre plus de souplesse à la représentation euclidienne.

2 Classification hiérarchique

Il existe plusieurs types de classifications hiérarchique, en particulier la *classification descendante* et la *classification ascendante*. Ici nous nous focaliserons sur des classifications hiérarchiques ascendantes sur plusieurs jeux de données mais effecturons néanmoins une classification descendante.

`hclust` permet de réaliser une telle classification. Comme vu en cours, il y a *plusieurs critères* d'aggrégation que l'on peut utiliser pour procéder à la classification. Présentons ceux proposés par `hclust`: - `ward.D` : on l'on cherche à trouver des clusters compact et sphériques (en aggrègeant selon le *critère de Ward*). Pour des raisons techniques, on ne l'utilise plus et on préfère : - `ward.D2` : qui identique à `ward` mais qui procède à une mise au carrée des dissimilarité avant de procéder aux calculs ; - `single` : on l'on aggrège selon la distance minimale entre clusters ; - `complete` : on l'on aggrège selon la distance maximale entre clusters ; - `average` : on l'on aggrège selon la distance moyenne entre clusters;

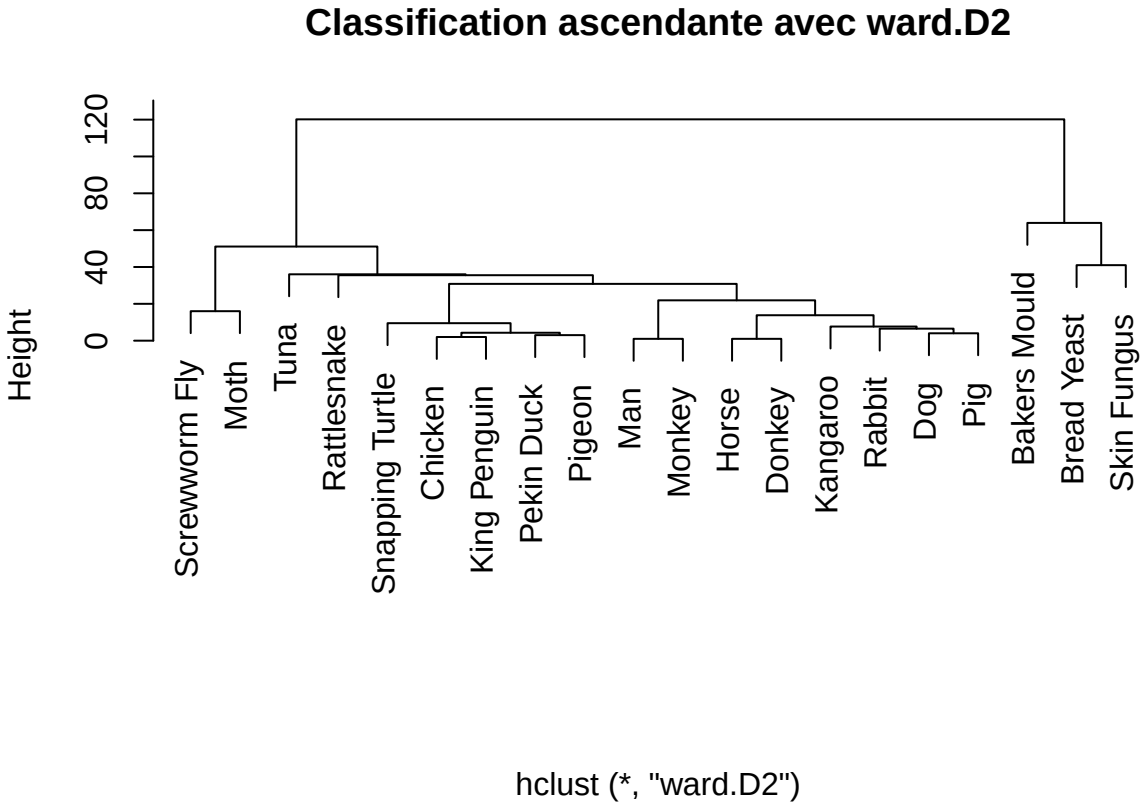
`mcquitty`, `median` et `centroid` sont trois autres méthodes que l'on peut utiliser.

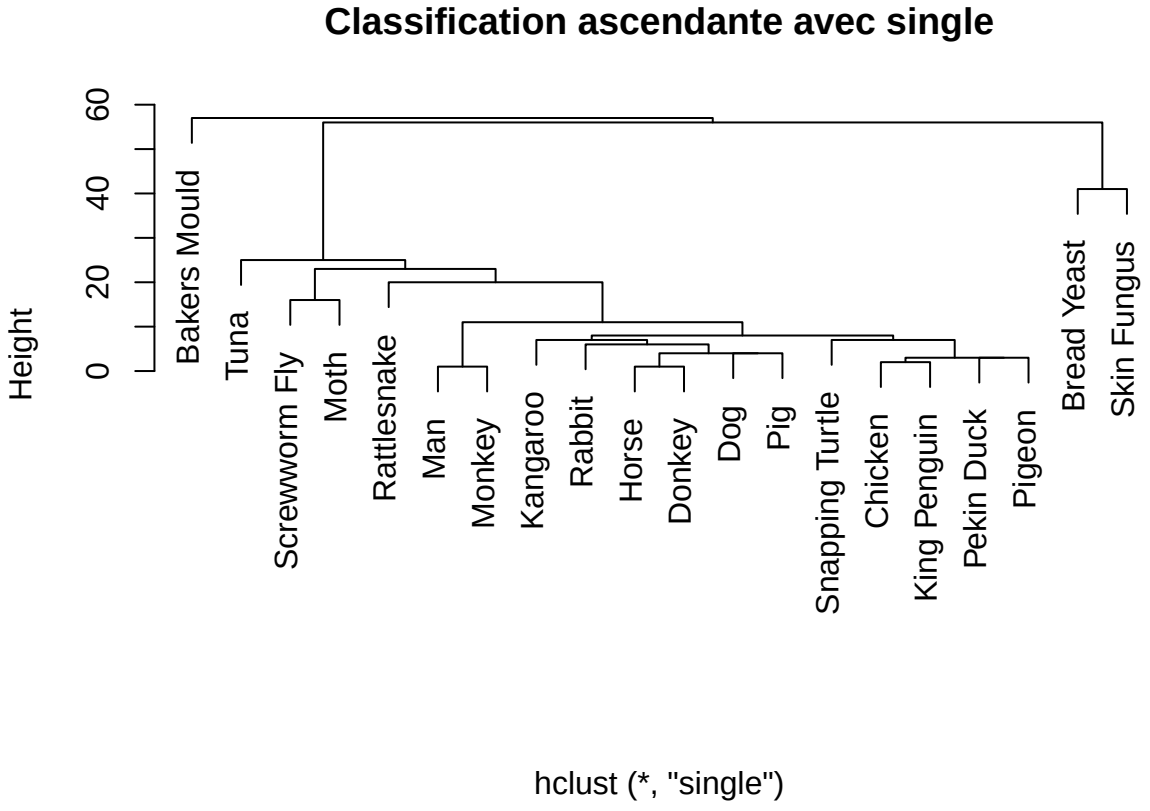
Appliquons cela à différents jeux de données.

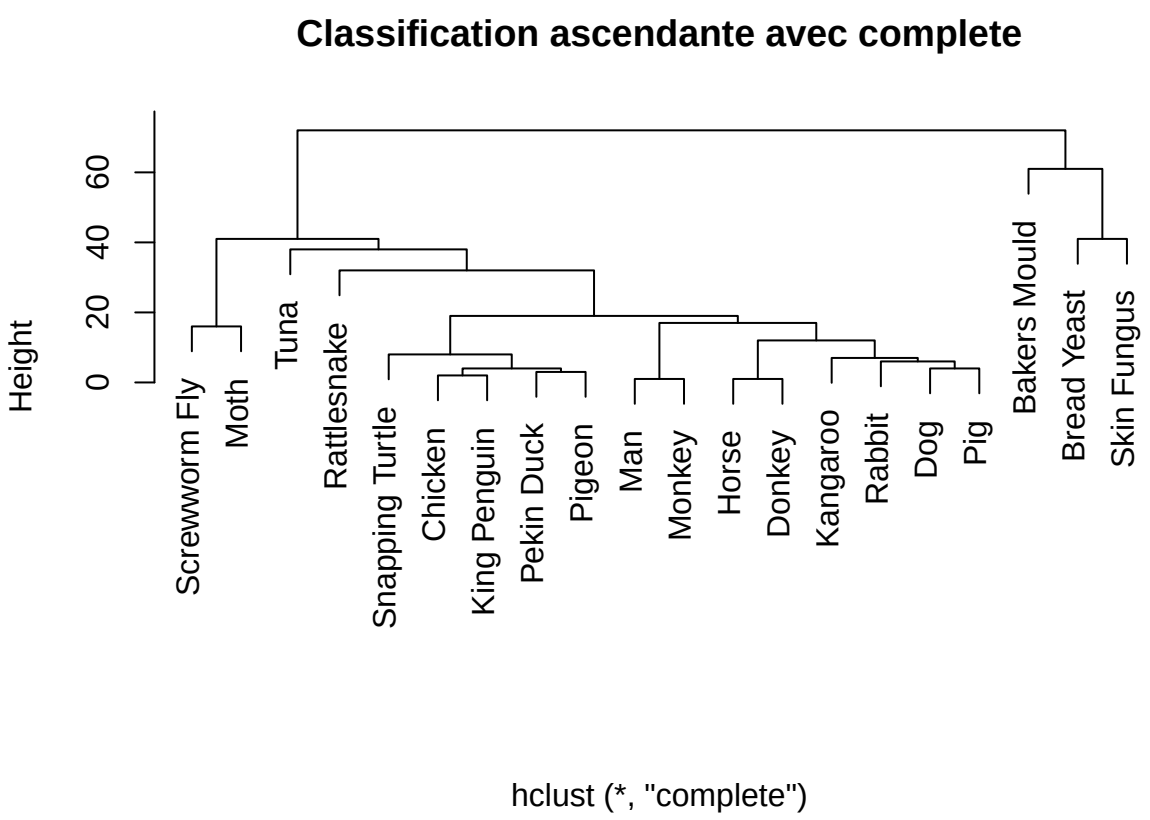
2.1 Classification ascendante sur les données Mutation

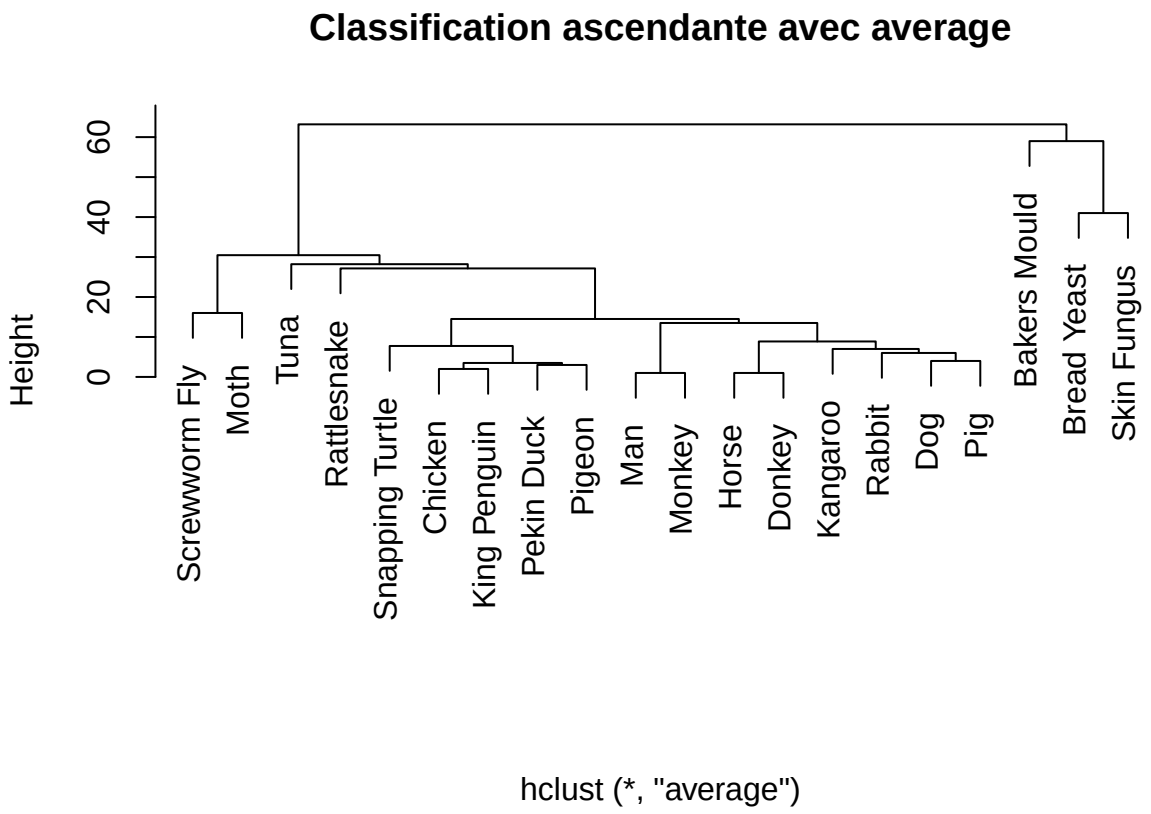
Représentons les dendogrammes associées à chaque des classifications utilisés précédemment:

```
methods = c("ward.D2", "single", "complete", "average")
for(method in methods){
  hierar = hclust(d = mut, method = method)
  plot(hierar,
       main = paste("Classification ascendante avec", method),
       xlab = "")
}
```


Classification ascendante avec ward.D2
Height
0
40
80
120
Screwworm Fly
Moth
Tuna
Rattlesnake
Snapping Turtle
Chicken
King Penguin
Pekin Duck
Pigeon
Man
Monkey
Horse
Donkey
Kangaroo
Rabbit
Dog
Pig
Bakers Mould
Bread Yeast
Skin Fungus
hclust (*, "ward.D2")


Classification ascendante avec single
Height
0
20
40
60
Bakers Mould
Tuna
Screwworm Fly
Moth
Rattlesnake
Man
Monkey
Kangaroo
Rabbit
Horse
Donkey
Dog
Pig
Snapping Turtle
Chicken
King Penguin
Pekin Duck
Pigeon
Bread Yeast
Skin Fungus
hclust (*, "single")


Classification ascendante avec complete
Height
0
20
40
60
Screwworm Fly
Moth
Tuna
Rattlesnake
Snapping Turtle
Chicken
King Penguin
Pekin Duck
Pigeon
Man
Monkey
Horse
Donkey
Kangaroo
Rabbit
Dog
Pig
Bakers Mould
Bread Yeast
Skin Fungus
hclust (*, "complete")



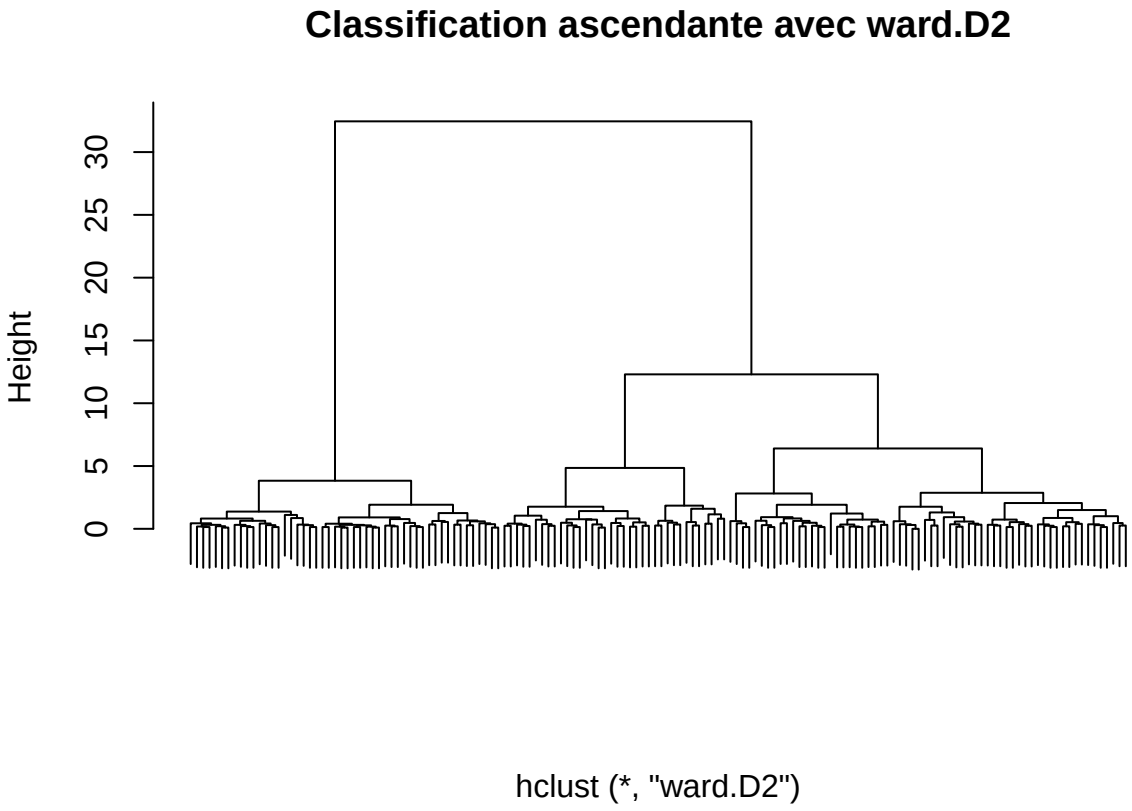
On peut voir qu'il y a une inversion d'indice (voir l'individu *Bakers Mould* à l'extrême gauche) dans le cas d'une classification ascendante avec `single`. Cela est un problème souvent rencontré lors de la classification. Les autres classifications donnent des résultats probants, en particulier `ward.D2`, `average` et `complete` ont des résultats similaires et prochent de ceux obtenues avec AFTD.

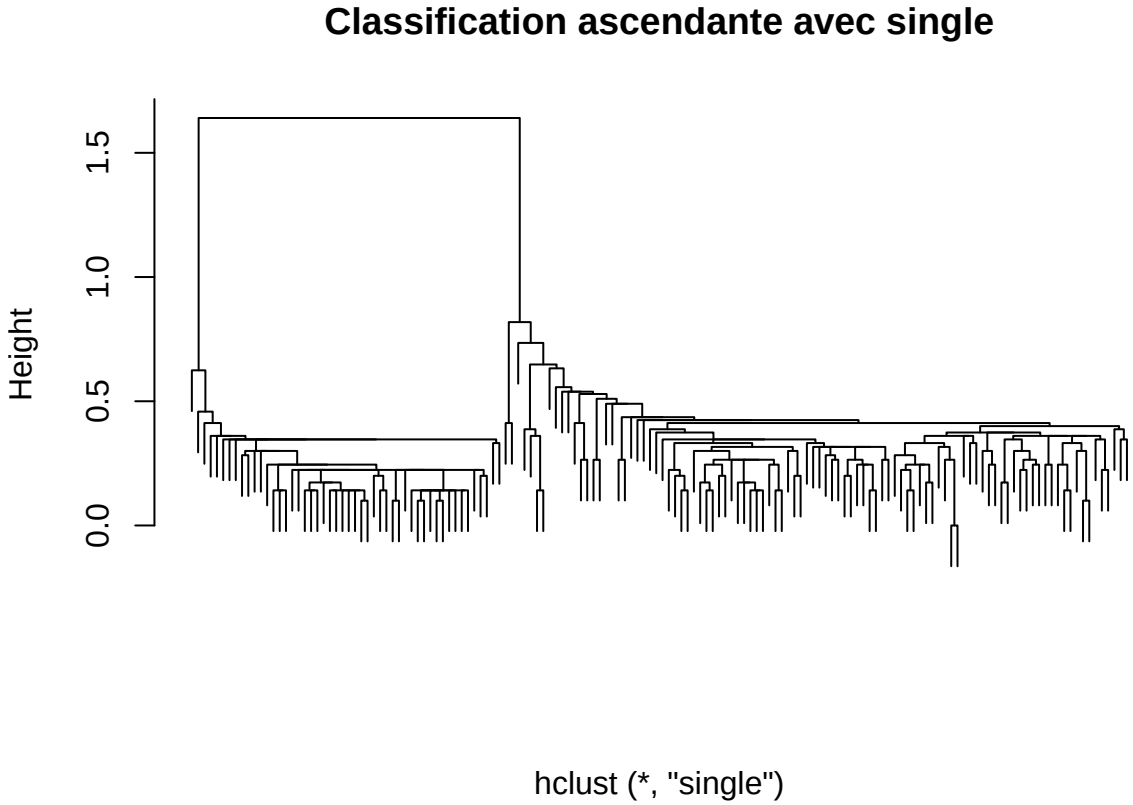
2.2 Classification ascendante sur les données Iris

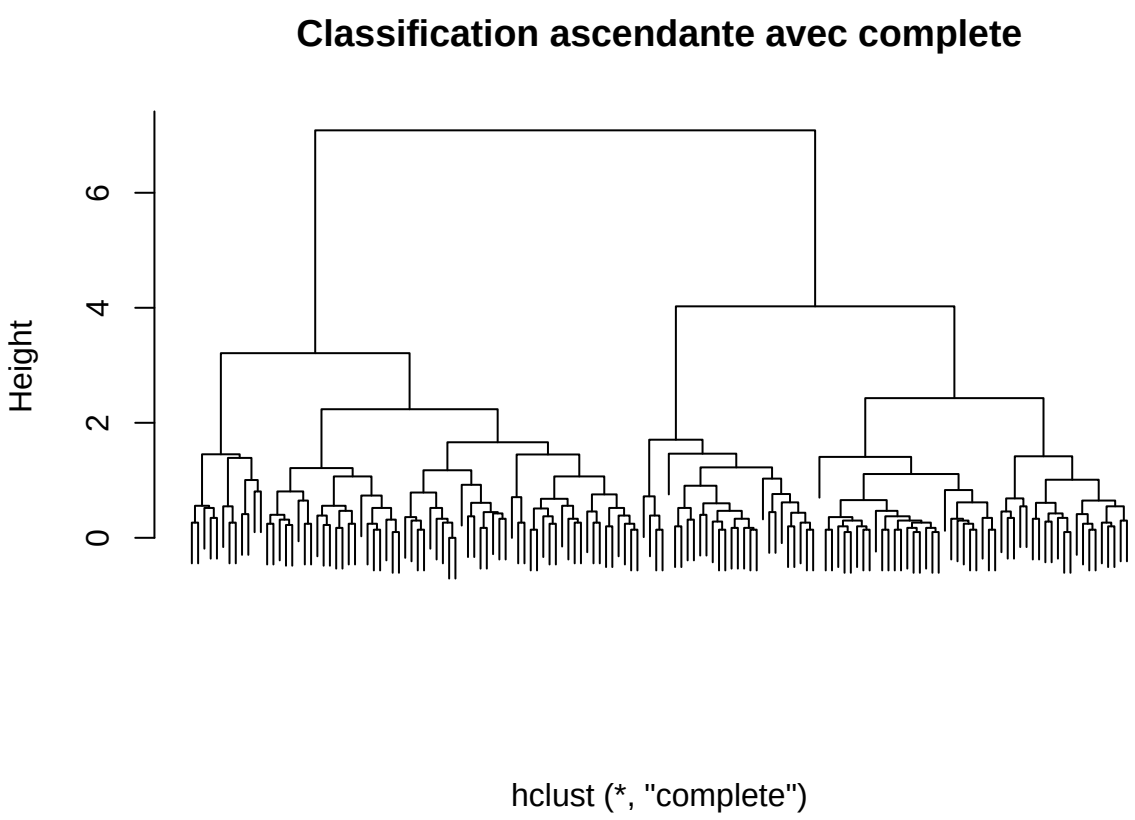
Construisons tout d'abord un tableau de distances du jeux de données.

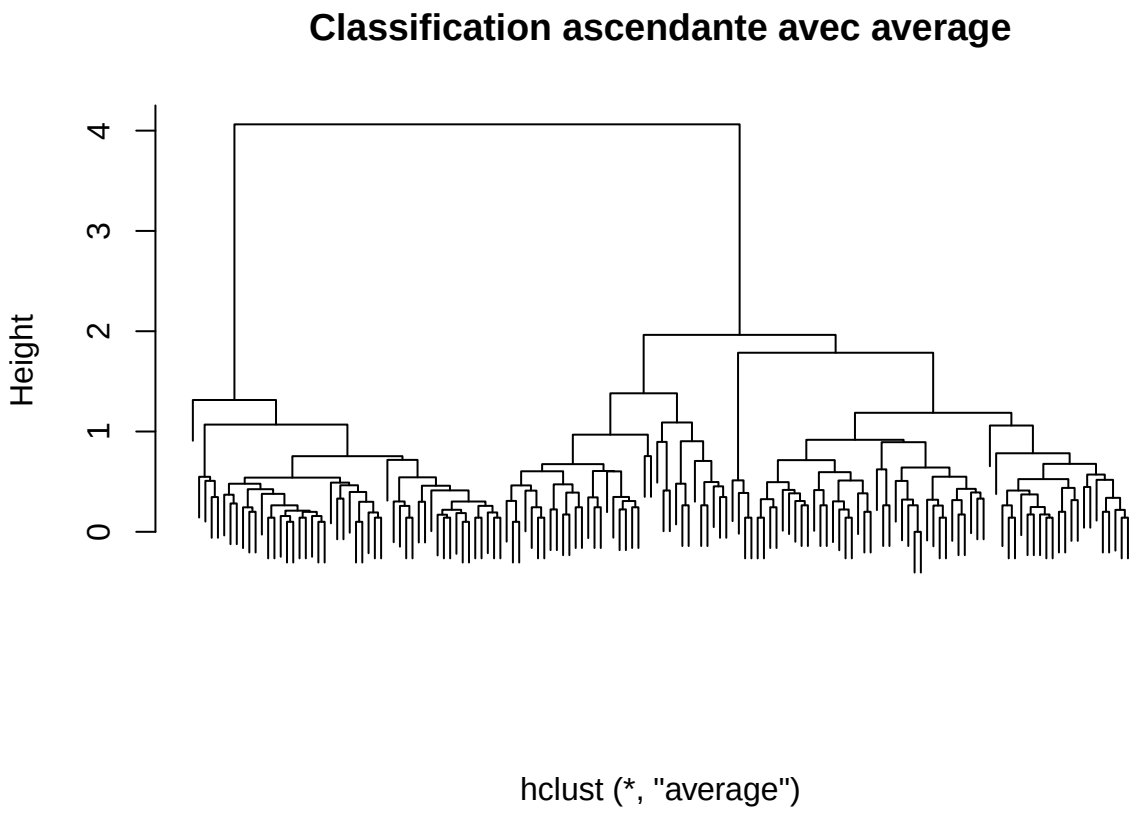
```
distIris = dist(x = iris[,-c(5)])
```

```
for(method in methods){
  hierar = hclust(d = distIris, method = method)
  plot(hierar,
       main = paste("Classification ascendante avec", method),
       xlab = "",
       labels = FALSE)
}
```


Classification ascendante avec ward.D2
Height
0
5
10
15
20
25
30
hclust (*, "ward.D2")


Classification ascendante avec single
Height
0.0
0.5
1.0
1.5
hclust (*, "single")


Classification ascendante avec complete
Height
0
2
4
6
hclust (*, "complete")

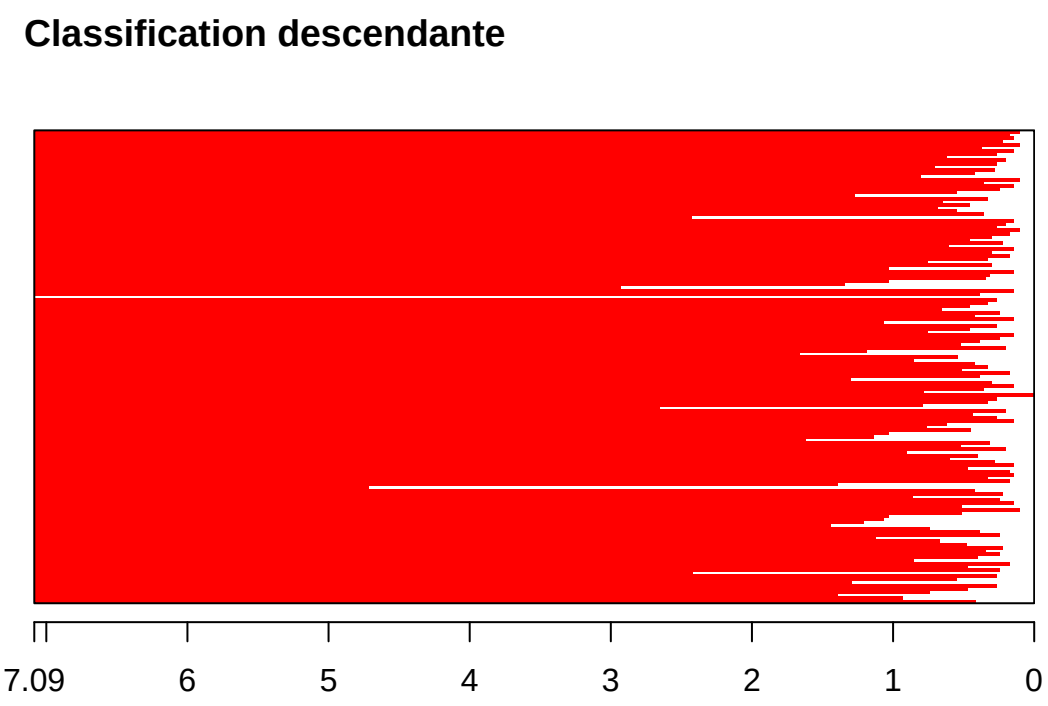


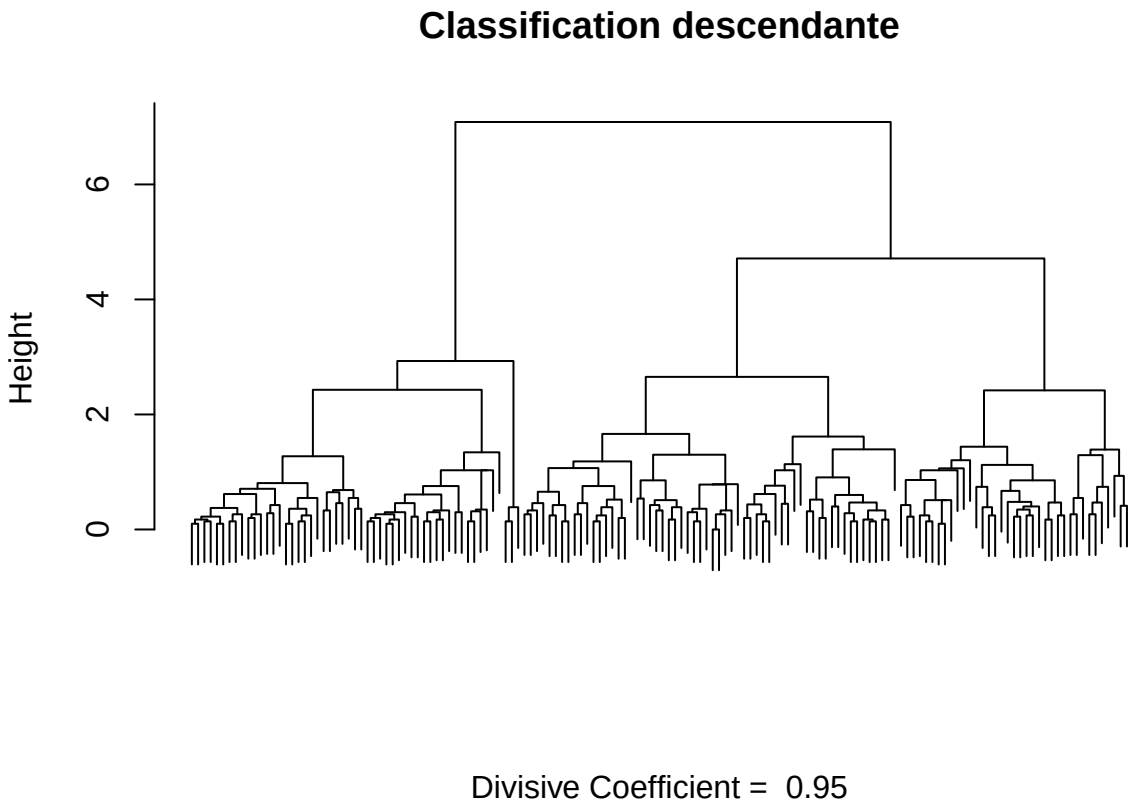
La classification selon le critère `single` donne un résultat assez dégénéré. Les autres classifications donnent une hiérarchie avec une distinction entre les trois classes qui peut être trouvée dans chaque cas avec les hauteurs suivantes: - pour `ward.D2` : 10 - pour `complete` : 3.75 - pour `average` : 2.8 (de manière plus difficile)

2.3 Classification descendante sur les données Iris

On peut effectuer une classification ascendante sur un jeu de données avec `diana` (pour *DIvisive ANAlysis Clustering*) de la bibliothèque `cluster`.

```
library(cluster)
hierarIrisDiana = diana(distIris)
plot(hierarIrisDiana,
      main = paste("Classification descendante"),
      xlab = "",
      labels = FALSE)
```


Classification descendante
7.09
6
5
4
3
2
1
0
Divisive Coefficient = 0.95


Classification descendante
Height
0
2
4
6
Divisive Coefficient = 0.95

3 Méthode des centres mobiles

On va tester l'algorithme canonique de la classification automatique, c'est à dire l'algorithme des centres mobiles.

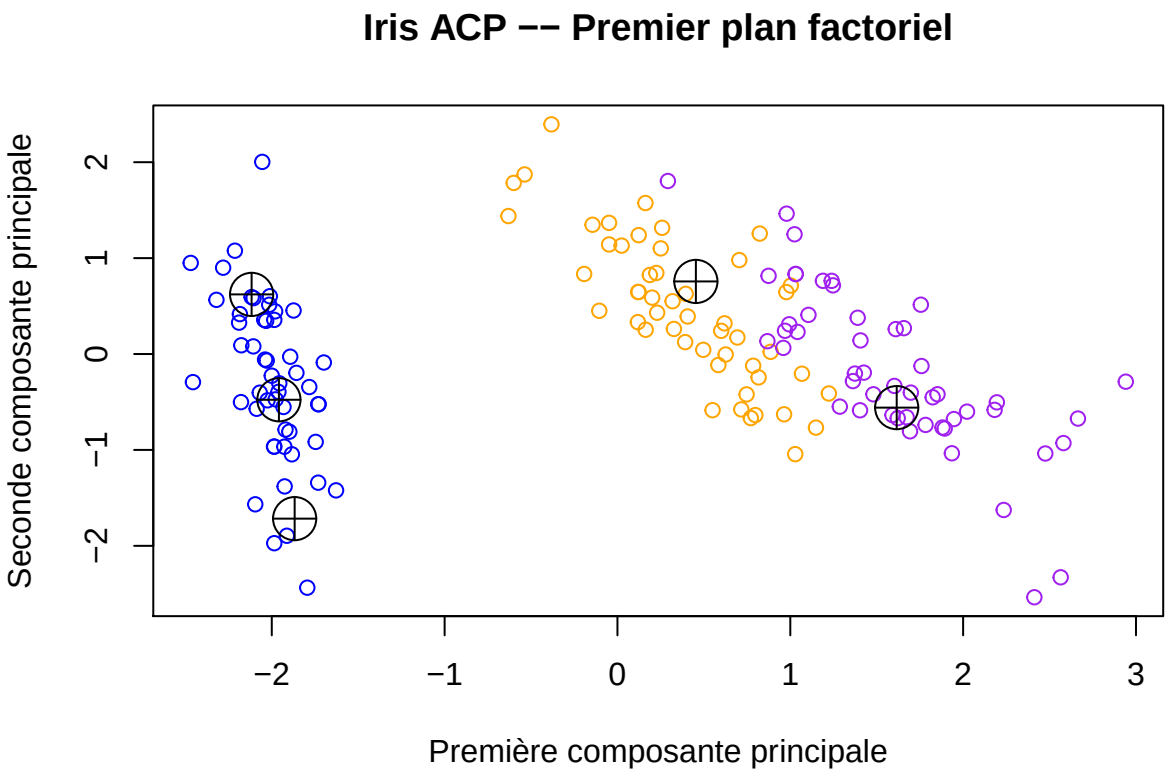
3.1 Clustering sur les données Iris

On peut appliquer tout d'abord l'algorithme

```
X = scale(iris[,-5], center = TRUE, scale = TRUE)
```

Si on répète cela plusieurs fosi pour $k = 5$, on voit que le résultat est changeant.

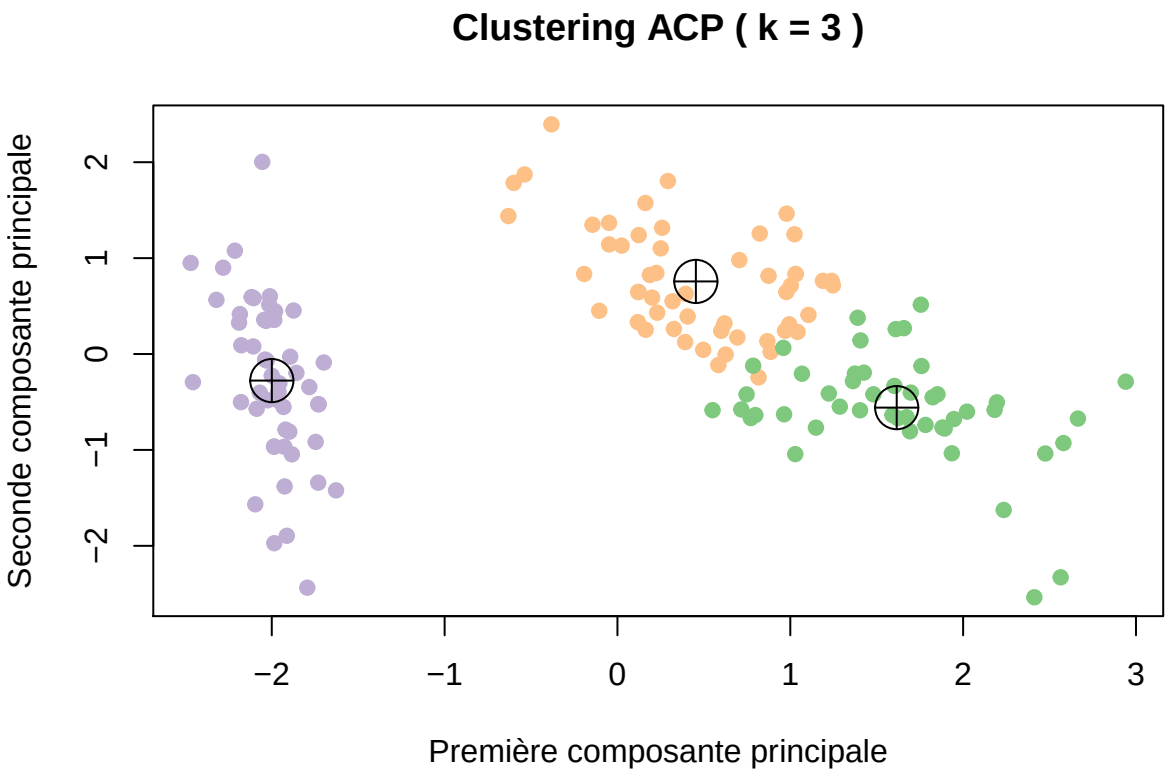
```
clustering = kmeans(x = X, 5)
centers = clustering$centers

# Projection dans la base de l'ACP
centersACP = centers %*% U

plot(XACP[,1],
     XACP[,2],
     xlab = "Première composante principale",
     ylab = "Seconde composante principale",
     main = "Iris ACP -- Premier plan factoriel",
     col = c("blue","orange","purple","black")[iris$Species]
)

points(centersACP, col="black", pch = 10, cex = 3)
```

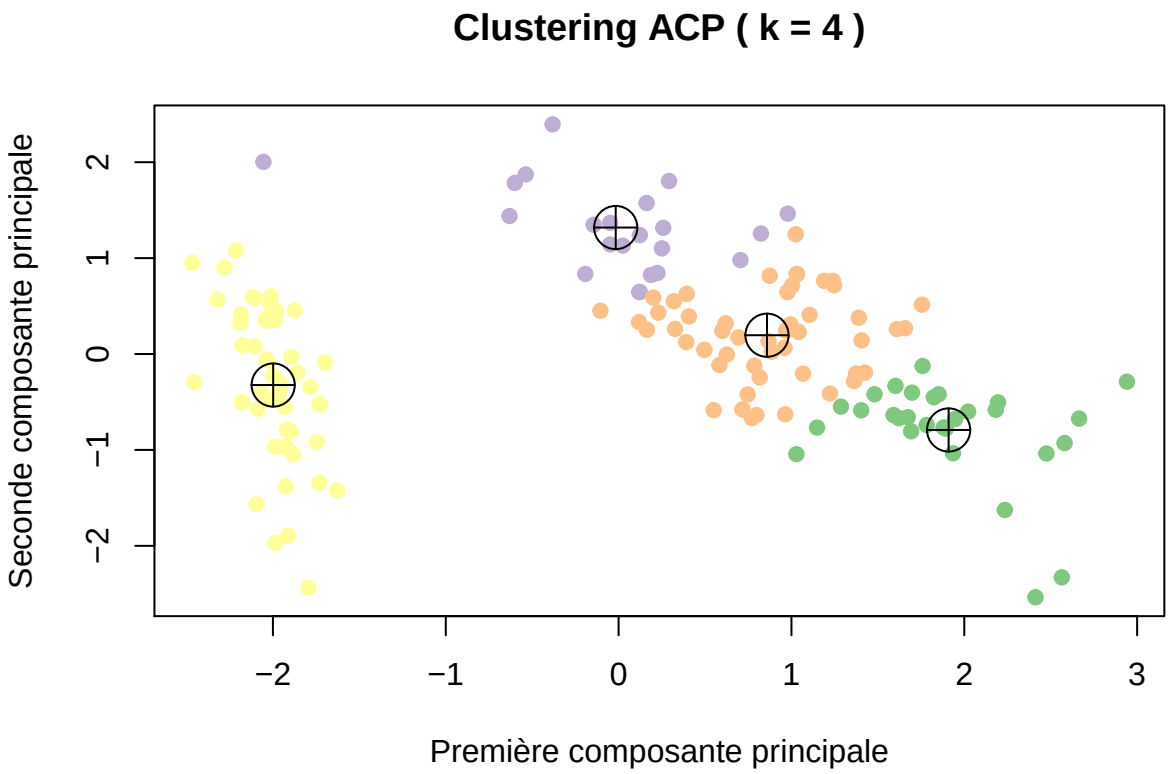


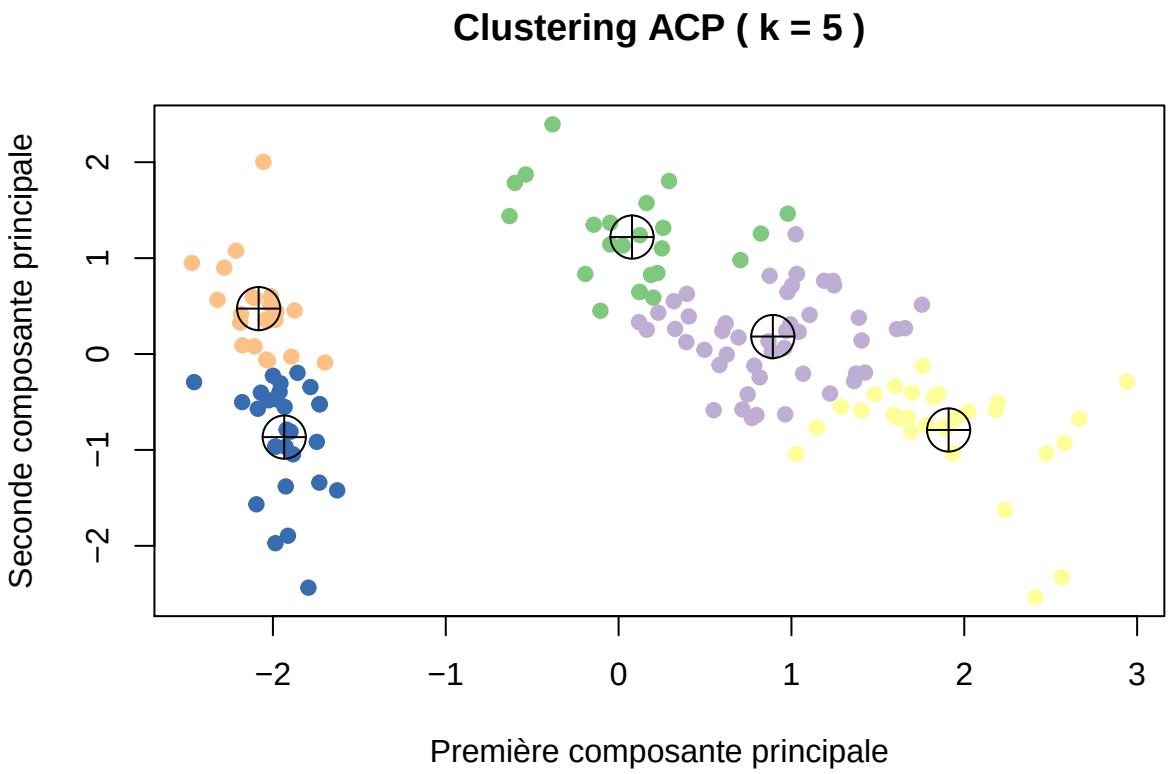
Ici, on utilise une bibliothèque particulière pour utiliser une palette de couleurs rococo, et ce, quelque soit le nombre de classes du jeu de données.

```
library("RColorBrewer")
```

```
for(k in 3:5){
  centering = kmeans(x = XACP, centers = k)
  colors = brewer.pal(k, name = "Accent")
  plot(XACP[,1],
       XACP[,2],
       xlab = "Première composante principale",
       ylab = "Seconde composante principale",
       main = paste("Clustering ACP ( k =",k,")"),
       col = colors[centering$cluster], pch= 19
  )
  points(centering$centers, col = "black", pch = 10, cex = 3)
}
```


Clustering ACP (k = 3)
Seconde composante principale
Première composante principale
−2
−1
0
1
2
3

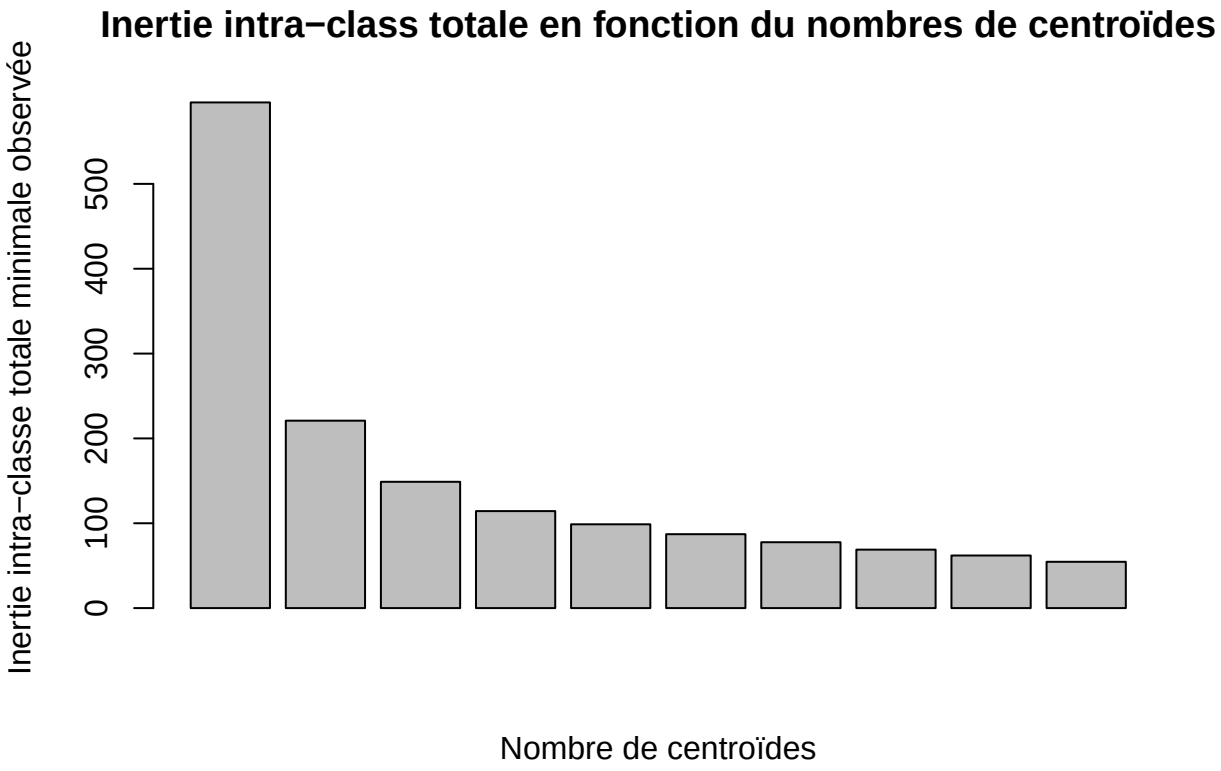

Clustering ACP (k = 4)
Seconde composante principale
Première composante principale
−2
−1
0
1
2
3



Cherchons à déterminer le nombre de classes optimal. Pour cela, on peut réaliser, pour différents nombres k de centroïdes, `kmeans` un nombre `iter` de fois. On peut ensuite prendre l'inertie minimum observée pour chaque valeur de k, les afficher et procéder à la méthode du coude pour choisir combien de centroïdes utiliser.

Ici, on réalise cela pour $k \in [[2, 10]]$ et $iter = 100$:

```
inertieSelonNombreClasses = c()
for(k in 1:10){
  inerties = c()
  for(iter in 1:100){
    centering = kmeans(x = XACP, centers = k)
    inerties[iter] = centering$tot.withinss
  }
  inertieSelonNombreClasses[k] = mean(inerties)
}
barplot(inertieSelonNombreClasses,
        main = "Inertie intra-class totale en fonction du nombres de centroïdes",
        xlab = "Nombre de centroïdes",
        ylab = "Inertie intra-classe totale minimale observée")
```



Ici, on choisirait bien 3 clusters pour procéder à la classification, c'est à dire le nombre de classes du jeu de données.

3.2 Clustering sur les données Crabs

```
require("MASS")
```

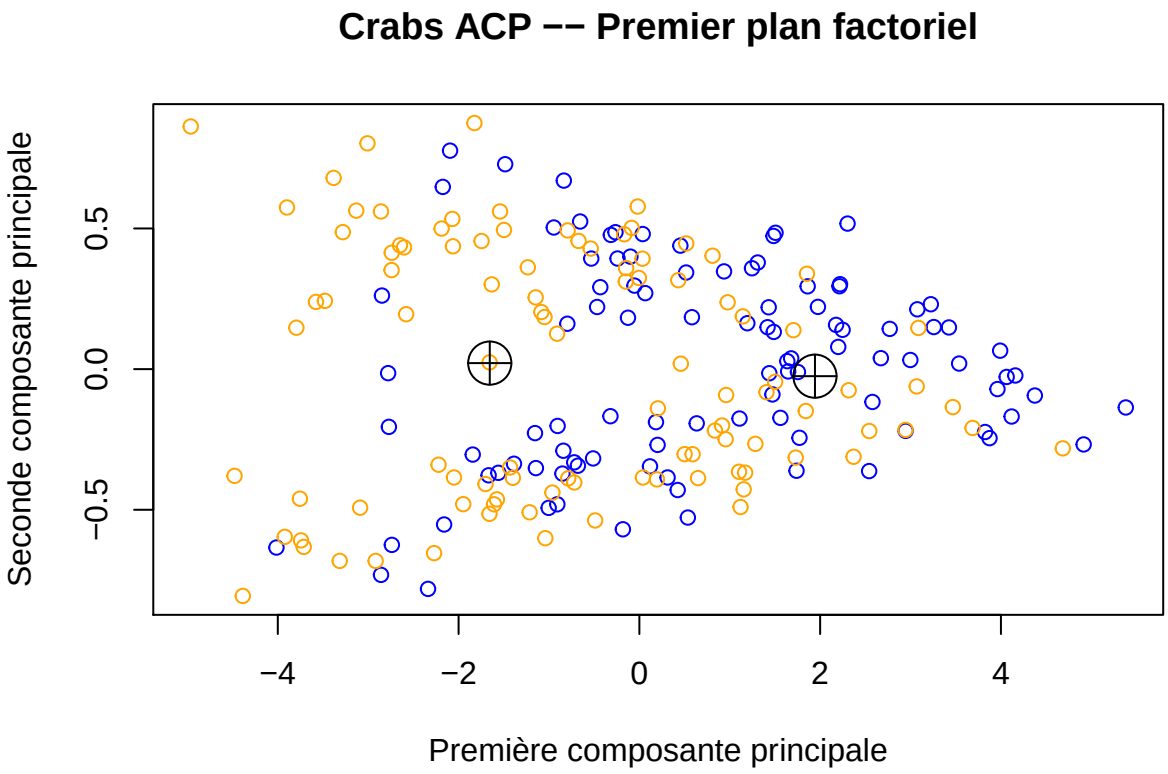
```
## Loading required package: MASS
```

```
length(unique(crabs$sp))
```

```
## [1] 2
```

```
XCrabs = scale(crabs[,-c(1,2,3)], center = TRUE, scale = TRUE)

covMat = cov(x = XCrabs)

ACPCrabs = eigen(covMat)
vPropres = ACPCrabs$values
U = ACPCrabs$vectors

XACPCrabs = XCrabs %*% U

clustering = kmeans(XACPCrabs, centers = 2)

plot(XACPCrabs[,1],
```

```
     XACPCrabs[,2],
     xlab = "Première composante principale",
     ylab = "Seconde composante principale",
     main = "Crabs ACP -- Premier plan factoriel",
     col = c("blue","orange")[crabs$sp]
)

points(clustering$centers, col="black", pch = 10, cex = 3)
```



3.3 Clustering sur les données Mutations

Pour exéctuer `kmeans` sur ce jeu de données, il faut se ramener à une représentation euclidienne. Ici, on choisira un espace de $k = 5$ dimensions.

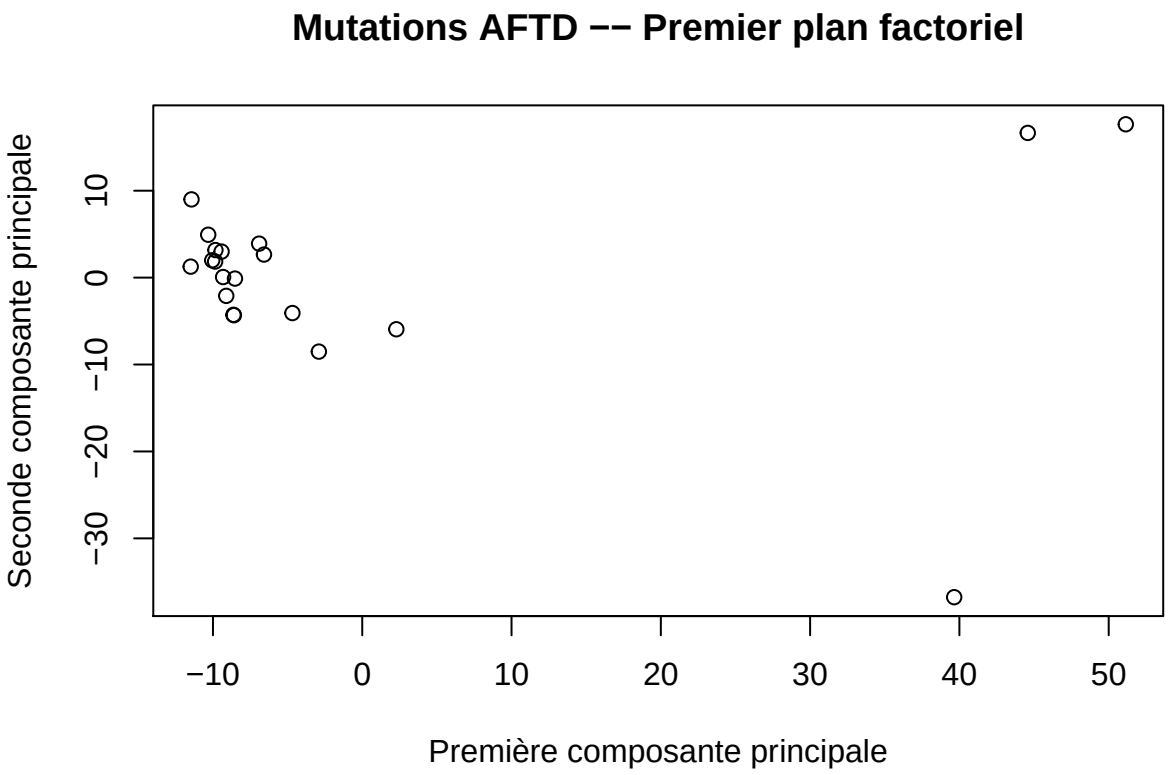
Procédons à cette représentation:

```
mds = cmdscale(mut,
               eig = TRUE,
               k = 5
               )
Xmutation = mds$points
```

Représentons rapidement ce jeu de données dans le premier plan factoriel de l'AFDT:

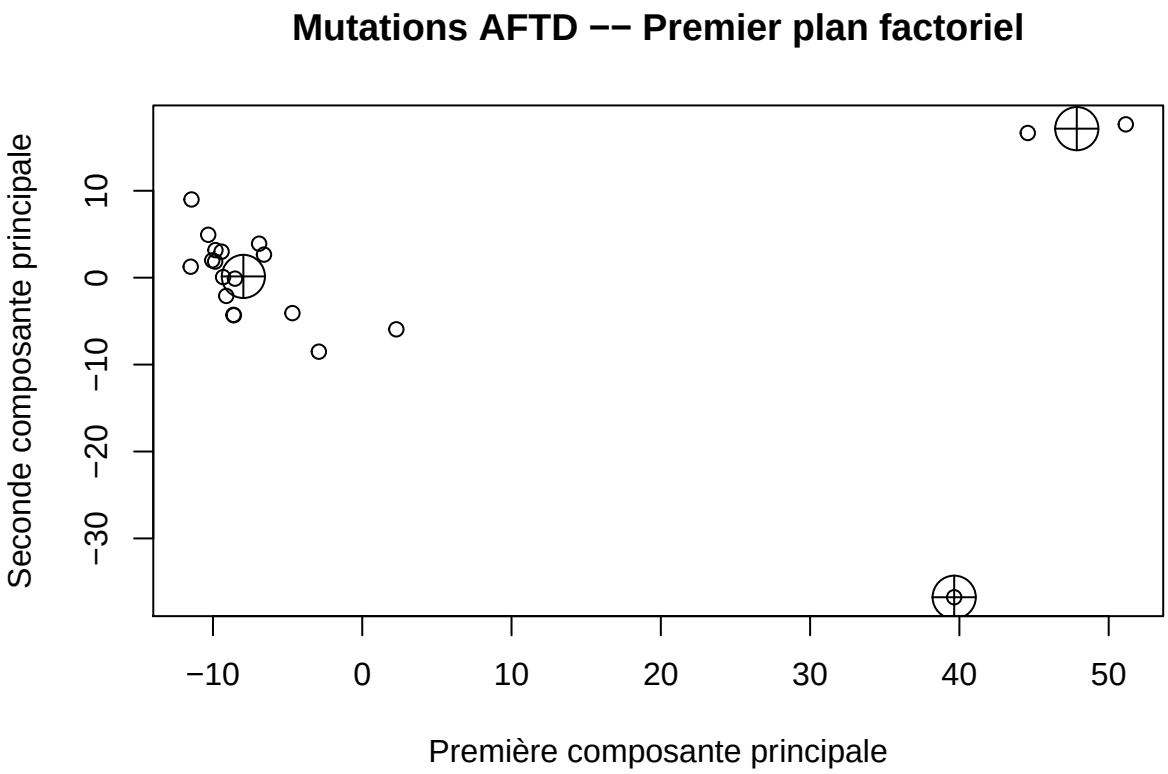
```
plot(Xmutation[,1],
     Xmutation[,2],
```

```
     xlab = "Première composante principale",
     ylab = "Seconde composante principale",
     main = "Mutations AFTD -- Premier plan factoriel"
)
```



Effectuons maintenant une classification avec $K = 3$ classes avec `kmeans`.

```
clustering = kmeans(x = Xmutation, centers = 3)
centers = clustering$centers

plot(Xmutation[,1],
     Xmutation[,2],
     xlab = "Première composante principale",
     ylab = "Seconde composante principale",
     main = "Mutations AFTD -- Premier plan factoriel"
)

points(centers, col="black", pch = 10, cex = 3)
```



Si on réexecute le code plusieurs fois, on peut voir que le résultat change. Pour étudier la stabilité du résultat, on peut s'intéresser à la variance de l'inertie:

```
nIter = 10000
inertie = 1:nIter
for(k in 1:nIter) {
  clustering = kmeans(x = Xmutation, centers = 3,nstart = 1)
  inertie[k] = clustering$tot.withinss
}
```

On peut s'intéresser au nombre de valeur de l'inertie qui correspond approximativement au nombre de configurations localement optimale à permutations près.

```
length(unique(inertie))
```

```
## [1] 6
```

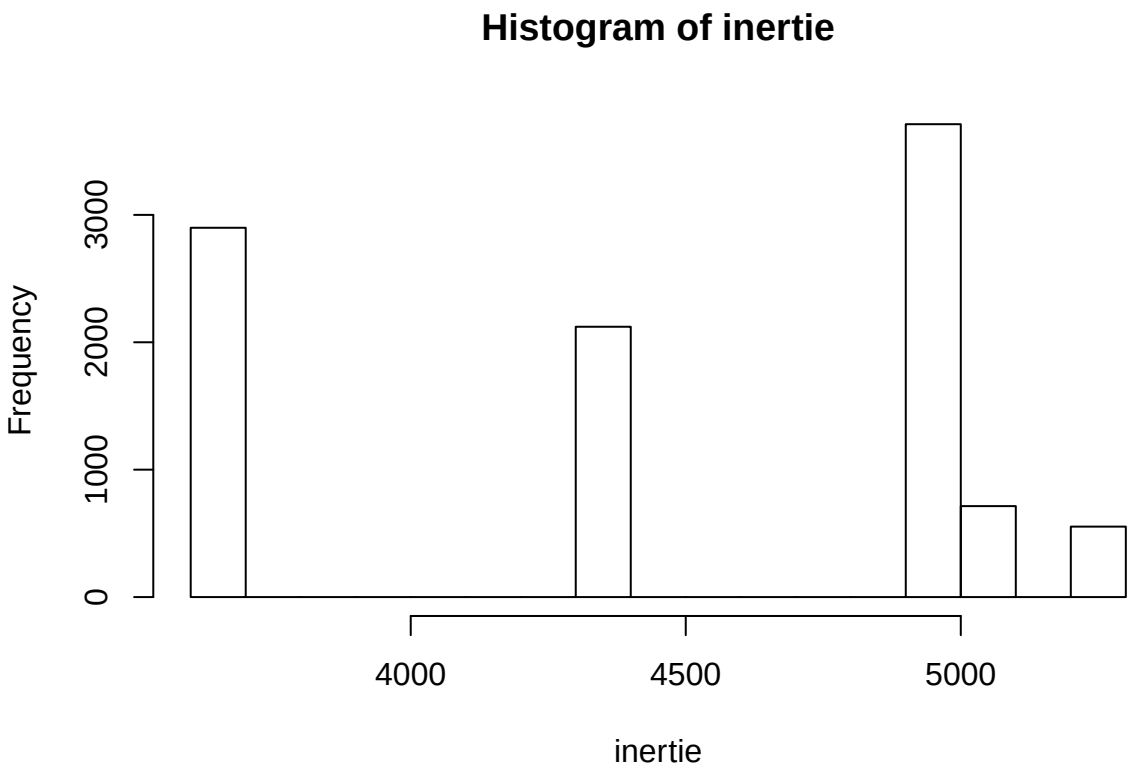
```
mean(inertie)
```

```
## [1] 4469.825
```

```
sd(inertie)
```

```
## [1] 593.1218
```

```
hist(inertie)
```



4 Autour des K-means

Soit z_{ik} un ensemble de variables indicatrices z_{ik} $in\{0,1\}$, pour tous entiers $i \in [[1,n]]$ et $k \in [[1,K]]$.

Le critère d'inertie optimisé par l'algorithme des K-means en fonction des $\mathbf{z}_i$ et des représentnations des groups μ_k est :

$$I = \sum_{k=1}^{K} \sum_{i=1}^{n} z_{ik} \ \|x_i - \mu_k\|^2$$

On calcule la dérivée de I par rapport à μ_k:

$$\frac{\partial I}{\partial \mu_k} = -2 \sum_{i=1}^{n} z_{ik} \ (x_i - \mu_k)$$

Par les conditions d'optimalité on a nécessairement:

$$\frac{\partial I}{\partial \mu_k} = 0$$

Soit de manière équivalente :

$$\sum_{i=1}^{n} z_{ik}\ x_i = \sum_{i=1}^{n} z_{ik}\ \mu_k$$

C'est à dire, puisque $n_k = \sum_{i=1}^{n} z_{ik}$

$$\mu_k = \frac{1}{n_k} \sum_{i=1}^{N} z_{ik}\ x_i$$

Montrons que la hessienne de I, notée $\mathbf{H_I}$ est définie positive.

On a, pour tout $k \in [[1, n]]$:

$$\frac{\partial^2 I}{\partial \mu_k^2} = 2 \sum_{i=1}^{1} z_{ik} = 2\ n_k > 0$$

Ainsi que, pour tout $(k, l) \in [[1, n]]^2$:

$$\frac{\partial^2 I}{\partial \mu_k \partial \mu_l} = 0$$

Ainsi, $\mathbf{H}_I$ est diagonale de termes strictement positifs donc elle est définie positive.

Ce qui montrer que les centres de gravités sont les représentants des groupes minimisant le critère d'inertie.

Supposons que les $(\mu_k)_k$ sont maintenant fixés. Optimiser I selon les x_i revient à minimiser pour chaque x_i la quantité $\|x_i - \mu_k\|^2$. Cette quantité est minimisée en prenant le μ_k le proche de x_i.