Analyse de données

Implémentation de l'algorithme des k-means++

Yoann Bonnet (@ENSIIE)

Semestre 3, 2022-2023

Exercice 1 – L'algorithme des k-means++

Programmation de l'algorithme décrit dans l'article scientifique

On implémente plusieurs **fonctions auxiliaires**.

1a. La première d'entre elles est une fonction qui renvoie un centre choisi aléatoirement dans notre ensemble de points $\mathcal{X}$.

```
randomCenters <- function(X,d){
  C_1 <- X[,sample(1:ncol(X),1)]
  ind <- which(sapply(1:ncol(X), function(i) all(X[,i] == C_1)))
  X <- matrix(X[,-ind], nrow = d)
  return(C_1)
}
```

1b. Pour réaliser cette étape de l'algorithme, nous avons d'ores et déjà besoin de programmer une fonction auxiliaire qui calcule $D(x)$, la distance la plus courte entre un point de $\mathcal{X}$ et le centre le plus proche que nous avons déjà choisi.

```
distance_fun <- function(x,C){ # x est un point de X, C l'ensemble des centres
  matrix <- x-C
  d <- sqrt(sum(matrix^2))
  return(d)
}
```

Une fois cette fonction implémentée, nous pouvons maintenant en implémenter une nouvelle renvoyant un tableau contenant les distances $D(x)$ pour chaque $x \in \mathcal{X}$ à l'ensemble des centres $\mathcal{C}$.

```
allDistancesSquared <- function(X,C){
  tmp <- matrix(0, nrow = 1, ncol = ncol(X))
  for (i in 1:ncol(X)){
    tmp[,i] <- ((distance_fun(X[,i],C))**2)
  }
  return(tmp)
}
```

Finalement, nous pouvons sélectionner les k centres en utilisant la probabilité calculée.

```
library(MASS)
kmeansPP <- function(X,k,d){
  i <- 2
  C <- matrix(0, nrow=d, ncol=k)
  C[,1] <- randomCenters(X,d)
```

```
  while(i <= k){
    p <- c()
    for (q in ncol(X)){
      s <- c()
      Y <- X[,q]
      for (k in ncol(C)){
        s <- c(s, distance_fun(Y,C[,k]))
      }
      p <- c(p, sum(s))
    }
    j <- which.max(p)
    x <- X[,j]
    C[,i] <- x
    ind <- which(sapply(1:ncol(X), function(j) all(X[,j] == x)))
    X <- matrix(X[,-ind], nrow=d)
    i <- i + 1
  }
  return(kmeans(t(X),t(C)))
}
```

Simulation des données NORM-10 et NORM-25

Nous générons dans un premier temps un hypercube de longueur de côté 500 :

```
createHypercube <- function(d,n,l){
 hypercube <- matrix(0, nrow = d, ncol = n)
 for (i in 1:d){
   for (j in 1:n){
     hypercube[i,j] <- sample(1:l, size = 1)
   }
 }
 return(hypercube)
}
```

Afin de générer nos ensembles de données `NORM-10` et `NORM-25`, nous sélectionnons aléatoirement 10 (respectivement 25) colonnes de notre hypercube.

```
randomSelection <- function(d,n,l,k){
  k <- sample(1:n, size=k)
  hypercube <- createHypercube(d,n,l)
  return(hypercube[,k])
}
```

Nous pouvons, enfin, créer notre fonction qui nous permettra de créer nos ensembles `NORM-10` et `NORM-25` ainsi que toute autre fonction similaire.

```
library(MASS)
adding_points <- function(d,z){
  Sigma <- matrix(0,d,d)
  for (i in 1:d){
    Sigma[i,i] <- 1
  }

  m = 200
  norm <- matrix(0, nrow = d, ncol = z+m*z)
```

```
  sel <- randomSelection(5,10000,500,z)

  norm[,1:z] <- sel[,1:z]

  for (i in 1:z){
    a <- (z+1) + 200*(i-1)
    b <- z + 200*i
    norm[,a:b] <- t(mvrnorm(m,sel[,i],Sigma))
  }

  return(norm)
}
```

Comparaison au kmeans classique

En utilisant les deux fonctions précédentes, nous pouvons générer les deux ensembles qui nous sont demandés afin de pouvoir réaliser la comparaison avec le `kmeans` classique.

```
norm_10 <- adding_points(d = 5, z = 10)
norm_25 <- adding_points(d = 5, z = 25)
```

Nous programmons ensuite la fonction de coût qui nous est suggérée dans l'article scientifique.

```
phi <- function(X,C){
  d <- c()
  for(j in 1:ncol(C)){
    d <- c(d, min(allDistancesSquared(X,C[,j])))
  }
 return(sum(d))
}
```

Enfin, nous créons notre fonction afin d'obtenir les données voulues pour l'algorithme que nous venons de programmer, à savoir le minimum et la moyenne de la fonction de coût ainsi que le temps d'exécution.

```
comparison_PP <- function(X,k,n,d){
  phi_values <- c()
  time <- c()
  for(i in 1:20){
    startTime <- Sys.time()
    CPP <- kmeansPP(X,k,d)$centers
    phi_values <- c(phi_values, phi(X,CPP))
    endTime <- Sys.time()
    time <- c(time, endTime - startTime)
  }

  min_phi <- min(phi_values)/n
  mean_phi <- mean(phi_values)/n
  mean_time <- mean(time)
  return(c(min_phi, mean_phi, mean_time))
}
```

Nous créons une fonction similaire afin d'obtenir ces données pour l'algorithme du `k-means` classique.

```
comparison_classique <- function(X,k,n,d){
  phi_values <- c()
  time <- c()
```

```
  for(i in 1:20){
    startTime <- Sys.time()
    C <- kmeans(X,k)$centers
    phi_values <- c(phi_values, phi(X,C))
    endTime <- Sys.time()
    time <- c(time, endTime - startTime)
  }

  min_phi <- min(phi_values)/n
  mean_phi <- mean(phi_values)/n
  mean_time <- mean(time)
  return(c(min_phi, mean_phi, mean_time))
}
```

Nous pouvons maintenant lancer ces fonctions de comparaison sur notre jeu de données `NORM-10`, avec $n = 10000$ et $d = 5$.

```
kmeans_PP_10 <- comparison_PP(norm_10,10,10000,5)
kmeans_classique_10 <- comparison_classique(t(norm_10),10,10000,5)
```

Nous pouvons finalement créer un tableau similaire à celui de l'article scientifique afin de comparer nos résultats :

```
# Définition de la matrice pour comparer nos résultats
comparison_norm_10 <- matrix(0, nrow = 2, ncol = 3)
rownames(comparison_norm_10) <- c("kmeans++", "kmeans")
colnames(comparison_norm_10) <- c("min(phi)", "mean(phi)", "mean(time)")
comparison_norm_10[1,] <- kmeans_PP_10
comparison_norm_10[2,] <- kmeans_classique_10
comparison_norm_10
```

```
##            min(phi)  mean(phi)  mean(time)
## kmeans++   102.5713   109.5057 0.065181291
## kmeans   24785.1050 31116.6518 0.001950717
```

Nous pouvons faire de même pour `NORM-25`, toujours avec les mêmes valeurs pour n et d.

```
kmeans_PP_25 <- comparison_PP(norm_25,25,10000,5)
kmeans_classique_25 <- comparison_classique(t(norm_25),25,10000,5)
```

Nous pouvons finalement créer un tableau similaire à celui de l'article scientifique afin de comparer nos résultats :

```
# Définition de la matrice pour comparer nos résultats
comparison_norm_25 <- matrix(0, nrow = 2, ncol = 3)
rownames(comparison_norm_25) <- c("kmeans++", "kmeans")
colnames(comparison_norm_25) <- c("min(phi)", "mean(phi)", "mean(time)")
comparison_norm_25[1,] <- kmeans_PP_25
comparison_norm_25[2,] <- kmeans_classique_25
comparison_norm_25
```

```
##            min(phi)  mean(phi)  mean(time)
## kmeans++   332.6342   341.8478 0.282672858
## kmeans   86357.8035 93758.7399 0.005759859
```

Conclusion. Nous pouvons voir, dans les deux cas, que notre algorithme du `kmeans++` est légèrement moins rapide que l'algorithme des `kmeans` classique, mais qu'il est beaucoup moins coûteux que l'algorithme classique d'un **facteur 100** environ.

Exercice 2 – Données `iris`

Application de l'algorithme au jeu de données iris

On récupère tout d'abord les données `iris` :

```
library(MASS)
df <-  iris
dataframe <- df[,1:4]
df_PP <- t(dataframe)
dataframe_PP <- data.frame(df_PP)
#head(dataframe)
```

Dans un premier temps, il nous faut déterminer de manière optimale le nombre de clusters à choisir. Pour cela, on utilise la fonction `fviz_nbclust` du package `factoextra` :

```
library("NbClust")
library(factoextra)
```

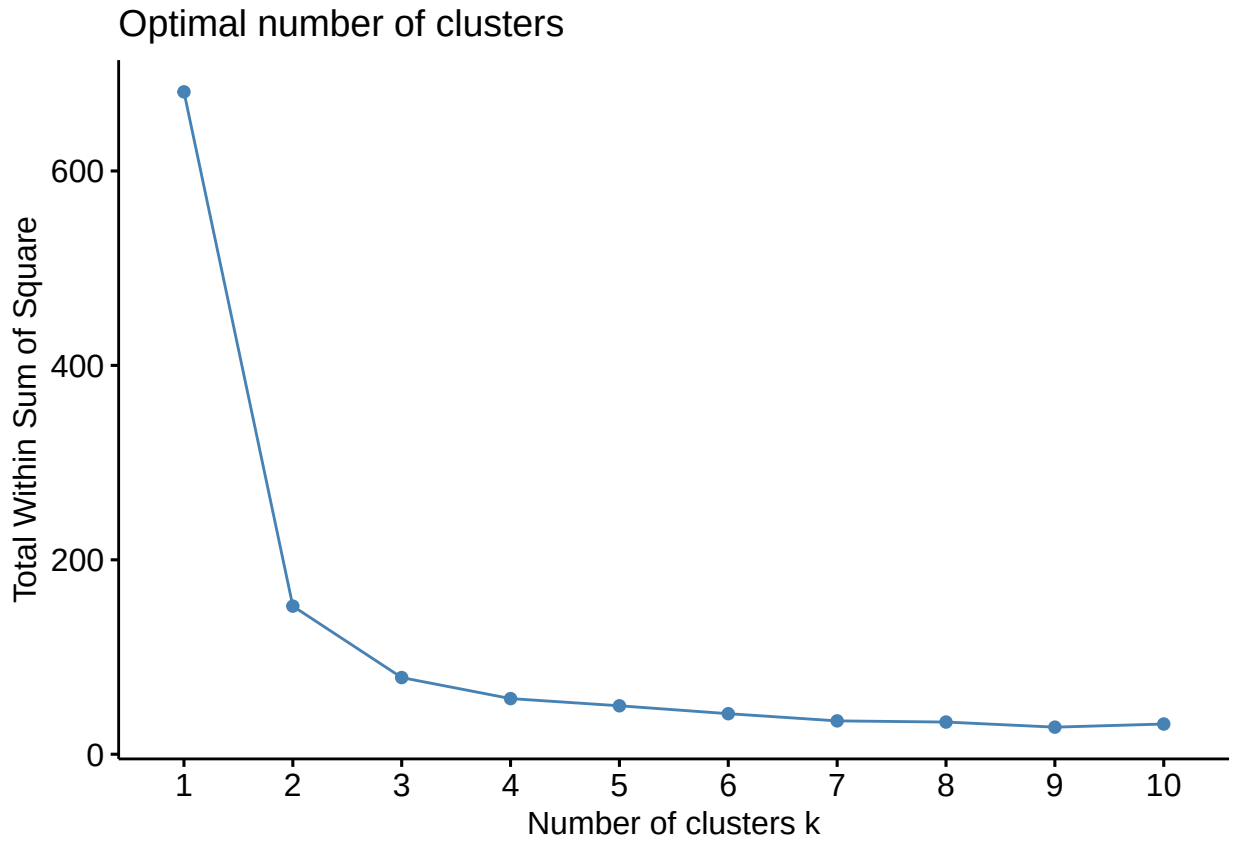
```
## Warning: le package 'factoextra' a été compilé avec la version R 4.2.3
```

```
## Le chargement a nécessité le package : ggplot2
```

```
## Warning: le package 'ggplot2' a été compilé avec la version R 4.2.3
```

```
## Welcome! Want to learn more? See two factoextra-related books at https://goo.gl/ve3WBa
```

```
fviz_nbclust(dataframe, kmeans, method = "wss")
```



Le nombre optimal de clusters à appliquer à la méthode kmeans est l'abscisse du point où la courbe se casse. Ici, on trouve $k = $.

On peut donc appliquer l'algorithme que nous venons de programmer à notre jeu de données.

Pour obtenir les valeurs concernant ϕ et la durée, nous implémentons une fonction similaire aux fonctions de comparaison précédentes :

```
library(mclust)
```

```
## Warning: le package 'mclust' a été compilé avec la version R 4.2.3
```

```
comparison_mclust <- function(X,k,n,d){
  phi_values <- c()
  time <- c()
  for(i in 1:20){
    startTime <- Sys.time()
    C <- Mclust(X, k)$parameters$mean
    phi_values <- c(phi_values, phi(X,C))
    endTime <- Sys.time()
    time <- c(time, endTime - startTime)
  }

  min_phi <- min(phi_values)/n
  mean_phi <- mean(phi_values)/n
  mean_time <- mean(time)
  return(c(min_phi, mean_phi, mean_time))
}
```

Nous lançons les fonctions de comparaison pour les trois méthodes utilisées :

```
k <- 3
kmeans_classique_iris <- comparison_classique(dataframe,3,10000,4)
kmeans_PP_iris <- comparison_PP(df_PP,3,10000,4)
kmeans_mclust_iris <- comparison_mclust(dataframe,3,10000,4)
```

Nous pouvons finalement créer un tableau afin de comparer nos résultats :

```
# Définition de la matrice pour comparer nos résultats
comparison_iris <- matrix(0, nrow = 3, ncol = 3)
rownames(comparison_iris) <- c("kmeans++", "kmeans", "mclust")
colnames(comparison_iris) <- c("min(phi)", "mean(phi)", "mean(time)")
comparison_iris[1,] <- kmeans_PP_iris
comparison_iris[2,] <- kmeans_classique_iris
comparison_iris[3,] <- kmeans_mclust_iris
comparison_iris
```

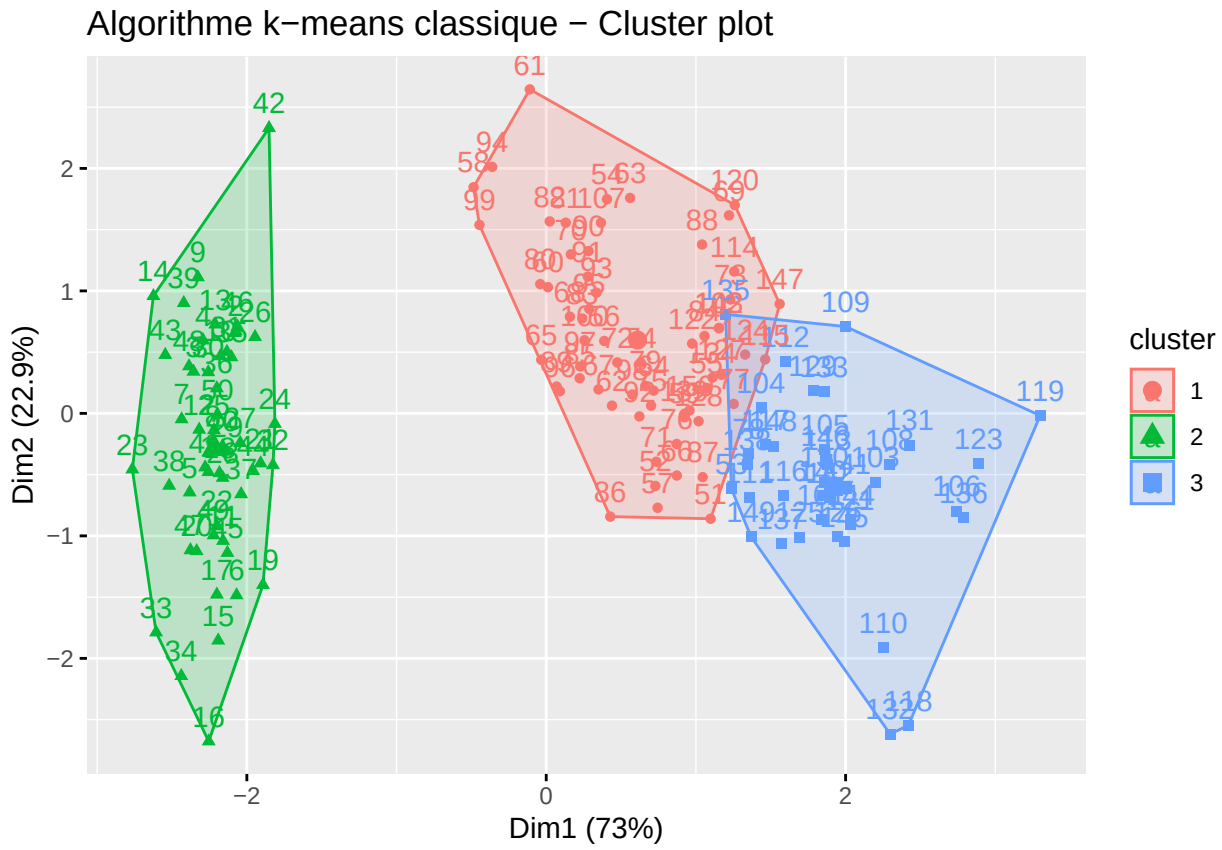
```
##             min(phi)   mean(phi)   mean(time)
## kmeans++ 0.005672241 0.006043497 0.0477446556
## kmeans   0.081790619 0.095932400 0.0007709622
## mclust   0.179691687 0.179691687 0.0586342335
```

Conclusion. Nous pouvoir voir que, dans le cas des données `iris`, notre algorithme du `kmeans++` est également le moins couteux, suivi de près par l'algorithme classique puis, avec un coût 10 fois supérieur, le clustering réalisé avec la fonction `Mclust`. Concernant le temps d'exécution, il est, *en moyenne*, plus faible avec le `kmeans` classique qu'avec notre algorithme du `kmeans++`, ce dernier ayant un temps d'exécution moyen relativement proche de l'algorithme `Mclust`.

Visualisation des différentes partitions du dataset sur les premiers plans d'une analyse en composante principale

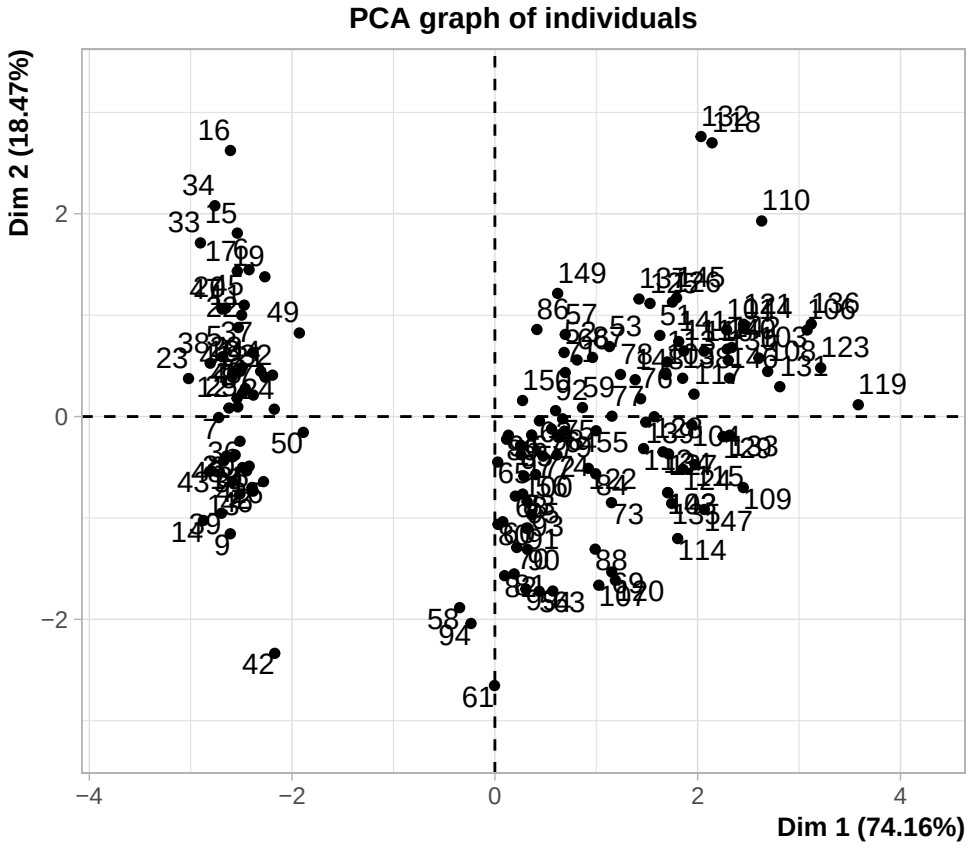
- Algorithme des `k-means` classique

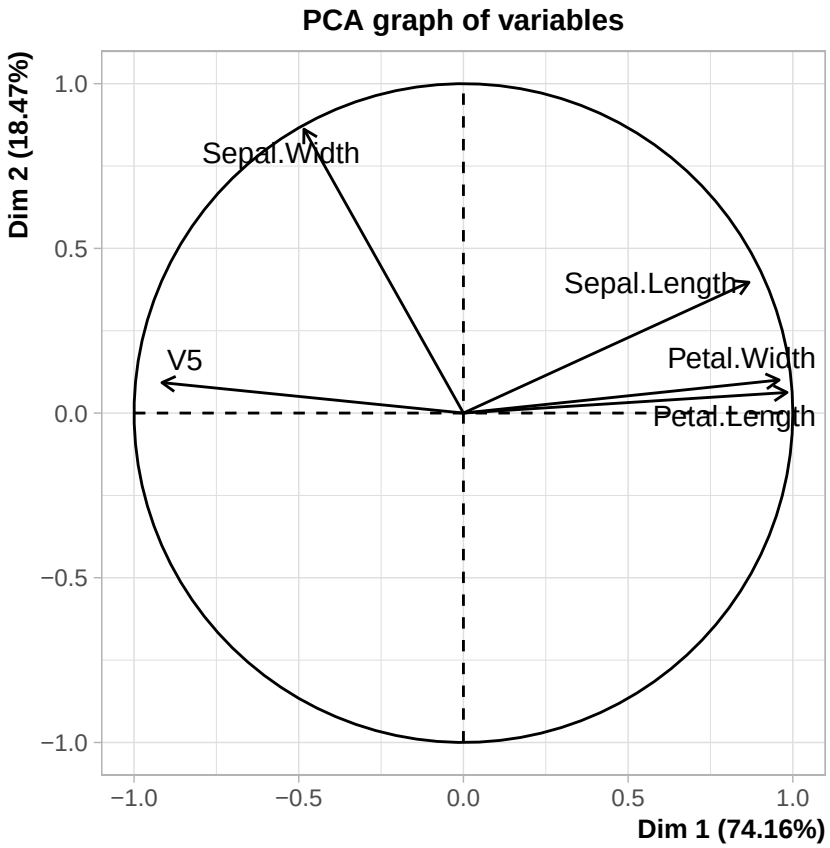
```
library(cluster)
kmeans_iris <- kmeans(dataframe, 3)
fviz_cluster(kmeans_iris,
             data = dataframe,
             main = "Algorithme k-means classique - Cluster plot",
             outlier.color = "purple")
```



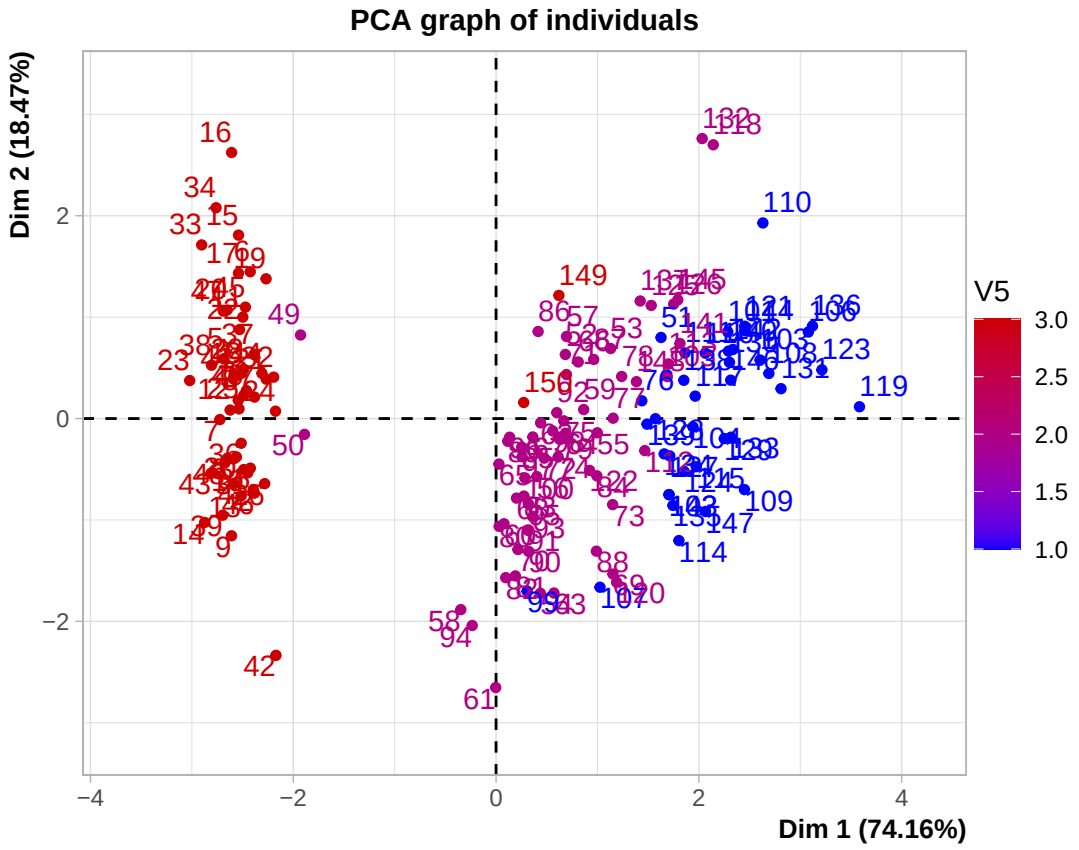
- Algorithme des `k-means++`

```
library(FactoMineR)
kmeans_PP_iris <- kmeansPP(df_PP, 3, 4)
x <- as.data.frame(t(rbind(df_PP, kmeans_PP_iris$cluster)))
pca <- PCA(x, scale.unit = TRUE, ncp = 4)
```


PCA graph of individuals
Dim 2 (18.47%)
Dim 1 (74.16%)
2
0
−2
−4
−2
0
2
4

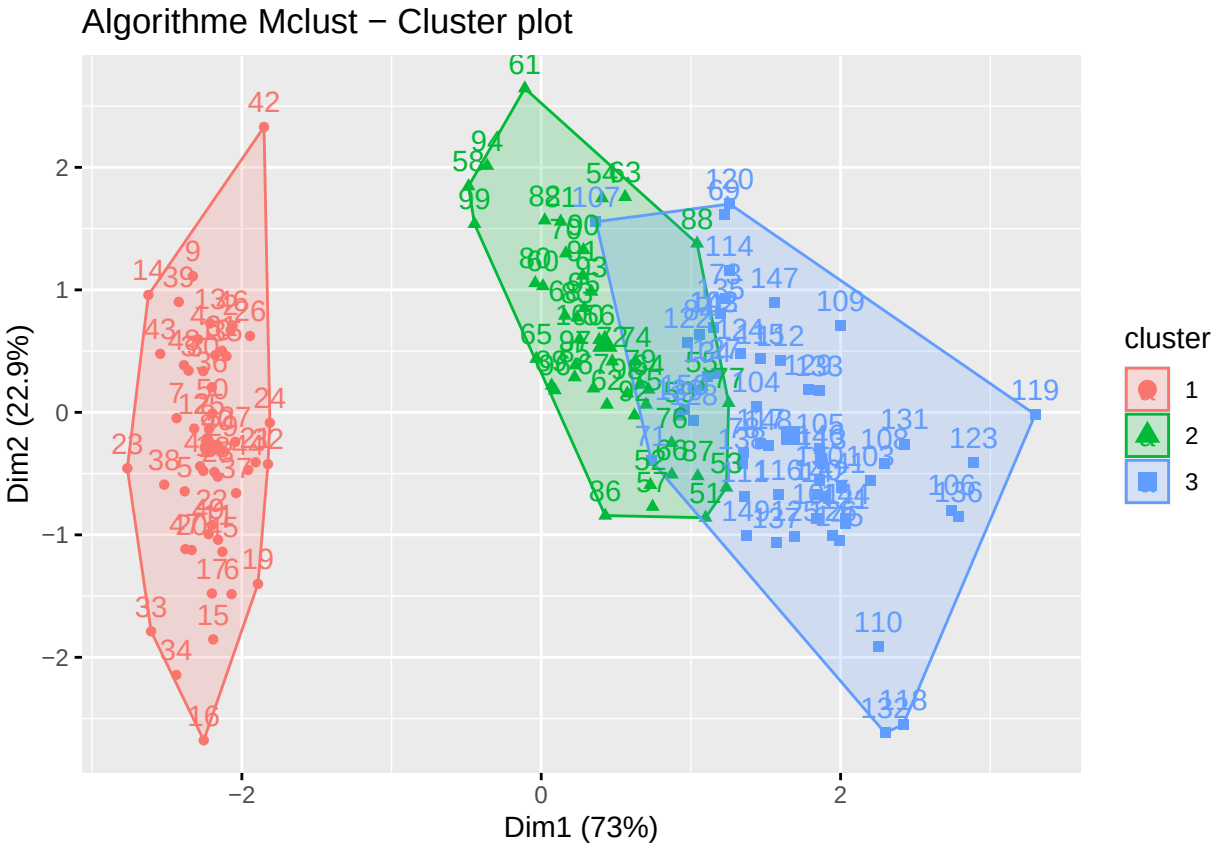



```
plot.PCA(pca, habillage = 5)
```



- Algorithme `Mclust`

```
Mclust_iris <- Mclust(dataframe, 3)
fviz_cluster(Mclust_iris,
             data = dataframe,
             main = "Algorithme Mclust - Cluster plot",
             outlier.color = "purple")
```



Commentaires

Nous pouvons voir que près de **96** de la variance totale est exprimée par les deux premières composantes, ce qui suffit à dire que les **deux premiers axes suffisent à retranscrire l'information**. Nous pouvons donc réaliser une ACP en deux composantes.

Toutefois, le résultat de l'ACP sur notre `k-means++` est le moins performant, les trois clusters restant très rapprochés.