Project - Kmeans++

Adib Habbou - Alae Khidour

11-12-2022

Exercice 1 - Algorithme des Kmeans++ :

Question 1 :

Distance euclidienne :

```
# fonction qui caclcule la distance euclidienne entre deux points
distance <- function(x, y)
{
  # on stocke la longueur de nos vecteurs et on initialise la distance par 0
  n <- length(x)
  d <- 0
  # on parcours tous les éléments de nos vecteurs
  for (i in 1:n)
  {
    # à chaque itération on ajoute le carré différence entre les deux éléments
    d <- d + (x[i] - y[i])^2
  }
  return (sqrt(d))
}
```

Distance au centre le plus proche :

```
# fonction qui renvoie la distance au centre le plus proche
D <- function(x, centres)
{
  # on initialise avec la distance au premier centre
  min <- distance(x, centres[1,])
  # si il y un seul centre en renvoie min sinon on parcours tous les centres
  if (nrow(centres) == 1) return (min)
  for (i in 2:nrow(centres))
  {
    # on calcule la distance si elle est inférieur au min on met à jour la valeur
    tmp <- distance(x, centres[i,])
    if (tmp < min) min <- tmp
  }
  return (min)
}
```

Initialisation des centres :

```
# fonction qui initialise les centres suivant l'algorithme kmeans++
init_centres <- function(data, k)
{
  # on transforme nos données en matrice pour éviter les erreurs de type
  data <- as.matrix(data)

  # on stocke le nombre de ligne de notre matrice
  n <- nrow(data)

  # on stocke le nombre de colonne de notre matrice
  dim <- ncol(data)

  # on initialise les centres en prenant aléatoirement un point de nos données
  centres <- matrix((data[sample(1:n, 1),]), nrow = 1, ncol = dim)

  # on cherche à trouver les k - 1 centres manquants
  for (j in 2:k)
  {
    # on initialise avec la distance entre le premier point et les centres déjà choisis
    max <- c(D(data[1,], centres), data[1,])
    # on parcours tous les points de nos données
    for (i in 2:n)
    {
      # à chaque itération on calcule la distance avec les centres déjà choisis
      tmp <- c(D(data[i,], centres), data[i,])
      # si la distance est supérieur au maximum on met à jour la valeur
      if (tmp[1] > max[1]) max <- tmp
    }
    # on ajoute aux centres déjà choisis le plus éloigné qu'on a trouvé
    centres <- rbind(centres, max[2:(dim+1)])
  }

  # on renvoie la liste des centres
  return (centres)
}
```

Centre le plus proche :

```
# fonction qui renvoie le centre le plus proche par rapport au point donné
centre_proche <- function(x, centres)
{
  # on initialise avec la distance au premier centre
  min <- distance(x, centres[1,])
  # on initialise le centre le plus proche avec le premier centre
  centre <- centres[1,]
  # si il y un seul centre en renvoie min sinon on parcours tous les centres
  if (nrow(centres) == 1) return (centre)
  for (i in 2:nrow(centres))
  {
    # on calcule la distance au centre si elle est inférieur au min on met à jour
    tmp <- distance(x, centres[i,])
    if (tmp < min)
    {
      min <- tmp
      centre <- centres[i,]
    }
  }
  # on renvoie le dernier centre qui est donc le plus proche du point donné
  return (centre)
}
```

Barycentre :

```
# fonction qui calcule le barycentre d'un ensemble de points
barycentre <- function(x)
{
  # on stocke le nombre de point sans prendre en compte les valeurs manquantes
  n <- length(x) - sum(is.na(x))
  # on stocke la dimension de chaque point
  dim <- length(x[1])
  # on initialise le barycentre avec que des 0
  bar <- rep(0, dim)
  # on parcours chaque composante de chacun de nos points
  for (i in 1:dim)
  {
    for (k in 1:n)
    {
      # on ajoute à chaque fois la valeur de la ci-ème omposante du k-ème point
      bar[i] <- bar[i] + x[k][i]
    }
    # on divise la somme obtenu par le nombre de points
    bar[i] <- bar[i] / n
  }
  # on renvoie au final le vecteur des barycentres
  return(bar)
}
```

Algorithme kmeans++ :

```
# fonction qui applique l'algorithme des kmeans++ manuellement
kmeans_pp <- function(data, k)
{
  # on stocke le temps de début d'exécution
  debut <- Sys.time()
  # on transforme nos données en matrice pour éviter les erreurs de type
  data <- as.matrix(data)
  # on stocke le nombre de ligne de notre matrice
  n <- nrow(data)
  # on stocke le nombre de colonne de notre matrice
  dim <- ncol(data)
  # on initialise les centres suivant la méthode kmeans++
  centres <- init_centres(data, k)

  # on applique algorithme kmeans++
  while (TRUE)
  {
    # on initialise la matrice des clusters avec des NA
    matrice_clusters <- array(data = NA, dim = c(n, k, dim))
    # on initialise la première ligne de la matrice des clusters avec les centres calculés
    matrice_clusters[1,,] <- centres

    # on assigne chaque point de nos données au cluster du centre le plus proche
    for (i in 1:n)
    {
      # si le point considéré est un centre on arrête pour éviter d'avoir des doublons
      if(!is.na(row.match(data[i,], centres))) break
      # le cluster auquel appartient le point grâce à son centre le plus proche
      colonne <- which(centres == centre_proche(data[i,], centres), arr.ind = TRUE)[1]
      # la position où on peut ajouter le point suivant le remplissage des colonnes
      ligne <- match(NA, matrice_clusters[, colonne,])
      # on ajoute le i-ème point au bon cluster
      matrice_clusters[ligne, colonne,] <- data[i,]
    }
    # on stocke les centres de l'itération d'avant
    historique_centres <- centres
    # on calcule les nouveaux centres grâce au barycentre
    for (i in 1:k)
    {
      for (j in 1:dim)
      {
        # on met à jour chaque centre par le barycentre des points du cluster
        centres[i,j] <- barycentre(matrice_clusters[,i,j])
      }
    }
    # on vérfie si les centres ont changé
    if (sum(historique_centres - centres == matrix(0, nrow=k, ncol=dim)) == k*dim) break
  }

  # on initialise les clusters avec nos données et une colonne contenant que des 0
  # la dernière colonne contiendra une valeur indiquant à quel cluster appartient le point
```

```
  clusters <- cbind(data, rep(0,n))
  # on initialise le potentiel à 0
  potentiel <- 0
  for (i in 1:n)
  {
    # on parcours chaque cluster de notre matrice
    for (j in 1:k)
    {
      # on ajoute à la dernière colonne l'indice de la colonne auquel appartient le point
      if(!is.na(which(data[i,] == matrice_clusters[,j,], arr.ind = TRUE)[1]))
      {
        clusters[i,(dim+1)] <- j
      }
    }
    # on calcule la somme des distances entre chaque point et son centre le plus proche
    potentiel <- potentiel + distance(data[i,], centres[clusters[i,(dim+1)],])^2
  }

  # on stocke le nombre de point par clusters
  taille <- vector(length = k)
  for (i in 1:k)
  {
    # pour chauqe cluster on compte le nombre de point lui appartenant
    taille[i] <- sum(clusters[,(dim+1)] == i)
  }

  # on stocke le temps de fin d'exécution
  fin <- Sys.time()
  # on calcul le temps d'exécution de l'algorithme
  temps <- difftime(fin, debut)

  # on renvoie l'ensemble des résultats de l'algorithme
  return(list(nb_clusters = k, # nombre de clusters
              centres = centres, # liste des centres
              clusters = clusters, # liste des clusters
              taille = taille, # nombre d'éléments par clusters
              potentiel = potentiel, # somme des distances au carré
              temps = as.numeric(temps, units = "secs") # temps d'execution en secondes
  ))
}
```

Algorithme kmeans++ optimisé :

```
# fonction qui applique l'algorithme des kmeans++ en utilisant la fonction kmeans de R
opti_kmeans_pp <- function(data, k)
{
  # on appelle la fonction kmeans de R en initialisant les centres suivant les kmeans++
  return (kmeans(data, init_centres(data, k)))
}
```

Question 2 :

Génération des "vrais" centres :

```
# fonction qui génère n "vrais" centres choisis dans un hypercube
# hypercube de dimension dim de côté de taille a
generation_centre <- function(n, dim, a = 500)
{
  # on initialise la liste avec n points à dim composantes NA
  res <- matrix(NA, nrow = n, ncol = dim)
  # on parcours chacun des lignes et on génére un point à dim composante dans [0,a]
  for (i in 1:n) { res[i,] <- sample(0:a, dim) }
  return (res)
}
```

Génération des nuages de points :

```
# fonction qui génère un nuage de points de taille size
# autour du centre grâce à une gaussienne de variance 1
augmentation_data <- function(centre, size)
{
  # on stocke le nombre de composantes de nos centres
  dim <- length(centre)
  # on renvoie la matrice qui génére une gaussienne  de size point
  # autour du centre avec une variance 1
  return (matrix(rnorm(dim * (size-1), mean = centre, sd = 1),
                 nrow = size-1, ncol = dim, byrow = TRUE))
}
```

Génération des datasets :

```
# fonction qui génère un dataset à partir de n centres de dimension dim en créant autour
# de ces centres des nuages de points de taille size grâce à une gaussienne de variance 1
generation_data <- function(n, dim, size)
{
  # on génère n centre à dim composantes
  CENTRE <- generation_centre(n, dim)
  # on initialise notre dataset avec les centres générés
  NORM <- CENTRE
  # on parcours les centres générés et on ajoute le nuage de point généré autour de lui
  for (i in 1:nrow(CENTRE)) { NORM <- rbind(NORM, augmentation_data(CENTRE[i,], size)) }
  return (NORM)
}
```

Génération de NORM-10 et NORM-25 :

```
# on généré un data set de 10000 points avec 10 centres
# choisis dans un hypercube de dimension 5 de côté de taille 500
NORM_10 <- generation_data(n = 10, dim = 5, size = 1000)
# on généré un data set de 10000 points avec 25 centres
# choisis dans un hypercube de dimension 15 de côté de taille 500
NORM_25 <- generation_data(n = 25, dim = 15, size = 400)
```

Application du Kmeans++ sur NORM-10 :

```
# on applique l'algorithme des kmeans++ qu'on a codé sur NORM_10
kmeans_pp_NORM_10 <- kmeans_pp(NORM_10, 10)
```

```
kmeans_pp_NORM_10$nb_clusters
```

```
[1] 10
```

```
kmeans_pp_NORM_10$centres
```

```
           [,1]       [,2]      [,3]      [,4]      [,5]
 [1,] 337.99650 434.000263 292.04893 265.96404 295.00707
 [2,]  29.98459  83.952485  84.94130 240.96944 466.02855
 [3,] 413.97546   1.981286 304.04036 153.00040 339.01099
 [4,]  63.05566  34.970771 330.99492 343.02727 201.00649
 [5,] 266.08010 324.934973 134.01461  61.06004  50.00095
 [6,] 297.98723  16.990065  65.98333  21.96708 307.04125
 [7,] 289.97698 276.994286  55.98389 196.98823 287.98721
 [8,] 448.01052 452.971048  61.97411 194.05640 288.97242
 [9,] 326.96688 376.030131 450.95873 239.98961 300.00607
[10,] 320.95877 480.048920 117.98028  80.97514  50.05572
```

```
kmeans_pp_NORM_10$clusters[sample(1:10000,10),]
```

```
           [,1]      [,2]      [,3]     [,4]      [,5] [,6]
 [1,] 321.61146 480.32690 118.55630  81.6790  49.53629   10
 [2,] 449.71045 452.18990  62.53210 193.9377 288.76814    8
 [3,] 447.40535 452.09957  60.94977 193.6002 287.17914    8
 [4,] 338.65429 434.74442 291.52524 267.0323 296.59419    1
 [5,] 288.92157 276.50799  55.73017 197.0352 288.40815    7
 [6,] 448.29321 451.91234  62.44323 193.5370 290.31601    8
 [7,]  28.27688  81.85168  85.73746 239.9524 465.77042    2
 [8,] 340.13077 434.83098 292.85008 264.0453 294.64683    1
 [9,] 338.11235 432.15354 291.78182 265.6844 293.96262    1
[10,] 320.78748 479.70858 118.10918  80.8239  49.97897   10
```

```
kmeans_pp_NORM_10$taille
```

```
 [1] 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
```

```
kmeans_pp_NORM_10$potentiel / nrow(NORM_10)
```

```
[1] 5.045518
```

```
kmeans_pp_NORM_10$temps
```

```
[1] 114.0415
```

En appliquant la fonction des Kmeans++ qu'on a codée sur le dataset NORM_10 on s'assure qu'elle fonctionne bien, en effet elle arrive à trouver 10 centres assez éloignés les uns des autres et surtout à bien partitionner nos données en 10 clusters de 1000 points chacun.

Le potentiel pour l'itération effectuée est proche de 5 ce qui paraît cohérent avec les données. Malheureusement l'alogrithme met quasiment 2 minutes pour donner un résultat, c'est pourquoi l'utiliser sur NORM-10 et NORM-25 pour 3 nombres de clusters différents en effectuant à chaque fois 20 itérations ne paraît pas raisonnable ent terme de temps d'exécution.

On va donc préférer pour cela utiliser la fonction kmeans de R qui peut prendre en paramètre une liste de centres et lui donner donc en argument une liste de centre choisies suivant le protocole des Kmeans++.

Question 3 :

Fonctions de calcul de potentiel et de temps pour Kmeans :

```
tableau_resultat_kmeans <- function(data, K, nb_iter = 20)
{
  # on initialise les vecteurs potentiel et temps
  temps <- potentiel <- c()

  # on sauvegarde le nombre de point dans notre dataset
  n <- nrow(data)

  # on effectue nb_iter itérations sur nos données
  for (iter in 1:nb_iter)
  {
    # on stocke le temps de début d'exécution
    debut <- Sys.time()

    # on applique l'algorithme des kmeans
    kmeans <- kmeans(data, K)

    # on stocke à chaque fois le potentiel obtenu
    potentiel <- c(potentiel, kmeans$tot.withinss)

    # on stocke le temps de fin d'exécution
    fin <- Sys.time()

    # on calcule le temps d'exécution
    temps <- c(temps, as.numeric(difftime(fin, debut), units = "secs"))
  }

  mean_phi <- round(mean(potentiel),3)
  min_phi <- round(min(potentiel),3)
  mean_T <- round(mean(temps),3)

  return (t(setNames(data.frame(c(mean_phi/n, min_phi/n, mean_T),
                  row.names = c("Average $\\phi$", "Minimum $\\phi$", "Average $T$")),
                  "Kmeans")))
}
```

Fonctions de calcul de potentiel et de temps pour Kmeans++ :

```
tableau_resultat_kmeans_pp <- function(data, K, nb_iter = 20)
{
  # on initialise les vecteurs potentiel et temps
  temps <- potentiel <- c()

  # on sauvegarde le nombre de point dans notre dataset
  n <- nrow(data)

  # on effectue nb_iter itérations sur nos données
  for (iter in 1:nb_iter)
  {
    # on stocke le temps de début d'exécution
    debut <- Sys.time()

    # on applique l'algorithme des kmeans++
    kmeans <- opti_kmeans_pp(data, K)

    # on stocke à chaque fois le potentiel obtenu
    potentiel <- c(potentiel, kmeans$tot.withinss)

    # on stocke le temps de fin d'exécution
    fin <- Sys.time()

    # on calcule le temps d'exécution
    temps <- c(temps, as.numeric(difftime(fin, debut), units = "secs"))
  }

  mean_phi <- round(mean(potentiel),3)
  min_phi <- round(min(potentiel),3)
  mean_T <- round(mean(temps),3)

  return (t(setNames(data.frame(c(mean_phi/n, min_phi/n, mean_T),
                  row.names = c("Average $\\phi$", "Minimum $\\phi$", "Average $T$")),
                  "Kmeans++")))
}
```

Kmeans pour 10 clusters sur NORM-10 :

```
res_kmeans <- tableau_resultat_kmeans(NORM_10, 10)
```

Kmeans++ pour 10 clusters sur NORM-10 :

```
res_kmeans_pp <- tableau_resultat_kmeans_pp(NORM_10, 10)
```

```
df_10 <- rbind(res_kmeans, res_kmeans_pp)
```

Kmeans pour 25 clusters sur NORM-10 :

```
res_kmeans <- tableau_resultat_kmeans(NORM_10, 25)
```

Kmeans++ pour 25 clusters sur NORM-10 :

```
res_kmeans_pp <- tableau_resultat_kmeans_pp(NORM_10, 25)
```

```
df_25 <- rbind(res_kmeans, res_kmeans_pp)
```

Kmeans pour 50 clusters sur NORM-10 :

```
res_kmeans <- tableau_resultat_kmeans(NORM_10, 50)
```

Kmeans++ pour 50 clusters sur NORM-10 :

```
res_kmeans <- tableau_resultat_kmeans_pp(NORM_10, 50)
```

```
df_50 <- rbind(res_kmeans, res_kmeans_pp)
```

Comparaison Kmeans et Kmeans++ sur NORM-10 :

```
df_10 <- cbind(data.frame("Nb Clusters" = c(10,10)), df_10)
knitr::kable(df_10)
```

	Nb.Clusters	Average ϕ	Minimum ϕ	Average T
Kmeans	10	8659.016685	2804.367941	0.005
Kmeans++	10	5.045518	5.045518	1.286

```
df_25 <- cbind(data.frame("Nb Clusters" = c(25,25)), df_25)
knitr::kable(df_25)
```

	Nb.Clusters	Average ϕ	Minimum ϕ	Average T
Kmeans	25	294.881838	4.132085	0.013
Kmeans++	25	4.129035	4.098526	7.487

```
df_50 <- cbind(data.frame("Nb Clusters" = c(50,50)), df_50)
knitr::kable(df_50)
```

	Nb.Clusters	Average ϕ	Minimum ϕ	Average T
Kmeans	50	3.252201	3.232451	28.765
Kmeans++	50	4.129035	4.098526	7.487

On obtient des résultats très proches de ceux de la Table 1 du papier de recherche. On observe que Kmeans++ est systématiquement plus performant que Kmeans en atteignant une valeur potentielle plus faible en moyenne et en minimum. Cependant, Kmeans++ est légèrement plus lent que Kmeans, même s'il aide la recherche locale à converger après moins d'itérations.

Kmeans pour 10 clusters sur NORM-25 :

```
res_kmeans <- tableau_resultat_kmeans(NORM_25, 10)
```

Kmeans++ pour 10 clusters sur NORM-25 :

```
res_kmeans_pp <- tableau_resultat_kmeans_pp(NORM_25, 10)
```

```
df_10 <- rbind(res_kmeans, res_kmeans_pp)
```

Kmeans pour 25 clusters sur NORM-25 :

```
res_kmeans <- tableau_resultat_kmeans(NORM_25, 25)
```

Kmeans++ pour 25 clusters sur NORM-25 :

```
res_kmeans_pp <- tableau_resultat_kmeans_pp(NORM_25, 25)
```

```
df_25 <- rbind(res_kmeans, res_kmeans_pp)
```

Kmeans pour 50 clusters sur NORM-25 :

```
res_kmeans <- tableau_resultat_kmeans(NORM_25, 50)
```

Kmeans++ pour 50 clusters sur NORM-25 :

```
res_kmeans <- tableau_resultat_kmeans_pp(NORM_25, 50)
```

```
df_50 <- rbind(res_kmeans, res_kmeans_pp)
```

Comparaison Kmeans et Kmeans++ sur NORM-25 :

```
df_10 <- cbind(data.frame("Nb Clusters" = c(10,10)), df_10)
knitr::kable(df_10)
```

	Nb.Clusters	Average ϕ	Minimum ϕ	Average T
Kmeans	10	141935.4	123638.9	0.01
Kmeans++	10	116388.4	114297.0	1.46

```
df_25 <- cbind(data.frame("Nb Clusters" = c(25,25)), df_25)
knitr::kable(df_25)
```

	Nb.Clusters	Average ϕ	Minimum ϕ	Average T
Kmeans	25	49532.52914	26462.31060	0.015
Kmeans++	25	14.98365	14.98365	9.045

```
df_50 <- cbind(data.frame("Nb Clusters" = c(50,50)), df_50)
knitr::kable(df_50)
```

	Nb.Clusters	Average ϕ	Minimum ϕ	Average T
Kmeans	50	14.14140	14.12959	32.896
Kmeans++	50	14.98365	14.98365	9.045

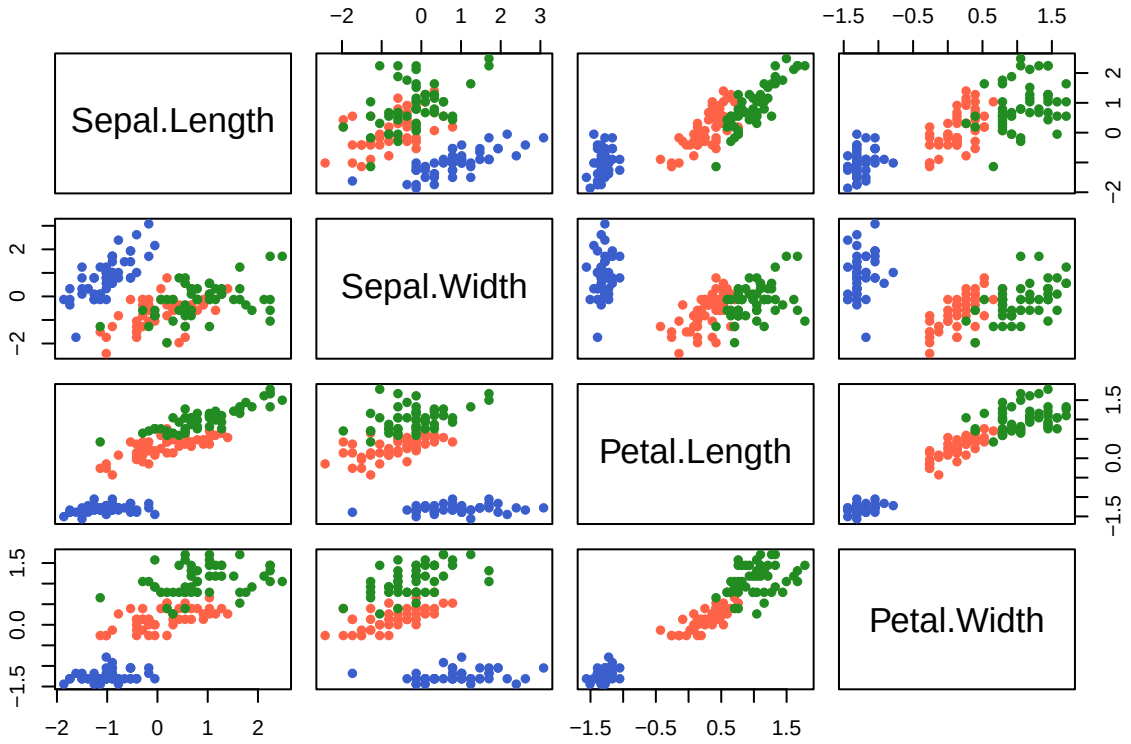
On obtient des résultats très proches de ceux de la Table 2 du papier de recherche. On observe que Kmeans++ est systématiquement plus performant que Kmeans en atteignant une valeur potentielle plus faible en moyenne et en minimum. Cependant, Kmeans++ est légèrement plus lent que Kmeans, même s'il aide la recherche locale à converger après moins d'itérations.

Exercice 2 - Données iris :

Question 1 :

Dataset Iris :

```
data(iris)
iris_quant <- scale(iris[,1:4])
n <- nrow(iris_quant)
pairs(iris_quant, col = c("royalblue3","tomato1","forestgreen")[iris$Species], pch = 16)
```



Le dataset comprend 50 échantillons de chacune des trois espèces d'iris (Iris Setosa, Iris Virginica et Iris Versicolor). Quatre caractéristiques ont été mesurées à partir de chaque échantillon : la longueur et la largeur des sépales et des pétales, en centimètres.

On observe dans le plot au-dessus que les 3 espèces sont assez visuellement séparées : par exemple pour les variables Petal.Length et Petal.Width en dessous de la valeur -0.5 on ne retrouve que des points bleus.

Kmeans sur Iris :

```
temps <- potentiel <- c()
for(iter in 1:20)
{
  debut <- Sys.time()
  kmeans_cluster <- kmeans(iris_quant, 3)
  fin <- Sys.time()
  potentiel <- c(potentiel, kmeans_cluster$tot.withinss)
  temps <- c(temps, as.numeric(difftime(fin, debut), units = "secs"))
}
metrics <- c(round(mean(potentiel)/n,3), round(min(potentiel)/n,3), round(mean(temps),3))
res_kmeans <- t(setNames(data.frame(metrics,
        row.names = c("Average $\\phi$", "Minimum $\\phi$", "Average $T$")),
        "Kmeans"))
```

Mclust sur Iris :

```
temps <- potentiel <- c()
for(iter in 1:20)
{
  debut <- Sys.time()
  mclust_cluster <- Mclust(iris_quant, 3)
  fin <- Sys.time()
  potentiel <- c(potentiel,NA)
  temps <- c(temps,as.numeric(difftime(fin,debut),units="secs"))
}
metrics <- c(round(mean(potentiel)/n,3), round(min(potentiel)/n,3), round(mean(temps),3))
res_mclust <- t(setNames(data.frame(metrics,
        row.names = c("Average $\\phi$", "Minimum $\\phi$", "Average $T$")),
        "Mclust"))
```

Kmeans++ sur Iris :

```
temps <- potentiel <- c()
for(iter in 1:20)
{
  debut <- Sys.time()
  kmeans_pp_cluster <- opti_kmeans_pp(iris_quant, 3)
  fin <- Sys.time()
  potentiel <- c(potentiel, kmeans_pp_cluster$tot.withinss)
  temps <- c(temps, as.numeric(difftime(fin, debut), units = "secs"))
}
metrics <- c(round(mean(potentiel)/n,3), round(min(potentiel)/n,3), round(mean(temps),3))
res_kmeans_pp <- t(setNames(data.frame(metrics,
                  row.names = c("Average $\\phi$","Minimum $\\phi$","Average $T$")),
                  "Kmeans++"))
```

Comparaison Kmeans, Mclust et Kmeans++ sur Iris :

```
res_iris <- rbind(res_kmeans, res_mclust, res_kmeans_pp)
knitr::kable(res_iris)
```

	Average ϕ	Minimum ϕ	Average T
Kmeans	1.015	0.926	0.000
Mclust	NA	NA	0.087
Kmeans++	1.013	0.926	0.006

On remarque d'après la table au-dessus que l'algorithme Mclust est beaucoup plus lent que Kmeans et Kmeans++. Concernant les potentiels, Kmeans et Kmeans++ obtiennent des résultats similaires en terme de moyenne et de minimum, ce qui indique qu'ils trouvent quasiment les mêmes clusters.

La notion de potentiel n'existe pas pour Mclust puisqu'ils ne calculent pas des centres mais cherche plutôt à estimater des paramètres via l'algorithme EM pour les modèles de mélange normal avec une variété de structures de covariance, et des fonctions pour la simulation à partir de ces modèles.

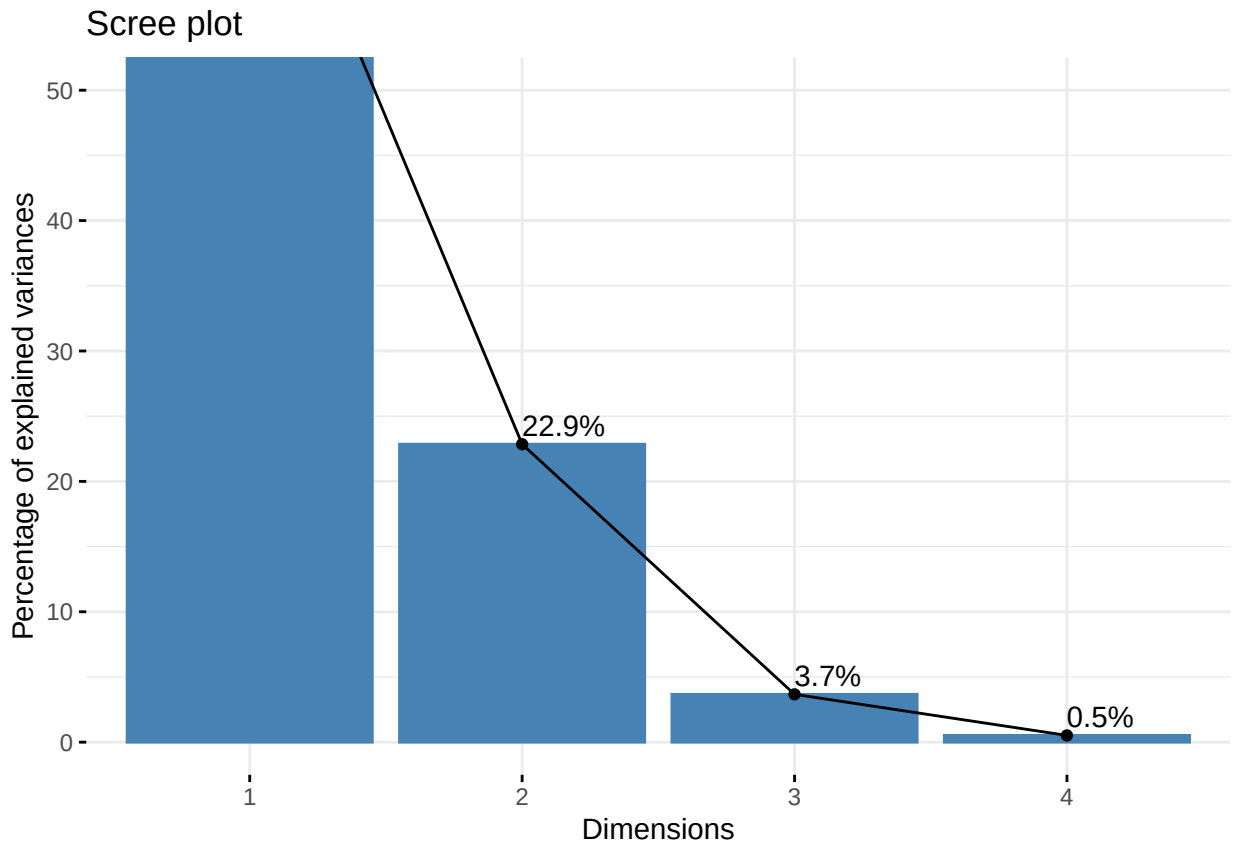
Question 2 :

PCA sur Iris :

```
pca_iris <- PCA(iris_quant, scale.unit = TRUE)
```

Diagramme des éboulis :

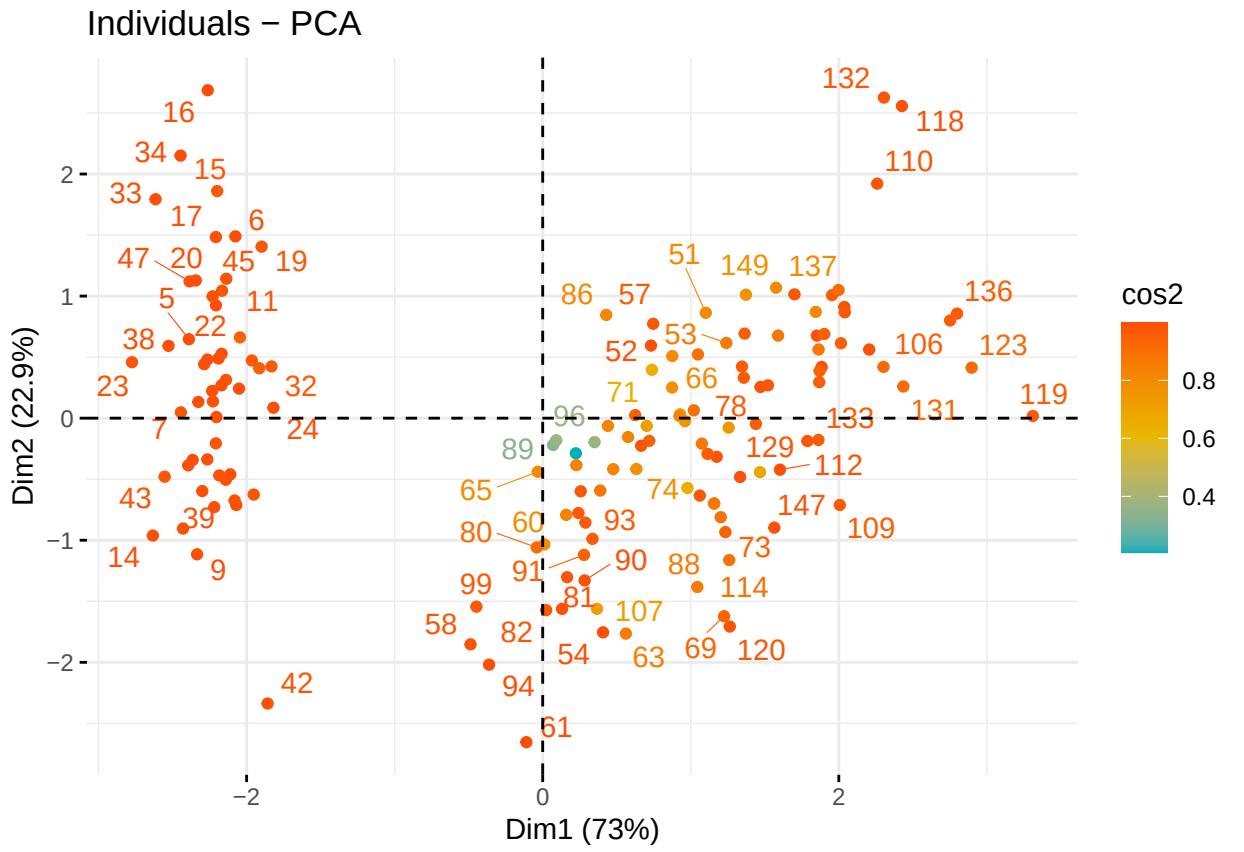
```
fviz_eig(pca_iris, addlabels = TRUE, ylim = c(0, 50))
```



On peut dire d'après le graphe ci-dessus que le nombre de dimension adéquate pour expliquer au mieux nos données est 2 puisqu'à partir de la 3-ème dimension on chute à moins de 4% de variance expliquée. On peut également voir que la dimension 1 explique environ 73% de nos variances tandis que la dimension 2 en explique environ 23%.

Graphe des individus :

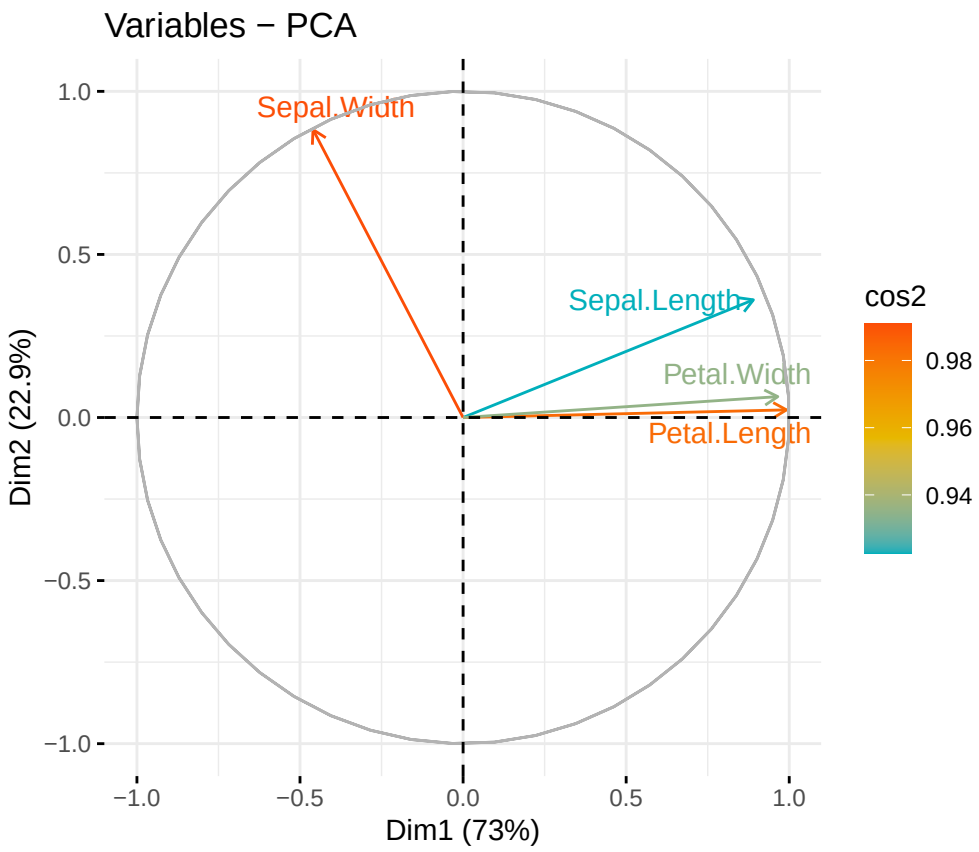
```
fviz_pca_ind(pca_iris, col.ind = "cos2", addlables = TRUE,
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"), repel = TRUE)
```



On remarque sur les graphes des individus qu'une espèce se distingue des autres suivant la dimension 1, tandis que la distinction entre les 2 espèces restantes est beaucoup plus difficile à visualiser.

Graphe des variables :

```
fviz_pca_var(pca_iris, col.var = "cos2", addlables = TRUE,
             gradient.cols = c("#00AFBB","#E7B800","#FC4E07"), repel = TRUE)
```

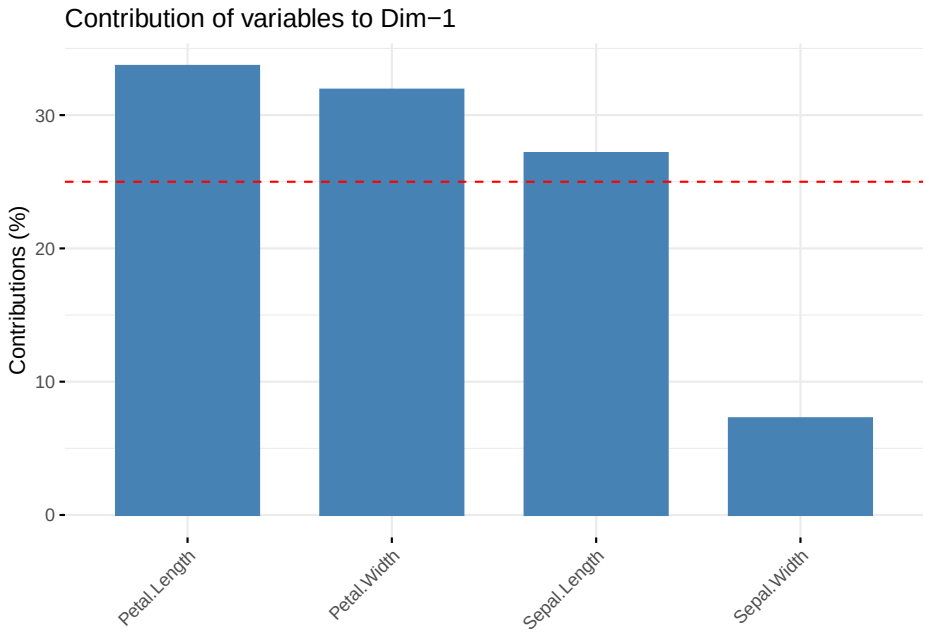


On remarque sur les graphes des variables que certaines sont assez corrélées notamment Petal.Width et Petal.Length de plus elles sont bien expliquées par la dimension 1. Contrairement à Sepal.Width qui elle est plus expliquée par la dimension 2.

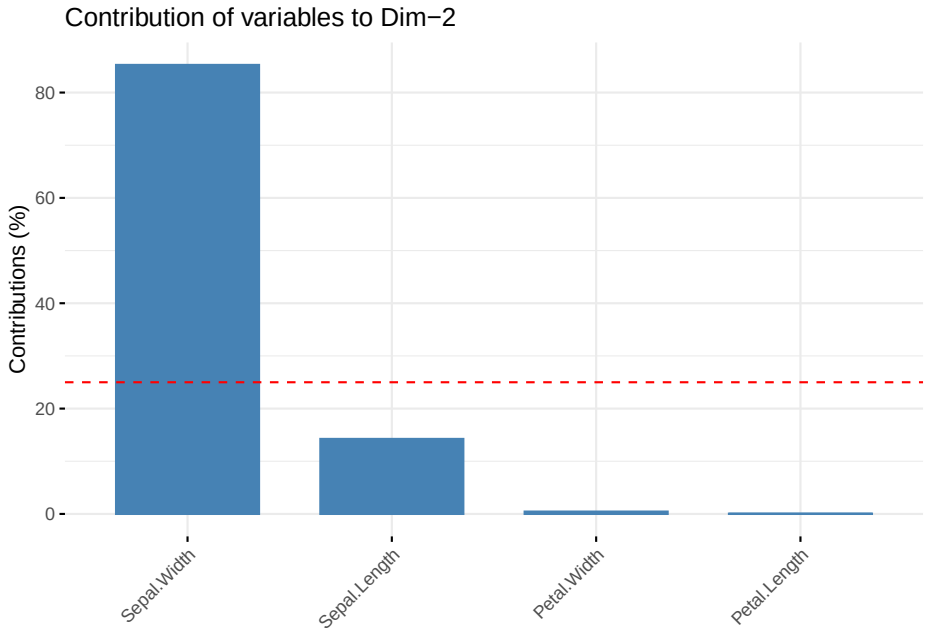
Graphe des contributions :

On peut ici voir les taux de contributions de chacune de nos variables pour les dimensions 1 et 2 :

```
fviz_contrib(pca_iris, choice = "var", axes = 1, top = 10)
```

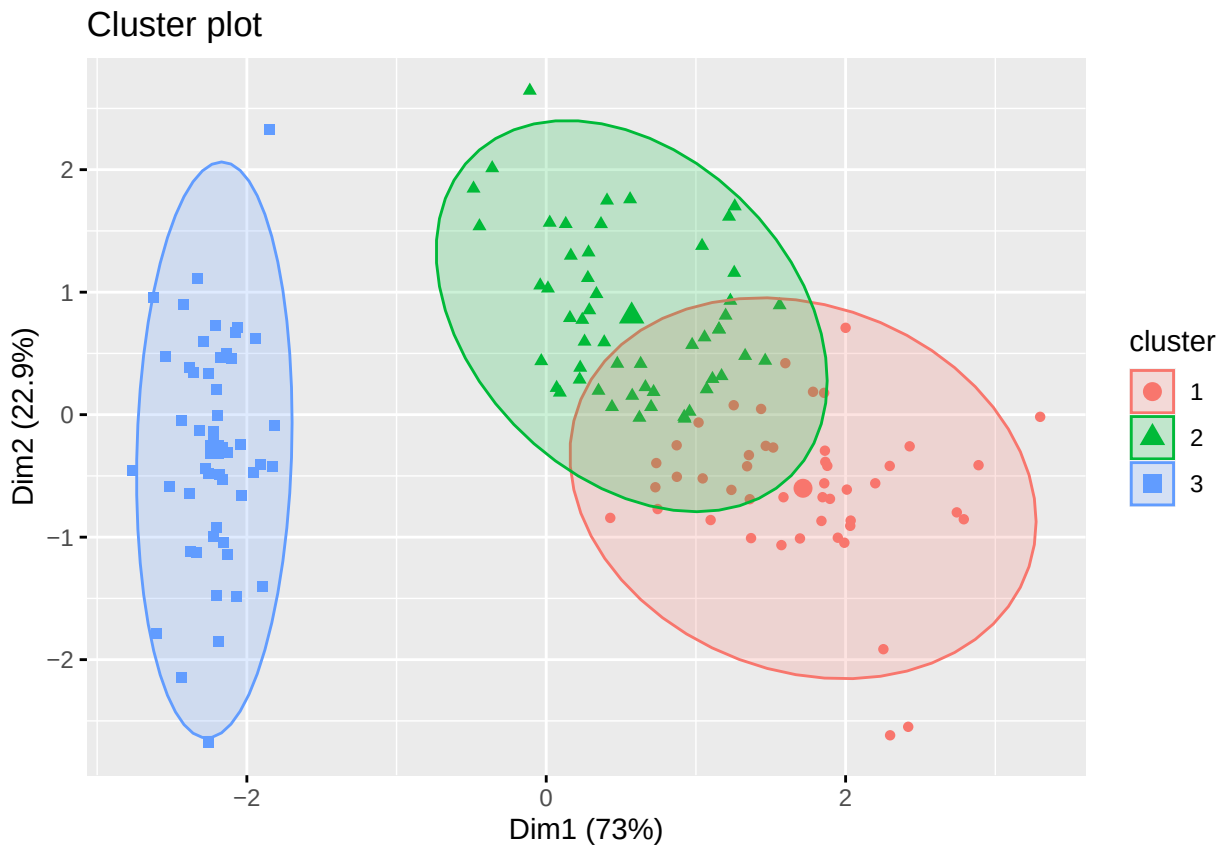



```
fviz_contrib(pca_iris, choice = "var", axes = 2, top = 10)
```



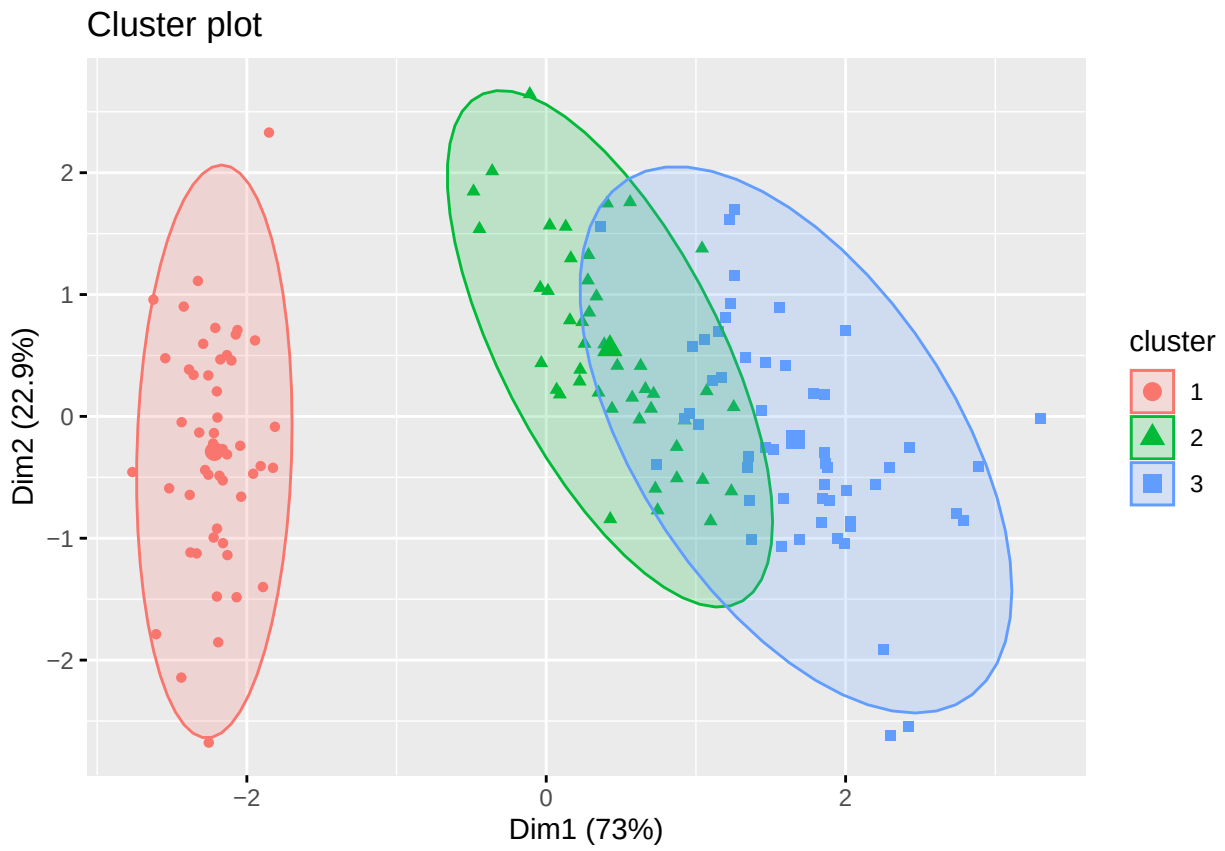
PCA sur les clusters de Kmeans pour Iris :

```
fviz_cluster(kmeans_cluster, data = iris_quant, geom = c("point"), ellipse.type = "norm")
```



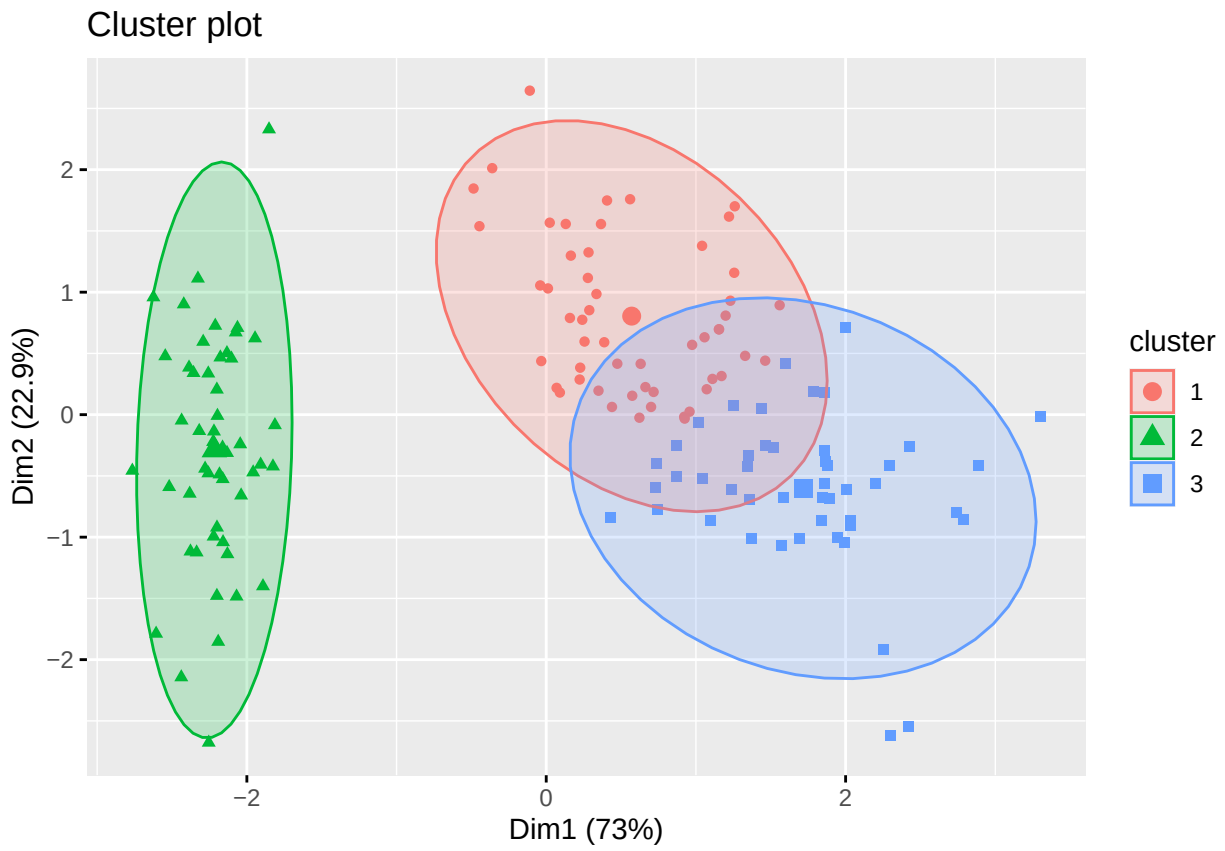
PCA sur les clusters de Mclust pour Iris :

```
fviz_cluster(mclust_cluster, data = iris_quant, geom = c("point"), ellipse.type = "norm")
```



PCA sur les clusters de Kmeans++ pour Iris :

```
fviz_cluster(kmeans_pp_cluster, data = iris_quant, geom = c("point"), ellipse.type = "norm")
```



Question 3 :

D'après la visualisation des clusters de Kmeans, on remarque que la pertinence des clusters obtenus à la fin de l'algorithme dépend grandement des centres choisis aléatoirement au début, ce qui rend donc la pertinence des résultats instables.

Tandis que pour Kmeans++, vu qu'on initialise les centres de manière largement moins aléatoire, on retrouve beaucoup plus facilement un résultat cohérent avec un bon clustering.

Le Mclust quant à lui obtient 3 clusters beaucoup plus pertinents que ceux du Kmeans. Sûrement grâce à l'initialisation de l'algorithme EM effectuée en utilisant les partitions obtenues à partir du clustering hiérarchique agglomératif.

Depuis le début, nous avons étudié une amélioration de l'algorithme des Kmeans. Il apparaît au final assez clairement d'après nos applications sur les données simulées NORM-10 et NORM-25 ainsi que sur le dataset Iris que l'algorithme des Kmeans++ dépasse les algorithmes Kmeans et Mclust en termes de précision et de pertinence des résultats.

On pourrait cependant imaginer encore des améliorations de Kmeans++ en sélectionnant plusieurs nouveaux centres à chaque itération en fonction de celui qui diminue le plus possible le potentiel phi.

D'autres alternatives à l'algorithme des Kmeans existent également. Parmi elles ont peu cité les Kmédoïds qui contrairement à l'algorithme Kmeans, Kmédoïds choisit des points de données réels comme centres et permet ainsi une plus grande interprétabilité des centres de cluster que dans Kmeans, où le centre d'un cluster n'est pas nécessairement l'un des points des données.