Rapport Projet

Léos Coutrot

13-11-2023

Contents

Projection orthogonale

Considérons une droite D définie par un point A et un vecteur directeur $\overrightarrow{v}$ dans R^p, et un point x également dans R^p. Ecrivez les formules explicites et le code informatique associé qui donne :\ • les coordonnées de la projection orthogonale de x sur D\ • la distance entre x et D.

Les coordonnées de la projection orthogonale de x sur D, notée $\text{proj}_D(x)$, sont données par :

$$\text{proj}_D(x) = A + \frac{(x - A) \cdot \overrightarrow{v}}{\| \overrightarrow{v} \|^2} \cdot \overrightarrow{v}$$

Et on définit la distance entre x et D, notée $d(x, D)$, par :

$$d(x, D) = \| x - \text{proj}_D(x) \|$$

Codons en R la projection d'un point sur une droite. De plus, soit K droites $D_1, ..., D_K$ et un point x dans R^p. Ecrivons le code qui détermine la droite la plus proche de x. En cas d'égalité de distance un tirage aléatoire sera utilisé.

```r
# Fonction pour représenter une droite
creer_droite <- function(a, u) {
  list(a = a, u = u / sqrt(sum(u^2)))
}

# Fonction pour calculer la projection d'un point sur une droite
projection_sur_droite <- function(x, droite) {
  a <- droite$a
  u <- droite$u
  return(a + sum((x - a) * u) * u ) # Calcul de la projection
}

distance_droite <- function(x, d) {
  proj <- projection_sur_droite(x, d)
  return(sqrt(sum((x - proj)^2)))
}

# Fonction pour trouver la droite la plus proche parmi une liste de droites
droite_la_plus_proche <- function(x, droites) {
  distances <- numeric(length(droites))
  for(i in 1:length(droites)) {
    distances[i] <- distance_droite(x, droites[[i]])
  }
  indices_minimaux <- which(distances == min(distances))
  if (length(indices_minimaux) > 1) {
    index_plus_proche <- sample(indices_minimaux, 1)
  } else {
    index_plus_proche <- indices_minimaux
  }

  return(index_plus_proche)  # Retourne la droite la plus proche de x
}
```

```r
# Exemple
d1 <- creer_droite(c(100, 2), c(1, 0))
d2 <- creer_droite(c(2, 3), c(0, 1))
droites <- list(d1, d2)
x <- c(2, 3)

droite_proche <- droite_la_plus_proche(x, droites)
droite_proche
```

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

Kmeans Généralisés

L'algorithme des kmeans généralisés est une variante de l'algorithme de clustering kmeans classique. Il est particulièrement adapté pour gérer des clusters non sphériques et des structures de données plus complexes.

Description de l'Algorithme

Dans l'algorithme des kmeans généralisés, la notion de centre de cluster est remplacée par celle de représentant de classe, qui peut être un point, une droite, un plan, ou toute autre structure géométrique adaptée. L'algorithme vise à minimiser un critère de coût défini comme la somme des distances entre les points de données et le représentant le plus proche de leur classe.

Processus de l'Algorithme

1. **Initialisation :** Choisir aléatoirement les représentants initiaux pour chaque cluster.
2. **Affectation :** Assigner chaque point de données au cluster dont le représentant est le plus proche, en fonction d'une mesure de distance spécifiée.
3. **Mise à jour :** Mettre à jour les représentants de chaque cluster pour minimiser la fonction de coût. Cette étape dépend de la nature des représentants (points, droites, etc.).
4. **Itération :** Répéter les étapes d'affectation et de mise à jour jusqu'à ce que la convergence soit atteinte, c'est-à-dire lorsque les changements dans les associations de cluster ou les représentants deviennent négligeables.

Critère de Coût

La fonction de coût J est définie comme suit :

$$J(C, D) = \sum_{i=1}^{N} \sum_{k=1}^{K} c_{ik} d^2(x_i, D_k)$$

où :

- N est le nombre total de points.
- K est le nombre de clusters.
- $C = (c_{ik})$ est la matrice de classification indiquant l'appartenance des points aux clusters.

- $D = \{D_1, \ldots, D_K\}$ est l'ensemble des représentants des clusters.
- $d^2(x_i, D_k)$ est la distance au carré entre le point x_i et le représentant D_k.

```
generer_droites_aleatoires <- function(K, data) {
  droites <- list()
  n <- nrow(data)

  for (i in 1:K) {
    indices <- sample(1:n, 2)  # Sélectionner aléatoirement deux points
    point_a <- data[indices[1], ]
    point_b <- data[indices[2], ]
    vecteur_directeur <- point_b - point_a
    droites[[i]] <- creer_droite(point_a, vecteur_directeur)
  }

  return(droites)
}


C_update <- function(matrice_points, liste_droites) {
  nb_points <- nrow(matrice_points)
  classification <- numeric(nb_points)

  for (i in 1:nb_points) {
    point <- matrice_points[i, ]
    classification[i] <- droite_la_plus_proche(point, liste_droites)
  }

  return(classification)
}

# Fonction pour calculer la somme des distances au carré d'un ensemble de points à une droite
sumOfSquares <- function(droite_params, points) {
  a <- droite_params[1:2]  # Le point sur la droite
  u <- droite_params[3:4]  # Le vecteur directeur de la droite
  u <- u / sqrt(sum(u^2))  # Normaliser le vecteur directeur
  droite <- list(a = a, u = u)

  sum_squares <- 0
  for (i in 1:nrow(points)) {
    distance <- distance_droite(points[i, ], droite)
    sum_squares <- sum_squares + distance^2
  }

  return(sum_squares)
}

# Fonction pour mettre à jour une seule droite
mettreAJourUneDroite <- function(points) {
  init_params <- runif(4)  # Paramètres initiaux aléatoires pour la droite
  optim_result <- optim(init_params, sumOfSquares, points = points)
  return(optim_result$par)
}
```

```r
# Fonction pour mettre à jour toutes les droites
D_update <- function(Donnees, C, K) {
  droites <- list()
  for (k in 1:K) {
    points_droite <- Donnees[C == k, ]
    if (nrow(points_droite) > 0) {
      params_droite <- mettreAJourUneDroite(points_droite)  # Récupérer les paramètres de la droite
      a <- params_droite[1:2]
      u <- params_droite[3:4]
      droites[[k]] <- creer_droite(a, u)
    }
  }
  return(droites)
}
```

```r
kmeans_gen <- function(data, K, max_iter = 100) {
    n <- nrow(data)  # Nombre de points
    droites <- generer_droites_aleatoires(K, data)
    classifications <- numeric(n)
    for (iter in 1:max_iter) {
        previous_classes <- classifications
        classifications <- C_update(data, droites)
        if (all(previous_classes == classifications)) {
            return(list(droites = droites, classifications = classifications))
        }
        droites<-D_update(data,classifications,K)
    }

    return(list(droites = droites, classifications = classifications))
}
```

Jeu de données iris

On va tout d'abord charger le jeu de données iris

```r
data(iris)
iris_data<-iris[,c("Sepal.Length","Sepal.Width","Petal.Length","Petal.Width")]
```

Avec notre algorithme des nuées dynamiques

```r
set.seed(222)
res_gen <- kmeans_gen(iris_data, K = 3)
```

Avec l'algorithme des kmeans de R

```r
set.seed(222)
res_km <- kmeans(iris_data, centers = 3)
```

Avec le modèle de mélange gaussien

```
res_mclust <- Mclust(iris_data, G=3, verbose = FALSE)
```

Comparaisons des résultats

```
iris_labels <- as.numeric(factor(iris$Species))

confusion_matrix_kgen <- table(Predicted = res_gen$classifications, True = iris_labels)
confusion_matrix_kgen[, c(1, 3)] <- confusion_matrix_kgen[, c(3, 1)]
confusion_matrix_kmeans <- table(Cluster = res_km$cluster, TrueLabels = iris_labels)
confusion_matrix_kgen[, c(1, 3)] <- confusion_matrix_kgen[, c(3, 1)]
confusion_matrix_mclust <- table(Cluster = res_mclust$classification, TrueLabels = iris_labels)

# Afficher les matrices de confusion
print("Matrice de confusion pour les K-means généralisées")
```

```
## [1] "Matrice de confusion pour les K-means généralisées"
```

```
print(confusion_matrix_kgen)
```

```
##          True
## Predicted  1  2  3
##         1  0  3 35
##         2 50  4  0
##         3  0 43 15
```

```
print("Matrice de confusion pour l'algorithme des K-means")
```

```
## [1] "Matrice de confusion pour l'algorithme des K-means"
```

```
print(confusion_matrix_kmeans)
```

```
##        TrueLabels
## Cluster  1  2  3
##       1  0  2 36
##       2 50  0  0
##       3  0 48 14
```

```
print("Matrice de confusion pour l'algorithme des mélanges de gaussiennes (Mclust)")
```

```
## [1] "Matrice de confusion pour l'algorithme des mélanges de gaussiennes (Mclust)"
```

```
print(confusion_matrix_mclust)
```

```
##        TrueLabels
## Cluster  1  2  3
##       1 50  0  0
##       2  0 45  0
##       3  0  5 50
```

Ici l'algorithme le plus efficace est celui utilisant mclust (algorithme aux mélange des gaussiennes). L'algorithme des kmeans généralisés est le moins performant de tout ns modèles, mais il n'est pas cependant particulièrement mauvais.

```
precision_kmeangen <- sum(diag(confusion_matrix_kgen)) / sum(confusion_matrix_kgen)
precision_kmeans <- sum(diag(confusion_matrix_kmeans)) / sum(confusion_matrix_kmeans)
precision_mclust <- sum(diag(confusion_matrix_mclust)) / sum(confusion_matrix_mclust)

cat("Précision des nuées dynamiques:", precision_kmeangen)
```

```
## Précision des nuées dynamiques: 0.1266667
```

```
cat("Précision des K-means:", precision_kmeans)
```

```
## Précision des K-means: 0.09333333
```

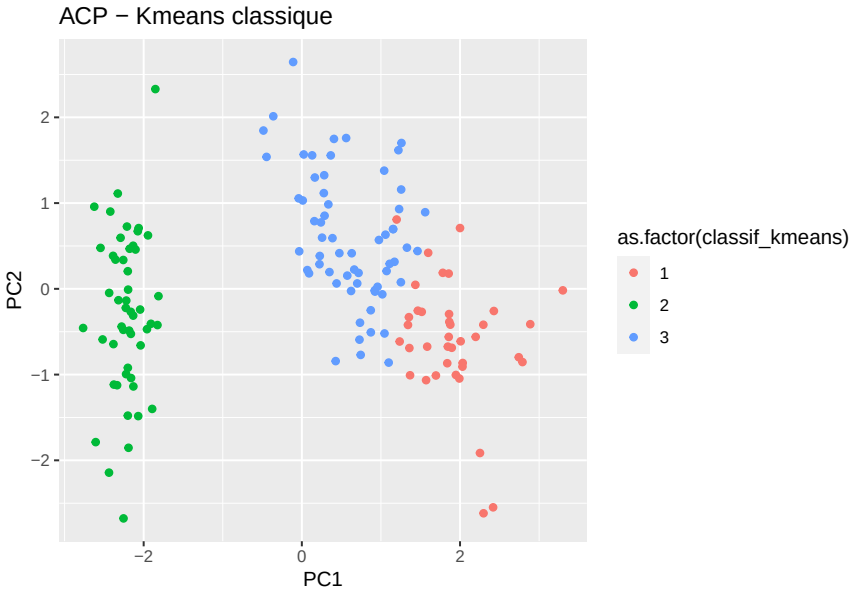
```
cat("Précision des Mclust:", precision_mclust)
```

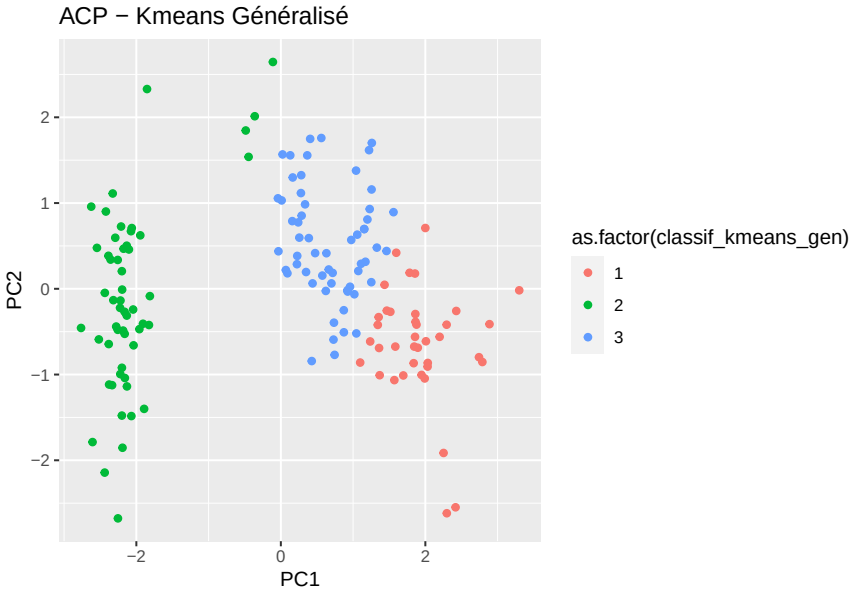
```
## Précision des Mclust: 0.9666667
```

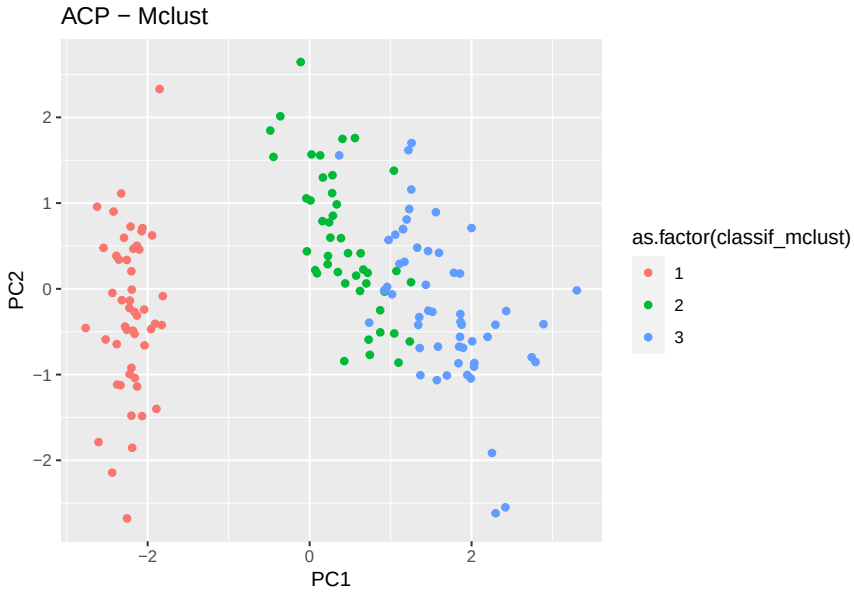
ACP

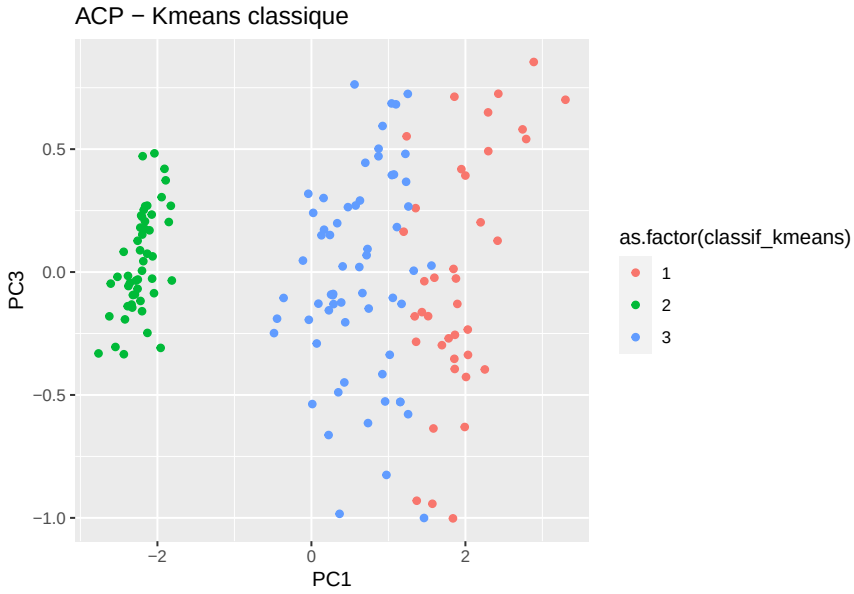
On va effectuer une ACP sur notre jeu de données

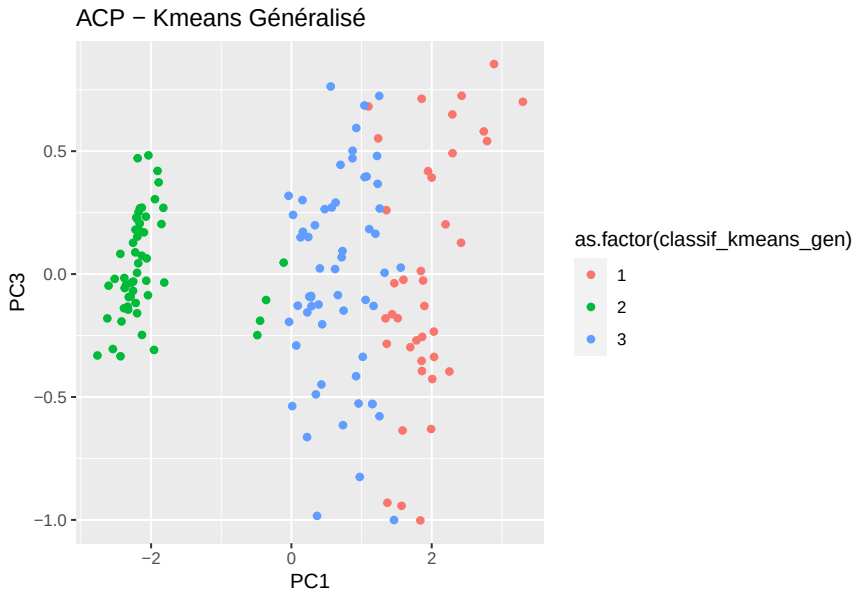
```
pca_res <- prcomp(iris_data, scale. = TRUE)

# Créer un dataframe pour la visualisation
pca_data <- data.frame(pca_res$x)

# Ajouter les classifications à pca_data
pca_data$classif_kmeans <- res_km$cluster
pca_data$classif_kmeans_gen <- res_gen$classifications
pca_data$classif_mclust <- res_mclust$classification
```

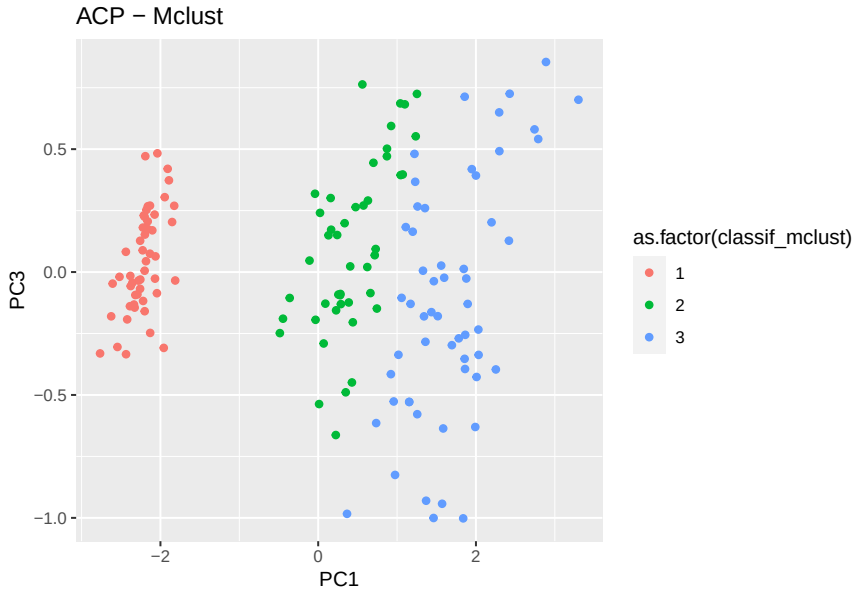

ACP − Kmeans classique
PC2
PC1
2
1
0
−1
−2
−2
0
2
as.factor(classif_kmeans)
1
2
3

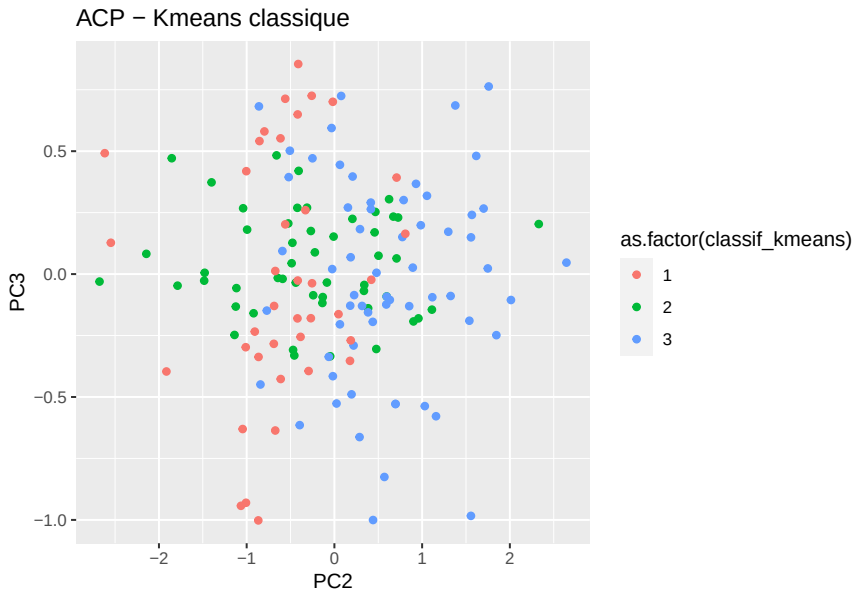

ACP − Kmeans Généralisé
PC2
PC1
2
1
0
−1
−2
−2
0
2
as.factor(classif_kmeans_gen)
1
2
3

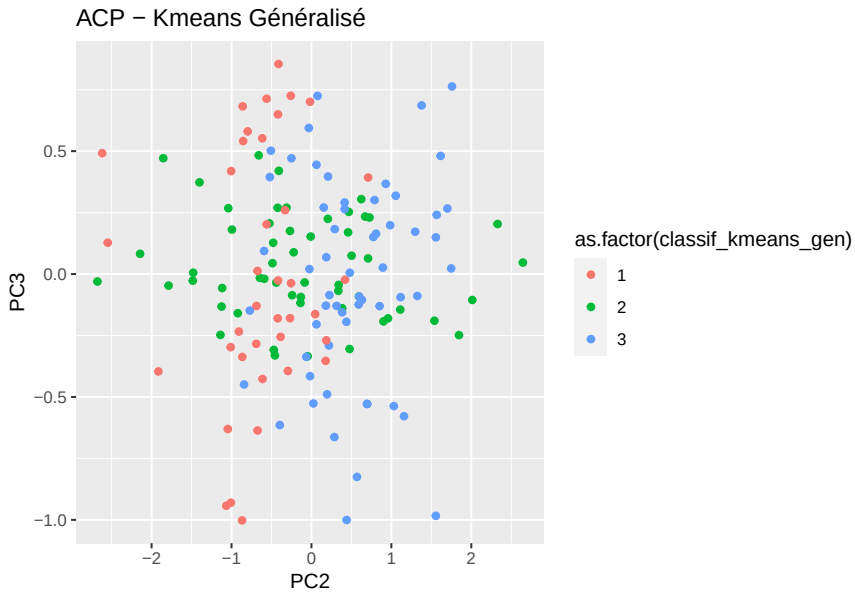

ACP − Mclust
PC2
PC1
2
1
0
−1
−2
−2
0
2
as.factor(classif_mclust)
1
2
3

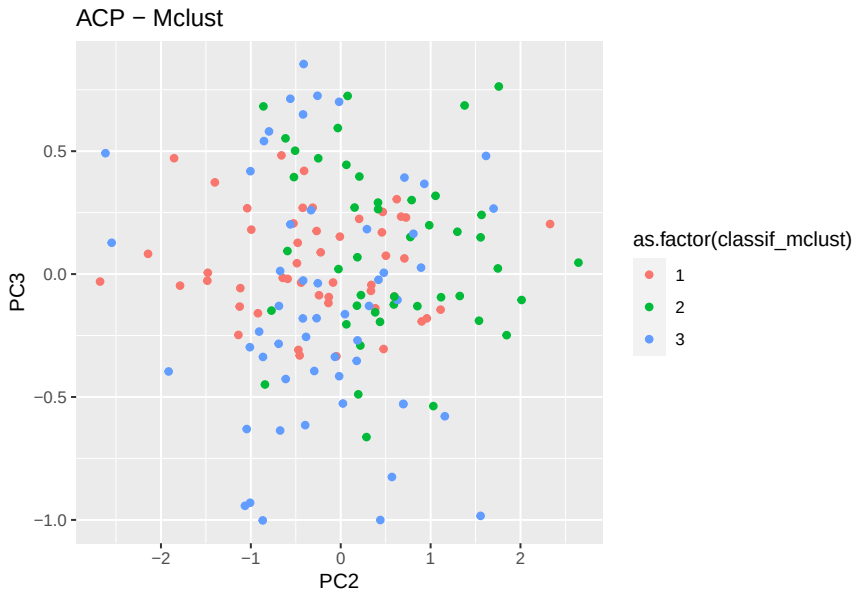

ACP − Kmeans classique
PC3
PC1
0.5
0.0
−0.5
−1.0
−2
0
2
as.factor(classif_kmeans)
1
2
3


ACP − Kmeans Généralisé
0.5
0.0
−0.5
−1.0
PC3
−2
0
2
PC1
as.factor(classif_kmeans_gen)
1
2
3


ACP − Mclust
0.5
0.0
−0.5
−1.0
PC3
−2
0
2
PC1
as.factor(classif_mclust)
1
2
3


ACP − Kmeans classique
PC3
PC2
0.5
0.0
−0.5
−1.0
−2
−1
0
1
2
as.factor(classif_kmeans)
1
2
3


ACP − Kmeans Généralisé
PC3
PC2
0.5
0.0
−0.5
−1.0
−2
−1
0
1
2
as.factor(classif_kmeans_gen)
1
2
3



Graphiquement, on voit encore que les kmeans généralisés sont moins performants que les deux autres méthodes présentés.

Simulation de Données pour Kmeans Généralisés

Pour évaluer l'efficacité des kmeans généralisés, il est utile de simuler des jeux de données qui ne sont pas bien adaptés aux méthodes de clustering classiques telles que les kmeans standards. Une approche consiste à simuler des données qui se répartissent le long de droites dans R^2, ce qui présente un défi pour les méthodes de clustering basées sur la distance euclidienne centrée.

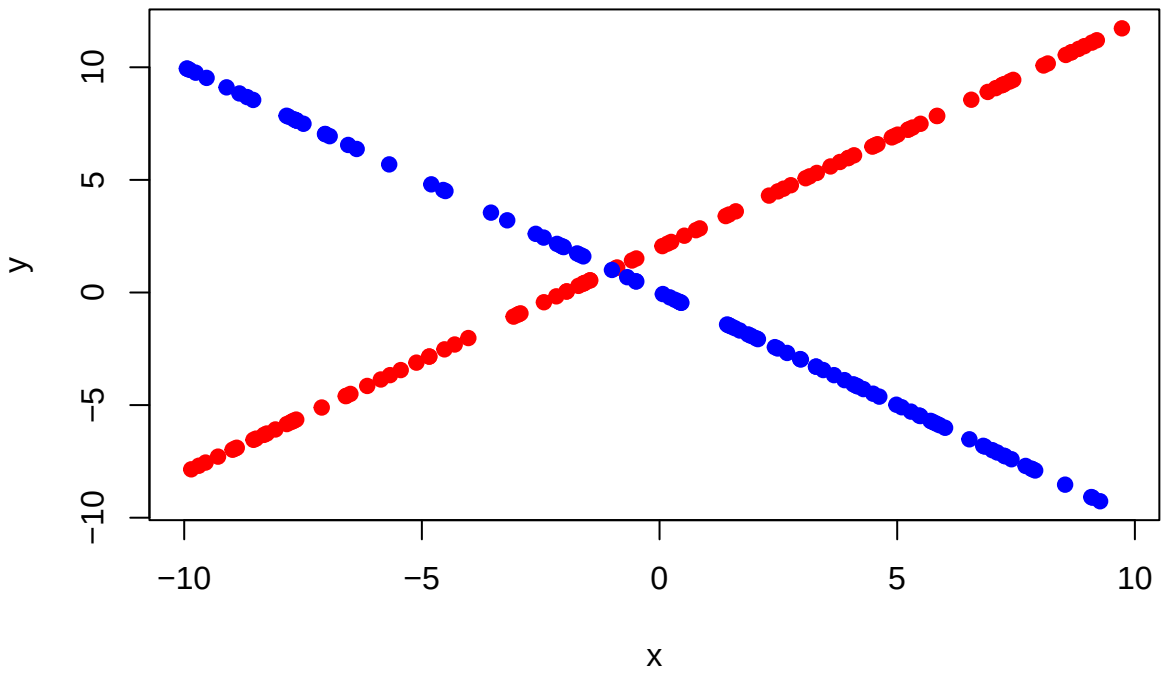
Paramètres des Droites

Considérons deux droites dans R^2 définies par les équations suivantes :

- Première droite : $y = a_1 \cdot x + b_1$
- Deuxième droite : $y = a_2 \cdot x + b_2$

où a_1, a_2 sont les pentes et b_1, b_2 sont les ordonnées à l'origine des droites respectivement. Par exemple, nous pouvons définir :

- Pour la première droite : $a_1 = 1$, $b_1 = 2$
- Pour la deuxième droite : $a_2 = -1$, $b_2 = 0$

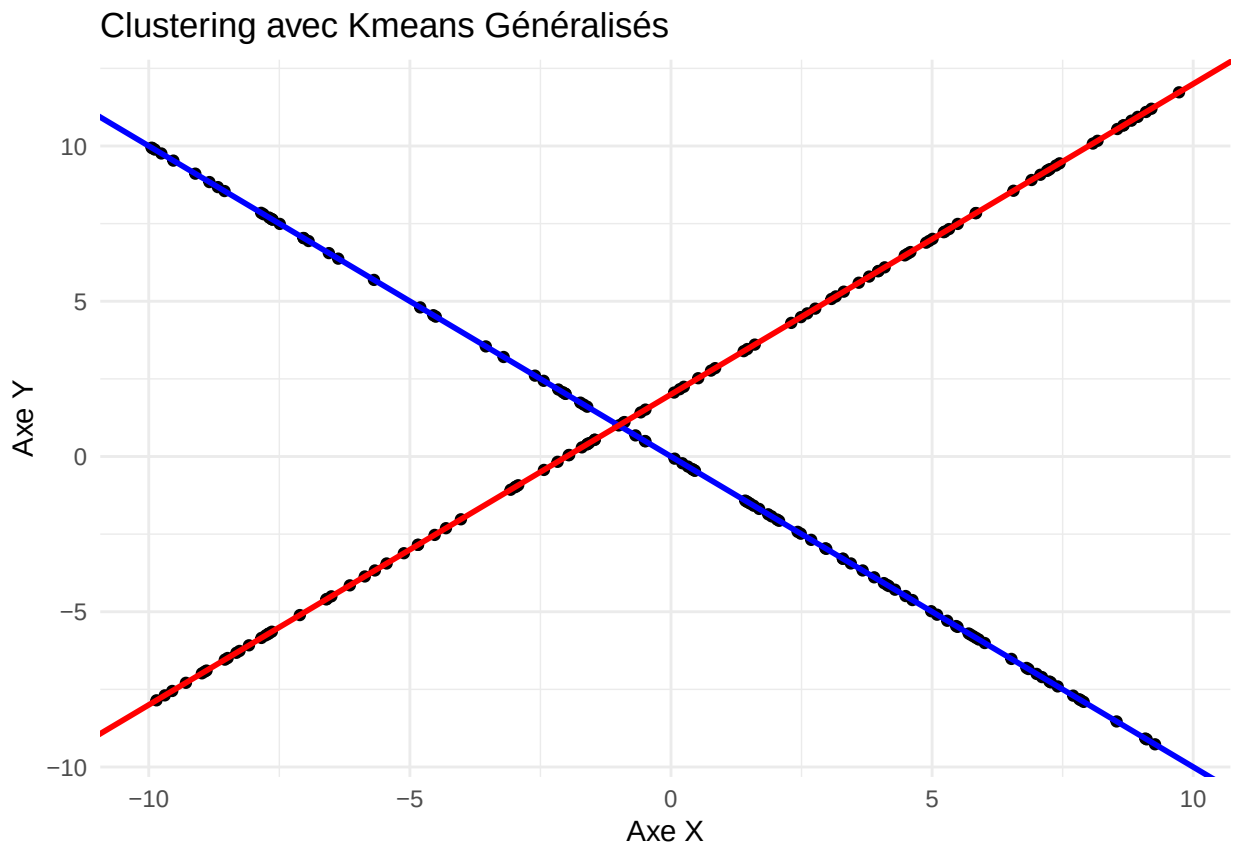


On lance l'algorithme :

```
set.seed(222)
res_gen<-kmeans_gen(data,2)
```

```
droites <- res_gen$droites

# Tracer les points de données
ggplot(data, aes(x, y)) +
  geom_point() +
  theme_minimal() +
  # Ajouter les droites de clustering
  geom_abline(slope = droites[[1]]$u[2] / droites[[1]]$u[1],
              intercept = droites[[1]]$a[2] - (droites[[1]]$a[1] * droites[[1]]$u[2] / droites[[1]]$u[1
              color = "blue", size = 1) +
  geom_abline(slope = droites[[2]]$u[2] / droites[[2]]$u[1],
              intercept = droites[[2]]$a[2] - (droites[[2]]$a[1] * droites[[2]]$u[2] / droites[[2]]$u[1
              color = "red", size = 1) +
  # Titre et labels
  labs(title = "Clustering avec Kmeans Généralisés", x = "Axe X", y = "Axe Y")
```

```
## Warning: Using 'size' aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use 'linewidth' instead.
## This warning is displayed once every 8 hours.
## Call 'lifecycle::last_lifecycle_warnings()' to see where this warning was
## generated.
```



On voit que notre algorithme est ici particulièrement performant (il faut tout de même noter que le jeu de données est “assez simple”).

Comparaison au kmeans

```
res_km <- kmeans(data, centers = 2)
```

```
data$classif_reelle <- c(rep(1, n), rep(2, n))
confusion_matrix_kgen <- table(Predicted = res_gen$classifications, True = data$classif_reelle)
confusion_matrix_kmeans <- table(Cluster = res_km$cluster, TrueLabels = data$classif_reelle)

print("Matrice de confusion pour les K-means généralisées")
```

```
## [1] "Matrice de confusion pour les K-means généralisées"
```

```
print(confusion_matrix_kgen)
```

```
##           True
## Predicted   1   2
##         1   0 100
##         2 100   0
```

```r
print("Matrice de confusion pour l'algorithme des K-means")
```

```
## [1] "Matrice de confusion pour l'algorithme des K-means"
```

```r
print(confusion_matrix_kmeans)
```

```
##         TrueLabels
## Cluster   1   2
##       1 100  74
##       2   0  26
```

On voit bien que notre algorithme est ici beaucoup plus performant que l'algorithme des kmeans classiques.

Question subsidiaire

Je n'ai pas pu finir cette partie, cependant voici le code permettant de projeter une matrice de point sur un sous espace engendré par une matrice Z. Bien que cela soit incomplet, cela serait un bon point de départ pour implémenter notre algorithme dans un espace où les représentants de classes sont des hyperplans.

```r
projection_orthogonale <- function(X, Z) {
  M=Z%*%solve((t(Z)%*%Z))%*%t(Z)
  return(M%*%X)
}
```

Conclusion

En résumé, ce projet a exploré l'efficacité et l'applicabilité des kmeans généralisés, une méthode de clustering avancée capable de gérer des structures de données complexes et non sphériques. À travers la simulation de données linéaires en (R^2) et l'application de l'algorithme sur ces jeux de données, nous avons démontré que les kmeans généralisés offrent une flexibilité et une performance supérieures par rapport aux méthodes de clustering traditionnelles dans certaines situations.

Les kmeans généralisés se sont avérés particulièrement adaptés pour identifier des clusters linéaires ou de forme irrégulière, là où les kmeans classiques échouent souvent. Cela a été illustré par la capacité de l'algorithme à regrouper correctement des points le long de droites spécifiques dans l'espace bidimensionnel.

Perspectives Futures

Bien que les résultats actuels soient prometteurs, plusieurs pistes pourraient être explorées dans les recherches futures :

- **Optimisation de l'Algorithme :** Poursuivre le développement de stratégies d'initialisation et de convergence pour améliorer la performance et la stabilité de l'algorithme. Surtout dans mon cas où l'algorithme est particulièrement lent.
- **Comparaisons avec d'Autres Méthodes :** Comparer de manière approfondie les kmeans généralisés avec d'autres techniques de clustering, notamment celles basées sur l'apprentissage automatique et les réseaux de neurones.

En conclusion, ce projet souligne l'importance et la valeur des kmeans généralisés dans le domaine du clustering de données, offrant une méthode puissante et flexible pour découvrir des structures cachées dans des ensembles de données complexes.