Projet ANDO

Colin Coërchon - Alexandre Fort

18-12-2023

Contents

Projection orthogonale

Projection sur une droite

Considérons une droite $\mathcal{D}$ et un point x dans $\mathbb{R}^p$. Pour n'importe quelle représentation de notre droite $\mathcal{D}$ donnée, on peut se ramener à cette représentation :

$$\mathcal{D} = a + \text{Vect}(u)$$

Avec :

- u vecteur unitaire de $\mathbb{R}^p$ ($\|u\| = 1$), vecteur directeur de la droite.

- $a \in \mathbb{R}^p$, un point de la droite.

De la même manière qu'une projection sur une droite vectorielle dans $\mathbb{R}^p$, la projection $\widehat{x}$ de x sur la droite $\mathcal{D}$ est alors donnée par :

$$\widehat{x} = \text{proj}_{\mathcal{D}}(x) = a + \langle x - a \mid u \rangle \, u$$

On en déduit alors la distance $d(x, \mathcal{D})$ de x à la droite $\mathcal{D}$:

$$d(x, \mathcal{D}) = \inf_{y \in \mathcal{D}} d(x, y) = d(x, \widehat{x}) = \|x - \widehat{x}\|_2$$

Projection sur K droites

On considère maintenant K droites $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K$ dans $\mathbb{R}^p$. On note $\forall k \in [\![1, K]\!]$, $\widehat{x_k} = \text{proj}_{\mathcal{D}_k}(x)$.

La droite la plus proche de x est alors celle qui minimise les distances $(d(x, \mathcal{D}_k))_{k \in [\![1,K]\!]}$.

Implémentation en R

Dorénavant, suivant un ensemble de K droites et un point x de $\mathbb{R}^p$ (muni de sa norme), nous savons trouver mathématiquement quelle est la droite la plus proche de x suivant la norme choisie.

On prendra pour l'implémentation la norme euclidienne de $\mathbb{R}^p$, et on considèrera effectivement que tout droite est définie selon un point et son vecteur directeur. On peut alors en toute logique écrire le code souhaité :

```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
  a + sum((x - a) * u) * u  # Calcul de la projection
}

# Fonction pour trouver la droite la plus proche parmi une liste de droites
droite_la_plus_proche <- function(x, droites) {
  distances <- lapply(droites, function(droite) {
    proj <- projection_sur_droite(x, droite)
    sqrt(sum((x - proj)^2))
  })
  droites[[which.min(distances)]]  # Retourne la droite la plus proche de x
}
```

Les nuées dynamiques (ou K-means généralisés)

Explication brève de l'idée derrière l'algorithme

Nous avons vu en cours et en TD que l'algorithme k-means classique est une méthode de clustering largement utilisée qui vise à **minimiser l'inertie intra-classe**. Elle est notée I_W, et est donnée par la formule :

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

où K est le nombre de clusters, $C = (c_{ik})$ est une matrice de classification des points dans les différentes classes, x_i sont les points de données, et μ_k **est le centroïde du cluster** k.

Dans cette approche, **le centroïde est le point moyen de tous les points dans le cluster**, ce qui en fait un représentant efficace lorsque les clusters sont compacts et globalement sphériques.

Cependant, dans l'algorithme des nuées dynamiques (ou algorithme des k-means généralisés), nous remplaçons le concept de centroïde par celui de représentant de classe, qui peut prendre différentes formes géométriques, telles qu'une droite dans $\mathbb{R}^p$.

Ici, au lieu de minimiser l'inertie intra-classe, **l'objectif est de minimiser une fonction de coût différente**, qui mesure la somme des distances entre chaque point et le représentant géométrique de sa classe. Cette nouvelle fonction de coût est définie comme :

$$\boxed{J(C, D) = \sum_{i=1}^{n} \sum_{k=1}^{K} c_{ik}\, d^2(x_i, \mathcal{D}_k)}$$

où $C = (c_{ik})$ est *encore* une matrice de classification, $D = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$ représente l'ensemble des représentants de classe (dans notre cas, **des droites**), et $d^2(x_i, \mathcal{D}_k)$ est la distance au carré entre le point x_i et la droite $\mathcal{D}_k$.

Présentation de l'algorithme

L'algorithme opère en plusieurs étapes principales :

1. **Initialisation** : Choisir K droites initiales pour représenter les K classes. Chaque droite est définie par un point a_k et un vecteur unitaire u_k.
2. **Affectation** : Pour chaque point x_i, calculer sa distance à chaque droite $\mathcal{D}_k$ et attribuer le point à la classe dont le représentant est le plus proche.
3. **Mise à jour des représentants de classe** : Ajuster chaque droite représentative $\mathcal{D}_k$ pour minimiser la somme des distances au carré des points de la classe à cette droite.
4. **Itération** : Répéter les deux étapes précédentes jusqu'à convergence.

L'initialisation et l'Affectation

Les deux premières étapes semblent très accessibles. Pour la première, on peut considérer $2K$ points et créer alors les K premières droites initialisant l'algorithme. Pour l'affection, à l'aide de notre travail mathématiques de la première partie, cela semble très accessible d'affecter chaque point x_i à son représentant de classe

Seulement, pour la troisième étape, cela semble beaucoup plus complexe...

Explication de la méthode de mise à jour des représentants de classe

L'étape de mise à jour des représentants de classe constitue un aspect crucial de l'algorithme des nuées dynamiques. Deux options principales sont envisageables pour cette mise à jour :

1. Appliquer une régression linéaire classique pour estimer la meilleure droite au sens des moindres carrés, en utilisant la fonction `lm` en R, qui minimise la somme des carrés des erreurs verticales entre les points et la droite estimée (*notamment vue en MERR*).
2. Utiliser une méthode d'optimisation numérique qui ajuste directement la position et la direction de chaque droite afin de minimiser une fonction de coût basée sur la distance perpendiculaire des points à la droite.

J'ai choisi d'utiliser dans mon implémentation de l'algorithme des nuées dynamiques **la méthode d'optimisation numérique**. Elle est préférée car elle permet une minimisation plus directe et géométriquement appropriée de la distance des points à la droite. Cela est particulièrement important dans des espaces de dimension supérieure à 2 où la notion d'erreur verticale devient moins pertinente (notion davantage favorisée par `lm`).

Seulement, pour utiliser la fonction `optim` de R, il nous faut nécessairement **une fonction de coût à minimiser**.

C'est donc pour cela que j'ai définie la fonction de coût $C(a, u)$. Elle est donc définie, pour une droite explicite, par la somme des distances perpendiculaires au carré de tous les points de la classe à la droite. Pour une droite définie par un point a et un vecteur unitaire u, la fonction de coût pour l'ensemble des points (x_i) est donnée par:

$$C(a, u) = \sum_{i=1}^{n} \left(d(x_i, \mathcal{D})\right)^2,$$

Cette fonction de coût est directement reliée à la minimisation du critère $J(C, D)$ qui est la somme des distances au carré des points à la droite représentative de leur classe respective, c'est-à-dire:

$$J(C, D) = \sum_{k=1}^{K} \sum_{i=1}^{n} c_{ik} \left(d(x_i, \mathcal{D}_k)\right)^2,$$

où c_{ik} est un coefficient binaire indiquant si le point x_i appartient à la classe représentée par la droite $\mathcal{D}_k$. **L'objectif est de trouver les paramètres de la droite** (a_k, u_k) **qui minimisent cette somme pour chaque classe** k.

De plus, la méthode d'optimisation choisie pour ajuster les droites est "L-BFGS-B", une variante de l'algorithme de quasi-Newton qui est, paraît-il, bien adaptée aux problèmes de grande dimension avec des contraintes sur les paramètres.

Implémentation de l'algorithme

On reprends les 4 idées phares développées juste avant : Initialisation, Affectation, Mise à jour des représentants de classe, Itération jusqu'à convergence.

Voici donc l'implémentation de mon algorithme des nuées dynamiques :

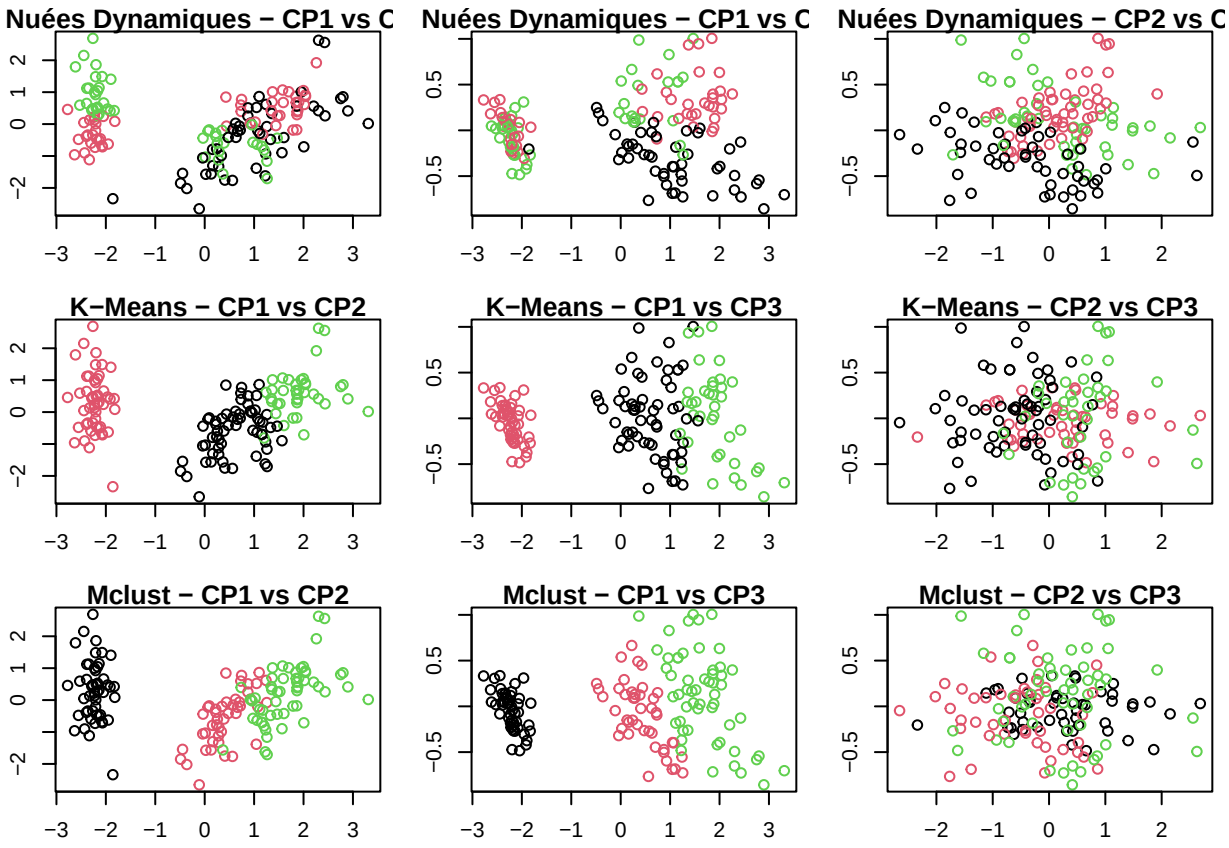
```r
# Fonction pour calculer la projection d'un point sur une droite (représentée par a et u)
projection_sur_droite <- function(x, a, u) {
  a + sum((x - a) * u) * u
}

# Fonction pour calculer la distance au carrée d'un point à une droite (représentée par a et u)
distance_perpendiculaire <- function(x, a, u) {
  proj <- projection_sur_droite(x, a, u)
  sum((x - proj)^2)
}

# Fonction pour représenter une droite avec 2 points de cette dernière en données.
creer_droite_points <- function(P1, P2) {
  u <- P2 - P1
  list(a = P1, u = u / sqrt(sum(u^2)))
}

# Initialisation des droites pour l'algorithme des nuées dynamiques
initialisation_droites <- function(X, K) {
  indices_aleatoires <- sample(nrow(X), 2 * K, replace = FALSE)
  points_aleatoires <- X[indices_aleatoires, ]

  droites <- vector("list", K)
  for (k in 1:K) {
    P1 <- points_aleatoires[2 * k - 1, ]
    P2 <- points_aleatoires[2 * k, ]
    droites[[k]] <- creer_droite_points(P1, P2)
  }
  droites
}

# La fonction de coût pour optimiser une droite.
# "params" définit les vecteur a et u à optimiser (la première moitié = a, la seconde = u)
fonction_de_cout <- function(params, points) {
  a <- params[1:ncol(points)]
  u <- params[(ncol(points)+1):(2*ncol(points))]
  u <- u / sqrt(sum(u^2))

  # On calcule ensuite la somme des distances des points à la droites
  sum(sapply(1:nrow(points), function(i) {distance_perpendiculaire(points[i, ], a, u)} ))
}

# Fonction d'optimisation de la droite pour un cluster
optimiser_droite <- function(points, a_initial, u_initial) {
  # Combinaison des paramètres a et u en un seul vecteur
  params_initiaux <- c(a_initial, u_initial)

  # Exécution de l'optimisation en utilisant les valeurs actuelles de a et u comme point de départ
  # La partie clef de l'optimisation
  result <- optim(params_initiaux, fonction_de_cout, points = points, method = "L-BFGS-B")

  a_opt <- result$par[1:ncol(points)]
  u_opt <- result$par[(ncol(points)+1):(2*ncol(points))]
```

```
  u_opt <- u_opt / sqrt(sum(u_opt^2))

  return(list(a = a_opt, u = u_opt))
}


# Algorithme des k-means généralisés avec des droites comme représentants de classe
nuees_dynamiques_droites <- function(X, K, max_iterations = 10000) {
  n <- nrow(X)

  # Initialisation des représentants de classe (droites)
  droites <- initialisation_droites(X, K)
  classifications <- matrix(0, nrow = n, ncol = K)

  for (iteration in 1:max_iterations) {
    classifications_old <- classifications
    classifications <- matrix(0, nrow = n, ncol = K)

    # Affectation des points aux classes
    for (i in 1:n) {
      x_i <- X[i, ]
      distances <- sapply(1:K, function(k) {
        droite <- droites[[k]]
        distance_perpendiculaire(x_i, droite$a, droite$u)
      })
      classifications[i, which.min(distances)] <- 1
    }

    # Vérifier si les classifications ont changé
    if (all(classifications == classifications_old)) {
      break
    }

    # Mise à jour des représentants de classe
    for (k in 1:K) {
      points_cluster_k <- X[classifications[, k] == 1, ]
      if (nrow(points_cluster_k) > 0) {
        droite_optimisee <- optimiser_droite(points_cluster_k, droites[[k]]$a, droites[[k]]$u)
        droites[[k]] <- droite_optimisee
      }
    }
  }
  return(list(droites = droites, classifications = classifications))
}
```

Un premier exemple avec “iris”

```
set.seed(265)

data(iris)
X_iris <- as.matrix(iris[, 1:4])
```

```r
# Avec notre algorithme des nuées dynamiques
nuees_resultats <- nuees_dynamiques_droites(X_iris, K = 3)

# Avec l'algorithme des kmeans
kmeans_resultats <- kmeans(X_iris, centers = 3)

# Avec l'algorithme des mélanges de gaussiennes (Mclust)
mclust_resultats <- Mclust(X_iris, G=3, verbose = FALSE)
```




```
## [1] "Précision des nuées dynamiques: 0.553"
```

```
## [1] "Précision des K-means: 0.893"
```

```
## [1] "Précision des Mclust: 0.967"
```

On remarque que l'algorithme K-means et la méthode de mélange des gaussiennes (Mclust) donnent des résultats très similaires (0.96 et 0.89 de précision). Et, à l'opposé, les résultats fournis par notre algorithme des nuées dynamiques ne semblent pas très concluants (0.553 de précision).

En effet, après une analyse en composantes principales réalisée par la fonction `PCA`, les résultats selon les 2 premières composantes et selon la première et la troisième donnent des résultats très intéressants :

- Les résultats fournis par `Mclust` et `kmeans` sont effectivement quasiment identiques. Ils ont tous les deux trouvés les mêmes 3 clusters principaux, relatifs aux 3 espèces de fleurs présentes dans les données (cf. les matrices de confusion). **Le clustering est donc réussi pour ces deux algorithmes**.

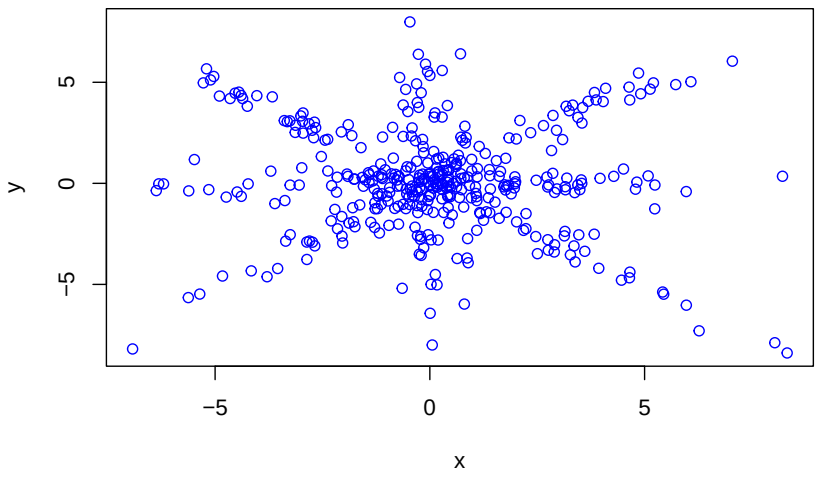
- En revanche, les performances de notre algorithme de nuées dynamiques laissent à désirer. En pratique, il arrive difficilement à identifier clairement un cluster et mélange un peu les 3. Il apparaît donc que **les données du dataset `iris` ne sont pas bien adaptées à un clustering par nuées dynamiques** qui utilise des droites comme représentants des classes.

Un deuxième exemple

Pour cet exemple, j'ai choisi de me placer dans $\mathbb{R}^2$ et j'ai eu l'idée de placer des points aléatoirement autour de plusieurs droites prédéfinies. J'ai choisi pour cela 4 droites :

- $\mathcal{D}_1$: La droite portée par la fonction "$y = x$".
- $\mathcal{D}_2$: La droite portée par la fonction "$y = 0$".
- $\mathcal{D}_3$: La droite portée par la fonction "$y = -x$".
- $\mathcal{D}_4$: La droite portée par l'équation "$x = 0$".

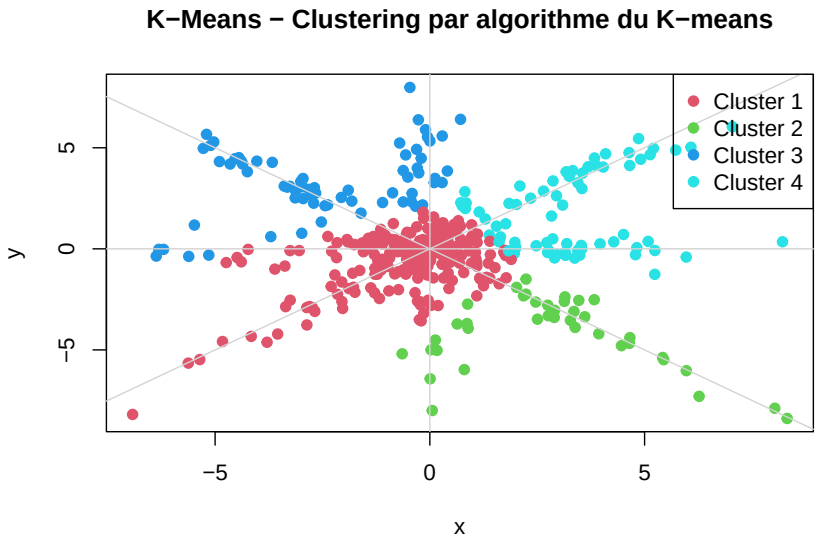
Exemple 2 : Nuage de points portés par 4 droites



Et de la même manière que durant l'exemple précédent, j'ai appliqué sur cet exemple les 3 algorithmes que l'on souhaite comparer, avec l'espoir qu'ils retrouvent tous les 3 les clusters portés par les 4 droites du plan.

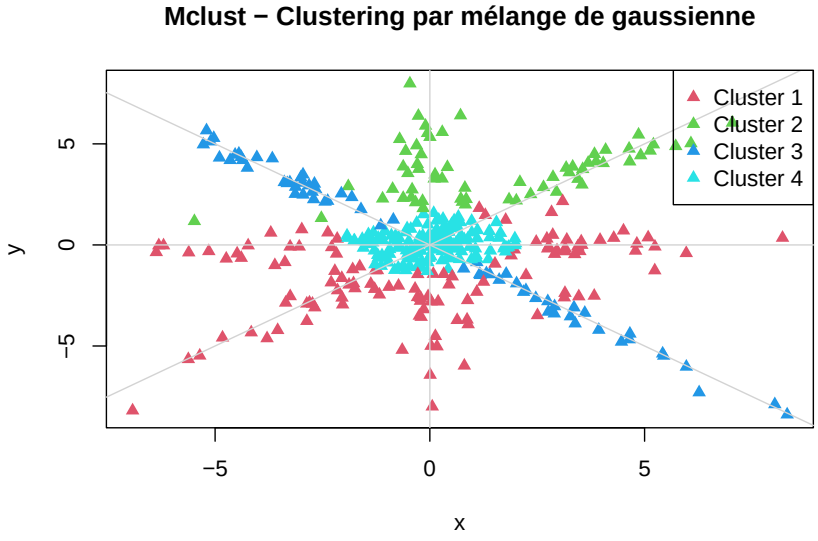
Avec l'algorithme des kmeans de R

```
kmeans_resultats <- kmeans(X, centers = K)
```



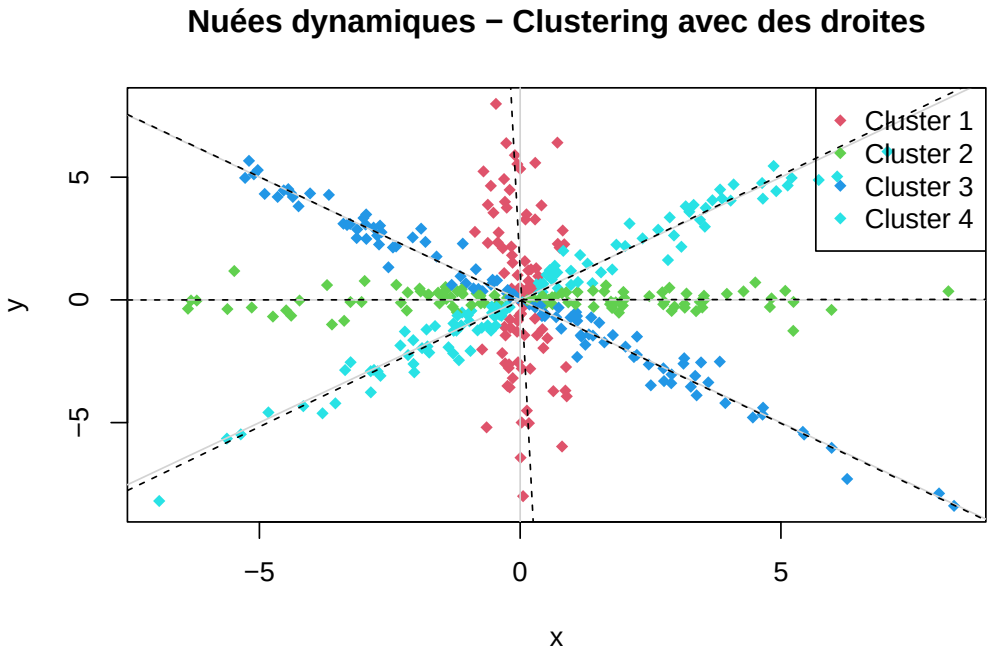
Avec le modèle de mélange gaussien

```
mclust_resultats <- Mclust(X, G=K, verbose = FALSE)
```

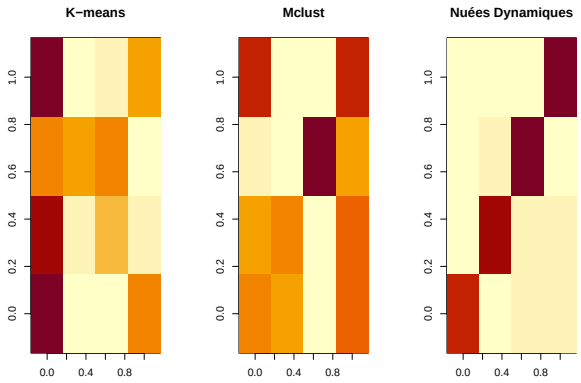


Avec mon modèle de nuées dynamiques

```
set.seed(251)

# Appliquer l'algorithme des nuées dynamiques
resultats <- nuees_dynamiques_droites(X, K = K)
```



Comparaison des résultats




```
## [1] "Précision des K-means: 0.355"
```

```
## [1] "Précision des Mclust: 0.448"
```

```
## [1] "Précision des nuées dynamiques: 0.787"
```

Les résultats des matrices de confusion montrent que notre algorithme des nuées dynamiques a été particulièrement performant dans cet exemple spécifique. Il a réussi à identifier correctement (avec une précision supérieure à 0.78) les quatre clusters associés aux droites préétablies $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3,$ et $\mathcal{D}_4$.

Alors que l'algorithme Mclust peut parfois atteindre ce niveau de performance dans cet exemple (selon la chance d'initialisation qu'on a, ainsi que l'agencement initial des points), Kmeans n'a jamais réussi à

distinguer les quatre clusters distincts. Cela illustre **une amélioration significative par rapport à l'approche Kmeans**, démontrant l'efficacité de notre algorithme de nuées dynamiques dans des situations où les données sont réparties le long de structures linéaires distinctes dans l'espace.

Conclusion

Pour conclure, on comprend bien que l'algorithme des nuées dynamiques construit ci-dessus ne convient pas universellement à tous les types de structures de données. Cette méthode, bien qu'efficace dans certains contextes, montre ses limites face à des distributions plus regroupés et circulaires, ou à l'inverse davantage complexe. Dans ces situations, d'autres algorithmes comme les "K-means", "Mclust" ou encore "Kernel PCA" peuvent s'avérer plus performants. Cette expérience souligne l'importance cruciale de la sélection de l'algorithme en fonction des caractéristiques spécifiques des données traitées. Elle rappelle également que dans le domaine du clustering, il n'existe pas de solution unique, mais plutôt une nécessité d'adapter l'approche analytique à la nature des données pour extraire des résultats pertinents et précis.

Il est important de noter que dans l'exemple que j'ai élaboré, il arrive que l'algorithme des nuées dynamiques ne parvienne pas toujours à identifier correctement les clusters. Bien que cela soit un évènement rare dans cet exemple, cette possibilité existe en raison de **la forte dépendance de l'algorithme vis-à-vis de son initialisation**, qui est entièrement aléatoire.

Ce défi lié à **l'initialisation semble être un problème récurrent et essentiel** dans de nombreux algorithmes similaires. Actuellement, il semble que la recherche d'une bonne heuristique pour initialiser de tels algorithmes soit un sujet de préoccupation et de débat actuel dans le domaine.

Questions subsidiaires

Idée mathématique

L'algorithme des nuées dynamiques que nous venons de présenter faisait intervenir la minimisation de cette fonction :

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

La question subsidiaire porte alors sur l'extension de dimension des représentants de classes dans notre algorithme. Actuellement, les représentants des classes étaient les droites $\mathcal{D}_k$. Au lieu de ça, on va s'intéresser à prendre un hyperplan de $\mathbb{R}^p$ comme représentant pour chaque classe k.

Lorsque nous passons de l'utilisation de droites comme représentants de classe à des hyperplans dans $\mathbb{R}^p$, nous devons **ajuster la fonction de coût** pour tenir compte de la distance d'un point à un hyperplan. Seulement, le grand avantage que l'on a avec les hyperplans, c'est qu'ils peuvent être décrits par l'équation "$\langle n \mid x \rangle + b = 0$", où n est un vecteur normal à l'hyperplan, et b est le terme de biais.

On considère donc un hyperplan $\mathcal{H}$ de $\mathbb{R}^p$. Cet hyperplan peut être décrit par :

- un point a quelconque de $\mathcal{H}$.
- un vecteur normal $\overrightarrow{n}$ de $\mathcal{H}$.

Selon cette définition, la distance d'un point x_i à l'hyperplan $\mathcal{H}$ est alors donnée par la formule :

$$d(x_i, \mathcal{H}) = \frac{\left| \langle \overrightarrow{n} \mid \overrightarrow{x_i - a} \rangle \right|}{\| \overrightarrow{n} \|}$$

En conséquence, nous adaptons la fonction de coût $J(C, D)$ à des hyperplans en la reformulant comme suit :

$$\boxed{J(C, H) = \sum_{i=1}^{n} \sum_{k=1}^{K} c_{ik}\, d^2(x_i, \mathcal{H}_k)}$$

Cette nouvelle fonction de coût $J(C, H)$ vise à minimiser la somme des carrés des distances des points à l'hyperplan le plus proche, pondérée par les indicateurs c_{ik}, qui spécifient si le point x_i est attribué à la classe k. Pour chaque classe k représentée par l'hyperplan $\mathcal{H}_k$, les paramètres n et a doivent être ajustés pour minimiser $J(C, H)$.

Implémentation de l'algorithme

```r
# Fonction pour créer un hyperplan à partir d'un point et d'un vecteur normal
creer_hyperplan <- function(a, n) {
  list(a = a, n = n)
}

# Fonction pour calculer la projection d'un point sur un hyperplan
projection_sur_hyperplan <- function(x, a, n) {
  n_norm <- n / sqrt(sum(n^2))
```

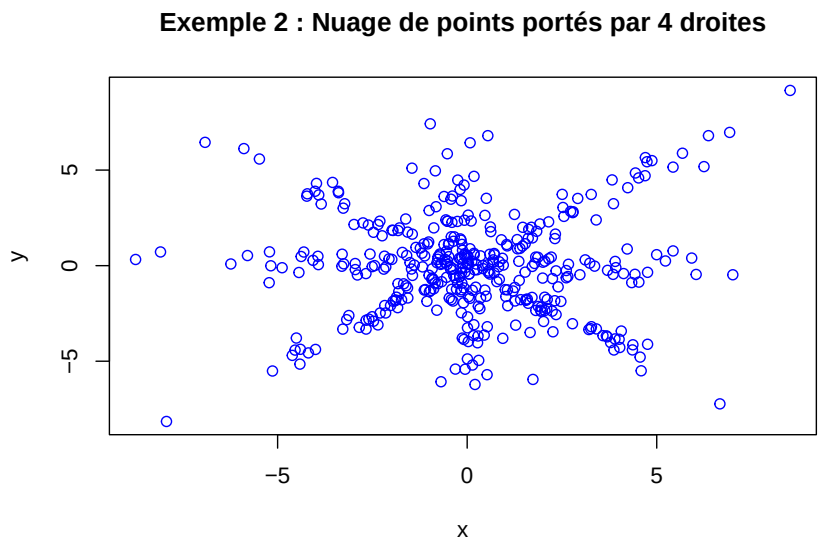
```r
  a + ((x - a) %*% n_norm) * n_norm
}

# Fonction pour calculer la distance d'un point à un hyperplan
distance_perpendiculaire_hyperplan <- function(x, a, n) {
  n_norm <- n / sqrt(sum(n^2))
  abs(sum((x - a) * n_norm))
}

# Fonction pour initialiser les hyperplans pour l'algorithme des nuées dynamiques
initialisation_hyperplans <- function(X, K) {
  n_dim <- ncol(X)
  hyperplans <- vector("list", K)

  for (k in 1:K) {
    index_a <- sample(nrow(X), 1)
    a <- X[index_a, ]
    n <- rnorm(n_dim)
    hyperplans[[k]] <- creer_hyperplan(a, n)
  }
  hyperplans
}

# Fonction de coût pour optimiser un hyperplan
fonction_de_cout_hyperplan <- function(params, points) {
  a <- params[1:ncol(points)]
  n <- params[(ncol(points) + 1):(2 * ncol(points))]
  n <- n / sqrt(sum(n^2))

  sum(sapply(1:nrow(points), function(i) {
    d <- distance_perpendiculaire_hyperplan(points[i, ], a, n)
    d^2
  }))
}

# Fonction d'optimisation de l'hyperplan pour un cluster
optimiser_hyperplan <- function(points, a_initial, n_initial) {
  # Combinaison des paramètres A et n en un seul vecteur
  params_initiaux <- c(a_initial, n_initial)

  # Exécution de l'optimisation en utilisant les valeurs actuelles de A et n
  # comme point de départ
  result <- optim(params_initiaux, fonction_de_cout_hyperplan, points = points,
                  method = "L-BFGS-B")

  a_opt <- result$par[1:ncol(points)]
  n_opt <- result$par[(ncol(points) + 1):(2 * ncol(points))]
  n_opt <- n_opt / sqrt(sum(n_opt^2))

  return(list(a = a_opt, n = n_opt))
}
```

```r
# Algorithme final des nuées dynamiques avec des hyperplans comme représentants de classe
nuees_dynamiques_hyperplans <- function(X, K, max_iterations = 1000) {
  n <- nrow(X)

  # Initialisation des représentants de classe (hyperplans)
  hyperplans <- initialisation_hyperplans(X, K)
  classifications <- matrix(0, nrow = n, ncol = K)

  for (iteration in 1:max_iterations) {
    classifications_old <- classifications
    classifications <- matrix(0, nrow = n, ncol = K)

    # Affectation des points aux classes
    for (i in 1:n) {
      x_i <- X[i, ]
      distances <- sapply(1:K, function(k) {
        hyperplan <- hyperplans[[k]]
        distance_perpendiculaire_hyperplan(x_i, hyperplan$a, hyperplan$n)
      })
      classifications[i, which.min(distances)] <- 1
    }

    # Vérifier si les classifications ont changé
    if (all(classifications == classifications_old)) {
      break
    }

    # Mise à jour des représentants de classe (hyperplans)
    for (k in 1:K) {
      points_cluster_k <- X[classifications[, k] == 1, ]
      if (nrow(points_cluster_k) > 0) {
        hyperplan_optimise <- optimiser_hyperplan(points_cluster_k,
                                                  hyperplans[[k]]$a,
                                                  hyperplans[[k]]$n)
        hyperplans[[k]] <- hyperplan_optimise
      }
    }
  }

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

Test de l'algorithme sur le même exemple

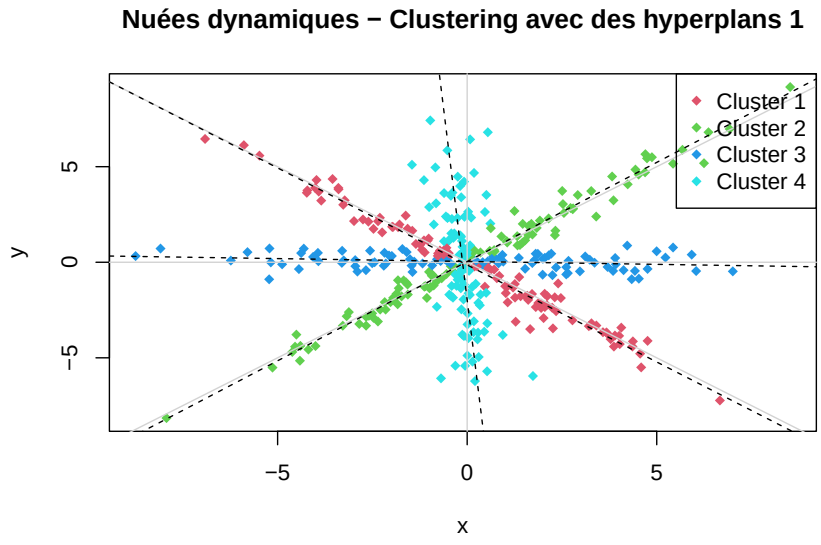
On se place encore une fois dans $\mathbb{R}^2$. Dans cet espace, les hyperplans sont des droites !

Il va donc être plutôt facile de les représenter dans notre exemple.



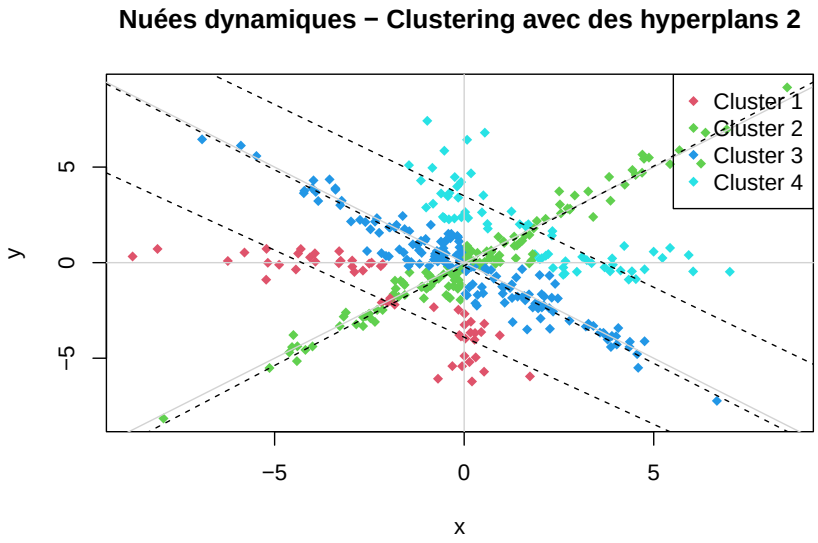
Exemple 1 :

```
set.seed(334)

resultats_hyperplans <- nuees_dynamiques_hyperplans(X, K)
```



Exemple 2 :

```
set.seed(335)

resultats_hyperplans <- nuees_dynamiques_hyperplans(X, K)
```



Ainsi, l'algorithme arrive parfois à retrouver les 4 clusters initiaux selon notre chance au moment de l'initialisation. Mais contrairement aux résultats dans le cadre des nuées dynamiques où les représentants de classe étaient des droites, on arrive ici davantage à des résultats où seulement 2 clusters sont retrouvés.

En effet, l'exemple 2 est l'exemple le plus courant que j'ai réussi à avoir. Et on remarque effectivement que **3 hyperplans (des droites ici) deviennent colinéaires au fil des convergences**. Ce qui impacte forcément les clusters finaux.

Mais, en dépit de ces petites imprécisions, **l'algorithme a montré une certaine capacité à discerner la structure sous-jacente des données**, même si tous les clusters n'ont pas été parfaitement isolés.