Correspondance entre classes étiquettées et Clusters prédits par algorithme hongrois

Willy Kinfoussia, Aurélien Henriques

2023-2024 M2DS

Introduction

Un problème notable dans l'apprentissage supervisé est la labélisation d'un grand nombre de données afin d'entraîner un modèle. En effet, Il arrive souvent que le nombre de données labelisées soit dérisoire comparément aux données dont l'on dispose au total. Certains algorithmes, par exemple l'algorithme SWAV, permettent de s'affranchir de ce problème en créant une représentation de classe par alteration faible des exemples labélisés, mais ils nécessitent un apprentissage contrastif afin de pouvoir identifier une classe.

Les algorithmes de clusterisation non-supervisé peuvent alors permettre de résoudre ce problème, en clusterisant les données par cluster pourvu que nous puissions identifier correctement les clusters aux étiquettes de classe. Dans ce cas, il suffirait alors de s'assurer de la bonne représentation du modèle des différentes classes via les clusters, et d'apposer aux données des clusters les étiquettes de classe issue de l'association cluster/classe.

Dans ce projet, nous proposons d'appliquer l'algorithme hongrois à ce problème. L'algorithme hongrois est un programme qui permet de trouver la meilleure association entre N agents et N tâches à l'aide de poids associés aux arrêtes entre agents et tâches. En pratique, il existe N**2 poids d'arrête et N ! façons différentes d'associer les agents aux tâches. La recherche de l'optimum des associations peut donc être pénible et utiliser l'algorithme hongrois permet de résoudre en partie ce problème, grâce une complexité polynômiale en N. Il repose sur une recherche du poids d'arrête minimal dans une matrice de coût qui contiendrait tous les poids des arrêtes entre agents et tâches (dans notre cas, entre cluster et classe). Nous utilisons ici cet algorithme ainsi qu'un algorithme naîf parcourant les N! possibilités

Construction de la matrice de coût

Nous collectons les données MNIST, ainsi que les classes associées pour obtenir une matrice de coût. Pour ceci, nous effectuons une PCA à 50 dimensions sur les données puis nous clusterisons par k-nn les données obtenues en 10 clusters (le nombre de classe dans MNIST). Nous calculons alors les centres d'inertie des classes et des clusters puis l'on utilise la distance euclidienne entre chaque couple classe/cluster.

$$\mathrm{d}(c,k) = \sqrt{\sum_{l=1}^{50}(c_l - k_l)^2}$$

Où : - c_l représente la valeur du centre d'inertie de la classe c en la l-ème dimension, - k_l représente la valeur du centre d'inertie du cluster k en la l-ème dimension,

```r
mnist_train <-  read.csv('mnist_test.csv')

x_train <- mnist_train[, -1]
y_train <- mnist_train[, 1]

x_train_vec <- x_train / 255

constant_columns <- apply(x_train_vec, 2, function(col) length(unique(col)) == 1)
x_train_vec <- x_train_vec[, !constant_columns]

pca <- prcomp(x_train_vec, center = TRUE, scale. = TRUE, rank = 50)

x_train_pca <- predict(pca, x_train_vec)
```

```r
library(stats)

k <- 10

kmeans_model <- kmeans(x_train_pca, centers = k)

clusters_centroid <- kmeans_model$centers

assignement_cluster <- kmeans_model$cluster

classes_centroid <- list()
```

Optimisation

La matrice de coût est alors :

$$\mathrm{M} = (d_{ck})$$

Où : - c et k varient entre 1 et 10.

La matrice X d'association classe/cluster de taille $(10, 10)$ est définie comme :

$$X_{ck} = \begin{cases} 1 & \text{si la classe } c \text{ et le cluster } k \text{ sont associés} \\ 0 & \text{sinon} \end{cases}$$

avec la contrainte qu'il ne doit y avoir qu'un seul 1 par ligne et des 0 sur le reste, correspondant à ne pas associer la classe c aux autres clusters.

$$\sum_{k=1}^{10} X_{ck} = 1$$

```r
for (class_label in 0:(k - 1)) {

  class_data <- x_train_pca[y_train == class_label, ]

  centroid <- colMeans(class_data)

  classes_centroid[[as.character(class_label)]] <- centroid
}

cost_matrix_mnist <- matrix(0, nrow = k, ncol = k)

for (i in 0:k-1) {
  for (j in 1:k) {

    distance <- sqrt(sum((classes_centroid[[as.character(i)]] - clusters_centroid[j, ])^2))

    cost_matrix_mnist[i+1, j] <- distance
  }
}

cost_matrix_mnist <- lapply(1:k, function(i) as.vector(cost_matrix_mnist[i, ]))
```

```r
#Génération de matrice de coût aléatoire

matrice_couts <- function(n) {

  cost_matrix_random <- matrix(rnorm(n * n, mean = 100, sd = 15), nrow = n, ncol = n)
  liste_matrice <- lapply(1:n, function(i) as.vector(cost_matrix_random[i, ]))

  return(liste_matrice)
}
```

Comme on souhaite aussi qu'un cluster corresponde à une ligne (éventuellement il est possible de faire concorder plusieurs clusters à une classe et inversement en enlevant les contraintes sur la bijection de notre association), une seconde contrainte est qu'il ne doit y avoir qu'un 1 par ligne et des 0 sur le reste.

$$\sum_{c=1}^{10} X_{ck} = 1$$

La fonction de coût à minimiser est la somme des produits $X_{kl} \times M_{kl}$ sur l et k :

$$\text{Coût(association X)} = \sum_{c=1}^{10}\sum_{k=1}^{10} X_{ck} \times M_{ck}$$

Cette fonction correspond au coût de l'association classe/cluster choisie. En la minimisant, on s'assure que l'association permet au maximum de réduire la distance séparant le cluster de la classe.

Algorithme naîf

L'algorithme naîf permet de résoudre ce problème brutalement en calculant toutes les coûts possibles. C'est-à-dire calculer pour chaque bijection classes-cluster la somme des arrêtes associant un cluster à une classe et renvoyer la bijection qui minimise le coût. La complexité théorique de cet algorithmique est N! où N est le nombre de classes/clusters. En effet, pour la classe 1, il y a 10 clusters possibles. Une fois un cluster choisi, on obtient le premier coefficient de coût. Il reste donc pour la classe 2 9 clusters possibles, 8 pour la classe 3, etc. Ceci donne bien N! associations possibles donc N! somme des coûts possibles, ce qui donne la complexité de cet algorithme.

Explication détaillée de l'algorithme :

```
library(Rcpp)
```

```
## Warning: le package 'Rcpp' a été compilé avec la version R 4.3.3
```

```
sourceCpp("NaiveAlgorithme.cpp")
source("NaiveAlgorithme.R")
```

Fonction NaiveAlgorithme :

Cette fonction résout le problème d'affectation en générant toutes les permutations possibles de l'assignation et en trouvant celle avec le coût minimal. Elle prend en entrée la matrice d'association. Elle initialise le vecteur minAssignment pour stocker l'assignation optimale avec le coût minimal et le coût minimal lui-même à une valeur maximale. Ensuite, elle initialise un vecteur currentAssignment pour stocker l'assignation courante. Elle génère tous les indices de permutation de 0 à n-1. Puis, elle itère à travers toutes les permutations possibles : Elle construit l'assignation courante en utilisant les indices de permutation. Elle calcule le coût de l'assignation courante en utilisant la fonction calculateCost. Elle met à jour l'assignation optimale et le coût minimal si le coût de l'assignation courante est inférieur au coût minimal actuel. Après avoir exploré toutes les permutations, elle renvoie une paire contenant l'assignation avec le coût minimal et le coût minimal lui-même.

Fonction calculateCost :

Cette fonction calcule le coût total d'une affectation donnée en fonction de la matrice d'association et de l'assignation. Elle initialise le coût total à zéro. Ensuite, elle parcourt chaque ligne de la matrice d'association. Pour chaque ligne, elle ajoute le coût de l'élément correspondant dans la matrice d'association selon l'assignation donnée. Enfin, elle retourne le coût total calculé.

Exemple d'utilisation du package :

```
cost_matrix <- list(
  c(1, 1, 5),
  c(5, 9, 7),
  c(8, 1, 9)
)
naive_adj_matrix <- NaiveAlgorithme(cost_matrix)
naive_adj_matrix
```

```
## [[1]]
## [1] 1 0 0
##
## [[2]]
## [1] 0 0 1
##
## [[3]]
## [1] 0 1 0
```

```r
# Exemple d'utilisation :
# Créer une matrice de coût (exemple)
costMatrix <- matrix(c(1, 1, 5, 5, 9, 7, 8, 1, 9), nrow = 3)

# Appeler l'algorithme naïf
NaiveAlgorithme_R(costMatrix)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    0    1
## [3,]    0    1    0
```

Complexité Théorique

La complexité de l'algorithme naïf est de O(n!), où n est la taille de la matrice d'association. Cette complexité est due à la génération de toutes les permutations possibles des affectations et au calcul du coût pour chaque permutation.

Génération de toutes les permutations :

- La fonction utilise la fonction next_permutation, qui génère toutes les permutations possibles des indices de 0 à n-1.
- Il y a n! permutations possibles dans le pire des cas, car il y a n éléments à permuter.
- La complexité de la génération de chaque permutation est O(n), car elle nécessite un tri lexicographique des indices.

Calcul du coût pour chaque permutation :

- Pour chaque permutation générée, le coût de cette permutation est calculé en utilisant la fonction calculateCost.
- La fonction calculateCost parcourt chaque ligne de la matrice d'association une fois, ce qui prend O(n).
- Dans le pire des cas, la fonction calculateCost est appelée n! fois, une fois pour chaque permutation.

Ainsi, la complexité totale de l'algorithme naïf est de O(n!), car la génération de toutes les permutations possibles et le calcul du coût pour chaque permutation contribuent tous deux à cette complexité. Cette approche est pratique pour de petites tailles de matrices, mais devient rapidement inefficace pour des tailles de problèmes plus importantes en raison de sa complexité exponentielle.

Complexité expérimentale

```
temps_execution <- function(n_liste,p,algo) {

  temps <- numeric(length(n_liste))

  for (j in seq_along(n_liste)) {
    n <- n_liste[j]
    cout <- matrice_couts(n)

    start_time <- Sys.time()

    for (i in 0:(p-1)) {
    algo(cout)
    }

    end_time <- Sys.time()
    temps[j] <- (end_time - start_time)/p
  }
  return(temps)
}
```

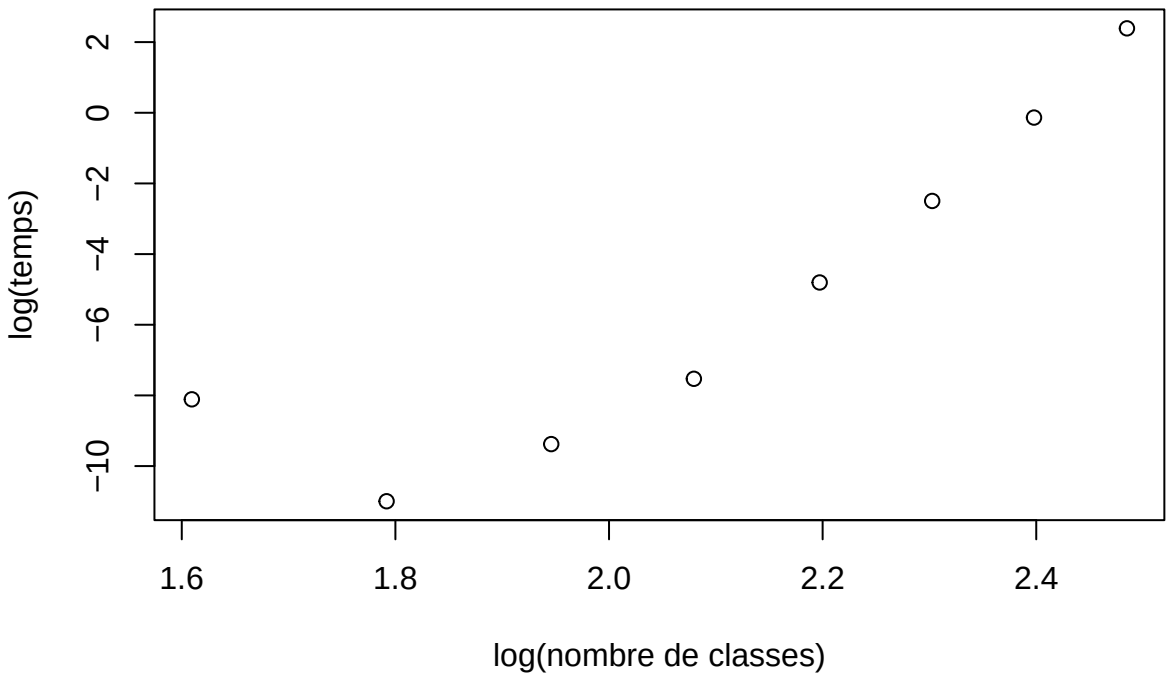
Pour obtenir la complexité, on détermine la régression log(T)=f(n*log(n)-n) où l'on a utilisé la formule de stirling par simplicité:

```
liste_dimension <- seq(from = 5, to = 12, by = 1)

logn_n <- (log(liste_dimension)-1)*liste_dimension

temps_exe <- log(temps_execution(liste_dimension,5,NaiveAlgorithme))

plot(log(liste_dimension),temps_exe,xlab='log(nombre de classes)',ylab='log(temps)')
```




```
reg <- lm(temps_exe~logn_n)

summary(reg)
```

```
##
## Call:
## lm(formula = temps_exe ~ logn_n)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.3292 -1.2424 -0.1511  0.4688  3.0247
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -13.7634     1.2798 -10.755 3.82e-05 ***
## logn_n        0.8625     0.1150   7.498 0.000291 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.581 on 6 degrees of freedom
## Multiple R-squared:  0.9036, Adjusted R-squared:  0.8875
## F-statistic: 56.22 on 1 and 6 DF,  p-value: 0.0002911
```

```
print(coef(reg)[2])
```

```
##    logn_n
## 0.8625484
```

La régression donne des valeurs proches de 1 (0.88-0.94) avec un Rˆ2>0.88. Ainsi, on a bien une complexité en N! par équivalence avec n*log(n)-n.

Algorithme Hongrois

L'algorithme hongrois, également connu sous le nom de méthode hongroise ou algorithme de Kuhn-Munkres, est un algorithme utilisé pour résoudre le problème d'affectation, qui consiste à trouver la meilleure correspondance entre deux ensembles d'éléments, tout en minimisant le coût total de cette affectation.

- Étape 0 : Soustraire le minimum de chaque ligne à tous les éléments de cette ligne, puis soustraire le minimum de chaque colonne à tous les éléments de cette colonne dans la matrice de coût.

- Étape 1 : Sélectionner le maximum de zéros indépendants, c'est-à-dire un seul zéro par ligne et par colonne, en parcourant tous les zéros non sélectionnés et en les sélectionnant s'ils ne partagent pas la même ligne ou colonne qu'un zéro déjà sélectionné.

- Étape 2 : Couvrir chaque colonne ayant un zéro sélectionné. Puis, pour chaque zéro non couvert, marquer la colonne de ce zéro et couvrir la ligne de ce zéro si aucune colonne n'est marquée pour ce zéro. Répéter jusqu'à ce qu'il n'y ait plus de zéros non couverts.

- Étape 3 : Trouver la valeur minimum des éléments non couverts dans la matrice. Ajouter cette valeur à toutes les lignes couvertes et la retirer à toutes les colonnes non couvertes. Recommencer à l'étape 1

Explication détaillée de l'algorithme :

```
sourceCpp("Hungarian.cpp")
source("Hungarian.R")
```

Voici une explication complète du fonctionnement de l'algorithme hongrois :

Initialisation :

La fonction Hungarian est appelée, prenant en entrée une matrice de coûts matrix qui représente les coûts d'affectation entre deux ensembles d'éléments. Cette matrice peut être rectangulaire mais est transformée en une matrice carrée pour l'algorithme. Une copie de la matrice originale est faite pour sauvegarder les valeurs initiales.

La fonction adjust_matrix est utilisée pour rendre la matrice d'entrée carrée en ajoutant des zéros aux rangées ou aux colonnes supplémentaires si nécessaire.

Des vecteurs temporaires sont initialisés pour suivre les affectations et les couvertures de lignes et de colonnes. La matrice M est initialisée pour stocker les affectations (zéros étoilés et zéros primés).

Itération à travers les étapes de l'algorithme :

- Étape 1 - Initialisation :

L'algorithme commence par l'étape 1, où il effectue des ajustements sur la matrice d'entrée pour garantir qu'elle soit carrée. Il soustrait le minimum de chaque ligne de la matrice et le minimum de chaque colonne, de sorte que dans chaque ligne et chaque colonne, il y ait au moins un zéro. Cela crée une configuration initiale pour l'algorithme.

- Étape 2 - Étoiles dans la matrice :

À l'étape 2, l'algorithme cherche des zéros non couverts dans la matrice d'entrée. Si un zéro est trouvé, il est étoilé (marqué). Les lignes et les colonnes contenant ces zéros étoilés sont ensuite couvertes (marquées). L'algorithme continue à chercher des zéros non couverts jusqu'à ce qu'il n'y en ait plus.

- Étape 3 - Couvertures des colonnes :

À l'étape 3, l'algorithme examine les zéros étoilés dans chaque colonne de la matrice. Si une colonne contient un zéro étoilé, cette colonne est couverte. L'algorithme continue à parcourir les colonnes jusqu'à ce que toutes les colonnes contenant des zéros étoilés soient couvertes.

- Étape 4 - Chemins augmentants :

À l'étape 4, l'algorithme cherche à construire des chemins alternés (chemins augmentants) à travers les zéros étoilés et primés dans la matrice. Ces chemins sont utilisés pour effectuer des changements dans les affectations et sont un aspect clé de l'algorithme. L'algorithme peut construire plusieurs chemins alternés à partir de zéros étoilés et primés.

- Étape 5 - Mise à jour des affectations :

À l'étape 5, l'algorithme met à jour les affectations en utilisant les chemins alternés construits à l'étape précédente. Les zéros étoilés sont transformés en zéros primés, et les zéros primés sont transformés en zéros étoilés. Ensuite, les lignes et les colonnes sont à nouveau couvertes.

- Étape 6 - Réduction des coûts :

À l'étape 6, l'algorithme réduit les coûts dans la matrice d'entrée en ajoutant ou en soustrayant une valeur spécifique à chaque élément. Cette valeur est déterminée en fonction des zéros étoilés et primés et des lignes et colonnes couvertes. L'objectif est de rendre la matrice plus propice à la recherche de nouvelles affectations.

Fin de l'itération :

Une fois toutes les étapes terminées, l'algorithme vérifie s'il a trouvé une solution optimale. S'il n'y a plus de changements à apporter aux affectations, cela signifie qu'une solution optimale a été trouvée, et l'algorithme se termine. Sinon, il retourne à l'étape 3 et continue l'itération.

L'algorithme sort de la boucle une fois qu'une solution optimale est trouvée ou qu'il n'y a plus d'étapes à exécuter.

Fonctions auxiliaires :

Diverses fonctions auxiliaires sont utilisées pour effectuer des opérations spécifiques, telles que la recherche du plus petit élément dans une matrice, la recherche de zéros dans une matrice, la gestion des valeurs négatives, etc.

Exemple d'utilisation du package :

```
cost_matrix <- list(
  c(9, 1, 5),
  c(1, 9, 7),
  c(8, 1, 9)
)
hungarian_adj_matrix <- Hungarian(cost_matrix, verbose = FALSE)

print("Matrice solution :")
```

```
## [1] "Matrice solution :"
```

```
hungarian_adj_matrix
```

```
## [[1]]
## [1] 0 0 1
##
## [[2]]
## [1] 1 0 0
##
## [[3]]
## [1] 0 1 0
```

```
# Définir une matrice d'entrée
matrix_input <- matrix(c(9, 1, 8, 1, 9, 1, 5, 7, 9), nrow = 3)

# Appliquer l'algorithme Hungarian
result <- Hungarian_R(matrix_input, verbose = FALSE)

# Afficher la solution trouvée
print("Matrice solution :")
```

```
## [1] "Matrice solution :"
```

```
print(result)
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    1
## [2,]    1    0    0
## [3,]    0    1    0
```

Complexité Théorique

La complexité de l'algorithme hongrois est de O(n^3), où n est la taille de la matrice carrée d'entrée. Cette complexité est due principalement à la recherche de chemins augmentants dans le graphe bipartite associé à la matrice d'entrée.

Initialisation :

- Copier la matrice originale : O(n^2)
- Ajuster la matrice en une matrice carrée : O(n^2)
- Initialiser la matrice masquée M, les vecteurs de couverture des lignes et des colonnes, et d'autres variables temporaires : O(n^2)

Boucle principale :

- La boucle principale s'exécute jusqu'à ce que l'algorithme soit terminé, ce qui peut prendre plusieurs étapes.
- Chaque étape de la boucle principale implique généralement la recherche de chemins augmentants dans le graphe bipartite associé à la matrice d'entrée.
- La recherche de chemins augmentants utilise l'algorithme de l'arbre enraciné, qui a une complexité de O(n^3) dans le pire des cas.

Calcul de la solution :

- Une fois que l'algorithme est terminé, la complexité de la dernière étape dépend de la manière dont la solution est extraite de la matrice d'affectation.
- Dans certains cas, cela peut impliquer une simple boucle sur la matrice d'affectation, ce qui ajoute une complexité supplémentaire de O(n^2).

Ainsi, la complexité totale de l'algorithme hongrois est dominée par la recherche de chemins augmentants, ce qui donne une complexité globale de O(n^3). Cette complexité fait de l'algorithme hongrois une méthode efficace pour résoudre le problème d'assignation pondérée dans de nombreuses applications.

On fonction de l'implementation de l'algorithme hongrois on peut avoir une autre complexité :

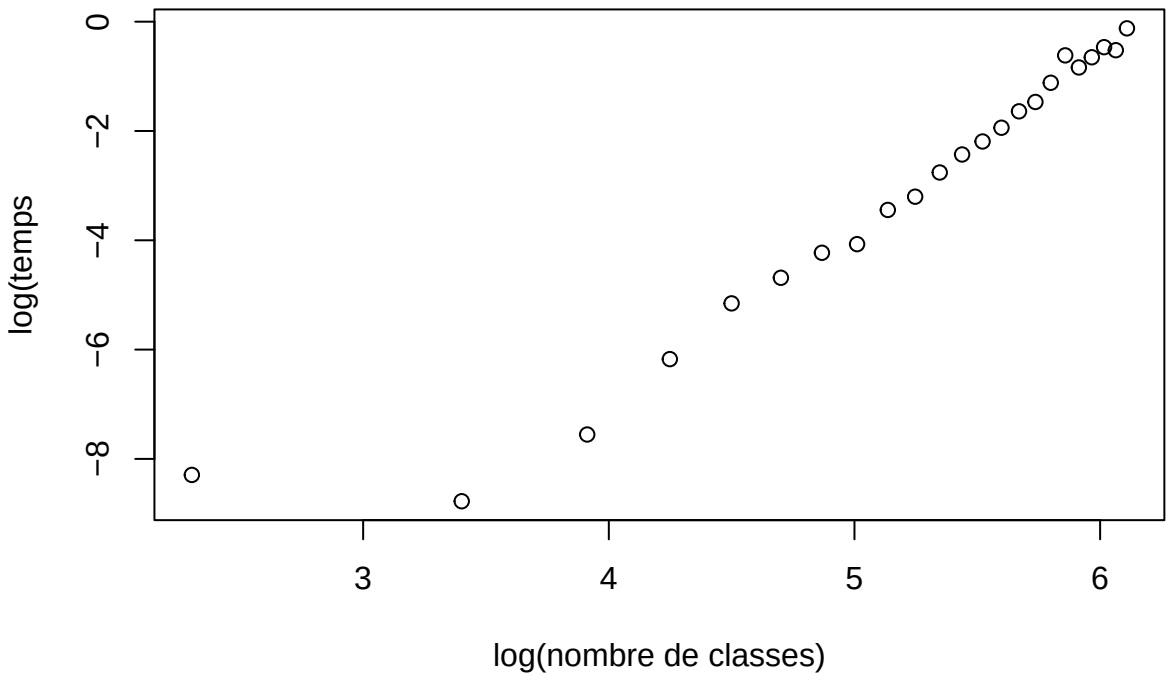
Les différentes étapes sont faites avec une complexité de 1 sur la totalité de la matrice de coût donc ont une complexité maximale de O(n^2). De plus, Une itération de ces étapes garantit d'avoir au moins un zéro indépendant, ce qui indique que l'algorithme résout le problème au maximum en n^2 itérations, correspondant au cas le cas le plus extrême, à devoir itérer sur chacun des coefficients de la matrice d'assignation, qui a n^2 coefficients. On en déduit que l'algorithme a une complexité théorique en O(n^4)

Complexité Expérimentale

```
liste_dimension <- seq(from = 10, to = 450, by = 20)

temps_exe_hong <- log(temps_execution(liste_dimension,5,Hungarian))

plot(log(liste_dimension),temps_exe_hong,xlab='log(nombre de classes)',ylab='log(temps')
```




```
reg <- lm(temps_exe_hong~log(liste_dimension))

summary(reg)
```

```
##
## Call:
## lm(formula = temps_exe_hong ~ log(liste_dimension))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.16243 -0.33594 -0.03334  0.30056  2.33027
##
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)          -16.6786     0.8225  -20.28 2.84e-15 ***
## log(liste_dimension)   2.6297     0.1573   16.72 1.30e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7032 on 21 degrees of freedom
## Multiple R-squared:  0.9301, Adjusted R-squared:  0.9268
## F-statistic: 279.6 on 1 and 21 DF,  p-value: 1.301e-13
```

```
print("complexité polynomiale :")
```

```
## [1] "complexité polynomiale :"
```

```
print(coef(reg)[2])
```

```
## log(liste_dimension)
##             2.629676
```

Comparaison de vitesse R et C++

Nous générons une matrice de coût identique pour les deux algorithmes.

```
generateCostMatrix <- function(n, k) {
  matrix <- vector("list", n)
  for (i in 1:n) {
    matrix[[i]] <- vector("numeric", n)
    for (j in 1:n) {
      matrix[[i]][[j]] <- (i-1)*n + (j-1) + ((i-1) + (j-1)) %% k
    }
  }
  return(matrix)
}

cost_matrix <- generateCostMatrix(6, 4)
cat("Generated Matrix:\n")
```

```
## Generated Matrix:
```

```
print(cost_matrix)
```

```
## [[1]]
## [1] 0 2 4 6 4 6
##
## [[2]]
## [1]  7  9 11  9 11 13
##
## [[3]]
## [1] 14 16 14 16 18 20
##
## [[4]]
## [1] 21 19 21 23 25 23
##
## [[5]]
## [1] 24 26 28 30 28 30
##
## [[6]]
## [1] 31 33 35 33 35 37
```

Chaque algorithme est exécuté plusieurs fois afin d'obtenir une moyenne et un intervalle de confiance.

```r
cost_matrix <- generateCostMatrix(1000, 100)

# Nombre de fois à exécuter l'algorithme
num_executions <- 100

# Vecteur pour stocker les temps d'exécution
execution_times <- numeric(num_executions)

# Exécuter l'algorithme plusieurs fois et mesurer le temps d'exécution à chaque fois
for (i in 1:num_executions) {
  start_time <- Sys.time()
  # Appeler la fonction pour exécuter l'algorithme hongrois
  hungarian_adj_matrix <- Hungarian(cost_matrix)
  end_time <- Sys.time()
  execution_times[i] <- as.numeric(end_time - start_time, units = "secs") * 1000  # en millisecondes
}

# Calculer la moyenne des temps d'exécution
mean_execution_time <- mean(execution_times)

# Calculer l'écart type
standard_deviation <- sd(execution_times)

# Calculer l'intervalle de confiance à 95%
confidence_interval <- 1.96 * (standard_deviation / sqrt(num_executions))

# Afficher les résultats
cat("Moyenne du temps d'exécution :", mean_execution_time, "millisecondes\n")
```

```
## Moyenne du temps d'exécution : 22.81603 millisecondes
```

```r
cat("Intervalle de confiance (95%) :", mean_execution_time - confidence_interval,
    "à", mean_execution_time + confidence_interval, "millisecondes\n")
```

```
## Intervalle de confiance (95%) : 19.92318 à 25.70887 millisecondes
```

Pour l'algorithme hongrois en Rcpp, nous observons approximativement :

- Moyenne du temps d'exécution : 36.61147 millisecondes
- Intervalle de confiance (95 %) : 34.08878 à 39.13415 millisecondes

Tandis que pour l'algorithme hongrois en C++, nous obtenons environ :

- Moyenne du temps d'exécution : 145.686 millisecondes
- Intervalle de confiance (95 %) : [144.176, 147.197] millisecondes

Ces résultats mettent en évidence l'impact significatif de l'implémentation sur les performances des algorithmes, montrant clairement l'avantage de l'implémentation en R en termes de vitesse d'exécution.

Performances sur MNIST

On utilise les clusters associés aux classes pour prédire le test de mnist. Pour ce faire, nous utilisons la matrice d'assignation classe/cluster obtenue pour assigner les classes aux clusters prédits par l'algorithme des k-means. Nous comparons ensuite avec la vraie classe des données test mnist avec celle prédite par cette association.

```
assignation <- Hungarian(cost_matrix_mnist) # matrice bijection classes/clusters trouvés par l'algorith

association <-numeric(k) # version liste de la bijection

for (i in seq_along(assignation)) {

  association[i] <- which(assignation[[i]] == 1)
}

assignement_classe<-association[assignement_cluster] # assignement des données à des classes prédites p

précision <- mean(assignement_classe != y_train) * 100

print(précision)
```

```
## [1] 93.61
```

Nous obtenons une efficacité supérieure à 85%. Nous pouvons donc supposer au-dela du fait que la structure des classes est bien captées par l'algorithme kmeans que l'algorithme hongrois permet de trouver une association cohérente entre les clusters et les classes.