TD Clustering

radu.ciucanu@insa-cvl.fr

Table des matières

Modalités de contrôle de connaissances : Contrôle continu intégral. Pour rappel, le rendu de ce sujet est noté et fait partie du contrôle continu intégral. Le rendu sera une archive (par exemple `zip`). Si vous travaillez seul, l'archive sera nommée `NOM-Prénom`. De manière similaire, si vous travaillez en binôme, l'archive sera nommée `NOM1-Prénom1_NOM2-Prénom2`. Un seul rendu par binôme est suffisant. Les fichiers à inclure dans l'archive sont :

- Votre code source (qui doit fonctionner sur un système d'exploitation de type Unix).
- Un compte rendu en format `pdf` qui explique comment utiliser votre code pour vérifier la qualité de votre travail, ainsi que les réponses aux différentes questions. Le `pdf` sera produit avec Markdown ou Latex ; il est interdit d'utiliser des logiciels type Word.

Alternativement, si vous utilisez Jupyter Notebook, vous pouvez rendre un fichier `.ipynb` qui contient à la fois le code et le compte rendu.

Objectifs.

- Comprendre l'importance des tâches de clustering et comprendre l'algorithme k-means.
- Utiliser Python pour implémenter k-means et visualiser les résultats pour des données 2D.
- Comprendre le clustering hiérarchique, avec différentes méthodes de distance (notamment *single link* et *complete link*) et visualiser les résultats avec un dendrogramme.
- Réaliser une étude de cas sur un jeu de données réelles, via un workflow complet : préparation des données, clustering (k-means ou hiérarchique), visualisation et interprétation des résultats.

1 Clustering k-means

1. Implémentez l'algorithme k-means. On suppose des points de dimension arbitraire (1D, 2D, ...), tous les points ayant bien évidemment la même dimension. Chaque composante d'un point est une variable quantitative. Utilisez la distance euclidienne pour calculer la distance entre 2 points.

   Votre fonction prend comme paramètres :

   - Le jeu de données, sous la forme d'une collection de points.
   - Le nombre de clusters k.
   - La collection de k centroïdes initiaux (paramètre optionnel). Si ce paramètre est `=None`, alors la fonction tire au hasard les k centroïdes initiaux.

   Votre fonction retourne l'erreur (SSE) du clustering obtenu et affiche tous les résultats intermédiaires, comme dans l'exemple suivant. Supposons le jeu de données 1D : `[[1], [2], [18], [20], [31]]`.

   - Avec centroïdes initiaux `[[1], [2], [18]]`, la fonction est censée afficher :

```
Iteration 1
Centroid [1]  Points [[1]]
Centroid [2]  Points [[2]]
Centroid [18]  Points [[18], [20], [31]]
Iteration 2
Centroid [1.0]  Points [[1]]
Centroid [2.0]  Points [[2]]
Centroid [23.0]  Points [[18], [20], [31]]
Fin clustering, erreur =  98.0
```

— Avec centroïdes initiaux `[[18], [20], [31]]`, la fonction est censée afficher :

```
Iteration 1
Centroid [18]  Points [[1], [2], [18]]
Centroid [20]  Points [[20]]
Centroid [31]  Points [[31]]
Iteration 2
Centroid [7.0]  Points [[1], [2]]
Centroid [20.0]  Points [[18], [20]]
Centroid [31.0]  Points [[31]]
Iteration 3
Centroid [1.5]  Points [[1], [2]]
Centroid [19.0]  Points [[18], [20]]
Centroid [31.0]  Points [[31]]
Fin clustering, erreur =  2.5
```

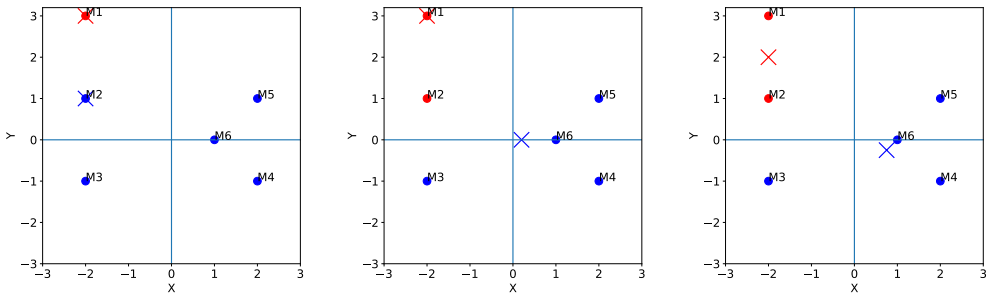
2. Testez votre fonction sur le jeu de données 1D : `[[2], [4], [6], [12], [24], [30]]`
   (a) Avec centroïdes initiaux `[[2], [6]]`
   (b) Avec centroïdes initiaux `[[12], [24]]`

   Pour chaque configuration de centroïdes initiaux, ajoutez les résultats dans le compte rendu, ainsi qu'une courte discussion, en précisant notamment si vous avez obtenu un clustering meilleur que l'autre.

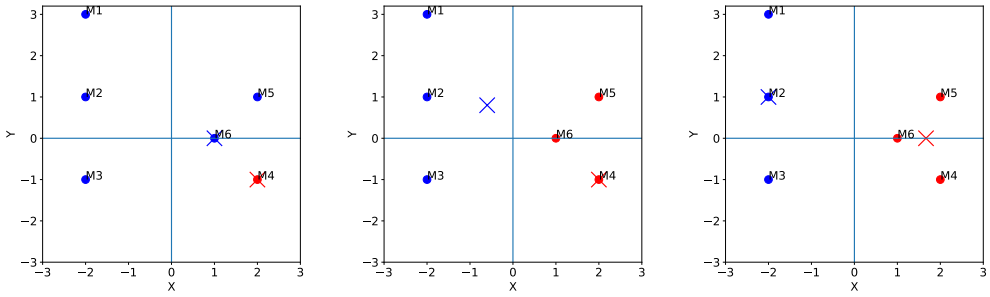
3. Améliorez votre fonction de l'Exercice 1 afin de générer des visualisations 2D sur l'allocation des points dans des clusters au fil des itérations. Pour y arriver, n'hésitez pas à ajouter d'autres paramètres optionnels, afin de faciliter la génération de visualisations 2D. Par exemple, un paramètre utile est la collection des noms des points, qui permet d'étiqueter les points sur chaque figure. Pour chaque paramètre optionnel que vous ajoutez, n'oubliez pas d'inclure une courte explication dans le compte rendu.

   Testez votre fonction sur le jeu de données suivant : $M_1(-2, 3)$, $M_2(-2, 1)$, $M_3(-2, -1)$, $M_4(2, -1)$, $M_5(2, 1)$, $M_6(1, 0)$ et pour les 2 configurations de centroïdes initiaux ci-dessous. Dans le compte rendu, incluez chaque séquence de figures et expliquez comment la reproduire à partir de votre code. Remarquez que les points sont représentés par des $\bullet$ et annotés avec leur nom et les centroïdes sont représentées par des $\times$. Les points d'un même cluster et leur centroïde sont représentés avec la même couleur.

   — Avec centroïdes initiaux M_1 et M_2, la fonction est censée générer la succession de figures :



   — Avec centroïdes initiaux M_4 et M_6, la fonction est censée générer la succession de figures :



4. Comme vous l'avez peut-être déjà deviné, k-means est un algorithme tellement classique qu'il existe déjà des systèmes qui l'implémentent. Par exemple, *SciKit-Learn*[1] est une bibliothèque Python qui implémente de nombreux algorithmes d'analyse de données, y inclus k-means. SciKit-Learn est utile surtout si une seule machine suffit pour analyser les données. Vous avez besoin de *NumPy*[2] afin de manipuler des `array`.

   Pour installer NumPy et respectivement SciKit-Learn :

   ```
   sudo apt install python3-numpy
   sudo apt install python3-sklearn
   ```

   Pour tester l'implémentation k-means de SciKit-Learn sur le jeu de données de l'Exercice 1 :

   ```
   from sklearn.cluster import KMeans
   from numpy import array

   data1 = [[1], [2], [18], [20], [31]]

   print(KMeans(n_clusters=3, n_init=1, init=array([[1], [2], [18]])).fit(data1).labels_)
   # affiche [0 1 2 2 2]

   print(KMeans(n_clusters=3, n_init=1, init=array([[18], [20], [31]])).fit(data1).labels_)
   # affiche [0 0 1 1 2]
   ```

   Expliquez pourquoi cet affichage signifie que l'on a obtenu les mêmes clusters de l'Exercice 1. Ensuite, refaites les clusterings des Exercices 2 et 3 avec SciKit-Learn et vérifiez que vous obtenez les mêmes clusters que vous avez obtenus avec votre code.

2 Clustering hiérarchique

Nous utilisons l'implémentation de clustering hiérarchique disponible dans *SciPy*[3] et nous utilisons des dendrogrammes[4] pour visualiser les résultats. Pour installer SciPy, il suffit de faire : `sudo apt install python3-scipy`

L'implémentation de clustering hiérarchique disponible dans SciPy accepte deux formats possibles pour les données d'entrée[5]. En gros, si tout ce que nous avons est une matrice de distance, alors il faut donner l'entrée dans un format 1D qui représente la matrice de distance en format compact. C'est ce que l'on va faire dans cet exercice. Plus tard, pour l'étude de cas, nous utiliserons l'autre format.

Prenons la matrice de distance suivante, qui, comme toute matrice de distance, est symétrique.

	p_1	p_2	p_3	p_4	p_5
p_1	0	0.1	0.9	0.35	0.8
p_2	0.1	0	0.3	0.4	0.5
p_3	0.9	0.3	0	0.6	0.7
p_4	0.35	0.4	0.6	0	0.2
p_5	0.8	0.5	0.7	0.2	0

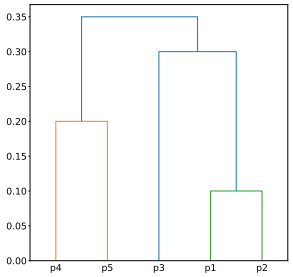
1. https://scikit-learn.org/stable/
2. https://numpy.org/
3. https://www.scipy.org/
4. https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.dendrogram.html
5. https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html

Pour encoder cette matrice de distance en tant qu'entrée du clustering hiérarchique sous SciPy, il suffit de lire les distances au-dessus de la diagonale principale. On obtient ainsi :

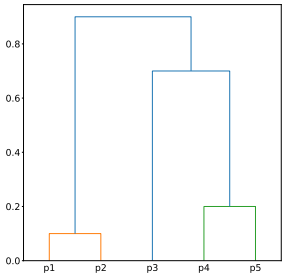
```
data = [0.1, 0.9, 0.35, 0.8, 0.3, 0.4, 0.5, 0.6, 0.7, 0.2]
```

En utilisant SciPy, réalisez un petit programme qui utilise le jeu de données ci-dessus afin de générer les dendrogrammes ci-dessous.

Single link



Complete link

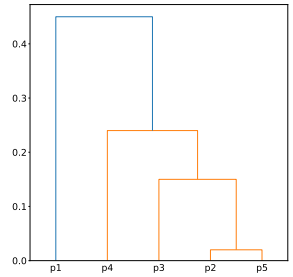


Utilisez votre code en prenant en compte la matrice de similarité suivante [6] :

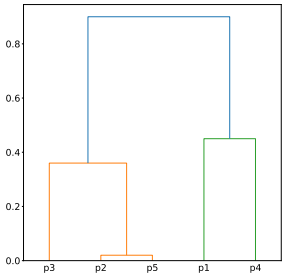
	p1	p2	p3	p4	p5
p1	1.00	0.10	0.41	0.55	0.35
p2	0.10	1.00	0.64	0.47	0.98
p3	0.41	0.64	1.00	0.44	0.85
p4	0.55	0.47	0.44	1.00	0.76
p5	0.35	0.98	0.85	0.76	1.00

Une subtilité importante est que l'on a une matrice de similarité, donc avant de lancer le clustering, il faut la convertir en matrice de distance : distance $(p_i, p_j) = 1 - \text{similarité } (p_i, p_j)$. Vous êtes censés obtenir les dendrogrammes [7] :

Single link



Complete link



3 Etude de cas

1. En pratique, les données brutes que l'on veut analyser ne sont pas par défaut dans un format "idéal" comme celui des exercices précédents. Le but de l'étude de cas est de réaliser un workflow complet qui comprend la préparation des données, le clustering, ainsi que l'interprétation des résultats. Nous utilisons un jeu de données réelles, le fichier `Country-data.csv`, que vous pouvez télécharger sur Celene ou directement sur Kaggle [8].

   Pour la *préparation des données*, faites d'abord du code Python qui parse le fichier `Country-data.csv` et extrait une liste de 167 éléments (car il y a 167 pays dans le jeu de données). Chaque élément de la liste est à son tour une liste de 9 éléments (car chaque pays est caractérisé par 9 variables). Après avoir parsé le

6. cf. exercice 16 du livre https://www-users.cse.umn.edu/~kumar001/dmbook/ch8.pdf
7. donnés aussi dans les solutions des exercices du livre, p 134 https://www-users.cse.umn.edu/~kumar001/dmbook/sol.pdf
8. https://www.kaggle.com/rohan0301/unsupervised-learning-on-country-data

fichier, votre liste doit commencer par `[[90.2, 10.0, 7.58, 44.9, 1610.0, 9.44, 56.2, 5.82, 553.0], [16.6, 28.0, 6.55, 48.6, 9930.0, 4.49, 76.3, 1.65, 4090.0], ...` Dans la suite, nous appellerons cette liste `data_country`. Pensez aussi à stocker les noms des pays dans une liste car ils seront utiles pour étiqueter vos figures (voir, par exemple les Figures 1...4).

Malheureusement, la préparation des données n'est pas encore finie, pour plusieurs raisons, que vous avez idéalement vues par vous-même :

- Chaque point est caractérisé par 9 variables, mais l'humain n'est pas capable de visualiser dans 9 dimensions si on veut visualiser les points et les clusters.
- Les variables ont des échelles très différentes. Par exemple, si vous analysez visuellement les données, vous remarquez que la dernière variable (`gdpp`) a au moins un ordre de grandeur de plus que la plupart des autres variables. Donc si on calcule naïvement des distances sur ces données, la dernière variable aura un poids beaucoup plus important que les autres. Cela n'est pas raisonnable car on veut que chaque variable ait le même poids dans le clustering.

La bonne nouvelle est qu'il existe une recette assez classique pour attaquer ces problèmes. Il s'agit de l'*analyse en composantes principales (en anglais PCA = Principal Component Analysis) sur des données centrées réduites*. Voici un squelette de programme, que vous compléterez afin de finaliser la préparation des données.

```
from numpy import array, identity, transpose, matmul, std, mean
from numpy.linalg import eig

# TODO code pour lire data_country à partir du fichier Country-data.csv

# Les mêmes données de la matrice data_country, dans le format "array" exploitable pour la suite
X = array(data_country)

# Le nombre de points
n = len(countries)

# Le nombre de variables
p = len(data_country[0])

# La matrice des données centrées, c'est-à-dire la somme = 0 sur chaque colonne
Y = X - matmul(transpose(array([n * [1]])), transpose(array([[mean(variable)] for variable in transpose(X)])))

# La matrice des données centrées et réduites, qui en plus a l'écart type constant = 1 sur chaque colonne
Z = matmul (Y, array(list(map (lambda variable : [1./std(variable)], transpose(X)))) * identity(p))

# La matrice (symétrique) de variance/covariance des données centrées réduites
R = matmul (matmul(transpose(Z), 1./n * identity (n)), Z)

# Les vecteurs propres de R
eigenvectors = eig(R)[1]

# Les 2 composantes principals = 2 nouvelles variables contenant le plus d'information possible des 9 variables initiales
components = [matmul(Z, eigenvectors[:,0]), matmul(Z, eigenvectors[:,1])]

# La matrice initiale, projetée sur 2 nouvelles colonnes qui représentent les 2 composantes principales
data_reduced = [[components[0][i], components[1][i]] for i in range(n)]
```

Au final, le résultat `data_reduced` est censé commencer par `[[2.91, -0.1], [-0.43, 0.59], ...`

C'est sur ce jeu de données que vous ferez l'exercice suivant, car il permet de le visualiser facilement en 2D.

2. Sur le jeu de données résultat de l'exercice précédent, générez au moins 2 figures pour chaque type de clustering que l'on a vu.
   - Pour le clustering k-means, réutilisez votre code de la Section 1. Vous obtiendrez des figures similaires aux Figures 1 et 2, pour $k = 2$ et respectivement $k = 3$.
   - Pour le clustering hiérarchique, réutilisez votre code de la Section 2. Une subtilité est que maintenant vous avez la liste des points (dans le format `data_reduced`) et non pas une matrice de distance comme en Section 2. Heureusement que `hierarchy.linkage` accepte les 2 formats. Vous obtiendrez des figures similaires aux Figures 3 et 4, pour single link et respectivement complete link.

   Ajoutez les figures obtenues dans le compte rendu, ainsi que des courtes discussions sur leur interprétation. En particulier, discutez si chaque clustering obtenu semble avoir du sens par rapport à l'équilibre entre les clusters, à la proximité des pays dans les clusters vs sur la carte du monde, discutez s'il y a des proximités entre les pays auxquelles vous vous attendiez ou ce sont des proximités qui vous surprennent, etc.

   Pour k-means avec des centroïdes initiaux tirés au hasard, discutez aussi si on aurait pu obtenir des clusterings considérablement différents de ceux de mes figures.

FIGURE 1 – Jeu de données *Country* : k-means avec $k = 2$.

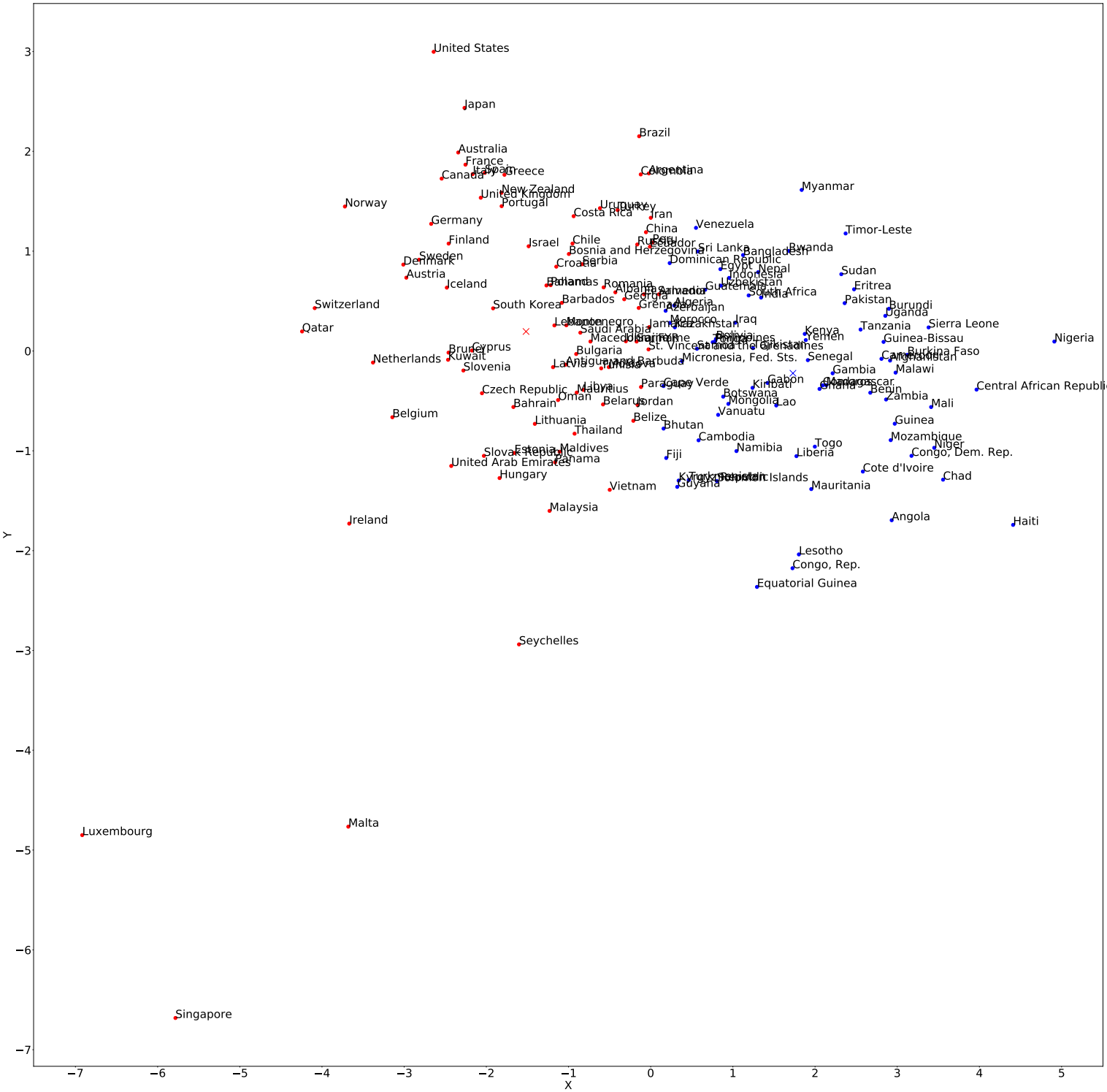


FIGURE 2 – Jeu de données *Country* : k-means avec $k = 3$.

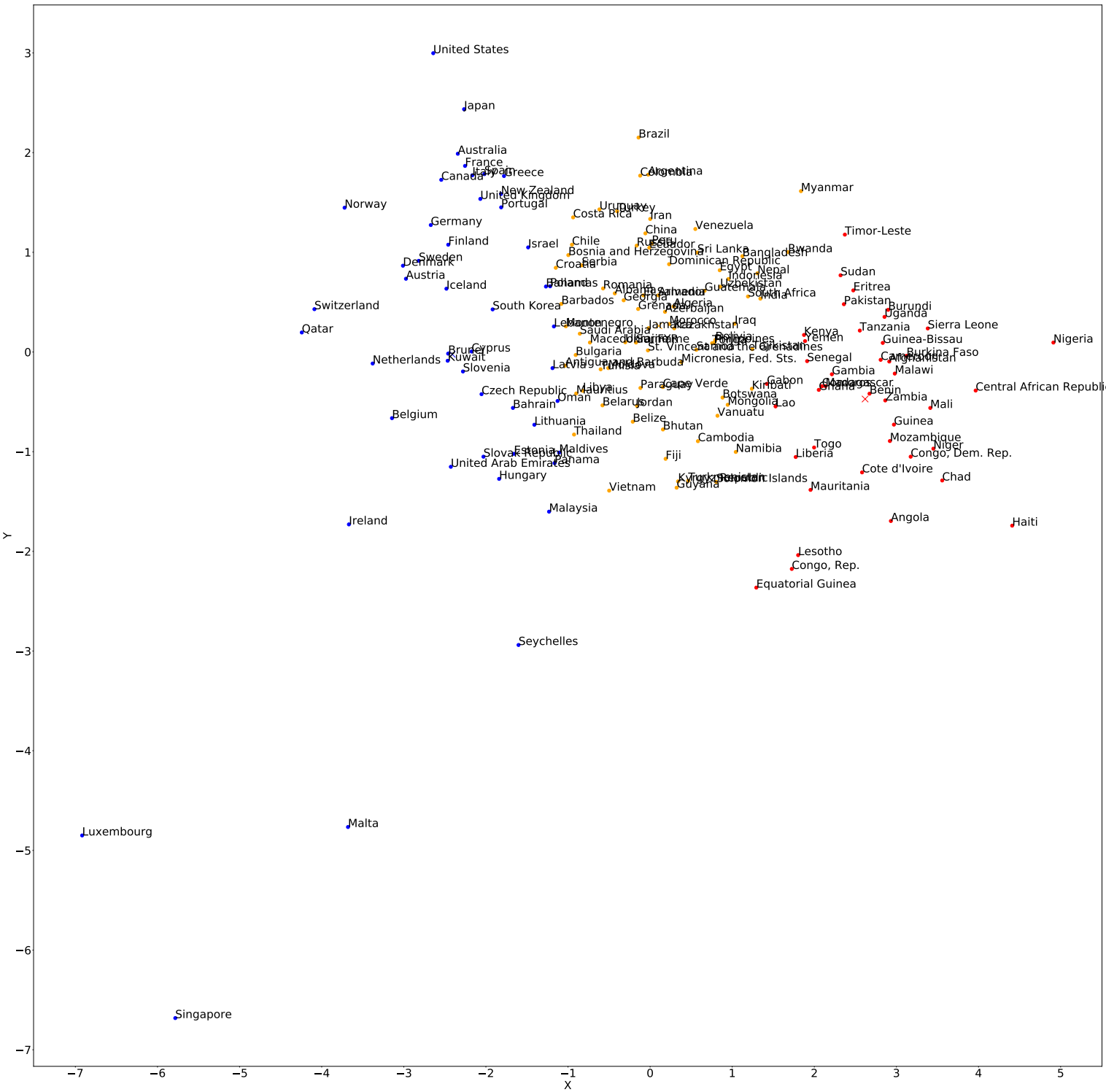


FIGURE 3 – Jeu de données *Country* : clustering hiérarchique avec *Single link*.

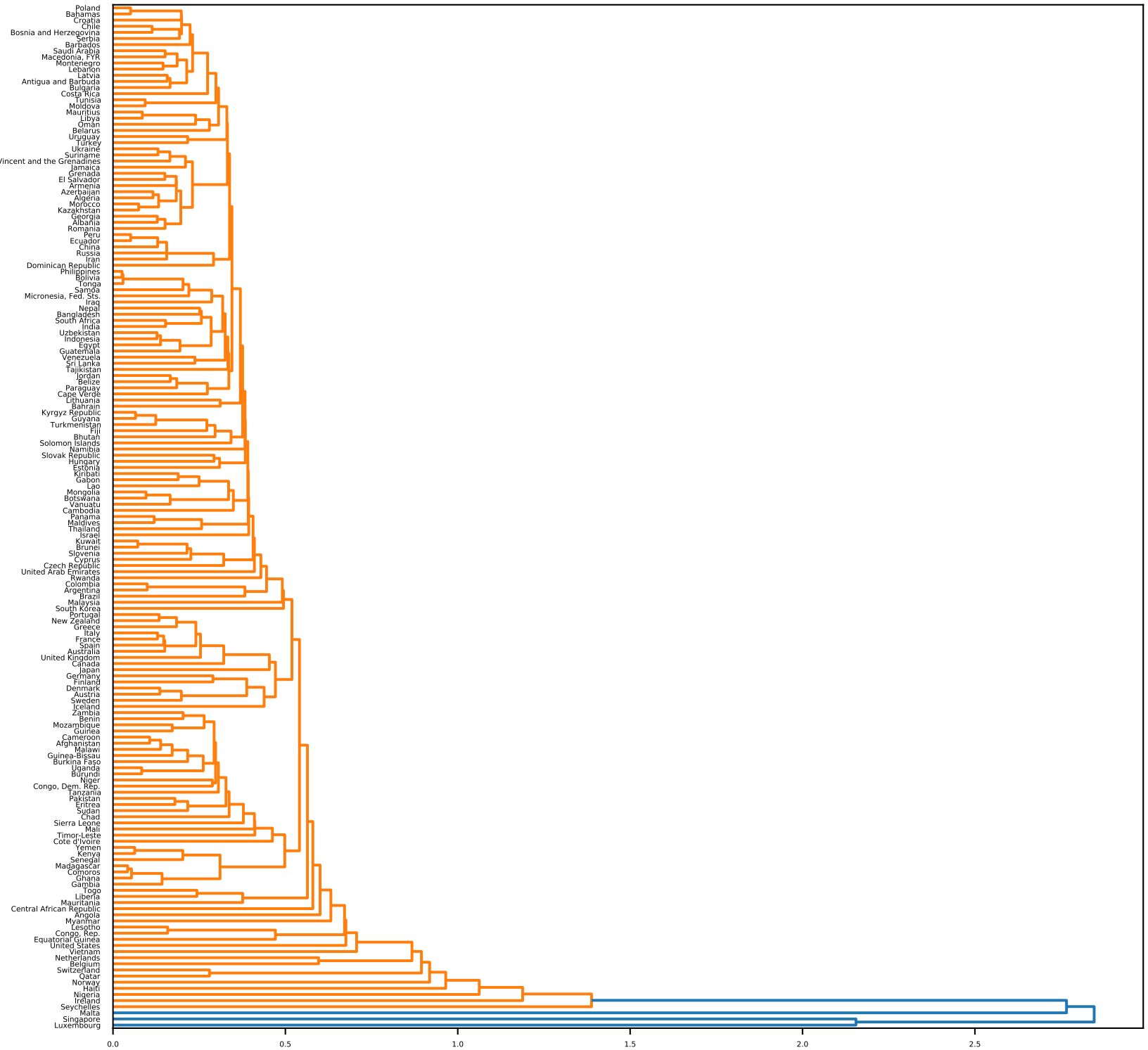


FIGURE 4 – Jeu de données *Country* : clustering hiérarchique avec *Complete link*.