UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

Coclustering en utilisant MapReduce

Auteurs:
DAHMANI LYDIA **21920110**
LABDI WASSIM **21918209**
CHERIFI WISSEM **21604782**
KA WILLIAM **21700589**
LOPES FERNANDES
DYLAN **21807209**

february 2023

Contents

1 Introduction

1.1 Co-clustering

Le coclustering, également connu sous le nom de biclustering ou sous-matrice de clustering, est une technique d'analyse de données qui consiste à regrouper simultanément les lignes et les colonnes d'une matrice de données.

Contrairement aux techniques de clustering traditionnelles qui se concentrent uniquement sur les regroupements des données selon leurs attributs ou caractéristiques, le coclustering permet d'identifier des sous-groupes de données qui sont interdépendants à la fois en termes de lignes et de colonnes.

Le coclustering est largement utilisé dans de nombreux domaines tels que la bioinformatique, la génomique, la recommandation de produits, la segmentation de marché, la reconnaissance de forme, etc. Il peut être utilisé pour découvrir des relations entre les variables et les observations, pour réduire la complexité des données, pour effectuer des tâches de classification, de prédiction et de visualisation des données.

Dans cette technique, les sous-groupes obtenus peuvent avoir des tailles différentes et les variables peuvent être partagées entre différents sous-groupes. Le coclustering peut être effectué soit de manière supervisée, soit de manière non supervisée, en fonction des informations disponibles sur les données.

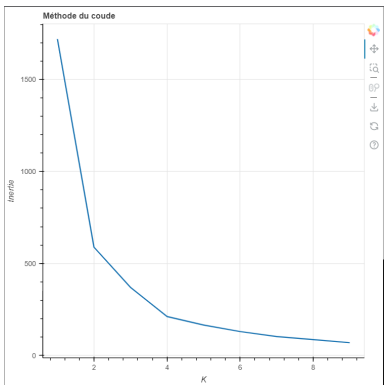
Dans cette perspective, le coclustering est une technique puissante pour l'analyse de données multidimensionnelles complexes, offrant une approche complémentaire et utile aux méthodes de clustering et de classification traditionnelles.

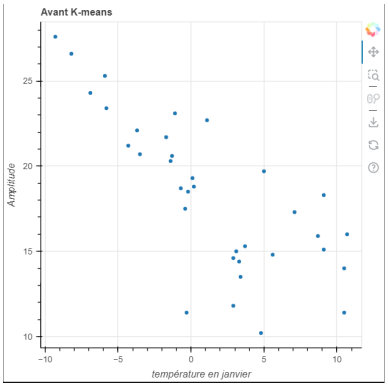
pour faire un parallèle un K-means est un algorithme de clustering non supervisé couramment utilisé en apprentissage automatique pour regrouper un ensemble de données en K clusters. L'objectif est de minimiser la variance intra-cluster, c'est-à-dire la somme des carrés des distances entre chaque point et le centre de son cluster.
pour ce faire :

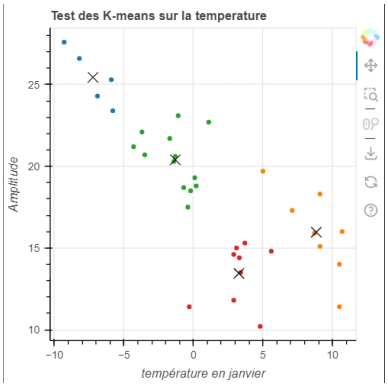
- Le nombre de clusters K est déterminé et les K centres initiaux sont choisis de manière aléatoire ou avec une méthode spécifique
- Chaque point de données est affecté au cluster dont le centre est le plus proche
- le centre de chaque cluster est déplacé vers le centre de gravité de ses points
- Ces étapes sont répétées jusqu'à ce que la variance intra-cluster ne puisse plus être réduite ou jusqu'à ce que le nombre d'itérations maximal soit atteint

un exemple sur un dataset Températures qu'on a pu réaliser avec bokeh pour un algo K-means nous donne les résultats suivants:

on commence par déterminer le nombre de clusters "K"



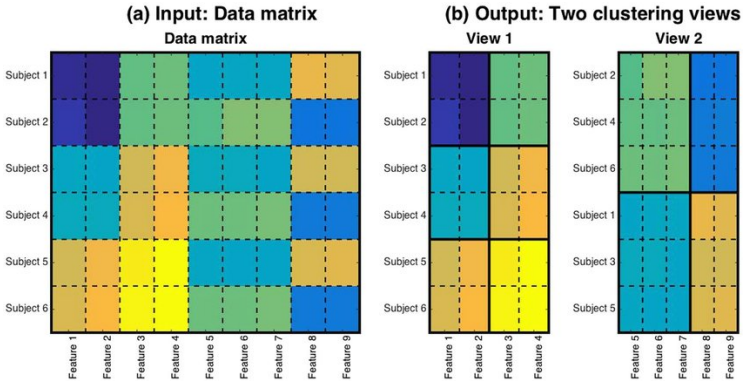




Il est observable que l'ensemble des points de mon jeu de données ont été adéquatement répartis sur les centres de cluster sélectionnés.

Pour faire le lien avec le coclustering évoqué précédemment, ce procédé consisterai a faire un double clustering, autrement dit un k-means sur les ligne et parallèlement sur les colonnes pour réunir les point les plus ressemblant.

voyons ici un résultat d'un coclustring:



nous verrons et analyserons plus en détail le fonctionnement de cet algorithme par la suite.

1.2 Self Organization Maps Network Architecture -SOM-

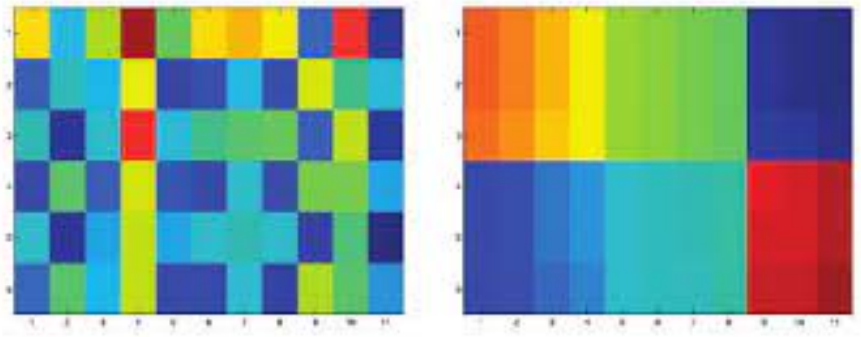
Les cartes d'auto-organisation (ou SOM pour Self-Organizing Maps) sont une technique d'apprentissage non supervisé utilisée en apprentissage automatique et en traitement de données. Elles permettent de visualiser et d'analyser des données multidimensionnelles complexes en les projetant dans un espace de plus petite dimension.

Le fonctionnement des cartes d'auto-organisation se base sur l'idée que les données similaires ont tendance à être regroupées dans l'espace des caractéristiques. La carte est constituée d'un réseau de neurones artificiels organisés en deux dimensions, où chaque neurone représente un point dans l'espace. Lors de la phase d'apprentissage, les neurones sont initialement disposés de manière aléatoire dans la carte.

Ensuite, les données d'entrée sont présentées à la carte, et chaque neurone calcule une mesure de similarité entre lui-même et les données d'entrée. Les neurones les plus similaires aux données d'entrée sont activés, ainsi que les neurones voisins dans la carte. Cette activation est propagée dans la carte à travers un processus appelé "compétition-coopération", qui permet de regrouper les neurones ayant des réponses similaires.

Au fur et à mesure que les données sont présentées à la carte, les neurones se déplacent vers les zones les plus actives de la carte, créant ainsi des groupes de neurones similaires. Ainsi, la carte apprend à représenter les données d'entrée dans un espace de plus petite dimension, tout en conservant leur structure et leur organisation.

Une fois que la carte est entraînée, elle peut être utilisée pour visualiser les données, identifier des groupes et des relations entre les données, et même pour effectuer des tâches de classification ou de clustering. Les cartes d'auto-organisation sont largement utilisées dans de nombreux domaines, tels que la reconnaissance de formes, l'analyse de données, la vision par ordinateur, la bioinformatique, etc..



Une nouvelle approche de biclustering nommée BiTM, basée sur les cartes auto-organisatrices SOM, est présentée.
Nous verrons plus en détail tout au long de cet ouvrage cette nouvelle approche.

2 BiTM

BiTM est une méthode de coclustering qui utilise une approche basée sur les seuils pour identifier les groupes de lignes et de colonnes similaires dans une matrice de données. Cette méthode repose sur une itération de deux étapes:
tout d'abord, des seuils sont utilisés pour séparer les données en sous-ensembles, puis les moyennes des sous-ensembles sont calculées pour former les clusters.

autrement dit la principale visée de BiTM consiste à convertir la matrice de données D en une configuration de blocs ordonnée sur une carte topologique.

Point de vue code , cela peut se traduire par la classe BiTM qui a comme signature :

```
  class BiTM(val nbRowNeuron: Int, val nbColNeuron: Int, datas:
RDD[NamedVector], val topoFactor: TopoFactor) extends Serializable
```

Elle prend en paramètre:

- le nombre de neurones en ligne **K** qu'on appellera observations
- et le nombre de neurones en colonnes **L** qu'on appellera variables
- ainsi que le vecteur contenant l'ensemble des données qui est de type RDD (Resilient Distributed Datasets) qui est une collection distribuée appartenant a Spark
- et finalement TopoFactor qui est un paramètre agissant dans le calcul de la fonction de voisinage qu'on expliquera dans le calcul de l'affectation des observations .

2.1 Initialisation de l'algorithme

l'algorithme BiTM est initialisé par un choix aléatoire des **prototypes** " les représentants " comme on peut le voir :

Initilization
{Random initialization of prototypes}
{Random initialization of columns assignements **W**}

Cette partie consiste a créer la matrice G " matrice de prototype " qui est de taille K x L **(NbNeurons dans le code)** qui est le nombre de clusters par ligne et par colonne .

Dans le code , cela se traduit de la manière suivante :

```
private val _rand = new Random()
 private var _colNeuronAffectation = Array.tabulate(_nbDataCol){i => _rand.nextInt(_nbNeurons)}
 private var _neuronMatrix = initNeurons(_nbNeurons)

 val quantErrors = new mutable.Stack[Double]

 protected def initNeurons(nbNeurons: Int) = {

   val rand = new Random()
   val samples = datas.takeSample(withReplacement = false, nbNeurons, rand.nextInt())
```

```
    new Matrix(samples.map{samp =>
      Array.tabulate(nbNeurons){i =>
        samp(rand.nextInt(samp.length))
      }
    })
  }
```

Comme on peut le voir, l'affectation des colonnes est aussi aléatoire et c'est stocké dans la variable

```
private var _colNeuronAffectation = Array.tabulate(_nbDataCol){i => _rand.nextInt(_nbNeurons)}
```

Après avoir fait cela , la fonction **training** est appelé et c'est dans cette fonction que l'ensemble des fonctions seront implémentés (Affectations des lignes/colonnes , mise a jour ...)
Dans le code :

```
def training(nbIter: Int) {
    // training

    for (iter <- 0 until nbIter) {

    // Affectation des colonnes
      _colNeuronAffectation = new BiTMCol(_neuronMatrix, _nbNeurons,
      _nbDataCol, _colNeuronAffectation).computeColAffectation()

      // Affectation des lignes et re-calcul du modèle
      _neuronMatrix = new BiTMRow(_neuronMatrix, _nbNeurons,
      _colNeuronAffectation, nbIter, iter).computeNewNeurons
    }
  }
```

Tant que le nombre d'itérations donné en paramètre n'est pas atteint, on répète l'algorithme, et comme mentionné dans les commentaires, l'affectation des variables "colonnes" est faites par cette partie du code (explication voir partie **2.2**)

```
  _colNeuronAffectation = new BiTMCol(_neuronMatrix, _nbNeurons,
      _nbDataCol, _colNeuronAffectation).computeColAffectation()
```

l'affectation des lignes "observations" et la phase de mise à jour correspond à la partie du code :

```
    _neuronMatrix = new BiTMRow(_neuronMatrix, _nbNeurons,
      _colNeuronAffectation, nbIter, iter).computeNewNeurons
```

2.2 Affectation des variables

L'affectation des variables " colonnes " correspond au code :

```
  _colNeuronAffectation = new BiTMCol(_neuronMatrix, _nbNeurons,
      _nbDataCol, _colNeuronAffectation).computeColAffectation()
```

et dans l'algorithme a:

$\{$**Assignment of columns** $\}$
for all $\mathbf{x}_i \in \mathbf{D}$ **do**
  $\langle (J, L); V \rangle \leftarrow \text{ColMapper}(\mathbf{x}_i)$
end for
$\langle (J, L); V \rangle \leftarrow \text{ColReducer}(\langle (J, L); V \rangle)$
for each column reduce value $j \in J$ **do**
  $\phi_w(\mathbf{x}^j) = \arg\min(\langle (j, L); V \rangle)$
end for

Concentrons nous sur la classe :

```
class BiTMCol(neuronMatrix: Matrix, nbNeurons: Int, nbDataCol: Int,
colNeuronAffectation: Array[Int])
```

et qui a comme fonction "principale" la fonction :

```
def computeColAffectation(): Array[Int]
```

cette dernière retournera à la fin l'ensemble des affectations des colonnes dans un tableau d'entiers.
On trouve dans cette fonction :

```
  val distByColNeuron = datas.flatMap(localColDists).reduceByKey(_ + _)
```

La fonction **flatMap** applique pour chaque ligne de datas la fonction donné en paramètre qui est **localColDists** cette étape correspond a **Algorithme 15: ColMapper**

Algorithm 15 ColMapper($\mathbf{x}_i$)
  for each column $j = 1..d$ **do**
    for each feature prototype $l = 1..L$ **do**
      emit $\langle (j, l); (x_i^j - g_r^l)^2 \rangle$
    end for
  end for

quant au code cela correspond a:

```
  protected def localColDists(dataRow: Vector) =  Array.tabulate(nbDataCol * nbNeurons) {i =>
    val neuronColId = i % nbNeurons
    val neuronRowId = findBestRowNeuron(dataRow)
    //val neuronRowId = 0
    val dataColId = i / nbNeurons
    val dataVal = dataRow(dataColId)
    val neuronVal = neuronMatrix(neuronRowId, neuronColId)
    val localDist = (neuronVal - dataVal) * (neuronVal - dataVal)
    (i, localDist)
  }
```

voyons ça plus en détails:

Pour chaque élement de la ligne qu'on traite, on calculera la distance entre le prototype et la valeur de la colonne actuelle, ceci peut etre appelé aussi comme "une moyenne", on recupère les indices

correspondant au prototype et a la colonne puis on calcule la distance dans la variable **localDist**.

et donc à la fin on retourne une paire clé valeur qui correspond à l'indice de la colonne ainsi que la distance calculée . Et puisqu'on a plusieurs prototypes, chaque colonne aura sa propre distance avec chaque prototype.

Puis afin de réunir les distances correspondantes a une même colonne mais avec des prototypes différents on fait un reduce, dans l'algorithme cela correspond à :

Algorithm 16 ColReducer($key(j, l)$, V)

$s_v = 0$
for each map value $v \in V$ **do**
  $s_v += v$
end for
emit $\langle (j, l); s_v \rangle$

ceci est fait avec le code qui suit:

```
                    .reduceByKey(_ + _)
```

Ici on voit clairement qu'on utilise le paradigme de **MapReduce** , la partie **Map** consiste à appliquer la fonction **localDists** sur l'ensemble des lignes, et la partie reduce regroupe les distances aux prototypes d'une meme colonne .

```
val colsBestNeuron = distByColNeuron.map(d => mapToNeuronDist(d._1,
d._2)).reduceByKey(minNeuronDist).map(d => (d._1, d._2.colNeuron)).collectAsMap()
      Array.tabulate(nbDataCol)(i => colsBestNeuron(i))
```

Passons à la dernière étape pour l'affectation des variables, dans cette partie on crée une nouvelle classe

```
 protected class NeuronDist(val colNeuron: Int, val dist: Double) extends Serializable
```

Chaque élement de **distByColNeuron** qui contient des couples clés-valeurs sera remplacer par (dataColId, new NeuronDist(colNeuron, dist)) d'ou l'utilisation de la fonction **map**.

De nouveau, on utilise le **reduceByKey** qui prend en paramètre une fonction calculant la distance et conserve donc le neurone ayant la distance minimale (auquel il sera affecté).
A ce moment la, nous avons plus besoin de stocker la distance puisqu'on a besoin de stocker que les affectations et c'est pour cela qu'on utilise le 2éme **map** qui ne conserve que l'indice de la colonne du prototype associé à la colonne traitée .

Cette étape correspond donc a la dernière partie d'affectation des colonnes de notre algo:

for each column reduce value $j \in J$ **do**
  $\phi_w(\mathbf{x}^j) = \arg\min(\langle (j, L); V \rangle)$
end for

Et donc finalement, on peut dire qu'on obtient un tableau qui stocke a chaque indice de colonne le prototype auquel il est affecté .
Par exemple si on a dans la première case de ce tableau 5 , cela veut dire que la colonne 0 est associé au prototype 5 .

2.3 Affectation des observations

Passons à la phase d'affectation des lignes, cette étape correspond au code suivant:

```
// Affectation des lignes et re-calcul du modèle
      _neuronMatrix = new BiTMRow(_neuronMatrix, _nbNeurons,
      _colNeuronAffectation, nbIter, iter).computeNewNeurons
```

Nous faisons appel à la classe **BiTMRow** qui a comme signature :

```
class BiTMRow(neuronMatrix: Matrix, nbNeurons: Int, colNeuronAffectation:
Array[Int], maxIter: Int, currentIter: Int) extends Serializable
```

Nous allons utiliser l'affectation des colonnes qui a été stocker dans un tableau d'entiers afin de calculer l'affectation des observations.

La première étape dans le code de **computeNewNeurons(qui retournera une matrice à la fin)** est:

```
val allObs = datas.map(d => mapperRowAffectation(d))
```

Donc pour chaque ligne de datas on va appliquer la fonction **mapperRowAffectation** :

```
protected def mapperRowAffectation(dataRow: Vector): Array[Array[ObsElem]] = {
      // process best neuron for the dataRow
      val rowBestN = findBestRowNeuron(dataRow)
      //println("row"+dataRow+" : neuron"+rowBestN)

      // pre-process tological dist
      val topoFactors = topoFactor.gen(maxIter, currentIter, nbNeurons)
      //topoFactors.foreach(f => println("- "+f))

      // Process observations
      val obs = Array.fill(nbNeurons, nbNeurons)(new ObsElem())
      for (c <- 0 until dataRow.length) {
        val colBestN = colNeuronAffectation(c)
        val elem = dataRow(c)

        //obs(rowBestN)(colBestN).add(elem, 1.0)
        // parcours et met à jour tous les éléments de la map
        for(i <- 0 until nbNeurons; j <- 0 until nbNeurons) {
          val factor = topoFactors(abs(rowBestN - i) + abs(colBestN - j))
          obs(i)(j).add(elem / factor, 1 / factor)
        }
      }
      obs
    }
```

Comme pour l'affectations des colonnes, on cherche au début le neuronne le plus proche en calculant une distance(on recupère donc son indice et on le stocke dans rowBestN).

Ensuite, Le facteur topologique est calculé à l'aide de la classe TopoFactor celui-ci dépend de l'itération courante, du nombre de neurones, de la distance entre les neurones de rangée et de colonne, ainsi que de la distance maximale dans la grille.

Comme on peut voir :

```
object BiTMTopoFactor extends TopoFactor {
  def gen(maxIter:Int, currentIter:Int, nbNeurons: Int) = {
    val T:Double = (maxIter - currentIter + 1) / maxIter.toDouble
    //val T = 0.9
    Array.tabulate(nbNeurons*2)(dist => exp(-dist / T))
  }
}
```

Revenons un peu sur l'algorithme,

{Assignment of rows }
for all $\mathbf{x}_i \in \mathbf{D}$ **do**
  $\langle(\mathbf{CM}, \mathbf{CN})\rangle = \text{RowMapper}(\mathbf{x}_i)$
end for

En résumer la fonction RowMapper permet de determiner pour chaque X le proptotype le plus proche, en effet cet étape permet de transformer les données en un ensemble de vecteurs compressées.

Algorithm 17 RowMapper($\mathbf{x}_i$)

{MAP rows distance: distributed loop over all input vectors ($\mathbf{x}_i = (x_i^1, x_i^2, ..., x_i^d)$) **}**
for each $k = 1..K$ **do**
  $bmu(k) = \| \mathbf{x_i} - \mathbf{g_k} \|^2$
end for
$\phi_z(\mathbf{x}_i) = \arg\min(\mathbf{bmu})$

ici $\phi_Z(x_i)$ renvoie l'indice du prototype le plus proche.

Après avoir appeler la fonction de voisinage, nous allons créer les vecteurs compréssser **CM** et **CN**

- CM : représente la somme des valeurs de chaque variable $X_i^j \in$ a la même ligne que x_i et qui a été affectées au même prototype que x_i.
- CN : quant a lui représente la somme des valeurs $\phi_Z(x_i)$ pour chaque $x_j \in$ a la même ligne que x_i.

CM et CN seront stocké dans le même objet **obs**: qui est un tableau de obsElem qui contient deux champs

- **value** :L'ensemble de value remplira la matrice CM
- **sumFactor** : L'ensemble de sumFactor remplira la matrice CN

Maintenant, le suite du code nous montre comment ces deux matrices seront remplies :

```
for (c <- 0 until dataRow.length) {
        val colBestN = colNeuronAffectation(c)
        val elem = dataRow(c)

        //obs(rowBestN)(colBestN).add(elem, 1.0)
        // parcours et met à jour tous les éléments de la map
        for(i <- 0 until nbNeurons; j <- 0 until nbNeurons) {
        //Cette distance échiquier n'est pas correcte .
          val factor = topoFactors(abs(rowBestN - i) + abs(colBestN - j))
```

```
        // La formule ici non plus .
          obs(i)(j).add(elem / factor, 1 / factor)
        }
      }
      obs
```

Remarque. il est important de noter que ce code correspond au code avant modification et quelques erreurs ont été soulevé , en l'occurrence la distance échiquier, l'utilisation de la variable factor, l'ensemble de ces changements ont étaient détaillés dans la section 4 **Modification du programme**.

L'objectif de cette partie était de faire correspondre les formules de la thèse au partie de code et comprendre le fonctionnement de l'algorithme.

Ce code correspond dans l'algorithme à la partie de création des vecteurs compressés et des matrices CM et CN.

Pour mettre à jour les vecteurs de prototype en utilisant l'expression définie dans l'équation:

$$g_r^l = \frac{\sum_{k=1}^{K} K^T(\delta(k,r)) \sum_{x_i^j \in B_k^l} \pi_r^l x_i^j}{\sum_{k=1}^{K} K^T(\delta(k,r)) \sum_{x_i^j \in B_k^l}}$$

Comme on peut le voir, on itère sur l'ensemble des éléments de la colonne (dataRow.length) et pour chaque élément de l'ensemble des colonnes,on stocke dans la variable **colBestN** l'indice du neuronne auquel il est affecté.
Après avoir fait cela on récupère la valeur de la colonne actuelle.
Cela nous sera utile pour la matrice CM .

Passons maintenant à la deuxième boucle, on itère sur l'ensemble des neurones, et on cherche le factor correspondant (la valeur de la fonction de voisinage en fonction des indices actuelles et les indices des lignes/colonnes les plus proches) et finalement on remplit la matrice obj avec **elem/factor** qui correspondra à la matrice CM et **1/factor** qui correspondra à la matrice CN .

Après avoir fait **rowMapper**, maintenant on doit faire **rowReducer**, qui se basera sur les résultats des deux matrices.

Remarque. A noté que CM et CN ont été calculé pour chaque ligne du dataset ! Donc on a plusieurs matrices CM et CN .

Pour ce qui s'agit du code, dans la variables **allObs** pour chaque ligne, on a stocké une matrice obsElem(Matrice CM et CN).
On fera ensuite la partie reduce qui consiste tout simplement à faire une somme des CM entre eux et des CN entre eux .

Ces résultats seront stocké dans **sumAllObs** qui correspondrai a CMs et CNs dans notre algorithme .

2.4 Phase de mise a jour

Pour finir ,la dernière étape est la mise à jour. Elle consiste tout simplement à diviser CMs par CNs, qui corresponderai dans le code à:

```
new Matrix(sumAllObs.map(row => row.map(_.compute())))
```

la fonction compute s'occupe de la division et le résultat est stocké dans une matrice comme on a pu le voir dans la dernière partie de l'algo BiTM :

```
{ Mise à jour des prototypes }
G = CMs/CNs
```

3 Parallélisme

3.1 BitM & MapReduce

Le paradigme de programmation MapReduce est actuellement l'une des solutions les plus adaptées au traitement des larges volumes de données

- Les fonctions de **Map** servent généralement à extraire l'information utile d'une partie des données. Elles fournissent en sortie un ou plusieurs couples de clé-valeur

- Les fonctions de **Reduce** sont généralement utilisées pour agréger les données en sortie de la fonction de Map. Elles ont en entrée une clé et l'ensemble des valeurs associées à cette clé.

Map et **Reduce** peuvent être exécutées de façon autonomes sur toutes les machines du cluster simultanément

La majorité des algorithmes d'apprentissage lisent plusieurs fois les données afin d'optimiser un paramètre or cette lecture des données est assez lente sur une architecture Hadoop. Le framework de traitement de données distribué Spark 2 résout ce problème en permettant de stocker les données en mémoire. Lors de l'exécution d'un algorithme d'apprentissage les données sont chargées en mémoire (depuis le disque) dès le début de l'apprentissage, par la suite elles seront relues depuis la mémoire. Ainsi, les gains de temps sont très significatifs lors des nombreuses itérations.

Le principe se résumerait comme suit pour notre algorithme :
itérer sur 2 phases de **MapReduce**

- l'une pour traiter les lignes
- l'autre pour traiter des colonnes
- puis a la fin de chaque itération on synchronise avec Reduce en mettant a jour les prototypes.

La plupart des fonctions de cet l'algorithme ont été implémentées en utilisant Map et Reduce, ce qui permet leur exécution simultanée sur toutes les machines du cluster Spark. En conséquence, l'algorithme est capable de se scaler efficacement, ce qui est un de ses avantages majeurs.

4 Modifications du programme

4.1 TopoFacotor

La formule pour le calcul du voisinage du code départ ne correspondait pas à celle de l'algorithme BiTM, celle-ci a dû donc être réécrite afin de correspondre a la formule suivante:

$$T = T_{max}(\frac{T_{min}}{T_{max}})^{\frac{t}{Iterations}}$$

Avec **t** l'itération actuelle et **itérations** le nombre total d'itérations .

Nous utilisons la fonction ci-dessous pour définir le voisinage :

$$K^T(\sigma(c_r, c_s)) = e^{\frac{-\sigma(c_r,c_s)}{T}}$$

La méthode devient donc:

```
object BiTMTopoFactor extends TopoFactor {
  def gen(maxIter:Int, currentIter:Int, nbNeurons: Int) = {
    val Tmin: Double = 0.9
    val Tmax: Double = 8
    val Tmp: Double = Math.pow(Tmin / Tmax, currentIter / maxIter)
    val T = Tmax *Tmp

    Array.tabulate(nbNeurons)(dist => { exp(-dist / T)
    })
```

T représente la température qui diminue graduellement de façon exponentiel en fonction de **Tmax** et **Tmin**, afin de contrôler la taille du voisinage qui influence une cellule donnée sur la carte

4.2 Mapper Row Affectation

La clé de l'algorithme BiTM est le calcul des distances on a donc implémenter d'une distance échiquier:

On a donc changer la fonction distance afin de calculer celle-ci entre le meilleur neurone et l'ensemble des clusters de tel sorte à respecter le formule:

abs(f(rowbestN) - f(i) +j(rowbestN)-g(i))

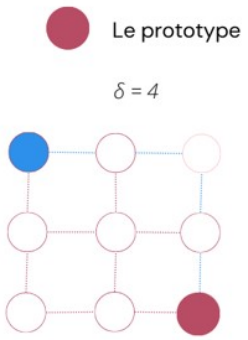
Avec **f** une fonction qui récupère les coordonnées en ligne et **g** une fonction qui récupère les coordonnées en colonne.

La partie du code qui correspond a cette étape est la suivante:

```
    protected def mapperRowAffectation(dataRow: Vector): Array[Array[ObsElem]] = {
     // Recherche du meilleur neurone pour chaque vecteur
     val rowBestN = findBestRowNeuron(dataRow)
```

```
    // Calcul de la distance avec la formule
    val topoFactors = topoFactor.gen(maxIter, currentIter, nbNeurons)

    val obs = Array.fill(nbNeurons, nbNeurons)(new ObsElem())
    for (c <- 0 until dataRow.size) {
      // val colBestN = colNeuronAffectation(c)
      val elem = dataRow(c)


      // parcours et met à jour tous les éléments de la map
      for (i <- 0 until nbNeurons ;j <- 0 until nbNeurons) {
        val line1 = getLine(rowBestN, nbRowNeuron) - getLine(i, nbRowNeuron)
        val col1 = getCol(rowBestN, nbRowNeuron) - getCol(j, nbRowNeuron)
        val factor = topoFactors(abs(line1) + abs(col1))
        obs(i)(j).add(elem * factor, factor)
      }
    }

    obs
  }
  protected def getLine(value : Int ,nbLine : Int) :Int  =
  {
    value/nbLine
  }

  protected def getCol(value: Int, nbCol: Int): Int = {

    value % nbCol
  }
```

On peut voir sur ce petit exemple que pour un réseau de 9 neurones la distance retourner par notre fonction entre le prototype et le premiers neurone est de $\delta = 4$



4.3 Clusters Colonnes

L'exécution du code de départ prend un nombre de clusters prédéfini entre lignes et colonnes. L'affectation aléatoire pour chaque colonne pouvait donc aller de 0 à nombre de clusters . Idem pour les lignes.

Comme il n'y pas forcément de raison pour que le nombre de cluster soit le même entre les lignes et les colonnes et pour éviter d'avoir des clusters vides sur les colonnes on a du :

Créer une nouvelle variable pour spécifier le nombre de clusters sur les colonnes de tel sorte que l'affection aléatoire se fasse entre 0 et cette nouvelle variable.

4.4 Clusters Lignes

Quant à l'affectation des lignes
Le code de départ bouclait sur le nombre de neurones sur les lignes pour rechercher le meilleur neurone or ceci ne correspond pas à l'algorithme BiTM .

De plus cela avait engendré des résultats peu concluants .

On a donc changé la boucle de telle sorte que la recherche du meilleur neurones se fasse en parcourant le nombre de neurones plutôt que le nombre de neurones sur les lignes.

4.5 Run programme

Une fonctionnalité supplémentaire ajouté à notre programme est lors du lancement de celui-ci
En effet on a le choix entre:

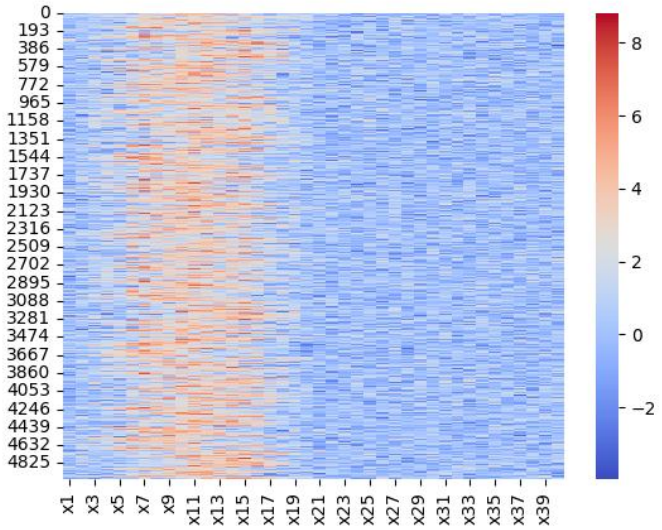
- Un lancement Random avec un choix aléatoire de prototype au début de l'algorithme.
- Un lancement récupère les résultats d'une précédente exécution.

```
$ ./run.sh use_saved
```

5 Interprétation des résultats

Nous avons d'abord commencer par visualiser notre Dataset **Waveform** de départ avant passage dans l'algorithme.
Waveform est composé de 40 colonnes et que 5000 lignes
la colonne Class a été ignoré.

La visualisation est la suivante:



Sur plusieurs exécution, nous avons visualiser la sortie de BiTM donc l'organisation des données Comme dit auparavant Affectation réorganise le Dataset en fonction de l'affectation des lignes et des colonnes afin de regrouper les attributs qui se ressemblent le plus et le retourne .

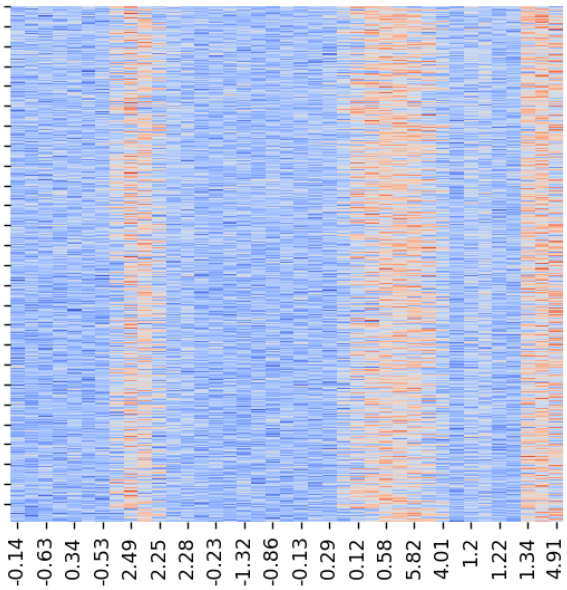


Figure 1: Dataset organisé en 3 clusters

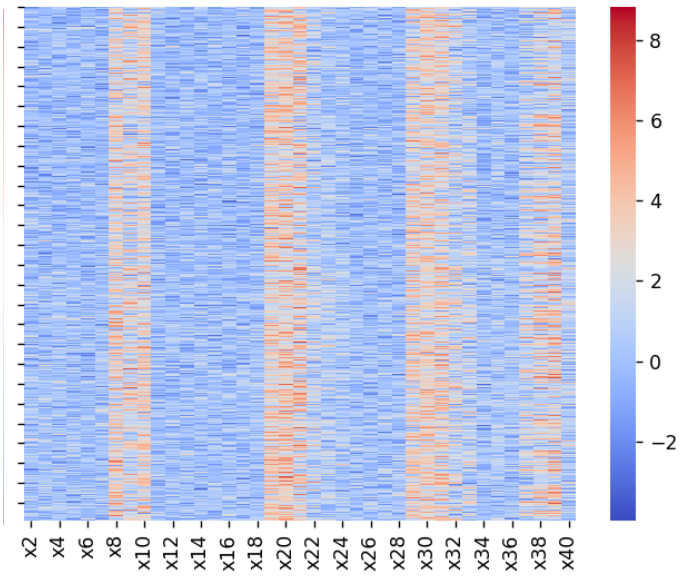
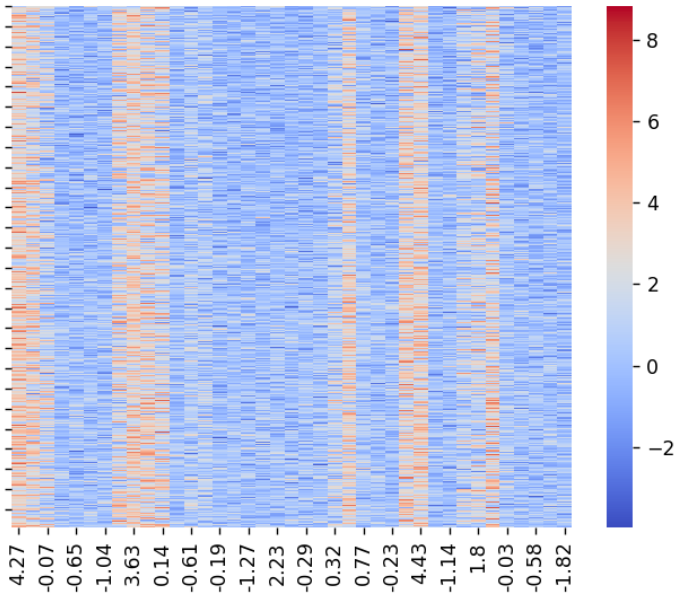


Figure 2: Dataset organisé en 4 clusters

Et la dernière figure pour une organisation du dataset en 5 clusters.

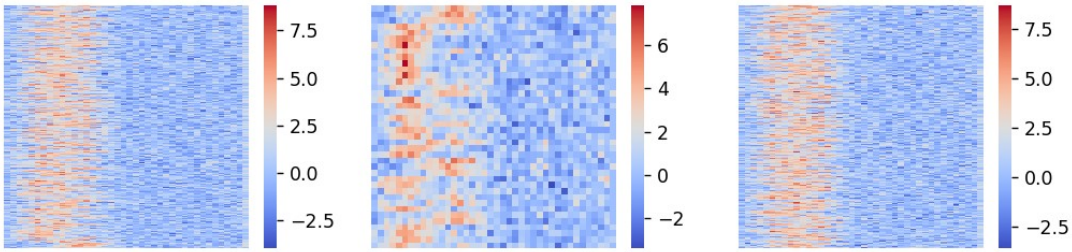


La deuxième partie des visualisations était :
la visualisions de **AffectationS** qui gère l'affectation de chaque ligne au cluster le plus proche.

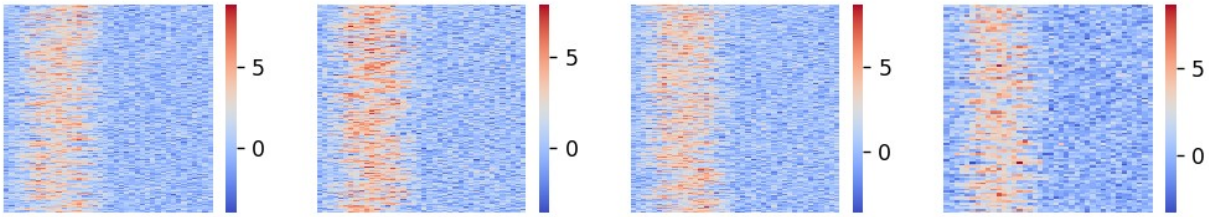
Ici chaque ligne sera affecté à une sous carte : valeur variante de 0 au nombre de clusters .

voici certains de nos résultats:

Pour le Dataset Waveform.
Le nombre de clusters choisi est 9
Le nombre d'itérations est 100



Pour le même Dataset.
Le nombre de clusters choisi est 16
Le nombre d'itérations est 100



6 Score de pureté

Pour mesurer la qualité de notre algorithme, nous calculons des indices

6.1 ACC

L'exactitude **ACC** : mesure la proportion de données correctement attribuées à leur classe d'origine par rapport à toutes les données.

6.2 NMI

L'information mutuelle normalisée **NMI**: mesure la quantité d'information partagée entre les clusters trouvés par l'algorithme BiTM et les classes réelles des données.

Les ACC et NMI sont compris entre 0 et 1 , où 0 indique une classification aléatoire et 1 la correspondance parfaite.

6.3 ARI

L'indice de Rand ajusté **ARI** : mesure la similitude entre les clusters trouvés par l'algorithme BiTM et les classes réelles des données. Il est compris entre -1 et 1, où

- 1 indique une correspondance parfaite entre les clusters et les classes réelles,
- 0 indique une correspondance aléatoire,
- -1 indique une correspondance complètement opposée.

6.4 RI

Le RI mesure la similarité entre deux partitions de clustering en calculant le nombre de paires d'observations qui sont affectées à la même classe dans les deux partitions (vrai positif), ainsi que le nombre de paires d'observations qui sont affectées à des classes différentes dans les deux partitions (vrai négatif).
Le RI varie de -1 (pas de similitude) à 1 (similitude parfaite).

Nous avant implémenter un code en python qui calcule les indices à partir de l'output de notre algorithme
les résultats obtenus sont les suivants:

```
ARI =  0.0005084170051013902
NMI =  0.0008424056674870993
ACC =  0.1564
RI =  0.5189929985997199
```

7 Conclusion

Plusieurs axes d'améliorations et autres modifications reste possible pour ce projet:

- Faire un co-clustering multi-vues

  Le co-clustering multi-vues est une technique de clustering qui permet de regrouper simultanément plusieurs vues ou aspects d'un même ensemble de données. Contrairement au clustering classique qui se concentre sur un seul aspect ou une seule variable, le co-clustering multi-vues considère plusieurs aspects ou variables pour la partition des données.

  Dans le co-clustering multi-vues, chaque vue peut être vue comme une matrice de données, où chaque ligne représente une observation et chaque colonne représente une variable l'objectif est de regrouper les observations similaires en groupes ou clusters, tout en préservant les relations entre les différentes vues.

  Le co-clustering multi-vues est une technique de clustering puissante qui permet de combiner des informations provenant de plusieurs vues pour obtenir une partition conjointe des données qui préserve les relations entre les différentes vues.

- DataSet ALOI: Faire correspondre les clusters et retrouver les objets associés .

- Passer sur des versions Scala/spark plus récentes et plus performantes
- Approfondir l'étude sur la possibilité d'introduire plus de réduction et donc améliorer les performance de l'algorithme
- Approfondir l'études des scores pour évaluer au mieux l'algorithme
- Faire un dashboard englobant l'ensemble des visualisations et analyses effectuées pour un dataset .

8 Version

Les versions utiliser pour ce projet sont:

- Scala: 2.12
- Spark: 2.4.3
- java: 8

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <encoding>UTF-8</encoding>
    <scala.version>2.12.4</scala.version>
    <scala.compat.version>2.12</scala.compat.version>
    <scala.binary.version>2.12</scala.binary.version>
    <spark.version>2.4.3</spark.version>
    <spec2.version>4.2.0</spec2.version>
```

Après avoir tester l'algorithme on a remarquer que de meilleurs performances sont obtenu en utilisant:

- scala: 2.13.10
- spark: 3.3.2
- java: 17

En effet avec ces version un simple build sur un 2x2 prend 5 min, alors que la même manipulation prend 42 secondes avec les versions les plus récentes

Un axe d'amélioration de ce projet serait donc de passé sur les versions les plus récentes de Spark/Scala

9 Bibliographie

- Apprentissage massivement distribué dans un environnement Big Data: Tugdual Sarazin.
- Self Organization Maps Network Architecture
- https://github.com/unsupervise/NPLBM
- https://github.com/unsupervise/ACOMI
- Jeu de données images: https://aloi.science.uva.nl/