Datamining

Glossaire

Xavier Dubuc
(xavier.dubuc@umons.ac.be)

28 novembre 2018

Table des matières

1 Classification

↪ *voir fichier txt*

2 Règles d'association

2.1 Introduction

Le support d'une règle $X \to Y$ est s si $s\%$ des transactions contiennent les élements de $X \cup Y$.

La confiance d'une règle $X \to Y$ est c si $c\%$ des transactions qui contiennent X contiennent aussi Y.

Une règle d'association est une expression implicative de la forme $X \to Y$ où X et Y sont des ensembles disjoints. La puissance d'une règle peut être mesurée en terme de **confiance** et de **support**.

Le problème "Association Rule Discovery" consiste en : pour un set de transactions T, trouver toutes les règles ayant un support $\geq$ minsup et une confiance $>$ minconf. (avec minsup proche de 0.1 en général et minconf proche de 1 en général, ces 2 bornes étant données)

Afin d'éviter la génération de règles inutiles, on va générer des itemsets et la première application que l'on peut faire est que si on a un itemset $\{A, B, C\}$ qui est pas assez 'supporté', alors toutes les règles $A \to B$, $A \to C$, $AB \to C$, ... ne le seront pas non plus, et donc on peut directement les oublier.
On procède donc de la manière suivante :

1. Trouver les frequent itemset (ceux qui ont un support suffisant)
2. Extraire les strong rules des frequent itemset, c'est-à-dire les règles ayant une haute valeur de confiance.

La partie 1. reste exponentielle, on entrevoit 2 solutions :

A. réduire le nombre d'itemset candidats à être frequent itemset (FI) (**apriori** utilise cette approche),

B. réduire le nombre de comparaisons en utilisant des structures plus avancées. (cf plus loin)

2.2 Apriori

Le principe apriori se base sur le théorème et le corollaire suivant :
Théorème : Si un itemset est fréquent, alors tous ses sous-ensembles doivent l'être aussi.
Corollaire : Si un itemset est non-fréquent alors tout superset le contenant ne l'est pas non plus.
(superset = set plus grand)
L'algorithme va utiliser ces 2 principes pour pruner l'arbre/le graphe, on appelle ça le **support-based pruning**. Ceci est rendu possible car le support est anti-monotone, cad support(A) n'est jamais plus grand que le support des sous ensembles de A.

Generation des FI dans l'algorithme apriori
On calcule le support de chaque élément isolé et on "supprime" ceux qui n'ont pas un support assez grand. On itère avec les 2-itemset créés à partir des 1-itemset restants, et ainsi de suite jusqu'à ne plus en trouver.

→ Cet algorithme est "level-wise", c'est-à-dire qu'il traverse le treillis des itemsets niveau par niveau des 1- itemset jusqu'à la taille maximale des frequent itemsets.

→ Il emploie une strategie "generate-and-test", c'est-à-dire qu'il génère à chaque itération les candidats puis il calcule pour chaque le support et supprime ceux qui ne sont pas assez supportés.

⇒ $O(k_{max} + 1)$ où k_{max} est la taille maximale des frequent itemsets.

2.3 Génération de candidats et pruning

L'algorithme apriori fait appel à 2 opérations (qui n'accèdent pas à la DB) :
* générer des candidats,
* pruner/élaguer les candidats

Mettons en lumière cette dernière opération, on considère un k-itemset X tel que $X = \{i_1, i_2, \ldots, i_k\}$. L'algorithme doit déterminer si tous les sous-ensembles de X, $X - \{i_j\}$ $(\forall j = 1, \ldots, k)$ sont fréquents et si l'un d'entre eux ne l'est pas, X est élagué directement. Cette opération se fait en $O(k)$ par itemset de taille k et même en $O(k - m)$ en fait avec m le nombre d'itemsets de taille $k - 1$ utilisés pour générer l'itemset de taille k considéré. (logique vu qu'il ne faut plus que vérifier que les $k - m$ autres sont fréquents eux aussi)
Pour générer les itemsets de manière efficiente il faudrait :

1. éviter de générer trop d'itemsets candidats inutiles,
2. assurer que l'ensemble des candidats est complet,
3. un itemset ne doit être générer qu'une seule fois

On peut soit tout générer et puis faire du pruning mais complexité exponentielle alors on utilise plutot une récursivité :

$$\text{frequent } k\text{-itemset} = \text{frequent } (k-1)\text{-itemset} \cup \text{frequent } 1\text{-itemset}$$

Malgré cela, ça se peut qu'on génère plusieurs fois le même itemset (par exemple $\{a, b, c\} = \{b, c\} \cup \{a\} = \{a, b\} \cup \{c\}$) mais on peut éviter ça en gardant les itemsets triés par ordre lexicographique et en ne générant que les k-itemsets tels que le $(k-1)$- itemset est avant le 1-itemset dans l'ordre lexicographique.
⇒

```
{Bread,Diaper} + {Milk} OK
{Bread,Milk} + {Diaper} NON
{Diaper,Milk} + {Bread} NON
```

Cette approche n'est pas encore parfaite car elle génère tout de même quelques itemsets inutiles. $Bread$, $Diaper$, $Milk$ est généré avec $\{Bread, Diaper\} + \{Milk\}$ alors que $Bread$, $Milk$ n'est pas frequent. Plusieurs heuristiques existent pour éviter ça comme par exemple le fait que chaque élément d'un k-itemset fréquent doit apparaitre dans **au moins** $k - 1$ $(k-1)$-itemset fréquents sinon le k-itemset n'est pas fréquent.

↪ L'approche réellement utilisée par **apriori** est de fusionner des $(k-1)$-itemsets et ce, uniquement si leur $(k-2)$ premiers éléments sont identiques (et seulement eux). Il faut cependant s'assurer que les $(k-2)$ autres sous-ensembles sont également fréquents.

Compter le support count des itemsets
La manière naïve de le faire est de regarder chaque itemset généré et de les comparer à chaque transaction afin de compter le support count. On peut également générer tous les itemsets contenus par une transaction et augmenter le support de chacun d'entre eux qui sont candidats fréquents.
Il faut donc comparer les itemsets générés aux itemsets candidats afin d'incrémenter le support des candidats matchés. On peut le faire avec un arbre de hashage. → **Voir Figure 6.11 page 19**.

2.4 Rule Generation

Chaque itemset fréquent peut générer $2^k - 2$ règles différentes (on ignore les règles avec un ensemble vide). Une règle peut se générer en partitionnant l'itemset en 2 ensembles $non-vides : X \subseteq itemset \ : \ X \rightarrow Y - X$, il faut cependant que cette règle satisfasse au seuil de confiance. Pour calculer la confiance de la règle, il n'est requis aucun calcul supplémentaire car c'est un ratio de support count qui ont été calculés précédemment (vu que les sous ensembles des itemsets fréquents sont des itemsets fréquents)

Pruning basé sur la confiance On a pas de règle d'antimonotonie mais on a tout de même le théorème suivant :

Si une règle $X \rightarrow Y - X$ ne satisfait pas le seuil de confiance, alors pour tout $X' \subseteq X$ la règle $X' \leftarrow Y - X'$ ne la satisfait pas non plus.

Génération de règles dans l'algorithme Apriori C'est un algorithme level-wise, même style de principe que pour les frequent itemsets. On génère les règles avec le plus d'éléments à gauche de la fleche puis, si la règle n'a pas assez de confiance on abandonne la génération à partir de cet itemset. Sinon, si $\{a, b, c\} \rightarrow d$ et $\{a, b, d\} \rightarrow c$ ont une confiance assez élevée on teste $\{a, b\} \rightarrow \{d, f\}$.

2.5 Représentation compacte des frequent itemsets

Il est important de pouvoir identifier un sous-ensemble de frequent itemsets représentatifs desquels on peut déduire tous les autres frequent itemsets.2 représentations sont données :

1. **Maximal frequent itemsets**
   Un **maximal frequent itemset** est un itemset frequent tel qu'aucun de ses supersets directs *("fils" dans le treilli)* n'est fréquent. C'est l'ensemble le plus petit d'itemsets fréquents tels que tous les autres itemsets fréquents peuvent en être dérivés. Le problème de ceci est que on perd les informations du support sur les sous-set de ces maximal frequent itemsets. Parfois il est préférable d'avoir une représentation un peu plus "grande" mais qui conserve ces informations afin de ne pas devoir les recalculer à nouveau.
2. **Closed frequent itemsets**
   Un frequent itemset est un **closed frequent itemset** si aucun de ses supersets immédiats n'a le même support count que lui. Vu autrement, un frequent itemset n'est pas closed si au moins un de ses supersets immédiats a le même support que lui.
   Aussi, une règle $X \rightarrow Y$ est **redondante** s'il existe $X' \subseteq X$ et $Y' \subseteq Y$ tels que $X' \rightarrow Y'$ avec le même support et la même confiance. Ce genre de règle ne sont pas générées si les closed frequent itemsets sont utilisés pour la génération.

2.6 Méthodes alternatives pour générer des FI

Même si apriori améliore de manière considérable les performances pour générer les règles d'associations, il souffre toujours d'un overhead I/O assez important car il faut passer plusieurs fois sur la BDD des transactions. La génération des FI peut être vue comme un traversée du treillis des itemsets. Plusieurs manières peuvent être envisagées pour la traversée :

- ⋆ *general-to-specific* (de haut en bas) (utilisé par Apriori)
- ⋆ *specific-to-general* (de bas en haut), utile pour trouver des itemsets très spécifiques dans des transitions denses, la limite des "FI" est localisée vers le bas du treillis.
- ⋆ *bidirectionnal* : combinaison des 2 précédentes ⇒ recquiert plus d'espace de stockage.
- ⋆ *classes d'équivalence* : on pourrait imaginer que la recherche se fait d'abord dans une classe particulière avant de se poursuivre dans une autre. Par exemple dans apriori les classes d'équivalence sont définies au travers de la taille des itemsets. On peut aussi les définir selon le préfixe ou le suffixe des itemsets. Dans ces 2 derniers cas on utilise un arbre de préfixe/suffixe.
- ⋆ *DF / BF (profondeur/largeur)* : apriori utilise un parcours en largeur, on pourrait alors imaginer un parcours en profondeur, l'avantage est qu'on trouve les maximal FI plus rapidement en trouvant la bordure de FI plus rapidement.

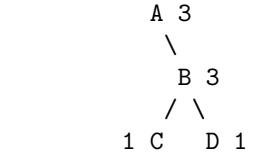
La représentation du data set de transactions (cette représentation a de l'influence sur le coût d'I/O) peut être vue de 2 façons :

1. **horizontale** : transaction associée à une liste d'item
2. **verticale** : item associé à une liste de transactions contenant l'item

Ces 2 représentations demandent beaucoup de mémoire, il existe une meilleure manière de faire, développée dans la section suivante.

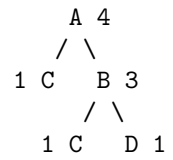
2.7 FP-Growth Algorithm

Représentation On utilise une structure de données plus compacte : le **FP-Tree**. C'est un peu comme un prefix-tree où on compte le nombre de chemins matchés.



Ce **FP-tree** signifie qu'on a 3 transactions contenant A et B et 1 contenant C et 1 contenant D, on a donc le set de transactions suivant : $\{AB, ABC, ABD\}$.
Dans le **FP-tree** on ajoute également des liens entre tous les noeuds correspondant des liens entre tous les

noeuds correspondant à un même item pris dans des transactions différentes (afin de compter rapidement le support count)



Ici, pareil qu'avant sauf qu'on a $\{AC\}$ comme transaction supplémentaire. On met alors un lien entre les deux C afin de compter rapidement que C est supporté $1 + 1 = 2$ fois.

On considère que chaque transaction est triée dans l'ordre du support global, dans l'exemple :

$$A > B > C > D \text{ (en support count } 4 > 3 > 2 > 1))$$

Generation des frequent itemsets dans l'algorithme FP-Growth (CF. FIGURES DU PDF) On va explorer le **FP-Tree** du bas vers le haut. Dans l'exemple précédent, l'algorithme va d'abord rechercher les itemsets fréquents se terminant par e, puis par d, c, b et a. Cette façon de faire est équivalente à la façon suffix-based vue plus tôt. On considère que l'itemset $\{e\}$ est fréquent, dès lors l'algorithme doit résoudre les sous-problèmes des frequents itemsets finissant par de, ce, be et ae. On construit alors le **conditional FP-Tree** de e, c'est-à-dire qu'on enlève les transactions qui ne contiennent pas e, on enlève les noeuds contenant e (ils sont inutiles à présent). A présent cet arbre reflète les transactions existantes contenant à la fois les éléments mentionnés et e, il se peut alors que certains éléments ne soient plus fréquents (comme b qui a un support count de 1 (et comme on avait pris un support count minimum de 2)). On applique l'algorithme pour trouver les frequent itemsets finissant par d dans le conditionnal tree de e et on trouve ainsi les frequent itemsets se terminant par de. (et ainsi de suite)

3 Clustering

3.1 Différents types de clustering (pages 5-7)

Hierarchical clustering création d'une hiérarchie de clusters avec une relation d'inclusion entre les différents niveaux, avec un nombre de clusters variant de 1 à k.

Partitional clustering partition des données en k sous-ensembles (clusters).

Complete clustering aucun point n'est ommis.

Partial clustering certains points sont ommis (ils sont considérés comme du bruit).

3.2 Différents types de clusters (pages 7-9)

Well-separated un cluster est défini comme un ensemble d'objets dans lequel chaque objet est plus proche de chacun des autres objets du cluster que de n'importe quel autre objet n'étant pas dans le cluster.

Prototype-based un cluster est défini comme un ensemble d'objets qui est plus proche du prototype représentant le cluster que de tout autre prototype.

Graph-based un cluster est défini comme une composante connexe (c'est-à-dire une partie graphe où d'un noeud on peut aller à tous les autres). Ceci assume le fait que les données sont définies comme des noeuds de graphes reliés entre eux.

Density-based un cluster est défini comme une région dense d'objets entourée par une région de faible densité.

Shared-propriety un cluster est un ensemble d'objets qui partagent la même propriété.

3.3 K-means (pages 10-29)

Crée un partitional, prototype-based clustering dans lequel les prototypes sont les centroïdes des clusters (la moyenne des données en général). L'idée est simple, elle consiste à choisir K points comme centroïdes initials, assigner ensuite les points au prototype le plus proche et recalculer les centroïdes et ensuite itérer jusqu'à ce que les centroïdes ne soient plus modifiés.

Symbol	Description
x	un objet
C_i	le $i^{\text{ème}}$ cluster
c_i	le centroïde du cluster C_i
c	le centroïde de tous les points
m_i	le nombre d'objets dans le $i^{\text{ème}}$ cluster
m	le nombre d'objets dans le dataset
K	le nombre de clusters

Le but du clustering est de minimiser une fonction objectif. Dans le cas de données dans des espaces euclidiens, on peut prendre la *Sum of the Squared Error* **(SSE)** ou appelé également *scatter*.

$$\boxed{SSE = \sum_{i=1}^{K} \sum_{x \in C_i} dist(c_i, x)^2}$$

Ainsi, le centroïde du $i^{\text{ème}}$ cluster est alors défini par :

$$\boxed{c_i = \frac{1}{m_i} \sum_{x \in C_i} x}$$

c'est à dire la moyenne du cluster, comme dit plus haut. (en utilisant la distance euclidienne, pour la distance de Manhattan par exemple il s'agit de la médiane du cluster)

Pour la sélection des centroïdes initiaux on peut faire ça de manière aléatoire ou via le **farthest-first traversal**, c'est-à-dire en choisissant un point au hasard comme centre et puis prendre le centre le plus loin de ce point, ensuite le point le plus loin des 2 sélectionnés jusque là, etc...

Améliorer k-means via postprocessing On peut identifier le cluster ayant la SSE la plus grande et le splitter en 2 et ensuite fusionner 2 autres qui augmentent le moins le SSE (ceux dont le centroides sont les plus proches). On peut également ajouter un nouveau centroïde ou disperser un cluster en supprimant son centre et réassignant les points aux autres clusters.

Bisecting k-means (page 22-24) Cela consiste à séparer un cluster "au choix" en 2 à chaque étape (en commençant avec le cluster contenant tous les points) jusqu'à obtenir k clusters.

3.4 Agglomerative Hierarchical Clustering (pages 29-40)

On peut faire des clustering hiérarchique en partant du k-clustering avec les clusters ne contenant qu'un élément et au fur et à mesure les fusionner ensemble pour arriver au 1-clustering. On peut également partir du 1-clustering pour descendre au k-clustering en divisant des clusters en 2.

Dans la manière "agglomerative", à chaque itération on fusionne les 2 clusters les plus proches et on met la matrice des distances à jour.

Proximité entre clusters (page 31)

— **single-link**, la distance entre les clusters C_i et C_j est donnée par :

$$\min_{x \in C_i,\ y \in C_j} proximity(x, y)$$

— **complete-link**, la distance entre les clusters C_i et C_j est donnée par :

$$\max_{x \in C_i,\ y \in C_j} proximity(x, y)$$

— **group average**, la distance entre les clusters C_i et C_j est donnée par :

$$\left(\frac{1}{m_i + m_j}\right) \sum_{x \in C_i} \sum_{y \in C_j} proximity(x, y)$$

On peut également définir le proximité entre 2 clusters, dans le cas d'un clustering prototype-based, comme la distance entre les centroïdes de ces clusters. La méthode de **Ward** permet de faire quelque chose de ce genre en exprimant la distance entre 2 clusters en terme d'augmentation de **SSE** en cas de fusion entre ces 2 clusters.

3.5 DBSCAN (pages 40-46)

Density based clustering . La densité est définie via l'approche "center-based", c'est-à-dire que la densité pour un point est le nombre de points se situant dans un rayon ϵ défini.
↪ On peut alors classifier les points selon leur densité :

— core point : point vraiment à l'intérieur d'un cluster,
— border point : point sur le bord d'un cluster,
— noise point : point à ignorer car ne fait pas partie d'un cluster (le clustering density-based est partial).

Algorithme L'algorithme **DBSCAN** est alors simple, il consiste à labeller les noeuds comme noise, border ou core point, d'éliminer ensuit les noise points, on connecte ensuite chaque core point se trouvant à moins de ϵ de distance (données représentées dans un graphe). On considère ensuite tous les noeuds connectés ensemble comme un seul cluster. Finalement on assigne les border points au cluster le plus proche.

Définir ϵ L'idée est de prendre pour ϵ la distance du $k$ème voisin le plus proche (la "k-dist").
(en général $k = 4$)

Caractéristiques **DBSCAN** permet de trouver des clusters que k-**means** serait incapable de trouver, mais il peut se réveler très mauvais dans le cas de clusters de densités très variables.

3.6 Cluster Evaluation (pages 46-69)

" Just as people can find patterns in clouds, data mining algorithms can find clusters in random data "

3.7 Unsupervised (using Cohesion et Separation)

Nous cherchons à donner une valeur à la validité d'un cluster et on définit la validité d'un clustering comme :

$$\sum_{i=1}^{K} w_i\ validity(C_i)$$

où w_i est un poids associé au cluster C_i. La fonction $validity$ peut, dans ce chapitre, être définie comme la cohésion ou la séparation.

graph-based

$$cohesion(C_i) = \sum_{x,\ y \in C_j} proximity(x, y)$$

$$separation(C_i, C_j) = \sum_{x \in C_i,\ y \in C_j} proximity(x, y)$$

prototype-based

$$cohesion(C_i) = \sum_{x \in C_i} proximity(x, c_i)$$

$$separation(C_i, C_j) = proximity(c_i, c_j)$$

$$separation(C_i) = proximity(c_i, c)$$

Lorsque la proximité est la distance euclidienne au carré

$$cohesion(C_i) = SSE(C_i)$$

$$separation(C_i) = SSB(C_i) = dist(c_i, c)^2$$

Et on a également :

$$TOTAL\ SSB = \sum_{i=1}^{K} m_i dist(c_i, c)^2$$

$$SSB + SSE = constante = TSS\text{(Total Square of Sum)}$$

Silhouette Coefficient Pour le $i$ème objet, soient :

$$a_i = \text{ distance moyenne par rapport à tous les autres objets du cluster}$$

$b_i =$ distance moyenne par rapport à tous les objets contenus dans un autre cluster, le cluster le plus proche alors le coefficient silhouette de i est :

$$s_i = \frac{b_i - a_i}{max(a_i, b_i)}$$

Plus il est proche de 1 plus il est "pur" c'est-à-dire que a_i est proche de 0 et que donc l'objet a une très bonne cohésion. Dès qu'il est négatif, il y a un problème car cela signifie que $a_i > b_i$ ce qui veut dire que l'élément n'est pas dans le bon cluster.

3.8 Unsupervised (using proximity matrix)

Correlation between similarity matrix and ideal similarity matrix La matrice idéale contient un 1 entre deux objets appartenant au même cluster et 0 autrement, en groupant les éléments par cluster, cette matrice a une structure en blocs de 1. Pour la matrice "actuelle" de similarité on peut, par exemple, modifier les distances comme suit :

$$s_{ij} = \frac{d_{ij} - \min_{i,j} d_{ij}}{\max_{i,j} d_{i,j} - \min_{i,j} d_{ij}}$$

3.9 Unsupervised for hierarchical

The cophenetic distance la distance cophenetic entre 2 objets est la proximité à laquelle un algorithme agglomératif de clustering hiérarchique met ces 2 objets dans le même cluster.

The cophenetic correlation coefficient (CPCC) est la corrélation entre la matrice des distances cophenetics et la matrice original de dissimilarité.

3.10 Deviner le nombre de clusters

On peut regarder sur un graphique refletant l'évolution de la mesure d'évaluation du cluster en fonction du nombre de cluster là où il y a un perturbement significatif (une chute, un pic, ...).

3.11 Clustering tendency

Il s'agit de deviner si un jeu de données à des clusters ou pas. L'idée est de générer des points aléatoirement et de sommer les distances vers le plus proche voisin de chacun de ces points. On divise cette somme ensuite par cette même somme à laquelle on ajoute la somme des distances vers le plus proche voisin des points du jeu de données. En fonction du nombre obtenu, on est à même de dire qu'il y a de fortes chances qu'il y ait des clusters ou non. Si ce coefficient est proche de 1, alors le jeu de données est fortement clusterisé (car la somme des distances vers le plus proche voisin n'influence pratiquement pas le quotient. Dans le cas contraire, une valeur proche de 0 indique que cette somme influence particulièrement ce quotient et que donc elle est important ; cela signifie que les points sont plutôt assez bien disséminés.

3.12 Supervised

classification-oriented Identique à la classification avec entropy et autres, juste que les labels prédits sont ceux assignés aux clusters.

similarity-oriented On compare la matrice idéale de clustering, c'est-à-dire 1 pour chaque paire d'éléments d'un même clustering, 0 sinon à la matrice idéale des classes dans laquelle on a 1 en ij si i et j ont la même classe, 0 sinon. On peut alors calculer une corrélation ou utiliser la Rand statistic et le coefficient de Jacquard.

Rand statistic et le coefficient de Jaccard On définit :

f_{00} comme le nombre de paire d'objets de classe différente et de cluster différent

f_{01} comme le nombre de paire d'objets de classe différente et de cluster identique

f_{10} comme le nombre de paire d'objets de classe identique et de cluster différent

f_{11} comme le nombre de paire d'objets de classe identique et de cluster identique

Alors , la Rand statistic est définie comme :

$$Rand\ statistic = \frac{f_{00} + f_{11}}{f_{00} + f_{01} + f_{10} + f_{11}}$$(càd la proportion d'objets "bien classés")

et le coefficient de Jacquard :

$$Jaccard = \frac{f_{11}}{f_{01} + f_{10} + f_{11}}$$(idem, sans tenir compte des objets de classes différentes))

3.13 Cluster Validity for Hierarchical Clustering

On considère TOUS les clusters (donc de TOUS LES NIVEAUX) comme un seul ensemble de clusters. On va calculer le fait qu'au moins un cluster est relativement pur, c'est-à-dire qu'il ne contient que des éléments (ou presque) d'une seule et même classe. Pour ce faire on calcul les F-mesures de tous les clusters et pour chaque classe on prend le maximum des F-mesures atteinte par n'importe quel cluster. On prend ensuite une moyenne pondérée de ces F-mesures et cela nous donne la validité du clustering hiérarchique.

$$F - mesure(i,j) = \frac{(2precision(i,j)recall(i,j))}{(precision(i,j) + recall(i,j))}$$

$$F_{total} = \sum_j \frac{m_j}{m} \max_i F(i,j)$$

4 Dasgupta & Long

Le clustering qu'ils effectuent cherche à minimise le rayon maximum d'un cluster (et donc la distance maximale entre 2 éléments d'un même cluster). Cet algorithme, qui en fait génère un arbre T, se base sur une première étape consistant à effectuer un "farthest-first traversal" (**FFT**). Ce FFT consiste à choisir un point aléatoirement et d'ensuite sélectionner le point le plus loin du premier choisi, puis celui qui est le plus loin de tous les points déjà sélectionnés jusque là, etc. A chaque itération on relie le point j que l'on vient de sélectionner au point i des points déjà sélectionnés auparavant le plus proche de lui et on note la distance entre ces 2 points R_j.

Ensuite on divise l'arbre en "niveau de granularité", ces niveaux sont des sous ensembles d'éléments définis comme suit :

$$L_0 = \{1\}$$
$$L_j = \left\{ i \mid \frac{R}{2^j} < R_i \leq \frac{R}{2^{j-1}} \right\} \ \forall j > 0$$

(où $R = R_2$, la distance maximale dans les R_i)

On relie ensuite les éléments de la manière suivante : chaque noeud i est relié à son voisin le plus proche à un niveau inférieur. On obtient alors un nouvel arbre que l'on nomme T'. On ordonne les arêtes de cet arbre de manière décroissante et on renomme ces arêtes $R'_2 > R'_3 > \dots R'_{k+1}$.

Finalement on obtient un k-clustering en enlevant dans l'ordre les $(k-1)$ arêtes de l'arbre (donc si on veut un 2 clustering, on enlève l'arête R'_2, pour un 3-clustering on enlève R'_2 et R'_3, etc).