Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'électronique et d'informatique
Département d'informatique

2ème Année Master Informatique, Semestre 1

Option : Systèmes informatiques Intelligents (SII)

Module : Data Mining

Rapport
Techniques de DataMining

Monôme :
- OUHOCINE Sarah

Professeur :
- Mme BABA ALI

Année universitaire 2019/2020

SOMMAIRE

INRODUCTION GENERALE

Le *Data Mining* est en fait un terme générique englobant toute une famille d'outils facilitant l'exploration et l'analyse des données contenues au sein d'une base décisionnelle de type Data Warehouse ou DataMart. Les techniques mises en action lors de l'utilisation de cet instrument d'analyse et de prospection sont particulièrement efficaces pour extraire des informations significatives depuis de grandes quantités de données.

Pour mener à bien un projet de *Data Mining*, il faut évidemment d'abord définir clairement la problématique à étudier. Ensuite, il est crucial de sélectionner parmi l'ensemble des données disponibles, celles qui pourront être utilisées. C'est-à-dire celle dont la qualité ne laisse aucune place au doute, par exemple. Le tout en s'assurant que le nombre de données exploitées reste en corrélation avec la complexité du problème traité.

Plus le problème est complexe, plus il faudra de données. Vient alors l'étape de paramétrage du modèle construit à partir de techniques issues des méthodes statistiques, des analyses de données et de l'informatique. L'objectif peut être d'extrapoler de nouvelles données à partir d'une base, de mettre en évidence des données existantes noyées dans la masse ou de réduire la masse des données. Enfin, il faut procéder à l'étude des résultats.

Dans cette deuxième partie du projet Data Mining nous allons mettre en application certaines techniques de datamining vues en cours. Pour ce faire on va exploiter le data set Heart_stat.dat déjà étudié dans la première partie du projet (prétraitement des données).

Nous avons implémenté 4 techniques de **classification non supervisée** :

- 2 techniques d'Extraction des motifs fréquents et des règles d'association : Apriori et Eclat après une discrétisation des données à analyser ainsi effectuer une étude comparative entre les résultats des deux techniques.
- 2 techniques de Clustering basés sur le partitionnement : K-Medids et CLARANS après avoir effectué une formule de calcul de distance, ainsi faire une étude comparative entre les résultats des deux techniques.

Nous avons ainsi décrit les structures de données utilisées, calculé la complexité pour chaque technique implémentée puis Illustré à travers l'IHM : L'extraction des motifs fréquents et des règles d'association, Les clusters des instances obtenus. Le temps d'exécution des programmes.

1 – Extraction des motifs Fréquents et des règles d'association :

Etant donné que le prétraitement des données à analyser a été effectué dans la première partie du projet, nous allons débuter droitement par la discrétisation des instances à fin d'appliquer les deux algorithmes Apriori et Eclat.

1.1 Discrétisation :

Pour réaliser une discrétisation, il faut choisir le nombre de classes et les bornes de classe. Pour réaliser une bonne discrétisation, il faut justifier à la fois le nombre de classes et les bornes de classe, le terme "bonne" faisant référence à des critères explicitement définis.

Il existe quelques formules "toute faites" pour déterminer à l'aveugle le nombre **n** de classes à partir du nombre **N** de données ,une formule qui est censées être la plus précise, mettent en jeu :

```
Scott=(b-a)/(3.5*sig*N**(-1/3))
```

- le minimum **a**, le maximum **b** des données
- les paramètres de la dispersion : **sig**➔ l'écart-type

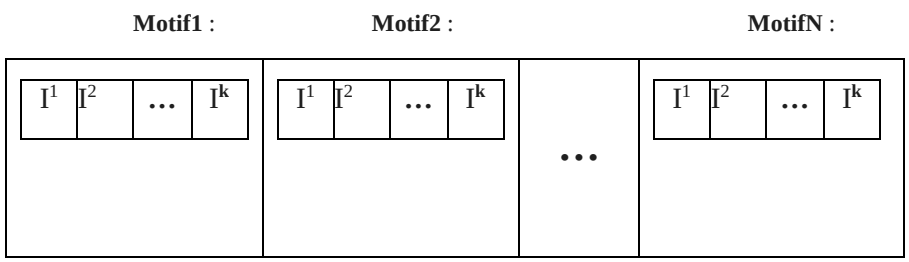
1.2 Apriori :

1.2.1 Structure de données :

Afin d'implémenter l'algorithme A-priori, nous avons utilisé les structures suivantes :

- Motifs fréquents :

C'est une liste HashSet ; pour ne pas avoir de doublons ; qui possède la structure ci-dessous :



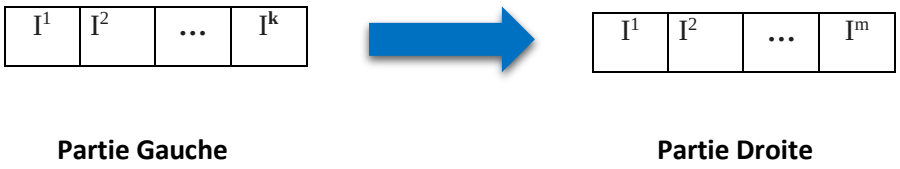
- Règles d'association :

C'est un Objet qui possède la structure ci-dessous :

Def Struct : RegleAssociation

PartieDroite : liste HashSet de réels. // *la partie droite de la règle.*

PartieGauche : liste HashSet de réels. // *la partie Gauche de la règle.*



1.2.2 Implémentation de l'algorithme :

L'algorithme Apriori s'exécute en deux étapes, Soient minsupp **l'indice de support minimum** donné, et minconf **l'indice de confiance** donné :

- Génération de tous les itemsets fréquents c'est-à-dire
- Génération de toutes les règles d'associations de confiance à partir des itemsets fréquent

Algorithme A-priori :

Var :

itemsTuple : Structure de motifs fréquents. // *contient les itemset candidats*
itemsTuplePris : Structure de motifs fréquents. *// contient les itemset candidats fréquents pris*
itemsTuplePrisSauv : Structure de motifs fréquents. *// Sauvegarde des motifs fréquents potentiels.*
RegleAssoc : tableau de RegleAssociation .//*contiendra les règles d'association.*
Niveau : entier *//longueur des itemsets.*

Début :

- On récupère les items du DataSet.
- On construit le premier niveau d'itemsets (les itemsets de longueur 1)

Pour chaque **Row** dans **Contenu** du DataSet :
Pour chaque **élément** dans **Row** :
On ajoute l'élément à itemsTuple.

//On calcule les supports
Pour chaque **élément** dans **itemsTuple** :
- On initialise le cpt à 0

Pour chaque **Row** dans **Contenu** du DataSet :
si Row contient **élément** on incrémente cpt.

Si **cpt>support** alors :

On ajoute élément à **itemsTuplePris**.

- On passe maintenant à la construction des autres niveaux.

Tant que (itemsTuplePris.taille <> 1) **et** (itemsTuplePris.taille <> 0) **Faire** :

- On initialise Deb à 1
- On réinitialise itemsTuple.

Pour chaque **élément** dans **itemsTuplePris** :

- On incrémente **Deb.**

Pour i allant de **Deb** à **itemsTuplePris.taille-1 :**

- On concatène le contenu de **itemsTuplePris[i]** avec celui de **élément** dans une liste **temporaire**.

Si temporaire.taille = Niveau alors :

- On trie la liste **temporaire** pour éviter les doublons et on l'ajoute à **itemsTuple.**

- On sauvegarde les **itemsTuplePris** dans **itemsTuplePrisSauv** et On réinitialise **itemsTuplePris**

Pour chaque **élément** dans **itemsTuple** :

- On initialise le cpt à 0

Pour chaque **Row** dans **Contenu** du DataSet :

- On initialise un booléen **Find** à **vrai.**

Pour chaque **item** dans **élément :** Si Row ne contient pas un item on met **Find** à Faux.

- On concatène le contenu de **itemsTuplePris[i]** avec celui de **élément** dans une liste **temporaire**.

Si temporaire.taille = Niveau alors :

- On trie la liste **temporaire** pour éviter les doublons et on l'ajoute à **itemsTuple.**

- On sauvegarde les **itemsTuplePris** dans **itemsTuplePrisSauv** et On réinitialise **itemsTuplePris**

Pour chaque **élément** dans **itemsTuple** :

- On initialise le cpt à 0

Pour chaque **Row** dans **Contenu** du DataSet :

- On initialise un booléen **Find** à **vrai.**

Pour chaque **item** dans **élément :** Si Row ne contient pas un item on met **Find** à Faux.

Si Find == vrai on incrémente cpt.

Si **cpt>support** alors :

On ajoute **élément** à **itemsTuplePris**.

- On incrémente **Niveau**.

//Construction des règles d'association.

- On réinitialise **itemsTuple.**

Pour chaque **motif fréquent** dans **itemsTuplePris :**

Pour chaque **item** dans **motif fréquent :**

- On ajoute item à **itemsTuple.**

//Construction des parties gauches des règles.

- On utilisera le même principe que pour la construction des itemSet sauf qu'on omet la partie de calcul des support et on varie la longueur des parties gauches de 1 à **longueur (motif fréquent) -1.** Une fois toutes les possibilités construites et ajoutées à **itemsTuple**, on passe à la construction des règles.

Pour chaque **élément** dans **itemsTuple :**

- On crée une nouvelle règle.
- On met le contenu de **élément** dans la partie gauche de la règle.

  Pour chaque **item** dans **Motif fréquent :**

  Si partieGauche de la règle ne contient pas **item** on l'ajoute à **partieDroite.**
- On ajoute la règle à **RegleAssoc.**

//Calcul des niveaux de confiance

Pour chaque **Règle** dans **RegleAssoc :**

- On utilisera le même principe que pour le calcul des supports pour les itemSet.
- On calcule le support1 du **motif fréquent**.
- On calcule le support2 de la **partieGauche** de la **Règle.**
- NivConf= (support1/support2)*100

Si (NivConf>NC) Alors On prend cette **R**

Fin

1.2.3 Complexité :

L'algorithme Apriori recherche les sous-ensembles ayant un support supérieur à Minsup appelés itemsets fréquents. Cette recherche a tout de même une complexité forte, de l'ordre de $\mathbf{2^n}$, où **n** est le nombre d'items (complexité exponentielle). En fait, dans la pratique, nous avons beaucoup moins de **n** items par transaction et par conséquent, la complexité réelle est bien moins. De plus, grâce à la détermination d'un support minimum nous supprimons les itemsets non-fréquents avant la génération de plus grands itemsets.

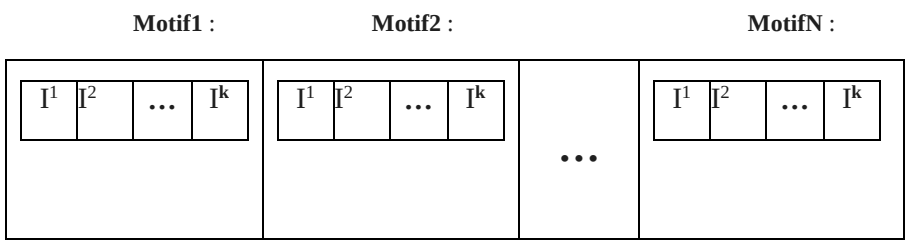
1.3 Eclat :

1.3.1 Structure de données :

Afin d'implémenter Eclat, nous avons utilisé les structures suivantes :

- Motifs fréquents :

C'est une liste de listes



- Règles d'association :

C'est un Objet qui possède la structure ci-dessous :

Def Struct : RegleAssociation

PartieDroite : liste HashSet de réels. // *la partie droite de la règle.*

PartieGauche : liste HashSet de réels. // *la partie Gauche de la règle.*

1.3.2 Implémentation de l'algorithme :

Ck: Candidate itemset of size k

Lk : frequent itemset of size k

L1 = {frequent items};

for (*k* = 1; *Lk* !=Ø; *k*++) **do begin**

Ck+1 = candidates generated from *Lk*;

for each transaction *t* in database do

increment the count of all candidates in *Ck+1* that are contained in *t*

Lk+1 = candidates in *Ck+1* with min_support

end

1.3.3 Complexité :

Complexité exponentielle ➔ d'ordre 2^n.

1.4 Etude comparative entre les résultats obtenus pour Apriori et ECLAT :

1. **Besoins en mémoire:** Étant donné que l'algorithme ECLAT utilise une approche de recherche en profondeur d'abord, il utilise moins de mémoire que l'algorithme Apriori.
2. **Vitesse:** l'algorithme ECLAT est généralement plus rapide que l'algorithme Apriori.
3. **Nombre de calculs: l'algorithme** ECLAT n'implique pas le balayage répété des données pour calculer les valeurs de support individuelles.

Conclusion : L'algorithme Eclat a plusieurs avantages, il utilise des classes d'équivalences qui sont indépendantes, il peut donc être parallélisé et après le partitionnement aucune communication n'est nécessaire entre les processus. Les classes d'équivalences sont basées sur la notion de préfixe commun, par exemple, les motifs ABC, ABD et ABE font partie de la même classe d'équivalence, car ils ont le motif AB pour préfixe commun. Les 4-motifs qui peuvent être générés par cette classe d'équivalence sont ABCD, ABCE et ABDE. Notons que ces motifs ne peuvent pas être générés par une autre classe d'équivalence.

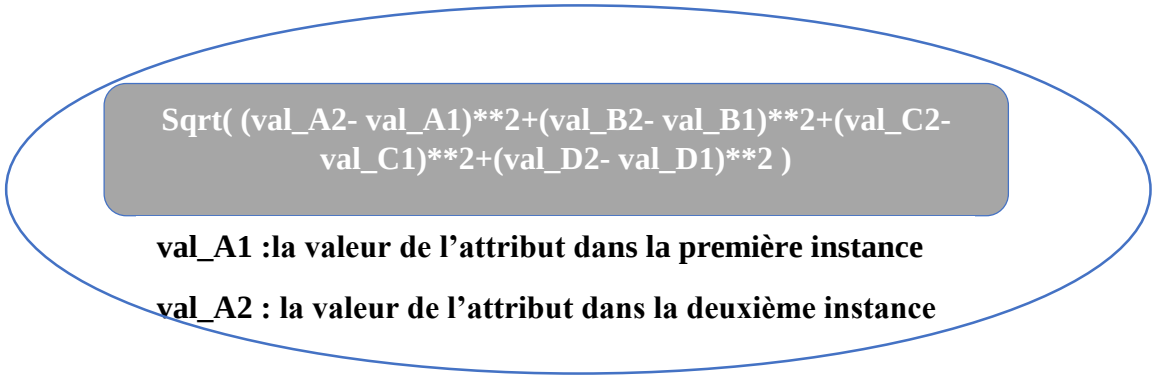
2 - Classification non supervisée des instances du dataset

En procédant à une classification, on cherche à construire des ensembles homogènes D'individus, c'est-à-dire partageant un certain nombre de caractéristiques identiques. Nous allons implémenter deux méthodes du clustering, et pour cela nous allons définir une formule de distance.

2.1 Calcul de la distance :

Les clusters sont formés en **optimisant un critère de partitionnement objectif**, tel qu'une fonction de dissimilarité basée sur la distance, de sorte que les **objets d'un cluster sont «similaires»** les uns aux autres et **«dissemblables» aux objets des autres clusters** en termes d'attributs de l'ensemble de données.

Dans notre cas, nous allons procéder par la formule euclidienne pour le calcul de la distance entre 2 instances:



2.2 K-medoids :

Dans cette partie, nous allons passer à l'implémentation de l'algorithme de partitionnement en k-medoids :

Objectifs :

- Former k-clusters avec une similarité intra-classes élevée et une similarité interclasses faible.

  En entrée, on aura notre DataSet, et **k** le nombre de clusters souhaité.

2.2.1 Implémentation de l'algorithme :

Algorithme de partitionnement en k-medoids :

Var :
Clusters : liste d'éléments de type Cluster.
DistancePrs : réel.
ClusterChoisi : Cluster.

Début :
//*initialiser les clusters.*
Pour i allant de 1 à **k** :
- On crée un nouveau cluster.
- On sélectionne aléatoirement un entier entre 0 et longueur du DataSet.
- On vérifie s'il n'a pas été déjà choisi comme centre.
- On l'affecte au centre du cluster créé et On ajoute le cluster à **Clusters**.

- On initialise un booléen **continu** à **Vrai**.
Tant que (continu == Vrai) **Faire** :
- **Pour** chaque cluster on initialise sa liste d'éléments.

Pour chaque **Row** dans **Contenu** du DataSet :
Pour chaque **Cluster** dans **Clusters :**
- On calcule la distance Euclidienne de **Row** avec **Cluster.centre** selon la formule suivante :

$$distance = \sqrt{\sum_{i=1}^{i=nbrAtt} (Row.set(i) - Cluster.centre.set(i))^2}$$

Si DistancePrs==0 **Alors :** // *c'est le cas où Row n'a pas été affecté encore.*
- On affecte la distance calculée à **DistancePrs.**
- On affecte **Cluster** courant à **ClusterChoisi.**

Sinon
Si DistancePrs> distance **Alors :**
- On affecte la distance calculée à **DistancePrs.**
- On affecte **Cluster** courant à **ClusterChoisi.**

- On affecte l'indice de **Row** à **ClusterChoisi. listeElements.**

- On met **continu** à faux.
Pour chaque **Cluster** dans **Clusters :**
- On sauvegarde le centre actuel **Cluster.centre**.
- On sauvegarde **ClusterChoisi. listeElements** dans un Tableau temporaire dans un ordre aléatoire.
- On met **continu2** à vrai.
- On initialise à 0 i.

Tant que ((continu2 == Vrai)&&(i<Taille(Tableau))) **Faire** :
- On initialise un nomedoid avec **Tableau[i].**
- On initialise les couts à 0.

- On calcule les distances Euclidienne selon la formule précédemment citée entre le nomedoid et les autres éléments puis on somme pour avoir le coût2.
- On fait de même pour le centre actuel du **Cluster** coût1.

Si coût1> coût2 **Alors :**

- On affecte le nomedoid comme nouveau centre du cluster.
- On met continu2 à Faux.

Sinon

- On incrémente i

Si le nouveau centre <> centre sauvegardé **Alors :**

- On met **continu** à vrai.

Fin

2.2.2 Complexité :

La complexité de Kmedoids est en **O (k (n-k)^2)** en terme de nombre d'observations

K : le nombre de clusters

2.3 CLARANS :

L'algorithme CLARANS (Clustering Large Applications based upon RANdomized Search) dans le contexte de la classification des données spatiales utilise une recherche aléatoire pour générer les voisins en commençant avec un nœud arbitraire et en vérifiant aléatoirement le paramètre "maxneighbor" des voisins.

2.3.1 Implémentation de l'algorithme :

Input: *A training set D, D = {O_h} $_{h=1..n}$; n is the size of D*

Initialize: *f (V) $_{max}$ = 0; iteration = 0;*

Repeat

1 Set C an arbitrary node from D; (C = [R_1, R_2, ..., R_k])

2. Set j =1;

3. Repeat

. *Consider a random neighbor C* of C;*

. *Compute TS_{ih} of C* and TS'_{ih} of C;*

. ***If*** *TS_{ih} > TS'_{ih}* ***then***

C = C;*

j = 1;

Else

j = j + 1;

Until *j = Maxneighbor;*

4. For each object $O_h \in D$ **do**

. Compute the similarity score of O_h with each medoid R_i $(i \in [1..K])$, using Smith Waterman algorithm;

. Assign O_h to the cluster with the nearest R_i ;

5. Compute $f(V)$;

6. If $f(V) \leq f(V)_{max}$ **then**

$iteration = iteration + 1$;

Else

$f(V)_{max} = f(V)$;

$BestSets = CurrentSets$;

Go back to Step3;

Until $iteration = q$;

End

Output: BestSets; BestSets is the best partition of D into K clusters; each cluster is defined by a medoid R_i

2.3.2 Complexité :

La complexité de CLARANS est en $O(N^2)$ en terme de nombre d'observations.

Parameters	K-mediods	CLARANS
Complexity	$O(k(n-k)^2)$	$O(n^2)$
Implementation	Complicated	Complicated
Sensitive to outliers?	No	No
No. of clusters parameter required?	Yes	Yes
Optimized for	Seperated clusters, small dataset	Seperated clusters, large dataset

2.4 Etude comparative entre les résultats obtenus pour K-Medoids et CLARANS :

- **k-medoids :** Pour surmonter le problème de sensibilité aux valeurs aberrantes, au lieu de prendre la valeur moyenne comme centroïde, nous pouvons prendre le point de données réel pour représenter le cluster, c'est ce que fait K-medoids.

  Mais les méthodes k-medoids sont **très coûteuses** lorsque l'ensemble de données et la valeur k sont importants.

- **CLARANS :** (Clustering Large Applications based on RANdomized Search): Il présente un compromis entre **le coût et l'efficacité de l'utilisation d'échantillons** pour obtenir le clustering.

3 - Visualisation

Dans cette partie nous avons Intégrer les programmes des sections 1 et 2 dans l'IHM développée dans la première partie du projet (Prétraitement des données) :

3.1 Présentation de l'IHM :

3.1.1 L'extraction des motifs fréquents et des règles d'association :

3.1.2 Les clusters des instances obtenus de la section 2 :

3.1.2 Le temps d'exécution des programmes :

CONCLUSION GENERALE

Une méthode de classification non supervisée produit une variable qualitative dont les modalités précisent la classe retenue pour chaque individu. Chaque méthode, algorithme, chaque choix de critère, dont le nombre de classes, conduit à des solutions différentes. Les critères proposés dits de qualité : corrélation cophénétique, silhouette... n'aident pas spécialement à un choix entre les solutions. Une représentation aide mieux à ce choix en faisant également intervenir, ensuite, des compétences métier pour cerner au mieux l'objectif, Pour conclure on va citer les étapes qu'on doit suivre Pour bien mener un projet de DataMining d'après ce qu'on a appris durant ce projet :

- ✓ Identifier et énoncer clairement les besoins.
- ✓ Créer ou obtenir des données représentatives du problème
- ✓ Identifier le contexte de l'apprentissage
- ✓ Analyser et réduire la dimension des données
- ✓ Choisir un algorithme et/ou un espace d'hypothèses.
- ✓ Choisir un modèle en appliquant l'algorithme aux données prétraitées.
- ✓ Valider les performances de la méthode.