



Rapport

Module : Data mining

Master 1 SII

Partie III

Études d'algorithmes de data-mining : Extraction de motifs fréquents, Classification et Clustering

- Réalisé par :

BENHADDAD Wissam

BOURAHLA Yasser

Table des matières

Table des matières	2
1 Introduction et problématique	2
2 Extraction de motifs fréquents à l'aide de l'algorithme Apriori	3
2.1 Introduction	3
2.2 Définitions	3
2.3 Algorithme	4
2.4 Implémentation	5
2.4.1 InstanceApriori :	5
2.4.2 AprioriInstanceReader :	6
2.4.3 Searcher :	6
2.5 Interface graphique	8
2.6 Résultats expérimentaux	9
2.6.1 Commentaires	10
3 Classification à l'aide de l'algorithme K plus proches voisins (KNN)	11
3.1 Introduction	11
3.2 Définitions	12
3.2.1 Point	12
3.2.2 Distance	12
3.2.3 Voisinage	13
3.2.4 Classification	13
3.2.5 Précision	13
3.3 Algorithme	13
3.4 Implémentation	14
3.4.1 KNNClassifier	15
3.5 Interface graphique	16
3.6 Résultats expérimentaux	17
3.6.1 Choix du dataset	17
3.6.2 Résultats	17
4 Clustering à l'aide de l'algorithme Density-based spatial clustering of applications with noise (DBSCAN)	22
4.1 Introduction	22
4.2 Définitions	22
4.2.1 E-voisinage	23
4.2.2 Core-point	23
4.2.3 Point de bord (Border-point)	23
4.2.4 Bruit	23

4.2.5	Cluster	23
4.2.6	Centre de gravité	24
4.2.7	Inerties	25
4.3	Algorithme	25
4.4	Implémentation	26
4.4.1	Point	27
4.4.2	Cluster	28
4.4.3	DBSCANClusterer	29
4.5	Interface graphique	31
4.6	Résultats expérimentaux	31
4.6.1	Choix du dataset	31
4.6.2	Variations des paramètres	32
4.6.3	Résultats	32
5	Conclusion	35
5.1	Bilan récapitulatif	35
5.1.1	Partie I	35
	Bibliographie	37

CHAPITRE 1

INTRODUCTION ET PROBLÉMATIQUE

CHAPITRE 2

EXTRACTION DE MOTIFS FRÉQUENTS À L'AIDE DE L'ALGORITHME APRIORI

2.1 Introduction

Souvent confronté à un ensemble de données qui n'ont vraisemblablement pas une régularité ou des sous structures qui se répètent suivant un certain motif, Une des tâches la plus répandue dans le domaine du Data-Mining est l'extraction de ces dits **Motifs fréquents**.

De façon informelle, un motifs fréquent peut être un item(objet, article ...) une sous-séquences d'items, une sous-structure(sous-graphe, sous-ensemble ...) qui se répète un certain nombre minimum de fois dans la base de données, ce qui lui vaut le nom de motifs **fréquent**[1].

Dans ce qui suit nous allons voir deux algorithmes capables tout deux d'extraire de tels motifs, l'algorithme **Apriori** [2] et l'algorithme FP-Growth [3]

2.2 Définitions

Avant d'introduire les deux algorithmes, il faut d'abord définir quelques concepts qui sont intrinsèquement reliés au déroulement de ces deux derniers :

Items

Un item I_i est généralement un attribut associé à un dataset(Taille,Poids,Catégorie...), cet item a un domaine de définition D_{I_i} .

Transaction

Une transaction T_i est généralement une instance du dataset, elle se présente comme un ensemble d'items aux quels une valeur à été attribué : $T_i = \{t_1, t_2, ..., t_n\}$, on lui associe un identifiant unique

id_{D_i} .

Support

Un support S est un indicateur (une mesure) de combien de fois un ensemble d'item X apparaît dans un dataset T , il est définie comme le nombre de transactions t qui contiennent l'itemset X :

$$Support(X) = \frac{|t \in T; X \subseteq t|}{|T|}$$

2.3 Algorithme

Apriori est un algorithme proposé par Agrawal et Srikant en 1994 dans [2], son but est l'extraction de motifs fréquents dans une base de données de transactions 2.2.

Apriori construit les ensembles d'items candidats à partir d'un ensemble d'items singletons en générant à chaque itérations une extension de ces derniers en ajoutant un item à la fois tout en testant la condition de support minimum ainsi que la condition de sous-motifs fréquent¹ pour permettre l'élimination plus rapide des itemsets candidats, l'algorithme s'arrête quand aucune extension ne peut être générée, le pseudo code est le suivant :

Algorithme 1 : Apriori

Entrée : (T : Ensemble des transactions , Sup_{min} : entier)

Sortie : (L : Ensemble des items fréquents)

Var :

C_k : Itemset des candidats de taille K L_k : Itemset des items les plus fréquents de taille K

début

```

     $L_1 \leftarrow \{items\ les\ plus\ frequent\}$ ;
    pour ( $k \leftarrow 1$ ;  $L_k \neq \emptyset$ ;  $k \leftarrow k + 1$ ) faire
         $L_{k+1} \leftarrow \text{GenererCandidats}(L_k)$ ;
        pour chaque transaction  $t \in T$  faire
            pour candidat  $c \in C_{k+1}$  faire
                si Contient( $t, c$ ) alors
                    |  $compteur[c] \leftarrow compteur[c] + 1$ 
                fin
            fin
        fin
         $L_{k+1} \leftarrow \{c | c \in C_{k+1} \wedge compteur[c] \geq Sup_{min}\}$ 
    fin

```

fin

retourner $\bigcup_m L_m; m = 0, k$

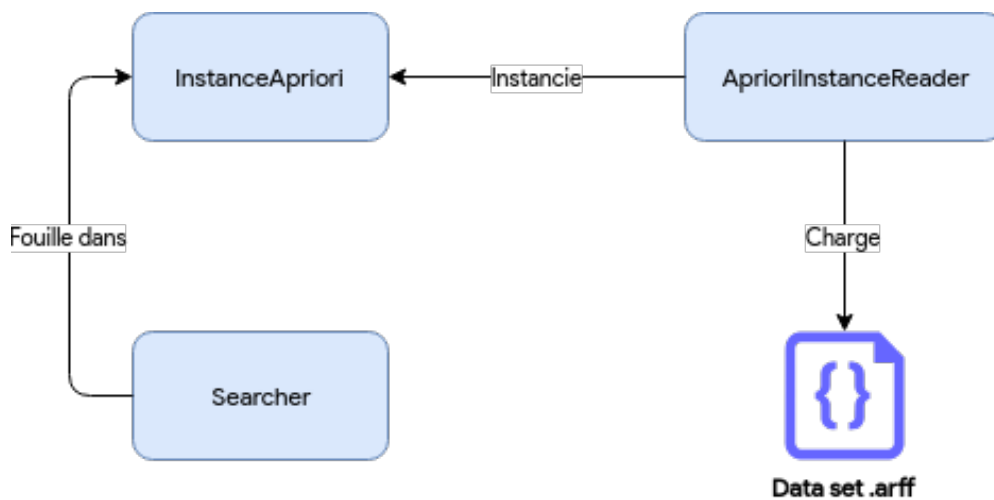
1. Si M est un motif fréquent alors $\forall m_i \in M$ m_i est aussi un item fréquent

2.4 Implémentation

Le module **Apriori** se décompose principalement de trois parties :

- **Chargement des données** : la phase où l'on récupère une instance d'un jeu de donnée à travers un fichier d'extension **.arff** qui contiendra les données et les méta-données associée (Nombre d'attributs, leurs types, valeurs manquantes ...), les chargeant en mémoire pour le traitement suivant.
- **Traitement sur les données** : c'est le coeur du système, c'est ici que l'algorithme va extraire les connaissances souhaitées en appliquant l'algorithme mentionné dans (REFE-REEEEEEEEEENCE TO APRIORI), les détails de l'implémentation seons mentionner

Pour se faire nous avons implémenté 3 classes **Java** qui effectuent chacune un travail bien précis, communiquant son résultat à une classe. Un schémas récapitulatif est présenté ici suivi d'une explication sur le fonctionnement de chacune des classes(avec les structures de données qu'elles manipulent)



2.4.1 InstanceApriori :

c'est la classe qui va contenir l'ensemble des transactions ainsi que les méthodes manipulant, les attributs sont les suivants :

```
public ArrayList<TreeSet<String>> transactions ;
ArrayList<TreeSet<String>> labeledTransactions;
```

Le choix de la structure **TreeSet** garantie un accès, ajout et suppression d'une entrée en $O(\log(n))$, les transactions seront donc représenté comme un vecteur d'ensemble de chaînes de caractères. la différence entre les deux attributs(attribut au sens Orienté-Objet) est que le 2e contient des ensemble de transactions de la forme **<attribut:valeur_attribut>**, c'est cette représentation qui sera choisie pour le traitement, l'autre sera utilisé pour l'affichage car il ne dispose pas de l'information sur l'attribut

Pour ce qui en est des méthodes utilisées :

```
public void addTransaction(int id, String[] items)
{...}
public void addLabeledTransaction(int id, String[] items)
{...}
@Override
public String toString()
{...}
public void printTransactions()
{...}
```

, les deux premières méthodes **addTransaction** et **addLabeledTransaction** font respectivement office d'interface pour ajouter une transaction à l'un des deux attributs (pour assurer une transparence d'utilisation de la classe). Les deux autres méthodes sont des méthodes d'affichage pour un éventuel débogage de l'application.

2.4.2 AprioriInstanceReader :

c'est la classe qui va construire un objet de la classe **InstanceApriori** à partir d'un fichier d'extension **.arff**, ou bien à partir d'un objet de la classe **weka.Instances** classe déjà existant.

Les attributs sont les suivants :

```
private File location;
private InstanceApriori instance;
```

l'attribut **instances** est donc un objet de la classe **InstanceApriori** vu précédemment, l'autre est un objet qui stock le descripteur du fichier **.arff**.

Les méthodes sont celles mentionnées dans 2.4.2, voici leurs prototypes :

```
public static InstanceApriori loadInstance(Instances data){...}

public static InstanceApriori loadInstance(File path) throws IOException {...}
```

2.4.3 Searcher :

Cette classe est celle qui va contenir à la fois les données en entrée (Instance et ses méta-données), les hyper paramètres ainsi que le résultat de l'algorithme (Itemsets fréquents et règles d'association).

Les attributs sont les suivants


```
private InstanceApriori instance;
private final int supMin;
private final double confMin;
public TreeMap<String, Double> rules = new TreeMap<>();
public TreeMap<String, Integer> Ls = new TreeMap<>();
```

Les trois premiers sont triviaux (voir 2.4.2), les deux derniers sont respectivement : une structure de hachage de la règle d'association à son degré de confiance **Règle -> Confiance** pour l'attribut **rules**, et une structure de hachage d'un itemset à sa fréquence d'apparition **Itemset -> Support**, et cela pour garantir un accès directe l'information désirée.

Pour ce qui est des méthodes, il en existe trois catégories :

- **Méthode principale** : essentiel au fonctionnement de l'algorithme. La seule méthode répondant à cette description est la méthode **search** :

```
public void search() {
    ArrayList<TreeSet<String>> prev = null;
    ArrayList<TreeSet<String>> L = generateCandidates(prev);
    TreeMap<String, Integer> fr;
    do {
        fr = getFreqSets(L);
        Ls.putAll(fr);
        prev = L;
        L = generateCandidates(prev);
    } while (!L.containsAll(prev) || !prev.containsAll(L));
    TreeMap<String, Double> unsrt = getAssRules();
    this.rules = new TreeMap<>(new rulesComparator(unsrt));
    this.rules.putAll(unsrt);
}
```

- **Méthode d'aide** : pour modulariser le traitement d'une méthode principale, les méthodes suivantes ont font partie :

```
private int getFreqOfItemSet(TreeSet<String> itemset) {...}
private TreeMap<String, Integer> getFreqSets(ArrayList<TreeSet<String>> C) {...}
private ArrayList<TreeSet<String>> generateCandidates(ArrayList<TreeSet<String>> prev) {...}
private TreeMap<String, Double> getAssRules() {...}
private ArrayList<TreeSet<String>> getSubSets(TreeSet<String> set) {...}
```

Dans l'ordre le rôle de chacune est le suivant :

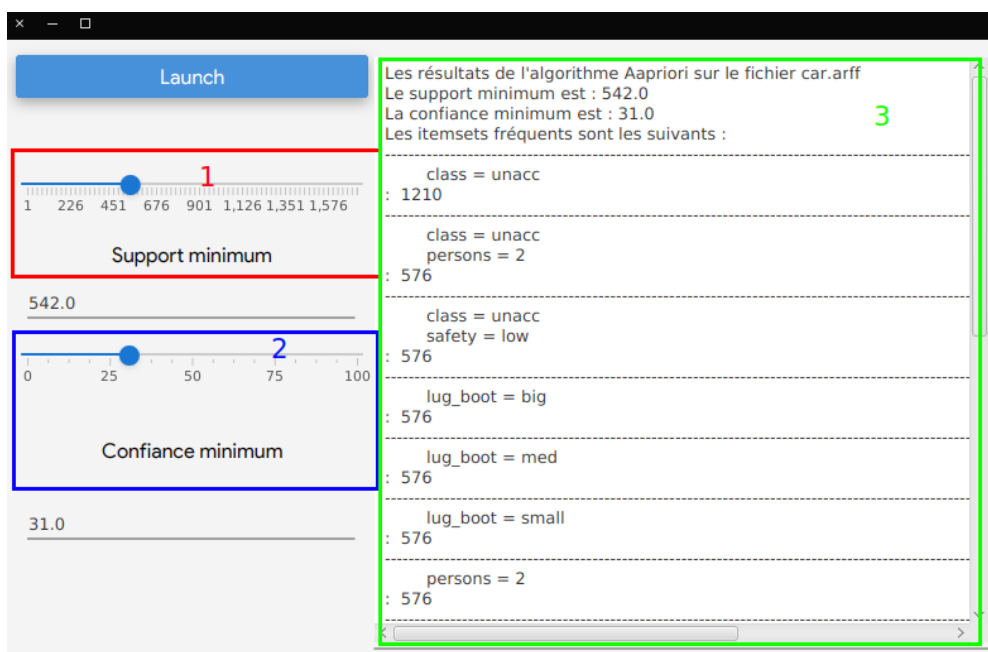
- **getFreqOfItemSet** Calcule le support d'un itemset
- **getFreqSets** Extrait les itemset fréquents (dont le support dépasse le support minimum)
- **generateCandidates** Génère un nouveau itemset candidat en combinant les item d'un itemset antérieur.
- **getAssRules** Extrait les règles d'association à partir des itemset fréquents extraits au préalable.
- **getSubSets** Extrait l'ensemble des parties d'ensemble d'un itemset.

- **Méthode débogage** : principalement servant à afficher de manière structurée les résultats finaux ou intermédiaire d'un des types de méthodes cités. Nous y trouvons les méthodes :

```
private static String setToString(TreeSet<String> set) {...}
private static TreeSet<String> stringToSet(String s) {...}
private class ItemComparator implements Comparator<String> {...}
private class rulesComparator implements Comparator<String> {...}
```

2.5 Interface graphique

Nous avons intégré l'interface graphique pour l'utilisation d'algorithme **Apriori** dans l'interface principale, pour que l'utilisateur puisse lancer le traitement sur une dataset préalablement traité (nettoyage et remplissage de valeur manquantes), voici l'interface choisie suivit de quelques explications :



L'interface se compose de 3 zones :

- **Zone 1 et 2** : permettre d'introduire les valeurs des hyper paramètres
- **Zone 3** : affichage du résultat comme une liste d'itemsets fréquents suivie d'une liste de règle d'association.

2.6 Résultats expérimentaux

2.6.0.1 Choix du dataset

Pour tester le comportement de notre implémentation de l'algorithme apriori, nous avons choisi de le tester sur un dataset purement nominal (**car.arff**).

2.6.0.2 Variations des paramètres

Nous avons choisi de faire varier les deux paramètres qui sont supMin et confMin de façon automatique, la plage du premier est l'intervalle $[0.1 \times \text{nombreInstance}, \text{nombreInstance}]$ avec un pas $step = \frac{\text{nombreInstance}}{10}$.

La plage du 2e est l'intervalle suivant : $[0.5, 1]$ avec un pas $step = 0.1$

2.6.0.3 Résultats

Les résultats sont récapitulés dans le tableau suivant :

supMin	confMin	nombre itemset freq	nombre de règle d'ass	temps (ms)
172	0.5	86	22	410
172	0.6	86	14	431
172	0.7	86	6	433
172	0.8	86	2	388
172	0.9	86	1	382
344	0.5	31	6	41
344	0.6	31	6	42
344	0.7	31	3	44
344	0.8	31	2	40
344	0.9	31	1	47
516	0.5	12	1	36
516	0.6	12	1	39
516	0.7	12	1	38
516	0.8	12	1	37
516	0.9	12	1	39
688	0.5	1	0	43
...
1204	0.9	1	0	35
...
1720	0.9	0	0	33

TABLE 2.1 – Tableau récapitulatif des résultats de l'algorithme apriori sur le dataset **car.arff**

Remarques : Les valeurs en vert désignent une stagnation des valeurs **nombre itemset freq** et **nombre de règle d'ass**

2.6.1 Commentaires

D'après le tableau, il est notable que le paramètre qui influe le plus sur le temps d'exécutions est le **support minimum**, c'est facilement remarquable si on prend une valeur fixe pour ce dernier en faisant varier l'autre paramètre **Confiance minimum**, par exemple :

$$\text{Si } supMin = 344 \text{ et } \forall confMin \rightarrow temps \in [40, 47]$$

Cela montre que la plus part des ressources sont mobilisé pour l'extraction des itemsets fréquents.

Il est aussi a noté que plus le support minimum augmente (on impose donc un forte condition sur la fréquence d'apparitions des itemsets) le nombre d'itemsets fréquent ainsi que le nombre de règles d'association ainsi que le temps de calcul diminuent en conséquence.

CHAPITRE 3

CLASSIFICATION À L'AIDE DE L'ALGORITHME K PLUS PROCHES VOISINS (KNN)

3.1 Introduction

Un des principaux problème auquel nous devons faire face quand nous exploitant des données réelles est l'inférence d'une étiquette pour une nouvelle donnée non rencontrée avant, c'est la définition d'un problème de classification, et il a beaucoup de domaine d'application (détection de fraude, détection d'intrusion, évaluation de la qualité d'un produit ...). L'un des nombreux algorithmes mis au point pour résoudre ce type de problème est l'algorithme des K plus proches voisins ou K-Nearest-Neighbors (KNN).

L'algorithme est basé sur une mesure de similarité appelé **Distance** dans un voisinage suivit de l'attribution d'une étiquette sur la base d'un vote a majorité (pondéré ou pas) des représentant du voisinage en question, l'étiquette est donc attribué selon la similarité entre un point (une instance) et ce voisinage.

3.2 Définitions

Avant de détailler le fonctionnement de l'algorithme nous allons introduire quelques notions pour mieux comprendre son fonctionnement

3.2.1 Point

Un point P est une instance du dataset, c.à.d un ensemble P_{att} de valeur d'attributs, plus formellement, un point est l'extrémité d'un vecteur :

$$\vec{V} = a_1.\vec{v}_1 + a_2.\vec{v}_2 + \dots + a_n.\vec{v}_n$$

où :

- \vec{v}_i : est un vecteur unité de base (vecteur caractéristique d'un attribut)
- n : le nombre d'attribut d'un point donné

Le point dont nous parlons est donc analogue à un point dans un espace multi-dimensionnel désigné par des coordonnées cartésiennes.

3.2.2 Distance

La notion de distance est une valeur (très souvent numérique) qui quantifie la mesure de similarité entre deux point dans une espace multi-dimensionnel, la distance tendra donc à être de plus en plus petite si deux points sont très similaires, et inversement elle sera de plus en plus grande si ces deux points sont de plus en plus différents.

Il existe plusieurs façon de mesurer la distance entre deux points donnée dans une espace multi-dimensionnel, nous avons choisi pour cela d'utiliser la distance d'ordre 2 (2-distance) plus connue sous le nom de distance euclidienne dont la formule est la suivante :

Soient deux points $X(x_1, \dots, x_n)$ et $Y(y_1, \dots, y_n)$

$$\text{Et soit la distance } D_p(X, Y) = \sqrt[p]{\sum_{i=1}^n |x_i - y_i|^p}$$

La distance euclidien 2-distance est la suivante :

$$D_2(X, Y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Il reste donc à définir la sémantique de l'opérateur $-$ qui lui introduit la distance élémentaire entre deux valeur d'un même attribut, dans le cas numérique cette opérateur est l'opérateur arithmétique classique de la soustraction, pour les valeurs nominales nous avons choisis une interprétation simple pour faciliter l'exploitation de l'algorithme par la suite qui est la suivante :

$$x_{nom} - y_{nom} = \begin{cases} 1 & \text{Si } x_{nom} = y_{nom} \\ 0 & \text{Sinon} \end{cases}$$

Elle représente la distance de Hamming simplifiée au cas d'une variable a valeur binaires seulement.

3.2.3 Voisinage

Le concept de voisinage d'un certain point P est une notion très importante en classification et clustering, elle désigne l'ensemble de points V qui sont similaires à un certain degré à P , plus particulièrement si $\forall v \in V$ on a $D_p(P, v) = \minDst$ où \minDst est la plus petite distance entre deux points, alors V est l'ensemble des voisins directs de P .

3.2.4 Classification

De manière informelle, étant donnée un ensemble d'instances (points) dont nous connaissons l'étiquette (ou la classe), la classification est l'opération d'affectation d'une classe à une nouvelle instance qui n'a pas encore été classifié avant. Ainsi un algorithme de classification est une fonction F telle qu'étant donné un ensemble de points E et un ensemble de classes C :

$$F : E \rightarrow C$$
$$F(p \in E) = c$$

3.2.5 Précision

Pour un classifieur F , nous sommes souvent amenés à mesurer son degré d'exactitude sur un ensemble de test, la précision est une mesure qui reste naïve mais donne une bonne approximation de la qualité de la classification, elle consiste en un rapport entre le nombre de classes correctement prédites sur le nombre total des instances à classifier dans l'ensemble de test. Plus formellement :

Soient \hat{F} une fonction qui donne toujours la classe exacte à une instance

Et $T = (Att, C)$ Un ensemble de paires Attributs et Classe

Et $Pred = \{t \in T | F(t) = \hat{F}(t)\}$ L'ensemble des points dont la prédiction est correcte

Alors la précision par rapport au classifieur F est :

$$P_F = \frac{|Pred|}{|T|}$$

3.3 Algorithme

L'algorithme **KNN** se base comme cité précédemment sur le principe du vote majoritaire, en effet si un individu est proche d'un ensemble d'autres individus qui lui sont similaires, ses caractéristiques seront elles aussi similaires aux individus dont l'étiquette est la plus dominante (cette façon de penser peut introduire la notion de bruit ou valeurs déviantes que nous verrons par la suite).

L'algorithme dispose d'un paramètre empirique :

- **K** : le nombre de voisins à analyser pour un point donné

Mais il dépend aussi de la taille et la diversité de l'échantillon d'apprentissage en entrée. Le choix du paramètre K est une étape cruciale qui déterminera les performances de l'algorithme, pour faciliter l'étape de classification (après le vote de la majorité) on prendra un $K = 2 * m + 1$ avec

$m \in \mathbb{N}$ qui sera donc impair pour qu'une majorité émerge, cependant cette restriction ne règle le problème que si $K > |C|$ où $|C|$ est le nombre de classes, en effet si l'on observe que K voisins dans le cas où $K \leq |C|$, il y aura toujours une chance que les voisins soient tous étiquetés avec une classe $c \in C$ différentes pour chacun.

Le pseudo-code suivant détaille les étapes à suivre :

Algorithme 2 : KNN

Entrée : (E : Ensemble des instances d'apprentissage, X : instance à classier , K : entier)

Sortie : (C : Classe inférée de X)

Var :

$Distances$: Ensemble des paires (point, $D(X,point)$) ;

$Distances \leftarrow \emptyset$;

pour $e \in E$ **faire**

$Distances \leftarrow Distances \cup \{(e, D_2(X, e))\}$;

fin

$Distances_tri \leftarrow \text{TrierParPlusPetiteDistance}(Distances)$;

$D_K \leftarrow \{D_i \in Distances_tri | i = 1..K\}$;

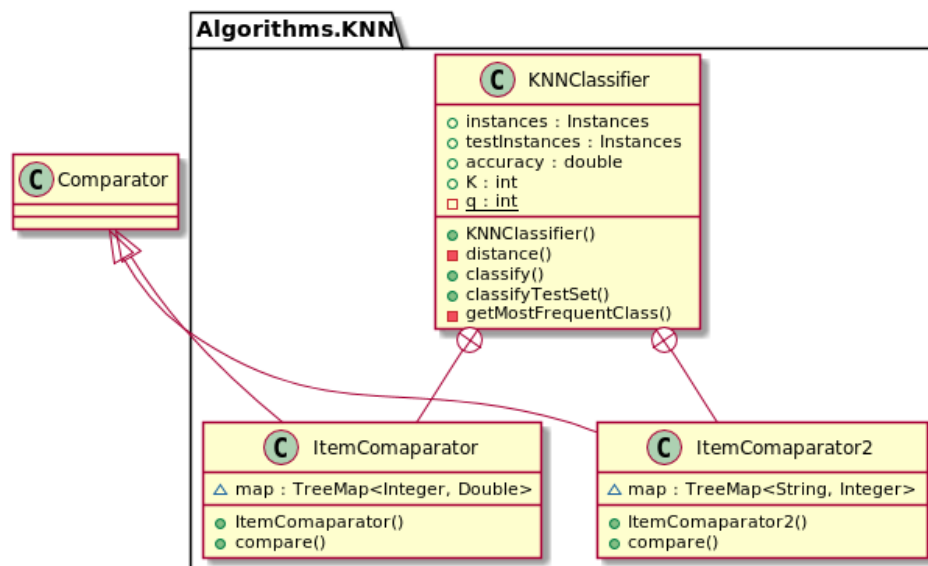
$C \leftarrow \text{ClasseDominante}(D_K)$;

retourner C

3.4 Implémentation

Pour mieux visualiser la structure interne de notre implémentation, nous allons présenter d'abord un diagramme UML puis ensuite détailler les composants du module **KNN** :

KNN's Class Diagram



PlantUML diagram generated by SketchIt! (<https://bitbucket.org/pmesmeur/sketch.it>)
For more information about this tool, please contact philippe.mesmeur@gmail.com

Le module est composé de deux parties majeures :

- Deux classes **ItemComparator** et **ItemComparator2** qui implémentent chacune la fonction **compare** de l'interface **Comparator**, cela est dû au fait que nous avons eu besoin de trier deux structures de hachage de la classe **TreeMap** selon les **valeurs** dans l'ordre croissant et décroissant.
- La classe **KNNClassifier** qui contiendra les méthodes et attributs nécessaire pour l'exécution de l'algorithme.

3.4.1 KNNClassifier

Il est maintenant temps de détailler le contenu de la classe **KNNClassifier** :

- Les attributs sont les suivants :

```
public Instances instances;
public Instances testInstances;
public double accuracy = 0;
public int K;
```

Les deux premiers sont des objets de la classe **weka.Instances**, ils représentent respectivement les échantillons d'apprentissage et de test. Ensuite nous avons défini les deux paramètres de notre algorithme **K** le nombre de voisins à tester et **ratio** le taux de division entre échantillons d'apprentissage et de test. Ici **q** le paramètre de distance **p** vu dans 3.2.2

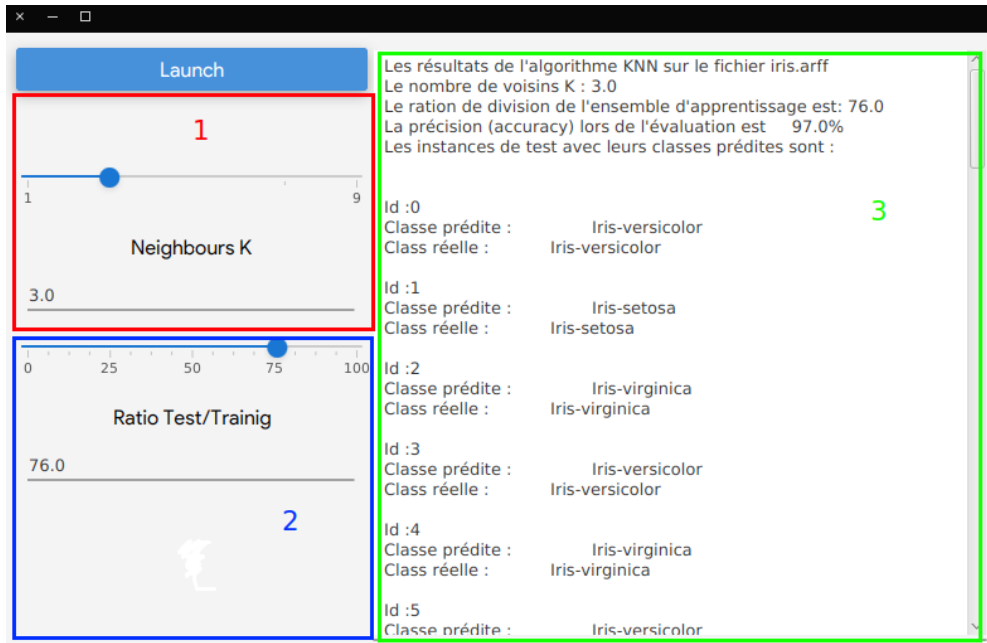
- Les méthodes implémentées sont les suivantes :

```
public String classify(Instance instance) {...}
public TreeMap<Integer, String> classifyTestSet(Instances test) {...}
private double distance(Instance A, Instance B) {...}
private String getMostFrequentClass(String[] cls) {...}
```

- **classify** c'est la méthode principale qui prend une instance en entrée et retourne sa classe prédite en sortie.
- **classifyTestSet** c'est une méthode qui fait appel à **classify** sur un ensemble de test dont les classes sont connues, et calcule la précision de la classification effectuée
- **distance** la méthode de calcul d'une distance 3.2.2 entre deux points (instances) données
- **getMostFrequentClass** elle extrait d'un ensemble de classes, celle qui est la plus fréquente en terme d'apparition.

3.5 Interface graphique

De manière équivalente à ce que nous avons fait pour le module précédent (2) Nous avons intégré à l'interface graphique principale l'algorithme **KNN**, voici l'interface choisie suivie de quelques explications :



L'interface se compose de 3 zones :

- **Zone 1 et 2 :** permettre d'introduire les valeurs des hyper paramètres (K et ratio)
- **Zone 3 :** affichage du résultat comme une liste d'instances avec leurs classes prédites et leurs classes réelles, ainsi que la précision de la classification.

3.6 Résultats expérimentaux

3.6.1 Choix du dataset

Puisque nous disposons d'un moyen de calculer une distance entre deux points dont les attributs sont soit nominaux soit numériques, nous avons choisis de tester les 3 combinaisons de ces choix, c.à.d :

- Dataset purement numérique : nous avons choisis le dataset **iris.arff**
- Dataset purement nominal : nous avons choisis le dataset **car.arff**
- Dataset hybride nous avons choisis le dataset **credit-g.arff**

la taille des datasets est un critère important à prendre en compte, plus la taille est grande plus le temps d'inférence (étiquetage d'une nouvelle instance) pourrait être grand, car il faudra donc parcourir toutes les instances dans l'échantillon d'apprentissage, néanmoins, si nous posons N comme le nombre de ces instances, et K le nombre d'attributs pour chaque instances, il faudra donc parcourir les K attributs de chacune des N instances puis insérer la distances D dans une

structure de hachage en $O(\log(M))$ où M est le nombre de voisins, ce qui nous donne donc une complexité temporelle $O(N * K * \log(M))$ ce qui reste assez raisonnable a plus grande échelle.

3.6.1.1 Variations des paramètres

Pour ce qui est des paramètres, nous avons décidé de faire varier le nombre de voisins K ans l'intervalle $[3, 2 * \text{NombreDeClasses} + 1]$ avec un pas $step = 2$ (pour s'assurer que les valeurs restent impairs).

Le paramètre **ratio** sera lui varié dans l'intervalle $[0.1, 0.9]$ (ce qui représente la taille de l'échantillon d'apprentissage)

3.6.2 Résultats

Nous avons lancé l'algorithme sur les datasets mentionnés plus haut en gardant à chaque fois les informations nécessaires pour comparer les résultats (précision, temps d'exécution), les tableaux comparatifs suivants résument le processus, ils seront ensuite accompagnés de quelques commentaires :

Résultats pour car.arff

K	Nb instances apprentissage	Nb instances test	Précision	ratio	temps(s)
3	172,8	1555,2	0,89	0,1	4,686
3	345,6	1382,4	0,88	0,2	3,395
3	518,4	1209,6	0,9	0,3	2,729
3	691,2	1036,8	0,89	0,4	2,44
3	864	864	0,89	0,5	1,976
3	1036,8	691,2	0,9	0,6	1,556
3	1209,6	518,4	0,89	0,7	1,201
3	1382,4	345,6	0,89	0,8	0,816
3	1555,2	172,8	0,93	0,9	0,4
5	172,8	1555,2	0,79	0,1	3,575
5	345,6	1382,4	0,82	0,2	3,181
5	518,4	1209,6	0,75	0,3	2,859
5	691,2	1036,8	0,86	0,4	2,367
5	864	864	0,83	0,5	1,993
5	1036,8	691,2	0,83	0,6	1,605
5	1209,6	518,4	0,83	0,7	1,172
5	1382,4	345,6	0,8	0,8	0,792
5	1555,2	172,8	0,85	0,9	0,412
7	172,8	1555,2	0,73	0,1	4,242
7	345,6	1382,4	0,74	0,2	3,125
7	518,4	1209,6	0,7	0,3	2,759
7	691,2	1036,8	0,76	0,4	2,376
7	864	864	0,78	0,5	1,986
7	1036,8	691,2	0,83	0,6	1,581
7	1209,6	518,4	0,84	0,7	1,384
7	1382,4	345,6	0,83	0,8	0,791
7	1555,2	172,8	0,75	0,9	0,396

TABLE 3.1 – Résultats de l'algorithme KNN sur le dataset car.arff

Pour mieux visualiser les résultats, nous avons décidé de fixer un des paramètres (K) en faisant varier l'autre pour analyser en quoi ces variations pourraient affecter la précision de l'algorithme :

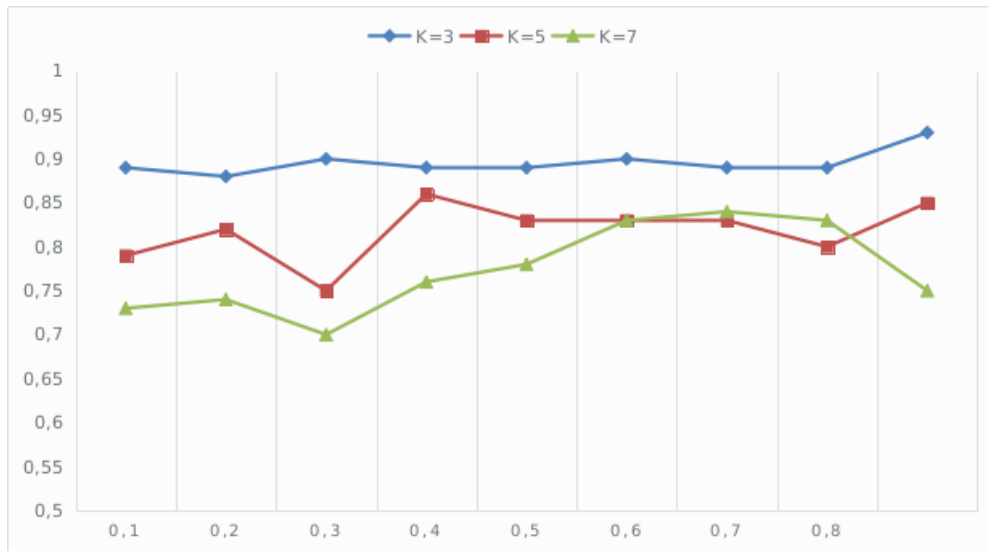


FIGURE 3.1 – Graphes comparatif des résultats de l’algorithme sur le dataset car.arff

Commentaires Du point de vue intrinsèque au choix de K c.à.d si l’on ne compare pas entre les résultats de deux variations de ce paramètres différentes, on peut noter que le changement du ration d’apprentissage/test n’affecte pas beaucoup sur la précision, mise à part dans le cas où l’on choisi un trop gros échantillons d’apprentissage ce qui va biaiser le résultat final, auquel cas le classifieur aura déjà une grande connaissance sur le dataset et la classification sera plus aisée.

D’un point de vue extrinsèque aux choix de K , c.à.d si on compare la précision par rapport aux choix de ce dernier, il est a noté qu’un plus petit choix de ce paramètre donne systématiquement une meilleure précision avec de petites fluctuations (ceci n’est pas une règle générale, juste une observation sur notre jeu de tests).

Résultats pour iris.arff

K	Nb instances apprentissage	Nb instances test	Précision	ratio	temps(s)
3	15	135	0,97	0,1	0,088
3	30	120	0,96	0,2	0,061
3	45	105	0,95	0,3	0,017
3	60	90	0,96	0,4	0,016
3	75	75	0,94	0,5	0,01
3	90	60	0,98	0,6	0,008
3	105	45	1	0,7	0,008
3	120	30	1	0,8	0,005
3	135	15	1	0,9	0,003
5	15	135	0,96	0,1	0,023
5	30	120	0,95	0,2	0,019
5	45	105	0,97	0,3	0,02
5	60	90	0,96	0,4	0,017
5	75	75	0,96	0,5	0,012
5	90	60	0,95	0,6	0,011
5	105	45	1	0,7	0,008
5	120	30	0,96	0,8	0,005
5	135	15	0,93	0,9	0,003

TABLE 3.2 – Résultat de l'algorithme KNN sur iris.arff

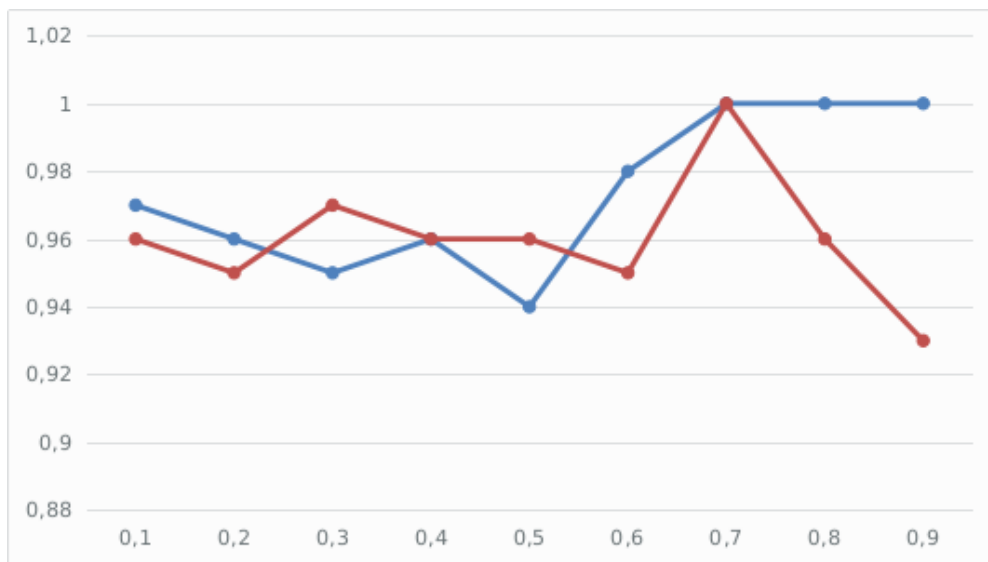


FIGURE 3.2 – Graphes comparatif des résultats de l'algorithme sur le dataset iris.arff

Commentaires : Pour ce dataset, contrairement au précédent, du point de vue intrinsèque au choix de K , il n'est pas garantie que la précision augmente si le taux d'apprentissage augmente, on peut le voir pour la courbe en rouge où la précision a diminué, certes d'un petit pas (de l'ordre de 0.08), mais à plus grande échelle cette quantité deviendrait cruciale.

Il est aussi à noter que la classification est assez rapide pour des valeurs numériques.

3.6.2.1 Résultats pour credit-g.arff

K	Nb instances apprentissage	Nb instances test	Précision	ratio	temps
3	100	900	0,85	0,1	3,982
3	200	800	0,85	0,2	4,066
3	300	700	0,87	0,3	3,132
3	400	600	0,86	0,4	2,632
3	500	500	0,86	0,5	2,289
3	600	400	0,86	0,6	1,791
3	700	300	0,87	0,7	1,321
3	800	200	0,86	0,8	0,886
3	900	100	0,83	0,9	0,446

TABLE 3.3 – Résultats de l'algorithme KNN sur le dataset credit-g.arff

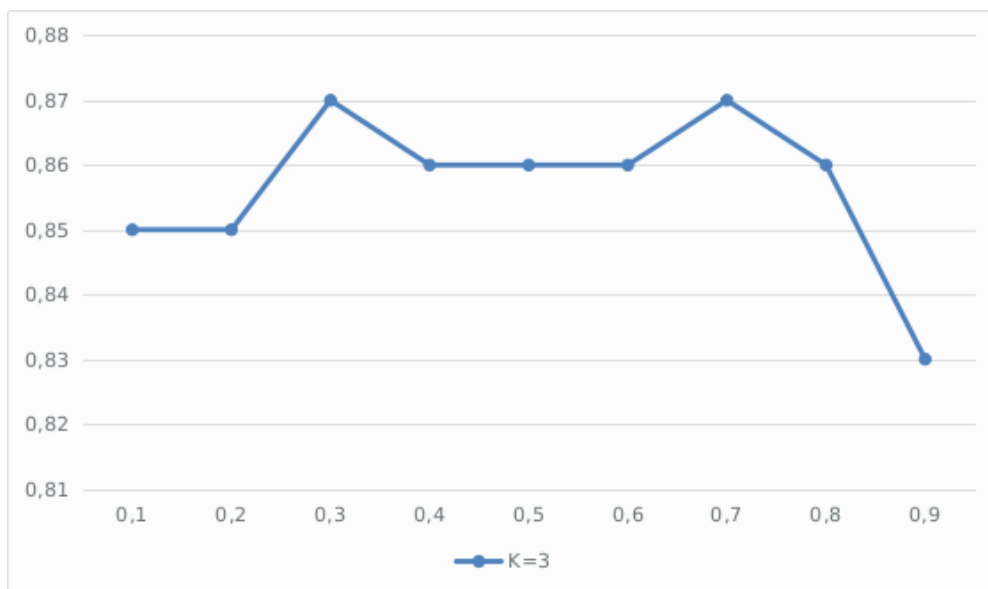


FIGURE 3.3 – Graphes comparatif des résultats de l'algorithme sur le dataset credit-g.arff

Commentaires : De manière analogue au dataset précédent, du point de vue intrinsèque au choix de K , la précision peut diminuer dans le cas où le taux d'apprentissage augmente, on peut le voir pour la courbe en [bleu](#) où la précision a diminué d'un facteur de 0.04

Il est aussi à noter que la classification a pris un temps assez considérable pour (peut être du à la présence d'attributs nominaux ?)

CHAPITRE 4

CLUSTERING À L'AIDE DE L'ALGORITHME DENSITY-BASED SPATIAL CLUSTERING OF APPLICATIONS WITH NOISE (DBSCAN)

4.1 Introduction

Quand nous sommes face à un gros ensemble de données dont nous connaissons les caractéristiques mais pas les relations qui lient chaque individu d'un échantillon, nous voudrions être capable de regrouper les instances (point, individu ...) qui sont similaires (petites distances entre eux) dans un sous ensemble bien défini, ce processus est plus communément appelé **Clustering**.

De manière informelle, le clustering est l'opération de regroupement d'objets dans un groupe compact nommé **Cluster** de telle sorte que les membres d'un même cluster soient similaires à un certain point, et arbitrairement différents (grandes distances) des objets d'un autre cluster.

Il existe une multitude d'algorithmes de clustering, nous avons décidé d'implémenter l'algorithme Density-based spatial clustering of applications with noise (DBSCAN) pour analyser son comportement, et ainsi critiquer ses forces, ses faiblesses et ainsi pouvoir l'utiliser dans des domaines qui lui conviendraient.

4.2 Définitions

Comme pour les chapitres précédents, nous allons donner quelques définitions pour faciliter la compréhension de l'algorithme, il est à noter que nous utiliserons quelques définitions du chapitre précédent, notamment (3.2.1 3.2.2 3.2.3)

4.2.1 E-voisinage

Semblable à la notion de voisinage vu dans 3.2.3, le e-voisinage d'un point P appartenant un ensemble de points E est l'ensemble V_ϵ des points qui sont à une distance $D \leq \epsilon$, plus formellement :

$$V_\epsilon(P) = \{x \in E | D_2(P, x) \leq \epsilon\}$$

De plus si P est un **core-point** (voir 4.2.2) alors l'ensemble V_ϵ formera un ensemble de points **directement-atteignables**.

4.2.2 Core-point

Un core-point, selon l'algorithme **DBSCAN** est un point P_c dans l'ensemble E dont le cardinal de son E-voisinage est supérieure ou égale à une constante $minPts = k$, l'ensemble de ces core-points formera un ensemble que l'on dénotera E_c , plus formellement

$$E_c = \{P \in E | k \leq |V_\epsilon(P)|\}$$

Les éléments du e-voisinage d'un core-point P sont des points qui sont directement-atteignables par ce point.

4.2.3 Point de bord (Border-point)

Un point de bord est un point P_b dans l'ensemble E dont le cardinal de son E-voisinage est inférieure au paramètre k et il existe un core-point P_c qui est dans ce même E-voisinage. L'ensemble de ces border-points formera un ensemble que l'on dénotera E_b , plus formellement :

$$E_b = \{P \in E | k > |V_\epsilon(P)| \wedge \exists P_c \in V_\epsilon(P) \cap E_c\}$$

4.2.4 Bruit

Un point P est considéré comme étant du bruit P_n s'il n'est ni un core-point ni un border-point, il appartiendra à un ensemble E_n tel que :

$$E_n = \{P \in E | P \notin E_c \wedge P \notin E_b\}$$

4.2.5 Cluster

Un cluster C est un ensemble de points dont un algorithme de clustering a déterminé qu'ils étaient similaires dans un certain sens, on peut donc dire que c'est l'ensemble des points dont la similarité est très forte, et dont la similarité avec les points d'autres clusters $C' \neq C$ est très faible.

4.2.6 Centre de gravité

Le centre d'un gravité d'un cluster C est un point G_C (qui n'existe pas forcément dans le cluster) qui sera désigné comme représentant de ce dit cluster, il existe beaucoup de façons de le déterminer, une des façon de le faire est l'utilisation de la formule suivante :

$$G_C = (\frac{\sum_{i=1}^N x_{i,1}}{N}, ..., \frac{\sum_{i=1}^N x_{i,M}}{N})$$

Avec :

- N : le nombre de points dans le cluster.
- M : le nombre d'attributs d'un point dans le cluster.

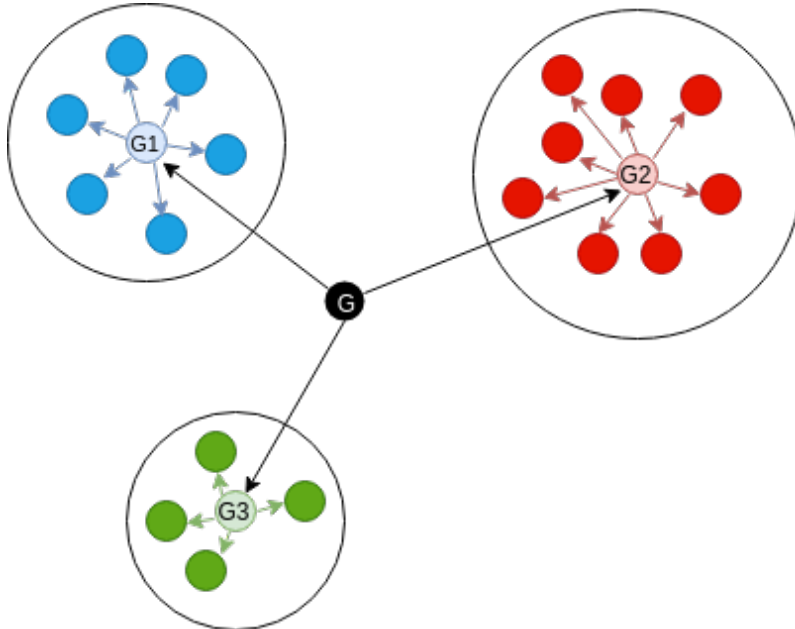
De même nous pouvons définir le centre de gravité du nuage de points en entier, en considérant G l'ensemble des centres de gravité de chaque clusters C , soit G_{pts} le point que nous cherchons, ce sera le centre de gravité de l'ensemble G , donc :

$$G_{pts} = (\frac{\sum_{i=1}^{N_G} x_{i,1}}{N_G}, ..., \frac{\sum_{i=1}^{N_G} x_{i,M}}{N_G})$$

où

- N_G : le nombre de points dans G .
- M : le nombre d'attributs d'un centre de gravité.

Voici un schema pour illustrer cela :



4.2.7 Inerties

Pour quantifier la similarité (inversement la dissimilarité) au sein d'un cluster, et a travers les clusters, nous avons décidé d'utiliser les deux mesures suivantes :

4.2.7.1 Inertie intra-classes

C'est une quantité qui détermine a quel point les éléments d'un cluster sont proches les uns des autres, elle est calculé en sommant les carrés des distances entre ces points et le centre de gravité d'un cluster. Nous disposons d'un ensemble de clusters C , dont les centres de gravité seront notés G_{C_i} , on définira l'inertie d'un cluster C comme étant :

$$Iner(C_i) = \sum_{x \in C_i} D_2(x, G_{C_i})^2$$

Par la suite, l'inertie intra-classe d'un ensemble de clusters C comme étant la somme des inerties des clusters le composant :

$$Intra(C) = \sum_{C_i \in C} iner(C_i)$$

De ce fait, plus cette quantité est petite, plus les éléments de chaque cluster sont proches les un des autres, c'est donc une quantité a minimiser par un algorithme de clustering.

4.2.7.2 Inertie inter-classes

C'est une quantité qui détermine a quel point les éléments d'un cluster sont loins des éléments des autre clusters, elle est calculé en sommant les carrés des distances entre les centre de gravités de chaque cluster avec le centre de gravité du nuage de points G_{pts} , Nous disposons d'un ensemble de clusters C , dont les centres de gravité seront notés $g \in G_c$, on définira l'inertie inter-classe d'un ensemble de cluster C comme étant :

$$Inter(C) = \sum_{g \in G_C} D_2(g, G_{pts})^2$$

De ce fait, plus cette quantité est grande, plus les éléments de chaque cluster sont loins par rapport aux éléments des autre clusters, , c'est donc une quantité a maximiser par un algorithme de clustering.

4.3 Algorithme

Le principe de l'algorithme est assez simple, en commençant d'un core-point arbitraire, il es-sayera de construire un cluster incluent ce point et tout les points indirectement-atteignables à partir de lui même, en marquant chaque point visité pour ne pas le retraiter par la suite, les limites d'un cluster seront donc délimités par les border-points, ainsi, les clusters seront formés de points très proches les uns des autres (d'après la définition des points directement(indirectement)-atteignables), ce qui aura pour effet de construire des clusters très **Denses**. Le pseudo code est le suivant :

Algorithme principal :

Algorithme 3 : DBSCAN

Entrée : (E : Ensemble des instances, ϵ : réel, minPts : entier)

Sortie : (C : Ensemble des clusters)

```
 $C \leftarrow 0$ ;  
 $NV \leftarrow E$  Ensemble des points non visités;  
 $V \leftarrow \emptyset$  Ensemble des points visités;  
 $N \leftarrow \emptyset$  Ensemble des points bruits;  
pour chaque  $p \in NV$  faire  
|  $NV \leftarrow NV - \{p\}$ ;  
|  $V \leftarrow V \cup \{p\}$ ;  
|  $\text{Voisinage} \leftarrow \text{E-Voisinage}(p, \epsilon)$ ;  
| si  $|\text{Voisinage}| < \text{minPts}$  alors  
| |  $N \leftarrow N \cup \{p\}$ ;  
| sinon  
| |  $C \leftarrow C + 1$ ;  
| |  $\text{EtendreCluster}(p, \text{Voisinage}, C, \epsilon, \text{minPts})$   
| fin  
fin  
retourner  $C$ 
```

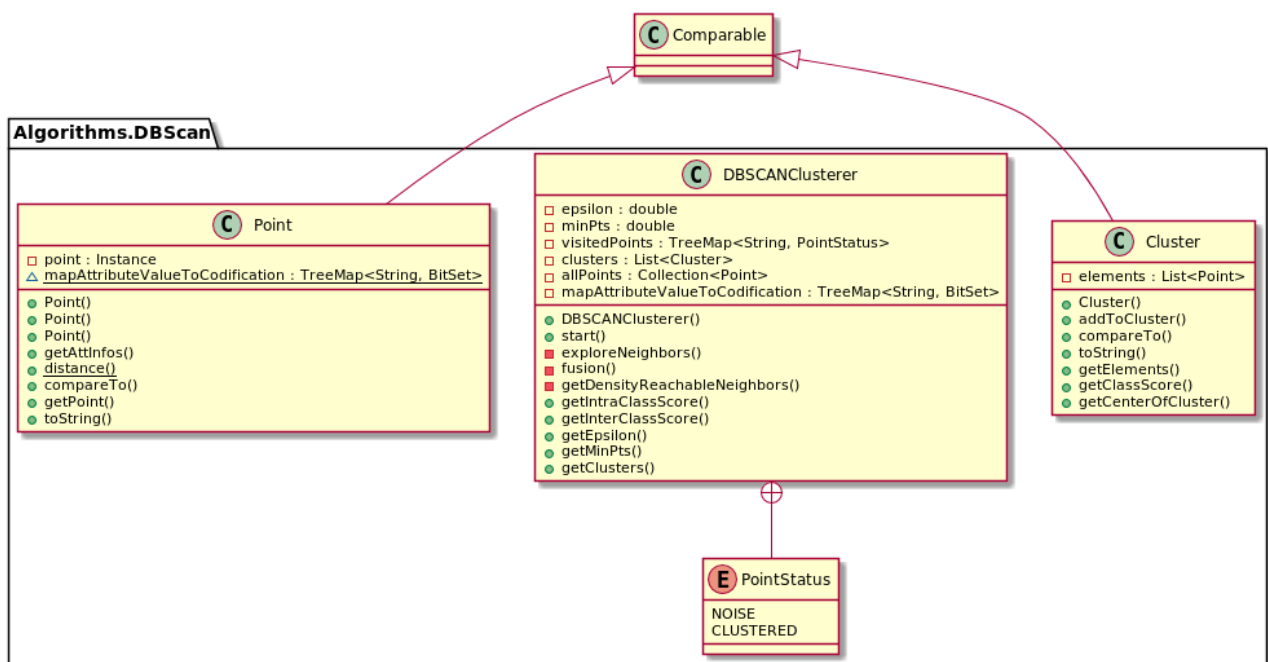
Procédure $\text{EtendreCluster}((p : \text{Point}, \text{Voisinage} : \text{ensemble}, C : \text{cluster}, \epsilon : \text{réel}, \text{minPts} : \text{entier}))$

```
 $C \leftarrow C \cup \{p\}$ ;  
pour chaque  $v \in \text{Voisinage}$  faire  
| si  $v \notin V$  alors  
| |  $NV \leftarrow NV - \{v\}$ ;  
| |  $V \leftarrow V \cup \{v\}$ ;  
| |  $\text{Voisinage}' \leftarrow \text{E-Voisinage}(v, \epsilon)$ ;  
| | si  $|\text{Voisinage}'| \geq \text{minPts}$  alors  
| | |  $\text{Voisinage} \leftarrow \text{Voisinage} \cup \text{Voisinage}'$ ;  
| | fin  
| fin  
| si  $v \notin C \forall C \in \text{Clusters}$  alors  
| |  $C \leftarrow C \cup \{v\}$ ;  
| fin  
fin
```

4.4 Implémentation

Pour mieux visualiser la structure interne de notre implémentation, nous allons présenter d'abord un diagramme **UML** puis ensuite détailler les composants du module **DBSCAN** :

DBSCAN's Class Diagram



PlantUML diagram generated by SketchUML (<https://bitbucket.org/pmesmeur/sketch.uml>)
For more information about this tool, please contact philippe.mesmeur@gmail.com

Le module se compose donc de trois grande classes :

- **Point** qui servira d'encapsulation à la notion de point dans un dataset, utilisant des méthodes de manipulation d'un point ou de points.
- **Cluster** qui servira d'encapsulation à la notion de cluster comme étant un ensemble d'objets de la classe **Point**, utilisant des méthodes de manipulation de clusters et de calcul des centre de gravité.
- **DBSCANClusterer** qui est la classe qui va effectuer le travail de clustering en manipulant les objets des deux classes précédentes.

4.4.1 Point

- Nous commençons par détailler les attributs de la classe :

```
private Instance point;
```

L'unique attribut de la classe st un objet de la classe **weka.instance**, car à la base un point est une instance du dataset.

- Les méthodes implémentées sont les suivantes :

```
public Point() {}
public Point(Instance instance) {...}
public Point(Point p) { this.point = (Instance) p.getPoint().copy(); }

public ArrayList<Attribute> getAttInfos()
{...}
public static double distance(Point p1, Point p2, double q) {...}

@Override
public int compareTo(Point point) { return (int) distance( p: this, point, q: 1); }

@Override
public String toString() {...}
```

- **Point()** Ce sont les différents constructeur de la classe qui instancient soit avec une valeur nulle, à partir d'un objet de la classe **weka.Instance** ou bien en copiant les attribut d'un objet de la classe **Point**
- **getAttInfos** Une méthode d'aide qui retourne la liste des attributs d'une instance.
- **Distance** la méthode de calcul de la p-distance 3.2.2 entre deux points (instances) données
- **compareTo** implémentation de la méthode compareTo de l'interface **Comparable** pour qu'on puisse comparer deux objets de la classe Point.

4.4.2 Cluster

```
private List<Point> elements;

public Cluster() {...}

public void addToCluster(Point p) {...}

public double getClassScore() {...}

public Point getCenterOfCluster() {...}
```

- L'unique attribut de la classe st une liste d'objets de la classe **Point**, car un cluster est un ensemble de points.
- Les méthodes implémentées sont les suivantes :
 - **Cluster()** Le constructeur de la classe qui initialise la liste de points.
 - **addToCluster** Méthode d'aide qui ajoute un point au cluster.
 - **getCenterOfCluster** la méthode qui calcul le centre de gravité du cluster.
 - **getClassScore** méthode qui retourne la valeur de l'inertie intra-classe du cluster.

4.4.3 DBSCANClusterer

- Nous commençons par détailler les attributs de la classe :

```
private final double epsilon;
private final double minPts;
private TreeMap<String, PointStatus> visitedPoints = new TreeMap<>();
private List<Cluster> clusters = new ArrayList<>();
private Collection<Point> allPoints = new ArrayList<>();

private enum PointStatus {
    NOISE,
    CLUSTERED
}
```

- **epsilon** et **minPts** sont les deux paramètres de l'algorithme.
 - **visitedPoint** une structure de hachage qui associe à chaque point son status (Dans un cluster ou Bruit).
 - **clusters** une liste d'objets de la classe Cluster qui contiendra les clusters construits par l'algorithme.
 - **allPoints** l'ensemble des points du dataset.
- Les méthodes implémentées sont les suivantes :

```
public void start() {
    for (Point point : allPoints) {
        if (visitedPoints.get(point.toString()) != null) {
            continue;
        }
        List<Point> neighbors = getDensityReachableNeighbors(point, allPoints);
        if (neighbors.size() >= minPts) {
            Cluster cluster = new Cluster();
            clusters.add(exploreNeighbors(cluster, point, neighbors, allPoints, visitedPoints));
        } else {
            visitedPoints.put(point.toString(), PointStatus.NOISE);
        }
    }
}
```

- **start** : c'est la boucle principale de l'algorithme qui va parcourir l'ensemble des points pour former les clusters.

```
private Cluster exploreNeighbors(Cluster cluster, Point point, List<Point> neighbors,
                                Collection<Point> allPoints,
                                Map<String, PointStatus> visited) {
    cluster.addToCluster(point);
    visited.put(point.toString(), PointStatus.CLUSTERED);
    List<Point> startingNeighbors = new ArrayList<>(neighbors);
    int index = 0;
    while (index < startingNeighbors.size()) {
        Point current = startingNeighbors.get(index);
        PointStatus pointStatus = visited.get(current.toString());
        if (pointStatus == null) {
            List<Point> currentNeighbors = getDensityReachableNeighbors(current, allPoints);
            if (currentNeighbors.size() >= minPts) {
                startingNeighbors = fusion(startingNeighbors, currentNeighbors);
            }
        }
        if (pointStatus != PointStatus.CLUSTERED) {
            visited.put(current.toString(), PointStatus.CLUSTERED);
            cluster.addToCluster(current);
        }
        index++;
    }
    return cluster;
}
```

- **exploreNeighbors** : c'est la procédure **étendreCluster** vu précédemment, elle va chercher à rassembler tout les points indirectement-atteignables depuis un point donné pour former un nouveau cluster.

```
private List<Point> getDensityReachableNeighbors(Point point, Collection<Point> points) {
    List<Point> neighbors = new ArrayList<>();
    for (Point neighbor : points) {
        double dst = Point.distance(point, neighbor, 2);
        if (point != neighbor && dst <= epsilon) {
            neighbors.add(neighbor);
        }
    }
    return neighbors;
}
```

- **getDensityReachableNeighbors** : c'est la procédure **E-voisinage** vu précédemment, elle va retourner l'ensemble des points directement-atteignable depuis le point donné.

```
public double getIntraClassScore() {
    if (clusters.size() == 0)
        return -1;
    double sum = 0;
    for (Cluster cl : clusters) {
        sum += cl.getClassScore();
    }
    return sum;
}

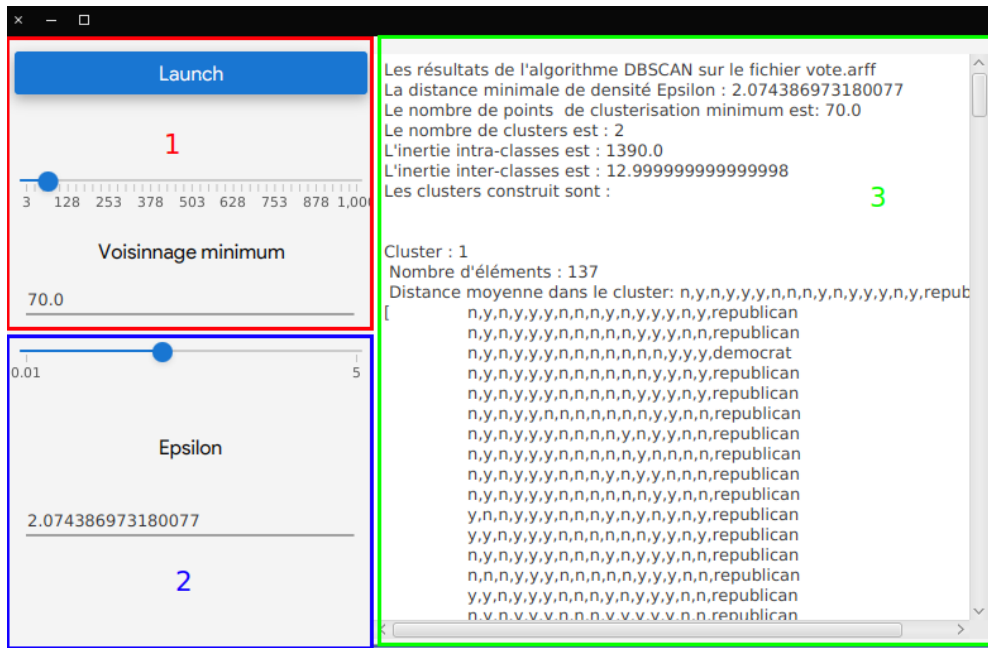
public double getInterClassScore() {
    if (clusters == null)
        return -1;

    Cluster allGs = new Cluster();
    for (Cluster cl : clusters) {
        allGs.addToCluster(cl.getCenterOfCluster());
    }
    return allGs.getClassScore();
}
```

- • **getIntraClassScore** elle va calculer l'inertie intra-classes des clusters.
- **getInterClassScore** elle va calculer l'inertie du cluster formé des centres de gravité de chaque cluster, donc l'inertie inter-classes.

4.5 Interface graphique

De manière analogue à ce que nous avons fait pour les modules précédents (2 3) Nous avons intégré à l'interface graphique principale l'algorithme **DBSCAN**, voici l'interface choisie suivie de quelques explications :



L'interface se compose de 3 zones :

- **Zone 1 et 2** : permettre d'introduire les valeurs des hyper paramètres ϵ et minPts.
- **Zone 3** : affichage du résultat comme une liste de clusters avec les points qui les composent, ainsi que les valeurs des inerties inter et intra-classes.

4.6 Résultats expérimentaux

4.6.1 Choix du dataset

Nous avons choisi de tester le comportement de l'algorithme dans les deux cas où les données sont soit purement numériques soit purement nominales :

- Dataset purement numérique : nous avons choisis le dataset **iris.arff**
- Dataset purement nominal : nous avons choisis le dataset **vote.arff**

4.6.2 Variations des paramètres

Pour ce qui est des paramètres, nous avons décidé de faire varier le nombre de points minimum **minPts** car il affecte grandement le choix d'un point comme core-point, il prendra ses valeurs dans l'intervalle $[0.1 * \text{nbrInstance}, \frac{\text{nbrInstance}}{3}]$ avec un pas $\text{step} = \frac{\text{nbrInstance}}{10}$.

Le 2e paramètre ϵ quantifiera la notion de **densité** dans le nuage de point, en effet si la valeur de paramètre est grande, l'algorithme tendra à considérer des points assez distants de lui comme étant **proches**. Malheureusement la normalisation de valeurs d'attributs des points ne garanti pas qu'on aura une distance qui sera elle aussi normalisée, pour illustrer cela nous allons prendre l'exemple suivant :

et

$$Y = (y_1, \dots, y_n)$$

Si on pose $x_i, y_i \in [0, 1], \forall i = 1..n$, Plus particulièrement donnons à chaque attribut x_i la valeur 0.5, et à chaque attribut y_i la valeur 1 ainsi $x_i = 1, y_i = 0.5, \forall i = 1..n$, et posons $n = 10$ le nombre d'attributs, ainsi la formule de la 2-distance 3.2.2 deviendra :

$$D_2(X, Y) \approx 1.58 \notin [0, 1] \quad (4.5)$$

4.6.3 Résultats

Remarques Dans le cas où aucun cluster n'est construit les deux mesures de performances vaudront -1.

Résultats pour iris.arff

ϵ	minPts	Nb cluster	L'inertie intra-classes	L'inertie inter-classes	temps
0,111803399	5	5	2,062029911	4,361321025	8
0,111803399	10	1	0,907413281	0	4
0,111803399	15	1	0,315181554	0	5
0,223606798	5	2	18,62857928	0,85701552	5
0,223606798	10	2	18,00560921	0,85701552	6
0,223606798	15	2	13,04401911	0,715499107	5
0,223606798	20	2	10,41219874	0,715499107	5
0,223606798	25	2	7,711637591	0,823875133	6
0,223606798	30	2	3,223514733	0,611410909	4
0,223606798	35	2	2,408509621	0,702501631	5
0,223606798	40	0	-1	-1	5
0,223606798	45	0	-1	-1	3
0,335410197	5	2	19,50523183	0,85701552	6
...
0,335410197	30	2	19,17890257	0,85701552	4
0,335410197	35	2	14,3095591	0,715499107	6
0,335410197	40	2	13,49945477	0,715499107	6
0,335410197	45	2	13,2196536	0,715499107	6
0,447213595	5	2	19,50523183	0,85701552	5
...
0,447213595	45	2	19,50523183	0,85701552	6
0,559016994	5	1	101,5489152	0	5
...
2,124264579	45	1	101,5489152	0	5

TABLE 4.1 – Résultats de l'algorithme DBSCAN sur le dataset iris.arff

Commentaires Nous pouvons extraire de ce tableau que le meilleur couple de paramètre est celui tout en haut du tableau, car il construit plusieurs cluster, et minimise l'inertie intra-classe tout en maximisant l'inertie interclasse. Il est à noter que dans certains cas, on peut tomber sur une valeur très grande pour l'inertie intra-classe, en effet comme nous l'avons démontré dans 4.1, une distance de deux points normalisés peut dépasser 1, si on considère la formule de l'inertie d'une classe vu dans 4.2.7.1, qui est la somme d'un nombre de carrés de distances égal au nombre de points dans un cluster, nous pouvons facilement atteindre des valeurs très grandes, les principales causes sont donc un très grand nombre d'attributs et un très grand nombre de points dans un même cluster. Ceci peut expliquer une très grande différence entre les deux mesures de performance, mais ceci importe peu car le but n'est pas de minimiser la différence absolue entre ces deux valeurs, mais de considérer chaque valeur comme un objectif à minimiser ou à maximiser, c'est donc une fonction à double objectifs que nous voulons optimiser.

Résultats pour vote.arff

ϵ	minPts	Nb cluster	L'inertie intra-classes	L'inertie inter-classes	temps
1,236931688	14	2	273	14	606
1,236931688	28	2	86	14	620
1,236931688	42	2	16	14	612
1,64924225	14	2	902	13	619
1,64924225	28	2	709	13	615
1,64924225	42	2	640	13	617
1,64924225	56	2	375	13	621
1,64924225	70	2	314	13	617
2,061552813	56	2	1399	13	628
2,061552813	70	2	1390	13	621
2,061552813	84	2	1341	13	620
2,061552813	98	2	1285	13	621
2,061552813	112	2	1193	13	621
2,061552813	126	2	1042	14	615
2,061552813	140	2	977	14	611
1,64924225	84	1	134	0	615
2,061552813	14	1	2395	0	607
2,061552813	28	1	2387	0	627
2,061552813	42	1	2380	0	607
2,473863375	14	1	2395	0	620
...
3,710795063	140	1	2395	0	615

TABLE 4.2 – Résultats de l'algorithme DBSCAN sur le dataset vote.arff

Commentaires Nous pouvons extraire de ce tableau que le meilleur couple de paramètre est celui en vert, en effet il sépare les données en deux clusters(ça tombe bien puisque nous savions que les attributs du dataset ont deux classes possibles) mais aussi il maximise l'inertie inter-classe et minimise l'inertie intra-classe.

Comme expliqué précédemment, le fait qu'on obtienne une valeur pour l'inertie inter-classe inférieur à celle de l'inertie intra-classe n'a rien d'alarmant, les deux quantité ne sont pas faites pour être comparées entre elles, mais plutôt à être optimisées chacune séparément. Un autre point à soulever est la présence de très grandes valeurs pour l'inertie intra-classe, puisque le dataset est purement nominal, composée d'instances à 17 attributs, ajouté au fait qu'on utilise une distance nominale naïve (voir 3.2.2 et 3.2.2) ainsi que la présence de grand nombre de points dans les clusters, ces valeurs sont de prime à bord logiques.

Au terme de ce projet, nous avons donc pu explorer 3 des aspects du data-mining qui sont : l'extraction de motifs fréquents, la classification des données et le clustering. en implémentant à chaque étapes un algorithme rudimentaire du sous domaine en question, arrivé a ce point nous disposons donc de notre propre outillage pour l'exploration et l'exploitation des ensemble de données.

Il reste toute fois à dresser un bilan récapitulatif qui recense dans chaque partie une analyse de l'algorithme utilisé ainsi que des critiques sur ce dernier.

5.1 Bilan récapitulatif

5.1.1 Partie I

À la fin du chapitre II (voir 2), nous avons pu nous initier à une technique basique d'extraction de motifs fréquents sous forme d'items , et cela depuis un dataset. En implémentant l'algorithme Apriori et en le testant sur un ensemble de benchmark, nous avons pu en tirer les conclusions suivantes :

- **Point forts :**

- Il est très facile à implémenter, omettant les amélioration des structures de données en terme de temps d'accès (ce qui a poussé à utiliser des structures plus développés que de simple vecteur(tableaux)), la facilité de l'implémentation d'Apriori est une facteur non négligeable quand nous sommes amenés à développer une solution rapidement.
- Il est très simple a comprendre, de par sa nativité et son approche qu'on peut qualifier de **directe**, dans le sens où aucune tentative d'optimisation des opérations n'est effectuée. Il suffit généralement de dérouler un petit exemple à la main pour comprendre comment l'intuition de sa conception à été trouvée.
- Mise à part le dataset choisi pour nos tests, nous avons aussi eu l'occasion de le tester sur un large ensemble d'items, sa complexité temporelle quasi polynomiale donnait d'assez bon résultats lors de la mise à l'échelle, c'est aussi dû au choix de paramètre de

façon intelligente qui a permis cela (prendre un support minimum relatif à la taille du dataset par exemple et non pas une constante non adaptée à chaque dataset, de même pour la confiance minimum).

- **Point faibles :**

- La génération des candidats à chaque itération est une opération très coûteuse en temps, effectuant des opérations de jointures qui, si elles ne sont pas optimisées, peuvent alourdir le processus dans le cas d'un large dataset.
- L'extraction des règles d'association impose le calcul de l'ensemble des sous-ensembles de chaque itemset, c'est aussi une opération très coûteuse en temps.
- Le calcul du support minimum impose le parcours de la table des items en entier, et cela à chaque itérations de l'algorithme, de plus l'algorithme assume que cette table soit chargée en mémoire de façon permanente, ce qui peut poser problème si sa taille atteint un seuil critique.

Partie II

Partie III

- [1] J. Han, M. Kamber, and J. Pei, *Data Mining : Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2011.
- [2] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, (San Francisco, CA, USA), pp. 487–499, Morgan Kaufmann Publishers Inc., 1994.
- [3] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” *ACM SIGMOD Record*, vol. 29, pp. 1–12, jun 2000.