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

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

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 4.2.2.

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 fréquent [1] 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
  $L_1 \leftarrow \{itemslesplusfrquent\}$;
  pour $(k \leftarrow ; L_k \neq \emptyset ; k \leftarrow k+1)$ **faire**
    $L_{k+1} \leftarrow$ `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 $\bigcup_m L_m; m = 0, k$

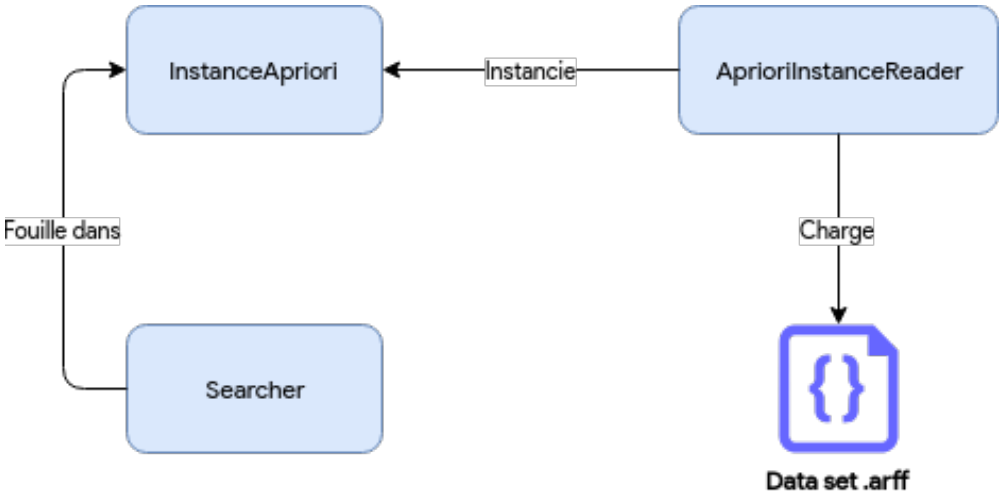
1. Si M est un motif fréquent alors $\forall m_i \in M$ m_i est aussi un item fréquent

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-donnés 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 une 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 entré (Instance et ses méta-donnés), 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 :

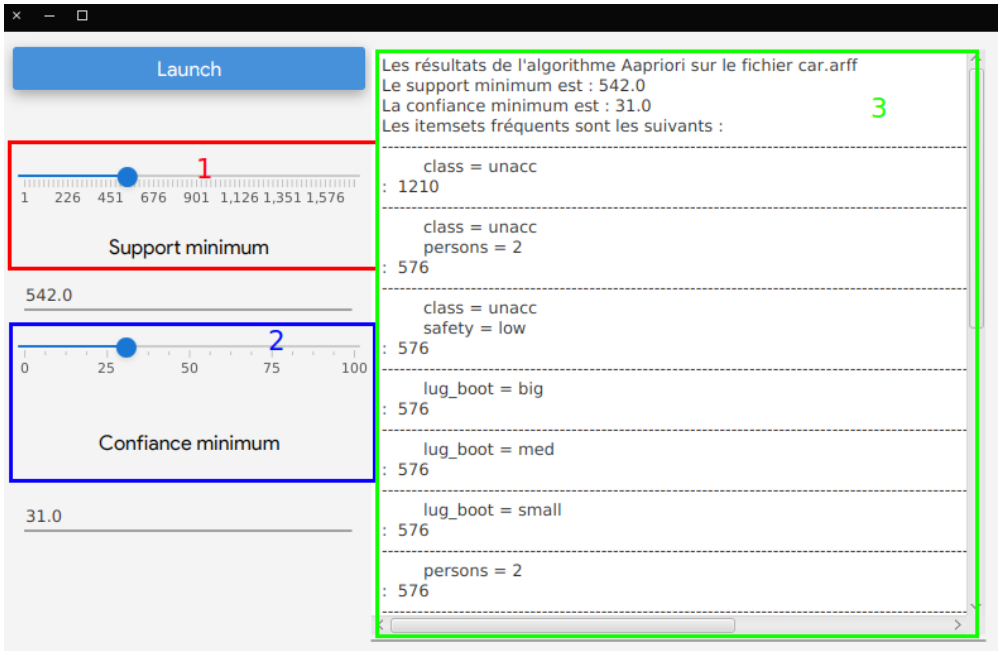
  - **getFreqOfItemSet** Calcule le support d'un itemset
  - **getFreqSets** Extrait les itemset fréquents (dont le support dépasse le support minimum)
  - **generateCandidates** Génère un nouveau itemset candidat en combinant les item d'un itemset antérieur.
  - **getAssRules** Extrait les règles d'association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

- **Méthode deboggage** : 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 ItemComaparator 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 datase 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***nombreInstance** , **nombreInstance**] avec un pas $step = \frac{nombreInstance}{10}$.

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 récapitulé 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 un 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.

2.7 Brève conclusion

À la fin de ce chapitre nous avons pu nous initier à une technique rudimentaire de l'extraction d'itemset fréquents dans un dataset. Cependant l'algorithme choisie possède des lacune en terme de complexité temporelle (lors de la génération des candidats la jointure effectuée est très coûteuse en temps), pour plus de détails, une section d'analyse de cette approche sera introduite dans le dernier chapitre.

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 est l'extrémité d'un vecteur :

$$\vec{V} = a_1.\vec{v_1} + a_2.\vec{v_2} + ... + a_n.\vec{v_n}$$

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 $X(x_1, ..., x_n)$ et $Y(y_1, ..., y_n)$

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

La distance euclidienen 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 à $D_p(P, v) = minDst$ où $minDst$ est la plus petite distance entre deux points, alors V est l'ensemble des voisins directes de P.

3.2.4 Classification

Da 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 :

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

3.2.5 Précision

Pour un classifier F, nous somme souvent amené a 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 un 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 prediction est correcte
Alors la précision par rapport au classifier 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'autre 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 donnée

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 performance 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 :

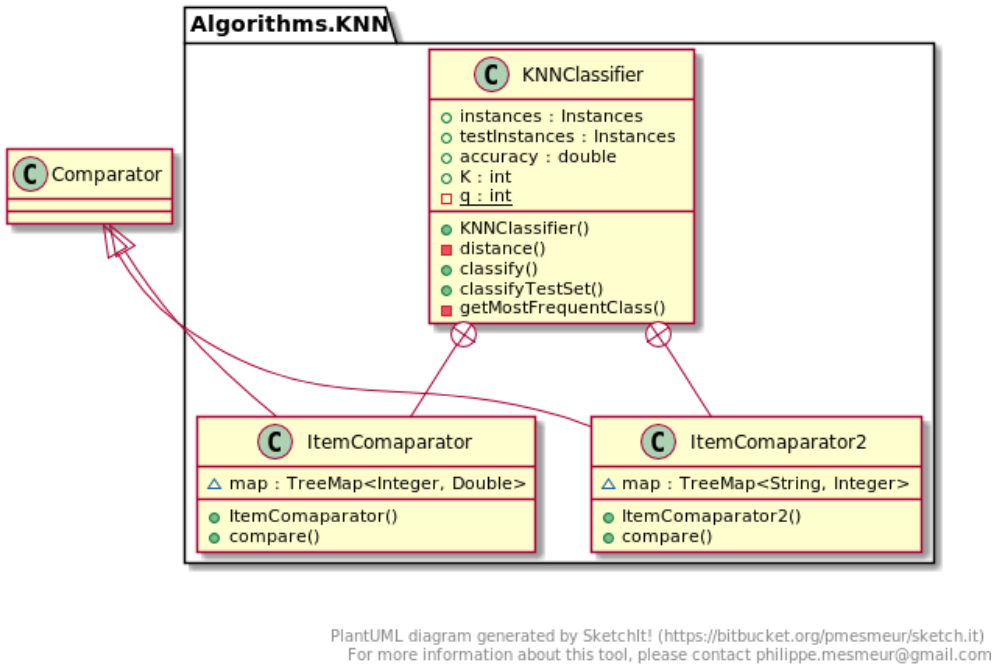
Algorithme 2 : KNN

Entrée : (E : Ensemble des instances d'apprentissage, X : instance à classifier , 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**
| $Distances \leftarrow Distances \bigcup \{(e, D_2(X, e))\}$;
fin
$Distances\_tri \leftarrow$ `TrierParPlusPetiteDistance(`$Distances$`)` ;
$D_K \leftarrow \{D_i \in Distances\_tri | i = 1..K\}$;
$C \leftarrow$ `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 paramètre 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

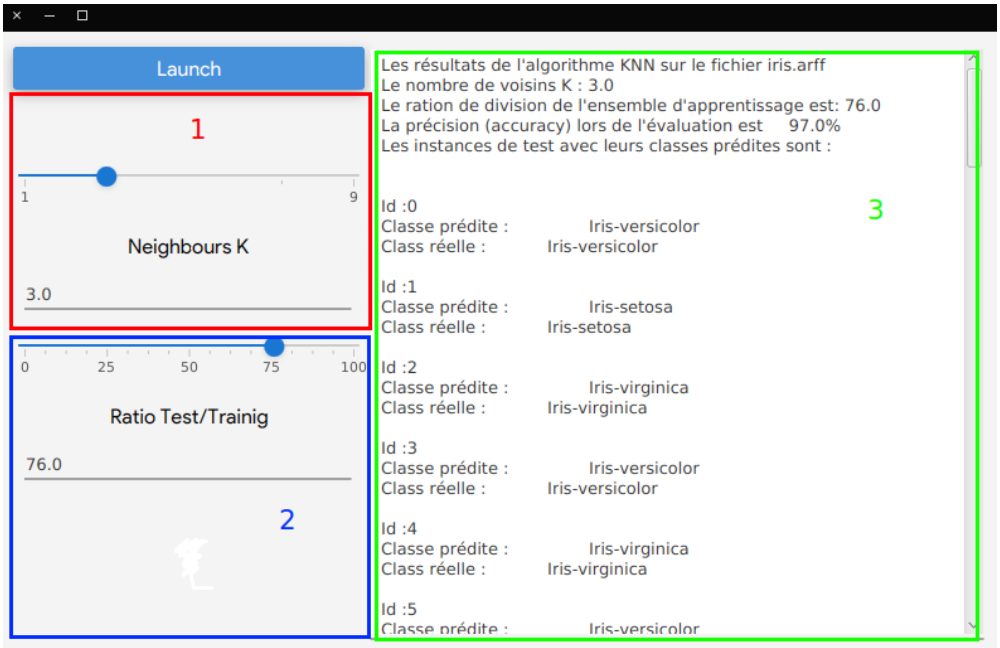
- Les méthodes implémentées sont les suivantes :

```
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
  - 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 suivit 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***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 serons 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 :

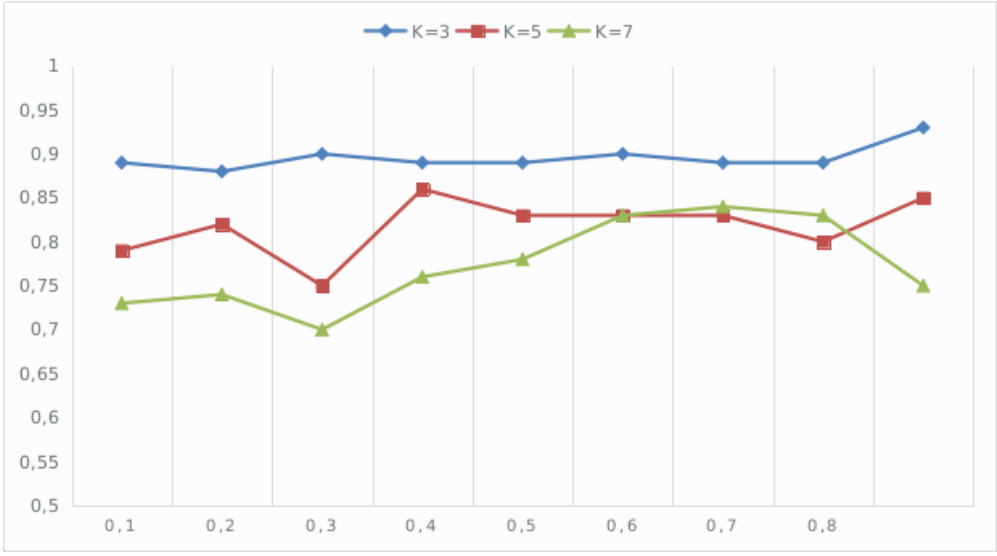


FIGURE 3.1 – Graphes comparatif des résultats de l'algorithme sur le dataset car.arff

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 classifier 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

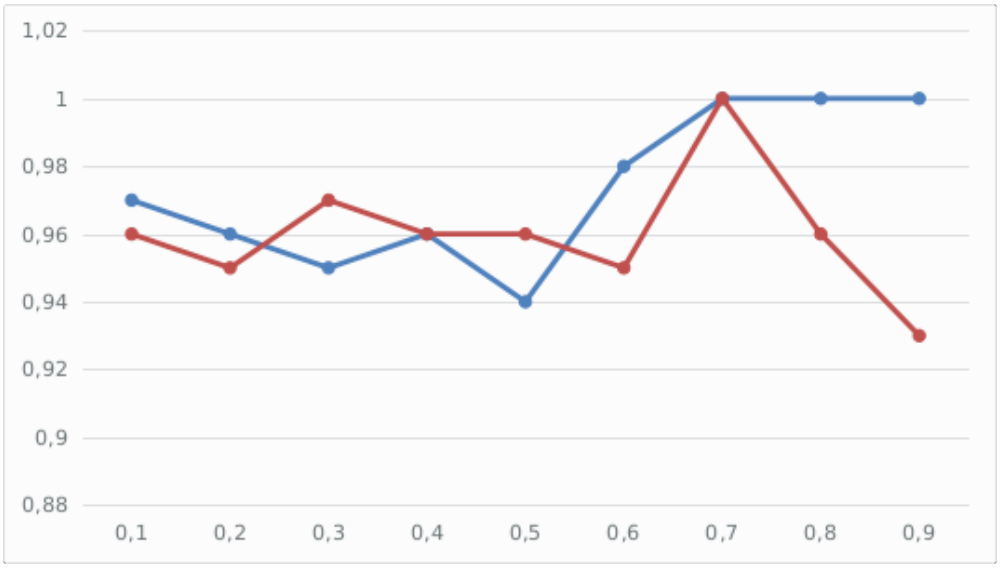


FIGURE 3.2 – Graphes comparatif des résultats de l'algorithme sur le dataset iris.arff

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, certe d'un petit pas (de l'ordre de 0.08), mais à plus grande échelle cette quantité deviendrait cruciale.

Il est aussi a noté 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

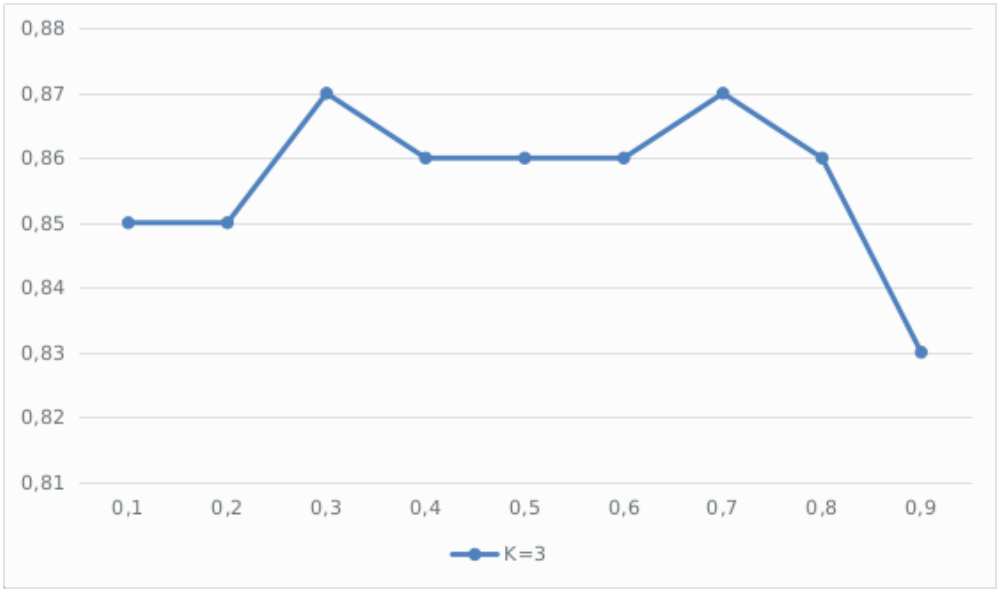


FIGURE 3.3 – Graphes comparatif des résultats de l'algorithme sur le dataset credit-g.arff

Commentaires : De manière analogue au dataset précédent, du point de vue intrinsèque au choix de K, laa précision peu 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 a noté que la classification à pris un temps assez considérable pour(peut être du à la présence d'attributs nominales ?)

3.7 Conclusion

CHAPITRE 4

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

4.1 Introduction

4.2 Définitions

4.2.1 Point

4.2.2 Distance

4.2.3 Voisinage

4.2.4 Core-point

4.2.5 Point de bord (Border-point)

4.2.6 Bruit

4.2.7 Cluster

4.3 Algorithme

4.4 Implémentation

4.4.1 Langage de programmation

4.4.1.1 Schémas d'exécution

4.4.1.2 Structures de données

4.5 Interface graphique

4.6 Résultats expérimentaux

4.6.0.1 Choix du dataset

4.6.0.2 Variations des paramètres

4.6.0.3 Résultats

4.6.0.4 Commentaires

4.7 Conclusion

CHAPITRE 5

CONCLUSION

5.1 Bilan récapitulatif

Partie I

Partie II

Partie III

5.2 Critiques

5.3 Perspectives futures

BIBLIOGRAPHIE

[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.