SORBONNE UNIVERSITE

ANNEE 2019-2020

Makhatch ABDULVAGABOV, Florent VAlBON, Xiaoyuan WANG

PROJET DE C++

Table des matières

Introduction

Si on note C_{ij} l'ensemble des clients achetant deux produits i, et j, alors on appelle *corrélation*, le cardinal de cet ensemble. L'objectif de ce projet est de déterminer, algorithmiquement, la corrélation entre deux produits achetés par un même client.Notre algorithme doit être capable de déterminer l'ensemble des 100 couples les plus corrélés. Le problème est que si le nombre de données est trop élevé, l'algorithme sera beaucoup trop lent.

Dans un premier temps, nous analyserons la vitesse de l'algorithme dans le cas où il n'est pas optimisé : c'est à dire le cas où toutes les corrélations sont calculées. Dans un second temps, nous expliquerons l'heuristique statistique, permettant à l'algorithme de retourner le bon résultat, sans pour autant avoir à calculer toutes les corrélations. Enfin, nous comparerons les performances de l'algorthme non optimisé, à celui obtenue avec l'heuristique.

1 Algorithme

Dans cette partie, on présente l'algorithme de calcul des corrélations.

1.1 Présentation des fonctions

1.1.1 Fonction read_data

Avant tout on commence par lire et stocker les données. Pour les stocker plusieurs possibilités ont étés envisagées. Mais notre choix s'est arrêté sur les multmap pour cet usage. En plus d'avoir la possibilité de cumuler les clés de même valeurs, la multimap les classe directement dans l'ordre (ici décroissant). Voici le code de la fonction qui lit et stocke les données :

```cpp
#include "fonctions.hpp"

using namespace std;

//Fonction qui lit le fichier avec les donnees et les stocke dans une multimap
//les donnes sont regrouppees de maniere suivante :
// - A chaque produit on associe tableau des clients qui l'ont achete, avec pair<string, vector<string>>
// - Les couples (produit, tableau des clients) sont ensuite stockes
//      dans l'ordre decroissant selon la taille du tableau des clients, dans une multimap
multimap<int, pair<string, vector<string> >, greater<int> > read_data(const char* name)
{
        ifstream file(name);
        map<string, vector<string> > temp;
        int kp = 0, ka = 0;
        string client, produit, pointvirgule;
        while(!file.eof())
        {
                file >> client;
                file >> pointvirgule;
                file >> produit;
                ka++;
                if(file.eof()) break;
                if(temp.find(produit) == temp.end())
                {
                        (temp[produit]).push_back(client);
                        kp++;
                }
                else
                        (temp[produit]).push_back(client);
        }
        multimap<int, pair<string, vector<string> >, greater<int> > data;
        for(map<string, vector<string> >::iterator it = temp.begin(); it != temp.end(); it++)
                data.insert(make_pair((it -> second).size(), make_pair(it -> first, it -> second) ) );
        return data;
}

```

1.1.2 Fonction print_map_data

Une fois les données stockées, on peut avoir besoin de les afficher dans le terminal. Pour tester la fonction read_data par exemple. Pour cela, on définit une fonction qui parcourt la multimap avec un itérateur et qui affiche ses valeurs.

```
56
57 //Fonction qui affiche un par un les produits avec tous les clients qui les ont achetes
58 void print_map_data(multimap<int, pair<string, vector<string> >, greater<int> > &data)
59 {
60         map<int, pair<string, vector<string> >, greater<int> >::iterator itm;
61         for(itm = data.begin(); itm != data.end(); itm++)
62         {
63                 cout << "le " << (itm -> second).first << " a ete achete par :\n";
64                 for(vector<string>::iterator itv = (itm -> second).second.begin(); itv != (itm -> second).second.end(); itv++)
65                         cout << "\t le " << *itv << endl;
66                 cout << endl << endl << endl;
67         }
68 }
69
70
```

Le première valeurs du test de cette fonction avec le fichier data-100.csv donne

```
le produit_7 a ete achete par :
         le client_3
         le client_1
         le client_1
         le client_5
         le client_8
         le client_1
         le client_2
         le client_2
         le client_2
         le client_8
         le client_8
         le client_3
         le client_6
         le client_9
         le client_7
         le client_8



le produit_5 a ete achete par :
         le client_0
         le client_6
         le client_7
         le client_3
         le client_1
         le client_4
         le client_0
         le client_9
         le client_7
         le client_8
         le client_3
         le client_9
         le client_7



le produit_8 a ete achete par :
         le client_6
         le client_9
         le client_3
         le client_1
         le client_0
         le client_7
         le client_1
         le client_3
         le client_3
         le client_2
         le client_0
         le client_5
         le client_9
```

1.1.3 Fonction card_inter

Le but du projet étant de calculer les corrélations (corrélation étant définie comme le cardinal de l'intersection entre les ensembles des clients de deux produits différents), on a besoin d'une fonction qui pour deux tableaux (ici vector) donnés calcule le nombre de clients qui apparaissent dans les deux tableaux.

```
83
84 //Fonction qui pour deux tableaux de donnees (clients) retourne le cardinal de leur intersection
85 int card_inter(vector<string> &U, vector<string> &V)
86 {
87         int corr = 0;
88         vector<string> min,max;
89         if(U.size() >= V.size())
90         {
91                 max = U;
92                 min = V;
93         }
94         else
95         {
96                 max = V;
97                 min = U;
98         }
99         for(vector<string>::iterator itv = min.begin(); itv != min.end(); ++itv)
100        {
101                if(find(max.begin(), max.end(), *itv) != max.end())
102                        corr++;
103        }
104        return corr;
105 }
106
```

1.1.4 Fonction fnt_corr

Une fois qu'on a toutes les fonctions nécessaires, on peut coder la fonction qui va calculer et ranger les corrélations dans l'ordre décroissant. Dans la fonction fnt_corr on crée et retourne une nouvelle multimap qui va contenir comme clé les corrélations et le couple produi_i produit_j pour chaque clé. Comme les clé sont des corrélations, on n'aura pas besoin de faire un tri, car la map trie les clés automatiquement.

```
133
134 //Fonction qui retourne une map rangee de correlations avec les combinaisons ij
135 multimap<int, pair<string, string>, greater<int> > fnt_corr( multimap<int, pair<string, vector<string> >, greater<int> > &data, int m)
136 {
137         int card;
138         multimap <int, pair<string, string>, greater<int> > res;
139         //On parcourt les boucles pour faire les combinaisons ij
140         multimap <int, pair<string, vector<string> >, greater<int> >::iterator it1;
141         for(it1 = data.begin(); it1 != data.end(); it1++)
142         {
143                 multimap <int, pair<string, vector<string> >, greater<int> >::iterator it2;
144                 for(it2 = it1; it2 != data.end(); it2++)
145                 {
146                         if(it1 != it2) //On ne calcule pas la correlation si on a deux fois le meme tableau
147                         {
148                                 card =  card_inter((it1 -> second).second , (it2 -> second).second );
149                                 res.insert(make_pair(card, make_pair((it1 -> second).first , (it2 -> second).first ) ));
150                         }
151                 }
152
153         }
154         return res;
155 }
156
```

1.1.5 Fonction print_max_corr

Les corrélations sont déjà stockées et rangés dans l'ordre, il ne manque plus qu'une fonction pour les afficher dans le terminal. La fonction print_max_corr a été codée pour cette opération.

```
174
175 //Fonction qui affiche les m plus grandes correlations de produits
176 void print_max_corr(multimap<int, pair<string, string >, greater<int> > &res, int m)
177 {
178         int nb_combi = 0;
179         multimap<int, pair<string, string >, greater<int> >::iterator it = res.begin();
180         while(it != res.end() && nb_combi < m)
181         {
182                 cout << "correlation = " << it -> first << endl;
183                 cout << "\tpour le produit " << (it -> second).first << endl;
184                 cout << "\tpour le produit " << (it -> second).second << endl << endl;
185                 nb_combi++;
186                 it++;
187         }
188 }
189
```

Le test de cet algorithme donne pour le jeu de donnée data-100000.cvs avec les premiers résultats.

```
correlation = 233
        pour le produit produit_875
        pour le produit produit_381

correlation = 231
        pour le produit produit_875
        pour le produit produit_623

correlation = 230
        pour le produit produit_875
        pour le produit produit_876

correlation = 228
        pour le produit produit_122
        pour le produit produit_120

correlation = 227
        pour le produit produit_875
        pour le produit produit_368

correlation = 226
        pour le produit produit_367
        pour le produit produit_627

correlation = 225
        pour le produit produit_129
        pour le produit produit_376

correlation = 224
        pour le produit produit_381
        pour le produit produit_129

correlation = 224
        pour le produit produit_876
        pour le produit produit_882

correlation = 224
        pour le produit produit_629
        pour le produit produit_131

correlation = 223
        pour le produit produit_875
        pour le produit produit_129
```

Sur un jeu de donnée de cette taille le temps de calcul commence déjà à paraître long. Il a fallu 38 seconde pour obtenir ce résultat.

2 Optimisation de l'algorithme initial

2.1 Description de l'algorithme

Le problème principal de l'algorithme précédent c'est qu'il calcule la corrélation pour toutes les combinaisons possibles. Pour les fichiers de petites taille, ça ne pose pas de problèmes, mais à partir d'une certaine taille des données, le temps pris par les calculs devient remarquable. L'énoncé ne nous demande que les 100 couples les plus corrélées, il est donc inutile de calculer toutes les corrélations éxistantes. Pour diminuer le temps du calcul on vise à calculer le minimum possible de corrélation pour obtenir le résultat nécessaire. Voici une optimisation possible : Le

conteneur multimap a été choisi pour stocker les donnée justement en vue de pouvoir optimiser les calculs après. Dans la multimap data les clés sont les tailles des tableaux des clients ayant acheté un certain produit. On ne veut que les 100 couples de produits les plus corrélés. Donc quand on fait les combinaison des produits, comme les tableaux des clients qui leurs sont associés sont classés dans l'ordre décroissant, à partir d'un moment la taille de plus petit tableau du couple produit_i, produit_j sera inférieur à la 100-ème corrélation stockée. Il sera donc inutile de continuer les calculs car l'intersection des tableaux associer au couple de produits ne pourra pas être plus grande que la 100-ème corrélation.

Pour appliquer cette optimisation à notre algorithme précédent, on aura besoin de pouvoir accèder à la 100-ième valeur d'une map si elle existe. Comme les map sont des objets dont les valeurs sont accéssibles par les clés et non pas par les indices, on crée la fonction suivant pour réaliser cette tâche.

```
111
112 //Focntion qui retourne la correlation du m-ieme element de la map
113 int corr_ind_m(multimap<int, pair<string, string>, greater<int> > &res, int m)
114 {
115         assert(res.size() >= m);
116         multimap<int, pair<string, string>, greater<int> >::iterator it = res.begin();
117         int x;
118         for(int k = 0; k < m-1; k++)
119         {
120                 it++;
121         }
122         x = (*it).first;
123         return x;
124 }
125
```

2.2 Algorithme optimisé

Ayant tous les outils nécessaires pour faire l'optimisation on peut modifier la fonction fnt_corr en suivant l'algorithme plus optimal ci-dessus.

```
133
134 //Fonction qui retourne une map rangee de correlations avec les combinaisons ij
135 multimap<int, pair<string, string>, greater<int> > fnt_corr( multimap<int, pair<string, vector<string> >, greater<int> > &data, int m)
136 {
137         int card;
138         multimap <int, pair<string, string>, greater<int> > res;
139         //On parcourt les boucles pour faire les combinaisons ij
140         multimap <int, pair<string, vector<string> >, greater<int> >::iterator it1;
141         for(it1 = data.begin(); it1 != data.end(); it1++)
142         {
143                 multimap <int, pair<string, vector<string> >, greater<int> >::iterator it2;
144                 for(it2 = it1; it2 != data.end(); it2++)
145                 {
146                         if(it1 != it2) //On ne calcule pas la correlation si on a deux fois le meme tableau
147                         {
148                                 if(res.size() >= m)
149                                 {
150                                         //      Si la m-ieme correlation est plus grande que la taille du tableau
151                                         //      c'est plus la peine de continuer car les tableaux sont ranges par ordre de taille et
152                                         //      leur intersection ne peut pas etre plus grande que la m-ieme correlation
153                                         if(corr_ind_m(res,m) > min(it1 -> first, it2 -> first))
154                                         {
155                                                 return res;
156                                         }
157                                 }
158                                 card =  card_inter((it1 -> second).second , (it2 -> second).second );
159                                 res.insert(make_pair(card, make_pair((it1 -> second).first , (it2 -> second).first ) ));
160                         }
161                 }
162
163         }
164         return res;
165
166 }
167
```

On test maintenant l'algorithme optimisé sur le même jeu de données que précédement :

```
correlation = 233
        pour le produit produit_875
        pour le produit produit_381

correlation = 231
        pour le produit produit_875
        pour le produit produit_623

correlation = 230
        pour le produit produit_875
        pour le produit produit_876

correlation = 228
        pour le produit produit_122
        pour le produit produit_120

correlation = 227
        pour le produit produit_875
        pour le produit produit_368

correlation = 226
        pour le produit produit_367
        pour le produit produit_627

correlation = 225
        pour le produit produit_129
        pour le produit produit_376

correlation = 224
        pour le produit produit_381
        pour le produit produit_129

correlation = 224
        pour le produit produit_876
        pour le produit produit_882

correlation = 224
        pour le produit produit_629
        pour le produit produit_131

correlation = 223
        pour le produit produit_875
        pour le produit produit_129
```

On obtient exactement le même résultat, ce qui est rassurant. Mais surtout on l'obtient plus rapidement. Même si le gain du temps n'est que de 4 secondes, car il a fallu 34 secondes au nouveau algortihme pour faire les calculs. Pour un jeu de donnée plus massif, le gain de temps pourraît être encore plus signifiant.

Cet algorithme, bien que plus optimal que l'algorithme précédent n'est pour autant pas suffisamment rapide pour résoudre le problème dans un délai raisonnable, dans le cas où les dimensions des données deviennents trop importantes. Puisque la force brute ne suffit plus, il faut adopter une autre stratégie. On pourrait par exemple choisir une heuristique statistique.

Une première idée était de ne garder qu'une valeur sur deux pour produits et clients lors de la lécture des données. Si leur distribution dans les données se font de manière aléatoire, on pourrait espérer de garder la proportionnalité avec cette méthode. Cependat, en pratique, on se rend compre assez rapidement que cette conjecture est fausse. Trop d'information est pérdue et en comparant les résulats obtenues avec et sans cette méthode pour un même jeu de données, on remarque que les couples les plus corrélées ne sont pas les mêmes.