ANNÉE 2019-2020

Makhatch Abdulvagabov, Florent Valbon, Xiaoyuan Wang

PROJET DE C++

Détermination des 100 produits les plus corrélés d'un data set

---

Sommaire

Introduction

Si on note C_{ij} l'ensemble des clients achetant deux produits i et j, alors on appelle *corrélation* de ces deux produits, le cardinal de cet ensemble. L'objectif de ce projet est de déterminer algorithmiquement la corrélation entre deux produits achetés par un même client. Nous adopterons la convention suivante : si un couple de produits est acheté $m > 1$ fois par un seul client, alors la contribution de ce client sur la corrélation sera égale à m. Ce qui est discutable d'un point de vue statistique, mais au moins nous ne perdons pas d'informations.

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. En effet, pour un data set de $N \to +\infty$ données, le nombre de corrélations à calculer avant de trouver les 100 premières est de :

$$\binom{N}{2} = \frac{N(N-1)}{2} = O(N^2).$$

Un algorithme se basant sur une méthode qui viserait à calculer toutes les corrélations aurait donc $O(N^2)$ corrélations à calculer : ce qui est potentiellement énorme pour un data set de plus de 100000 données. Si de plus chaque produit est stocké avec les clients qui lui sont associés, la complexité de l'algorithme sera très grande.

Après une présentation de l'algorithme, nous analyserons sa vitesse 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'algorithme non optimisé, à celles obtenues avec l'heuristique.

L'ensemble de l'algorithme est écrit en langage C++ (qui est un langage efficace lorsque des data sets contenant beaucoup de données doivent être traités).

1 Présentation générale de l'algorithme de calcul des 100 premières corrélations

Dans cette partie, on présente l'algorithme de tri des 100 premières corrélations.

1.1 Présentation des fonctions

Affichage des valeurs

Voici l'algorithme utilisé pour afficher les résultats dans le terminal, et dans un data set.

```
int nb_combi = 0;
        int rang = 1;


        priority_queue< pair<int, pair<string, string> > >
        res(fnt_corr_optim(data, tau));



        pair<int, pair<string, string> > top;
        ofstream output("output.csv");
        output<< "rang" << " ; " << "couple" << " ; " << "corrélation"<< endl;
        while(! res.empty() && nb_combi <= 99)
        {
            top = res.top();
            cout << "rang : "<< rang << ", ";
                cout << "corrélation = " << top.first << endl;
                cout << endl;
                cout << "\t   pour le : " << (top.second).first << endl;
                cout << "\tet pour le : " << (top.second).second;
                cout << endl << endl;
                output<< rang << " ; " << "(" <<(top.second).first << "," <<
                (top.second).second <<")"<< " ; " << top.first << endl;
                res.pop();
                rang++;
                nb_combi++;
        }
```

Fonction `card_inter`

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

Arguments :

— `U` : vecteur de chaines de caractères.

— `V` : vecteur de chaines de caractères.

Sortie : Un entier.

Explication :

Cette fonction retourne la corrélation entre deux produits `U` et `V`.

```
int card_inter(vector<string> &U, vector<string> &V)
{
        int corr = 0;
        vector<string> min,max;
        if(U.size() >= V.size())
        {
                max = U;
                min = V;
        }
        else
        {
                max = V;
                min = U;
        }
        for(vector<string>::iterator itv = min.begin(); itv != min.end(); ++itv)
        {
                if(find(max.begin(), max.end(), *itv) != max.end())
                        corr++;
        }
        return corr;
}
```

Fonction `fnt_corr_optim`

Arguments :

— `data` : `map` décrivant l'ensemble

$$(\text{produit}_i, \textit{clients ayant achetés le produit}_i)_{i\in[produit_0,produit_{nombre\ de\ produits+1}]}.$$

Sortie : Un entier.

Explication :

Cette fonction retourne les corrélations calculées de chacun des couples de produits `(U,V)`, en les classant du moins corrélé au plus corrélé.

1.2 Objets utilisés

Présentons les objets que nous avons utilisés pour cet algorithme.

— `data` : `map` décrivant l'ensemble

$$(produit_i, clients\ ayant\ achetés\ le\ produit_i)_{i\in[produit_0, produit_{nombre\ de\ produits-1}]}.$$

Cet objet est utilisé par l'algorithme pour lire et stocker les données. Pour ce faire, plusieurs possibilités ont étés envisagées. Mais notre choix s'est porté sur les `map` pour cet usage. En effet, en plus d'avoir la possibilité de stocker les produits, la `map` les classe directement selon leurs indices. Voici le code de la `map data` :

```
map<string, vector<string> > data;
//data = (produit, clients achetant le produit)
ifstream file(argv[1]);
int  kp = 0, ka = 0;
string  client, produit, pointvirgule;
while(!file.eof())
{
        file >> client;
        file >> pointvirgule;
        file >> produit;
        ka++;

        if(file.eof()) break;
        if(data.find(produit) == data.end())
        {
                (data[produit]).push_back(client);
                kp++;
        }
        else
                (data[produit]).push_back(client);

}
```

— `data_nprod` : L'objet suivant sera utilisé pour l'optimisation de l'algorithme (dans la fonction `fnt_corr_optim`). Il n'est pas utilisé dans un premier temps. Il s'agit d'une `multimap` décrivant l'ensemble

$$(n_i, produit_i)_{i\in[produit_0, produit_N]}.$$

Où n_i désigne le nombre de clients ayant achetés le $produit_i$, et $N+1$ le nombre de produits du data set.

```
multimap<int,string, greater<int> > data_nprod;

        for(map<string, vector<string> >::iterator it = data.begin();
        it != data.end(); ++it)
        {
                data_nprod.insert( pair<int,string>( (it->second).size() ,
                (it->first) ) );
        }
```

1.3 Démarche générale

Dans cette partie, nous présentons le fonctionnement général de l'algorithme.

Entrée :

Un fichier de données de $N+1$ lignes, répertoriant les clients et leurs achats de la façon suivante :

$client_{i0}$; $produit_{i0}$
$client_{i1}$; $produit_{i1}$
...
$client_{iN}$; $produit_{iN}$

Sortie :

Un fichier de données de 101 lignes, renvoyant un fichier de la forme :

rang ; couples de produit ;

1 $(produit_{n0}, produit_{m0})$; $corrélation_1$

2 $(produit_{n1}, produit_{m1})$; $corrélation_2$

...

N $(produit_{nN}, produit_{mN})$; $correlation_N$

et un affichage dans le terminal du temps d'exécution du programme.

Étapes :

1. Début du calcul du temps d'exécution du programme : nous avons recours à la librairie `time.h` pour donner le temps d'exécution de l'algorithme à la seconde près ;
2. Création de la map `data` ;
3. Affichage des 100 premières corrélations : nous calculons toutes les corrélations possible à l'aide de la fonction `fnt_corr(data)` ;

4. Fin du calcul du temps d'exécution du programme.

1.4 Résultats obtenus

Voici les résultats obtenus pour les quatre data sets que nous avons utilisés pour notre algorithme : Nous affichons ici les 25 premières valeurs obtenues par notre algorithme.

Data set de 100 données :

Les 25 premières valeurs sont :

rang	couple	corrélation
1	(produit_7,produit_8)	11
2	(produit_5,produit_8)	11
3	(produit_5,produit_6)	11
4	(produit_7,produit_5)	10
5	(produit_5,produit_3)	10
6	(produit_7,produit_6)	9
7	(produit_8,produit_9)	7
8	(produit_8,produit_6)	7
9	(produit_8,produit_1)	7
10	(produit_7,produit_9)	7
11	(produit_7,produit_3)	7
12	(produit_7,produit_1)	7
13	(produit_5,produit_9)	7
14	(produit_5,produit_10)	6
15	(produit_3,produit_6)	6
16	(produit_1,produit_9)	6
17	(produit_8,produit_3)	5
18	(produit_8,produit_10)	5
19	(produit_8,produit_0)	5
20	(produit_7,produit_0)	5
21	(produit_5,produit_1)	5
22	(produit_5,produit_0)	5
23	(produit_3,produit_9)	5
24	(produit_3,produit_10)	5
25	(produit_3,produit_1)	5

data set de 100 données

Le temps d'exécution est de 0 secondes. Ce qui est très rapide.

Data set de 1000 données

Les 25 premières valeurs sont :

rang	couple	corrélation
1	(produit_13,produit_64)	24
2	(produit_89,produit_88)	22
3	(produit_13,produit_63)	22
4	(produit_65,produit_39)	21
5	(produit_65,produit_37)	21
6	(produit_89,produit_65)	19
7	(produit_64,produit_63)	19
8	(produit_64,produit_39)	19
9	(produit_64,produit_38)	19
10	(produit_13,produit_87)	19
11	(produit_89,produit_87)	18
12	(produit_89,produit_63)	18
13	(produit_89,produit_62)	18
14	(produit_65,produit_90)	18
15	(produit_65,produit_38)	18
16	(produit_13,produit_89)	18
17	(produit_13,produit_88)	18
18	(produit_13,produit_65)	18
19	(produit_13,produit_39)	18
20	(produit_62,produit_37)	17
21	(produit_38,produit_90)	17
22	(produit_13,produit_90)	17
23	(produit_13,produit_37)	17
24	(produit_89,produit_61)	16
25	(produit_64,produit_65)	16

data set de 1000 données

Le temps d'exécution est de 0 secondes. Ce qui est très rapide. Il est cohérent que les corrélations obtenues n'excèdent pas 30, puisque le data set ne contient que 1000 données. Une corrélation supérieure à 100 serait suspecte car elle indiquerait un couple de produits beaucoup trop souvent acheté. Enfin, tout naturellement, les corrélations ne sont pas trop éloignées les unes des autres. L'inverse, pourrait indiquer (sauf exception pour un couple de de produits très sollicité) que l'algorithme "oublie" des couples.

Data set de 100 000 données

Les 25 premières valeurs sont :

rang	couple	corrélation
1	(produit_875,produit_381)	233
2	(produit_875,produit_623)	231
3	(produit_875,produit_876)	230
4	(produit_122,produit_120)	228
5	(produit_875,produit_368)	227
6	(produit_367,produit_627)	226
7	(produit_129,produit_376)	225
8	(produit_876,produit_882)	224
9	(produit_629,produit_131)	224
10	(produit_381,produit_129)	224
11	(produit_876,produit_621)	223
12	(produit_875,produit_129)	223
13	(produit_629,produit_878)	223
14	(produit_882,produit_368)	222
15	(produit_875,produit_128)	222
16	(produit_629,produit_877)	222
17	(produit_381,produit_633)	222
18	(produit_381,produit_367)	222
19	(produit_875,produit_628)	221
20	(produit_875,produit_377)	221
21	(produit_627,produit_625)	221
22	(produit_367,produit_628)	221
23	(produit_882,produit_126)	220
24	(produit_381,produit_868)	220
25	(produit_367,produit_128)	220

data set de 100 000 données

Le temps d'exécution est de 28 secondes.

Data set de 1000000 données

Les 25 premières valeurs sont :

rang	couple	corrélation
1	(produit_8686,produit_8765)	52
2	(produit_6243,produit_6132)	52
3	(produit_1202,produit_8708)	50
4	(produit_8768,produit_8728)	49
5	(produit_6167,produit_6277)	48
6	(produit_3773,produit_8809)	48
7	(produit_3757,produit_1307)	48
8	(produit_3755,produit_1155)	48
9	(produit_3699,produit_3828)	48
10	(produit_3664,produit_3750)	48
11	(produit_1312,produit_8800)	48
12	(produit_1307,produit_1298)	48
13	(produit_1237,produit_3662)	48
14	(produit_1210,produit_6298)	48
15	(produit_8812,produit_8738)	47
16	(produit_8767,produit_3739)	47
17	(produit_8738,produit_6360)	47
18	(produit_8699,produit_8661)	47
19	(produit_3818,produit_6300)	47
20	(produit_3810,produit_8762)	47
21	(produit_3748,produit_8797)	47
22	(produit_1333,produit_6342)	47
23	(produit_1270,produit_8786)	47
24	(produit_1254,produit_3680)	47
25	(produit_1184,produit_3753)	47

data set de 1 million de données

Le temps d'exécution est de 1 h 04. Ce qui est relativement élevé. Dans la pratique, un tel temps d'exécution serait beaucoup trop long, surtout si l'algorithme est utilisé pour faire du Machine Learning par exemple. Nous allons donc l'optimiser.

2 Optimisation de l'algorithme "force brute"

On se base premièrement sur le fait que pour deux produits $produit_i$ et $produit_j$, achetés respectivement n_i et n_j fois,

$$\text{Correlation}(produit_i, produit_j) \leqslant n_i.$$

Les produits de la `multimap` `data_nprod`, sont rangés dans l'ordre décroissant du nombre de ventes. Commençons par calculer la corrélation de chaque couple de la `multimap`. La `multimap` a l'avantage de pouvoir stocker dans l'odre croissant plusieurs clés, même si elles sont égales, contrairement à une `map`. Cela nous permet de stocker les produits dans l'ordre décroissant du nombre de ventes.

Effectuons des itérations sur (i, j) dans la `multimap` `data_nprod` :
Soit $L_{(i,j)}$ une liste correspondant aux 100 plus grandes corrélations calculées à l'itération (i, j).

À i_0 fixé, si pour $j \geqslant i_0$,

$$n_{i0} \leqslant \min(\text{correlation} \in L_{(i_0,j)}),$$

alors, du fait de l'organisation de `data_nprod`, il ne sera pas possible de trouver une corrélation plus élevée aux itérations $i_0 + n$, n entier strictement positif (puisque les produits suivant sont moins souvent achetés). Nous imposons donc ce critère d'arrêt à l'algorithme.

Cela nous assure, que même si nous n'avons pas calculé toutes les corrélations, il sera impossible d'obtenir une corrélation plus élevée parmi les valeurs que nous n'avons pas encore calculées.

Cela évite à l'algorithme de calculer des corrélations qu'il n'est pas utile de calculer (ce qui est coûteux en temps de calcul).

Notons que nous avons préféré utiliser le type `list` qui trie des valeurs plus rapidement que le type `vector`.

La fonction `mysort` permet à l'algorithme de trier astucieusement (et donc rapidement) la liste $L_{(i,j)}$ à chaque itération (i, j), car elle tient compte du fait que la liste est déjà triée à l'itération précédente.

Deuxièmement, nous nous basons sur l'intuition suivante : avec forte probabilité, les produits les plus achetés seront ceux dont la corrélation sera la plus forte.

On note τ le nombre de produits dont on calcule, à chaque itération $i \leqslant \tau$, toutes les $\tau - i - 1$ premières corrélations non calculées, avant l'arrêt de l'algorithme.

Par exemple, si notre data set contient 100 produits classés dans l'ordre décroissant de leur nombre de ventes, l'algorithme calculera les corrélations entre le premier produit et les τ premiers autres, puis celles du deuxième produit et des τ premiers autres, ainsi de suite, puis enfin celles du produit numéro τ et des τ premiers autres avant de s'arrêter.

Dans notre data set de un million de données, cette méthode (le fait de fixer τ suffisement petit) retourne les 100 meilleures corrélations en 25 minutes.

Cette méthode étant probabiliste, nous demandons à l'utilisateur de l'algorithme de choisir τ, en fonction de si il souhaite attendre (pour avoir le résultat le plus précis possible) ou non. D'autre part, l'utilisateur pourra relancer l'algorithme un certain nombre de fois, jusqu'à sa convergence. Notons enfin que pour que la vitesse de l'algorithme sera très sensible à la valeur de τ pour un data set avec beaucoup de données.

3 Comparaison de l'algorithme non optimisé avec l'algorithme optimisé

Voici les comparaisons des temps d'exécutions de l'algorithme optimisé par rapport à celui non optimisé :

Nb. de données	algo. non optimisé	algo. optimisé	τ	nb. valeurs exactes
100	0 s	0 s	nb. produits	100
1000	0 s	0 s	nb. produits	100
100 000	28 s	4 s	100	100
1 000 000	54 min	20 min	1500	100

comparaison des performances entre les algorithmes oprimisé et non optimisé

Pour le data set de un million de données, le gain de rapidité est fort (1,9 fois plus rapide), mais moins fort que pour 100000 (4 fois plus rapide). Si on veux un résultat exact, il est difficile de gagner encore en rapidité sur le data set de 1 million de données : pour $\tau = 1000$ (19 min), certaines valeurs sont encore manquantes.

Conclusion

L'heuristique statistique que nous avons utilisée permet à l'algorithme de calcul des 100 premières corrélations de gagner en vitesse de calcul. La vitesse de calcul peut être augmentée à l'infini, presque (en diminuant le paramètre τ, mais au prix de la certitude des résultats en sortie).
Notons que si nous arrivons à calculer un intervalle de confiance sur les valeurs obtenues en sortie en fonction du paramètre τ et du nombre de données, alors nous pourrons contrôler la précision de l'algorithme, et diminuer le nombre de fois que l'utilisateur relancera l'algorithme, ou encore faire fonctionner cet algorithme automatiquement (dans un algorithme de Machine Learning par exemple).