ÉCHANTILLONNAGE DE GRANDES DONNÉES ET APPLICATIONS : SHAKESPEARE, LE WEB ET LES RÉSEAUX

SUJET PROPOSÉ PAR BRUNO SALVY
VERSION 1.2
DIFFICULTÉ : DE MOYENNE À DIFFICILE

Résumé. Pour des données très grosses et qui nécessitent des traitements très rapides, l'exactitude peut avoir un coût rédhibitoire. En revanche, des algorithmes probabilistes peuvent apporter des réponses approchées de très bonne qualité avec une très faible consommation en mémoire et en temps. Des applications naturelles sont la surveillance de réseau (dimensionnement ; détection de virus, de vers, d'attaques par déni de service), l'optimisation de requêtes dans des bases de données, ou la recherche de similitudes sur des grands ensembles de documents.

Les quantités de données qui nécessitent un traitement informatique croissent de manière explosive. De plus en plus fréquemment, des informations exactes sur ces données deviennent inaccessibles en temps ou en espace mémoire praticables. Cependant, dans bien des contextes, une information correcte à quelques pour cent suffit et peut-être obtenue par des techniques élégantes et non-triviales d'échantillonnage.

Par exemple, les moteurs de recherche indexent des milliards de pages et ne peuvent pas se permettre de les parcourir trop souvent. Les pages qui sont des variantes les unes des autres leur coûtent du stockage et nuisent à la clarté de leurs réponses. Il leur faut donc les repérer rapidement, et les contraintes de stockage sont cruciales.

Un autre contexte est fourni par l'ensemble données qui transitent dans un routeur, à une vitesse allant jusqu'au Terabit par seconde. Ce flot est organisé en millions de paquets, chaque paquet comprenant l'adresse de la source et celle du destinataire. Il est alors important de savoir repérer rapidement le nombre de paires distinctes, le nombre de celles qui servent peu (appelées des *souris*), et de celles qui forment une part importante du trafic (les *icebergs*). Il est utile aussi de savoir donner ces informations par tranche de temps, car les évolutions brutales du nombre de paires d'adresses distinctes par unité de temps peuvent être utilisées comme alertes pour détecter des attaques. Les vers et les virus ouvrent en effet un grand nombre de connections différentes qui peuvent passer inaperçu dans un fort trafic, mais qui sont facilement repérables via cette cardinalité.

Aussi, dans le contexte des bases de données, l'optimisation d'une requête portant sur plusieurs critères (par exemple âge, sexe, lieu d'habitation) pousse à filtrer d'abord sur le critère le plus discriminant. Il s'agit alors d'estimer rapidement le nombre de valeurs distinctes que peut prendre chaque critère.

Pour synthétiser, dans tous ces cas, il faut extraire des informations utiles de très grandes données. Le modèle considéré consiste en un très grand multi-ensemble $\mathcal{M}$ de données prises dans un domaine $\mathcal{D}$ également supposé très grand. Les contraintes sont les suivantes :

- une seule passe sur les données ;
- une faible mémoire auxiliaire (au maximum de l'ordre de quelques kilo-octets) ;
- peu de perte de temps (un petit facteur par rapport au simple comptage des séparateurs entre les données) ;
- peu d'erreur sur le résultat (de l'ordre de 1 ou 2%).

Pour les routeurs, $\mathcal{M}$ est un multi-ensemble d'éléments du domaine $\mathcal{D}$ des paires d'adresses. Dans le cas du web, au lieu d'un multi-ensemble, on a une collection de multi-ensembles $\mathcal{M}$ correspondant aux pages web, dont les éléments proviennent d'un domaine $\mathcal{D}$ de mots. Dans le

cas des bases de données, le multi-ensemble $\mathcal{M}$ est l'ensemble des données et $\mathcal{D}$ est tour à tour l'ensemble des valeurs de chaque critère.

Sans aller jusqu'à traiter des tailles de l'ordre du web ou d'une trace de routeur, votre programme doit pouvoir manipuler les œuvres complètes de Shakespeare [1]. L'avantage d'un tel jeux de données de taille intermédiaire (les textes de Shakespeare totalisent un petit million de mots) est qu'il est possible de tester les programmes en calculant aussi les valeurs exactes et non simplement des valeurs approchées. Cependant, si le temps le permet et que vous en avez envie, n'hésitez pas à expérimenter avec d'autres données, comme des logs de serveurs web, des textes dans différentes langues, des morceaux de génome,...

Avec le programme que vous allez écrire, vous devrez pouvoir :

(1) estimer la taille totale du vocabulaire de Shakespeare avec 2% d'erreur en moins d'une seconde ;

(2) extraire en une seule passe une quinzaine de mots de Hamlet, tels qu'une recherche Google retrouve immédiatement qu'ils proviennent de Hamlet et non d'un texte anglais quelconque ;

(3) repérer la proportion de mots qui n'apparaissent qu'une fois ou que deux fois (les souris dans le contexte des réseaux), ainsi que le nombre d'icebergs ;

(4) comparer les pièces deux à deux et estimer leur similarité (une version simple d'un algorithme de recherche de textes presque dupliqués sur le web).

1. Hachage

Pour gérer simplement des situations très variées, on commence par faire opérer une fonction de hachage qui convertit chaque élément de $\mathcal{D}$ en une suite de bits, que l'on peut interpréter comme l'écriture binaire d'un élément de l'intervalle $[0, 1]$. Deux éléments identiques seront envoyés sur la même valeur. Ainsi, quels que soient N éléments distincts que l'on aura permutés et répliqués autant de fois que l'on veut, on obtiendra N valeurs uniformément réparties dans $[0, 1]$.

Implantation. D'un point de vue pratique, la fonction de hachage renverra des entiers sur 32 bits, ou 64 bits si les données attendues sont en quantité vraiment gigantesque.

Comme la fonction de hachage doit être appliquée à chaque donnée, elle aura un impact crucial sur l'efficacité globale de votre programme. Il faut d'une part s'assurer par des tests assez variés qu'elle réussit bien à produire des valeurs qui se comportent à peu près comme une distribution uniforme (par exemple en traçant des histogrammes, ou en utilisant un test du χ^2). D'autre part, il ne faut pas hésiter à tester des variantes d'implantation et à mesurer leurs vitesses.

Une bonne source sur ce sujet se trouve à l'url http://www.burtleburtle.net/bob/hash/.

2. Comptage approché

Pour déterminer le nombre N d'éléments *distincts* dans un flot, un algorithme très simple appelé HyperLogLog, dû à Flajolet *et alii*, parvient à n'utiliser que $O(\log\log N)$ bits de mémoire [2].

Le principe d'une version dégradée de l'algorithme est assez simple : si N joueurs tirent à pile ou face pour savoir qui fera le plus de pile consécutifs (dès qu'un joueur tire face, il a perdu) alors le vainqueur aura tiré une suite de piles dont la longueur est d'ordre $\log_2 N$. Ce nombre lui-même peut être stocké en seulement $\log_2\log_2 N$ bits.

Une première estimation grossière du nombre de joueurs est donc 2^p où p est le record du nombre de piles. Pour raffiner cette méthode, on pourrait effectuer la même expérience un certain nombre de fois et prendre la moyenne des résultats. On aboutit au même résultat, mais plus efficacement, en séparant d'abord les joueurs en $m = 2^b$ groupes selon leurs b premiers tirages, avant de les faire jouer dans chaque groupe et de prendre la moyenne des

1. Disponibles sur la page web du projet http://algo.inria.fr/salvy/INF431/Projet.
2. Cet article, ainsi que les autres références mentionnées ci-dessous, sont accessibles sur la page du projet.

résultats $2^{p_1}, \ldots, 2^{p_m}$. Les derniers réglages fins de l'algorithme proviennent d'une analyse assez sophistiquée. D'abord, il vaut mieux prendre la moyenne harmonique des résultats plutôt que la moyenne arithmétique : elle gomme mieux les records "accidentels". Ensuite, il faut corriger par un facteur constant qui ne dépend que de m :

$$\alpha_m = \left(m \int_0^\infty \log_2^m \left(\frac{2+u}{1+u} \right) du \right)^{-1}. \tag{1}$$

On voit dans un premier temps la fonction de hachage comme produisant des mots infinis sur l'alphabet $\{0,1\}$ (les suites de "pile ou face" de nos joueurs). On définit une procédure annexe ρ qui prend un tel mot et renvoie la position de son premier 1 ($\rho(1\ldots) = 1$, $\rho(01\ldots) = 2$,...). Le pseudo-code est alors très compact :

Hypothèses :
$h : \mathcal{D} \to \{0,1\}^\infty$ fonction de hachage;
$\rho : \{0,1\}^\infty \to \mathbb{N}^\star$ position du premier 1;
$m = 2^b$ avec $b \in \mathbb{N}$.
procédure HyperLogLog()
pour $i \leftarrow 0$ **à** $m-1$ **faire** $M[i] \leftarrow -\infty$
pour $v \in \mathcal{M}$
faire
- $x \leftarrow h(v)$;
- $j \leftarrow \langle x_1 x_2 \cdots x_b \rangle_2$; — entier en base 2
- $w \leftarrow x_{b+1} x_{b+2} \cdots$;
- $M[j] \leftarrow \max(M[j], \rho(w))$

renvoyer $\alpha_m m^2 / \sum_{j=0}^{m-1} 2^{-M[j]}$, avec α_m donné par (1).

Le choix de la valeur de m est un compromis entre la mémoire et la qualité de l'approximation. Tandis que l'espérance de la valeur renvoyée par l'algorithme converge vers N lorsque $N \to \infty$, l'écart-type est en $\approx \beta_m N/\sqrt{m}$, où β_m est très proche de 1. Ainsi avec $m = 2048$, l'erreur est de l'ordre du pour-cent.

Implantation. Votre programme laissera l'utilisateur choisir la valeur m dans l'intervalle $\{2^4, \ldots, 2^{16}\}$, avec un défaut à 2048. Les valeurs correspondantes de α_m seront tabulées une fois pour toutes comme

$$\alpha_{16} = 0.673, \quad \alpha_{32} = 0.697, \quad \alpha_{64} = 0.709, \quad \alpha_m = 0.7213/(1 + 1.079/m) \text{ au-delà.}$$

La consommation mémoire est donnée par m fois la valeur maximale de ρ. Avec une fonction de hachage qui renvoie des entiers sur 32 bits ou 64 bits, 5 ou 6 bits suffisent pour chaque entrée de la table, qui prend donc au plus $8m$ bits dans une bonne implantation.

Votre programme prendra comme entrée sur `stdin` un fichier de texte dont on souhaite compter les mots distincts. Il pourra être utile d'implanter aussi des filtres auxiliaires pour enlever les majuscules, ou adapter le flot d'entrée à ce que l'on souhaite mesurer. (Par exemple, pour un fichier de log de serveur web, on pourra vouloir extraire les machines d'origines uniquement, pour un log de routeur, les paires machine de départ–machine d'arrivée,...).

Pour mesurer la vitesse, si vous utilisez une variante d'Unix, vous pourrez comparer aux commandes `wc` et `wc -l`.

3. Similarités entre ensembles de données

Le tableau M construit par l'algorithme HyperLogLog donne une sorte d'empreinte très compacte du fichier, dont on peut extraire d'autres informations. Ainsi, pour tester rapidement la similarité de deux fichiers A et B, Broder *et alii* proposent de considérer la quantité

$$r(A,B) := \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|},$$

où la notation $|X|$ représente la cardinalité de l'ensemble X. La fonction $S(A)$ donne l'ensemble des *"k-shingles"* du texte, c'est-à-dire ses suites de k mots consécutifs. Avec $k = 1$, c'est exactement ce que mesure l'algorithme HyperLogLog. Pour des valeurs de k légèrement

supérieures, il suffit de hacher non plus des mots, mais des k-uplets de mots consécutifs (et qui se chevauchent d'un k-uplet au suivant).

La mesure de l'union est très facile avec les empreintes M_A et M_B des fichiers A et B : il suffit de considérer $M_{A\cup B}$ défini par

$$M_{A\cup B}[j] = \max(M_A[j], M_B[j]), \qquad j = 0, \ldots, m-1.$$

Ensuite l'intersection est donnée par

$$|A \cap B| = |A| + |B| - |A \cup B|.$$

Implantation. Votre programme prendra en entrée un dossier (ou si vous préférez une liste d'url), calculera et sauvegardera les empreintes de tous les fichiers qu'il contient (ou vers lesquels la liste pointe), puis calculera les similarités deux à deux, renvoyant de manière lisible les paires dépassant un certain seuil modifiable par l'utilisateur. (Ainsi, un seuil très proche de 1 indique une similarité très forte, des seuils moindres permettent de distinguer la langue du document,...). La valeur k sera demandée à l'utilisateur, mais il est inutile de chercher à optimiser pour des valeurs de $k > 4$.

4. Fenêtre glissante

Pour des analyses dynamiques, par exemple en surveillance de réseaux, le nombre d'éléments distincts depuis le lancement du programme ne suffit pas, et c'est l'évolution de ce nombre qu'il s'agit de mesurer. Une idée de Datar *et alii* mène à une modification très simple de l'algorithme HyperLogLog répondant au problème. On affecte à chaque élément une date (ou ici simplement un numéro) d'arrivée. Il suffit ensuite de modifier la ligne

$$M[j] \leftarrow \max(M[j], \rho(w))$$

pour prendre en compte ces numéros. Si W est la taille de la fenêtre de temps considérée (ou ici le nombre d'éléments consécutifs), cette ligne devient

si $\text{num}(w) - \text{num}(M[j]) > W$ **ou** $\rho(w) > M[j]$
alors $M[j] \leftarrow \rho(w)$.

Implantation. Il faut garder trace des numéros d'arrivée sans alourdir la consommation mémoire et sans ralentir le calcul. Un affichage graphique de l'évolution de la cardinalité est possible, mais dans un premier temps, une sortie en texte avec une indication des variations brutales suffit.

5. Échantillonnage

L'échantillonnage consiste en une extraction de k éléments représentatifs du multi-ensemble $\mathcal{M}$. Un tirage aléatoire dans un texte privilégie les éléments qui apparaissent souvent (comme les articles dans un texte en français). L'échantillonnage vise donc un tirage aléatoire parmi les éléments *distincts*. Pour cela, on part des valeurs hachées et on garde en mémoire une profondeur b initialisée à 0. On remplit un sac de taille environ $2k$ avec les valeurs hachées. Chaque fois que le sac est plein, on y fait de la place en incrémentant b et en ne conservant dans le sac que les éléments dont les b premiers bits sont 0.

Implantation. Il faut réfléchir un peu à la bonne structure de données pour ce sac. Votre programme prendra k en entrée. Le test mentionné dans l'introduction sur l'identification de Hamlet par une poignée de mots devra fonctionner.

6. Souris

Les k-souris sont les éléments de $\mathcal{M}$ qui y apparaissent k fois (on pense à k petit entre 1 et 10). L'explosion soudaine de leur nombre peut être une indication d'attaque de virus ou autres. Pour estimer leur nombre, il suffit de modifier légèrement l'échantillonnage pour garder trace aussi du nombre de fois qu'un même élément est observé.

Implantation. Une évolution simple de la précédente.

7. Icebergs

Les icebergs sont les éléments qui interviennent à une fréquence supérieure à un certain seuil. Par exemple, il s'agit de repérer dans le log des ventes d'Amazon les ouvrages qui constituent plus d'1% des transactions. Plus généralement, pour trouver les éléments apparaissant avec une fréquence $\theta \in]0, 1]$, un algorithme très simple, dû à Karp *et alii* procède en une seule passe. Un sac de taille $\lceil 1/\theta \rceil$ est rempli par les éléments, auxquels on attache un compteur initialement à 1. Lorsqu'un élément est déjà dans le sac, son compteur est incrémenté. Chaque fois que le sac est plein, on décrémente tous les compteurs et enlève du sac les éléments dont le compteur est tombé à 0. À la fin du traitement le sac contient, entre autres, tous les éléments de fréquence supérieure à θ. Une deuxième passe permettrait de filtrer très rapidement les vrais icebergs, mais n'est pas autorisée par nos contraintes.

Implantation. Attention, l'algorithme paraît très simple, mais il faut un peu de soin dans les choix de structures de données pour ne pas utiliser trop de mémoire, ou de temps.

8. Page Web

La page web http://algo.inria.fr/salvy/INF431/Projet contient ce sujet, les références bibliographiques ci-dessous, des pointeurs vers des données avec lesquelles vous pourrez expérimenter. Elle évoluera en fonction de vos questions ; il sera donc judicieux de la consulter régulièrement.

Références

[1] Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. Syntactic clustering of the web. In *6th International World Wide Web Conference* (1997).

[2] Datar, M., Gionis, A., Indyk, P., and Motwani, R. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing 31*, 6 (2002), 1794–1813.

[3] Flajolet, P. On adaptive sampling. *Computing 43*, 4 (1990), 391–400.

[4] Flajolet, P., Fusy, Éric., Gandouet, O., and Meunier, F. Hyperloglog : the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms 2007 (AofA07)* (2007), P. Jacquet, Ed., Discrete Mathematics and Theoretical Computer Science Proceedings, pp. 127–146.

[5] Karp, R. M., Shenker, S., and Papadimitriou, C. H. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems 28* (March 2003), 51–55.