Indexation Automatique - EARIA 2014

Eric Gaussier

AMA/LIG
Université Grenoble Alpes
UFR-IM^{2}AG
Informatique, Mathématiques et Mathématiques Appliquées de Grenoble
Eric.Gaussier@imag.fr

Octobre 2014

Table des matières

Objectifs du cours

- L'objectif de cours est d'introduire les structures et algorithmes fondamentaux pour l'indexation automatique (RI, classification, *clustering*) de textes
- Nous nous intéresserons en particulier :
  - A l'indexation automatique des documents textuels
  - Aux mécanismes permettant des (dis)similarités dans les grands espaces vectoriels :
    - Représentation creuse des documents
    - Fichier inverse
    - Les approches "à la" *minhash*

Table des matières

Etapes de l'indexation textuelle

1. Segmentation
   - Découper un texte en mots :

     l'information donne sens à l'esprit
     l', information, donne, sens, à, l', esprit

     soit 7 mots mais seulement 6 types ; nécessité d'un dictionnaire (au moins léger) pour certains mots ou expressions
2. Filtrage par un anti-dictionnaire des mots vides
3. Normalisation
   - De la casse, des formes fléchies, des familles lexicales
   - Lemmatisation, racinisation

$\rightarrow$ Sac-de-mots : *inform, don, sens, esprit*

Représentation vectorielle des docs (1)

- L'ensemble des types forme le vocabulaire d'une collection. Soit M la taille de ce voc., et soit N le nombre de doc. dans la coll. On considère l'espace vectoriel à M dimensions ($\mathbb{R}^M$), dans lequel chaque axe correspond à un type
- Chaque document est alors représenté par un vecteur de $\mathbb{R}^M$ dont les coordonnées sont les poids des mots dans le document :
  - Présence/absence ou nbre d'occurrences du terme dans le doc. : $w_i^d = 0/1$ ou $w_i^d = \text{tf}_i^d$
  - Nombre d'occurrences normalisé par la long. du doc. : $w_i^d = \frac{\text{tf}_i^d}{\sum_{i=1}^{M} \text{tf}_i^d}$
  - Le *tf*idf* : $w_i^d = \frac{\text{tf}_i^d}{\sum_{i=1}^{M} \text{tf}_i^d} \underbrace{\log \frac{N}{\text{df}_i}}_{\text{idf}_i}$
    où df_i représente la fréquence documentaire du mot i

Représentation vectorielle des docs (2)

La représentation vectorielle adoptée permet d'avoir directement accès aux outils mathématiques associés : distances, similarités, réduction de dimensions, ...

Différentes mesures de similarité/dissimilarité (distance)

- Cosinus $cos(d, d') = \frac{\sum_{i=1}^{M} w_i^d w_i^{d'}}{\sqrt{\sum_{i=1}^{M}(w_i^d)^2 \sum_{i=1}^{M}(w_i^d)^2}} = \frac{<d,d'>}{\sqrt{<d,d><d,d'>}}$
- Distance euclidienne
  $dist(d, d') = ||d - d'||_2 = \sqrt{\sum_{i=1}^{M}(w_i^d - w_i^{d'})^2}$

Illustration

	programmation	langage	C	java	$\cdots$
d_1	0	1	0	1	$\cdots$
d_2	1	1	0	0	$\cdots$

Exercices

- Chaque document est représenté par un tableau à M dimensions contenant les poids (coordonnées) des termes (types, mots) ; écrire un algorithme qui calcule le produit scalaire entre 2 documents ($\text{scal}(d, d') = \sum_w t_w^d t_w^{d'}$)
- Quelle est la complexité d'un algorithme qui calcule le produit scalaire entre un document et tous les autres documents de la collection ?

Une représentation creuse !

La majorité des termes de la collection n'apparaissent pas dans un document donné ; chaque document a donc la majeure partie de ses coordonnées nulles ; un gain d'espace peut être obtenu en ne représentant pas ces coordonnées nulles

Exemple de représentation creuse :

document d {

int l	(long. du doc. (types))
TabTermes int[l]	(indices des termes par ordre d
TabPoids float[l]	(poids des termes)
...	

Calcul du produit scalaire sur représentation creuse

1. Initialisation : $scal = i_1 = i_2 = 0$;
2. Tant que $(i_1 < d.l)$ et $(i_2 < d'.l)$
   - Si $(d.TabTermes[i_1] < d'.TabTermes[i_2])$ i_1++
   - Sinon si $(d.TabTermes[i_1] > d'.TabTermes[i_2])$ i_2++
   - Sinon { $scal += d.TabPoids[i_1] \times d'.TabPoids[i_2]$; i_1++; i_2++; }
3. Retourner *scal*

Illustration

	programmation	langage	C	java	$\cdots$
d_1	0	1	0	1	$\cdots$
d_2	1	1	0	0	$\cdots$

Le fichier inverse

Possibilité d'accélérer le calcul dans le cas de représentations creuses, en utilisant un *fichier inverse* qui fournit, pour chaque terme, l'ensemble des documents dans lesquels il apparaît :

$$\text{terme } i \begin{cases} \text{int I} & \text{(nbre de docs assoc.)} \\ \text{TabDocs int[I]} & \text{(indices des docs)} \\ \ldots & \end{cases}$$

On procède alors en 2 étapes :

1. Construction de l'ensemble des documents qui contiennent au moins un terme de la requête
2. Calcul de la similarité/distance entre requête et les documents de cet ensemble

Remarque Avantageux (gain de 3 à 5 ordres de grandeur) avec toute mesure (distance, similarité) qui ne fait pas intervenir les termes non présents dans un document. Produit scalaire ?, cosinus ?, distance euclidienne ?

Illustration

Collection

	programmation	langage	C	java	...
d_1	3 (1)	2 (1)	4 (1)	0 (0)	...
d_2	5 (1)	1 (1)	0 (0)	0 (0)	...
d_0	0 (0)	0 (0)	0 (0)	3 (1)	...

Fichier inverse

	d_1	d_2	d_3	...
programmation	1	1	0	...
langage	1	1	0	...
C	1	0	0	...
...	...	...	...	

Construction du fichier inverse

Dans le cadre d'une collection statique, 3 étapes principales régissent la construction du fichier inverse :

1. Extraction des paires d'identifiants *(terme, doc)*, passe compléte sur la collection
2. Tri des paires suivant les id. de terme, puis les id. de docs
3. Regroupement des paires pour établir, pour chaque terme, la liste des docs

Ces étapes ne posent aucun probléme dans le cas de petites collections où tout se fait en mémoire

Quid des grandes collections ?

Cas de mémoire insuffisante

Il faut dans ce cas stocker des informations temporaires sur disque
Trois étapes :

1. Collecte des paires TermId-DocId et écriture dans un fichier
2. Lecture par blocs de ce fichier, inversion de chaque bloc et écriture dans une série de fichier
3. Fusion des différents fichier pour créer le fichier inversé

Algorithme associé : *Blocked sort-based indexing* (BSBI)

L'algorithme BSBI (1)

1. $n \leftarrow 0$
2. while (tous les docs n'ont pas été traités)
3. do
4. $\quad n \leftarrow n + 1$
5. $\quad$ block $\leftarrow$ ParseBlock()
6. $\quad$ BSBI-Invert(block)
7. $\quad$ WriteBlockToDisk(block,f_n)
8. MergeBlocks(f_1, ..., f_n ;f_{merged})

L'algorithme BSBI (2)

L'inversion, dans BSBI, consiste en un tri sur les identifiants de termes (premiére clé) et les identifiants de documents (deuxiéme clé). Le résultat est donc un fichier inverse pour le bloc lu.

Exemple

w_1 = "brutus", w_2 = "caesar", w_3 = "julius", w_4 = "kill", w_5 = "noble"

$(w_1 : d_1)$	$(w_2 : d_4)$	$(w_2 : d_1)$	$\rightarrow$	$(w_1 : d_1)$	$(w_2 : d_1)$	$(w_2 :$
$(w_3 : d_{10})$	$(w_1 : d_3)$	$(w_4 : d_8)$	$\rightarrow$	$(w_1 : d_3)$	$(w_3 : d_{10})$	$(w_4 :$
$(w_5 : d_5)$	$(w_2 : d_2)$	$(w_1 : d_7)$	$\rightarrow$	$(w_1 : d_7)$	$(w_2 : d_2)$	$(w_5 :$

L'algorithme BSBI (3)

La fusion dans l'algorithme BSBI consiste à parcourir en parallèle les différents fichiers, en ordonnant les éléments suivant les indices de termes, puis de documents

Exemple

$(w_1 : d_1)$	$(w_2 : d_1)$	$(w_2 : d_4)$
$(w_1 : d_3)$	$(w_3 : d_{10})$	$(w_4 : d_8)$
$(w_1 : d_7)$	$(w_2 : d_2)$	$(w_5 : d_5)$

Complexité en $O(T \log T)$ où T est le nombre de paires terme-document (mais étapes de lecture et de fusion peuvent être plus longues)

Table des matières

Minhash : calcul efficace (et approximatif) de similarité entre documents

1. On considère deux ensembles A et B ; coefficient de similarité de Jaccard :

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

2. Soit h une fonction de hachage ; $h_k(S)$: k éléments de S qui minimisent h
3. $X = h_k(h_k(A) \cup h_k(B))$; $Y = X \cap h_k(A) \cap h_k(B)$
4. $\frac{|Y|}{k}$ est un estimateur sans biais de $J(A,B)$, espérance erreur $O(1/\sqrt{(k)})$

Minhash : complexité

- L'intérêt de Minhash tient au fait que l'on ne compare deux documents que sur k éléments, et que la comparaison est efficace car utilisant une liste triée de minima. La complexité de la comparaison entre deux documents vaut :

$$O(\log(k))$$

- Minhash peut donc remplacer avantageusement le calcul du produit scalaire lorsque les documents comparés sont "grands"
- Minhash est utilisé par exemple pour la détection de plagiat

Table des matières

Conclusion

- Importance du fichier inverse pour les collections creuses (ou rendues creuses)
- Combinaison avec des méthodes de *clustering* qui permettent également de réduire les temps de calcul (collections non creuses)
- Tous ces processus sont parallélisables (et parallélisés !)

Références générales et accessibles

- Recherche d'information : Applications, modèles et algorithmes - *M.-R. Amini, E. Gaussier* - Eyrolles, 2013

- Introduction to Information Retrieval - *C. Manning, P. Raghavan, H. Schütze* - Cambridge Univ. Press, 2008