Rapport

Tristan CADET, Victor LEZAUD

Nous implémentons des algorithmes de recherche pour un jeu de données textuelles. Nous avons codé notre projet sur Linux en python 3.7.4. Le code est constitué des deux fichiers `build.py` et `search.py` mais requiert des bibliothèques (nous conseillons de les télécharger avec anaconda).

Guide utilisateur

1. Mettre le dataset dans `./latimes` sans oublier de retirer les 2 .txt qui sont des readme, il reste donc 730 fichiers.
2. Exécuter build.py (`$ python build.py`) pour générer l'inverted file et autres tables associées (~2min ~900Mo sur mon laptop).
3. Exécuter search.py en mode interactif (`$ python -i search.py`) puis jouer avec la fonction `example(q, k)`

⚠ Avec la configuration par défaut les tests sont longs à exécuter. Réduire la taille de `qs`, `rep` ou `exp` les rend plus rapide mais moins complet.

⚠ Les fonctions `naive` et `fagin` renvoient des ids qu'on peut convertir en `<DOCNO>` via la table `docs_no` (cf `example`).

Présentation du code

Les détails de l'implémentation se trouvent dans les commentaires et le code lui-même qui est court. Ce qui suit est une description de chaque fichier.

`build.py`

Ce fichier charge et nettoie le dataset latimes. Il calcule son tf-idf puis enregistre la table des fichiers inversés, son index, les docs_no et le vocabulaire sur le disque.

Le parsage se fait via un parser HTML qui semblait adapté vu la structure des documents. Le nettoyage des données et le calcul du tf-idf est réalisé à l'aide de la bibliothèque sklearn. On précise les types des variables avant de les enregistrer afin d'économiser de la place et du temps, on utilisera les mêmes types dans `search.py`. Le choix de float16 pour les scores est justifié car nous n'avons pas besoin d'une grande précision.

`search.py`

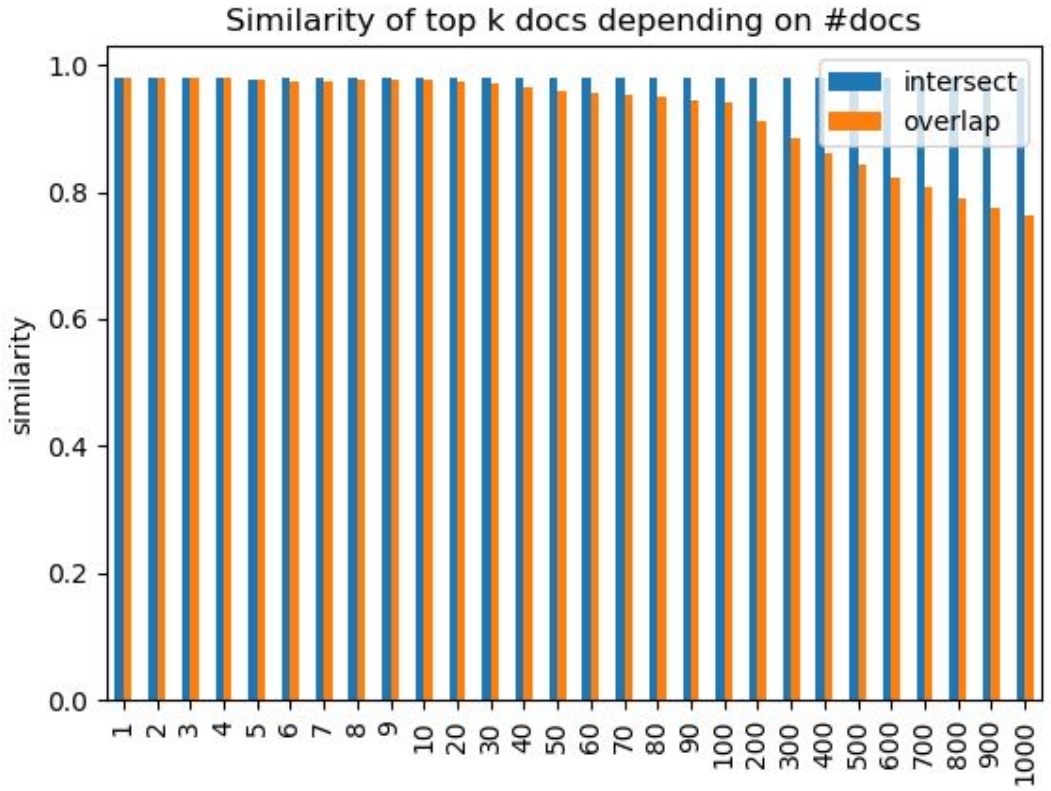
Ce fichier charge les `docs_no`, le vocabulaire et l'index du fichier inversé. Il implémente une fonction de recherche `naive` et celle de `fagin`. Il contient aussi une fonction `example` pour

montrer comment utiliser le fichier et deux tests : un sur la vitesse et un sur la similarité des résultats entre les deux algorithmes.

Analyse des résultats des tests

`testSimilarity`

Moyenne sur 50 requêtes de la similarité entre les résultats des deux algos en fonction du nombre de documents k considérés dans le top k.



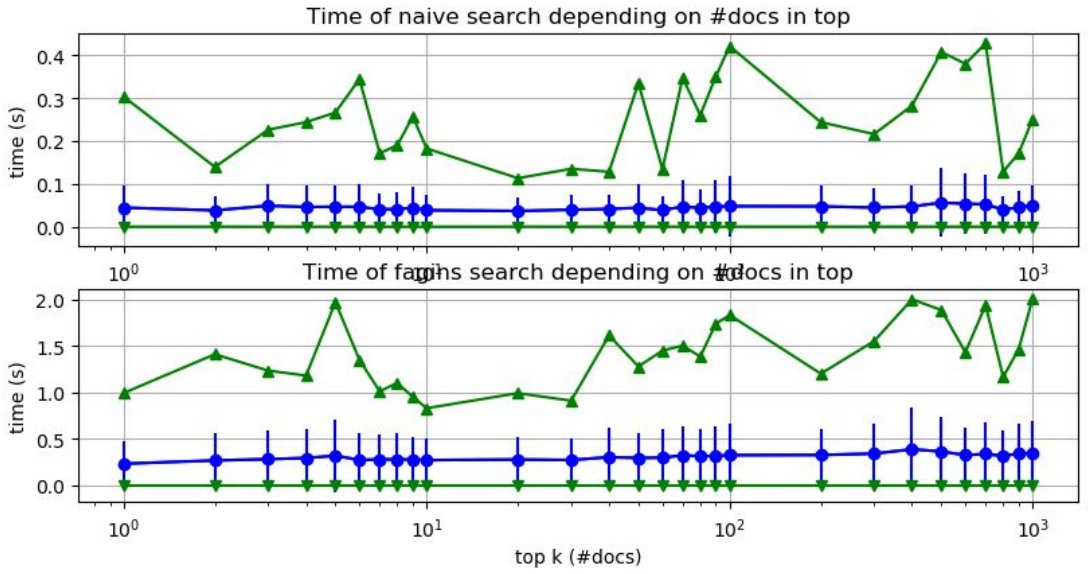
Quelques valeurs :

intersection = 0.98, overlap = 0.980 #1
intersection = 0.98, overlap = 0.975 #10
intersection = 0.98, overlap = 0.940 #100
intersection = 0.98, overlap = 0.760 #1000

Le taux d'intersection des deux top_k reste très élevé, même pour k élevé. Le taux d'overlap (même valeur au même rang) baisse lorsqu'on augmente k. On en déduit que les 2 algorithmes renvoient à peu près les mêmes documents mais que l'ordre varie plus pour les derniers documents.

`testSpeed`

Moyenne sur 50 requêtes de la vitesse des deux algos en fonction du nombre de documents k considérés dans le top k ; chaque requête est répétée 5 fois pour augmenter la précision.



En bleu le temps moyen avec les barres d'écart-type.
En vert le temps maximal et minimal.
L'échelle de l'axe y est différente en haut et en bas.

Quelques valeurs:

naive	*fagin*	*k*
t = 0.04	*t = 0.23*	*# 1*
t = 0.04	*t = 0.27*	*# 10*
t = 0.05	*t = 0.32*	*# 100*
t = 0.05	*t = 0.34*	*# 1000*

À notre surprise, l'algorithme naïf est 6 à 7 fois plus rapide que l'algorithme de fagin. Dans les deux cas le temps d'exécution augmente avec le nombre de documents. On observe que le temps minimal est proche de zéro, c'est sans doute dû à des requêtes dont tous les mots sont hors vocabulaire et donc ignorés. Potentielles explications pour la lenteur de fagin :

L'algorithme de fagin devrait être plus rapide que l'algorithme naïf. Il me semble que l'intérêt principal de l'algorithme de fagin est qu'il n'examine pas tous les documents car il sort de la boucle lorsque `len(c) == k`. Or dans 60% (k~1) à 80% (k~100) des cas l'algorithme sort de la boucle car il n'y a plus de documents à examiner, i.e. il a examiné autant de documents que dans le cas naïf. L'algorithme se met alors à examiner les documents qui n'ont pas été vu par tous les query terms alors qu'on est sûr de ne pas les

trouver dans les qt puisque tous les documents ont été vu. On fait donc des opérations supplémentaires par rapport à l'algorithme naïf ; *N.B.: après avoir corrigé ce comportement l'algorithme naïf n'est plus que 4,5 fois plus rapide*. Mais l'implémentation reste bien plus lourde puisqu'on utilise pour chaque document de `m` 2 listes pour stocker les scores et les qt à vérifier, et qu'on fait tout un tas d'opérations supplémentaires de vérifications et de mise à jour des listes.

Je note aussi que j'ai implémenté l'algorithme de fagin en série alors qu'il est proposé de réaliser les opérations en parallèle (mais comme le nombre de query terms est faible et qu'il est coûteux de mettre à jour des structures concurrentes, il n'est pas sûr qu'on.bénéficie du parallélisme).

De plus notre corpus contient (seulement) 130 000 documents, ce qui fait que chaque query term a peu de documents associés (notre inverted file contient ~25 millions de valeurs pour ~250000 mots, soit ~100 documents associés à chaque mot en moyenne) et tous les parcourir est rapide, ce qui favorise l'algorithme naïf.

Il faudrait réaliser une analyse plus poussée et optimiser l'implémentation des 2 algorithmes à fond pour pouvoir les comparer justement.