Joseph-Marie Valleix - Février 2017

Projet de recherche d'information

Installation

Requirements:

- Python (3.5)
- nltk (3.2)
- matplotlib et numpy pour les plot

Données:

Dans un dossier `data`, copier les données des collection cacm et cs276 tel que le contenu du dossier soit :

```
data/
  - cacm/
      - cacm.all
      - common_words
      - qrels.text
      - query.text
 - cs276/ (NB: cs276 search is not completly implemented, hence this folder is not mandatory)
      - pa1-data/
          - 0
          - ...
```

Lancer le programme

Dans un shell :

```
python3 ri.py <source> <type> <flags>
# source : cacm (or cs276 - but the implementatio is not complete yet)
# type   : bin or vec for binary index and vectorial index

python3 ri.py cacm bin  # starts a search shell where the user can search the collection
python3 ri.py cacm vec --evaluate --plot  # run the tests on cacm
```

NB: dans le shell de recherche, taper `#123` renvoie les informations du document dont l'id est 123.

Structure du programme

Le code est organisé de la façon suivante :

- `README.md`
- `ri.py`: point d'entrée du programme, parse les arguments de la command line
- `data/`: collections de données
- `tools/`: contient le code spécifique aux différentes collections, chaque collection expose un ensemble de fonctions qui sont utilisés pour créer les indexes et effectuer les recherches
  - `cacm.py`: code de la collection cacm + common_words
  - `cs.py`: code de la collection cs276 (inachevé)
  - `measure.py`: code contenant les fonction MAP, F1_measure et de plot pour les mesurer les performances de notre recherche
  - `search.py`: code pour la recherche, c'est ici que sont modifiables les poids tf-idf, dans les fonctions `normalize_term_in_document` et `normalize_term_in_collection`
  - `token.py`: code pour l'extraction des tokens

Choix d'implémentation

Au vu de la taille des jeux de données, toutes les formes d'indexes et de traitements sont effectués en mémoire.

La déroulement du programme est le suivante :

- les mots courants sont parsés,
- les documents sont parsés dans un `NamedTuple` et leurs tokens sont immédiatement filtrése lemmatizés et intégrés dans l'indexe inversé, qui est un dictionnaire python classique (ie une table de hachage donc l'accès en lecture et en écriture est en O(1)),
- si besoin (pour l'indexe vectoriel), on enregistre au fur et à mesure le nombre de token de chaque document,
- finalement :
  - soit le shell de recherche se lance pour l'utilisateur,
  - soit les tests de pertinences sont effectués sur la collection CACM

Choix d'implémentation et résultats sur CACM

Les différentes configuration de formules tf-idf ont été testées:

tf =

1. card, ie pas de normalisation

2. card/card_doc
3. log(1 + card/card_doc)

avec

card = cardinal de ce token dans le document

card_doc = nombre de token en tout dans le document

idf =

1. 1, ie pas de normalisation
2. log(nb_of_doc_in_collection/(1 + nb_of_doc_with_token))
3. log(nb_of_doc_in_collection/nb_of_doc_with_token)

avec

nb_of_doc_in_collection = nombre de documents dans la collection

nb_of_doc_with_token = nombre de documents contenant ce token

Commentaires

Le troisième tf (qui n'est pas très canonique !) et le troisième idf donnent les meilleurs résultats.

Pour tester les différents tfdif facilement, voir le fichier `tools/search.py`, les fonctions `normalize_term_in_document` et `normalize_term_in_collection` et décommenter les formules alternatives.

Courbe Rappel/Précision, F-mesure et MAP

Quelques exemples de courbes rappels précisions, que l'on peut obtenir avec la command `python3 ri.py cacm vec --plot`:

Pour tf-idf

rang k = 3

rang k = 5

rang k = 10

rang k = 20

Pour tf-idf

rang k = 3

rang k = 5

rang k = 10

rang k = 20

Améliorations possibles et conclusion

Poids different pour la query Pour le moment la requêtes est filtrée comme les documents le sont pour l'extraction de tokens mais les BIM 25