Information Retrieval

(Master 2 Informatique - IFI)

2023-2024

TP Moteur de recherche maison

Nicolas Sidere, La Rochelle Université - IFI

Objectifs

- Construire un index
- Représenter un ensemble des documents
- Tokenizer un texte
- Représenter un ensemble de documents avec TF-IDF
- Calcul de similarité cosinus
- Classement par pertinence

Un moteur de recherche met en relation une requête et des documents grâce à un index.

Nous allons donc voir aujourd'hui comment indexer un corpus de documents, comment associer une requête à des documents et enfin comment ranger les documents dans un ordre décroissant de pertinence.

Exercice 1 : Construire un index

Pour ne pas avoir à parcourir toute une base de données afin de trouver des documents pertinents pour chaque requête (et donc aller "lire" tous les documents à chaque requête), l'index est utilisé pour accéder directement à un ensemble de documents qui contiennent les mots de la requête.

En Python, un index peut être représenté par un dictionaire avec les mots indexés comme clé, puis un ensemble (ou une liste) de documents qui contient ce mot.

On a donc besoin de découper en mots chaque fichier du corpus et d'associer à chaque mot tous les fichiers où on le trouve.

Etape 1: Construire le vocabulaire et un index simple

Le vocabulaire c'est l'ensmeble des mots rencontrés dans le corpus. On n'indexe pas des mots qu'on a pas vu.

Imaginons que notre corpus soit composé de 3 phrases :

```
docA = "Le petit chat dort"
docB = "Le chat dort"
docC = "Jean dort"
```

Le résultat attendu serait

```
myIndex={"Le":["docA","docB"],"petit":["docA"],"Jean":["docC"],
         "chat":["docA","docB"],"dort":["docA","docB","docC",]}
```

Il nous faut un vrai corpus, nous allons utiliser le corpus disponible sur Moodle en se restreignant aux documents en français (sinon ça peut être un peu long). Pour notre cas cela va donner :

```
index = {}

for chemin in glob.glob("europarl/fr/*/*")[:5]:
  chaine = lire_fichier(chemin)
  mots = decouper_mots(chaine)
  for mot in mots:
    if mot not in index:
      index[mot] = set()#pour ne pas avoir de doublons
    index[mot].add(chemin)
```

Et ici la taille du vocabulaire, un échantillon de celui-ci et l'affichage des chemins de tous les documents contenant le mot "indique" :

```
print(len(index))#taille
print(list(index.keys())[:50])
print(index["indique"])
```

Maintenant que ça marche, on enlève les print, on travaille sur tout le corpus et on affiche la taille du vocabulaire.

À vous maintenant: affichez le nombre de textes dans lesquels on trouve "indique", "européenne" et "toto" (réfléchissez à une manière de gérer les erreurs).

On va maintenant stocker l'index pour ne pas avoir à le recalculer :

```
import json
#ligne obligatoire (retransformer le set() en liste) :
index = {mot:list(liste_fichiers)
         for mot, liste_fichiers in index.items()}

w = open("index.json", "w")
w.write(json.dumps(index))
w.close()
```

Exercice 2 : Requêter le corpus

Maintenant on voudrait requêter en associant plusieurs mots.

On charge l'index :

```
f = open("index.json", "r")
index = json.load(f)
f.close()
```

A vous de jouer maintenant, il s'agit de découper la requête en mots et de trouver tous les documents contenant un des mots de la requête

Ce qui revient à :

- créer une variable qui contient la requête, par exemple "Commission Européenne"
- découper la requête en mots
- chercher pour chaque mot les documents où il apparaît
- afficher la liste des documents en question

On va pouvoir faire des améliorations:

- afficher si aucun mot de la requête n'est connu
- ne pas afficher les doublons: si tous les mots de la requête sont présents dans un même document, on ne veut pas afficher plusieurs fois le nom de ce document
- mettre les documents dans l'ordre décroissant du nombre de mots de la requête qu'il contienne
- limiter l'affichage aux 10 premiers documents

Exercice 3 : Présenter les contextes

Avoir la liste des fichiers c'est bien joli mais on aimerait connaître les contextes. Integrez dans votre code la fontion suivante qui prend en entrée un texte et un mot et affiche tous les contextes d'apparition de ce mot dans le texte :

```
def afficher_contextes(chaine, terme, taille_contexte = 30):
  match = re.search(terme, chaine)
  contexts = []
  while match is not None:
    #Les bornes gauche et droite autour du mot :
    gauche = max(match.start()-taille_contexte-1, 0)
    droite =  match.end()+1+taille_contexte
    contexts.append(chaine[gauche:droite])
    chaine = chaine[match.end():]
    match = re.search(terme, chaine)
  for c in contexts:
    print(c)
```

Exercice 4 : Du texte au token

Pour finir on va implémenter un tokenizer plus complexe que un simple découpage de mots. Pour cela on va utiliser la fonction *word_tokenize()* de la bibliothèque *NLTK*:

```
import nltk
tokens = nltk.word_tokenize(content, language='french')
```

...à vous de le placer dans le bon endroit et comparer la qualité des résultats obtenus.

Exercice 5 : Construire l'index et l'index inversé

Ensuite, nous allons implémenter les fonctions `creer_index()` et `creer_index_inverse()`.

```
  index = creer_index()
  index_inverse = creer_index_inverse()
  print("Nombre de termes differents : ", len(index.keys()))
  print("Nombre de documents :", len(index_inverse.keys()))
```

```
Nombre de termes différents :
Nombre de documents :
```

Les deux index `index` et `index_inverse` nous permettront d'utiliser le reste des fonctions disponibles. Nous pourrions par exemple faire une requête à notre moteur de recherche en utilisant la fonction `requeter_documents(requete, index)`.

```
requete = "liberté humaine"
docs_trouves =requeter_documets(requete, index)
print("Nombre de documents trouvées :", len(docs_trouves))
requete = "sensibilisation minorités"
docs_trouves =requeter_documets(requete, index)
print("Nombre de documents trouvées :", len(docs_trouves))
```

```
Nombre de documents trouvées :
Nombre de documents trouvées :
```

Exercice 6 : Calculer les TF et IDF

Term Frequency (TF), est le taux d'apparition d'un mot dans un document. Il est toujours dans un intervalle de 0 à 1 (inclus). Cette mesure indique si ce mot est fréquent (donc important) par rapport aux autres mots du même document.

$$TF_{(m,d)} = \frac{\text{nombre d'occurrence de ce mot } m \text{ dans le document } d}{\text{nombre total de toutes les occurrences de tous les mots dans le document } d}$$

```
allDocs={ "docA":{"Le":0.25,"petit":0.25,"chat":0.25,"dort":0.25}
          "docB": {"Le":0.3333, "chat":0.3333, "dort":0.3333}
          "docC": {"Jean":0.5, "dort":0.5}}
```

Inverse Document Frequency (IDF), est une mesure globale pour un mot. Cette mesure est dans un intervalle de 0 à 1 (inclus). Elle représente l'importance d'un mot pour marquer un document dans la base. Si un mot n'est présent que dans un seul document, ce mot est important pour retrouver le document. En revanche, si un mot est présent dans tous les documents de la base, il n'est pas significatif et son score IDF sera faible.

$$IDF_{(m)} = log_{10}\frac{\text{nombre total de documents dans la base de données}}{\text{nombre de documents où ce mot } m \text{ est présent}}$$

```
mot2idf={ "Le":0.17609,
          "petit":0.47712,
          "Jean":0.47712,
          "chat":0.17609,
          "dort":0.0}
```

TF montre l'importance d'un mot dans un document tandis que IDF indique si un mot est important pour remarquer un document dans la base de données.

Le produit des scores TF et IDF d'un mot est utilisé ensuite pour la dimension de ce mot pour un document. Les documents sont ainsi représentés comme des vecteurs (avec tous les mots du vocabulaire comme dimensions).

Exercice 7 : Calcul du taux de similarité cosinus

Quand les documents sont représentés par les vecteurs, les calculs entre les documents sont devenus possibles.

Pour un moteur de recherche, le calcul lié est le taux de similarité entre une requête et un document pour pouvoir ensuite classer les documents par pertinence.

Une façon rapide pour calculer le taux de similarité est le cosinus. Ce taux de similarité se situe toujours entre 0 et 1 (inclus). 0 signifie que les deux documents n'ont aucun mot en commun tandis que 1 signifie que les deux documents sont identiques.

$$sim(r, A) = cos(r, A) = \frac{r \cdot A}{||r|| \cdot ||A||}$$

Rappel: le produit scalaire de deux vecteurs est la somme de tous les produits dans chaque dimension.

Si le vecteur de la requête r et le vecteur du document A sont:

$$r =< r_0, r_1, ..., r_V >$$

$$A =< A_0, A_1, ..., A_V >$$

V est la taille de vocabulaire. alors

$$r \cdot A = r_0 * A_0 + r_1 * A_1 + ... + r_V * A_V$$

La longueur d'un vecteur est définie par:

$$||A|| = \sqrt{A_0^2 + A_1^2 + ... + A_V^2}$$

Nous allons passer à l'implémentation de nos fonctions de similarité cosinus :

- `indexer_requete(requete)`
- `calculer_sim_cosinus(docA, docB)`
- `calculer_ponderation_cosinus(requete, index_inverse, documents_trouves)`

```
docs_trouves_pond = calculer_ponderation_cosinus(requete,
                                index_inverse, docs_trouves)
```

Exercice 8 : Classement par pertinence (ranking)

Un moteur de recherche retour toujours le résultat des documents classés par l'ordre décroissante de pertinence par rapport à la requête (taux de similarité entre la requête et un document).

Après avoir reçu le dictionnaire des similitudes cosinus, il reste à trier la pertinence de chaque document en fonction de sa similitude cosinus.

```
docs_trouves_pond_liste =[[sim, chemin] for chemin, sim
                            in docs_trouves_pond.items()]
docs_trouves_pond_liste = sorted(docs_trouves_pond_liste,
                                                reverse=True)
for sim, chemin in docs_trouves_pond_liste:
    print(sim, chemin.split("/")[-1])
```

```
    0.022675277068383765 ep-00-10-03-fr.txt
0.020478650616061803 ep-00-04-13-fr.txt
0.012191496227354267 ep-00-02-15-fr.txt
0.010717430319714566 ep-00-09-06-fr.txt
0.010655193648183531 ep-00-05-16-fr.txt
```

Exercice 9 : Structuration de l'espace de recherche

Le système précédent sera relativement efficace en termes de précision, mais aussi relativement long. Et encore, vous ne travaillez qu'avec une petite base de texte, mais en général, de nos jours, on travaille avec des bases qui contiennent des millions de documents !

Le problème vient surtout de l'utilisation de la méthode des k plus proches voisins, qui est très longue quand on fait une recherche exhaustive. Il existe de nombreuses variantes qui visent à être plus rapides (mais non-exhaustives). Mais nous n'allons pas nous attaquer à ce problème difficile ici, par manque de temps.

Ici, on va plutôt essayer d'améliorer la phase d'indexation, en structurant les index (descripteurs tf-idf). Pour accélérer le système, vous allez donc mettre en place, en plus, une phase de structuration des index.

Pour cela, vous allez implémenter une ou plusieurs des méthodes suivantes et comparer leurs efficacités avec la structuration d'origine :

- une méthode d'indexation basée sur les arbres. (KD-Tree)
- une méthode de clustering (K-Means).
- une méthode de hachage (LSH).

Pour chaque méthode choisie, vous devrez l'inclure dans votre schéma global puis la tester et en comparant leurs effets sur les résultats en terme de performance (précision), et de temps de calcul.