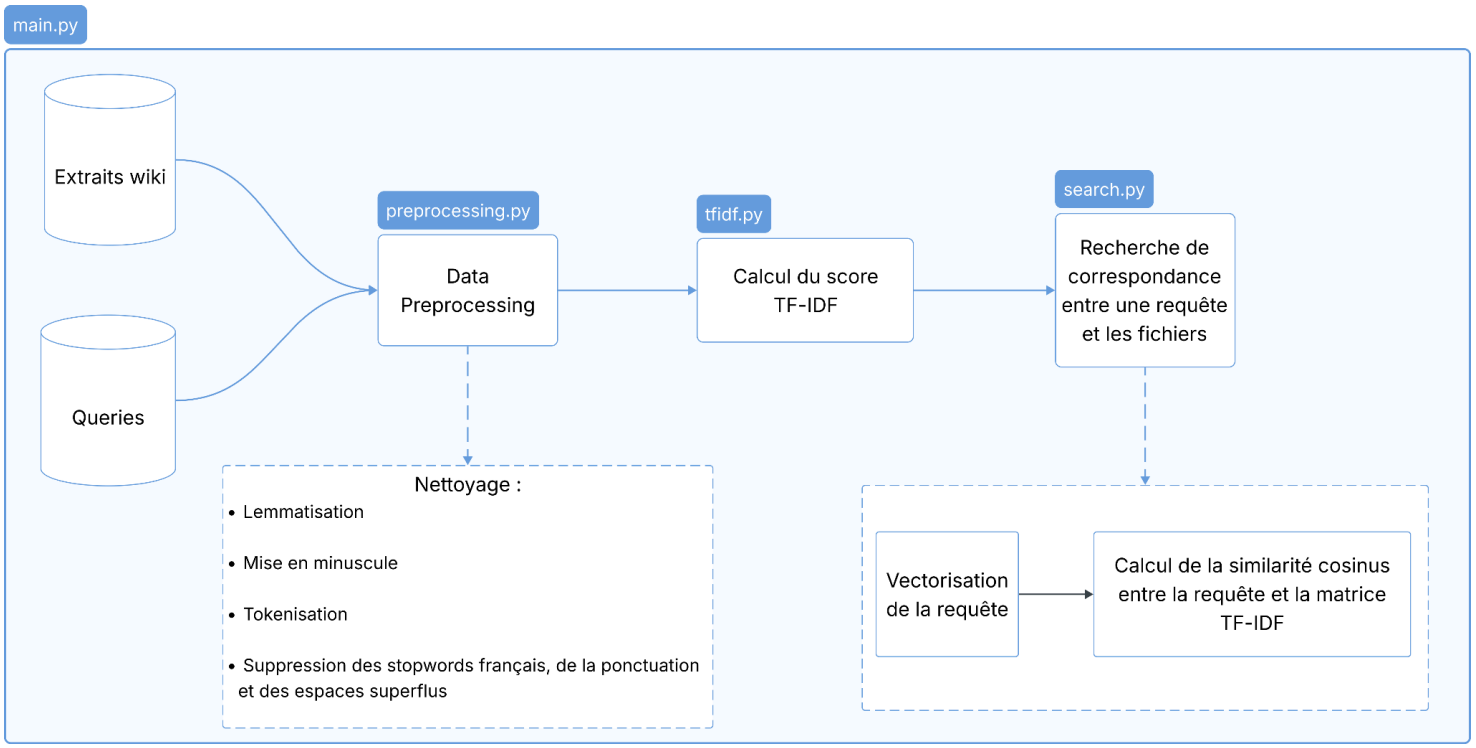
Rapport Traitement Automatique des Langues
Projet Information Retrieval

Clothilde MOLINATTI et Gautier TERNISIEN

Lien du code : https://github.com/gautierternisien/TAL_project_Ternisien_Molinatti

Méthodologie

Le schéma ci-dessous illustre l'approche adoptée pour ce projet de moteur de recherche.
Notre méthodologie s'articule autour de trois modules principaux : le prétraitement des données, la mesure du TF-IDF, ainsi que la fonction de recherche.



1. Prétraitement des données (preprocessing.py)

Le fichier preprocessing.py définit toutes les fonctions qui permettent le chargement des données (requêtes et extraits) ainsi que leur nettoyage. Ce nettoyage des données est nécessaire pour pouvoir manipuler des textes normalisés et facilement interprétables.

Nous considérons ici 4 étapes majeures pour le nettoyage (fonction *clean*) :

- La lemmatisation des mots (à l'aide de la librairie *SpaCy*)
- La mise en minuscule des textes
- La tokenisation (segmentation du document en mots)
- La suppression des stopwords français (de la librairie *SpaCy*), de la ponctuation, et des espaces superflus

Nous avons préféré utiliser SpaCy pour les stopwords français car leur bibliothèque en possède 507, alors que celle de NLTK, une autre bibliothèque populaire en NLP, en possède seulement 157.

De plus, SpaCy est optimisé pour la performance et la rapidité. En effet, la tokenisation effectuée par NLTK est considérée comme étant moins efficace.
Nous avons également testé le stemming à l'aide de Snowball Stemmer, mais les performances étaient bien inférieures à la lemmatisation.

2. Calcul du score TF-IDF (tfidf.py et tfidf_hand.py)

Pour cette partie, nous avons utilisé le modèle vectoriel TF-IDF (Term Frequency-Inverse Document Frequency).
Le fonctionnement de TF-IDF est le suivant :

- TF, ou Term Frequency, est la fréquence d'un mot par rapport au nombre total de mots dans ce document

$$TF\ (t,\ d)\ =\ \frac{Nombre\ d'occurrences\ de\ t\ dans\ d}{Nombre\ de\ termes\ dans\ d}$$

- IDF, ou Inverse Document Frequency, a pour but de réduire le score des mots les plus communs parmi tous les documents, en augmentant le score des mots les plus rares.

$$IDF\ (t,\ D)\ =\ log(\frac{Nombre\ de\ documents\ au\ total\ dans\ le\ corpus\ D}{Nombre\ de\ documents\ contenant\ le\ terme\ t})$$

$$TFIDF\ (t,\ d,\ D)\ =\ TF\ (t,\ d)\ \times\ IDF\ (t,\ D)$$

L'objectif de TF-IDF est donc de mettre en valeur les mots utilisés dans peu de documents, et dont leur importance dans ces documents est particulièrement élevée.
Après avoir essayé de l'implémenter à la main (tfidf_hand.py), nous avons réalisé que, même si les performances étaient correctes, le temps d'exécution de 15 minutes nous a convaincu à nous tourner vers le TF-IDF fourni par scikit-learn via sa fonction TfidfVectorizer.
Nous avons également choisi d'ajouter des paramètres à cette fonction TfidfVectorizer, notamment *ngram_range = (1, 2)* dans le but de créer des 1-gram et 2-gram afin d'améliorer les performances des recherches grâce à une meilleurs contextualisation de certains termes.
Enfin, nous avons choisi de passer le paramètre strip_accents de 'None' à 'unicode' afin d'augmenter légèrement nos performances en supprimant les accents.

Ainsi, une fois les données chargées et nettoyées, le vecteur TF-IDF apprend le dictionnaire de vocabulaire et renvoie la matrice document-terme associée.

3. Recherche de correspondance (search.py)

Lorsqu'une requête est soumise, elle passe d'abord par notre étape de prétraitement (appel à la fonction clean de preprocessing.py).
La requête nettoyée est transformée en un vecteur TF-IDF en utilisant le vocabulaire et les poids IDF appris lors de l'étape précédente.
Pour ensuite pouvoir classer les documents, nous calculons la similarité cosinus entre le vecteur de la requête et les vecteurs de tous les documents présents dans notre matrice TF-IDF.
Les documents sont finalement triés par score décroissant de similarité cosinus afin que les résultats les plus pertinents soient retournés en premier.

Tout cela est géré par un script principal (main.py) qui orchestre le chargement des données, la construction de l'index et l'évaluation des performances (calcul des métriques).

Protocole d'évaluation

Nous avons utilisé trois métriques principales pour évaluer notre modèle : $Accuracy$, $Recall@k$ et MRR (Mean Reciprocal Rank).
Nous avons choisi d'utiliser ces métriques afin d'avoir une vision globale et précise du bon fonctionnement de notre méthode.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Cependant, dans notre cas, cela vaut $Precision@1$. En effet, pour chaque requête, cette métrique est égale à 1 ou 0 en fonction de si le document pertinent est trouvé ou non, et globalement c'est donc la moyenne des $Accuracy$ locales.

Cette métrique nous indique le pourcentage global de requêtes pour lesquelles notre moteur de recherche a réussi à identifier correctement le document cible associé.

$$Recall@k_{local} = \frac{Nombre\ de\ documents\ pertinents\ dans\ les\ k\ premiers\ résultats}{Nombre\ total\ de\ documents\ pertinents}$$

$$Recall@k_{global} = \frac{\sum Recall@k_{local}}{Nombre\ total\ de\ prédictions}$$

Dans notre cas, étant donné que chaque requête est associée à un unique document pertinent, cette métrique s'apparente à la proportion de requêtes pour lesquelles la bonne réponse figure parmi les k premiers résultats divisé par le nombre total de prédictions. D'un point de vue local, pour chaque requête cela représente 0 ou 1 selon ce qui est retourné. Nous avons choisi de prendre $k = 5$ car nous considérons qu'un document pertinent présent dans le top 5 des prédictions est satisfaisant.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}$$

avec $rank_i$ la position du premier document intéressant et Q un ensemble de requêtes.

La valeur de cette métrique est diminuée par des mauvais résultats, et est augmentée lorsque les documents pertinents sont bien classés. L'objectif de cette métrique est de s'approcher de 1, ce qui montrerait que la majorité des requêtes sont satisfaites.

Analyse des résultats

D’abord, voici les différents résultats obtenus :

```
Accuracy: 75.00%
Recall@5: 96.00%
MRR@5: 0.8265
Temps d'exécution TF-IDF manuel: 923.62s
```

TF-IDF manuel

```
Accuracy: 82.00%
Recall@5: 97.00%
MRR@5: 0.8837
Temps d'exécution TF-IDF sklearn: 76.20s
```

TF-IDF scikit-learn

Avec notre TF-IDF manuel, nous obtenons une Accuracy de 75%, un Recall@5 de 96% et un MRR de 82,6%.
Pour le test avec le TF-IDF fourni par scikit-learn, nous obtenons une Accuracy de 81%, un Recall@5 de 97% et un MRR de 88,37%.
Ces scores démontrent une supériorité du TF-IDF de scikit-learn. De plus, nous avons remarqué que la vitesse d’exécution est beaucoup plus rapide lors de l’utilisation de la fonction de scikit-learn, comme on peut le voir au niveau des temps relevés (15 minutes pour notre code à la main contre seulement 1 minute pour celui utilisant la fonction de scikit-learn).

D’autre part, grâce au détail du rang des documents pertinents, on peut voir la position de la bonne réponse lorsque notre prédiction est fausse. Dans la plupart des cas, le document pertinent fait partie des 5 premiers (et plus précisément, du 2e prédit) mais pour la requête ‘elizabeth lère’, cela atteint le rang 1010.

```
[FAUX] Incorrect prediction for accident navett spatial 1: predicted wiki_051958.txt, expected wiki_048831.txt (rank=2)
[FAUX] Incorrect prediction for navett columbia 2: predicted wiki_045816.txt, expected wiki_048831.txt (rank=2)
[Correct] Correct prediction for mycologie 1
[Correct] Correct prediction for champignon comestible 2
[Correct] Correct prediction for conjonction coordination 1
[Correct] Correct prediction for conjonction coordination français 2
[FAUX] Incorrect prediction for elizabeth reine 1: predicted wiki_091455.txt, expected wiki_041649.txt (rank=2)
[FAUX] Incorrect prediction for elizabeth ière 2: predicted wiki_091455.txt, expected wiki_041649.txt (rank=1010)
```

Cela s’explique d’abord par le fait que dans le document, il est écrit ‘elizabeth Ire’ et non ‘lère’ donc cela influence une partie des résultats. En effet, pour tester, nous avons temporairement modifié cette requête en retirant le “è”, et l’extrait passe en rang 2 au lieu de 1010, ce qui montre l’intérêt d’avoir les requêtes les plus simples possibles, ou de bien les interpréter.

Enfin, les résultats affichés montrent un bon fonctionnement du modèle de recherche de documents pour la plupart des requêtes, ce qui est satisfaisant.

Références

https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

https://blog.devgenius.io/6-pre-processing-techniques-to-use-for-your-information-retrieval-system-f9baa9e4dd12

https://maelfabien.github.io/machinelearning/NLPfr/#6-létiquetage-morpho-syntaxique

https://www.kaggle.com/code/yassinehamdaoui1/creating-tf-idf-model-from-scratch