Rapport de projet : `Snake_search`

Emile Rolley Hugo Thomas Leo Schneider

2020/2021

Abstract

Ce document regroupe toutes les informations nécessaires à la compréhension et à l'utilisation de notre projet réalisé dans le cadre du cours *Linguistique Informatique* de l'Université de Paris.

Contents

Introduction

Ce projet a pour objectif l'implémentation d'un moteur de recherche sur un corpus de fichiers textes. Ainsi, notre problématique est la suivante : Comment obtenir à partir d'une requête en langage naturel et d'un corpus fixé, des résultats cohérents dans un temps limité ?

Avant d'expliquer nos choix d'implémentation, commençons par faire un rapide état de l'art sur l'indexation et l'extraction d'informations à partir de documents.

Etat de l'art

Différentes approches

On distingue quatre types d'approches[1] pour la recherche d'informations dans un corpus :

L'approche ensembliste

Elle se base sur la théorie des ensembles en représente les documents du corpus comme des ensembles de mots ou de phrases. Les similarités entre les éléments de ces ensembles sont obtenus grâce à des opérations ensemblistes. Le langage SQL1[2] utilise par exemple cette approche, dite aussi de logique de premier ordre. Elle est également utilisée dans le modèle booléen[3], dans lequel le corpus est considéré comme un ensemble de termes et la requête comme une expression logique sur ces termes. Un document est alors considéré comme pertinent lorsque l'expression logique de la requête est évaluée à *vrai* pour les termes de ce dernier.

L'approche probabiliste

Approche qui essaie de modéliser la notion de pertinence. Celle-ci est traitée comme une inférence statistique[4], autrement dit on déduit à partir d'un échantillon de texte la probabilité pour chaque document qu'il soit pertinent par rapport à la requête donnée. Une application de cette approche est la méthode de pondération Okapi BM25, considérée comme l'une des plus performantes dans le domaine.

L'approche algébrique

L'approche algébrique (ou vectorielle) représente les documents et les requêtes dans un même espace vectoriel. Le modèle vectoriel permet entre autres de prendre en compte la sémantique des termes de la requête et du corpus. Le contenu de chaque document est ainsi représenté par un vecteur v, dont la dimension correspond à la taille de l'ensemble des termes du corpus. Chaque composante v_i du vecteur v représente le poids (dans notre cas celui-ci est obtenu grâce au TF-IDF, mais ne l'est pas nécessairement) de la *feature* i d'un document. Un exemple simple est d'identifier v_i au nombre d'occurrences du terme i dans l'échantillon de texte. L'un des schémas de pondération les plus utilisés est le TF-IDF.

L'approche *feature-based*

Elle considère les documents comme des vecteurs dont les valeurs sont des *features*. Cette approche se rapproche grandement de l'approche vectorielle. La différence réside dans le fait que l'approche *feature-based* cherche à combiner de la façon donnant les meilleurs résultats différents *features* ou *features fonctions* afin d'obtenir un score de pertinence pour chaque document du corpus en fonction d'une requête. Autrement dit, elle permet de facilement incorporer d'autres modèles de recherche d'informations, ou des fonctionnalités afin d'affiner les résultats. Cette approche cherche la meilleure façon de combiner ces *features* afin d'obtenir un unique score de pertinence. Nous pouvons donc voir l'approche *feature-based* comme une généralisation de l'approche vectorielle.

[1] Wikipedia : Information Retrieval
[2] Cours sur le modèle relationnel
[3] Wikipedia : modèle booléen
[4] Wikipedia : inférence statistique

Différents algorithmes

Ces différentes approches utilisent toutes trois catégories d'algorithmes[5] :

- Les algorithmes de recherche (scans de séquence, recherche dans du texte indexé...)
- Les algorithmes de filtrage (suppression des stopwords, réduction des mots à leur radical...)
- Les algorithmes d'indexage (construction des matrices termes/documents) Ces catégories sont complémentaires et illustrent bien les différentes étapes du traitement des requêtes ainsi que de l'indexation de document.

  Remarque : cette description des différentes approches et algorithmes utilisés pour la recherche d'information et l'indexation de documents ne se veut pas exhaustive, mais a seulement pour but de rappeler brièvement le contexte.

Solutions retenues pour le projet

Pour `Snake_search`, nous avons retenu l'approche *feature-based*. En effet, ce choix nous a semblé le plus cohérent étant donné les deux possibilités qui s'offraient à nous :

- Soit nous implémentions nous-mêmes un ou plusieurs des algorithmes listés précédemment, auquel cas notre projet se serait beaucoup plus axé sur l'indexation des documents et notre implémentation des algorithmes de recherche d'informations ;
- ou bien, nous choisissions d'utiliser des librairies et modèles déjà existants afin de mettre en place plusieurs fonctionnalités, et ainsi avoir les résultats les plus pertinents possibles.

C'est donc la deuxième piste que nous avons finalement choisie. La possibilité de faire un bref tour d'horizon des technologies déjà existantes dans ce domaine, ainsi que de pouvoir délivrer un programme fonctionnel a motivé notre décision dans ce sens.

Langage et librairies utlisées

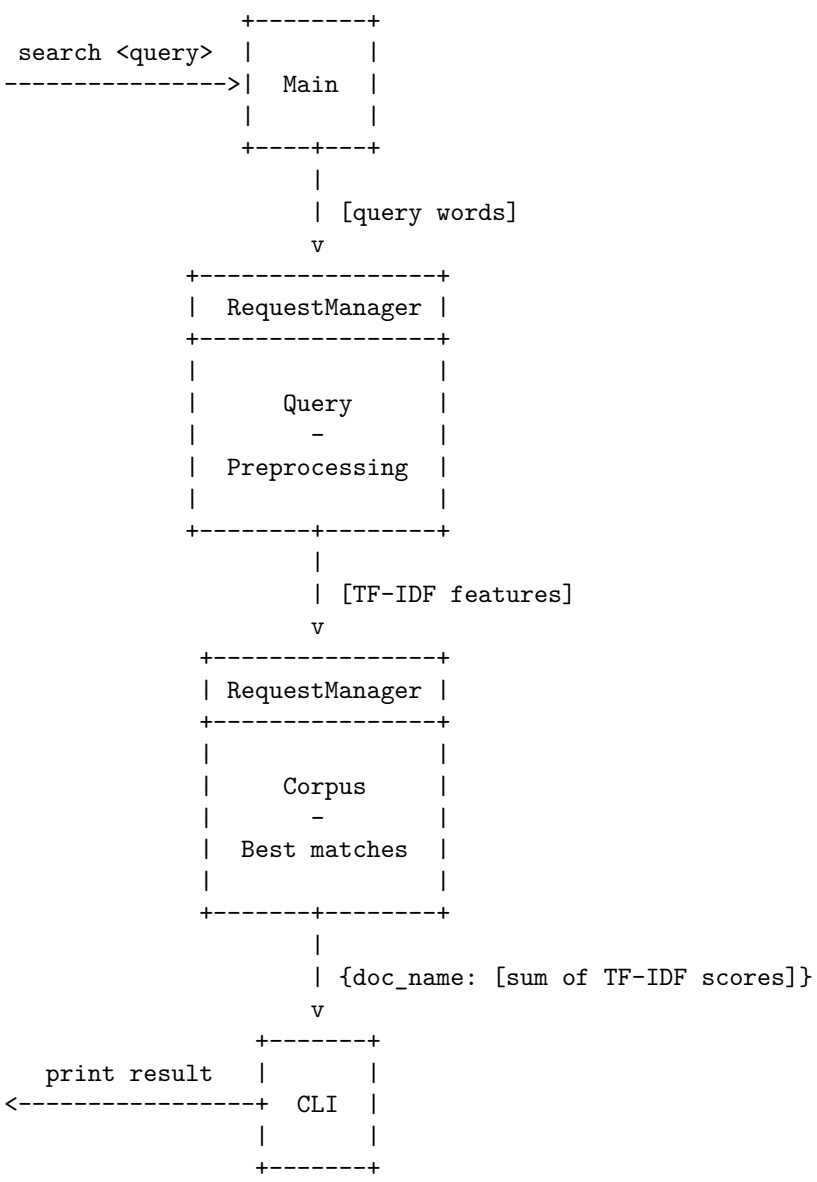
Comme énoncé précédemment, nous avons voulu dès le départ utiliser différentes technologies. Nous voulions également pouvoir implémenter une *cli* pour faciliter l'utilisation. Par la diversité et l'étendue des librairies de TAL, le choix de `Python` comme langage de programmation s'est imposé. Nous avons ainsi pu jouir de différents outils, de ce fait :

- L'indexation des documents a été réalisée à l'aide des librairies `scikit-learn` et `NLTK`.
- Pour le traitement des requêtes, nous avons utilisé un modèle `word2vec` (à l'aide de la libraire `gensim`) ainsi que la librairie `pyspellchecker`.
- L'interface utilisateur a été implémentée grâce à la librairie `Python Prompt Toolkit 3.0`.

[5] Algorithms for Information Retrieval

Explication du *flow* du traitement d'une requête

Le traitement de de cette requête suit alors le schéma suivant :



Lorsque l'utilisateur effectue une recherche, cette dernière est traitée par une instance de la classe `RequestManager`, qui va commencer par effectuer un *prétraitement* de la requête.

Le *prétraitement* est divisé en quatre étapes :

1. Récupérer tous les mots de la requête sous forme de liste.
2. Enlever de cette liste tous les *stopwords*.
3. Appliquer une stratégie pour garder uniquement les mots apparaissant dans le corpus, c'est-à-dire étant une *feature* de la matrice TF-IDF, et pour les autres trouver le mot du corpus le *plus proche*.
4. Enrichir la requêtes en ajoutant des mots sémantiquement proche.

Une fois le *prétraitement* terminé, pour chaque document du corpus, la somme des scores TF-IDF de chaque mot de la requête est effectuée. Une fois toutes les sommes effectuées, le résultat est affiché grâce aux fonctions du module `cli`.

Les différentes stratégies pour le prétraitement des requêtes

Nous avons donc implémenté différentes stratégies (`--strategy=`) :

- `levenshtein`, approche naïve qui consiste, pour chaque mot de la requête, à prendre la *feature* du corpus avec la plus petite distance de Levenshtein.
- `spellcheck`, utilise la librairie pyspellchecker avant de d'appliquer la stratégie `levenshtein` le cas échéant.
- `embeddings`, utilise la fonction `get_corrected_word` de la classe `Embedder` avant de d'appliquer la stratégie `levenshtein` le cas échéant.

Architecture

Dès le départ, il nous a semblé important d'avoir une architecture modulaire pour pouvoir facilement ajouter des fonctionnalités, d'où le découpage en plusieurs modules gérés par le module `main`.

Description des modules

Module *indexer* Regroupe les différentes classes utilisées pour indexer un ensemble de document texte en une instance de la classe `Corpus` contenant l'ensemble des informations du corpus nécessaires pour la recherche d'information, notamment la matrice des scores TF-IDF.

Module *request_manager* Contient la définition de la classe `RequestManager` qui permet de traiter les requêtes et de calculer les documents du corpus les plus pertinents à partir d'une requête sous forme de chaîne de caractère et d'une instance de la classe `Corpus`.

Module *main* Contient le point d'entrée du programme et sert d'interface entre la *cli*, le module *indexer* et le *request_manager* en traitant les commandes entrées par l'utilisateur dans le *prompt*.

Module *cli* Regroupe l'ensemble des fonctions et variables nécessaires pour l'écriture dans les sorties standards et pour la configuration du *prompt*.

Module *embedder* Sert d'interface pour l'utilisation des modèles, en proposant une fonction de correction des erreurs (qui est utilisée si la stratégie retenue est `embeddings`), et une fonction de recherche de mots sémantiquement similaires utilisée pour enrichir les requêtes.

Enfin, avant de terminer avec nos éventuelles perspectives d'amélioration, il serait bon de vous expliquer comment nous avons évalué les résultats de notre programme, pour évaluer les potentiels problèmes.

Évaluation qualificative du prétraitement des requêtes

Nous avons donc mis en place un système de *benchmarking* automatique.

Construction de la liste de référence

Il a tout d'abord fallu établir une liste de couples (requête, résultat attendu). Pour cela plusieurs corpus utilisés pour la correction automatique ont été examinés. Malheureusement, il en existe très peu pour la langue française. Nous avons donc choisi d'établir cette liste à la main à partir de la *Liste des fautes d'orthographes courantes* de Wikipédia, complétée de requêtes utilisées lors de nos tests manuels.

> **Remarque** : Les *tests* utilisés pour le *benchmarking* ne sont pas exhaustifs, et ce, parce qu'il est impossible d'établir une liste d'associations de manière objective couvrant tous les cas possibles. De plus, même si cela était possible, effectuer tous les *benchmarking* de toutes les stratégies prendrait bien trop de temps.

Ces couples sont stockés dans le fichier `YAML` *./benchmark/queries.yaml* de la façon suivante :

```yaml
- case-name:
        - {input: "input query", output: ["input", "query"]}
        ...
```

Où,

- `case-name` correspond au nom utilisé pour afficher les résultats et paralléliser l'exécution du *benchmarking*.
- `input` correspond à une requête rentrée par un utilisateur.
- `output` correspond à la liste de *features* attendues.

Utilisation

Ainsi, l'évaluation des différentes stratégies peut être lancée grâce à la commande :

`benchmark [filename] [--strategy=string] [--threshold=int]`

Où,

- `filename` correspond à chemin un vers un fichier `YAML` comme présenté précédemment (si omis, *./benchmark/queries.yaml* est utlisé).
- `--strategy=` permet de préciser la stratégie à évaluer (si omise, `embeddings` est utilisée).
- `--threshold=` permet de préciser le pourcentage minimum de similarité entre un mot de la requête et sa valeur prétraitée pour lequel cette dernière est gardée (si omis, `50` est utilisé).

Résultats

Avec l'implémentation de la commande `benchmark` nous avons donc pu tester de façon systématique et automatique toutes les stratégies de prétraitement des requêtes.

Avec le fichier *./benchmark/queries.yaml*, nous avons obtenu les résultats suivants :

Table 1: Résultats pour la stratégie `embeddings`.

Threshold (%)	*misswords* score	*stopwords* score	*sentences* score	Score total
0	48/59	2/8	3/5	0.74
25	48/59	2/8	3/5	0.74
50	51/59	4/8	3/5	0.81
75	41/59	7/8	3/5	0.71

Table 2: Résultats pour la stratégie `spellcheck`.

Threshold (%)	*misswords* score	*stopwords* score	*sentences* score	Score total
0	46/59	2/8	2/5	0.69
25	46/59	2/8	2/5	0.69
50	49/59	4/8	2/5	0.76
75	40/59	7/8	2/5	0.68

Table 3: Résultats pour la stratégie `levenshtein`.

Threshold (%)	*misswords* score	*stopwords* score	*sentences* score	Score total
0	46/59	2/8	2/5	0.69
25	46/59	2/8	2/5	0.69

Threshold (%)	*misswords* score	*stopwords* score	*sentences* score	Score total
50	49/59	4/8	2/5	0.76
75	40/59	7/8	2/5	0.68

Conclusion

Grâce à ces tests nous pouvons facilement conclure que la stratégie `embeddings` avec un seuil minimum de similarité fixé à `50%` donne le meilleur résultat. C'est pour cette raison que nous avons choisi d'utiliser ces valeurs comme valeurs par défauts.

Perspectives d'amélioration

Comme expliqué précédemment, nous avons fait le choix d'une piste qui nous a amené à nous concentrer principalement sur le pré-traitement des requêtes plutôt que sur l'indexation de documents et la recherche d'informations dans ces derniers.

Ainsi pour compléter ce projet, nous pourrions :

- Implémenter différents algorithmes pour la recherche d'informations puis mettre en place un système de *benchmarking* sur les scores de précisions et de rappels afin de comparer les différents algorithmes implémentés.
- Pour l'instant nous utilisons un relativement petit corpus de 100 documents pour 7075 termes. Il serait donc intéressant de voir comment `Snake_search` passe à l'échelle avec un corpus nettement plus étendu et modifier les algorithmes en conséquence pour avoir un temps de calcul raisonnable.
- Optimiser l'indexation du corpus pourrait également être une piste intéressante, en effet, actuellement `Snake_search` *ré-indexe* le corpus et par conséquent recalcule sa matrice TF-IDF à chaque lancement. Or, utilisant un corpus fixé il serait intéressant de l'indexer une unique fois, et de stocker ses représentations matricielles dans des fichiers.

  Remarque : Sur le corpus actuel, cela n'aurait sans doute que très peu d'impact. C'est pour cela que nous ne l'avions pas implémenté.

Utilisation

Installation

Dépendances

Les dépendances sont listées dans *./requirements.txt* et peuvent être installées avec la commande suivante :

```
pip install -r requirements.txt
```

Téléchargement du modèle

Pour pouvoir utiliser la stratégie *embeddings*, il est nécessaire de télécharger le modèle puis le placer dans le dossier `./models/`.

Lancer `Snake_search`

Le programme doit être lancé depuis la racine du projet, avec l'une des deux commandes suivantes :

```
./src/main.py
```

ou

```
python3 src/main.py
```

Une fois le programme lancé, les commandes disponible sont listées dans le *footer*. En particulier, la commande `help` permet de d'obtenir plus d'informations.

Commandes principales

Depuis la REPL, les différentes commandes disponibles sont :

- `search query [FLAGS]` : effectue une recherche avec la requête donnée.
- `quote [docname]` : après avoir fait au moins une recherche avec la commande `search`, affiche les citations de la dernière recherche dans le document dont le nom est donné en argument. Si omis, le nom du document ayant eu le meilleur score est utilisé.
- `benchmark [path] [FLAGS]` : évalue une stratégie spécifique en fonction d'un fichier de test. Par défaut le fichier *./benchmark/queries.yaml* est utilisé, cependant le chemin d'un autre fichier peut être spécifié.
- `download` : met à jour les modèles `nltk`, `punkt` et `stopwords`.
- `help` : affiche le message d'aide.
- `exit` : ferme le programme.

Flags disponibles

Les différents *flags* disponibles sont :

- `--verbose` : active le mode *verbose*.
- `--no-sim` : désactive l'enrichissement de requête.
- `--threshold=int` : spécifie un seuil minimum (en pourcentage) de similarité pour garder un mot de la requête.
- `--strategy=string` : spécifie la stratégie de pré-traitement à utiliser. (Voir les différentes stratégies disponibles).