Multi-Agent Systems

ESISAR 5A IR&C - CS534

TP - Recherche d'informations

Students: Livio Dadone, Gabriel Bragança de Oliveira
Supervisor: Annabelle Mercier

Contents

1 Introduction

La recherche d'information (RI) vise à *retrouver* et *classer* des documents pertinents en réponse à une requête utilisateur. Dans ce TP, nous avons construit un pipeline complet, allant de la préparation d'une collection de test à l'indexation, puis à l'interrogation via deux fonctions de correspondance (un modèle vectoriel TF-IDF et une mesure de proximité).

Explication

Problématique. Comment transformer une collection textuelle structurée (CACM) en une collection exploitable pour l'indexation et l'interrogation, et comment évaluer qualitativement l'intérêt de deux approches de classement : (i) similarité TF-IDF (cosinus) et (ii) pertinence basée sur la proximité des occurrences des termes de requête ?

Plan. Nous décrivons d'abord la collection CACM et son format, puis la chaîne de traitement (décodage, nettoyage, suppression des mots vides, génération HTML). Ensuite, nous présentons les calculs classiques (vocabulaire, DF), l'analyse statistique (Zipf), les représentations vectorielles et l'index inversé, avant de détailler les deux moteurs de recherche et les extensions (scraping HTML, Porter).

2 Collection de test CACM

2.1 Qu'est-ce qu'une collection de test en RI ?

Une collection de test (benchmark) est un ensemble de documents accompagné, idéalement, d'un ensemble de requêtes et de jugements de pertinence (*qrels*), permettant de comparer des méthodes d'indexation et de fonctions de correspondance de façon reproductible. [3]

Dans le TP, la collection utilisée est CACM (Communications of the ACM), une collection classique en RI, fournie sous forme d'un fichier principal structuré et de fichiers auxiliaires (requêtes, qrels, stop-list, etc.). [1]

2.2 Contenu du répertoire CACM et rôle des fichiers

À partir des fichiers fournis dans `RI/TP_Rech_Info/TPRIEtu2020/cacm/`, on observe notamment :

- `cacm.all` : fichier principal contenant tous les documents CACM au format balisé.
- `Collection` : liste des identifiants de documents (`CACM-1` à `CACM-3204`), utilisée pour itérer sur les documents.
- `common_words` : stop-list (mots vides) utilisée pour la version filtrée.
- `query.text` : ensemble de requêtes de test.
- `qrels.text` : jugements de pertinence (utile pour une évaluation quantitative : précision, rappel, MAP, nDCG, etc.).
- `cite.info` : informations de citations (utile pour d'autres expériences de RI ou d'analyse de graphe).
- `README` : description du format et des fichiers.

2.3 Structure de `cacm.all` et balises principales

Le fichier `cacm.all` est constitué de documents décrits par des balises. Les principales balises manipulées dans ce TP sont :

- `.I` : identifiant du document (index).
- `.T` : titre.
- `.A` : auteur(s).
- `.W` : résumé / contenu (abstract).
- `.B` : bibliographie.

Combien de fichiers dans le répertoire `cacm/` ?

Organisation des fichiers CACM. Dans la distribution originale du TP, les données CACM sont regroupées dans un répertoire `cacm/` contenant **7 fichiers**, comme vérifié par la commande `ls -1 cacm | wc -l`. Dans notre implémentation, nous utilisons principalement le fichier `cacm.all`, qui regroupe l'ensemble de la collection sous forme balisée.

Nombre de documents dans `cacm.all`. Le fichier `cacm.all` regroupe **3204** documents (repérés par les lignes `.I`), ce qui correspond à $N = 3204$ pour les calculs de DF/IDF et la construction des vecteurs.

2.4 Impact de la langue

La collection CACM est en anglais. Cela permet :

- l'usage direct d'une stop-list anglaise (fichier `common_words`),
- l'application de techniques classiques de normalisation (ex. stemming de Porter).

Si la collection était multilingue, il faudrait gérer (au minimum) la détection de langue, une stop-list et une normalisation spécifiques par langue, et éviter d'appliquer un stemmer unique (risque de dégradation des tokens). [3, 6]

3 Organisation du projet et chaîne de traitement

3.1 Structure du dépôt

Le travail est organisé en répertoires correspondant aux étapes du sujet, conformément aux pratiques demandées (décomposition, lisibilité, documentation par dossier) et au guide de rédaction du compte rendu. [2] La structure principale est :

- `/Python_scripts/5_Processus_en_python` : décodage + nettoyage + stop-words + génération HTML (V2).
- `/Python_scripts/6_Calcul_des_valeurs_classiques` : vocabulaire, DF.
- `/Python_scripts/7_Analyse_de_la_collection` : comptages, Zipf, vecteurs.
- `/Python_scripts/8_Construction_de_fichier_inverse` : index inversé.
- `/Python_scripts/9_Moteur_de_recherche` : moteurs TF-IDF (cosinus) et proximité.
- `/Python_scripts/10_Informations_MAIL` : scraping HTML, stemming Porter.
- `/outputs/` : sorties finales (HTML, voc/df, vecteurs, index, graphiques, résultats moteurs).

3.2 Versions de fichiers et extensions

Le TP demande de distinguer plusieurs versions des documents. Dans notre pipeline, les extensions (et formats) utilisés sont :

Version	Support	Description
(sans ext.)	`Collection/CACM-#`	Texte brut décodé depuis `cacm.all`, en conservant `.T, .A, .W, .B` (`DecodeCACMXX.pl`).
`.flt`	`Collection/CACM-#.flt`	Texte nettoyé : minuscules, accents/punctuations supprimés, espaces normalisés (`clean.pl`).
`.stp`	`Collection/CACM-#.stp`	Texte filtré : suppression des mots vides via `common_words` (`remove.pl`).
HTML (V2)	`outputs/Collection1.html`	Agrégation de tous les `.flt` dans un fichier HTML ; chaque document est un `<article class="cacm">` (`clean_v2.py`).
HTML (V2)	`outputs/Collection2.html`	Agrégation de tous les `.stp` (sans mots vides) dans un fichier HTML (`remove_v2.py`).
Porter (option)	`Collection_porter/*.stp`	Version stemmée (Porter) extraite depuis HTML (`porter_lemmatise_cacm.py`).

Table 1: Versions de fichiers et sorties produites

3.3 Enchaînement global des scripts

Explication

Chaîne de traitement

1. `DecodeCACMXX.pl` : `cacm.all` → fichiers `Collection/CACM-#` + liste `Collection/Collection`.
2. `clean.pl` : `Collection/CACM-#` → `Collection/CACM-#.flt`.
3. `remove.pl` : `Collection/CACM-#.flt` + `common_words` → `Collection/CACM-#.stp`.
4. `clean_v2.py` : agrège les `.flt` → `outputs/Collection1.html`.
5. `remove_v2.py` : agrège les `.stp` → `outputs/Collection2.html`.
6. `vocabulary.py` : calcule le vocabulaire → `outputs/vocabulaire.txt`.
7. `df.py` : calcule DF → `outputs/df.txt`.
8. `count.py` / `TermFreq.py` / `zipf_plot.py` : analyses → `counter.txt`, `termfreq.txt`, `zipf_plot.png`.
9. `vecteurBinaire.py` / `vecteurTF.py` / `vecteurTFIDF.py` : vecteurs → `vecteur*.txt`.
10. `indexInverse.py` : index inversé → `indexInverse.txt`.
11. `moteur_tfidf.py` / `moteur_proximite.py` : interrogation → `resultats_*.html`.

4 Préparation des documents : décodage, nettoyage, filtrage

4.1 Décodage : `DecodeCACMXX.pl`

Le script `DecodeCACMXX.pl`, situé dans le répertoire `5_Processus_en_python/`, lit le fichier source `cacm.all` et en extrait les documents qu'il contient afin de les rendre exploitables pour les étapes ultérieures du traitement. À partir de ce fichier unique et structuré, le script génère un fichier texte distinct pour chaque document de la collection, nommé `CACM-#` et stocké dans le répertoire `Collection`, dans lequel les différentes parties textuelles sont concaténées. En complément, un fichier global intitulé `Collection` est créé ; il répertorie l'ensemble des identifiants des documents générés et sert de référence pour le parcours systématique de la collection par les autres scripts du processus.

Pseudocode de `DecodeCACMXX.pl`

```
1. Ouvrir le fichier "cacm.all" en lecture.
2. Créer le répertoire "Collection" s'il n'existe pas.
3. Parcourir le fichier ligne par ligne :
   - Si une ligne commence par ".I", un nouveau document débute.
   - Extraire l'identifiant du document.
   - Créer un fichier "CACM-id" dans le dossier Collection.
4. Lire les lignes suivantes :
   - Après ".T", écrire le titre.
   - Après ".W", écrire le contenu principal.
   - Optionnellement traiter ".A", ".B", etc.
5. Répéter jusqu'à la fin du fichier.
6. Fermer tous les fichiers.
```

4.2 Nettoyage : `clean.pl`

Le script `clean.pl` intervient après la phase de décodage afin de produire une version normalisée des documents de la collection, enregistrée avec l'extension `.flt`. Son rôle est d'uniformiser le contenu textuel en appliquant plusieurs transformations successives : l'ensemble du texte est d'abord converti en minuscules, puis les accents sont supprimés par une translittération simple afin d'éliminer les variations orthographiques inutiles. La ponctuation ainsi que les caractères spéciaux sont ensuite retirés, ce qui permet de ne conserver que les termes

pertinents pour l'analyse. Enfin, les espaces multiples sont normalisés de façon à garantir une séparation homogène entre les mots. Cette étape de nettoyage constitue une phase essentielle pour assurer la cohérence des traitements statistiques et des opérations d'indexation réalisées par la suite.

Note

Caractères spéciaux : supprimer ou remplacer ? Dans `clean.pl`, nous **remplaçons** la ponctuation et les caractères spéciaux par un **espace**, puis nous normalisons les espaces. Ce choix évite de *fusionner* des tokens lors du nettoyage : par exemple `non-linear` devient `non linear` (deux termes), alors qu'une suppression "sans espace" produirait `nonlinear` (token artificiel). Cela améliore la stabilité de la tokenisation, des comptages (TF/DF) et des pondérations TF-IDF.

Pseudocode de `Clean.pl`

```
1. Ouvrir "Collection/Collection", qui contient la liste des fichiers à traiter.
2. Pour chaque nom de fichier F dans cette liste :
    a. Ouvrir "Collection/F" (document original).
    b. Créer "Collection/F.flt" (document nettoyé).
    c. Pour chaque ligne du fichier original :
        - remplacer le saut de ligne par un espace,
        - enlever les accents,
        - supprimer la ponctuation et les caractères spéciaux,
        - réduire les multiples espaces à un seul,
        - mettre la ligne en minuscules,
        - écrire le résultat dans "F.flt".
3. Fermer tous les fichiers.
```

4.3 Suppression des mots vides : `remove.pl`

Cette étape permet de réduire significativement la taille du vocabulaire et d'éliminer les termes peu discriminants pour la recherche d'information.

Le script `remove.pl` charge `common_words` dans une table de hachage (hash), puis filtre chaque `.flt` pour produire `.stp`.

Pseudocode de `Remove.pl`

```
1. Charger le fichier "commonwords" et stocker tous les mots vides dans un ensemble (set).
2. Ouvrir "Collection/Collection" pour obtenir la liste des fichiers à traiter.
3. Pour chaque nom de fichier F dans cette liste :
    a. Ouvrir "Collection/F.flt" (version nettoyée du document).
    b. Créer "Collection/F.stp" (version sans mots vides).
    c. Pour chaque ligne du fichier .flt :
        - découper la ligne en mots,
        - pour chaque mot :
        si le mot n'est pas dans l'ensemble des mots vides,
        l'écrire dans le fichier .stp,
        - reconstruire la ligne filtrée avec des espaces.
4. Fermer tous les fichiers.
```

4.4 Génération HTML V2 : `clean_v2.py` et `remove_v2.py`

L'énoncé demande une V2 permettant de regrouper tous les documents dans un seul fichier HTML, en marquant chaque document comme appartenant à CACM via une **classe** HTML. Nous avons donc produit :

- `outputs/Collection1.html` à partir des `.flt` (`clean_v2.py`),
- `outputs/Collection2.html` à partir des `.stp` (`remove_v2.py`).

Dans les deux cas, chaque document est encapsulé ainsi :

```
<article class="cacm" id="CACM-XXXX">
  [CACM-XXXX] ...texte...
</article>
```

5 Valeurs classiques : vocabulaire et DF

5.1 Vocabulaire : `vocabulary.py`

Le script `vocabulary.py` construit `outputs/vocabulaire.txt` en parcourant les documents `.flt` et en collectant les mots distincts.

Pseudocode

```
1. Initialiser une liste vide dans le code pour stocker le vocabulaire.
2. Ouvrir le dossier "Collection/" en lecture.
3. Pour chaque fichier dans ce dossier :
    a. Ouvrir ce fichier de document en lecture.
    b. Pour chaque ligne lue dans le fichier du document :
        - découper la ligne en mots (split sur les espaces),
        - pour chaque mot :
        si le mot n'est pas encore présent dans la liste (check avec python functions),
        l'ajouter à cette structure.
    c. Fermer le fichier du document.
4. Après avoir parcouru tous les documents, ouvrir un nouveau fichier "vocabulaire.txt" en
écriture.
5. Pour chaque mot dans la structure de vocabulaire : écrire ce mot dans le fichier
"vocabulaire.txt", un mot par ligne.
6. Fermer le fichier "vocabulaire.txt"
```

La taille du vocabulaire constitue un indicateur important de la complexité lexicale de la collection.

Résultat obtenu. Le fichier `outputs/vocabulaire.txt` contient **11528** termes distincts (version nettoyée `.flt`).

5.2 DF (Document Frequency) : `df.py`

Le script `df.py` calcule la fréquence des termes dans les documents, c'est-à-dire le nombre de documents contenant chaque terme. Les calculs sont réalisés à partir des fichiers `.stp`, après suppression des mots vides. Le terme `cacm` est également retiré du dictionnaire de fréquences afin d'éviter un biais dans les résultats.

Définition du DF et rôle de l'IDF

La fréquence documentaire (DF) correspond au nombre de documents contenant un terme. L'IDF (Inverse Document Frequency) permet de réduire le poids des mots très fréquents et de valoriser les termes rares, plus discriminants. L'utilisation du logarithme permet de lisser les variations.

Pseudocode

```
1. Créer un dictionnaire vide qui contiendra :
mot : nombre de documents où il apparaît.
2. Parcourir chaque fichier de la collection :
    a. Lire tout le contenu du fichier.
    b. Extraire les mots et construire une liste contenant chaque mot une seule fois pour ce
    document (même si un mot apparaît plusieurs fois).
    c. Pour chaque mot de cette liste unique :
        - si le mot n'est pas dans le dictionnaire df :
        df[mot] = 1 (première fois qu'il apparaît)
        sinon :
        df[mot] += 1 (il apparaît dans un document de plus)
```

```
3. Après avoir parcouru tous les documents,
    ouvrir un fichier "df.txt" en écriture.
4. Pour chaque mot dans le dictionnaire df,
    écrire une ligne de la forme :
    mot df[mot]
```

Extrait de `df.txt`.

```
algorithm 1194
computer 597
system 506
paper 435
method 390
time 383
```

Explication

Un terme apparaît-il dans tous les documents ? Que faire si c'est le cas ? Sur notre version `.stp` (après suppression des mots vides et retrait du token `cacm`), aucun terme n'a une fréquence documentaire égale à $N = 3204$. Le maximum observé dans `outputs/df.txt` est **1194** pour le terme `algorithm`, donc $\max_t df(t) \ll N$.
Si un terme vérifiait $df(t) = N$, alors son poids IDF deviendrait nul :

$$idf(t) = \log\left(\frac{N}{df(t)}\right) = \log(1) = 0,$$

ce qui signifie qu'il n'est **pas discriminant** pour le classement TF-IDF. Dans ce cas, on peut le laisser (il aura un poids nul) ou le filtrer comme une stopword "statistique" afin de réduire la taille de l'index.

Taille du vocabulaire sans mots vides Les fichiers `df.txt` et `counter.txt` contiennent **11172** termes (version `.stp`, après suppression du token `cacm` dans les scripts), ce qui montre une réduction par rapport aux **11528** termes en `.flt`. Cette différence correspond aux mots vides retirés (et à l'effet du nettoyage/filtrage sur la présence de certains tokens).

6 Analyse de la collection : comptages, Zipf, TermFreq

6.1 Comptage global : `count.py`

Le script `count.py` calcule le nombre total d'occurrences de chaque mot dans toute la collection (mode `.stp` par défaut), puis écrit `outputs/counter.txt` sous la forme `rang compte mot`, trié par fréquence décroissante.

Pseudocode

```
1. Initialiser un dictionnaire counter vide : mot -> nombre total d'occurrences.
2. Parcourir chaque document de la collection (fichiers .stp) :
    a. Lire le contenu du document et le découper en mots (split sur les blancs).
    b. Pour chaque mot (on conserve toutes les occurrences) :
        - si mot n'existe pas dans counter : counter[mot] = 1
        - sinon : counter[mot] += 1
3. Trier les termes par fréquence décroissante.
4. Écrire dans counter.txt des lignes : rang  fréquence  mot.
```

Mots les plus fréquents (extrait).

```
1 1544 algorithm
2 1103 system
3 992 computer
4 728 data
5 688 program
6 673 time
7 662 method
8 617 language
9 528 programming
```

```
10 519 systems
```

6.2 Fréquence moyenne conditionnelle : `TermFreq.py`

Le script `TermFreq.py` (générique) calcule, pour un ensemble de termes donnés, la moyenne d'apparition *conditionnelle* :

$$\text{moyenne}(t) = \frac{\sum_d \text{tf}(t,d)}{|\{d : \text{tf}(t,d) > 0\}|}.$$

Le fichier `outputs/termfreq.txt` contient les mesures effectuées.

Pseudocode

```
1. Définir la liste des termes à tester.
2. Pour chaque terme T :
    a. mettre total = 0
    b. mettre docs = 0
3. Parcourir les fichiers du répertoire Collection/ :
        - lire le contenu
        - découper en mots
        - compter nb = nombre d'occurrences de T dans ce fichier
4. si nb > 0 :
        - total += nb
        - docs += 1
5. Après avoir parcouru tous les fichiers :
        - si docs > 0 : moyenne = total / docs
        - sinon moyenne = 0
6. Ouvrir termfreq.txt en mode ajout et écrire :
        - T total docs moyenne
```

Extrait de `termfreq.txt`.

```
algorithm 1544 1194 1.2931
system 1103 506 2.1798
computer 992 597 1.6616
```

6.3 Loi de Zipf : `zipf_plot.py`

La loi de Zipf relie la fréquence $f(r)$ d'un mot à son rang r via une loi puissance : $f(r) \propto 1/r^s$, avec $s \approx 1$ dans de nombreux corpus. [3, 5]

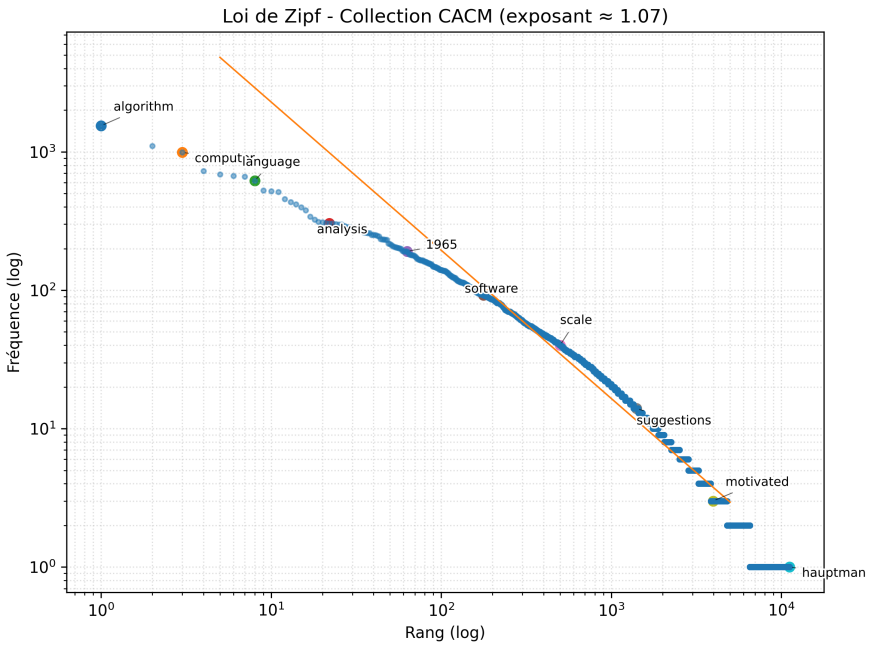


Figure 1: Vérification de la loi de Zipf sur la collection CACM (version `.stp`) : exposant estimé ≈ 1.07 (figure produite par `zipf_plot.py`).

L'observation du graphique en échelle log-log montre un alignement approximatif des points le long d'une droite, ce qui confirme le comportement zipfien de la collection. Ce résultat est particulièrement net après la suppression des mots vides, qui sont peu discriminants pour l'analyse statistique du corpus.

7 Représentations vectorielles des documents

Les représentations vectorielles sont des prérequis majeurs du modèle vectoriel en RI. Dans ce TP, nous générons successivement : binaire, TF, puis TF-IDF. [3, 4]

7.1 Vecteur binaire : `vecteurBinaire.py`

Le script `vecteurBinaire.py` produit `outputs/vecteurBinaire.txt`, une ligne par document :

$$\text{doc} \mapsto \{(id(t), 1) \text{ si } t \in d\}.$$

Chaque ligne est une suite `id:1` triée par identifiant de terme.

Extrait.

```
138:1 893:1 1882:1 3017:1 5527:1 ...
```

Pseudocode de `vecteurBinaire.py`

```
1. Lire le fichier vocabulaire.txt et construire :
    - une liste des mots du vocabulaire,
    - un dictionnaire indexVocab qui associe à chaque mot un identifiant de terme (1, 2, 3, . . . ).

2. Ouvrir Collection/Collection pour parcourir la liste des documents.

3. Ouvrir un fichier de sortie vecteurBinaire.txt en écriture.

4. Pour chaque nom de document F dans Collection/Collection :
    - ouvrir Collection/F.stp,
    -lire tout le texte du document et le découper en mots,
    - construire un ensemble des mots présents dans ce document,
```

- pour chaque mot de cet ensemble, si le mot est dans indexVocab, récupérer son identifiant de terme et l'ajoute

  - trier ces identifiants,
  - écrire dans vecteurBinaire.txt une ligne contenant :
      id1:1 id2:1 id3:1 ...
      (un couple idTerme:1 pour chaque terme présent dans le document).

5. Fermer tous les fichiers.

7.2 Vecteur TF : `vecteurTF.py`

Le script `vecteurTF.py` est similaire au binaire, mais remplace la valeur 1 par la fréquence du terme dans le document :

$$\text{doc} \mapsto \{(id(t), \text{tf}(t,d))\}.$$

Sortie : `outputs/vecteurTF.txt`.

7.3 Vecteur TF-IDF : `vecteurTFIDF.py`

Le script `vecteurTFIDF.py` combine TF et IDF :

$$\text{idf}(t) = \log\left(\frac{N}{\text{df}(t)}\right), \qquad w(t,d) = \text{tf}(t,d) \cdot \text{idf}(t).$$

Il lit `outputs/df.txt` et écrit `outputs/vecteurTFIDF.txt` avec une ligne par document au format `id:tfidf`.

Extraits des fichiers `vecteurTF.txt` et `vecteurTFIDF.txt`

Chaque ligne correspond à un document (ordre de `Collection/Collection`) et contient des couples `idTerme:poids`.

```
# vecteurTF.txt (doc 1 puis doc 20)
138:1 893:1 1882:1 3017:1 5527:1 6002:1 7763:1 8086:1 8869:1 9220:1
138:1 660:1 661:1 1073:1 1882:1 2694:3 2697:1 3381:1 3432:1 3861:1 5203:1 5348:1 ...

# vecteurTFIDF.txt (doc 1 puis doc 20)
138:4.408594 893:4.222008 1882:0.000000 3017:2.462684 5527:5.874931 6002:2.358423
    7763:5.587249 8086:5.076423 8869:3.897768 9220:6.462717
138:4.408594 660:8.072155 661:6.685861 1073:3.466985 1882:0.000000 2694:15.082899
    2697:6.280396 3381:2.444534 3432:7.379008 3861:4.029104 5203:5.769570 5348:6.973543
     ...
```

8 Construction de l'index inversé

8.1 Index inversé : `indexInverse.py`

L'index inversé associe à chaque terme la liste des documents dans lesquels il apparaît. C'est une structure centrale en RI pour accélérer l'appariement requête/documents. [3]

Le script `indexInverse.py` construit l'index inversé à partir des documents de la collection en s'appuyant sur le vocabulaire préalablement extrait. Il associe d'abord un identifiant numérique à chaque terme du vocabulaire, puis parcourt l'ensemble des fichiers `.stp` afin de collecter des paires `(idTerme, idDoc)` correspondant aux occurrences des termes dans les documents. Ces paires sont ensuite triées et regroupées de manière à constituer, pour chaque terme, la liste ordonnée des documents dans lesquels il apparaît. Le résultat est écrit dans le fichier `outputs/indexInverse.txt` sous la forme `idTerme mot doc_1 doc_2 . . .`, où l'identifiant des documents correspond à leur position dans le fichier `Collection/Collection`.

Explication

Construction en 3 phases (comme dans l'énoncé).

1. **Paires :** parcourir tous les documents et générer des paires $(idTerme, idDoc)$ pour chaque occurrence (ou une seule fois par document si l'on veut éviter les doublons).
2. **Tri :** trier lexicographiquement la liste des paires (d'abord par $idTerme$, puis par $idDoc$).
3. **Groupement :** parcourir la liste triée et regrouper toutes les paires ayant le même $idTerme$ afin de construire la liste de postings (documents) associée à chaque terme.

Extrait de `indexInverse.txt`.

```
1 0 298 533 536 727 1031 ...
2 000 441 695 825 851 ...
3 040 2035
```

9 Moteurs de recherche

Le moteur de recherche TF-IDF du script moteur_tfidf.py utilise un modèle vectoriel pour les documents et les requêtes. Il combine la fréquence des termes (TF) et la fréquence inverse des documents (IDF).

Au démarrage, il charge les fichiers des étapes précédentes.

- outputs/vocabulaire.txt: ce fichier associe chaque mot à un numéro.
- outputs/df.txt: ce fichier contient les fréquences de documents nécessaires pour l'IDF.
- outputs/vecteurTF.txt: ceci contient les vecteurs de fréquence des termes pour tous les documents.

Grâce à ces informations, chaque document devient un vecteur TF-IDF pondéré. Les pondérations sont calculées à l'aide de la formule IDF classique (et s'inscrivent dans les analyses classiques des distributions TF/IDF). [4, 8]

$$idf(t) = \log(N/df(t)),$$

où N représente le nombre total de documents. Nous calculons également la norme euclidienne de chaque vecteur afin de permettre une comparaison équitable.

Lorsque vous saisissez une requête, le processus est similaire:

Un vecteur TF-IDF est créé uniquement à partir des mots de votre requête présents dans notre vocabulaire, puis normalisé.

Nous évaluons ensuite la pertinence de chaque document par rapport à votre requête. Pour ce faire, nous calculons la similarité cosinus entre les vecteurs de la requête et du document. Cette similarité nous indique leur degré de proximité, indépendamment de leur longueur. Les documents sont ensuite classés du plus similaire au moins similaire, et seuls les meilleurs résultats sont conservés.

Enfin, les résultats de la recherche s'affichent dans votre terminal et sont enregistrés dans un fichier HTML cliquable, outputs/resultats_tfidf.html, ce qui facilite la consultation des documents trouvés par le moteur.

9.1 Moteur Proximité floue : `moteur_proximite.py`

Ce moteur calcule un score de proximité sur la base des positions des occurrences des termes de la requête dans le document, selon un principe de pertinence basée sur la proximité. [7]. Le script implémente une **zone d'influence triangulaire** de largeur k (par défaut $k = 5$) :

$$f(\Delta) = \max\left(\frac{k - |\Delta|}{k}, 0\right).$$

Pour chaque occurrence d'un terme de la requête à la position pos, on met à jour une proximité locale $p(x)$ pour les positions x dans la fenêtre (max des influences). Le score final est :

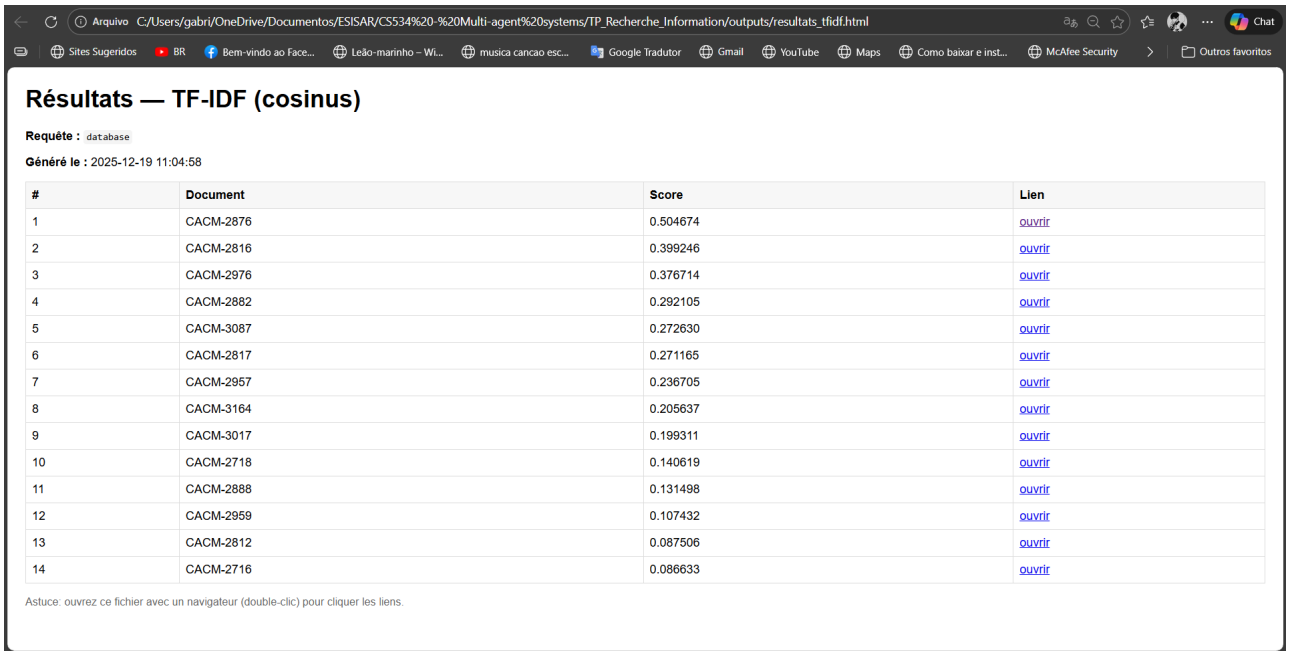
$$\text{score}(d, q) = \sum_{x=1}^{L} p(x),$$

où L est la longueur (en tokens) du document.

Les résultats sont exportés dans `outputs/resultats_proximite.html`.

9.2 Exemple de résultats (requête `database`)

Pour illustrer le fonctionnement, nous reprenons l'exemple effectivement présent dans les sorties HTML générées (`outputs/resultats_*.html`) pour la requête `database`.



Résultats — TF-IDF (cosinus)

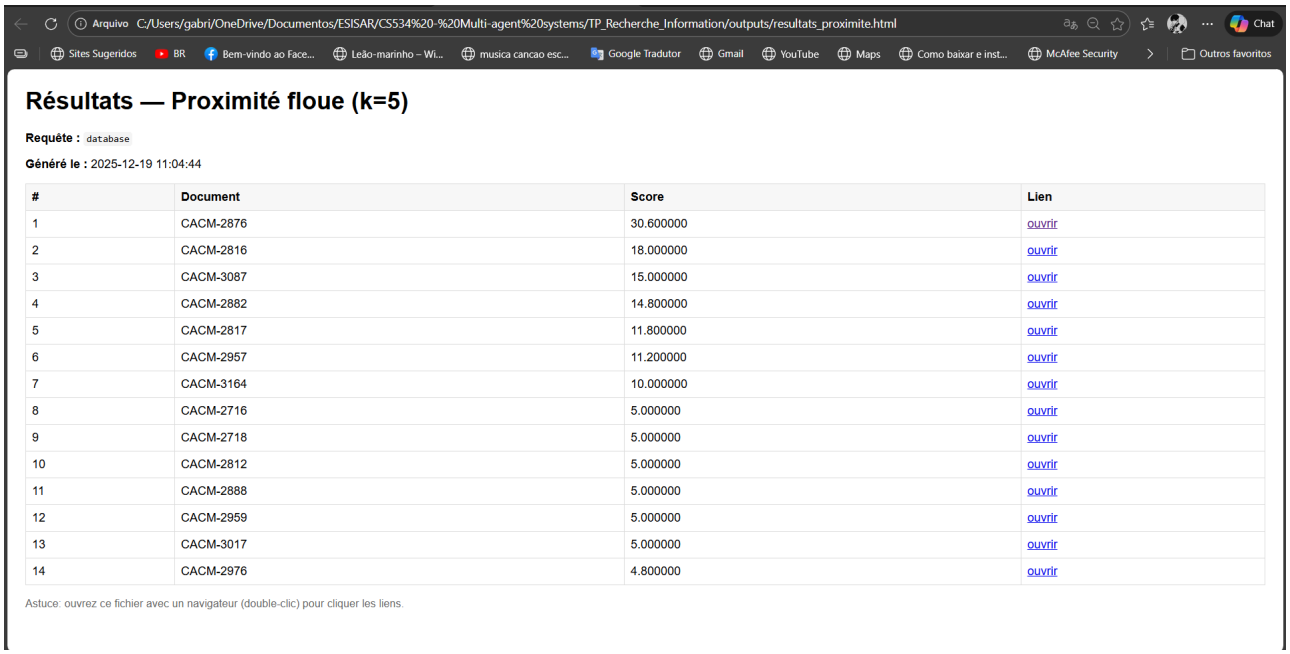
Requête : database

Généré le : 2025-12-19 11:04:58

#	Document	Score	Lien
1	CACM-2876	0.504674	ouvrir
2	CACM-2816	0.399246	ouvrir
3	CACM-2976	0.376714	ouvrir
4	CACM-2882	0.292105	ouvrir
5	CACM-3087	0.272630	ouvrir
6	CACM-2817	0.271165	ouvrir
7	CACM-2957	0.236705	ouvrir
8	CACM-3164	0.205637	ouvrir
9	CACM-3017	0.199311	ouvrir
10	CACM-2718	0.140619	ouvrir
11	CACM-2888	0.131498	ouvrir
12	CACM-2959	0.107432	ouvrir
13	CACM-2812	0.087506	ouvrir
14	CACM-2716	0.086633	ouvrir

Astuce: ouvrez ce fichier avec un navigateur (double-clic) pour cliquer les liens.

TF-IDF cosinus (`resultats_tfidf.html`)



Résultats — Proximité floue (k=5)

Requête : database

Généré le : 2025-12-19 11:04:44

#	Document	Score	Lien
1	CACM-2876	30.600000	ouvrir
2	CACM-2816	18.000000	ouvrir
3	CACM-3087	15.000000	ouvrir
4	CACM-2882	14.800000	ouvrir
5	CACM-2817	11.800000	ouvrir
6	CACM-2957	11.200000	ouvrir
7	CACM-3164	10.000000	ouvrir
8	CACM-2716	5.000000	ouvrir
9	CACM-2718	5.000000	ouvrir
10	CACM-2812	5.000000	ouvrir
11	CACM-2888	5.000000	ouvrir
12	CACM-2959	5.000000	ouvrir
13	CACM-3017	5.000000	ouvrir
14	CACM-2976	4.800000	ouvrir

Astuce: ouvrez ce fichier avec un navigateur (double-clic) pour cliquer les liens.

Proximité floue (`resultats_proximite.html`)

Figure 2: Captures d'écran des résultats pour la requête `database` exportés en HTML (Top documents et scores).

10 Extension : scraping HTML et stemming de Porter

L'extension demandée consiste à exploiter la version HTML V2 de la collection (ici `outputs/Collection2.html`) pour reconstruire une collection de documents, puis appliquer une normalisation supplémentaire via le stemming de Porter.

10.1 Scraping HTML : extraction des documents depuis `Collection2.html`

Nous utilisons un parseur HTML (BeautifulSoup) pour lire `Collection2.html`. [9] Chaque document est encodé comme un bloc : `<article class="cacm" id="CACM-#"> ... </article>`. L'objectif du script `scrape_cacm_html.py` est donc : (i) parcourir tous les `<article>` de classe `cacm`, (ii) récupérer le texte brut (via `get_text`), (iii) écrire un fichier texte par document dans un nouveau répertoire, et (iv) générer une nouvelle liste `Collection` des identifiants extraits, afin de pouvoir réutiliser le pipeline (vecteurs, index, moteurs) sur cette collection.

10.2 Stemming de Porter : réduction morphologique des tokens

Le script `porter_lemmatise_cacm.py` applique ensuite un stemming de Porter (NLTK) sur le texte : [6, 10]

- tokenisation simple : minuscules et remplacement des caractères non alphanumériques par des espaces,
- application du stemmer : chaque token est remplacé par sa racine (*e.g.* `computing` → `comput`),
- écriture d'une nouvelle collection stemmée (un fichier par document) et d'une liste `Collection`.

Le stemming tend à **réduire la taille du vocabulaire** et peut améliorer le **rappel** (plus de formes rapprochées), au prix potentiel d'une baisse de **précision** si des racines différentes sont fusionnées.

11 Conclusion et sujet d'ouverture

11.1 Conclusion

Dans ce TP, nous avons construit un pipeline complet de recherche d'information sur la collection CACM, depuis un fichier balisé unique (`cacm.all`) jusqu'à l'interrogation via deux moteurs. Après décodage, nettoyage et suppression des mots vides, nous avons produit des versions exploitables (texte et HTML), calculé les valeurs classiques (vocabulaire, DF), étudié la distribution statistique (loi de Zipf), puis construit des représentations vectorielles (binaire, TF, TF-IDF) et un index inversé.

Les deux fonctions de correspondance implémentées offrent des comportements complémentaires : (i) la similarité cosinus TF-IDF favorise les documents contenant des termes discriminants (IDF élevé) et fournit un classement robuste et standard, (ii) la mesure de proximité floue met en avant les documents où les termes de la requête apparaissent *près les uns des autres*, ce qui peut mieux capturer certaines requêtes « phrasées ». Enfin, l'extension (scraping HTML + stemming de Porter) montre comment réutiliser la chaîne sur une collection reconstruite depuis HTML et comment renforcer la normalisation lexicale.

11.2 Sujet d'ouverture

Une suite naturelle consiste à passer d'une évaluation qualitative (inspection des Top-k résultats) à une évaluation **quantitative** reproductible en exploitant `query.text` et `qrels.text` : précision@k, rappel, MAP ou nDCG. Cela permettrait de comparer objectivement TF-IDF et proximité, mais aussi d'explorer d'autres fonctions de correspondance (par exemple BM25). [11] On pourrait également comparer avec des implémentations éprouvées (Lucene) ou des frameworks d'expérimentation (PyTerrier). [12, 13] Enfin, sur des collections plus volumineuses, on pourrait étudier la compression de l'index inversé (postings) et l'impact de normalisations supplémentaires (Porter, lemmatisation) sur les performances et la qualité.

References

[1] A. Mercier. *Traitement de données – méthodes de RI : TP Recherche d'informations (2025–2026), énoncé et indications.* Support de cours CS534, Grenoble INP – ESISAR, 2025.

[2] A. Mercier. *Guide – Compte rendu TP Recherche d'informations (TPRI 2025).* Document fourni avec le sujet, 2025.

[3] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[4] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[5] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.

[6] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[7] A. Mercier and M. Beigbeder. Calcul de pertinence basée sur la proximité pour la recherche d'information. *Document numérique*, 9(1):43–60, 2006.

[8] M. Beigbeder and A. Mercier. Étude des distributions de tf et de idf sur une collection de 5 millions de pages HTML. Article fourni dans `RI/52618529 (2).pdf`, 2006.

[9] L. Richardson. *Beautiful Soup Documentation*. https://beautiful-soup-4.readthedocs.io/en/latest/

[10] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python / NLTK documentation*. https://www.nltk.org/

[11] S. Robertson. Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of Documentation*, 60(5):503–520, 2004. (BM25 et pondérations associées, références de contexte).

[12] Apache Software Foundation. *Apache Lucene*. https://lucene.apache.org/

[13] T. Macdonald and C. Macdonald. *PyTerrier: A Python framework for Terrier*. https://github.com/terrier-org/pyterrier