Université des sciences et de la technologie Houari Boumédiène

Master SII

Projet de TP en Recherche d'Information

AMMAR KHODJA Hichem
BOUDJENIBA Oussama

25 février 2021

Table des matières

Partie I : Introduction

Dans ce document, nous allons expliquer la procédure suivie pour construire un modèle de recherche d'information et l'appliquer sur la collection CACM. Ce document est diviser en plusieurs chapitres :

— Lecture et prétraitement de la collection CACM

— Conception du modèle Booléen

— Conception du modèle vectoriel

—

Partie II : Lecture et prétraitement de la collection CACM

II.1 Lecture et extraction

On commence par lire le fichier *cacm.all*. Ensuite, Chaque document dans CACM commence par **.I** ce qui nous permet de détecter le début et fin de chaque document. Après, on extrait de chaque document son numéro d'identification (qui vient après le **.I**), le titre (**.T**), le nom des auteurs (**.A**) et éventuellement le résumé (**.W**).

II.2 Tokenisation

Maintenant, on construit un dictionnaire qui a comme clé l'ID du document et la valeur contient le reste des informations (**.T, .A et .W**) regroupées en une seule chaîne de caractères qu'on mets **en minuscule**. On procède à présent à la phase de tokenisation, on définit un token comme étant une suite de caractères alphanumériques ce qui est équivalent à l'expression régulière suivante **"\w+"**. On utilise la **stopword liste de NLTK** pour éliminer les mots non-significatifs.

II.3 Résultat

Le résultat est un dictionnaire contenant comme clé l'identificateur d'un document et la valeur correspondante est une liste des tokens contenus de ce document.

Partie III : Modèle booléen

III.1 Structure de données

Une structure de données plus appropriée est requise pour optimiser le temps d'exécution. On obtient cette structure en modifiant le dictionnaire obtenu dans la

section II.3 suivant l'algorithme :

Algorithme 1 : Construction de la structure de données pour le modèle booléen

Entrées : D : Dictionnaire obtenu dans la section II.3
Sorties : D' : Dictionnaire prêt pour le modèle booléen

```
1  D' ← Dictionnaire() ;
2  pour id, L_tokens dans D faire
3  |  S ← ensemble(L_tokens) ;
4  |  pour token dans S faire
5  |  |  si token n'est pas dans D' alors
6  |  |  |  D'.ajouter(token, {}) ;
7  |  |  fin
8  |  |  D'[token].ajouter(id) ;
9  |  fin
10 fin
11 retourner D';
```

On sauvegarde aussi la liste de tous les documents dans un ensemble, ce qui sera utile par la suite pour l'opérateur **not** (Voir ci-dessous).

Le résultat est un dictionnaire contenant comme clé un **token** et comme valeur **l'ensemble des documents contenant ce token**.

III.2 Évaluation des requêtes

Un exemple d'une requête du modèle booléen :

('Mathematics' or 'Science') and not 'Algebra'

Comme on peut l'observer, les tokens sont entourés d'apostrophe et qui sont reliés entre-eux par des opérateurs booléens (sans apostrophes). Les trois opérateurs booléens acceptés sont : **and, or** et **not**. Les expressions booléennes peuvent être imbriquées.

Pour évaluer une requête, on commence par vérifier que la syntaxe est correcte. Ensuite, on remplace respectivement les opérateurs **and, or** et **not** par **&, |** et **-**. Si on suit l'exemple précédant, ça nous donne :

('Mathematics' | 'Science') & -'Algebra'

Maintenant, on entoure chaque token de la manière suivante : **'token'** → **Token('token')**. Ce qui nous donne :

(Token('Mathematics') | Token('Science')) & -Token('Algebra')

Token est une classe qu'on a codé dont le comportement ressemble à celui de la classe **set** dans Python. A l'initialisation d'une instance de cette classe de la manière

suivante : **Token(T)** où **T** est une chaîne de caractères, on récupère l'ensemble des documents contenant le token **T**.

Petite remarque : Cette manoeuvre justifie le choix de la structure de données présentée dans la section III.1 car, elle nous garantit l'accès en O(1).

Maintenant, il n'y a plus qu'à définir le comportement des opérateurs booléens :

- **La conjonction de deux Token (T1 and T2)** : retourne un nouveau **Token** contenant l'intersection de l'ensemble de documents de **T1** et l'ensemble de documents de **T2**.
- **La disjonction de deux Token (T1 or T2)** : retourne un nouveau **Token** contenant l'union de l'ensemble de documents de **T1** et l'ensemble de documents de **T2**.
- **La négation d'un Token T** : retourne un nouveau **Token** contenant le complément de l'ensemble des documents de **T** par rapport à l'ensemble de tous les documents.

Enfin, il suffit d'exécuter la commande **eval(R)** de Python, **R** étant la requête transformée, ce qui nous retourne le résultat de la requête par le modèle booléen.

III.3 Temps d'exécution

Pour évaluer les performances du modèle booléen par rapport au temps d'exécution de l'évaluation d'un requête, on a fait l'expérience suivante :

— On génère 100000 requête aléatoires : 1000 requête contenant 1 token, 1000 requête contenant 2 tokens, ..., 1000 requête contenant 100 tokens. On relie les tokens aléatoirement par les opérateurs booléens **(and, or, not)**.

— On évalue ces requêtes par le modèle booléen et en enregistre le temps d'exécution.

— On regroupe les résultats par nombre de tokens et on calcule le temps d'exécution moyen.

Maintenant, on affiche le graphe de la fonction **F(Nombre de tokens) = Temps d'exécution** dans la Figure 1. On observe que le temps d'exécution dépend linéairement du nombre de tokens dans la requête booléenne.

III.4 Exemple d'une requête

On teste notre modèle sur la collection CACM en lui demandant d'évaluer la requête suivante :

('science' or 'compiler') and not 'algebra' and 'code'

Le modèle nous retourne une série de documents (leurs identifiants) : **123, 1223, 1234, 1542, 1551, 1613, 1807, 2064, 2423, 2433, 2897, 2968, 3080** comme on peut le voir dans la Figure 2.

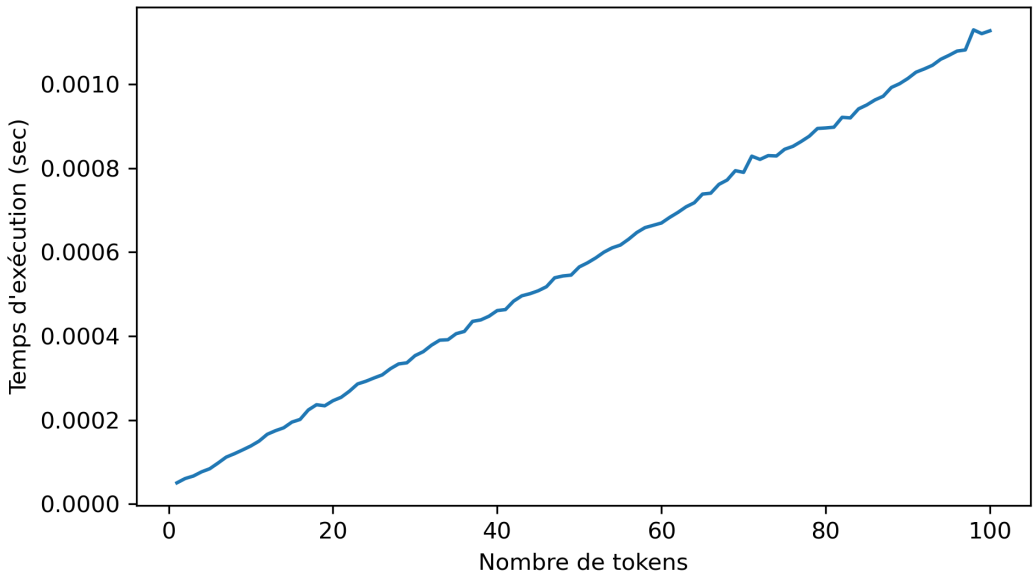

FIGURE 1 – Performances du modèle booléen en termes de temps d'exécution

Tester une requête (Modèle booléen) ¶

```
: query = " ('science' or 'compiler') and not 'algebra' and 'code'"
  bm.eval(query)
```

```
{123, 1223, 1234, 1542, 1551, 1613, 1807, 2064, 2423, 2433, 2897, 2968, 3080}
```

FIGURE 2 – Exemple d'une requête pour le modèle booléen

Partie IV : Modèle vectoriel

IV.1 Structure de données

Comme pour le modèle booléen, une structure de données plus appropriée est requise pour optimiser le temps d'exécution, à savoir, le **fichier inverse**. On obitent cette structure en modifiant le dictionnaire obtenu dans la section II.3 suivant l'algorithme :

La fonction Count(L) : Prend en entrée une liste **L** de tokens et retourne un dictionnaire qui a comme clé un token et comme valeur le nombre d'occurrences de ce token dans la liste **L**.
Exemple : **L** = ['A', 'B', 'C', 'B', 'C', 'D'] $\longrightarrow$ **Count(L)** = {'A' : 1, 'B' : 2, 'C' : 2, 'D' : 1}

On garde aussi la fréquence maximale de chaque document dans un dictionnaire qu'on nomme max_freq. Ce dictionnaire nous sera utile par la suite pour la pondération du fichier inverse, notamment, pour l'optimisation du temps d'exécution. On

sauvegarde aussi le nombre total de documents N pour les mêmes raisons.

Algorithme 2 : Construction de la structure de données pour le modèle vectoriel

```
Fonction Count(Liste L)
Debut
    D ← Dictionnaire();
    pour token dans L faire
        si token dans D.keys() alors
            D[token] ← 0;
        fin
        D[token] ← D[token] + 1 ;
    fin
    retourner D;
Fin
/* On passe maintenant à la construction du fichier inverse          */
```

Entrées : D : Dictionnaire obtenu dans la section II.3

Sorties : D' : Dictionnaire prêt pour le modèle vectoriel (Fichier inverse), max_freq : Dictionnaire contenant la fréquence de chaque document, N : le nombre total de documents

```
D' ← Dictionnaire() ;
max_freq ← Dictionnaire() ;
N ← taille(D) ;
pour id, L_tokens dans D faire
    S ← Count(L_tokens) ;
    max_freq[id] ← max(L_tokens) pour token, n dans S faire
        si token n'est pas dans D' alors
            D'.ajouter(token, Dictionnaire()) ;
        fin
        D'[token][id] ← n ;
    fin
fin
retourner D', max_freq, N;
```

IV.2 Les fonctions d'accès

- **Freq(token, id)** : Prend en entrée un token et l'identificateur d'un document (**id**) et retourne la fréquence du token dans ce document. Cette fonction est très simple : Si **D** est le fichier inverse (dictionnaire de dictionnaires), cette fonction retourne **D**[token][id]. Cette fonction est de complexité **O(1)**. Cette fonction retourne 0 si la paire **(token, id)** n'existe pas dans le fichier inverse.
- **Documents(token)** : Prend en entrée un token et retourne les documents contenant ce document avec la fréquence du token dans ces documents. Si **D** est le fichier inverse, cette fonction retourne **D**[token]. Cette fonction est de

complexité **O(1)**.

- **Tokens(id)** : Prend en entrée un identificateur d'un document (**id**) et retourne les tokens qui sont présents dans ce document et leurs fréquences dans ce document. Cette fonction parcourt l'ensemble du fichier inverse et enregistre les tokens qui sont présents dans le document **id** avec leurs fréquences. Cette fonction est donc, de complexité **O(n)**, **n** étant le nombre de tokens dans le fichier inverse.

IV.3 Pondération TF-IDF

Pour chaque paire **(token, document)** dans le fichier inverse, on calcule le poids de cette paire selon la formule TF-IDF suivante :

$$poids(t_i, d_j) = \frac{freq(t_i, d_j)}{max\_freq(d_j))} log_{10}\left(\frac{N}{n_i} + 1\right)$$

Tel que :

- $freq(t_i, d_j)$ est la fréquence du token t_i dans le document d_i.
- $max\_freq(d_j) = max(\{freq(t, d_j) : t \in d_j\})$ est la fréquence du token qui apparaît le plus dans le document d_j. Cet valeur a été calculée pour chaque document dans l'Algorithme 2 où les résultats sont stockées dans la variable max_freq, ceci nous évite de recalculer à chaque itération.
- N : le nombre total de documents qu'on a calculé dans l'Algorithme 2.
- n_i : Le nombre de documents contenant le token t_i.

On déroule, donc, l'algorithme suivant pour pondérer le fichier inverse :

Algorithme 3 : Pondération du fichier inverse selon la formule TF-IDF

Entrées : D : Fichier inverse, max_freq : la fréquence maximale de chaque document, N : le nombre total de documents

```
pour t, D_doc dans D faire
    n ← taille(D_doc) ;
    pour d dans D_doc.keys() faire
        D_doc[d] ← D_doc[d]/max_freq[d] * log10(N/n + 1) ;
    fin
fin
```

IV.4 Calcul des normes

Hors la fonction de similarité *produit interne*, les autres fonctions de similarité (Dice, Jaccard, Cosinus) requièrent le calcul de la norme euclidienne des documents. Pour optimiser la phase d'évaluation des requêtes, on calcule à l'avance cette norme pour tous les documents et on stock le résultat dans un dictionnaire **(document, norme)** qu'on appellera doc_norm.

Soit d un document de la collection. On calcule $doc\_norm[d]$ grâce à la formule suivante :

$$doc\_norm[d] = \sqrt{\sum_{i=1}^{T} d[t_i]^2}$$

Tel que :

— T : est le nombre de tokens dans la collection

— $d[t_i]$: est le poids de la paire (d, t_i) dans le fichier inverse pondéré. Ce poids vaut 0 si $t_i \notin d$.

On a besoin de prendre en considération que les tokens présents dans le document d. En optimisant la formule précédente, on obtient :

$$doc\_norm[d] = \sqrt{\sum_{t \in d} d[t]^2}$$

IV.5 Évaluation des requêtes

D'abord, on commence par extraire les tokens de la requête en utilisant l'expression régulière "\w+" qu'on stocke dans une liste. Ensuite, on mets en minuscule les tokens et on supprime de le liste les tokens qui sont présents dans la *stoplist* de NLTK. On peut légitimement supposer que les tokens qui apparaissent plusieurs fois dans la requête sont plus importants que ceux qui n'apparaissent qu'une seul fois. Donc, on applique la fonction **Count** décrite dans la Section IV.1 à cette liste, ce qui nous donne un dictionnaire **(token, fréquence)** qu'on nomme R et on divise chaque fréquence de R par la fréquence maximale de R. Le résultat est un dictionnaire **(token, valeur)** où **valeur** est un nombre entre 0 et 1. Cette valeur représente l'importance d'un token dans la requête. Si tous les tokens apparaissent le même nombre de fois dans la requête, on aura $valeur = 1$ pour chaque token dans R. Le calcul de la norme euclidienne de la requête R se fait de la manière suivante : $\|R\|_2 = \sqrt{\sum_{t \in R} R[t]^2}$

IV.5.1 Calcul de la similarité

La similarité se calcule entre un document et une requête. Dans ce document, on va étudier quatre fonctions de similarité qui sont : *produit interne, Dice, cosinus* et *Jaccard*.

- **Produit interne** : Il se calcule avec la formule suivante :

$$PI(d, R) = \sum_{i=1}^{T} d[t_i] * R[t_i]$$

tel que :

— $d[t_i]$: est le poids de la paire (t_i, d) dans le fichier inverse.

— $R[t_i]$: est le poids du token t_i dans R (calculé plus haut). Ce poids vaut 0 si $t_i \notin R$.

— T : est le nombre total de tokens dans la collection.

On peut voir que si $t_i \notin R$, alors $d[t_i] * R[t_i] = 0$. Par conséquent, on peut simplifier et optimiser par la même occasion la formule précédente :

$$PI(d, R) = \sum_{t \in R} d[t] * R[t]$$

Voici l'algorithme pour calculer le produit interne de manière optimale pour tous les documents :

Algorithme 4 : Calcul du produit interne entre la requête et tous les documents

Entrées : D : Fichier inverse pondéré, R : Dictionnaire représentant la requête, *all_documents* : L'ensemble des identificateurs de tous les documents

Sorties : *doc_score* : Dictionnaire contenant pour chaque document, son produit interne avec la requête R

```
1 doc_score ← Dictionnaire() ;
2 pour d dans all_documents faire
3 |   doc_score[d] ← 0 ;
4 |   pour t, w dans R faire
5 |   |   doc_score[d] ← doc_score[d] + D.poids(t, d) * w ;
6 |   fin
7 fin
8 retourner doc_score ;
```

- **Dice** : Cette fonction de similarité se calcule avec la formule suivante :

$$Dice(d, R) = \frac{2 * PI(d, R)}{doc\_norm[d]^2 + \|R\|_2^2}$$

- **Cosinus** : Cette fonction de similarité se calcule avec la formule suivante :

$$Cosinus(d, R) = \frac{PI(d, R)}{doc\_norm[d] * \|R\|_2}$$

- **Jaccard** : Cette fonction de similarité se calcule avec la formule suivante :

$$Jaccard(d, R) = \frac{PI(d, R)}{doc\_norm[d]^2 + \|R\|_2^2 - PI(d, R)}$$

Après avoir calculé le score de chaque document avec la fonction de similarité choisie, on ordonne ces documents suivant leurs scores. On choisi les k-premiers documents et on les présente au client (choisir k selon l'application).

IV.6 Temps d'exécution

Pour évaluer les performances du modèle vectoriel par rapport au temps d'exécution de l'évaluation d'un requête, on a fait l'expérience suivante :

— On génère 2000 requête aléatoires : 10 requête contenant 1 token, 10 requête contenant 2 tokens, ..., 10 requête contenant 200 tokens. Les requêtes sont composées de tokens tirés au hasard.

— On évalue ces requêtes par le modèle vectoriel et en enregistre le temps d'exécution.

— On regroupe les résultats par nombre de tokens et on calcule le temps d'exécution moyen.

Maintenant, on affiche le graphe de la fonction **F(Nombre de tokens) = Temps d'exécution** dans la Figure 3. On observe que le temps d'exécution dépend linéairement du nombre de tokens dans la requête.

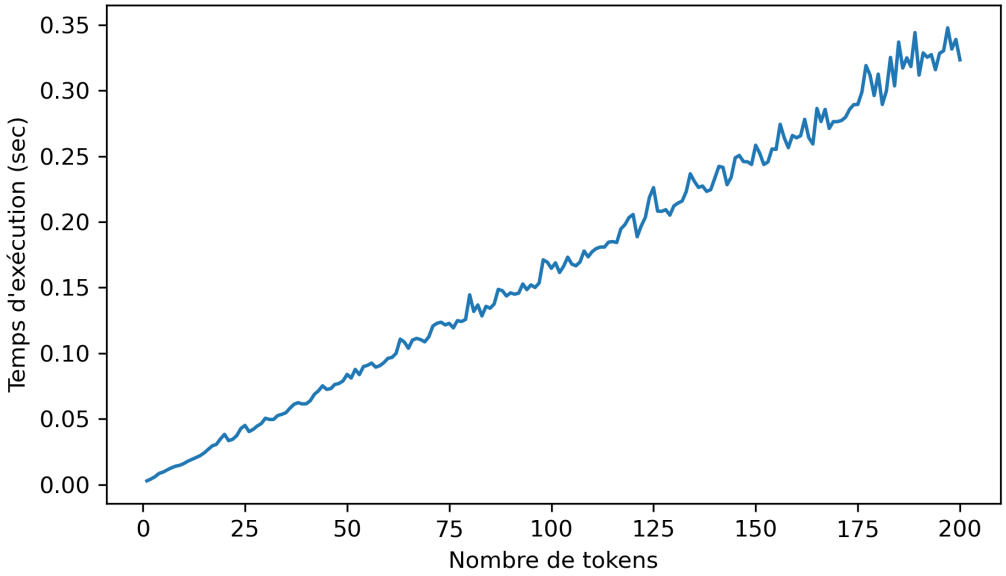

FIGURE 3 – Performances du modèle vectoriel en termes de temps d'exécution

IV.7 Exemple d'une requête

On teste notre modèle sur la collection CACM en lui demandant d'évaluer la requête suivante :

I want to consult a document about code optimization and compilers

Le modèle nous retourne une série de documents et leurs scores respectifs suivant la fonction de similarité choisie (Figure 4).

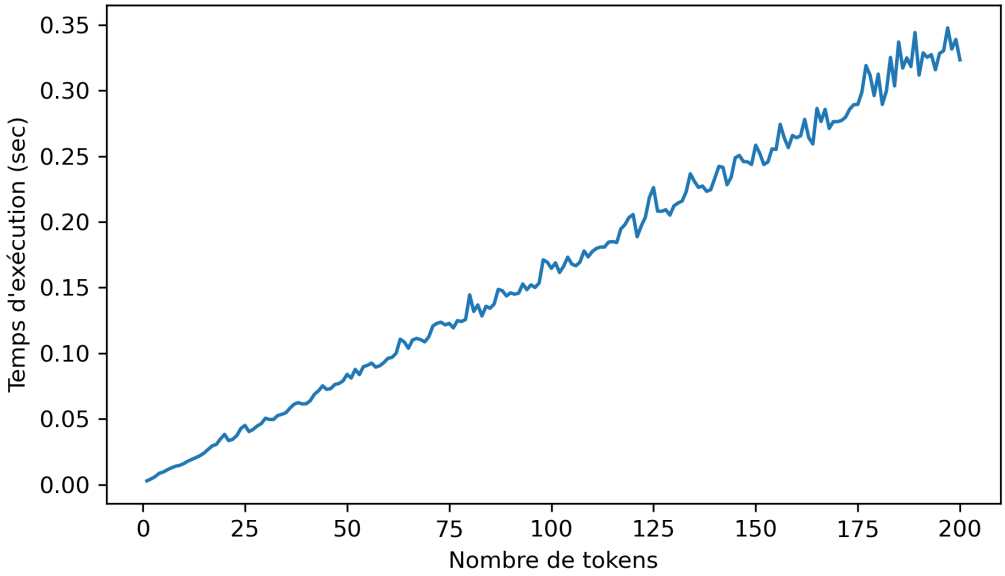

FIGURE 4 – Exemple du résultat d'évaluation d'une requête utilisant le modèle vectoriel avec comme fonction de similarité le *produit interne*

IV.8 Optimisation du modèle (Bonus/non demandé)

On a observé dans la section précédente que pour une requête de 100 tokens, l'évaluation de la requête prends 0.16s. Bien que ce temps parait court, il commence à poser problème si, par exemple, notre modèle reçoit 1000 requêtes simultanément. Un utilisateur attendra en moyenne 1000*0.16 = 16s ce qui est inacceptable compte tenu des standards actuels. Heureusement, il y a un meilleur moyen pour calculer la similarité requête/document : les matrices creuses.

La quasi totalité du temps d'exécution s'écoule lors du calcul du produit interne entre la requête et les documents. On peut utiliser les matrices creuses pour accélérer cela. Donc, au lieu d'avoir un dictionnaire de dictionnaires, on aura une matrice creuse M de dimension $T \times D$ (T : nombre de tokens, D : nombre de documents). La requête R est représentée par un vecteur ligne creux de dimension $1 \times T$. Ce vecteur contient 0 si le token n'appartient pas au vecteur sinon il contient le poids du token dans la requête.

Maintenant, pour calculer le produit interne entre R et tous les documents, il suffit de calculer $R \times M$, ce qui nous retourne un vecteur dont chaque composante est le produit interne entre un document et R. Ces matrices/vecteurs creux sont sous format **CSR (Compressed Sparse Row)** qui a été optimisé pour les opérations arithmétique (addition, soustraction, produit matriciel).

On refait la même expérience que dans la section IV.6 et cette fois-ci, les résultat sont nettement meilleur comme on peut le voir dans la Figure 5. On passe d'un temps d'exécution de 0.16s pour une requête de 100 tokens à 0.001s, ce qui est **160 fois plus rapide**.

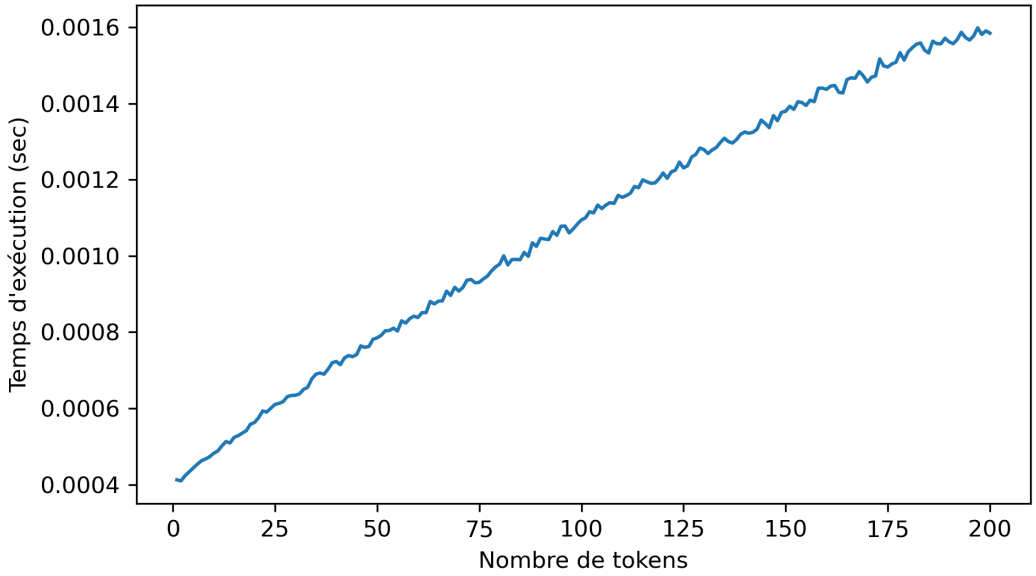

FIGURE 5 – Performances du modèle vectoriel avec matrices creuses en termes de temps d'exécution

Partie V : Évaluation du modèle vectoriel

Pour l'évaluation, on a codé une interface graphique (Figure 6) pour afficher les différentes métriques de performances pour chaque requête et pour chaque fonction de similarité (produit interne, cosinus, Dice et Jaccard). Ces métriques sont :

— Le rappel,

— La précision moyenne non-interpolée,

— La précision moyenne interpolée,

— La courbe précision-rappel non-interpolée,

— La courbe précision-rappel interpolée.

On a ajouté un paramètre supplémentaire qui est le **seuil**. Une fois qu'une requête est évaluée, le modèle vectoriel ne garde que les documents ayants un score plus grand que **seuil**. Ce seuil est égal à 0.05 par défaut et peut être modifié grâce au slider dans l'interface.

Dans la Figure 7, on a calculé la moyenne des courbes précision-rappel interpolées quand le seuil est égal à 0.05. On trouve que la fonction de similarité *Jaccard* est celle qui a les moins bonnes performances, suivi de *Dice*. Les deux meilleurs fonctions de similarité sont *Produit interne* et *Cosinus* avec des performances très similaires.

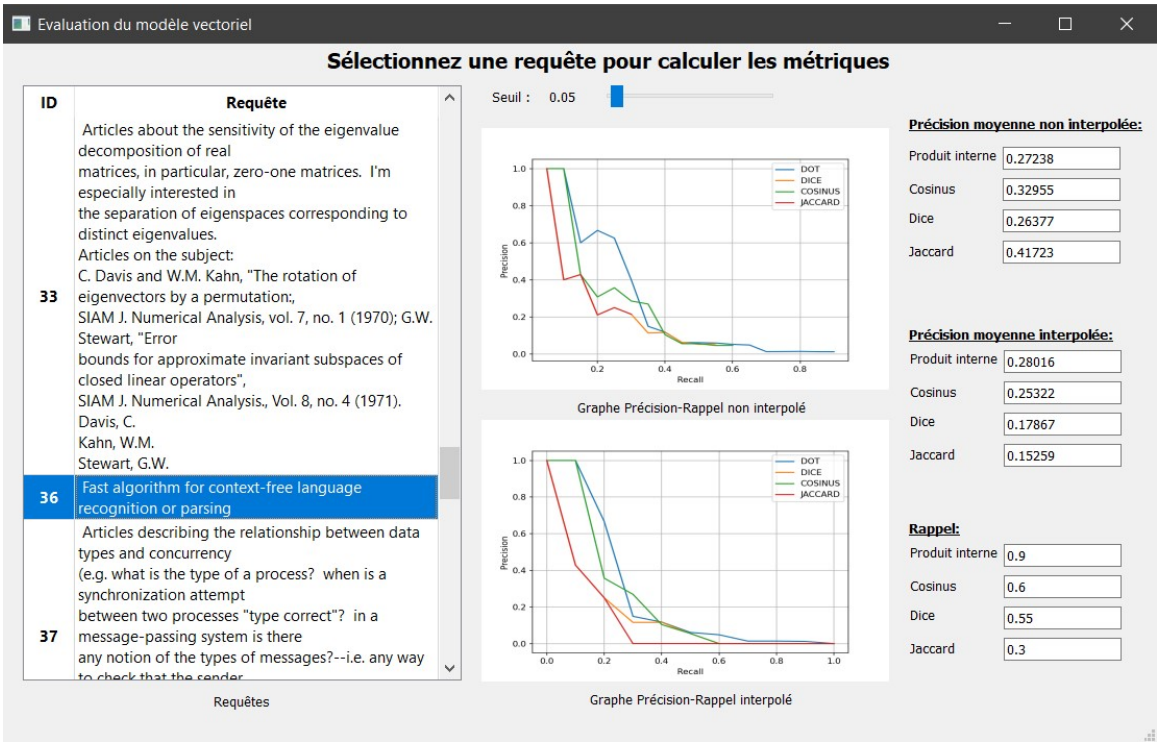

FIGURE 6 – Interface homme-machine

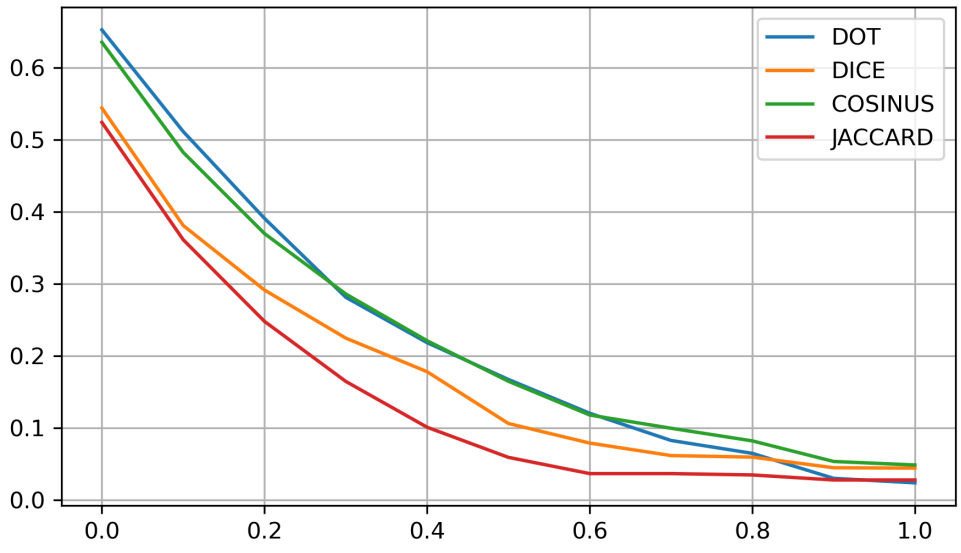

FIGURE 7 – Moyenne des courbes précision-rappel interpolées quand le seuil est égal à 0.05