Projet DAAR

Moteur de recherche d'une bibliothèque textuelle

Taha Yacine GUEZZEN
Mohamed Amine BENCHAA
Brahim HALLOUCHA DJEBBOUR

30 novembre 2025

Table des matières

1 Introduction

Ce projet s'inscrit dans le cadre du module **DAAR** et consiste à développer un moteur de recherche appliqué à une grande bibliothèque de textes issus du Projet Gutenberg. L'objectif principal est de mettre en pratique des concepts d'algorithmique et de structures de données autour :

- de la construction d'un **index inversé** sur un grand corpus textuel ;
- du calcul de **similarités** entre documents et de la construction d'un **graphe** de livres ;
- de l'utilisation d'un indice de **centralité** (PageRank) pour le classement ;
- de la prise en charge de **recherches avancées** via expressions régulières ;
- de la mise à disposition de ces fonctionnalités via une **API web** (Flask) et une **interface utilisateur moderne**.

Le moteur de recherche développé permet :

- la recherche simple par mot-clé.
- la recherche avancée par expression régulière (RegEx) appliquée sur le vocabulaire indexé.
- un classement hybride combinant le nombre d'occurrences et le score PageRank.
- des suggestions de livres similaires basées sur le graphe de similarité.
- le téléchargement direct des livres depuis l'interface.

Ce rapport décrit les différentes étapes du projet : constitution de la bibliothèque, prétraitement et indexation, construction du graphe de Jaccard, architecture de l'application web (backend et frontend), algorithmes de recherche et de classement, ainsi qu'une évaluation expérimentale des performances.

2 Constitution de la bibliothèque

2.1 Téléchargement automatique des livres

La constitution du corpus s'appuie sur un script Python dédié, `retrieveScript_corrigé`, qui automatise le téléchargement de livres depuis le Projet Gutenberg.

Le script :

- parcourt séquentiellement les identifiants de livres Gutenberg à partir de 1 ;
- construit pour chaque identifiant l'URL du fichier texte brut :

  `https://www.gutenberg.org/cache/epub/{id}/pg{id}.txt`

- vérifie que le téléchargement est réussi (code HTTP 200 et taille minimale) ;
- ne conserve que les livres dont le nombre de mots est supérieur à un seuil fixé à **10 000 mots** ;
- enregistre chaque livre valide dans un dossier `gutenberg_books/`.

Le paramètre `TARGET_BOOKS` est fixé à 1664, ce qui permet de respecter la contrainte de taille minimale de la bibliothèque. Une temporisation est également présente pour éviter de surcharger le serveur distant.

2.2 Critères de filtrage et nommage des fichiers

Pour chaque livre téléchargé, le script tente d'extraire le titre à partir de l'en-tête Gutenberg (champ `Title:`). Le nom de fichier est ensuite construit à partir de ce titre :

- si un titre est trouvé, il est utilisé comme nom de fichier, après nettoyage des caractères interdits ;
- si aucun titre n'est trouvé, le livre est ignoré.

Ainsi, la bibliothèque finale est constituée d'un ensemble de fichiers texte stockés dans `gutenberg_books/`, chacun représentant un livre avec au moins 10 000 mots.

3 Prétraitement et index inversé

3.1 Normalisation et nettoyage du texte

La phase de prétraitement est réalisée par le script `createIndex`.

Les étapes appliquées à chaque fichier de `gutenberg_books/` sont les suivantes :

- **Passage en minuscules** : tous les caractères sont convertis en minuscules.
- **Suppression des accents** : une normalisation Unicode (`NFKD`) est utilisée, puis les diacritiques sont retirés.
- **Découpe en mots** : le texte normalisé est découpé via `split()` en tokens simples.
- **Nettoyage des mots** : la fonction `clean_word()` :
  - supprime la ponctuation via une expression régulière ;
  - ne conserve que les mots de longueur strictement supérieure à 2 ;
  - ignore les mots présents dans une liste de **stopwords** (français et anglais).

Ce prétraitement permet de réduire la taille du vocabulaire et de se concentrer sur des mots informatifs.

3.2 Construction et sérialisation de l'index inversé

L'index inversé est construit sous la forme d'un dictionnaire imbriqué :

```
index[word][filename] = count
```

c'est-à-dire pour chaque mot, un sous-dictionnaire associant chaque livre au nombre d'occurrences du mot dans ce livre.

La structure en mémoire est un `defaultdict(lambda: defaultdict(int))`, ce qui permet d'incrémenter facilement les compteurs. Pour chaque mot nettoyé :

- si le mot est valide, on incrémente `index[mot][fichier]` ;
- on ajoute le nom du fichier dans une liste `book_list`.

À la fin du parcours de tous les livres :

— l'index est sérialisé au format JSON dans `index.json` avec la structure :

```
{"mot": {"files": [...], "occurrences": {fichier: nb_occ}}}
```

— la liste des livres est sauvegardée dans `books_list.json`.

Ces deux fichiers constituent la base de données principale utilisée ensuite par l'API Flask.

4 Construction du graphe de similarité Jaccard

4.1 Tokenisation normalisée par livre

Pour capturer la similarité globale entre livres, un **graphe de Jaccard** est construit à partir des mots contenus dans chaque livre.

Chaque livre est lu et tokenisé via une fonction `tokenize_text()` qui :

— extrait des mots uniquement alphabétiques au moyen d'une expression régulière ;
— applique la même normalisation Unicode (minuscules + suppression des accents) que pour l'index inversé ;
— ne conserve que les mots de longueur supérieure ou égale à un seuil (par défaut 3).

Les mots de chaque livre sont stockés dans un ensemble (`set`) afin de ne garder que les mots uniques, ce qui est cohérent avec l'utilisation de l'indice de Jaccard.

4.2 Calcul du coefficient de Jaccard

Soient deux livres A et B, représentés chacun par un ensemble de mots uniques W_A et W_B. La similarité de Jaccard est définie par :

$$J(A, B) = \frac{|W_A \cap W_B|}{|W_A \cup W_B|}$$

Une fonction `calculate_jaccard(set1, set2)` implémente directement cette formule, en prenant soin de gérer les cas où l'union est vide (retour de 0.0).

4.3 Seuil et création du graphe

Le graphe est représenté par un objet `networkx.Graph`. Les étapes sont :

— ajout d'un nœud pour chaque livre ;
— calcul du coefficient de Jaccard pour chaque paire de livres ;
— ajout d'une arête (A, B) avec attribut `weight = J(A,B)` si $J(A, B)$ dépasse un seuil fixé (0.1).

Des statistiques supplémentaires (nombre de nœuds, nombre d'arêtes, densité du graphe, degrés moyens/max/min, top livres les plus connectés, etc.) sont calculées et sauvegardées dans un fichier JSON.

Le graphe final est sauvegardé dans `jaccard_graph.gpickle` via `pickle`, et servira ensuite de base à l'API pour le calcul de PageRank et les suggestions.

5 Architecture globale de l'application

5.1 Vue d'ensemble

L'application complète suit une architecture en trois couches :

- **Couche données** : corpus de livres (`gutenberg_books/`), index inversé (`index.json`), liste des livres (`books_list.json`) et graphe de Jaccard (`jaccard_graph.gpickle`).
- **Backend** : API web développée en Flask, qui charge les données, répond aux requêtes de recherche, calcule les résultats et fournit des statistiques.
- **Frontend** : interface web moderne en HTML/CSS/JavaScript, basée sur Tailwind CSS, qui interagit avec l'API via des requêtes HTTP.

L'API peut être exposée localement ou via un tunnel (par exemple Cloudflare Tunnel), et le frontend consomme les endpoints de l'API en utilisant une constante `API_URL`.

6 Backend Flask : API de recherche

Au démarrage, l'application Flask :

- charge l'index inversé (`index.json`) ;
- charge la liste des livres (`books_list.json`) ;
- charge le graphe Jaccard (`jaccard_graph.gpickle`) ;
- calcule une fois pour toutes le **PageRank** de chaque livre via `networkx.pagerank`.

Une fonction utilitaire `normalize_text()` applique la même normalisation (minuscules + suppression des accents) aux requêtes utilisateurs.

7 API Backend : résumé des endpoints

7.1 /health

Renvoie un état synthétique de l'API : `status`, nombre de livres, taille du vocabulaire et dimensions du graphe.

7.2 /search (recherche simple)

Paramètres : `query` et `ranking` (`hybrid`, `occurrences`, `pagerank`). Pipeline : normalisation, vérification dans l'index, récupération des fichiers concernés puis classement via `rank_results()`, qui calcule occurrences, PageRank et score hybride :

$$\text{score} = \text{occurrences} \times (1 + 10 \times \text{pagerank}).$$

7.3 /advanced-search (RegEx)

Recherche avancée par expression régulière sur les mots de l'index. Étapes : compilation de la RegEx, filtrage des mots correspondants, agrégation des fichiers concernés et classement final avec `rank_results()`. La réponse fournit `matching_words` et `results`.

7.4 /suggestions

Génère des recommandations à partir d'un mot : normalisation, classement hybride, sélection des 3 meilleurs livres, récupération de leurs voisins dans le graphe, filtrage et tri des suggestions selon la similarité et le PageRank.

7.5 /book, /stats, /download

- `/book/<filename>` : métadonnées d'un livre (PageRank, degré, présence dans le graphe) ;
- `/stats` : statistiques globales (livres, vocabulaire, densité, top PageRank, etc.) ;
- `/download/<filename>` : téléchargement du texte brut d'un livre.

8 Frontend : interface utilisateur web

Le frontend, défini dans `index.html` et stylisé avec Tailwind CSS, s'organise en deux sections principales :

- **Accueil** : titre « BookSearch », champ de recherche, choix du type de requête (Simple ou RegEx), sélection du mode de classement (hybride, occurrences, PageRank) et affichage de statistiques globales.
- **Résultats** : récapitulatif de la requête, liste de résultats sous forme de cartes (rang, titre, métriques, téléchargement) et bouton de retour à l'accueil.

8.1 Autocomplétion et recherche assistée

L'interface propose une autocomplétion dynamique : à chaque saisie, une requête `/advanced-search` est envoyée avec le préfixe courant, les correspondances sont affichées dans une liste déroulante, et un clic sur une suggestion remplit le champ puis lance automatiquement la recherche complète.

8.2 Illustrations de l'interface

Cette section présente trois captures d'écran représentatives de l'interface utilisateur de BookSearch : autocomplétion sur un mot fréquent, affichage des résultats classés, puis autocomplétion en mode RegEx.

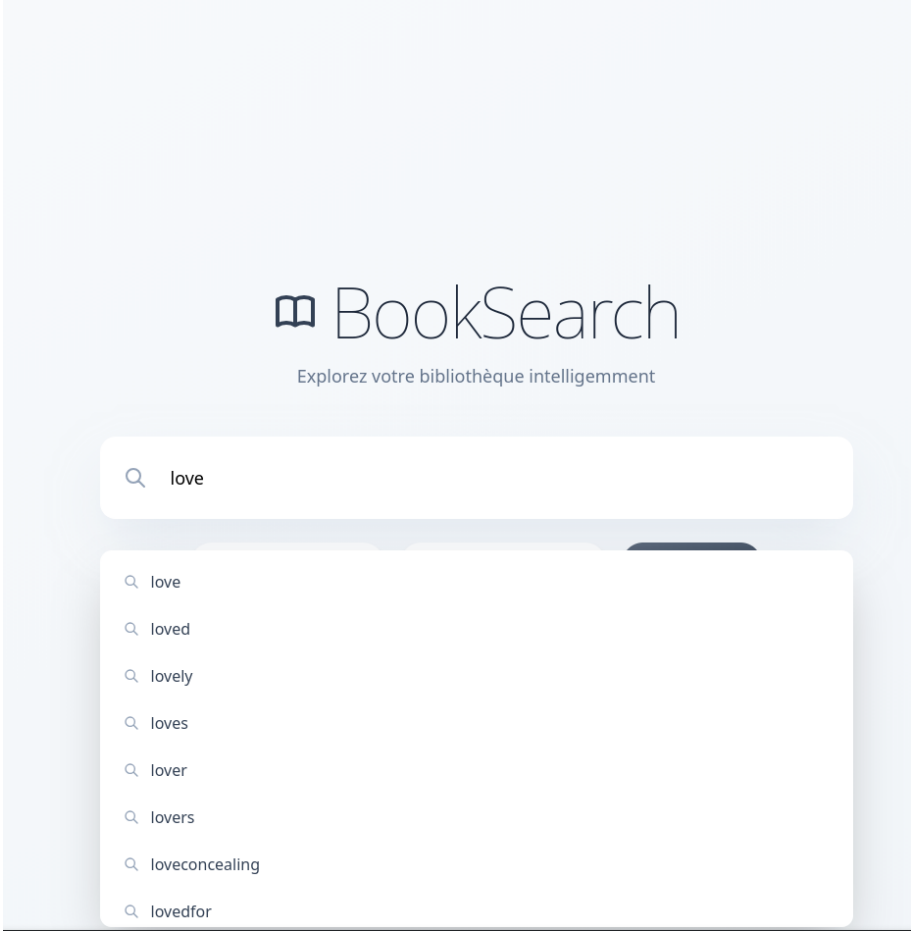

FIGURE 1 – Autocomplétion pour un mot fréquent (`"love"`).

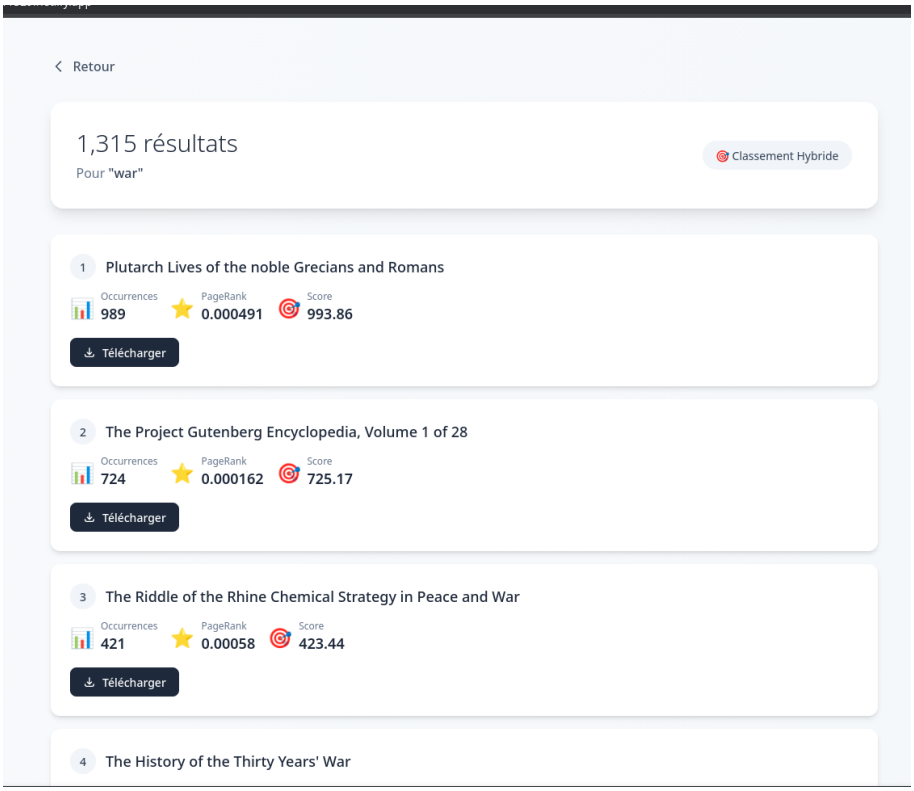

FIGURE 2 – Page de résultats pour la requête `"war"`.

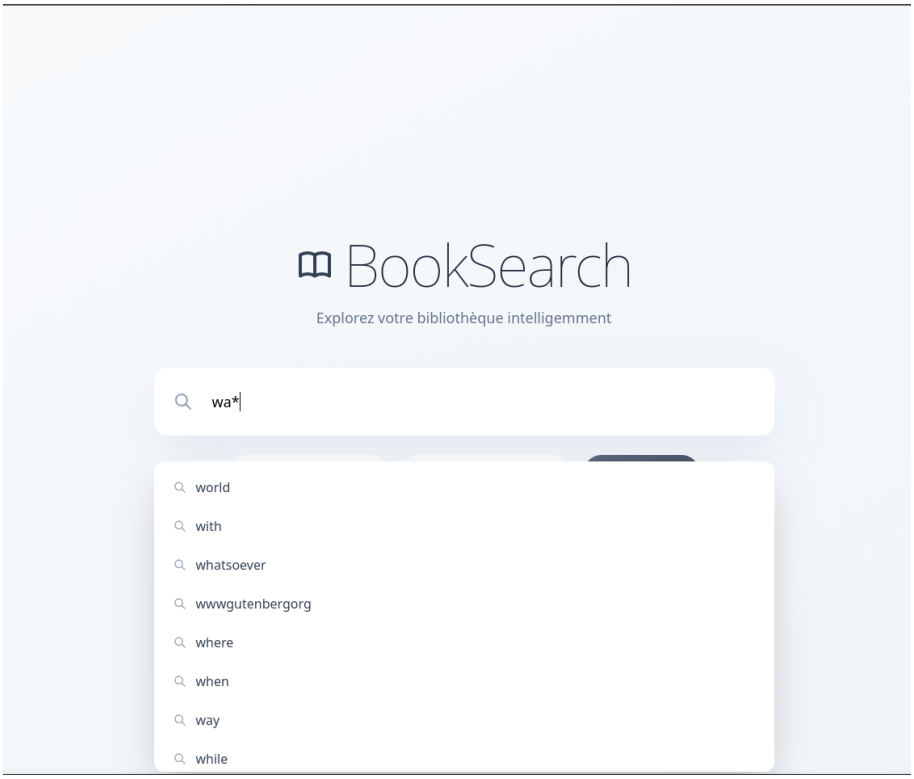

FIGURE 3 – Autocomplétion en mode RegEx (`"wa*"`).

9 Algorithmes de recherche, classement et suggestion

9.1 Recherche par mot-clé

Pour une recherche simple, la logique de base est :

1. normaliser le mot-clé saisi.
2. vérifier sa présence dans l'index.
3. récupérer la liste des livres où il apparaît.
4. pour chaque livre, obtenir :
   - le nombre d'occurrences du mot dans ce livre.
   - le score PageRank du livre.
5. calculer un score (en fonction du type de classement choisi).
6. trier les livres par score décroissant.

9.2 Recherche avancée par expressions régulières

La recherche avancée s'appuie sur le même index, mais remplace la condition d'égalité stricte par un motif RegEx appliqué aux mots. Dans le pire cas, tous les mots du vocabulaire doivent être testés, mais on évite un scan complet de tous les fichiers texte.

9.3 Classement hybride par occurrences et PageRank

Le classement hybride combine deux critères :

- **pertinence locale** : le nombre d'occurrences du mot dans le livre ;
- **importance globale** : le score PageRank dans le graphe de similarité.

La formule utilisée est :

$$\text{score} = \text{occurrences} \times (1 + 10 \times \text{pagerank})$$

ce qui donne plus de poids aux livres centraux dans le graphe.

9.4 Suggestions basées sur le graphe de Jaccard

Les suggestions utilisent le graphe comme modèle de *proximité* entre livres. On prend d'abord les meilleurs résultats d'une recherche, puis on propose leurs voisins dans le graphe, triés selon une combinaison de similarité et de PageRank.

10 Évaluation expérimentale

10.1 Protocole de test

Les performances de l'API ont été évaluées à l'aide d'un script Python dédié (`performance_test.py`) qui :

- envoie des requêtes HTTP aux endpoints `/search`, `/advanced-search` et `/suggestions` ;
- couvre toutes les catégories de requêtes (common, medium, rare, complex, suggestions) ;
- mesure pour chaque appel le temps de réponse, le code HTTP et le nombre de résultats ;
- réalise un test de charge avec 50 requêtes concurrentes ;
- génère automatiquement les fichiers JSON, les graphiques (.png) et un rapport texte.

Au total, 23 requêtes différentes ont été envoyées pour les tests de précision (15 recherches simples, 5 RegEx, 3 suggestions), complétées par un test de charge de 50 requêtes.

10.2 Statistiques globales des temps de réponse

Sur l'ensemble des 23 requêtes de mesure :

- temps de réponse minimum : **0,001 s** ;
- temps de réponse maximum : **2,354 s** (RegEx complexe) ;
- temps de réponse moyen : **0,172 s** ;
- médiane : **0,006 s** ;
- écart-type : **0,500 s**.

La très grande majorité des requêtes est donc traitée en moins de 10 ms, les temps élevés étant dus uniquement aux requêtes RegEx les plus complexes.

La figure 4 présente la distribution des temps de réponse, leur évolution au fil des requêtes et un résumé statistique.

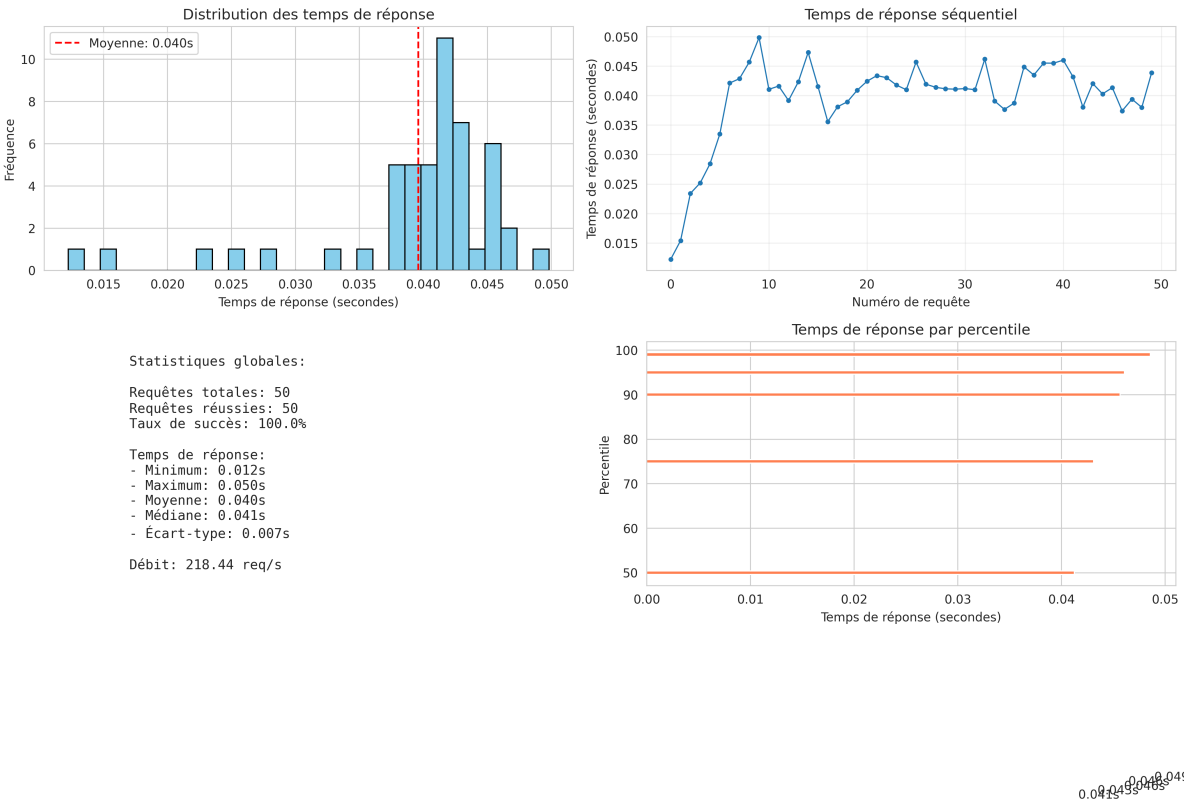

FIGURE 4 – Distribution des temps de réponse, évolution séquentielle et percentiles lors du test de charge.

10.3 Comparaison recherche simple vs RegEx

Les temps de réponse ont été analysés séparément pour les types de requêtes :

- **Recherche simple** : 15 requêtes, temps moyen **0,003 s**, min 0,001 s, max 0,007 s, écart-type 0,002 s ;
- **Recherche RegEx** : 5 requêtes, temps moyen **0,776 s**, min 0,202 s, max 2,354 s, écart-type 0,891 s ;
- **Suggestions** : 3 requêtes, temps moyen **0,006 s**.

La figure 5 (boxplot) illustre clairement l'écart de performance entre recherche simple et recherche RegEx : la recherche simple est quasi instantanée, alors que les RegEx complexes sont 2 à 3 ordres de grandeur plus lentes.

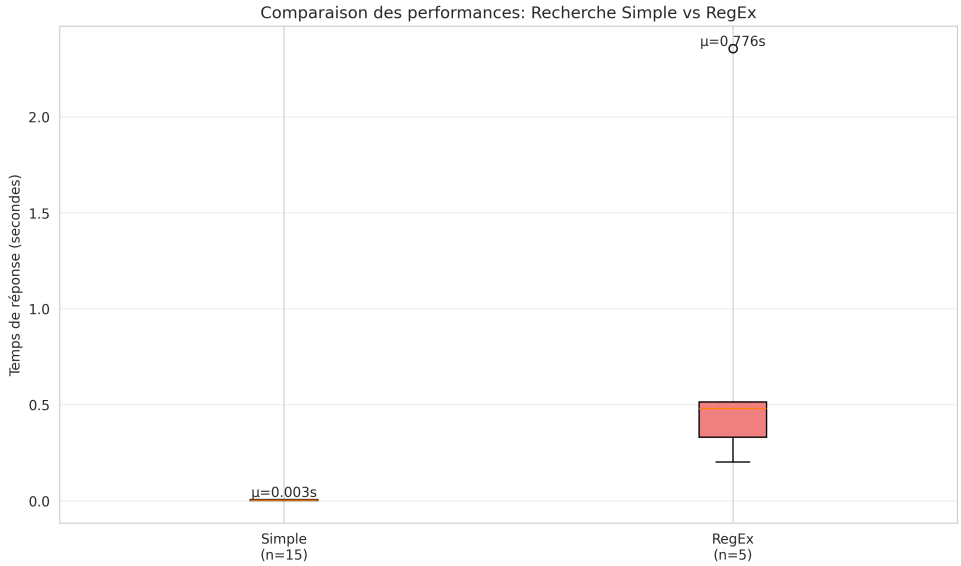

FIGURE 5 – Comparaison des temps de réponse entre recherche simple et recherche RegEx.

Ce résultat est cohérent avec le fait que la recherche simple se limite à une consultation de l'index inversé, alors que la recherche RegEx doit parcourir l'ensemble du vocabulaire indexé et filtrer les mots par motif.

10.4 Analyse par catégories de requêtes

Les requêtes simples ont été réparties en plusieurs catégories :

Les catégories de requêtes, leurs exemples et leurs temps moyens sont les suivants :

- *common* (ex. `the`, `and`) : **0,001 s** ;
- *medium* (ex. `love`, `time`) : **0,007 s** ;
- *rare* (ex. `quantum`, `serendipity`) : **0,002 s** ;
- *complex* (RegEx) : **0,776 s** ;
- *suggestions* (`/suggestions`) : **0,006 s**.

La figure 6 montre la distribution des temps de réponse par catégorie, confirmant que seules les requêtes *complex* (RegEx) sont coûteuses.

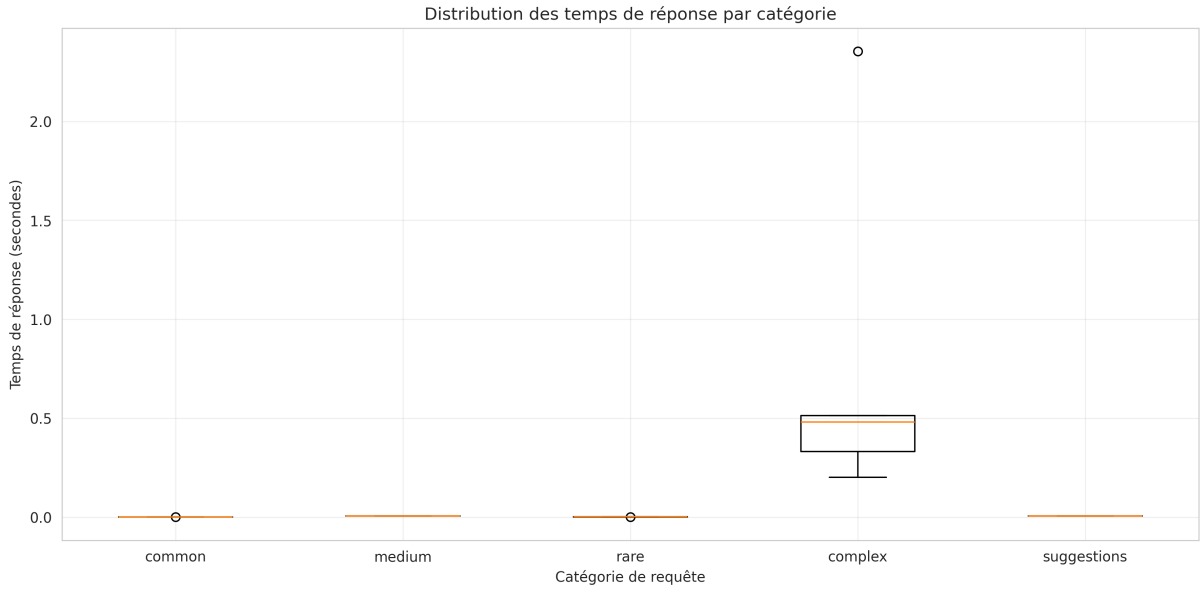

FIGURE 6 – Distribution des temps de réponse par catégorie de requête.

La figure 7 présente un résumé global : temps moyens par type de recherche, violin plot par catégorie, taux de succès (100 %) et une heatmap des temps moyens par couple (catégorie, type).

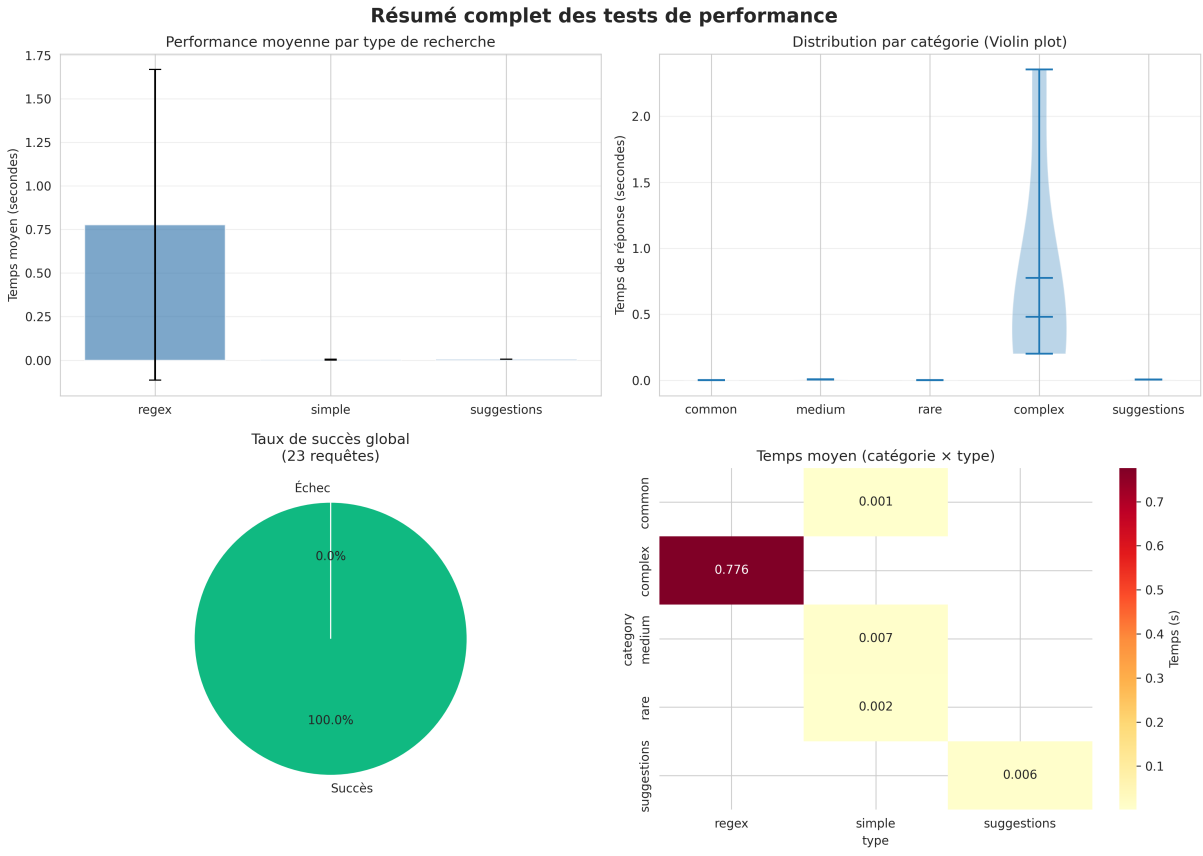

FIGURE 7 – Résumé complet des tests de performance (moyennes, distributions, taux de succès, heatmap).

On observe que :

— les requêtes simples et les suggestions restent toujours sous 10 ms ;
— les RegEx complexes constituent la principale source de latence ;
— le taux de succès est de 100 % pour l'ensemble des scénarios testés.

10.5 Comparaison des méthodes de classement

Les trois méthodes de classement (*hybrid*, *occurrences*, *pagerank*) ont été comparées en mesurant le temps de réponse pour la même requête (`"love"`) :

— **hybrid** : 0,0057 s ;
— **occurrences** : 0,0062 s ;
— **pagerank** : 0,0059 s.

Les statistiques montrent que les trois méthodes présentent des temps de réponse très proches (de l'ordre de quelques millisecondes). Cela s'explique par le fait que le calcul du PageRank est pré-calculé lors du chargement du graphe et que les opérations effectuées lors de la requête ne sont que des combinaisons de valeurs déjà disponibles.

10.6 Test de charge concurrente

Un test de charge a été effectué avec 50 requêtes simultanées, réparties sur 10 threads :

— temps total : **0,23 s** ;
— débit moyen : **environ 218 requêtes/s** ;
— taux de succès : **100 %**.

Ces résultats montrent que, dans la configuration actuelle, le backend est capable de servir plusieurs dizaines de requêtes par seconde avec une latence moyenne très faible, tant que les requêtes ne sont pas dominées par des motifs RegEx très coûteux.

11 Conclusion

Ce projet a permis de concevoir et d'implémenter un **moteur de recherche complet** appliqué à une large bibliothèque de textes. Il intègre :

— un pipeline de récupération automatique de livres depuis le Projet Gutenberg, avec filtrage par taille ;
— un prétraitement textuel avancé et la construction d'un index inversé enrichi ;
— un graphe de similarité fondé sur le coefficient de Jaccard, accompagné de statistiques et de calculs de PageRank ;
— une API REST sous Flask permettant la recherche simple, avancée via expressions régulières, les suggestions et le téléchargement ;
— un frontend moderne en Tailwind CSS offrant autocomplétion et différents modes de recherche.

Les tests ont montré d'excellentes performances pour la recherche simple et les suggestions, tandis que les requêtes utilisant des expressions régulières complexes restent plus coûteuses.