Rapport de Projet - NLP

Information Retrieval

Hajar Izandaz - Yacine Farhi - Hugo Da Cunha Carreira
ET5 IIM

16 octobre 2025

Université Paris-Saclay, Polytech Paris-Saclay, Bâtiment 620, maison de l'ingénieur, 91400, Orsay

POLYTECH PARIS-SACLAY

Sommaire

I. Introduction

Ce projet a pour objectif de créer un moteur de recherche à partir des compétences que nous avons acquises durant le cours de traitement automatique du langage. Nous disposons d'une base de données d'environ 2 000 fichiers texte provenant de Wikipédia. Ces textes abordent plusieurs sujets divers et variés, ce qui nous permettra de tester notre moteur de recherche.

On a également un fichier JSON contenant des requêtes à tester avec la réponse du fichier texte qui correspond le mieux.

Dans ce rapport, nous présentons les fonctionnalités que nous avons implémentées, l'architecture globale de notre application, ainsi que les méthodes de recherche utilisées. Nous détaillons aussi les métriques retenues pour évaluer la qualité des résultats et analysons les performances obtenues.

II. Fonctionnalités de l'application

Cette application implémente un moteur de recherche complet construit à partir d'un corpus d'environ 2 000 documents Wikipédia. Elle intègre l'ensemble des étapes d'un pipeline IR (Information Retrieval) : du prétraitement des textes jusqu'à l'évaluation des résultats.

Architecture de l'application

L'architecture est organisée en modules indépendants, chacun dédié à une étape du pipeline IR.

- src/ : cœur du projet, contenant les modules Python suivant :

- *corpus.py* : chargement et lecture du corpus.
- *preprocess.py* : nettoyage et transformation des textes.
- *index.py* : construction de l'index inversé.
- *search.py* : implémentation des méthodes de recherche (BM25, TF-IDF, RRF…).
- *metrics.py* : calcul des métriques d'évaluation.
- *suggest.py* : génération des suggestions de recherche (autocomplete).

- models/ : fichiers générés après la phase d'indexation :

- *index.json* : index inversé complet du corpus (TF, DF, longueur des documents, etc.).
- *edge_index.json* : index dédié aux suggestions de recherche (liste triée de préfixes et termes).

- data/ : ensemble des données nécessaires aux expérimentations :

- *wiki_split_extract_2k/* : répertoire contenant les ~2 000 documents Wikipédia utilisés pour construire l'index.

- *requetes.jsonl* : fichier regroupant les requêtes de test ainsi que les documents cibles pour l'évaluation.

- test/ : dossier utilisé au début du développement pour tester le prétraitement et l'indexation sur de petits exemples directement dans le terminal, avant de passer au corpus complet.

Enfin à la racine du projet nous avons les fichiers :

- *app.py / app_eval.py* : interfaces Streamlit pour la recherche et l'évaluation

Cette architecture modulaire permet de séparer clairement les données brutes, les fichiers générés, la logique métier et les interfaces utilisateur.

Prétraitement

Le prétraitement prépare les documents avant l'indexation afin d'obtenir une représentation propre et uniforme des textes. Notre pipeline, implémenté dans la classe **TextPreprocessor**, repose sur plusieurs étapes successives :

1. **Normalisation**

Passage en minuscules et suppression des accents pour uniformiser les mots.

2. **Tokenisation**

Extraction des tokens via une expression régulière qui ne garde que les caractères alphanumériques. Il est possible de retirer les tokens purement numériques.

3. **Filtrage (stopwords)**

Suppression des stopwords français définis dans une liste interne, ainsi que des tokens d'un seul caractère.
Cette étape réduit le bruit et élimine les mots très fréquents mais peu informatifs.

4. **Stemming**

Simplification des pluriels en retirant le *s* final (ex : "chats" → "chat").

5. **N-grams / edge n-grams**

Génération de bigrams pour les expressions et de préfixes pour l'autocomplétion.

Ce pipeline permet d'obtenir un texte propre et cohérent, facilitant l'indexation et améliorant la qualité de la recherche.

Indexation

L'indexation est assurée par la classe **InvertedIndex**. Elle parcourt tous les documents du corpus, applique le prétraitement, puis construit un index inversé où chaque terme est associé aux documents dans lesquels il apparaît ainsi qu'à sa fréquence

Pour chaque terme, on stocke :

- **postings** : liste des documents où le terme apparaît
    - terme → { doc_id : fréquence(tf) }
- **df** : nombre de documents contenant le terme
- **doc_len** : longueur de chaque document

- **N** : nombre total de documents
- **avgdl** : longueur moyenne des documents

L'index final est enregistré dans *index.json* pour éviter une reconstruction à chaque lancement.

Un second fichier, *edge_index.json*, contient les unigrams valides et leurs edge n-grams pour l'autocomplétion.

Autocomplétion

L'autocomplétion repose sur l'index dédié edge_index.json, généré à partir des unigrams du vocabulaire. Cet index contient, pour chaque mot, ses edge n-grams, c'est-à-dire ses préfixes successifs.

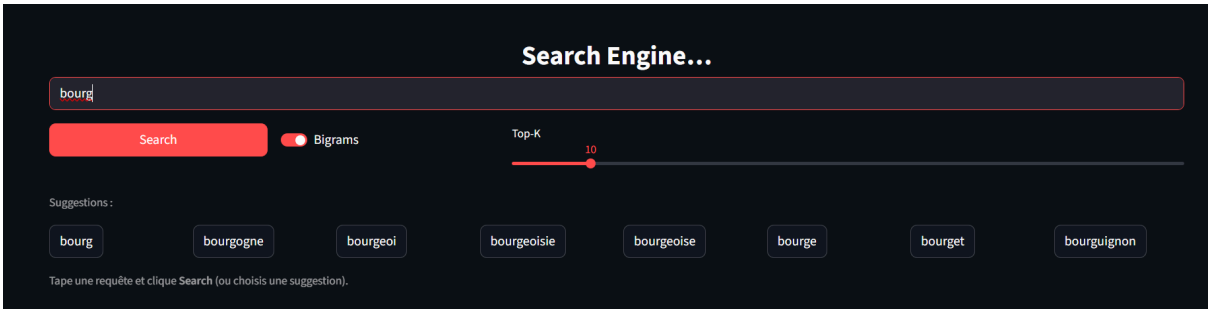
La classe Autocomplete charge cet index et utilise une liste triée de tous les mots pour retrouver rapidement les suggestions.
Le principe est simple :

- l'utilisateur tape un préfixe ;
- on cherche dans la liste des mots la première position où ce préfixe pourrait apparaître (via bisect_left) ;
- on renvoie les premiers mots qui commencent par ce préfixe, jusqu'à top_k résultats.

Cette méthode est légère, rapide et fonctionne en temps quasi instantané sans moteur externe comme Elasticsearch.

Interface de l'application

L'interface utilisateur a été développée avec Streamlit afin de proposer un moteur de recherche simple, interactif et agréable à utiliser. Elle permet d'exécuter des requêtes, d'afficher les suggestions, de visualiser les documents retrouvés et de naviguer facilement entre les résultats.

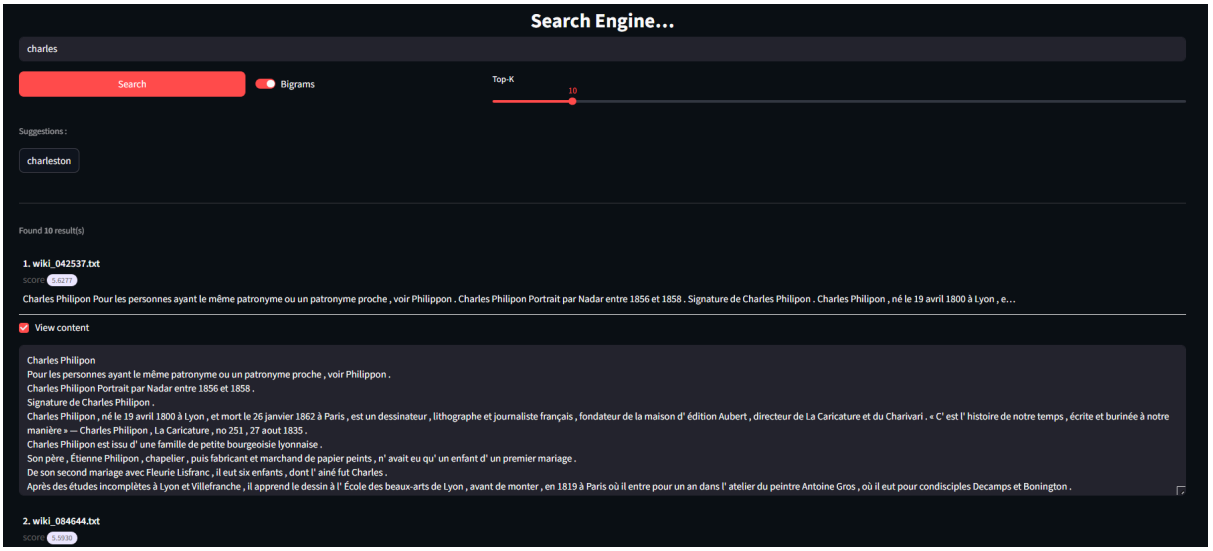


Barre de recherche

L'utilisateur saisit sa requête dans un champ dédié.
Deux options sont disponibles :

- **Bigrams** : active l'utilisation des bigrams dans la requête, comme pour l'indexation.
- **Top-K** : sélectionne le nombre de documents à retourner.

Un bouton *Search* permet de lancer explicitement la recherche. Lorsqu'on tape la touche *Entrée,* si l'utilisateur a tapé au moins 3 caractères, l'application affiche des suggestions basées sur l'index de préfixes. Celles-ci sont renouvelées lorsque l'utilisateur tape 3 caractères supplémentaires.



Résultats

Les résultats apparaissent sous forme de cartes contenant :

- l'ID du document,
- le score BM25,
- un extrait contextualisé,
- un bouton permettant d'afficher le contenu complet.

Une pagination permet de parcourir les résultats facilement.

Dashboard métriques

L'application propose un dashboard d'évaluation permettant d'évaluer et de comparer les différentes méthodes de recherche. Après avoir sélectionné un moteur et ses paramètres, l'utilisateur peut lancer l'évaluation sur l'ensemble des requêtes.

Le tableau de bord affiche :

- les scores moyens (P@k, R@k, Hit@1, MRR, nDCG@k),
- la distribution des rangs
- et un tableau détaillé par requête.

Une seconde rubrique permet de comparer directement les méthodes entre elles en affichant leurs scores côte à côte.

III. Méthodes de recherches

Dans cette partie nous présentons les différentes méthodes de recherche que nous avons implémentées et comparées dans notre moteur. Chaque approche utilise une façon spécifique de représenter les documents et de mesurer leur similarité avec une requête. Notre objectif a été d'observer leurs performances individuelles, puis d'étudier comment les combiner pour obtenir de meilleurs résultats.

1. BM25

BM25 est une méthode de recherche probabiliste très utilisée dans les moteurs modernes. Elle repose sur l'idée que plus un terme apparaît dans un document, plus ce document est susceptible d'être pertinent, tout en tenant compte de la longueur du document pour éviter de favoriser les textes trop longs.

Ce modèle introduit deux paramètres importants :

- k1 : contrôle l'importance de la fréquence des termes,
- b : permet de normaliser selon la longueur du document.

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

BM25 est souvent considéré comme une référence dans les systèmes de recherche traditionnels, car il combine de manière efficace fréquence des mots, rareté des termes et taille des documents.

2. TF-IDF Cosine

TF-IDF est une méthode classique qui représente les documents sous forme de vecteurs.

- **TF (Term Frequency)** : combien de fois un mot apparaît dans un document : plus il apparaît, plus il est important dans ce document.
- **IDF (Inverse Document Frequency)** : mesure la rareté d'un mot dans tout le corpus: un mot rare a plus de poids qu'un mot très fréquent.

Une fois les vecteurs créés, on utilise la cosine similarity pour mesurer la proximité entre la requête et chaque document. La similarité cosinus donne une valeur entre 0 et 1, ce qui permet de classer les documents selon leur pertinence.

Cette méthode est simple à mettre en place et offre des résultats souvent très corrects, surtout sur des corpus de taille moyenne.

3. Hybrid RRF (Reciprocal Rank Fusion)

Le RRF (Reciprocal Rank Fusion) est une méthode d'agrégation qui fusionne plusieurs classements indépendants. L'idée est de prendre en compte les rangs produits par différents modèles (ici on fusionne BM25 et TF-IDF) et de leur attribuer un score commun basé sur la formule :

$$\boxed{\text{score} = \frac{1}{k + \text{rang}}}$$

où *k* est un petit paramètre constant.

Cette approche est intéressante car elle ne dépend pas directement des scores bruts des modèles, mais uniquement de leur ordre de classement. RRF permet donc de combiner efficacement des méthodes très différentes et d'obtenir souvent un meilleur compromis entre précision et rappel.

4. Hybrid Interp

Enfin, l'approche Hybrid Interp consiste à combiner directement les scores produits par deux modèles différents, en utilisant une interpolation linéaire :

$$\boxed{\text{final\_score} = \alpha \cdot \text{score}_{\text{BM25}} + (1 - \alpha) \cdot \text{score}_{\text{TF-IDF}}}$$

Le paramètre α permet de contrôler le poids donné à chaque méthode. Cette technique est plus sensible que RRF car elle exploite les valeurs numériques des scores, ce qui permet d'ajuster précisément la contribution de chaque modèle.

IV. Analyse des résultats

Métriques utilisées

Dans le cadre de l'évaluation de notre moteur de recherche, il est essentiel de définir des métriques permettant de mesurer la qualité des différentes approches testées. Ces indicateurs servent à quantifier la pertinence des résultats retournés, à identifier les éventuelles limites de chaque méthode et, enfin, à comparer objectivement leurs performances afin de déterminer laquelle est la plus efficace dans notre contexte.

1. Précision : P@k

La métrique Precision@k mesure la proportion de documents pertinents présents parmi les *k* premiers résultats retournés par le moteur de recherche. Elle évalue donc la capacité du système à ne proposer que des résultats utiles en tête de classement.

$$\text{Precision} = \frac{\text{docs pertinents trouvés}}{\text{docs trouvés}}$$

Dans notre cas ou chaque requête ne possède qu'un seul document pertinent dans le fichier de teste, cette métrique devient :

- P@k = 1/k si le document pertinent apparaît dans les *k* premiers résultats
- P@k = 0 sinon

Precision@k met donc l'accent sur la capacité du moteur à maintenir un top-k "propre", c'est-à-dire composé de résultats réellement utiles.

2. Recall : R@k

Le Recall@k est une autre métrique qui répond à une question simple : est-ce que le moteur parvient à retrouver le document pertinent parmi les *k* premiers résultats ? Contrairement à la précision, le rappel ne cherche pas à savoir si les résultats non pertinents faussent le classement, mais simplement si le bon document a été retrouvé.

$$\text{Recall} = \frac{\text{docs pertinents trouvés}}{\text{tous les docs pertinents existants}}$$

Dans notre configuration, cette métrique devient binaire :

- Recall@k = 1 si le document apparaît dans les *k* premiers résultats
- Recall@k = 0 sinon

Le Recall@k mesure donc la capacité globale du système à retrouver correctement les documents attendus, sans s'intéresser à leur position exacte dans le classement.

3. Hit@k

Le Hit@k (également appelée Accuracy@k), mesure la proportion de requêtes pour lesquelles le document attendu apparaît parmi les k premiers résultats retournés par le moteur de recherche.

Dans notre cas, nous nous intéressons à la variante :

- Hit@1 : pourcentage de requêtes où le bon document est directement classé en première position.

Cette métrique permet donc d'évaluer la capacité du système à placer le bon document en tête du classement ou, au minimum, parmi les résultats jugés les plus pertinents.

4. Mean Reciprocal Rank : MRR

Le MRR évalue la qualité moyenne du rang du premier document pertinent. Pour chaque requête, on calcule le Reciprocal Rank (1 / rang du document pertinent), puis on moyenne ces valeurs.

$$\text{MRR} = \frac{1}{U}\sum_{u=1}^{U}\frac{1}{rank_i}$$

Un score de 1 signifie que la bonne réponse est toujours en première position. Un score de 0.5 correspond à une position moyenne autour de la deuxième place. Le MRR valorise donc fortement les documents placés en tout début de classement, et pénalise progressivement ceux qui apparaissent plus bas. Dans notre configuration, cette métrique donne une vision fine de la qualité du classement, au-delà d'un simple indicateur de présence dans le top-k.

5. nDCG@k

La métrique nDCG@k mesure la qualité du classement tout en tenant compte de la position exacte du document pertinent. Le score est maximal lorsque le document est classé en première position, et décroît lorsqu'il apparaît plus bas, selon une pénalisation logarithmique standard.

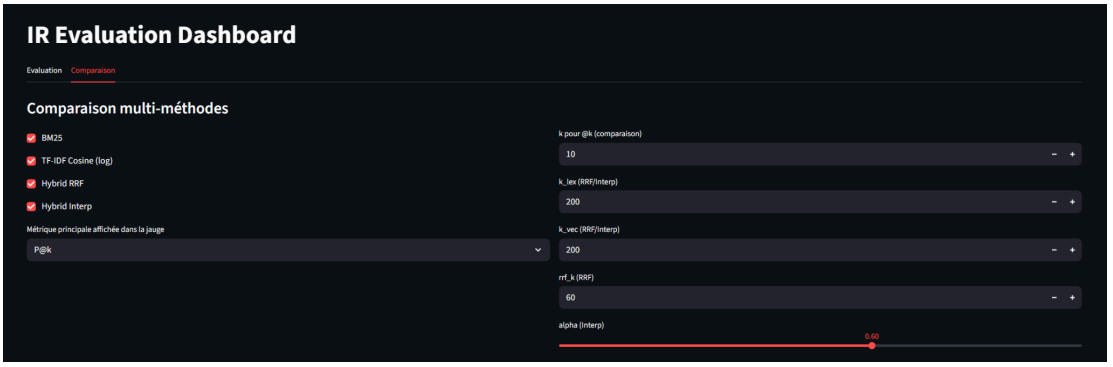
Dans notre cas, comme il n'y a qu'un seul document pertinent par requête, le nDCG@k sera:

- Si le document apparaît en première position, le score sera maximal (nDCG@k = 1)
- S'il est présent mais plus bas dans la liste, le score diminue progressivement.
- S'il n'est pas du tout dans les *k* premiers résultats, le nDCG@k vaut 0.

Cette métrique permet d'évaluer non seulement si le document attendu est retrouvé, mais aussi si le système est capable de le positionner de manière optimale dans les premières places.

Résultats obtenus

Dans cette partie nous allons comparer les différentes approches que nous avons vues plus haut. Dans le but de comprendre et d'extraire de l'information de ces différentes techniques de moteur de recherche, nous allons nous baser sur les métriques que nous avons définies.



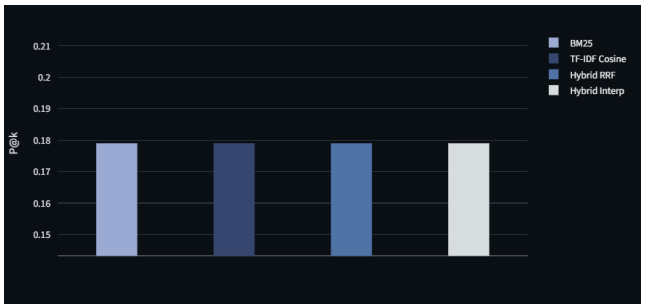
Pour la comparaison de nos différentes approches, nous avons fait en sorte de pouvoir modifier les paramètres constants de nos méthodes de recherches. Voici notre implémentation de base ci-dessus.

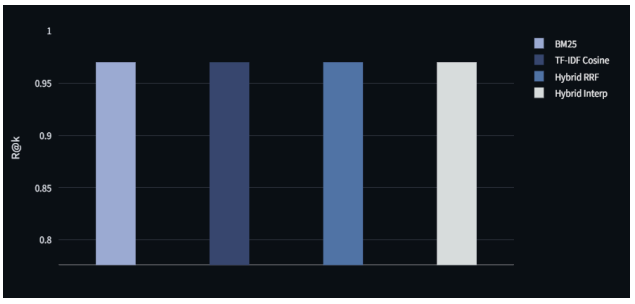
Voici le tableau correspondant aux comparaisons suivantes. Celui-ci sera détaillé par la suite a travers les différents graphes pour analyser chacune des métriques :

Méthode	P@k	R@k	Hit@1	MRR	nDCG@k	Latence (ms)
BM25	0.179	0.97	0.88	0.915	0.928	20
TF-IDF Cosine	0.179	0.97	0.92	0.937	0.945	31
Hybrid RRF	0.179	0.97	0.89	0.923	0.934	64
Hybrid Interp	0.179	0.97	0.92	0.938	0.946	70

Dans un premier temps nous allons nous concentrer sur la précision P@k, on observe sur le graphique ci-dessous de gauche que toutes nos approches ont le même résultat ce qui indique qu'il trouvent globalement le même nombre de documents pertinents dans le top-k.

On observe un schéma un peu similaire pour la métrique Recall (graphique de droite) qui montre encore une fois que chaque approche trouve tous les documents pertinents dans le top-k. Ceci reste cohérent avec le fait que nous n'avons qu'un seul document pertinent par requête.



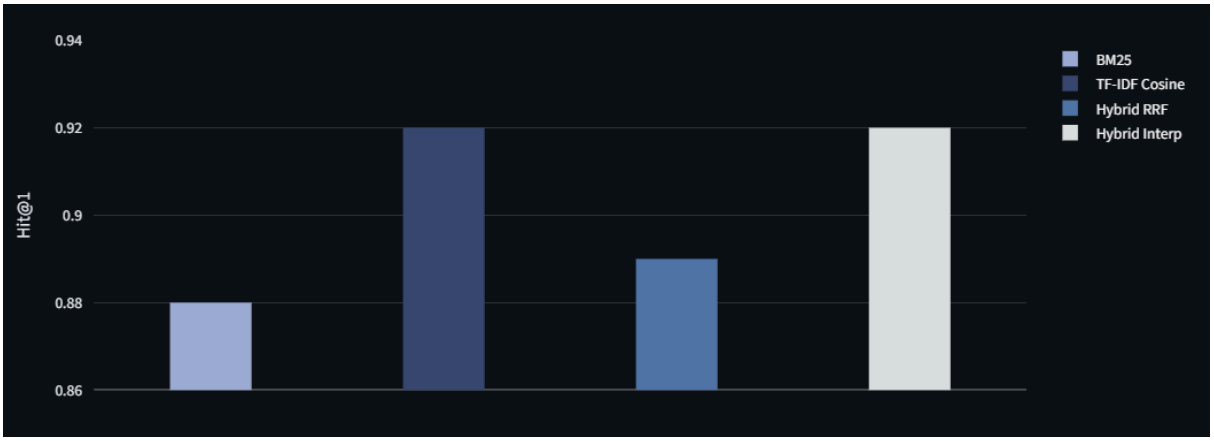


Cependant, on observe des différences plus marquées sur les métriques qui s'intéressent à l'ordre des résultats de nos recherches.

Pour la métrique Hit@1, on va regarder exclusivement le résultat qui arrive en première position. On comprend par exemple que l'approche BM25 obtient le moins bon résultat, cela peut s'expliquer par le fait qu'il atteint des limites sur le plan lexical notamment lorsqu'il rencontre une ambiguïté lexicale ou que le sens d'une requête est plus important que les mots employés.

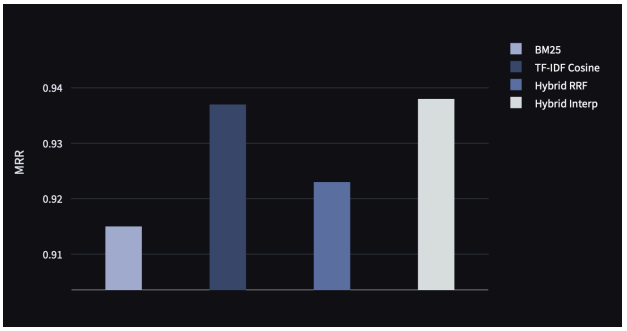
En l'occurrence TF-IDF cosine obtient le meilleur résultat avec un score de 0.92, donc dans 92% des cas le bon résultat est donné en première position. On comprend alors avec ce résultat que cette approche s'adapte mieux aux variations de vocabulaire.

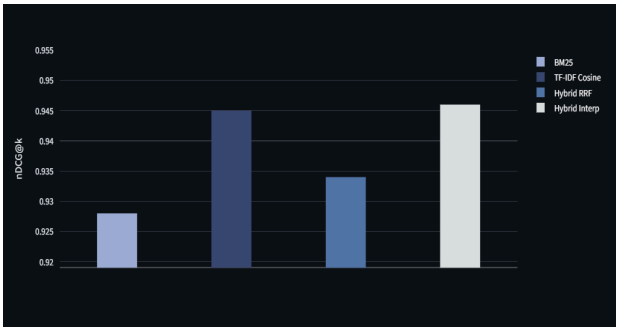
Ensuite, en ce qui concerne les approches hybrides, on obtient aussi un bon résultat pour l'interpolation qui se base sur les scores de nos deux approches BM25 et TF-IDF.



Enfin, les résultats du MMR et du nDCG@k, nous montrent que nos approches hybrides RFF et l'interpolation, offrent une meilleure qualité du classement de nos résultats. Ici Hybrid Interpolation obtient les meilleurs résultats avec en MMR (0.938) et en nDCG@k (0.946). Les résultats étant proches de 1 cela nous montre qu'il place souvent le document le plus pertinent très haut dans son classement et que l'organisation de son top-k est bien cohérente.

De plus, on voit que TF-IDF est une bonne approche lexical et qu'elle surpasse même BM25 sur les deux métriques. On voit aussi que hybrid RRF améliore les résultats lexicaux mais reste moins performant que son autre approche hybride.





En ce qui concerne la latence, BM25 est l'approche la plus rapide (20ms), TF-IDF s'exécute en 31ms. Les approches hybrides sont environ 2 fois plus lentes que TF-IDF en raison du calcul vectoriel et de la combinaison des méthodes.

VI. Conclusion

Pour finir on comprend de ces résultats que la plupart de nos approches parviennent à retrouver la majorité des résultats pertinents, mais que leur capacité à bien les classer varie un peu plus.

On relève que BM25 offre une bonne rapidité mais une moins bonne qualité de classement tandis que TF-IDF présente de meilleurs résultats. Ensuite, nos approches hybrides apportent en pertinence surtout celle de l'interpolation qui est la plus performante en exploitant un bon équilibre entre précision lexicale et compréhension sémantique. Cela nous

montre que combiner approche lexicale et vectorielle nous permet d'optimiser la qualité du moteur de recherche.

VII. Bibliographie

https://www.evidentlyai.com/ranking-metrics/evaluating-recommender-systems(46)
Evaluation Measures for Search and Recommender Systems - YouTube
https://www.alexmolas.com/2024/02/05/a-search-engine-in-80-lines.html
https://github.com/mdietrichstein/advanced-ir-search_engine/tree/master