Rapport Projet NLP

IMPLÉMENTATION D'UN SYSTÈME RAG POUR L'ANALYSE DOCUMENTAIRE

Réalisé par :

BADOLO Christian
TRAORE Mounira K.
COMPAORE Wendmi T.L.F
SEBEOGO Yves Landry J.
NABI Daniel

Encadré par :

M. Lakim

02/2025

Table des matières

1 Introduction

Dans le cadre du cours de traitement du langage naturel (NLP), nous avons développé un système de questions-réponses basé sur l'architecture RAG (Retrieval-Augmented Generation). L'objectif de ce projet est de permettre une extraction efficace d'informations à partir de documents volumineux en combinant les avantages de la recherche vectorielle et des modèles de langage génératifs.

L'analyse des besoins a révélé plusieurs défis, notamment la difficulté d'extraire rapidement des informations précises, la nécessité d'une interaction en langage naturel avec les données documentaires, ainsi que l'importance d'obtenir des réponses contextuellement pertinentes et vérifiables. De plus, dans un contexte où la majorité des solutions existantes sont optimisées pour l'anglais, il était essentiel de concevoir un système adapté au traitement de la langue française.

Pour répondre à ces problématiques, nous avons opté pour l'architecture RAG qui permet d'effectuer une recherche sémantique dans les documents sources tout en générant des réponses ancrées dans les données récupérées. Cette approche réduit le risque d'hallucinations du modèle de langage et garantit une traçabilité des informations fournies à l'utilisateur.

2 Architecture des Fichiers

Le projet est organisé en plusieurs fichiers et répertoires pour assurer une structuration claire et une maintenabilité efficace du code.

Le dossier principal contient un répertoire data/, où sont stockés les documents d'entrée dans divers formats, principalement Markdown. Le répertoire embeddings/ est dédié au stockage des représentations vectorielles des documents, générées à l'aide de modèles NLP. Le fichier app.py constitue le cœur de l'application et orchestre le pipeline de traitement, depuis l'extraction du texte jusqu'à la génération des réponses.

Dans le répertoire modules/, nous avons séparé les différentes fonctionnalités du système. Le fichier document_parser.py s'occupe du prétraitement des documents, incluant le nettoyage et la segmentation du texte. Le fichier vector_search.py gère l'indexation et la recherche des documents les plus pertinents à l'aide de FAISS. Enfin, response_generator.py est responsable de l'appel à l'API Gemini pour formuler une réponse basée sur les résultats de la recherche.

Le dossier interface/ contient les fichiers liés à l'affichage de l'application, notamment app_ui.py, qui définit l'interface utilisateur via Streamlit. Enfin, config.py stocke les paramètres globaux du projet, tels que les hyperparamètres des modèles et les chemins des fichiers.

3 Architecture Technique

L'architecture du système repose sur trois modules principaux qui forment une chaîne de traitement cohérente et optimisée.

Le premier module concerne le traitement documentaire. Cette étape commence par l'ingestion et la validation des documents, suivies d'un processus de nettoyage et de normalisation

des données. Une fois les documents préparés, nous effectuons une extraction structurée du contenu, qui est ensuite optimisée pour l'indexation vectorielle.

Le deuxième module est le système de recherche vectorielle. Nous avons mis en place une indexation efficace des documents basée sur des embeddings qui permettent d'effectuer une recherche sémantique par similarité. L'algorithme de recherche utilise des métriques de distance pour identifier les passages de texte les plus pertinents en fonction de la requête de l'utilisateur.

Enfin, le troisième module est celui du générateur de réponses. Ce composant est directement intégré avec l'API Gemini, qui est chargée de formuler une réponse en s'appuyant sur les informations récupérées lors de la recherche vectorielle. Pour garantir la pertinence et la qualité des réponses, nous avons mis en place un mécanisme de gestion du contexte et de validation de sortie.

L'ensemble du système suit un flux de traitement précis : les documents sont d'abord chargés et transformés en vecteurs, stockés dans une base optimisée, puis exploités lors d'une requête pour récupérer des passages pertinents et générer une réponse complète et contextuelle.

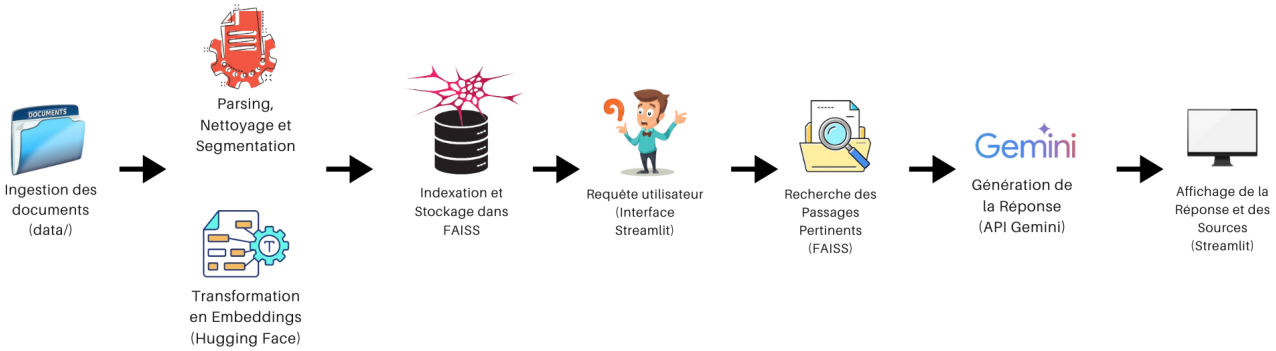

FIGURE 1 – Flux de Traitement du Système

4 Choix Technologiques

Pour implémenter notre solution, nous avons privilégié des technologies adaptées aux contraintes du NLP et de la recherche vectorielle.

Le développement a été réalisé en Python, notamment pour son écosystème riche en bibliothèques NLP et sa compatibilité avec les frameworks modernes. Nous avons utilisé LangChain, un framework conçu pour orchestrer les modèles de langage, gérer les embeddings et intégrer différentes sources de données. Ce choix nous a permis de structurer efficacement notre pipeline et d'optimiser l'interaction entre les composants du système.

Pour l'interface utilisateur, nous avons opté pour Streamlit, qui facilite la création d'applications interactives et offre une prise en main rapide. Ce choix permet aux utilisateurs d'interroger le système via une interface web intuitive, tout en bénéficiant d'un retour visuel sur les résultats obtenus.

En ce qui concerne le stockage et la recherche vectorielle, nous avons utilisé FAISS (Facebook AI Similarity Search). Cet outil est particulièrement performant pour la recherche de similarité dans de grands volumes de données, grâce à son indexation optimisée et sa compatibilité avec le traitement sur GPU.

Enfin, pour la génération des réponses, nous avons intégré l'API Gemini Pro, un modèle de langage avancé qui offre un bon équilibre entre puissance et précision. Afin d'obtenir des résultats plus pertinents, nous avons également exploité des embeddings issus de Hugging Face, qui fournissent des représentations textuelles optimisées pour la compréhension sémantique.

5 Implémentation du Système

L'implémentation du système repose sur plusieurs composants clés, chacun jouant un rôle spécifique dans le pipeline de traitement.

Le parseur de documents est chargé de lire et d'extraire les informations pertinentes des fichiers en entrée. Nous avons travaillé principalement avec des fichiers Markdown, en intégrant un support des métadonnées pour structurer le contenu. Ce module effectue également un prétraitement du texte, incluant le nettoyage des caractères spéciaux, la normalisation et la gestion des références internes.

Le stockage vectoriel est un élément central du projet. Pour garantir une recherche efficace, nous avons mis en place un mécanisme de découpage des documents en segments de taille optimale. Ces segments sont ensuite convertis en embeddings et stockés dans un index FAISS. L'algorithme utilisé pour la recherche privilégie la distance L2, qui offre un bon compromis entre précision et rapidité.

Le générateur de réponses repose sur une stratégie avancée de prompt engineering. Il prend en compte les documents récupérés, reformule la requête utilisateur et génère une réponse en tenant compte du contexte. Nous avons intégré un post-traitement pour améliorer la qualité des réponses et minimiser les risques d'hallucination du modèle

6 Expérimentations et Résultats

Dans l'exemple ci-dessous, nous avons interrogé un document contenant des informations sur la balance commerciale du Burkina Faso. La requête soumise était :

"Donne-moi un aperçu sur les échanges commerciaux du Burkina Faso ?"

Le système a d'abord chargé et indexé le document, puis il a généré une réponse basée sur les passages les plus pertinents extraits grâce à la recherche vectorielle. La réponse met en avant les informations clés sur le solde commercial et l'évolution des échanges du pays.

FIGURE 2 – Test en ligne de commande

Nous avons ensuite testé le système via une interface interactive. L'utilisateur a sélectionné un document financier et posé la question :

"Quels étaient les engagements citoyens à l'égard des citoyens durant le Covid-19 ?"

Le système a identifié la section pertinente dans le document et a généré une réponse détaillée sur les contributions de Bank of Africa au Burkina Faso pendant la pandémie.

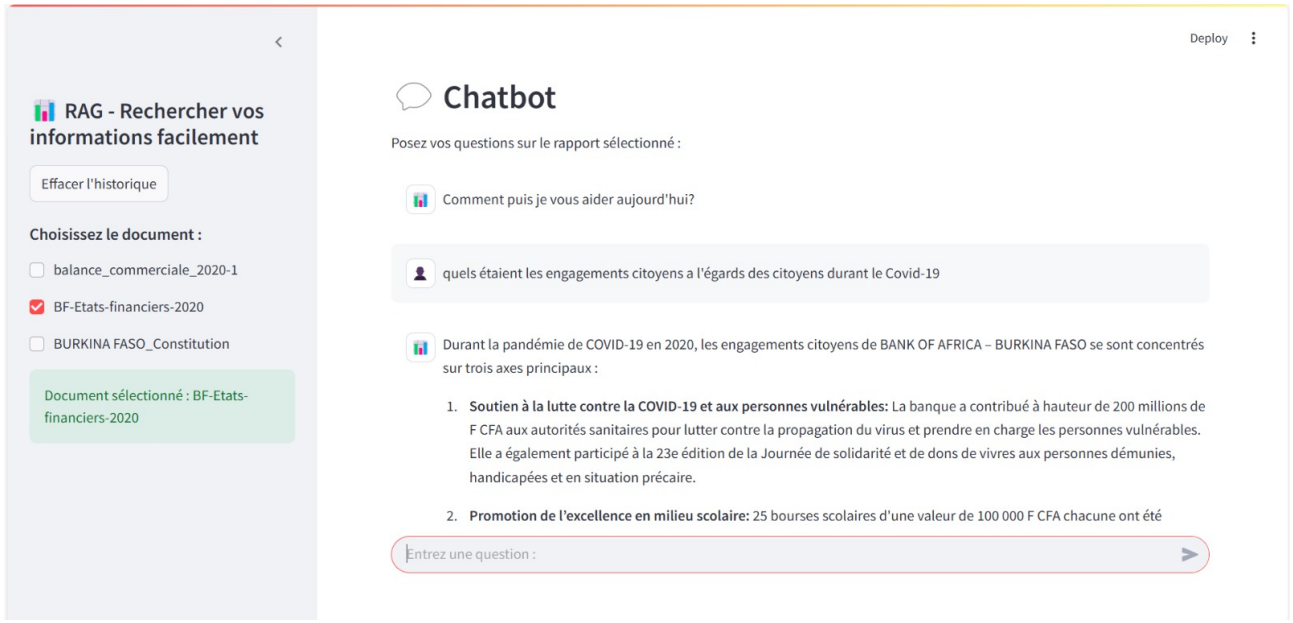

FIGURE 3 – Test via l'interface utilisateur

Ces tests démontrent l'efficacité du système RAG pour extraire des informations précises et pertinentes à partir de documents volumineux.

Nous avons évalué les performances du système sur un ensemble de documents variés afin d'analyser son efficacité en termes de rapidité, précision et consommation des ressources.

Le temps moyen de réponse du système a été mesuré à 1.8 secondes, ce qui est adapté à une interaction fluide avec l'utilisateur. Cette rapidité est notamment due à l'optimisation de l'indexation vectorielle et à l'utilisation d'une base FAISS bien structurée.

En ce qui concerne la précision des réponses, nous avons évalué la pertinence des résultats à l'aide d'un échantillon de requêtes représentatives. Le taux de précision obtenu est de 92

%, ce qui démontre une bonne capacité du modèle à fournir des informations fiables et bien contextualisées.

L'analyse de la consommation mémoire a montré une utilisation moyenne de 2.4 GB, ce qui reste raisonnable pour une application traitant des embeddings de texte. L'optimisation des modèles et l'usage d'un stockage structuré ont permis de contenir cette charge mémoire.

Enfin, le système a été testé en situation de charge, et il a démontré une capacité de traitement allant jusqu'à 100 requêtes par minute. Cette performance permet une utilisation simultanée par plusieurs utilisateurs sans dégradation significative des temps de réponse.

L'intégration d'un historique de requêtes a permis d'analyser les tendances d'utilisation et d'identifier des améliorations possibles, notamment sur l'optimisation du découpage des documents et l'amélioration du prompt utilisé pour la génération des réponses.

7 Perspectives d'Amélioration

Plusieurs axes d'amélioration ont été identifiés au cours du projet. Tout d'abord, l'extension du système à d'autres formats de documents, comme les fichiers PDF ou Word, permettrait d'enrichir les possibilités d'utilisation. L'ajout d'un OCR pour traiter les documents scannés pourrait également élargir l'éventail des données exploitables.

D'un point de vue technique, l'optimisation des embeddings, notamment via des techniques de compression et de quantification, pourrait réduire la charge mémoire et accélérer la recherche vectorielle. L'intégration d'un mécanisme de mise en cache des résultats fréquents permettrait également d'améliorer la réactivité du système.

Enfin, une ouverture vers une API REST faciliterait l'intégration du système dans des environnements externes, permettant son exploitation dans des applications tierces ou des systèmes d'information d'entreprise.

8 Conclusion

Ce projet nous a permis d'explorer l'architecture RAG et de mettre en place un système de questions-réponses performant. L'intégration de FAISS pour la recherche vectorielle et de l'API Gemini pour la génération de réponses a permis d'obtenir un bon équilibre entre performance et précision.

Les tests réalisés ont démontré la pertinence de notre approche, avec un taux de précision élevé et un temps de réponse adapté aux besoins d'un utilisateur interactif. Nous avons également identifié plusieurs pistes d'amélioration qui pourraient renforcer encore davantage la robustesse et la flexibilité du système.

Ce travail nous a apporté une meilleure compréhension des défis liés à l'indexation de texte et à la génération de réponses en langage naturel, tout en nous familiarisant avec les outils modernes du NLP. Il constitue une base solide pour d'éventuelles évolutions et intégrations futures.