Université de Technologie de Compiègne

Département Informatique

LO17

Indexation et Recherche d'Information

Guide Touristique

RAG — Retrieval Augmented Generation

Rapport de Projet

Samuel Beziat	Contribution : 30%
Emma Choukroun	Contribution : 30%
Jeanne Galet	Contribution : 20%
Selima Khessairi	Contribution : 20%

Ce rapport retrace les grandes étapes du projet, de la conception à la réalisation, en mettant en lumière les défis rencontrés ainsi que les solutions développées. Il offre un aperçu global du travail effectué dans le cadre du cours de LO17 (Indexation et Recherche d'Information).

Groupe LO17, TD2
Printemps 2025

Sommaire

Partie 1
Introduction

Contexte pédagogique

Ce projet entre dans le cadre du cours LO17 – *Indexation et Recherche d'Information*. En début de semestre, nous avons travaillé sur les bases de l'indexation, la structuration de l'information, et les techniques de recherche dans des bases textuelles. Notre premier projet consistait à construire un moteur de recherche capable d'interroger la base de données de l'ADIT, en s'appuyant sur des champs bien définis et des requêtes précises. Cette étape nous a permis de mieux comprendre comment organiser un corpus, structurer des métadonnées et concevoir un système de recherche efficace.

Réflexion collective et choix d'un sujet à impact

Lorsque nous avons abordé le second projet, une exigence s'est rapidement imposée à nous : choisir un sujet qui ait du sens, au-delà du cadre académique. Dès la première séance, notre groupe de quatre personnes a consacré du temps à des échanges approfondis pour déterminer une thématique engageante. Nous voulions éviter un projet artificiel ou purement démonstratif, et privilégier une idée qui nous touche personnellement et qui présente un véritable intérêt pour les utilisateurs finaux.

C'est dans ce contexte que nous avons convergé vers l'idée de concevoir un **guide touristique intelligent**, capable de répondre à des questions pratiques en langage naturel.

Motivations personnelles et pertinence du thème

Le choix du tourisme ne s'est pas fait au hasard. Tous les membres du groupe partagent un goût prononcé pour les voyages, les découvertes culturelles et les escapades improvisées. Nous avons tous été confrontés aux mêmes difficultés lors de la planification de courts séjours ou de vacances de dernière minute : les informations sont dispersées entre des blogs, des forums, des guides PDF, des plateformes commerciales... Aucun outil unique ne permet de centraliser les réponses à des questions pourtant simples : *"Que faire à Porto en trois jours ?", "Quel quartier privilégier à Barcelone pour un séjour tranquille et économique ?", "Où voyager en avril pour éviter la pluie et la foule ?"*. Ce constat d'expérience personnelle a renforcé notre volonté de proposer une solution utile et concrète.

Objectifs du projet

Notre objectif est donc de mettre en place une application basée sur le principe du *RAG*, qui combine recherche documentaire et génération de texte. Concrètement, notre système :

— indexe un corpus de données touristiques dans une base vectorielle (ChromaDB),
— interroge ces données de manière sémantique en fonction des questions posées,
— et génère des réponses contextualisées avec un modèle de langage (Gemini),
— le tout via une interface conviviale développée avec Streamlit.

Ce projet est à la fois une mise en application des concepts étudiés en cours, et une tentative de proposer un outil vraiment utile, pensé pour répondre à un besoin concret.

Partie 2
Extraction et indexation des données

2.1 Objectif

Cette étape vise à constituer une base documentaire structurée et interrogeable pour notre système RAG. Après avoir identifié des sources web fiables, nous avons extrait les URLs pertinentes, téléchargé le contenu des pages associées, et vectorisé l'ensemble dans une base de données persistante.

2.2 Choix et traitement des sources

Nous avons retenu trois sites aux profils complémentaires : **Wikivoyage**, pour son contenu encyclopédique collaboratif, structuré de manière homogène ; **Le Routard**, qui fournit des conseils pratiques et des retours d'expérience de terrain et **le Ministère des Affaires étrangères**, qui propose des fiches pays officielles, utiles à l'évaluation de la sécurité et des formalités de voyage.

Pour Wikivoyage, nous avons exploité un *dump* XML en français (daté du 20 mai 2025), analysé avec la bibliothèque `mwxml`. Les titres d'articles valides ont été convertis en URLs exploitables. Le Routard a nécessité un crawling dynamique en largeur, en filtrant les URLs pertinentes. Enfin, les fiches pays du site `diplomatie.gouv.fr` ont été extraites via une analyse ciblée du HTML de la page "Conseils par pays".

Nous avons extrait au total :

— 11 554 URLs depuis Wikivoyage
— 28 032 depuis le site du Routard
— 192 depuis Diplomatie.gouv.fr

Chaque jeu d'URLs a été sauvegardé dans un fichier CSV distinct.

2.3 Chargement des documents

Les URLs collectées ont été traitées à l'aide du composant `WebBaseLoader` de LangChain, qui transforme chaque page en un objet `Document`, enrichi d'une métadonnée indiquant sa source. Pour accélérer le processus, nous avons parallélisé le chargement via des threads.

L'ensemble des documents a été fusionné dans un fichier unique `docs_rag.pkl`, totalisant environ 39 000 pages web nettoyées et prêtes à être vectorisées.

2.4 Vectorisation et indexation

La génération des embeddings a été réalisée à l'aide du modèle `models/embedding-001` proposé par Google (Gemini). Avant vectorisation, les documents vides ou trop bruités ont été filtrés.

Les embeddings valides sont insérés dans une base vectorielle persistée via `ChromaDB`, utilisée ensuite pour le RAG. Le processus est entièrement automatisé tout en restant résilient aux interruptions.

Ainsi, malgré les obstacles liés à la volumétrie, aux quotas et à la qualité hétérogène des contenus, cette phase a permis de construire une base documentaire solide. Avec près de 39 000 documents nettoyés, indexés et vectorisés, nous disposons désormais d'un socle riche pour la recherche augmentée par récupération.

2.5 Contraintes techniques et gestion des quotas

Cette phase de collecte, de traitement et d'indexation a été particulièrement exigeante en raison des limitations techniques imposées par le nombre de sources à traiter et par l'utilisation de l'API gratuite de Gemini.

Ainsi, comme mentionné plus haut, la parallélisation via multi-threading s'est révélée nécessaire pour traiter tous les documents en un temps raisonnable. De plus, pour l'indexation et pour garantir la résilience aux interruptions liées aux quotas, un système de traitement par batchs de 10 documents a été mis en place. Entre le traitement de chaque batch, une pause automatique de **1 seconde** a été insérée pour ralentir volontairement le rythme des requêtes et limiter la pression sur les serveurs. Enfin un système de checkpoint (`checkpoint.json`) enregistre la progression après chaque batch traité avec succès. Cela permet une reprise automatique sans devoir recharger les documents précédemment indexés.

2.6 Structure du code source et documentation

Le code source du projet est organisé de manière lisible et bien structurée, et son usage est documenté dans un fichier `README.md` détaillé, disponible à la racine du dépôt Git. Celui-ci permet de nous guider à travers toutes les étapes du pipeline : extraction, chargement, vectorisation et indexation. L'extraction des URLs est assurée par trois scripts placés dans le dossier `/urls_extraction` : `wikivoyage.py` exploite un dump XML de Wikivoyage pour générer les liens valides ; `routard.py` réalise un crawling contrôlé du site du Routard ; enfin, `diplomatie_gouv.py` cible les fiches pays du site du Ministère des Affaires étrangères. Chaque script produit un fichier CSV distinct contenant les URLs extraites. Le chargement des documents web s'effectue via `documents.py` (répertoire `/index`), qui utilise `WebBaseLoader` de LangChain et le parallélisme par threads pour accélérer le traitement. Les documents sont sauvegardés au format `docs_rag.pkl` pour usage ultérieur. La vectorisation est réalisée par le script `chroma_index.py`, qui emploie les embeddings de Google Gemini et insère les vecteurs dans une base persistante Chroma. Un système de checkpoint (`checkpoint.json`) permet de relancer proprement le processus après interruption, et des pauses entre batchs assurent la stabilité des requêtes. Enfin, la qualité du code a été évaluée avec `pylint`, et nous avons obtenu un score très satisfaisant (voir ci-dessous), garantissant une base de code propre, maintenable et conforme aux bonnes pratiques Python.

```
Your code has been rated at 10.00/10 (previous run: 6.02/10, +3.98)
```

Partie 3
Évaluation du RAG

L'évaluation effectuée (liée au fichier `RAG_LO17_evaluation.ipynb`) se base sur la bibliothèque **RAGAS** (Retrieval-Augmented Generation Assessment Scores) de Python. Cette dernière requiert un formatage précis du jeu de données, structuré selon les champs suivants :

— **user_input** : la question sur laquelle on évalue le modèle,
— **response** : la réponse générée par le système RAG,
— **retrieved_contexts** : le contenu des sources retournées par le RAG,
— **reference** : la réponse de référence, considérée comme oracle.

Nous avons ainsi sélectionné six questions pour constituer notre jeu d'évaluation :

— Quels vaccins sont recommandés pour un voyage en Thaïlande ?
— Propose-moi un itinéraire à Lisbonne.
— Que faire à Tokyo ?
— Quels sont les plats typiques de Naples ?
— Quel est le climat en Albanie ?
— Les plus beaux quartiers de Buenos Aires.

Les métriques choisies pour l'évaluation sont :

— **Faithfulness** : mesure la fidélité de la réponse par rapport au contexte fourni (détection d'hallucinations),
— **Groundedness** : évalue l'ancrage de la réponse dans le contexte récupéré,
— **Answer Accuracy** : mesure la pertinence de la réponse générée par rapport à la réponse de référence,
— **Context Precision** et **Context Recall** : évaluent respectivement la précision et le rappel de la réponse par rapport au contexte.

Les résultats obtenus sont présentés dans la figure ci-dessous :

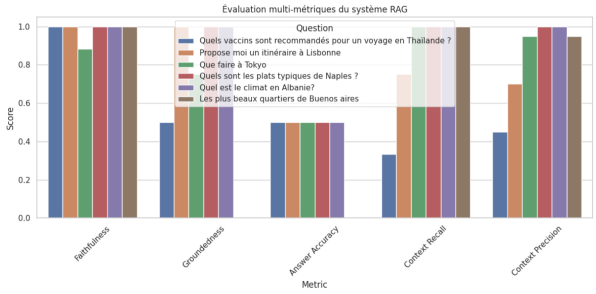

FIGURE 3.1 – Résultats des métriques d'évaluation sur les six questions

Nous remarquons une très bonne fidélité des réponses par rapport au contexte (*Faithfulness*). Pour la *Groundedness*, certains scores atteignent 1, tandis que d'autres sont légèrement plus bas. Cela peut s'expliquer par la diversité des sources (trois sites différents), dont les formulations variées peuvent impacter les scores.

La pertinence des réponses générées par rapport aux réponses de référence (*Answer Accuracy*) est moyenne, en raison de la richesse de la base documentaire et du caractère parfois incomplet des réponses de référence définies manuellement.

Enfin, les mesures de *Context Precision* et *Context Recall* sont globalement satisfaisantes, bien que quelques questions présentent des scores plus faibles à cause du bruit présent dans certains documents récupérés.

Partie 4
Interface – Application avec Streamlit

4.1 Présentation générale

Dans la continuité du développement de notre système RAG, nous avons conçu une interface simple et intuitive à l'aide de la bibliothèque Streamlit. Notre objectif était de permettre à un utilisateur non technique d'interagir naturellement avec notre assistant, en posant des questions en langage courant sur des destinations touristiques.

4.2 Pourquoi "Genova" ?

Nous avons choisi d'appeler notre assistant "**Genova**", un nom qui a plusieurs significations pour nous. D'une part, il évoque le mot *generative*, en référence directe au rôle central du modèle de langage dans notre projet. D'autre part, "Genova" est aussi le nom italien de la ville de Gênes, un port tourné vers le monde, historiquement lié aux échanges, aux découvertes et aux voyages. Ce double sens nous a semblé pertinent : notre application repose sur la génération d'information, mais elle est aussi une porte d'entrée vers la découverte d'autres lieux.

Ce nom incarne donc à la fois l'aspect technique et l'esprit de voyage de notre projet.

4.3 Conception de l'interface

Nous avons souhaité proposer une interface inspirée des systèmes RAG modernes que nous avons eu l'occasion de tester ou d'étudier. L'utilisateur interagit avec Genova via une **zone de chat** dans laquelle il peut poser ses questions librement, comme il le ferait avec un assistant personnel.

À gauche de l'écran, une **barre latérale** permet d'accéder à deux fonctionnalités principales :

— consulter l'**historique des conversations** précédentes,

— créer un **nouveau chat** à tout moment.

Cette structure permet de garder une trace des échanges, tout en favorisant la reprise de conversations passées ou la création de requêtes indépendantes.

Pour guider les premiers utilisateurs, nous avons également intégré une série d'**exemples de questions**, affichés au démarrage de l'application. Par exemple :

— "Que faire à Lisbonne en 3 jours ?"

— "quelle est la situation sanitaire au Bangladesh ?"

— "Quels quartiers visiter à Tokyo pour un séjour culturel ?"

Ces suggestions permettent de découvrir rapidement les capacités de l'outil et d'inciter l'utilisateur à explorer différentes requêtes.

4.4 Ergonomie et simplicité

L'ensemble a été pensé pour offrir une expérience fluide, sans surcharge d'options techniques. En limitant les éléments visibles à l'essentiel (zone de dialogue, historique, et exemples), nous favorisons une prise en main immédiate et naturelle. L'utilisateur peut ainsi se concentrer sur ses questions, sans avoir à comprendre le fonctionnement interne du système.

Partie 5
Bonus : gestion des hallucinations + déploiement de l'application

5.1 Améliorations du RAG

5.1.1 1. Limites d'un RAG standard

Un système RAG (Retrieval-Augmented Generation) présente certaines faiblesses que nous avons cherché à corriger :

- **Pertinence** : le moteur de recherche peut retourner des documents éloignés de la question.
- **Hallucination** : le modèle génératif peut inventer des réponses sans appui sur les documents.
- **Absence d'historique** : chaque question est interprétée isolément, ce qui nuit aux interactions suivies.
- **Réponses non adaptées** : les formats peuvent manquer de clarté, de concision, ou de structure.

5.1.2 2. Améliorations implémentées

Prompt Engineering renforcé

Le prompt est le principal levier pour guider le comportement du LLM. Nous avons conçu un **prompt structuré et contraignant** pour :

- définir un rôle explicite : *guide touristique expert*,
- interdire les suggestions ou généralisations hors contexte,
- exiger une formulation claire et directe,
- forcer la citation d'URLs exactes issues des documents.

Extrait du prompt utilisé :

> Tu es un guide touristique expert. Ta tâche est de répondre à la question de l'utilisateur en t'appuyant uniquement sur les informations fournies dans le contexte. [...] Si l'information n'est pas disponible dans le contexte, réponds : "Je ne sais pas." [...] Cite les URLs exactes, une seule fois.

Transparence et traçabilité

Chaque réponse générée est accompagnée de ses **sources explicites**. Cela permet à l'utilisateur de :

- retracer l'origine de l'information,
- identifier les documents utiles,
- repérer d'éventuelles erreurs de génération.

Mémoire conversationnelle (Historique)

Nous avons intégré un module de mémoire (`ConversationBufferMemory`) permettant au système de :

- conserver les échanges précédents,
- contextualiser les requêtes successives,
- maintenir la cohérence d'un dialogue long.

Détection des hallucinations

En complément, un système d'analyse a été mis en place pour repérer automatiquement des réponses potentiellement inventées. Pour cela, nous avons défini une liste de mots-clés indicateurs d'incertitude ou d'invention (*"je pense que", "il est probable que", "selon moi", etc.*) et vérifions leur présence dans les réponses.

5.2 Déploiement de l'application

Le déploiement de notre application GENOVA a nécessité une organisation adaptée aux contraintes de poids de la base vectorielle et à l'environnement d'hébergement.

5.2.1 Structure du dépôt

Le script principal `app.py` et le fichier `requirements.txt` sont regroupés dans un dépôt GitHub. L'interface repose sur Streamlit, ce qui permet un déploiement rapide et simple à partir du code source.

5.2.2 Gestion de la base vectorielle

La base `chroma_db`, issue de la phase d'indexation, pèse plus de 2 Go, ce qui excède les limites de GitHub et de Streamlit Cloud. Pour contourner cette contrainte, nous l'avons compressée dans une archive `.zip` hébergée sur Google Drive via un lien public. Lors de l'exécution de l'application, cette archive est automatiquement téléchargée puis extraite localement dans l'environnement Streamlit. Ce procédé permet de charger la base vectorielle sans l'inclure dans le dépôt.

5.2.3 Hébergement sur Streamlit Cloud

Le déploiement s'effectue via la plateforme Streamlit Cloud, en connectant directement le dépôt GitHub. À l'exécution, Streamlit installe les dépendances listées dans `requirements.txt`, télécharge et décompresse la base vectorielle, puis lance le script `app.py`. Ce flux automatisé permet une mise en ligne rapide de l'outil.

L'application finale est accessible à l'adresse suivante :

`https://genova-rag.streamlit.app/`

Partie 6
Conclusion

6.1 Résumé du travail

Dans le cadre de ce projet, nous avons construit une application nommée **Genova**, un assistant touristique intelligent basé sur une architecture *RAG* (Retrieval-Augmented Generation). Le système s'appuie sur des documents collectés depuis des sources fiables (*Le Routard*, *Wikivoyage*, *Diplomatie.gouv.fr*, *Lonely Planet*) et exploite le modèle *Gemini 1.5 Flash* via l'API Google pour générer des réponses contextualisées. Notre application permet à un utilisateur de poser librement des questions en langage naturel et d'obtenir des réponses claires, sourcées, et centrées sur les documents disponibles. Le tout est accessible via une interface ergonomique développée avec *Streamlit*.

6.2 Bilan

Nous avons travaillé à quatre sur ce projet. Si le début a été compliqué (principalement à cause des quotas limités de l'API Gemini qui ralentissaient notre progression), on a fini par trouver un bon rythme. Nous ne sommes pas partis de zéro, puisque nous avons utilisé l'infrastructure LangChain et l'API Gemini de Google. Cela dit, l'essentiel du travail restait à faire : nous avons collecté et structuré nous-mêmes les données, préparé les embeddings, géré les relances par lot en cas de quota, conçu des prompts précis et mis en place une évaluation rigoureuse des performances.

Ce n'était pas toujours simple, mais le projet nous a permis de voir très concrètement ce qu'implique la mise en place d'un assistant intelligent basé sur des documents réels.

6.3 Contributions

Le projet a été mené collectivement : chacun a contribué activement à toutes les étapes, de l'extraction des données à l'évaluation du système. Travailler à quatre nous a permis d'avancer plus vite et plus loin. On s'est souvent aidés mutuellement pour corriger des erreurs, ajuster le plan ou tester de nouvelles idées. Cette collaboration a été un vrai point fort du projet.

6.4 Perspectives

Notre système RAG remplit bien sa fonction de réponse à des questions touristiques pratiques, mais il présente aussi certaines limites. L'une des principales est sa dépendance stricte aux documents récupérés : il restitue les informations présentes, mais peine à les combiner intelligemment. Par exemple, lorsqu'on lui demande un itinéraire ou un programme structuré, il se contente souvent de lister des lieux, sans les articuler en étapes cohérentes de voyage. Il n'y a pas de réelle planification ou synthèse, mais plutôt une réponse fragmentée, fidèle aux sources mais peu proactive.

Avec plus de temps, nous aurions aimé héberger un modèle open-source localement pour gagner en autonomie, intégrer une vérification automatique des faits (*fact-checking*), proposer une version multilingue ou une interaction vocale, enrichir l'interface avec du retour utilisateur pour améliorer les réponses au fil des usages.

Malgré ces pistes non explorées, le projet constitue une base solide et exploitable. Il nous a permis de mettre en œuvre toutes les étapes d'un pipeline IA moderne : collecte, nettoyage, indexation, génération, évaluation et mise en ligne.