RI et graphe de données - TP RAG

Lors du TP précédent, vous avez mis en place l'ensemble des briques nécessaires à la construction d'une base de données vectorielle. Dans ce nouveau TP, nous allons prolonger ce travail en construisant un système complet de type RAG (Retrieval-Augmented Generation). L'idée est de combiner la recherche documentaire que vous avez développée avec un modèle de langage (LLM), afin de produire des réponses à partir de documents.

Concrètement, au lieu de demander directement au modèle de répondre à une question, on commence par rechercher dans une base documentaire les passages les plus pertinents. Ces passages sont ensuite fournis au modèle, qui s'appuie dessus pour formuler sa réponse.

Le déroulement est donc le suivant : un utilisateur pose une question, le système recherche des extraits pertinents dans la base vectorielle, construit un contexte textuel à partir de ces extraits, puis envoie ce contexte à un modèle de langage qui génère la réponse.

Compléments de cours

Pour interagir avec un modèle de langage, nous allons utiliser la bibliothèque `langchain-openai`. Cette bibliothèque fournit des abstractions simples pour envoyer des requêtes à un modèle de type « chat ».

Avant de commencer, vous devez ajouter cette dépendance à votre projet `uv`.

La classe `ChatOpenAI` permet d'instancier un modèle distant. Elle prend notamment en paramètre le nom du modèle, l'URL du service et une clé d'API. Une fois le modèle créé, il peut être utilisé pour générer des réponses.

Les échanges avec le modèle se font sous forme de messages. Un `HumanMessage` correspond à une entrée utilisateur, tandis qu'un `SystemMessage` permet de définir le comportement attendu du modèle. Ce dernier joue un rôle important, car il permet de contraindre la manière dont le modèle doit répondre.

La génération d'une réponse se fait en appelant la méthode `invoke`, à laquelle on fournit une liste de messages. Le modèle retourne alors un objet contenant la réponse générée.

1 Description de l'application

L'application du précédent TP va être enrichie avec une nouvelle commande en ligne de commande :

```
chatbot <modele> <url_llm> <cle_llm>
```

Cette commande permet de lancer un chatbot interactif. L'utilisateur peut saisir des questions, et le programme répond en s'appuyant sur les documents indexés dans la base vectorielle.

Les trois paramètres correspondent respectivement au nom du modèle à utiliser, à l'URL du service de génération, et à la clé d'API permettant d'y accéder.

Un nouveau module `rag.py` est ajouté au projet. Il contient les fonctions :

— `recuperer_contexte` qui sera responsable de la recherche des documents pertinents dans la base vectorielle. Elle devra réutiliser les fonctions développées lors du TP précédent.
— `construire_contexte` qui transformera les résultats de recherche en une chaîne de caractères structurée, qui sera injectée dans le prompt du modèle.
— `construire_messages` qui permettra de construire les messages envoyés au modèle, en séparant clairement les instructions (message système) et la requête utilisateur.
— `generer_reponse` qui orchestrera l'ensemble du processus : recherche, construction du contexte, appel au modèle et récupération de la réponse.
— `afficher_sources` permettra d'afficher les documents utilisés pour produire la réponse, afin de rendre le système plus transparent.

1.1 Travail à réaliser

Vous devez implémenter les différentes fonctions du module `rag`.

La fonction `recuperer_contexte` doit s'appuyer sur la base vectorielle pour retrouver les passages les plus pertinents en fonction d'une question donnée.

La fonction `construire_contexte` doit transformer ces résultats en un texte exploitable par le modèle. Il est important que ce texte soit bien structuré. Une bonne pratique consiste à numéroter les sources et à indiquer pour chacune le document d'origine, l'identifiant du chunk, le score de similarité et le contenu textuel. Cette structuration facilite à la fois la compréhension du modèle et l'analyse des réponses.

La fonction `construire_messages` doit créer les messages envoyés au modèle. Le message système doit définir clairement le comportement attendu : répondre en s'appuyant sur le contexte fourni, ne pas inventer d'informations, et signaler lorsque la réponse ne peut pas être trouvée dans les documents. Le message utilisateur doit contenir à la fois la question posée et le contexte construit précédemment, ainsi qu'une indication sur la langue ou le style de la réponse.

La fonction `generer_reponse` doit enchaîner les différentes étapes du pipeline RAG. Elle appelle la fonction de récupération, construit le contexte, prépare les messages, puis interroge le modèle à l'aide de la méthode `invoke`. Elle doit retourner à la fois la réponse générée et les éléments de contexte utilisés.

La fonction `afficher_sources` doit présenter de manière lisible les documents ayant servi à produire la réponse. On attend au minimum l'affichage du nom du document, de l'identifiant du chunk et du score de similarité.

Enfin, vous devez intégrer cette nouvelle fonctionnalité dans le programme principal en ajoutant la commande `chatbot` au parseur et en appelant la fonction correspondante.