Semantic Spotter Project

Problem statement

In today's digital age, food enthusiasts and home cooks are increasingly turning to online platforms for culinary inspiration, recipe discovery, and cooking tips. However, current solutions often lack the ability to provide personalized, contextually accurate responses that cater to specific user queries. Users face challenges in navigating large recipe datasets to find the exact information they need, such as adapting recipes to dietary restrictions, suggesting ingredient substitutions, or providing step-by-step cooking guidance.

This project addresses the need for a more interactive, intelligent system capable of answering culinary-related questions with precision and relevance. By leveraging a Retrieval-Augmented Generation (RAG) system, the goal is to bridge the gap between users' questions and comprehensive recipe data. The solution will integrate advanced natural language processing (NLP) techniques and efficient retrieval mechanisms, offering users an engaging and informative experience that enhances their culinary journey.

The key challenge lies in building a system that not only retrieves relevant data from a vast recipe dataset but also generates meaningful, context-aware answers, tailored to each user’s unique query.

Project goal

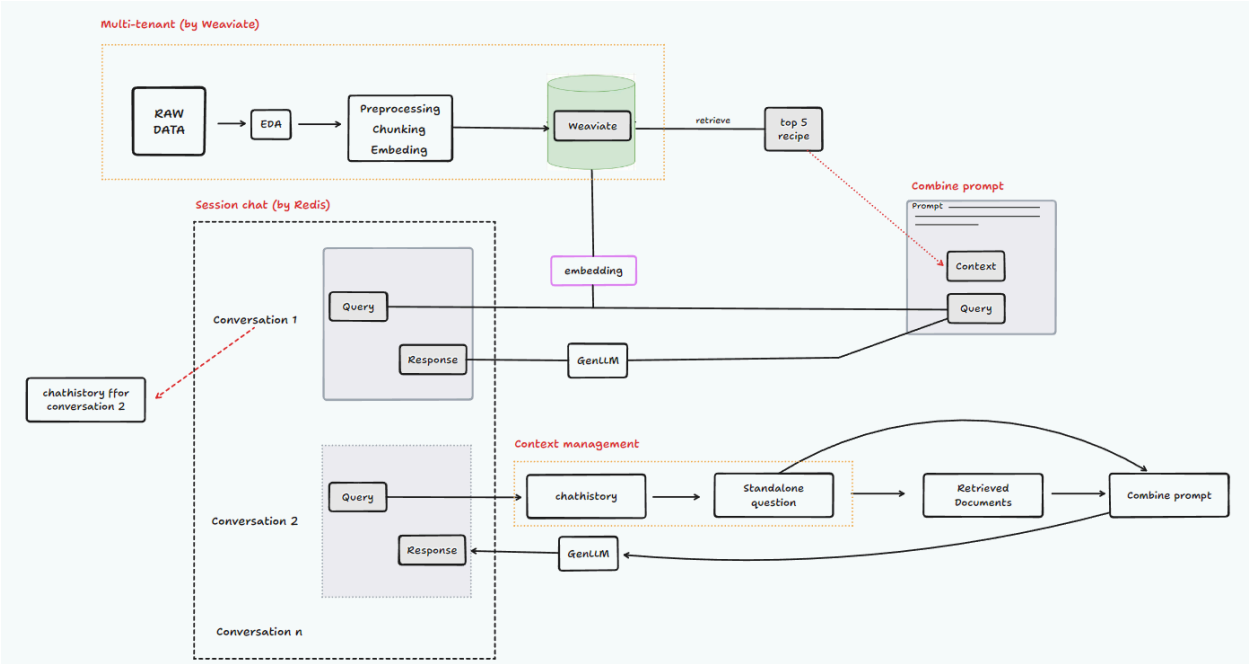
This project aims to develop a Retrieval-Augmented Generation (RAG) system that answers culinary-related questions by leveraging a comprehensive recipe dataset. The system will combine the power of natural language processing (NLP) and retrieval mechanisms based on Langchain to provide accurate, contextually relevant, and detailed answers. By efficiently retrieving relevant recipe information and generating responses, the system aims to assist users in discovering new culinary insights, enhancing their cooking experiences, and deepening their engagement with the platform.
For more details, you can access my notebook.

Data source

I use the Recipe dataset from Kaggle.
The project utilizes a recipe dataset sourced from Kaggle, specifically the recipes.csv file, which contains information on over 1,000 recipes. The dataset includes fields such as recipe names, preparation times, cooking times, ratings, ingredients, and step-by-step instructions. For the purpose of this project, I pre-processed and used 100 recipes. The relevant fields are:

- recipe_name: The name of the recipe. (String)
- prep_time: The amount of time required to prepare the recipe. (Integer)
- cook_time: The amount of time required to cook the recipe. (Integer)
- total_time: The total amount of time required to prepare and cook the recipe. (Integer)
- servings: The number of servings the recipe yields. (Integer)
- ingredients: A list of ingredients required to make the recipe. (List)
- directions: A list of directions for preparing and cooking the recipe. (List)
- rating: The recipe rating. (Float)
- url: The recipe URL. (String)
- cuisine_path: The recipe cuisine path. (String)
- nutrition: The recipe nutrition information. (Dictionary)
- timing: The recipe timing information. (Dictionary)
- img_src: Links to the image of the recipe

Design choices



The project implementation was carried out in several phases:

- Environment Setup
- Data Processing layer
- Chucking and Embedding layer
- Driver layer
- Conversation layer

1. **Environment Setup:**

- *Weaviate Docker*: Used Weaviate as the knowledge base to store the recipe data.
- *Redis Docker:* Used Redis as an in-memory database to store user chat histories.
- *Ollama Docker:* Integrated Ollama for local LLM model access when required.

2. **Data Processing layer:**
   - *Document processing*: Merged specific columns from the recipe dataset to feed into the knowledge database.
   - *EDA*: Examined document text lengths to determine optimal chunk sizes for processing.

3. **Chungking and Embeding layer:**
   - *Chunking Strategy*: Experimented with different chunk sizes and overlap strategies to ensure no important information is lost during retrieval. Two chunking methods from Langchain were explored: RecursiveCharacterTextSplitter and CharacterTextSplitter.
   - *Embedding Model Choice:* Tested embeddings from OpenAI's models and the Ollama library.

4. **Driver layer**
   - *Driver library*: Developed redisdb.py and weaviatedb.py for seamless interaction with the respective databases.

5. **Conversation Layer**
   - *Config*: Created a config/prompt_config.yml file to facilitate prompt customization.
   - *Tracing*: Used Langsmith to trace chatbot interactions, especially for debugging prompts. You can visit this link to see how Langsmith helps to visualize the chains.
   - *Chatbot chains*: Designed chatbot chains using Langchain to retrieve relevant documents, generate answers, and manage conversation histories.

Challenges faced

A significant challenge was selecting the appropriate chunk size and determining the optimal number of retrieved documents (Top-k). When too many recipes were retrieved, the language model (LLM) struggled to isolate the necessary ones to generate accurate answers. Moreover, the system had limitations on input and output token sizes.

Lessons Learned

1. **Effective System Design:**
    - The multi-layered architecture (Environment Setup, Data Processing, Chunking and Embedding, Driver, and Conversation Layers) ensured modularity and ease of management. Each phase handled specific tasks, allowing for efficient troubleshooting and scalability.
2. **Preprocessing Data:**
    - Preprocessing 100 recipes from the Kaggle dataset highlighted the importance of selecting relevant fields (e.g., recipe names, ingredients, directions) for accurate and contextually rich information retrieval. Structuring this data was crucial for the system's ability to generate meaningful answers.
3. **Chunking and Retrieval Optimization:**
    - Choosing the right chunk size and overlap strategy proved essential in ensuring that no vital information was lost. Exploring multiple chunking methods (RecursiveCharacterTextSplitter and CharacterTextSplitter) emphasized the balance needed between text size and overlap to improve retrieval quality without overwhelming the model.
4. **Embedding Model Comparison:**
    - Testing embeddings from OpenAI and Ollama offered insights into the varying performance of different models. Selecting the most effective embedding model played a pivotal role in enhancing the precision of retrieved documents and generated responses.
5. **Balancing Retrieval and Generation:**
    - One of the most challenging aspects was selecting the optimal number of retrieved documents (Top-k). Retrieving too many documents led to confusion in the generation phase while retrieving too few limited the comprehensiveness of responses. Fine-tuning the retrieval parameters to work within the system’s token limits was critical for improving the overall accuracy.
6. **Infrastructure Choices:**
    - Using Dockerized environments for Weaviate, Redis, and Ollama demonstrated the importance of infrastructure flexibility. Each component served a specialized role—Weaviate as a robust knowledge base, Redis for managing user history, and Ollama for local LLM processing—ensuring smooth operations and system reliability.
7. **Conversation Layer Configuration:**
    - Developing chatbot chains and customizing prompt configurations (via config/prompt_config.yml) allowed for adaptable and focused interactions. Tracing chatbot behaviors using Langsmith aided in refining prompts and improving conversation flows, particularly for debugging purposes.
8. **Importance of Tracing for Debugging:**
    - Utilizing Langsmith for tracing chatbot interactions was invaluable in identifying areas for prompt optimization and debugging interaction issues. The ability to

track conversation histories in Redis added transparency to the chat flow, ensuring user engagement was seamlessly maintained.