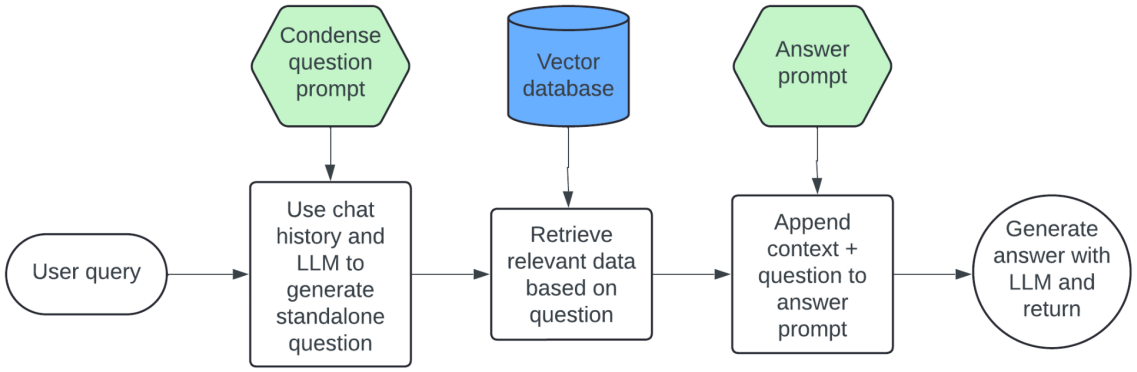
Project Overview

The first and most crucial step when creating a RAG application is preparing and storing the data for retrieval. My approach started with gathering relevant data from the Promtior.ai website and the provided PDF challenge document, efficiently storing it in a vector store for later use by the application.

The application consists of a pipeline of LangChain's runnables for processing user input and generating relevant and concise answers. Below is a detailed explanation of each component.

Try it out here 👈

Component Diagram



App Dependencies

- **BeautifulSoup:** Used for web scraping to fetch relevant data from the Promtior.ai website.
- **PyPDF:** Used for reading the challenge PDF and extracting relevant data.
- **Chroma:** Vector database used for storing and retrieving text.
- **OpenAI:** Provides embeddings (converting text into vectors) for Chroma and the LLM, using the GPT-3.5-turbo model.
- **LangChain:** Framework for developing AI applications, used to create a chain (pipeline of runnables) for conversational RAG.
- **LangServe:** Deploys the LangChain application as a REST API.
- **FastAPI:** Framework for developing REST APIs, used by LangServe.
- **Dotenv:** Loads environment variables from a `.env` file, used for loading API keys and other sensitive information.

Database Creation

The `database.py` script uses BeautifulSoup to scrape Promtior's website and LangChain's PyPDFLoader to load text from the challenge's PDF file. The RecursiveCharacterTextSplitter is then used to split large amounts of data into chunks for optimal storage and retrieval. Each text chunk is turned into a Document object and stored in the Chroma vector store, with a path to a persistent directory. This ensures that the database is created only once, and subsequent runs start from the persistent directory without repeating the fetching, embedding, and storage steps.

Application Details

Defining Prompt Templates

- **Condense Question:** Converts a follow-up question into a standalone question based on the chat history.
- **Default Document:** Formats retrieved documents into text.
- **Question Answering:** Answers questions based on retrieved context.

Helper Functions

- **_combine_documents:** Combines multiple documents into a single string using the default document prompt template.
- **_format_chat_history:** Formats the chat history into a readable string.

Setting Up the Vector Store and Retriever

Initializes the Chroma vector store and creates a retriever for querying the store using similarity search with `k=3` (retrieving the top 3 most similar documents).

Setting Up the LLM

Initializes the language model using OpenAI's GPT-3.5-turbo with a temperature setting of 0 (for deterministic output).

Chat History Model

Defines the structure of the chat history using a Pydantic BaseModel, consisting of a list of tuples (chat history) and a question. This is the first input in the chain.

Input Processing

Sets up a pipeline to process the input by:

- Assigning the formatted chat history and new question to the condense question prompt.
- Passing the prompt to the language model to generate the standalone question.

- Parsing the LLM's output into a string.

Context Processing

Defines a dictionary to pass inputs to the next step in the chain:

- **context:** Retrieves the standalone question, uses it to query the vector store (retriever), and combines the retrieved documents into a single text.
- **question:** Retrieves the standalone question.

Creating the Conversational QA Chain

Final chain for processing the conversational QA:

- Processes the input.
- Processes the context.
- Uses the QA prompt template and LLM to generate a response.
- Parses the response into a string and returns it.

Setting Up the FastAPI Application

Sets up the FastAPI application, adds a route to redirect home '/' to the chat playground, and defines routes for the conversational QA chain.

Running the Application

Runs the FastAPI application, serving it on `host=0.0.0.0` and `port=8000`.

Deployment

A Dockerfile and AWS Copilot were used to deploy a containerized version of the app.

Comments

I have been learning about RAG applications for the past three weeks, and this project helped me dive deeper into the RAG architecture. I found the process rather smooth and didn't encounter any major issues.

I hadn't heard about LangServe until now, but I was already planning to create a REST API to serve the application through an endpoint, handling chat history in the backend. Fortunately, the frontend provided by LangServe handled the chat history on the client side for me.

I learned quite a few things during this challenge, and I'm excited to continue learning more and level up my understanding of LangChain's chains, tools, and agents.

The chatbot successfully responded to the following questions:

- What services does Promtior offer?
- When was the company founded?
- What are their use cases?
- What is their contact information?