Documenting the Approach for Developing a RAG-Based QA Bot

Developer – Pratyush Puri Goswami

Email – ppurigoswami2002@gmail.com

Introduction

This document outlines the approach I took to develop a Retrieval-Augmented Generation (RAG) model for a Question Answering (QA) bot, including the implementation of an interactive interface. The project is divided into two main parts:

1. **Part 1: RAG Model for QA Bot**
2. **Part 2: Interactive QA Bot Interface**

Part 1: RAG Model for QA Bot

Overview

The primary objective was to develop a Retrieval-Augmented Generation (RAG) model capable of answering questions based on a provided document or dataset. My approach involved integrating a vector database for document embeddings and leveraging a generative model to create coherent, contextually relevant responses.

Key Components

1. **Vector Database:** I chose Chroma as the vector database due to its efficiency in storing and retrieving document embeddings. Chroma allows for scalable and quick access to document vectors, which is crucial for real-time query responses.
2. **Generative Model:** I utilized OpenAI's GPT-4o model for generating responses. This model is known for its advanced language capabilities, making it suitable for producing detailed and contextually accurate answers.

My Implementation approach and explanation-

1. **Environment Setup:**
    - **Package Installation:** I started by installing the necessary packages including langchain, langchain-community, langchain-chroma, beautifulsoup4, langchain_openai, and pdfplumber. These libraries provide the tools needed for document processing, embedding creation, and interaction with the OpenAI API.
    - **Configuration:** I configured environment variables to securely manage the Langchain and OpenAI API keys, ensuring that the application could

authenticate and access the necessary services without exposing sensitive information.

2. **Document Loading and Processing:**
    - **Document Extraction:** I developed functions to load documents from both PDFs and websites. For PDFs, I used pdfplumber to extract text from each page and create Document objects with metadata. For websites, I employed the WebBaseLoader from Langchain to fetch and parse web content.
    - **Text Splitting:** To handle large documents efficiently, I applied the RecursiveCharacterTextSplitter. This tool splits the text into manageable chunks with appropriate overlap, allowing for detailed context to be processed without overloading the model.
3. **Vector Store Creation:**
    - **Creating Chroma Vector Store:** I initialized a Chroma vector store from the document chunks. This involved adding the document embeddings to Chroma, which then facilitates quick similarity searches.
    - **Retrieval Functionality:** I implemented a retrieval function to search for relevant document chunks based on user queries. This function interacts with the Chroma vector store to fetch the most pertinent information.
4. **Generative Model Setup:**
    - **Integration with GPT-4o:** I integrated the GPT-4o model to generate responses based on the retrieved document context. The model was set up with a specific prompt template to ensure that answers are structured with headings, sub-headings, and clear paragraphs.
    - **Prompt Template:** I designed a prompt template to guide the model in producing detailed and coherent answers. The template includes placeholders for the context and query, ensuring that the responses are both informative and well-structured.
5. **Retrieval and Generation:**
    - **Query Processing:** I developed a function to handle queries by first retrieving relevant document chunks and then generating responses using the RAG model. This function ensures that the responses are based on the most pertinent information available.
    - **Handling No Relevant Information:** To address cases where no relevant information is found, I implemented a fallback mechanism. If the retrieval step yields no useful results, the system generates a default response indicating that no relevant information was found.

<u>Challenges I faced and My solutions to them</u>

1. **Handling Large Document Sizes:**

- **Challenge:** Processing large documents efficiently while maintaining context and relevance was a significant challenge. Large text blocks can overwhelm the model and impact performance.
- **Solution:** To manage this, I used text chunking with an appropriate overlap. This approach allows for the document to be divided into smaller, manageable chunks that preserve context while avoiding performance issues.

2. **Ensuring Accurate and Contextually Relevant Responses:**
    - **Challenge:** Generating responses that are both accurate and contextually relevant required careful handling of the document content and prompt design.
    - **Solution:** I fine-tuned the prompt template to guide the model towards producing structured and detailed answers. By clearly specifying the expected format and content, I improved the model's ability to generate coherent responses.

My Conclusion-

In summary, my approach to implementing the RAG model involved careful selection of components, thorough configuration, and attention to detail in both document processing and response generation. By addressing the challenges with targeted solutions, I was able to develop a robust system capable of providing accurate and detailed answers based on a given dataset.

Part 2: Interactive QA Bot Interface

Overview

In this part of the project, I developed an interactive interface using Streamlit to provide a seamless user experience for querying documents. The interface allows users to upload PDF files, submit questions, and receive real-time responses from the Retrieval-Augmented Generation (RAG) model established in Part 1. This integration ensures that users can interact with the chatbot and get answers based on the content of their uploaded documents.

My Implementation approach and explanation-

Frontend Development

To create a user-friendly interface, I chose Streamlit, which allows for rapid development of web applications with minimal effort. Here's what I did:

- **Streamlit Setup**: I used Streamlit to build a simple and intuitive web interface. This interface included features for uploading PDF documents and inputting text queries.
- **Document Upload**: Implemented a file uploader component in Streamlit, enabling users to upload PDF files directly through the web interface.

```
import os
import shutil
import uuid
import time
import pdfplumber
from langchain.llms import OpenAI
import streamlit as st
from dotenv import load_dotenv
from langchain.prompts import ChatPromptTemplate
from langchain.vectorstores.chroma import Chroma
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.schema.document import Document
```

I began by importing the necessary libraries. These include pdfplumber for PDF handling, langchain libraries for embedding and vector storage, and streamlit for the interactive web interface.

Document Handling

Handling the uploaded documents required processing and preparing them for querying:

- **PDF Processing**: I wrote functions to handle the uploaded PDF files, saving them locally and using pdfplumber to extract text.
- **Document Splitting**: After extracting text from the PDFs, I split the text into manageable chunks using RecursiveCharacterTextSplitter. This chunking was necessary to ensure that the documents were processed efficiently and could be embedded effectively.

```
@st.cache_data
def load_documents(pdf_file_path):
    """Load PDF document from the provided path and cache the result."""
    documents = []
    with pdfplumber.open(pdf_file_path) as pdf:
        for page in pdf.pages:
            text = page.extract_text()
            if text:
                documents.append(Document(page_content=text, metadata={"source":
pdf_file_path}))
    return documents
```

I used @st.cache_data to cache the result of load_documents() for efficiency. This function reads the uploaded PDF and converts its pages into Document objects, which I later use for embedding.

```
def split_documents(documents: list[Document]):
    """Split text into smaller chunks for processing."""
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,  # Increased chunk size for longer context
        chunk_overlap=50,  # Reduced overlap for faster processing
    )
    return text_splitter.split_documents(documents)
```

The split_documents() function uses RecursiveCharacterTextSplitter to divide the document text into manageable chunks. This helps in handling large documents more effectively.

Vector Store Creation

```
@st.cache_data
def add_to_chroma(_chunks: list[Document]):
  """Add document embeddings into Chroma."""
  global chroma_db  # Use global to avoid reinitializing Chroma DB

  if chroma_db is None:
    embedding_function = get_embedding_function()
    chroma_db = Chroma(persist_directory=CHROMA_PATH,
embedding_function=embedding_function)

  # Add new chunks to the Chroma database
  chroma_db.add_documents(_chunks)
  chroma_db.persist()
```

The add_to_chroma() function is responsible for adding document chunks to the Chroma vector store. I used @st.cache_data to cache this function, and a global variable chroma_db to avoid reinitializing the Chroma database.

Query Handling

Integrating query functionality with the RAG model was crucial for generating relevant answers:

- **Chroma Vector Store Integration**: I used Chroma to store and retrieve document embeddings. This involved clearing the previous Chroma database and adding new document chunks for each upload to ensure that the most recent document was always used for queries.
- **Query Functionality**: Implemented a mechanism to process user queries by interacting with the RAG model. This function retrieves relevant document chunks and generates responses using OpenAI's GPT-4o model.
- **Response Display**: The system was designed to show not only the generated answers but also the relevant document segments from which the answers were derived.

```
def query_rag(query_text: str):
  """Query the Chroma database and get a response from the OpenAI model."""
  global chroma_db, document_name

  if chroma_db is None:
    embedding_function = get_embedding_function()
    chroma_db = Chroma(persist_directory=CHROMA_PATH,
embedding_function=embedding_function)

  # Search for relevant chunks in the Chroma database
  results = chroma_db.similarity_search_with_score(query_text, k=5)
```

```
    context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results])

    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    prompt = prompt_template.format(context=context_text, question=query_text)

    # Use OpenAI model with a pre-defined temperature
    openai_model = OpenAI(temperature=0.5, max_tokens=2048)
    response_text = openai_model(prompt)

    # Use the document name for sources
    sources = [document_name] * len(results)
    formatted_response = f"Response:\n{response_text}\n\nSources: {', '.join(sources)}"
    return formatted_response
```

The query_rag() function queries the Chroma database to find relevant document chunks and uses the OpenAI model to generate a structured response. I use a predefined prompt template to ensure that responses are detailed and well-organized.

Clear and Refresh Functionality

To maintain the relevance of the responses and manage document updates, I added:

- **Database Clearing**: A function to clear the Chroma database, ensuring that each new upload starts with a fresh database.
- **Refreshing**: After adding new documents, the system was refreshed to update the model and vector store with the latest information.

```
def clear_database():
    """Clear the Chroma database."""
    global chroma_db
    if os.path.exists(CHROMA_PATH):
        try:
            if chroma_db:
                chroma_db._client.close()  # Close the Chroma client safely
            shutil.rmtree(CHROMA_PATH)
            chroma_db = None  # Reset the global variable after clearing
            print("Successfully removed the Chroma directory.")
        except Exception as e:
            print(f"Error removing Chroma directory: {e}")
```

This clear_database() function safely deletes the Chroma directory and resets the global chroma_db variable, allowing for fresh document uploads and processing.

Main app section-

```
def main():
    """Run the Streamlit app."""
    global document_name
    st.title("RAG Document Chatbot 📄")
```

```
# PDF Upload section
uploaded_file = st.file_uploader("Upload a PDF file", type="pdf")
if uploaded_file:
    # Save the uploaded PDF
    document_name = uploaded_file.name
    with open("uploaded_file.pdf", "wb") as f:
        f.write(uploaded_file.getbuffer())
    st.success("PDF uploaded successfully!")

    # Process the PDF
    st.write("Processing the uploaded document...")
    documents = load_documents("uploaded_file.pdf")
    chunks = split_documents(documents)

    # Clear the Chroma database before adding new documents
    clear_database()

    # Add the chunks to the Chroma database
    add_to_chroma(chunks)
    st.success("Document successfully added to the database!")

# Query section
query_text = st.text_input("Ask a question about the document:")
if st.button("Submit Query"):
    if query_text:
        response = query_rag(query_text)
        st.write(f"Response: {response}")
    else:
        st.warning("Please enter a query.")
```

In the main() function, I created the Streamlit app interface. Users can upload PDFs, which are then processed and stored in the Chroma database. Users can also submit queries, and the app retrieves and displays answers based on the uploaded documents.

Challenges I faced and My solutions to them

Challenge: Handling Queries Efficiently

Solution: To ensure the system could handle the queries efficiently, I optimized both the document processing and the vector store operations. This included improving the performance of text extraction and embedding storage to reduce delays and handle user interactions smoothly.

Challenge: Displaying Relevant Document Segments

Solution: Initially, the sources section showed document IDs, which lacked context for users. I modified this to display the document name instead, providing users with a clearer understanding of the source of the information used to generate the answer.

<u>My Conclusion</u>

The interactive QA bot interface successfully integrates with the RAG model to provide a user-friendly platform for document-based querying. By utilizing Streamlit, Chroma, and OpenAI's GPT-4o, the system delivers accurate and contextually relevant responses in real-time. The user interface enhances the experience by allowing for document uploads, efficient query handling, and clear presentation of sources. This project highlights the effectiveness of combining retrieval and generation techniques with a well-designed interactive interface to handle complex question-answering tasks.