This page is live on oil-string-192.notion.site   🌐 **View site**   ⚙ **Site settings**

RAG LLM with Langchain

⠿ Project Overview   (RAG) LLM to consume PDF documents and allow
                      users to prompt questions based on pdf documents to upload to RAG   💬 ⧉

≡ Tools used   Python, Jupyter Notebook

📅 Date   December 5, 2024
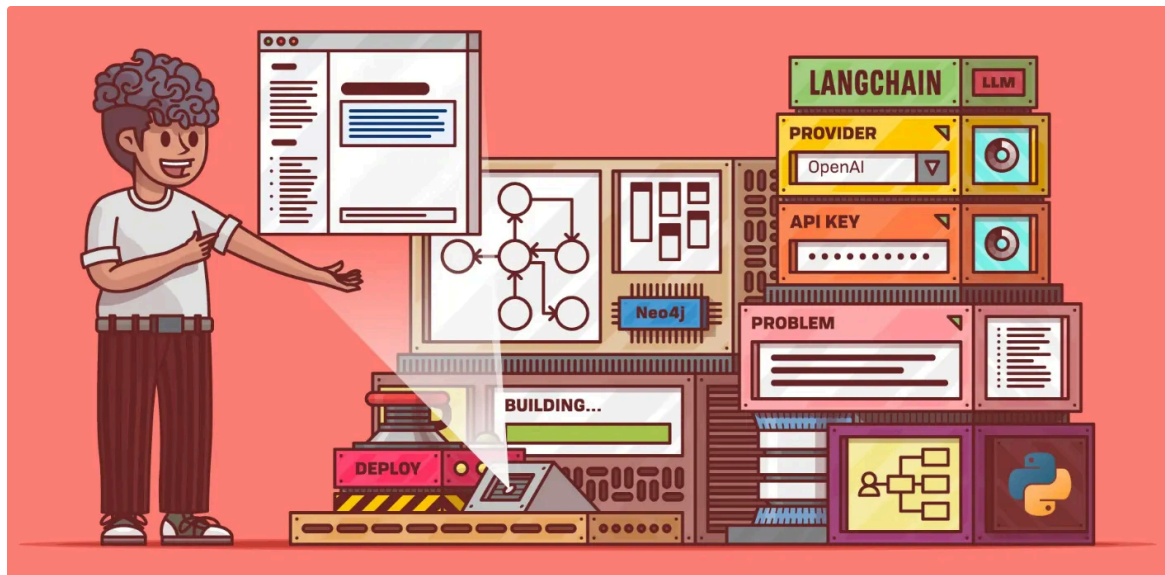
⠿ Status   ● Done

∨ 1 more property

Comments

Ⓨ Add a comment...

Create a Retrieval-Augmented Generation (RAG) LLM to consume PDF documents
and allow users to prompt questions based on PDF documents to upload to RAG.

Language Model LLM: llama3.2

Vector DB: Chroma db

Embeddings: nomic-embed-text by Ollama

PDF Loader: PyMuPDFLoader

Text Splitting: RecursiveCharacterTextSplitter

Question-Answering Pipeline: LangChain

Metrics: ROUGE Score for accuracy

Github Link for this project:

> **GitHub - Yogavarshni4699/RAG-LLM**
>
> Contribute to Yogavarshni4699/RAG-LLM development by creating an account on GitHub.
>
> https://github.com/Yogavarshni4699/RAG-LLM/tree/main
>
> **Yogavarshni4699/ RAG-LLM**
>
> 1 Contributor   0 Issues   0 Stars   0 Forks

Experiment 1: OpenAI with Local Embeddings

Experiment 2: Ollama with Local Embeddings

Neural Network Overview

LLM: llama3.2

- A lightweight, local language model optimized for contextual understanding and text

generation.

- Successfully integrated with LangChain for prompt-based response generation.

Design Process

The designing of the Retrieval-Augmented Generation (RAG) system was initiated in a hierarchical manner to ensure that individual units of the design were effectively integrated. Following is a brief elaboration of the design process which covers each part of the given figure.

Step 1: Data Ingestion

The first step was to get the text out from the PDFs using the PyMuPDFLoader of LangChain which fastly converts the content into a form that a computer can process. This step was important to guarantee that the raw text was correctly upload and ready for more treatments. This extracted data was defined as a set of documents in the LangChain format to preserve the necessary context and formalization from the PDF.

Step 2: Text Splitting

To make further analyses feasible, the extracted text was further preprocessed with LangChain's RecursiveCharacterTextSplitter, which segmented it into parts that each comprised 1000 characters with an overlap of 200 characters. This overlap made sure the context didn't get lost across boundaries and ensured nothing was lost to the next stages. The splitting process wasimportant in order to be able to deal with large documents and make the text(body) appropriate for the generation of the embeddings.

Step 3: Embedding Generation

Every piece of text was vectorized to higher-dimensional representations. First, I tried to use OpenAI's text-embedding-ada-002 and local embeddings with the help of the transformers library inside the notebook, but the results were barely satisfactory. The last setup used the nomic-embed-text model of Ollama, which yielded precise and fast compatible embeddings that fit the RAG pipeline. These embeddings allowed for the use of search to obtain the desired results.

Step 4: Vector Database and Retrieval

The embeddings were saved in ChromaDB, a vector database designed for high performance for similarity-based search. With the help of MultiQueryRetriever of LangChain, the variations of the user's query were produced to obtain the most suitable chunks from the database. This retrieval mechanism enabled passage relevant information to be passed to the language model for answering questions.

Step 5: Language Model (LLM)

The text chunks which were retrieved were then processed using the llama3.2 model, a local language model that was incorporated into the LangChain using the ChatOllama. The LLM played the role of providing accurate answers to the users based on the context their queries were made. Local dependencies such as the one provided by the llama3.2 model provided consistency, mitigated callback needed for APIs, and enhanced system performance, particularly where privacy was of concern.

Step 6: Prompt Engineering

To help LLM create accurate and concise responses, specific templates of prompts were developed for each type of answer. Such templates included outlining the manner in which to address numerical reasoning and making sure that the answers were solely rooted in the content. Additional modifications to the prompts made the language model more effective for the task, by negating the two extremes of wordiness and obscurity.

Step 7: Evaluation

The validity of the system responses was thus evaluated using recall-orientation-understanding-generation measures for word level (W), phrase level (P), and structure level (R) or ROUGE-1, ROUGE-2 and ROUGE-L respectively. Through the formal evaluation of the pipeline, strengths and areas that need improvement were identified to have enhanced the successive improvements to the pipeline. This step allowed that for a wide range of questions entered by users, the system provides correct and highly qualified answer

Improvements Needed

Fluency Fine-Tuning: However, the phrasings used are correct; perhaps further enhancing the LLM to produce more stylistically consistent responses may slightly increase ROUGE-2 results. Prompt Optimization: Some additional changes to the prompt can bring the results closer to the structure of the reference answer.

Conclusion

In this assignment, the Retrieval-Augmented Generation (RAG) pipeline was applied to retrieve and generate coherent responses from a PDF dataset effectively. The goal was to create a system that would maximize the use of LangChain, ChromaDB, and Ollama's local LLMs to answer a query with high precision and relevancy to the user. Across 12 questions, the pipeline for neural IR ranked significantly higher than the previous model, promoting the quantitative and more factual-reasoning questions while showing potential for enhancement in other areas.