An Open-source RAG framework

Overview:

The project is about implementing a Retrieval Augmented generation (RAG) framework that utilizes a provided set of patents (25,000) as knowledge base to generate answers for user queries. A schematic of the framework has been shown in Fig. 1.

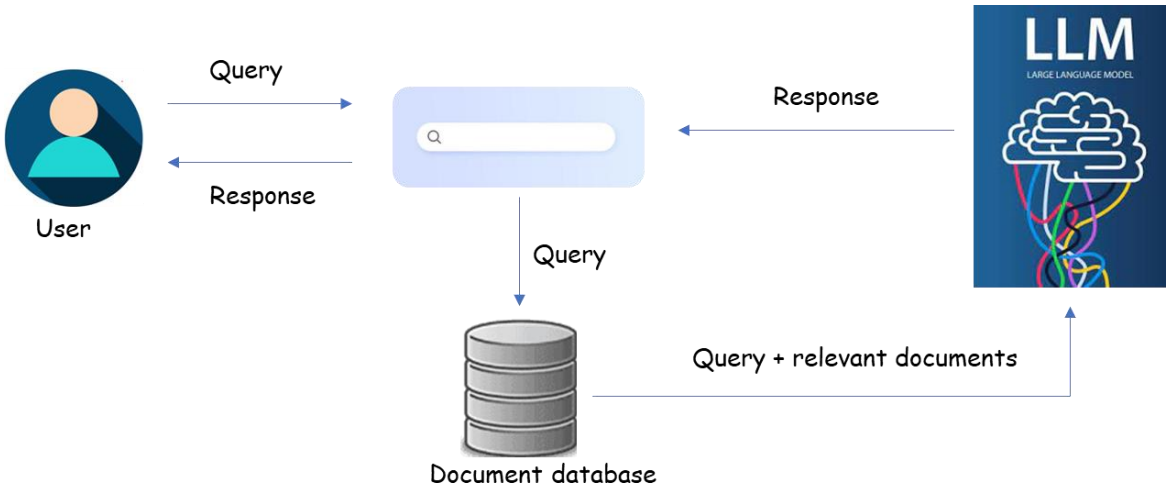

Fig. 1. Schematic of the implemented RAG framework

RAG is an AI framework for improving the quality of responses generated by large language models (LLM) by grounding the model on an external knowledge base which supplements the LLM's internal knowledge. The RAG uses generative AI to produce direct answers based on trusted data. The two major operations in the RAG system are retrieval and the generation. During retrieval algorithms search and retrieve information relevant to the user's prompt. During generation the LLM uses the prompt along with the information retrieved from the knowledge base to generate a response.

Components of RAG:

The framework has four main components:

1. Embedding model to embed the document database into embeddings
2. Vector store to store the generated vectors
3. The LLM
4. A language model integration framework

In the implemented RAG framework, all the components used are open-source and do not require the use of an external API. This approach ensures that there are no costs associated with using the models and provides the flexibility to update the components with the state-of-the art models when required.

An embedding model is used to transform a given text into a compact vector. This vector can be used for a number of purposes such as retrieval, classification, clustering, etc. The embedding model used in this implementation is the state-of-the-art 'BAAI/bge-base-en' model from the

Hugging Face library. The open-source vector database used is 'Chroma'. Chroma stores the embeddings and also helps retrieve the embeddings most relevant to the input prompt/query. The LLM used is 'HuggingFaceH4/zephyr-7b-beta' from the Hugging Face library. The language model generates the responses to the input queries by combining the knowledge gained from pre-training with the knowledge from the retrieved documents/pieces of texts. 'LangChain' is the open-source integration framework that was used. It allows us to combine the LLM with other components in the framework.

Working of Code:

1. Create a python environment for the project.
2. Install necessary packages using pip.
3. Import the required methods and functionalities.
4. Download the pretrained LLM using *AutoModelForCausalLM* class.
5. Download the tokenizer pertaining to the LLM using *AutoTokenizer.*
6. Load the downloaded corpus from the directory and split the text into chunks using *RecursiveCharacterTextSplitter*. Specify the required chunk size.
7. Download the '*BAAI/bge-base-en*' embedding model.
8. Specify the directory to which the embeddings have to be stored (persist_directory).
9. Convert the chunks into embeddings using the embedding model and store it in the vector database.
10. Initialize the retriever to retrieve the top '*k*' embeddings that are similar to the query.
11. Create a workflow for the LLM using *pipeline* function from *transformers.*
12. Create a chain (*qa_chain*) between the LLM and the retriever using the '*stuff*" chain type
13. Pass the query through the created chain.
14. The response is processed by the *process_llm_response* function and the processed response is printed.
15. If the questions are present in a text file, strip the questions and pass them to the *qa_chain*.

Hyperparameters:

1. *max_length*: maximum length of output text that will be generated by the model
2. *temperature*: This is a hyperparameter that controls the randomness of the model's output. A higher temperature value will result in more varied and creative output, while a lower temperature value will result in more factual and informative output.
3. *do_sample*: This is a boolean value that determines whether the model should generate samples or not. If set to 'true', the model will generate samples. If set to 'False', the model will return the most likely output.
4. *top_p*: This is a hyperparameter that controls the top probability value that will be returned by the model. A higher *top_p* value will result in a more confident output, while a lower *top_p* value will result in a more uncertain output.
5. *repetition_penalty*: This is a hyperparameter that controls the penalty for repeating the same output. A higher repetition penalty value will result in the model being penalized more for repeating the same output, while a lower repetition penalty value will result in the model being less penalized for repeating the same output.