Brance AI/ML Research Engineer Assessment Task

Name: Rishabh Dash
Linkedin Profile: https://www.linkedin.com/in/rishabh-dash-a7a955212/
Date Challenge Received: 22-07-2023
Date Solution Delivered: 24-07-2023

1. Problem Statement

The task was to create a Retrieval Augmented Generation (RAG) chatbot capable of answering user questions using information from a knowledge document. The architecture employs CTransformers for bindings for transformer models, Langchain for extensive integration, sentence-transformers for embedding representations and Facebook AI Similarity Search (FAISS) for efficient similarity search and clustering of dense vectors (Embeddings).

There was an attempt to prevent the RAG from generating false information (hallucination) and ensure it provided accurate answers.

Optional features like evaluating the relevance of responses, supporting multiple languages, and adding speech capabilities for voice input and output were not added due to time constraints. The model attempts to be efficient, accurate, and able to deliver contextually appropriate answers to user queries based on the stored reference document and knowledge.

2. Approach

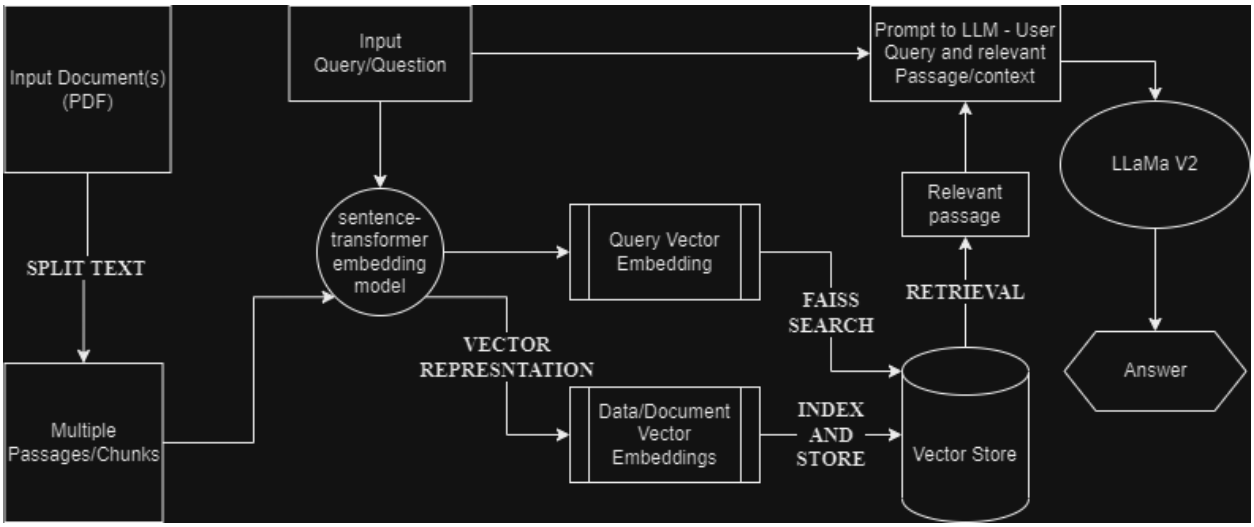
The approach to this problem was to

- Divy up the reference document into chunks and generate embeddings for them. (Huggingface embeddings via Langchain)
- Generate embeddings for queries (via sentence-transformer, sbert)
- Storing generated embeddings in a vectordb, and using FAISS for efficient similarity search and index them (generic K-means clustering and cosine comparison is also an option, but not the best.)

- Finally, use the closest context found via FAISS and the query to input to the LLM (Meta's LLaMa v2 in this case). This prevents hallucinations and make sure the answers are within the context of the document provided.
- A preemptive negative prompt discouraging hallucination and enforcing the strict adherence to Question/Answer format also helps discourage hallucinations.

3. Solution

Details about your solution. Illustrate performance and design with diagrams.



The code consists of mainly the following parts:

a. db_build.py – Preprocessing data and building the vector store

```python
# Import config vars, not encoding as utf8 can mess things up
with open('config/config.yml', 'r', encoding='utf8') as ymlfile:
    cfg = box.Box(yaml.safe_load(ymlfile))

# Build vector database
def run_db_build():
    # Read any and every pdf file than needs to be read
    loader = DirectoryLoader(cfg.DATA_PATH, glob='*.pdf', loader_cls=PyPDFLoader)
    documents = loader.load()
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=cfg.CHUNK_SIZE, chunk_overlap=cfg.CHUNK_OVERLAP)
    texts = text_splitter.split_documents(documents)

    embeddings = HuggingFaceEmbeddings(model_name='sentence-transformers/all-MiniLM-L6-v2', model_kwargs={'device': 'cpu'})

    vectorstore = FAISS.from_documents(texts, embeddings)
    vectorstore.save_local(cfg.DB_FAISS_PATH)
```

- The vector store will be generated and saved in the local directory named 'vectorstore/db_faiss'
- This will be the main db for semantic search and other operations.

b. prompts.py – Set up a template to prevent hallucination and reinforce question/answer format

```
# Make sure to follow the exact spacing and indentation in the prompt template for Llama-2-7B-Chat.
# It's sensitive to changes in whitespace!
# Incorrect spacing might cause issues in generating a summary from the provided context.

qa_template = """Use the following pieces of information to answer the user's question.
If you don't know the answer, just say that you don't know, don't try to make up an answer.

Context: {context}
Question: {question}

Only return the helpful answer below and nothing else.
Helpful answer:
"""
```

- Honestly, LLaMa V2 isn't well suited for RAG, compared to GPT, but LLaMa was available to run offline, so hence the choice.
- LLaMa unlike GPT isnt set up for Conversational Prompt-In Answer-Out, hence the need for this.

c. Models – Use the LLaMa V2 .bin, quantized to have 8-bit parameter values to reduce computational overhead. Can use 4/5/6 bit quantized bins as well, but the performance drops are significant.

d. llm.py – Construct LLM object (Langchain provided ctransformers wrapper for LLaMa)

```
# LLM object for the magic to happen
def build_llm():
    # Local CTransformers model
    llm = CTransformers(model=cfg.MODEL_BIN_PATH, model_type=cfg.MODEL_TYPE, config={'max_new_tokens': cfg.MAX_NEW_TOKENS, 'temperature'
    return llm
```

- Wrapper allows for simplified consistent interface to interact with the LLaMa.bin model.

e. utils.py – Enables performing document querying

Contains 3 main parts:

- Make a template for prompts in the form of [context, prompted_question]
- Define an object which will be used for searching vector store
- Instantiate the objects.

4. Future Scope

Due to time constraints and a lack of eprior xperience I was unable to implement evaluation metric, multilingual capability and speech capabilities to the RAG architecture.

- Evaluation of the answers: This can be done by using a variety of metrics, such as accuracy, relevance, and fluency.
- Supporting multi-linguality: This can be done by implementing a layer of open source or third-party translation model/API which would convert any input language into English, run the full model like normal, and while giving output would again interface with the model/API to convert the resultant English to whatever the initial language of choice was. Alternatively, if the source documents themselves are in a different language, using multilingual LLMs might be the correct path.
- Adding speech capabilities: Similarly, this can be done by implementing a speech-to-text engine and the reverse, text-to-speech engine layers over the input and output respectively.

As evident, there are a multitude of ways this RAG architecture can be improved. Building a serious state of the architecture, however, is a long iterative process with multiple trial and error steps, and always has room for experimentation. That, however, was not the scope of this assignment. We have demonstrated that using an inclusive orchestrator, multiple preexisting models can be woven and integrated together in a neat architecture to perform powerful operations.