Flight Crash Report Using LLMs

Tools used: LLaMa3, Python, Jupyter Notebook, LangChain, FAISS Vector Database

Summary/Abstract

The project focused on the following:

1. Installing and running LLama (Open source LLM) from the local machine
2. Creating a vector database and loading the aircraft crash report
3. Using Python script in Jupyter Notebook to display flight crash information

The Techniques used are RAG (Retrieval Augmented Generation) and Prompt Engineering. Once set up, the whole project can run offline without an internet connection.

What is LLama?

LLama is an open source large language model (LLM) by Meta which takes a prompt as input and predicts the next word to recursively generate a response.

Installation of LLama on local machine

Installing LLama onto a local machine helps in working through the development without internet connection. It helps in a faster development cycle.

The first step is to install Ollama, which can be found here. Select which operating system your device is running (Windows, Mac, Linux, etc) and select "Download". Once Ollama finishes downloading, open the Terminal and enter the command **ollama pull llama3.** This will download the Meta Llama 8B chat model.

Now that the model has been downloaded, enter the command **ollama run llama3**. This will result in "**>>>"** displaying to the console so that the user can enter a prompt. You can test using your favorite prompt.

```
(base) poojitha@Poojithas-MacBook-Air ~ % ollama run llama3
>>> What is the history on drones?
The concept of drones, also known as unmanned aerial vehicles (UAVs), has
been around for centuries. Here's a brief history:

**Early Experimentation**

* 1490: Leonardo da Vinci designed and drew plans for an ornithopter, a
mechanical bird that could be controlled remotely.
* 18th century: French inventor Jean-Baptiste Gribeauval created a
steam-powered flying machine that could carry small payloads.

**Modern Development**

* 1918: The first radio-controlled (RC) aircraft was built by the US Army
Signal Corps to test the feasibility of remote-controlled flight.
* 1940s: During World War II, the United States and Germany developed
early drone technology for military purposes, such as reconnaissance and
target practice.
* 1950s-1960s: The development of jet propulsion and guidance systems led
```

Hosting the Model on the local machine

- Ollama also provides a local API as shown in below script. Then, we define a function called llama3, which sends a prompt to a locally hosted API and generates a response.
- It imports two libraries: requests and json. Data is the dictionary which contains all the information to send in the POST request.
- Also included is the list messages which stores “role: user”, indicating that the one sending the message is the user, and “content: prompt” which contains the actual prompt.
- Headers is another dictionary that indicates that the data type of the request is JSON. Then, response stores the result from the API in JSON format and parses it into a Python dictionary.
- The llama3 function returns the value that corresponds to the key “content” from the “message” object from the response.

```
In [53]: import requests
         import json

         url = "http://localhost:11434/api/chat"
         def llama3(prompt):
             data = {
                 "model": "llama3",
                 "messages": [
                     {
                         "role": "user",
                         "content": prompt

                     }
                 ],
                 "stream": False,
             }

             headers = {
                 "Content-Type": "application/json"
             }

             response = requests.post(url, headers=headers, json=data)
             return response.json()["message"]["content"]
```

Feeding "specific" information/reports into Vector Database

FAISS is used as a Vector database store. The following section provides the steps and Python script to insert custom content into the vector database, retrieve its contents through RAG, and parse it into the LLM to generate a response.

1. **Installation of tools (LangChain, FAISS)**

First, we need to run these commands to install the following Python packages using the package manager **pip**. This block of code is required only for the first time, hence it can be separated into a different python script at a later point. For now, I commented it out in the Jupytor notebook script.

```
In [51]: #!pip install langchain
         #!pip install sentence-transformers
         #!pip install faiss-cpu
         #!pip install bs4
         #!pip install replicate
         #!pip install langchain_community
         #!pip install -U langchain-huggingface
```

Now that the packages are installed, this next snippet of code imports all the necessary libraries and functions from the LangChain community. These are crucial components of the Python script as they allow the model to parse content, split text, etc.

```
In [52]: from langchain_community.embeddings import HuggingFaceEmbeddings
         from langchain_community.vectorstores import FAISS
         from langchain.text_splitter import RecursiveCharacterTextSplitter
         from langchain_community.document_loaders import WebBaseLoader
         #from langchain_huggingface import HuggingFaceEmbeddings
         import bs4
```

2. Loading into the Vector database

The following block of code focuses on the retrieval abilities of the LLaMa model. ChatOllama is a wrapper class from LangChain which allows us to interact with llama3. “Llm” is an instance of ChatOllama using the llama3 model. The temperature parameter is set to 0 in order to get the most accurate response to our prompt. A higher temperature results in more random answers.

Now we import the ConversationalRetrievalChain, which allows the model to look up information and answer questions from a stored collection of data. Then we create “chat_chain”, an instance of ConversationalRetrievalChain that takes “llm” as a parameter. The second parameter, “vectorstore.as_retriever()”, sets up the vector database to retrieve the documents mentioned in the prompt. Finally, “return_source_documents” is set to true in order to understand and verify the model’s response.

```
In [54]: from langchain_community.chat_models import ChatOllama
         llm = ChatOllama(model="llama3", temperature = 0)

         from langchain.chains import ConversationalRetrievalChain

         chat_chain = ConversationalRetrievalChain.from_llm(llm,
                                                            vectorstore.as_retriever(),
                                                            return_source_documents=True)
```

```
In [41]: #Article from Dr. Namuduri
         loader = WebBaseLoader(["https://www.flyingmag.com/news/ntsb-releases-details-on-2-lockheed-12a-crashes/"])
         #Test file - What's new about Llama 3
         loader2 = WebBaseLoader(["https://huggingface.co/blog/llama3"])
         docs = loader.load()
         docs2 = loader2.load()
         # Split the document into chunks with a specified chunk size
         text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
         #Loading multiple documents
         all_splits = text_splitter.split_documents(docs + docs2)


         # Store the document into a vector store with a specific embedding model
         vectorstore = FAISS.from_documents(all_splits, HuggingFaceEmbeddings(model_name="sentence-transformers/all-mpnet-bas
```

In this block of code, method WebBaseLoader incorporates web pages into the LangChain framework so that the language model can extract information from it. There are 2 WebBaseLoaders: loader and loader2. Loader contains the article about NTSB’s aircraft crash

reports provided by Dr. Namuduri and loader2 contains a test link detailing the new features in Llama 3.

For now, we will focus on loader.

After inserting the article, the RecursiveCharacterTextSplitter splits the document into chunks of any size. We used a chunk size of 500. Additional research needs to be done to optimize the chunk size. Since the accident report is a lengthy document, this facilitates readability when the model reads it. Then the document is stored into a vector database with the name of the model.

What is a vector database?

A vector database is a collection of data that is stored as a numerical representation (vectors). This data can come in the form of text, images, videos, audios, etc. This allows for similarity searching within a model, meaning the user can ask for similar content within the data. We stored the NTSB article in the database, which gives the model access to it.

```
In [50]: #RAG from Dr. Namuduri's document
chat_chain = ConversationalRetrievalChain.from_llm(llm,
                                                   vectorstore.as_retriever(),
                                                   return_source_documents=True)
chat_history = []
question = "Can you tell me about the Chino, California incident?"
result = chat_chain({"question": question, "chat_history": chat_history})
print(result['answer'])
```

3. Retrieving the reports from Vector database (RAG)

Now that the article has been loaded into the vector database, we can prompt Llama3 about the incidents in Chino, California and Jackson, Georgia.

First, we need to use the ConversationalRetrievalChain in order to ask Llama follow up questions about the aircraft accidents. This creates a chain of all the prompts the user asked so that it can remember previous information and provide a more tailored response.

The user prompts Llama “Can you tell me about the Chino, California incident?” and stores the result in the chat chain. Then, print Llama’s response to the console.

```
According to the context, on June 15 in Chino, California, a Lockheed 12A aircraft (N93R) crashed during takeoff wh
ile participating in a Father's Day airshow. The pilot and copilot were killed in the accident. The NTSB investigat
ion found that the tailwheel lock control lever was in the locked position, but the locking tab on the control-whee
l assembly appeared to be unlocked. Additionally, there was no fluid leakage observed.
```

Benefits of using Llama

1. **Accessibility everywhere**

LLaMa is run locally on the device, so it can be accessed without an internet connection. Regardless of where or when the user runs Llama, it will generate responses without depending on Wi-Fi.

2. **Open Source LLM**

LLaMa allows researchers and developers to understand how to train the model. This opens up opportunities for organizations to use their own internal data to train LLaMa's models. Since the foundation is already there, it is convenient to build on it to generate better and customized outputs.

Where do we go from here?

Now that we know LLaMa is capable of parsing information from any document and providing accurate responses, we can explore LLaMa's capabilities further.

1. Implementing image generation. If the user asks for a picture to support a response provided by LLaMa, it should be able to generate such an image. Will explore the open source models to do text to image, text to video etc.

2. Scalability: The proof of concept can be converted into "Flight crash report" API and hosted for consumption. More crash documents can be loaded into the Vector database.

3. Minimizing the hallucinations in order to provide clear and logical responses. If the user asks for advice pertaining to a certain topic, LLaMa's output should be tailored to the situation without error. For example, if a person asks, "I'm feeling airsick, what should I do?" LLaMa should be able to provide a list of medicines he or she can take, suggest a few breathing exercises, and encourage the user to rest.