VIETNAMESE - GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MedFood Chatbot Report

Lecturer: Dr. Dinh Quang Vinh

Student 1: Huynh Tan Khai - 10421111
Student 2: Luong Duc Huy - 10421022

11 / 2024

Content

1 Background

Large language models (LLMs) have become the center of attention in the AI industry since the introduction of the GPT2 and GPT3 models by OpenAI in early 2020. It was quickly realized that LLMs could power a new generation of helpful chatbots that can, to some degree, give reason about their answers when asked a question, rather than merely parroting what they have learned from the dataset used for training these chatbots. This short report introduces a chatbot trained specifically to answer basic questions regarding medical conditions, diseases, as well as recipes for a variety of simple, homemade dishes.

There are many choices available when it comes to choosing what the bot is to be powered with, as well as the corpus to be used for training. Due to our limited computational resources and limited access to training materials, we have decided to choose a model that is slightly less capable than what the current state-of-the-art can achieve.

Figure 1: AI-Powered Chatbot

2 Tools and Libraries

A variety of tools were used to build this chatbot:

- **Model**: **LLaMa-2-7B** variant. We chose this model as it is free and open-source, and it can be configured to be deployed either on the cloud or on one's own machine - a feature that privacy-concerned users will appreciate. It also performs well for its relatively small size.
  - Parameter count: 7 billion parameters
  - Training dataset size: 1.0 trillion tokens from publicly available datasets (e.g., Common Crawl, C4, Wikipedia, GitHub)
  - Architecture: Transformer-based architecture
  - Training devices: NVIDIA A100 GPUs (up to 80 GB VRAM per GPU), distributed across multiple GPUs
  - Precision: Mixed precision (FP16) for memory and computation efficiency
- **LLM frameworks**: Chainlit. It is an all-inclusive open-source Python package for creating conversational AI web applications. It comes with several useful features, one of the most notable being the ability to store past conversations, useful when users may need to look up what they have asked before. Langchain, a popular framework for integrating LLMs into applications, was also used.
- **Other tools**: A variety of Python libraries were also used, primarily FAISS (Facebook AI Similarity Search, an efficient library for performing similarity search) as well as text/PDF file processing libraries from HuggingFace.

3 System Requirements for LLaMA-7B

1. **CPU**
    - **CPU**: Modern, multi-core CPU (8+ cores recommended, e.g., Intel i7/i9, AMD Ryzen 7/9)
    - **RAM**: 16–32 GB recommended
    - **Performance**: Much slower than GPU; feasible for small batch sizes
2. **GPU**
    - **VRAM**: 12–16 GB recommended; 16+ GB ideal for better performance
    - **Batch Size**: Lower VRAM may require smaller batch sizes
3. **Fine-Tuning (Optional)**
    - **VRAM**: At least 24 GB recommended
    - **Compute**: High computational power and storage needed
4. **Software Requirements**
    - **Python**: Version 3.8 or higher
    - **Libraries**: PyTorch, Hugging Face Transformers, CUDA, and cuDNN. Several Langchain libraries, notably langchain_core, langchain_community are also needed *__Note__*: To run LLaMA on CPU, we use Hugging Face CTransformers
    - **Optimization**: Libraries like DeepSpeed or Accelerate (optional)

4 System Architecture

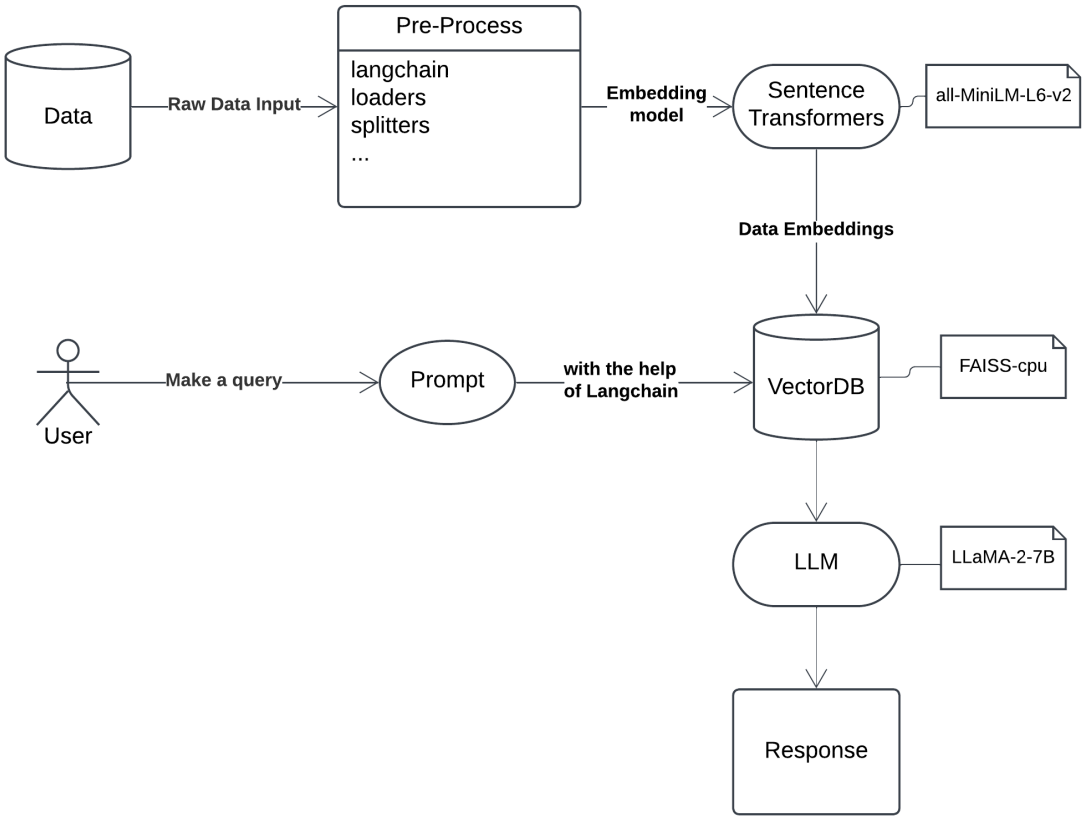

Figure 2: Chatbot Architecture

This diagram represents a chatbot architecture that demontrates the process to generate a response to the user based on their query:

- **Data pre-processing**: Raw input data are from 5 volumes of *Gale Encyclopedia of Medicine* and *101 Square Meals* PDF files, which are loaded, split and restructured for embedding using loaders and splitters provided by LangChain.
- **Embedding model**: After pre-processing, data is transformed into vector embeddings using a sentence transformer (*all-MiniLM-L6-v2*) from Hugging Face.
- Embeddings are stored in a vector database using *FAISS-cpu* for efficient storage and information retrieval.
- The user submits a query, which goes into a customized prompt. With the help of LangChain, the prompt retrieves the most relevant embeddings from the VectorDB.
- **LLM**: Retrieved embeddings are sent to a Large Language Model (*LLaMA-2-7B*), which generates a coherent response based on the embeddings to the user.

5 Building the Chatbot

5.1 Dataset Pre-Processing

Since our dataset is from a PDF file, we need to load it first. Langchain has the needed libraries for processing PDF files, so we need to import those.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import PyPDFLoader, DirectoryLoader
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS

DATA_PATH = "data/"
DB_FAISS_PATH = "vectorstores/db_faiss"

# Create vector database
def create_vector_db():
    # Load PDF documents
    pdf_loader = DirectoryLoader(DATA_PATH, glob='*.pdf', loader_cls=PyPDFLoader)
    documents = pdf_loader.load()
```

We then split the PDF file (now into a format that can be parsed) into chunks of text. Textual vectors can then be created out of these chunks.

```
 # Split documents into chunks
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
    texts = text_splitter.split_documents(documents)

    # Create embeddings and vector database
    embeddings = HuggingFaceEmbeddings(model_name='sentence-transformers/all-MiniLM-L6-v2', model_kwargs={'device': 'cpu'})
    db = FAISS.from_documents(texts, embeddings)
    db.save_local(DB_FAISS_PATH)

if __name__ == "__main__":
    create_vector_db()
```

5.2 Model Integration

In this step, we finally put the model to use by integrating it with the rest of the application. Langchain is once again extensively used for setting up various key features of the web application, most notably the ability to retain past conversations in a session (limited to 1, in other words, the chatbot can retain memory of the latest question-answer pair in a conversation).

Once again we start by importing the necessary libraries. We prompt the bot to generate more concise answers by way of a custom prompt:

```
DB_FAISS_PATH = "vectorstores/db_faiss"

custom_prompt_template = """You are a helpful assistant. Based on the information provided, provide a concise answer.

Conversation History: {chat_history}

Context: {context}

Question: {question}

Answer:
"""
```

Next, we load the model to be used - in this case, the 7b variant of LLaMa 2. We also set up the maximum context length to be used - depending on your hardware configuration, you may want to increase or decrease this value as your hardware allows.

```
def load_llm():
    llm = CTransformers(
        model="llama-2-7b-chat.ggmlv3.q8_0.bin",
        model_type="llama",
        config={
            "max_new_tokens": 2048,
            "context_length": 4096,
            "temperature": 0.5
        }
    )

    return llm
```

To force a topic change, the user types in `/reset` in the message box. This flushes the memory of the chatbot in a specific conversation, allowing the chatbot to start all over again. Once the

memory is successfully flushed, a success message is printed on the screen, and the user may start asking again. Previous messages in the same conversation will still be retained. Note that the code snippet below is part of the main driver code - we will return to this later in the report.

```
@cl.on_message
async def main(message):
    chain = cl.user_session.get("chain")  # Retrieve the chain from the session

    # Check if the user wants to reset the memory manually
    if message.content.strip().lower() == "/reset":
        chain.memory.clear()  # Clear the memory
        await cl.Message(content="Memory reset! You can now start a new conversation.").send()
        return
```

Naturally the chatbot cannot reset its current memory if it does not have one in the first place, so we initialize one. The role of the memory is to ensure the chatbot works as intended; to be more precise, it will take the history of a given conversation into account as it generates a response, making sure it can produce more contextually relevant answers. Here, the context window is limited to the most recent question-answer pair, although this can be modified to the user's liking, since the chatbot is intended to be run locally.

```
def initialize_memory():
    # Initialize memory with a window of the last 1 exchange
    return ConversationBufferWindowMemory(
        k=1, memory_key="chat_history", input_key="question", output_key="answer", return_messages=True
    )
```

In the next step, we link everything we have built so far - most notably the custom prompt, into a complete package, containing the LLM first and foremost, and a vector store database for the previous context that the LLM can use as additional reference that may help it answer questions.

```
def retrieval_qa_chain(llm, db, prompt):
    memory = initialize_memory()

    qa_chain = ConversationalRetrievalChain.from_llm(
        llm=llm,
        retriever=db.as_retriever(search_kwargs={'k': 2}),
        memory=memory,  # Memory for conversation history
```

```
        return_source_documents=True,
        combine_docs_chain_kwargs={"prompt": prompt},  # Pass the custom prompt here
        verbose=True  # Optional for debugging
    )
    return qa_chain
```

We start by initializing the memory that the bot will use to retain a conversation's context, helping it produce more contextually relevant answers. We proceed by initializing a conversational retrieval chain - a system that enables the model to answer questions both based on the current question and past questions, by loading the model itself, a vector store that is used for retrieving information relevant to the current question, and memory (otherwise known as the context) which, again, can be used as additional context for generating answers. The custom prompt at the beginning is also passed in here, helping the chatbot to operate more efficiently by providing concise answers based on the training data.

The next function defines the required components for the chatbot, including the embeddings (using a pretrained model provided by Hugging Face; this model transforms text into vector representations of text, or embeddings), a vector store database for retrieval of relevant information from the training data, and the same custom prompt from the previous function - this time, the custom prompt is used for initializing the custom prompt that will later be used in the actual conversational handling, as defined in the `ConversationalRetrievalChain`.

```
def qa_bot():
    embeddings = HuggingFaceEmbeddings(model_name='sentence-transformers/all-MiniLM-L6-v2', model_kwargs={'device': 'cpu'})
    db = FAISS.load_local(DB_FAISS_PATH, embeddings, allow_dangerous_deserialization=True)
    prompt = set_custom_prompt()
    llm = load_llm()

    # Pass memory when setting up the retrieval chain
    return retrieval_qa_chain(llm, db, prompt)
```

5.3 User-Chatbot Interaction

For added convenience, six starter questions are displayed when the user starts a session, provided by Chainlit's built-in Starter classes. This can be used when the user wants to quickly start a conversation with the bot (note that to keep the report concise, we truncated the code snippet below).

```
@cl.set_starters
async def set_starters():
    # Pre-set topics the user can select from to start the conversation
    return [
        cl.Starter(
            label="Diabetes",
            message="What is diabetes?",
            icon="public/diabetes.png",
        ),
        cl.Starter(
            label="Allergies examples",
            message="List some common allergies and their symptoms?",
            icon="public/allergy.png",
        ),
        cl.Starter(
            label="Relieve fever",
            message="How to relieve fever?",
            icon="public/fever.png",
        ),
```

Upon starting up a chatbot session, the user is greeted with the following screen. The user may either click on any of the starter options to quickly begin a conversation, or type a question in the message box below.

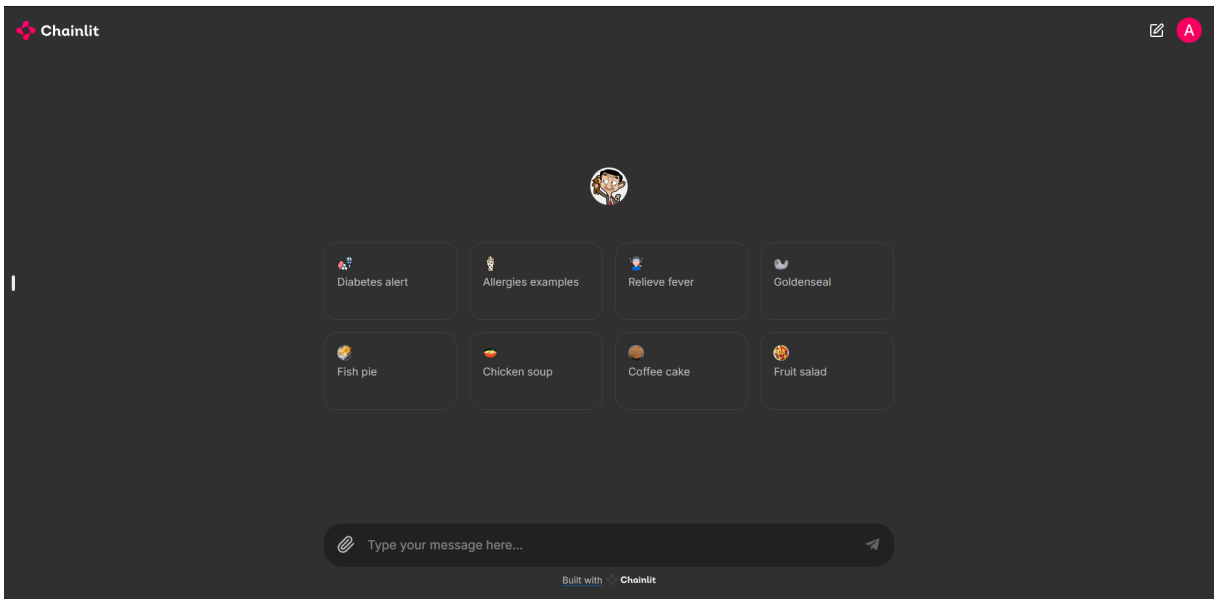

Figure 3: Welcome screen

Upon clicking on any of the starter questions, or sending a question in the message box, the chatbot will begin searching its dataset for context, and use it to produce an answer. During this process, the user may click on the downward arrow just above the bot's message to see what kind of process is being used to retrieve and produce the answer.

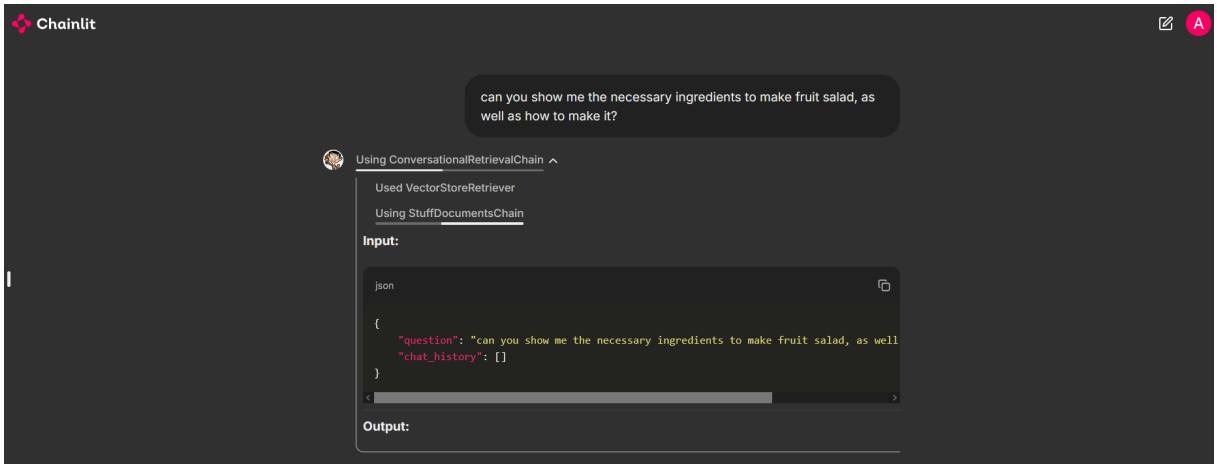

Figure 4: The chatbot looking up the relevant information to answer

Depending on the user's hardware specifications, the time taken for the bot to generate an answer may vary. In this report, the chatbot was run on a laptop with an Intel i7 CPU and 16 GB of RAM. The chatbot's UI will look like this once it has finished generating the answer:

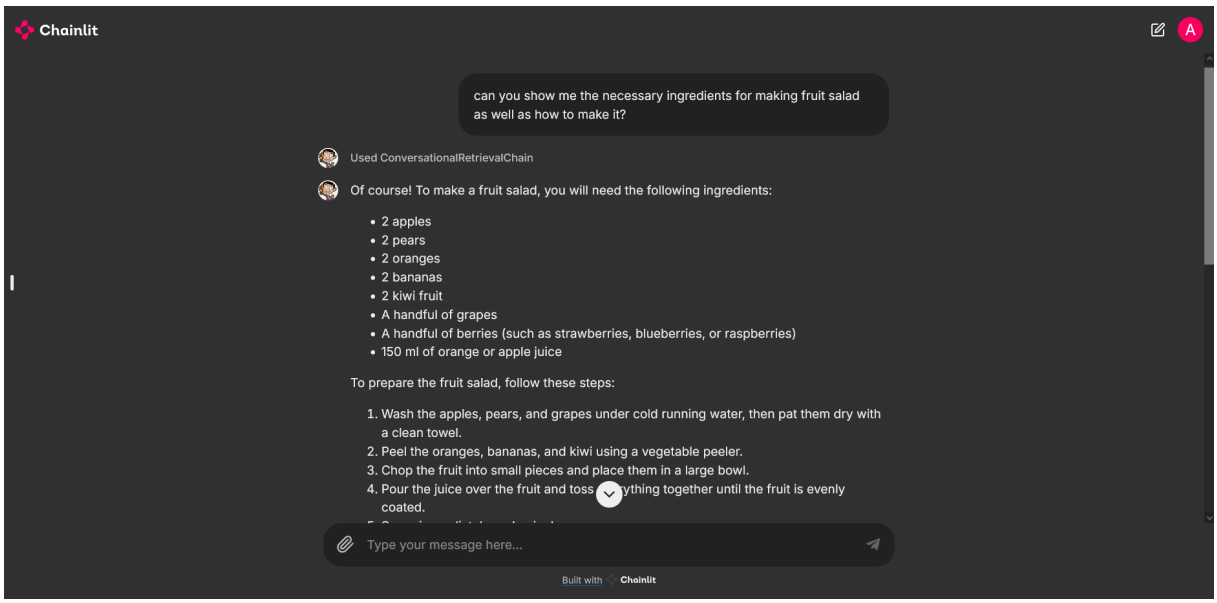

Figure 5: The UI after the bot has finished generating the answer

The user may continue the conversation using the message box below. They may also attach a file and ask the chatbot questions relevant to said file - though this button is not actually functional; we will go through this in a later section.

The next two functions initializes the chatbot when a new session is started, as well as enables chats to be resumed.

```
@cl.on_chat_start
async def start():
    # Set up the chain
    chain = qa_bot()
    cl.user_session.set("chain", chain)
```

A new conversational chain is created, with the arguments as described earlier in the report. This chain is then saved in the user's session; the chat's history is persisted in future sessions.

```
@cl.on_chat_resume
async def resume():
    chain = cl.user_session.get("chain")     # Retrieve chain stored in the session

    # Check if the chain is not set (in case of a new session)
    if not chain:
        chain = qa_bot()  # Recreate chain if missing
        cl.user_session.set("chain", chain)  # Save chain to the
```

```
    session
```

On resuming a chat, the relevant conversational chain is retrieved. If the user has initiated a new session instead, this chain is then reconstructed, then again, saved to the user's session.

We have made mention of the main driver code very early in section 5 of this report. Next, we will cover the rest of it.

```
cb = cl.AsyncLangchainCallbackHandler(
        stream_final_answer=True, answer_prefix_tokens=["FINAL", "ANSWER"]
    )
    cb.answer_reached = True
    chat_history = chain.memory.chat_memory.messages if chain.memory else []

    inputs = {
        "question": message.content,  # Pass the message content as 'question'
        "chat_history": chat_history  # Pass the current conversation history
    }
    res = await chain.acall(inputs, callbacks=[cb])
    answer = res.get("answer", "No answer found")
    await cl.Message(content=answer).send()
```

An `AsyncLangchainCallbackHandler` object is created for handling the streaming of answer, meaning that the answer will be produced and printed out on the screen as it is generated, rather than the complete answer being printed out at once. Once done, a flag is raised to notify that the answer is complete (`cb.answer_reached = True`). The chat history is then retrieved only if the memory for that conversation exists, allowing the chatbot to generate more contextually relevant answers if needed.

Recall that the first line of the main driver code looks like this: `async def main(message):`. The content of the argument `message` is then used as the question for the bot to answer, as well as the chat's history, which the bot can use as additional context during answer generation. These, and the callback handler object `cb`, are then used to call a conversational chain asynchronously. `cb` handles the streaming and handling of the final answer. The result is then returned as the object `res`.

If an answer is successfully generated, it is returned as an object, which will later be used to "send" the answer back to the user. Else, a default message is returned instead.

6 Performance & Data Persistence

For a simple question ("What is diabetes?"), the chatbot takes, on average, around 1 minute to generate an answer. Questions that require longer answers like recipes ("Ingredients and method to make fish pie?") often result in the chatbot taking longer to respond, about 2-3 minutes.

Every time a new conversation is created, it is then saved and can be resumed at any time using the panel on the far left side of the screen (indicated by the thin white rectangle). Clicking on the rectangle will reveal the current conversations, as well as past ones. To resume any of the chat, the user clicks on any chat on this pane, then click "Resume chat" on the screen section to the right. Message contents from past visits are retained.

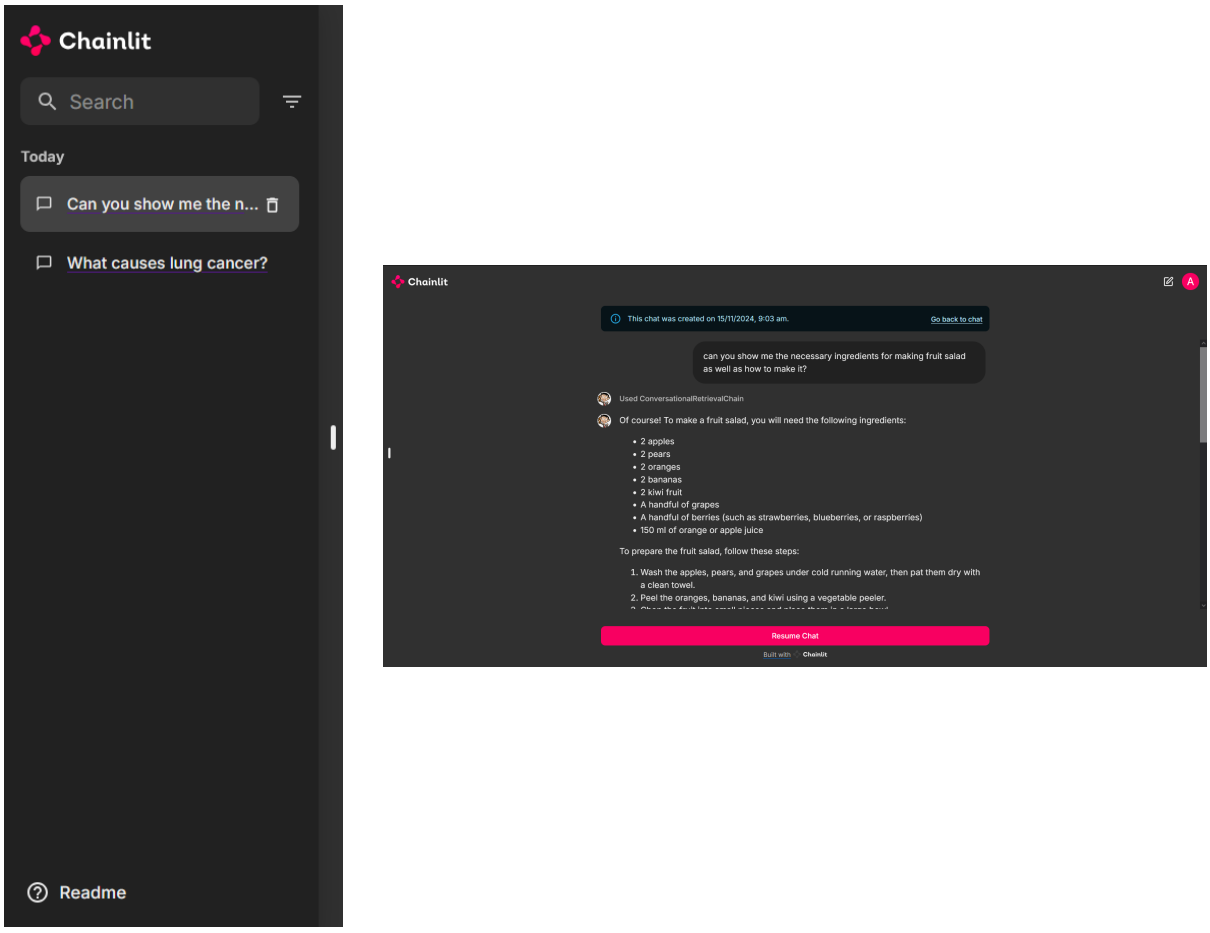

Figure 6: History Pane

7 Future Prospects

With our limited computational resources, we opted to use a very small training corpus, as we intend this to be mostly used on a local machine for answering simple medical questions, so the small training corpus has enough scope for handling this expected use case. As it is currently, our chatbot is somewhat limited in the scope of what it can do, so we have planned to expand its functionalities in future updates. Below we detail some of what we already have a clear plan of.

First and foremost, the user can manually type `/reset` should they want to manually update the context of the current conversation. This can be convenient for when the bot is incapable of detecting a topic change. In a future update, we plan to expand this functionality by giving it the ability to "notice" a topic change, and update the context as necessary. There are several ways this can be implemented; one approach we have used so far is to use cosine similarity, which has produced good results so far.

In section 5.3 we mentioned that the file attachment button is not actually functional. While the button itself does work - a dialog box opens when it is clicked, and a file can be selected and sent to the chatbot, courtesy of Chainlit - the chatbot is unable to actually process what is in the file, be it an image or a PDF file, because there is no function written that would let it process the uploaded file(s). It would, therefore, simply generate an answer based on the question asked, plus the text embeddings from its dataset. This is a feature that we also aim to implement in a future update to the bot, with functions incorporating file parsers for popular file types such as .pdf, .txt and .csv, among others, as well as image encoders for image files. This can be useful when a user wants to identify certain types of herbs.

Given what the chatbot does, a part of the user base is likely going to include people with motor impairment; they may only be able to partially use their arms or not at all. It is for this reason that we have also planned to implement a speech recognition model, allowing those users to use the bot as well.

8 Conclusion

Using publicly available tools, frameworks and training data, we have successfully built an LLM powered chatbot capable of structuring and generating concise, coherent answers to questions related to medical conditions and medicines, as well as the ingredients and how to prepare simple homemade dishes. While what we did was far from perfect - hangs and timeouts were common issues during the testing process - the process of constructing this chatbot did provide us with plenty of experience on how to set up, train and run a chatbot locally. This might have been a more computationally expensive approach as opposed to using APIs from proprietary LLMs, which would have meant faster answer generation speeds as well as access to better models (without the need for expensive local hardware), but we are privacy-centric users; a locally trained and run chatbot means our training data and conversations will never travel outside our own machines. Given that the scope of what our chatbot can do is rather limited at the moment, though, we hope to add the new features that we have detailed above.

9 Reference

9.1 Learning Resources

1. **LLaMA Research Paper**:
   https://doi.org/10.48550/arXiv.2302.13971
2. **Chainlit Documentation**:
   https://docs.chainlit.io/get-started/overview
3. **Literal AI Documentation**:
   https://docs.literalai.com/get-started/overview
4. **LangChain Tutorials**:
   https://python.langchain.com/docs/tutorials/llm_chain/
5. **LLaMA-2-7B Model**:
   https://huggingface.co/TheBloke/Llama-2-7B-Chat-GGML
6. **Sentence Transformers**:
   https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

9.2 Project Tools

1. **Visual Studio Code**
2. **Python**
3. **LangChain**
4. **Chainlit + Literal AI**
5. **FAISS** (Facebook AI Similarity Search)
6. **Hugging Face**

Figure 7: Project Tools