RAG Chatbots for Technical Documentation

Advanced Topics on Machine Learning 2024/2025

Athos Coghi Freitas 202108792
Félix Martins 202108837
Luís Du 202105385

Problem description

Build a **Retrieval-Augmented Generation (RAG)** chatbot that can answer complex regulatory questions based on technical documentation

Goal

Help users access and understand technical information efficiently, reducing the need to manually navigate large bodies of documentation

Tools and resources

Programing Language: Python

Framework: LangChain

- Provides tools for RAG Architecture
- Offers Evaluators and benchmarks to measure performance

LLM Provider: Gemini

- Provides API access to LLM “Gemini 1.5-flash”
- Provides API Access to embedding model “Text Embeddings 004”

Interface: Chainlit

- Easy to use event-driven design
- Integrations with many Orchestration tools, namely LangChain

Chosen document

Regulation (EU) 2017/745 of the European Parliament and of the Council concerning medical devices

Characteristics

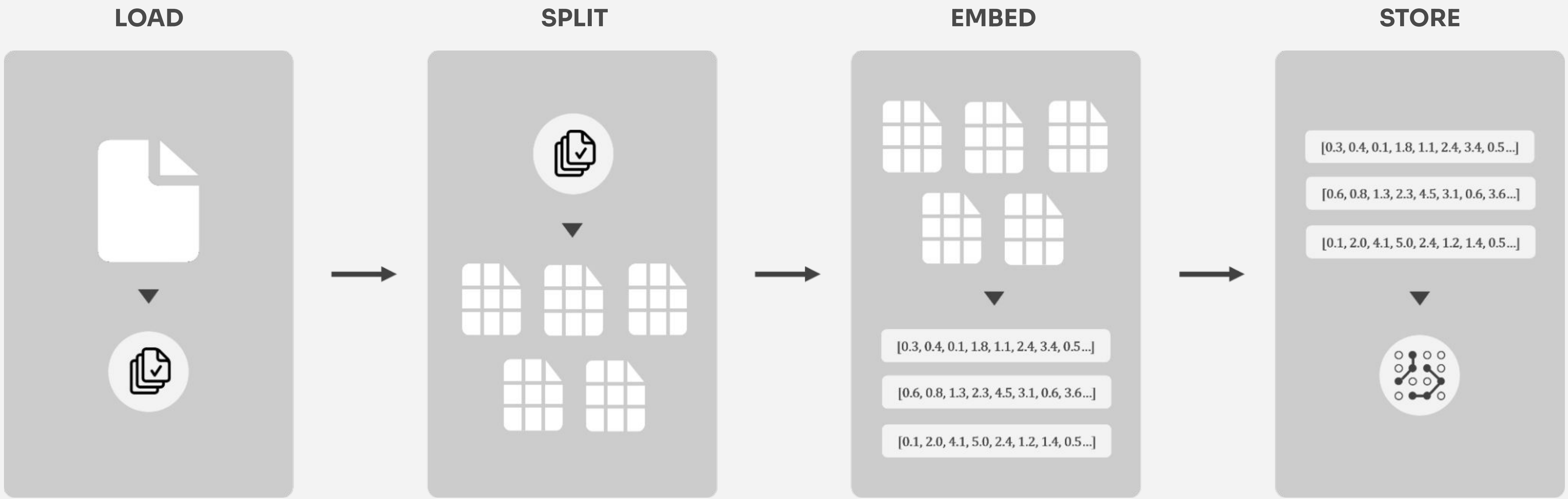
- 232 pages
- Divided into 10 Chapters,
  123 Articles and 17 Annexes
- Approximately 100.000 words

Main Themes

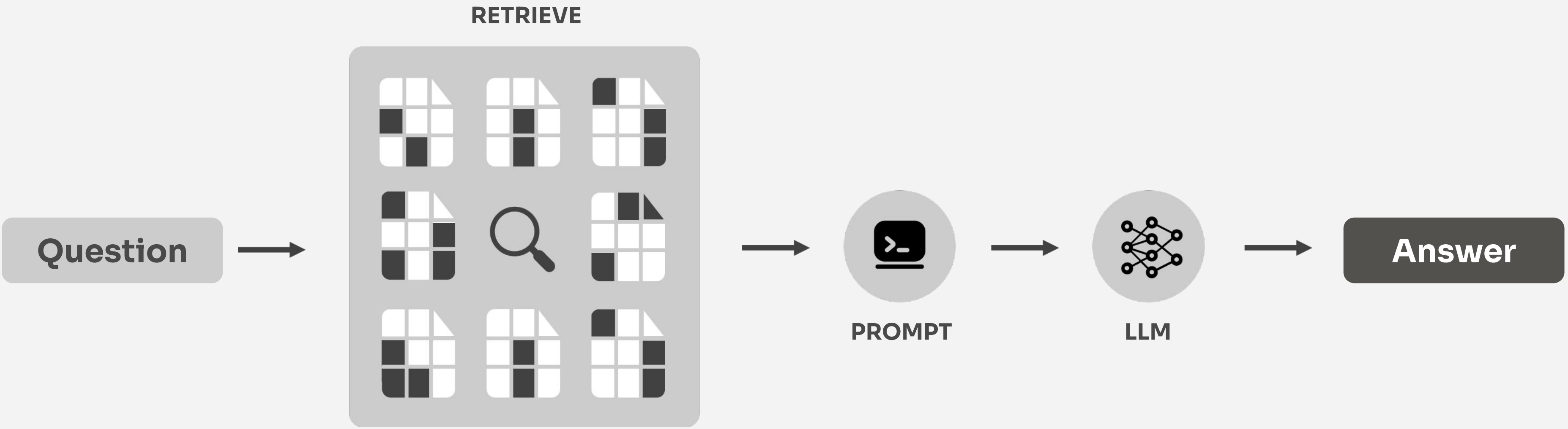
- Placement of medical devices on the market
- Standards for quality and safety in device development

Development steps

Indexing


LOAD
SPLIT
EMBED
STORE
[0.3, 0.4, 0.1, 1.8, 1.1, 2.4, 3.4, 0.5...]
[0.6, 0.8, 1.3, 2.3, 4.5, 3.1, 0.6, 3.6...]
[0.1, 2.0, 4.1, 5.0, 2.4, 1.2, 1.4, 0.5...]
[0.3, 0.4, 0.1, 1.8, 1.1, 2.4, 3.4, 0.5...]
[0.6, 0.8, 1.3, 2.3, 4.5, 3.1, 0.6, 3.6...]
[0.1, 2.0, 4.1, 5.0, 2.4, 1.2, 1.4, 0.5...]

Retrieval and generation



Indexing

Split the document

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import PDFPlumberLoader

loader = PDFPlumberLoader("document.pdf")
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200, add_start_index=True
)
pages = loader.load_and_split(text_splitter)
```

Generate and store the embeddings

```
from langchain_chroma import Chroma
from langchain_google_genai import GoogleGenerativeAIEmbeddings

embeddings = GoogleGenerativeAIEmbeddings(
      model="models/text-embedding-004",
      task_type="retrieval_document"
)
vectorstore = Chroma.from_documents(documents = pages, embedding = embeddings)
```

Retrieval

Retriever

Retrieve information to augment the input prompt with additional context prior to calling the LLM.

- **Search type:** Query the vector database for embeddings that are semantically similar to those on the input prompt.
- Based on those results, **top-k** relevant document text is retrieved.

```
retriever = vectorstore.as_retriever(
      search_type="similarity",
      search_kwargs={"k", 5}
)
```

LLM

Gemini 1.5 Flash - Google Gemini's fastest multimodal model

- 1 million token context window
- 1 million TPM (tokens per minute)
- 15 RPM (requests per minute)
- 1,500 RPD (requests per day)

```
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash")
```

Generation

Prompt

```python
from langchain_core.prompts import PromptTemplate

template = """Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
Use three sentences maximum and keep the answer as concise as possible. Mention in which
pages the answer is found.

Context: {context}

Question: {question}

Helpful Answer:"""

prompt = PromptTemplate.from_template(template)
```

RAG Chain

```python
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

def format_docs(docs):
    formatted_docs = []
    for doc in docs:
        page_number = doc.metadata["page"] + 1
        content_with_page = f"Page {page_number}:\n{doc.page_content}"
        formatted_docs.append(content_with_page)
    return "\n\n".join(formatted_docs)

rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
```

Invoke

This is a basic invocation of the RAG chain.

Improvements will be discussed:

- Show user the chunks of the document retrieved
- Include chat history

```
def process_input(user_input):
    answer = rag_chain.invoke(user_input)
    return answer
```

Chat history

LangChain's memory management features were used.

- All previous messages are kept in the chat history
- Chat history is sent to the model with updated context

```
rag_chain = (
    {
        "context": question_runnable | retriever | format_docs,
        "question": question_runnable,
        "chat_history": chat_history_runnable,
    }
    | question_answering_prompt
    | llm
    | StrOutputParser()
)
```

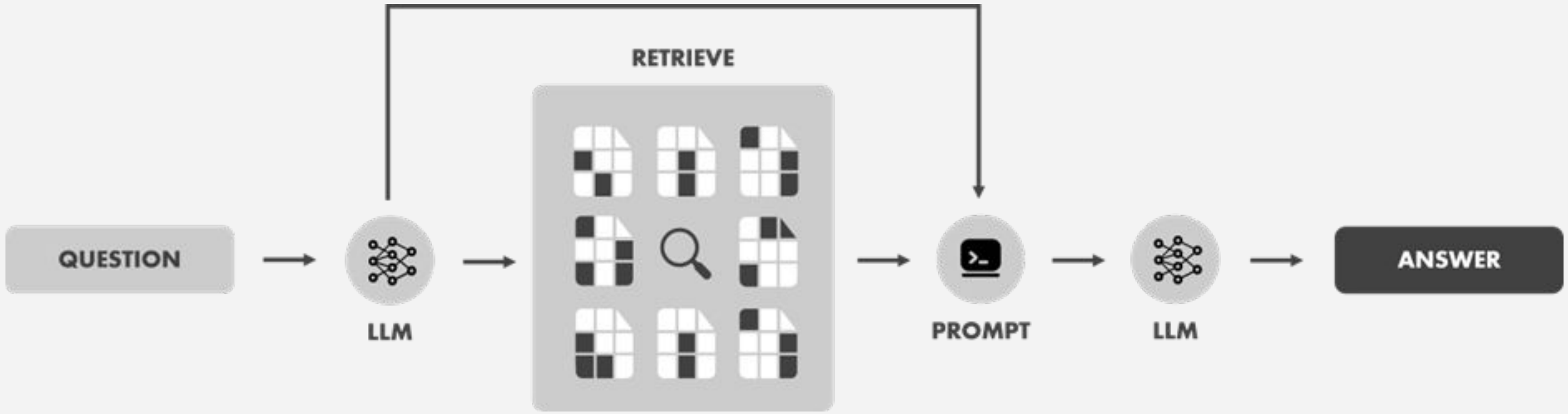
```
chat_history.add_user_message(user_input)
response = rag_chain.invoke(
    {
        "question": user_input,
        "chat_history": chat_history.messages
    }
)
chat_history.add_ai_message(response)
```

Improving basic RAG

Basic RAG: query database directly with user question

Improvements:

- Ask LLM to generate a question to query the database
- Allow LLM to filter sources by metadata



Improving basic RAG

Prepare prompt and metadata filters

```
query_with_history_template = """
(...)
<history>
{chat_history}
</history>
(...)
Question: {question}
"""
retriever_prompt_template = PromptTemplate.from_template(
    query_with_history_template
)
metadata_field_info = [
    AttributeInfo(
        name="page",
        type="int",
        description="The page number of the document",
    )
]
```

Set up retriever and chains

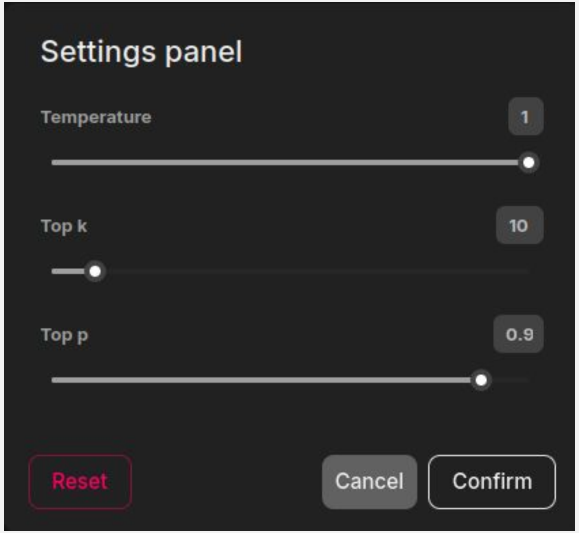
```
retriever = SelfQueryRetriever.from_llm(
    llm,
    vectorstore,
    document_content_description,
    metadata_field_info,
)
retriever_chain = retriever_prompt_template | llm | StrOutputParser()
rag_chain = question_answering_prompt | llm | StrOutputParser()
```

Invoke chain

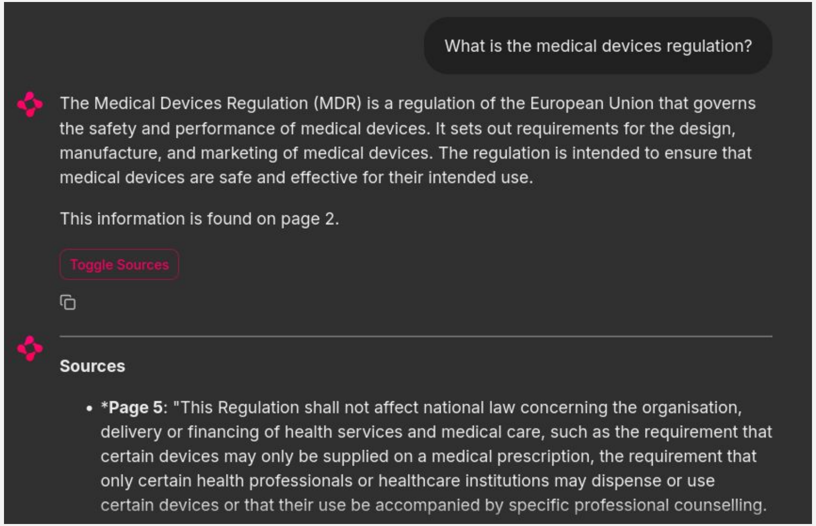
```
def invoke_rag_chain(user_input):
    llm_db_question = retriever_chain.invoke({
        "question": user_input,
        "chat_history": formatted_chat_history()
    })
    docs = retriever.invoke(llm_db_question)
    return rag_chain.invoke({
        "context": docs,
        "question": llm_db_question,
        "chat_history": chat_history.messages
    })
```

Interface

LLM Configuration



Show sources



Evaluation metrics

Benchmarks would provide the most accurate and objective way to assess the system's performance. Unfortunately, there are no benchmarks for the chosen document.

Alternatively, we explored:

- Parameter Tuning
- Model Comparison
- Prompt Tuning

Conclusions