NLP CHALLENGE

Deep Learning PE Assignment 2

Emalisa Antonioli, Jeffrey Stynen, Emma Schoofs

12 december 2024

Contents

1 Introduction

In this document, we provide a comprehensive guide detailing the process of creating a *personalized learning assistant* using *CrewAI*, *Streamlit* and, *Groq*. This assistant is able to take in a PDF documents, and is able to answer questions about its contents. During the development process, we took some inspiration from Akansha Saxena[1] to speed up the process. This report not only outlines the development journey but also delves into the challenges encountered along the way. We encountered several bugs during the implementation, and this document highlights these obstacles, offering insights into how we approached and resolved them. By the end of this report, you'll gain a deeper understanding of the technical and problem-solving aspects involved in building this tool.

2 Streamlit

We utilized Streamlit as the front-end framework for our application, which made it straightforward to build a fully functional chat interface. Streamlit's simplicity and built-in components allowed for rapid development and customization.

When the app is launched, the assistant is initialized with an instance of the Rag class, and an empty list is created to track the message history. This ensures the system is ready for new interactions.

```
st.session_state["assistant"] = Rag()
st.session_state.messages = []
```

To support file uploads, we used the *st.file_uploader* function. This built-in Streamlit component makes it easy to upload files, providing arguments for customization such as allowed file types and the option to upload multiple files simultaneously.

```
uploaded_files = st.file_uploader(
  "Upload the document",
  type = ["pdf"],
  key = "file_uploader",
  accept_multiple_files=True,
)
```

This setup provides a seamless way for users to upload PDF documents, which can then be processed and added to the vector database for further interaction.

3 CrewAI

This section will guide you through the complete *CrewAI* setup process, including the challenges and bugs we encountered along the way. The setup covers the necessary configurations, tools, and the initialization of the crew itself.

3.1 The Crew

Before configuring the agents and tasks in detail, we established the overall structure of the crew by implementing the *PersonalizedLearningAssistant* class. This class leverages *CrewAI* to orchestrate the workflow, ensuring a seamless process for content ingestion and

question answering. By encapsulating the configuration and initialization within this class, we achieved a modular and maintainable setup.

The crew consists of two agents — one for content ingestion and another for question answering — each paired with specific tasks. These agents and tasks are defined using configurations stored in separate files, enhancing flexibility and maintainability. The *agents.yaml* file contains settings for each agent, while the *tasks.yaml* file defines the tasks they perform. This separation allows changes to agent behaviour or task definitions without modifying the core code.

The class defines a content ingestion agent and a question answering agent using the *@agent* decorator. The content ingestion agent is responsible for processing user-uploaded files with the help of a language model and the *ProcessFile* tool. Similarly, the question answering agent handles user queries using the language model and the *ProcessInput* tool. Both agents operate in verbose mode, providing detailed logs for easier debugging and transparency.

Tasks are defined using the *@task* decorator. The content ingestion task directs the content ingestion agent to process and store documents in the vector database. The question answering task instructs the question answering agent to retrieve relevant data and generate responses based on the ingested content.

These agents and tasks are integrated into a cohesive workflow within the *crew* method, annotated with the *@crew* decorator. The method creates a Crew object, combining the agents and their respective tasks. The tasks are executed sequentially, ensuring that content ingestion occurs before question answering. The verbose mode (verbose=True) provides detailed execution logs, making it easier to track progress and troubleshoot issues.

The implementation of the *crew* method is shown below:

```
    @crew
    def crew(self) -> Crew:


        crew = Crew(
            agents=[self.content_ingestion_agent(), self.
    question_answering_agent()],
            tasks=[self.content_ingestion_task(), self.
    question_answering_task()],
            process=Process.sequential,
            verbose=True
        )
        return crew
```

This design supports a structured workflow for ingesting user-uploaded files, retrieving relevant information, and generating context-aware responses, ensuring a streamlined and efficient use of *CrewAI*.

3.2 Configuration

To enable our crew to function effectively, we first defined the various agents and tasks that the crew would handle. These agents are designed to execute their assigned tasks and communicate with each other seamlessly. For the crew to work, each agent requires a role, a backstory, and a goal.

The first agent we configured was the researcher. Since our objective is to extract meaningful insights from the uploaded files, we provided the researcher agent with a professional background. This background ensures that the agent performs thorough information gathering, leaving no detail overlooked. The researcher's primary goal is straightforward: to collect relevant information and store the retrieved data in a vector database.

The second agent we configured was the question-answering agent. Similar to the researcher, this agent was assigned a specific background and goal. Our aim was for this agent to deliver clear and easy-to-understand answers based on the information retrieved by the researcher. Therefore, the goal of the question-answering agent is to pull relevant data from the vector database created by the researcher agent and use it to respond effectively to queries.

With both agents defined, the next step was to assign them tasks. Each task was configured with a description, an expected output, and a designated agent to execute it.

The researcher agent was assigned a content ingestion task. This task requires the agent to process the content of uploaded PDF files and convert it into a format suitable for addition to the vector database. The expected output is a file that can be successfully integrated into the vector database.

The question-answering agent was assigned a question-answering task. This task aligns with the agent's primary goal. The task description specifies that the agent should provide a relevant answer to any received question by utilizing the RAG (Retrieval-Augmented Generation) tool. The output of this task is a *Streamlit* response that effectively delivers the retrieved information.

By configuring these agents and tasks, we established a streamlined workflow for personalized information retrieval and query handling.

3.3 Tools

We have defined two primary tools for our crew: *ProcessFile* and *ProcessInput*. These tools play a crucial role in managing the ingestion of user-uploaded files and handling user inputs, ensuring that our system operates smoothly and effectively.

The *ProcessFile* tool is responsible for processing PDF files uploaded by users and integrating their contents into the vector database, which the Retrieval-Augmented Generation (RAG) tool relies on for information retrieval. The process begins by resetting the assistant's state and clearing the message history. This ensures that any previous context or data does not interfere with new interactions, providing a clean slate for the current session.

Once the reset is complete, the uploaded PDF files are processed one by one. Each file's content is temporarily stored to facilitate smooth ingestion into the vector database. Specifically, the content of each file is written to a temporary location on the system using a temporary file. This temporary file acts as a bridge, allowing the assistant to handle the content in a format suitable for vector storage.

After the temporary file is created, the assistant's feed method is called to ingest the file's contents into the vector database. During this step, a status message ("Uploading the file") is displayed to inform the user that the system is processing their file. This ensures transparency and provides feedback on the system's progress. Once the file's contents have been successfully fed into the vector database, the temporary file is deleted to free up storage

and maintain system efficiency.

The code implementation of the *ProcessFile* tool is as follows:

```
    def _run(self):
        st.session_state["assistant"].clear()
        st.session_state.messages = []

        for file in st.session_state["file_uploader"]:
            with tempfile.NamedTemporaryFile(delete=False) as tf:
                tf.write(file.getbuffer())
                file_path = tf.name

            with st.spinner("Uploading the file"):
                st.session_state["assistant"].feed(file_path)
            os.remove(file_path)
```

The *ProcessInput* tool is responsible for generating responses to user queries based on the vector database populated by the *ProcessFile* tool. It allows the assistant to dynamically provide relevant answers during interactions.

The process begins when a user submits a question or command via the chat interface. The input is captured and displayed to the user, and the message is stored in the session state to maintain conversation history. This ensures the system can retain context across interactions.

The assistant then processes the user's input by querying the vector database using the ask method. This method retrieves the most relevant information previously ingested and generates a response. The response is displayed in the chat interface and stored in the session state alongside the user's message, ensuring the conversation flow is preserved.

The code implementation is as follows:

```
    def _run(self):
        if prompt := st.chat_input("What can i do?"):
            with st.chat_message("user"):
                st.markdown(prompt)

            st.session_state.messages.append({"role": "user", "content":
    prompt})

            response = st.session_state["assistant"].ask(prompt)

            with st.chat_message("assistant"):
                st.markdown(response)
            st.session_state.messages.append({"role": "assistant", "content":
     response})
```

4 Retrieval-Augmented Generation

The Rag class serves as the backbone of the Retrieval-Augmented Generation (RAG) pipeline, managing document ingestion, retrieval, and context-aware response generation. Upon initialization, the class sets up an instance of the *ChunkVectorStore* (imported as cvs) for handling document storage, a prompt template for formatting queries, and a language model (specifically, Groq's Llama 8b-8192) for generating responses. The language model is accessed via an API key stored in the environment variables.

To ensure consistent data processing, the class includes a static method *ensure_string*, which converts different data types such as strings, dictionaries, and lists into string format. This method ensures that all data passed through the system can be handled uniformly.

The document ingestion process is managed by the feed method. When a file is uploaded, this method first splits the document into smaller chunks using the *ChunkVectorStore* object. These chunks are then stored in the vector database, which enables efficient retrieval of relevant information later.

```
def feed(self, file_path):
    chunks = self.csv_obj.split_into_chunks(file_path)
    print(f"Chunks created: {len(chunks)}")

    self.vector_store = self.csv_obj.store_to_vector_database(chunks)
    print(f"Vector store initialized: {self.vector_store is not None}")

    if not self.vector_store:
        raise ValueError("Failed to initialize the vector store.")

    self.set_retriever()
    self.augment()
```

After storing the chunks, the *set_retriever* method initializes the retriever by linking it to the *retrieve_from_store* method, which is responsible for querying the vector database and returning relevant results. The augment method then sets up a processing pipeline, or chain, by combining the retriever, the prompt template, the language model, and an output parser. This pipeline retrieves context, formats the input, generates a response, and parses the output for display.

To generate responses, the ask method is used. When a user submits a query, this method queries the retriever for relevant context from the vector database. The context is combined with the user's question and formatted as input for the chain. The chain processes this input and returns a context-aware response. If the vector database has not been populated yet, the method prompts the user to upload a file first.

```
def ask(self, query: str):
    if not self.chain:
        print("Chain is not initialized!")
        return "Chain is not initialized. Please upload a PDF file first
."

    results = self.retriever(query, n_results=5)

    if "documents" not in results or not results["documents"]:
        return "No relevant documents found in the vector store."

    context = self.ensure_string(results["documents"][0])
    query = self.ensure_string(query)

    query_input = f"Context: {context}\nQuestion: {query}"

    response = self.chain.invoke(query_input)

    return response
```

The class also includes a clear method to reset the vector store, retriever, and chain,

allowing the system to be reinitialized as needed. This ensures that previous data does not interfere with new interactions.

Overall, the Rag class provides a streamlined and efficient way to ingest documents, retrieve relevant information, and generate accurate, context-aware responses. By combining document processing, retrieval, and language generation in a cohesive workflow, the class enables dynamic and interactive user experiences.

5 Vector Database

The *ChunkVectorStore* class is responsible for handling document processing and managing storage in a vector database. It uses *ChromaDB* to store document embeddings and allows efficient querying for relevant information. This class supports PDF document loading, splitting documents into manageable chunks, filtering metadata, and storing these chunks in the vector database for retrieval.

Upon initialization, the class sets up a persistent *ChromaDB* client with a specified storage path. It connects to or creates a collection named *my_collection* where the document chunks and their metadata are stored. This ensures that the data persists across different runs of the application.

The *split_into_chunks* method processes PDF files by first loading them using *PyPDFLoader* and then splitting the document into smaller chunks using *RecursiveCharacterTextSplitter*. This method breaks the document into chunks of 1024 characters with an overlap of 20 characters to maintain context between adjacent chunks. To keep metadata relevant and manageable, the method calls *filter_complex_metadata*, which filters the metadata fields to retain only essential information, such as page numbers and authors.

```
    def split_into_chunks(self, file_path: str):
        doc = PyPDFLoader(file_path).load()
        text_splitter = RecursiveCharacterTextSplitter(chunk_size=1024,
    chunk_overlap=20)
        chunks = text_splitter.split_documents(doc)

        chunks = self.filter_complex_metadata(chunks)

        return chunks
```

After chunking and filtering, the *store_to_vector_database* method adds the chunks to the *ChromaDB* collection. This method ensures that each chunk's content is converted into a string if necessary, and validates the metadata by ensuring it is a non-empty dictionary. If the metadata is missing or invalid, a default placeholder (e.g., 'source': 'unknown') is assigned. Each document chunk is assigned a unique identifier (doc_0, doc_1, etc.), and the documents are added to the collection along with their metadata.

```
    def store_to_vector_database(self, chunks):
        documents = [str(chunk.page_content) if not isinstance(chunk.
    page_content, str) else chunk.page_content for chunk in chunks]

        metadatas = []
        for chunk in chunks:
            if not chunk.metadata:
                chunk.metadata = {'source': 'unknown'}  # Default metadata (
    you can adjust this)
```

```
            elif not isinstance(chunk.metadata, dict):
                chunk.metadata = {'source': str(chunk.metadata)}  # Convert non-dict metadata into a dict
            metadatas.append(chunk.metadata)

        ids = [f"doc_{i}" for i in range(len(chunks))]

        self.collection.add(documents=documents, metadatas=metadatas, ids=ids)

        return self.collection
```

The *query_vector_database* method enables querying the *ChromaDB* collection for relevant information. It accepts either a string query or a dictionary containing a question. The method queries the collection for a specified number of results and ensures that the returned documents are valid. If no documents are found, it raises an error to indicate the issue.

```
    def query_vector_database(self, query_input, n_results=2):
        """
        Query the ChromaDB collection. Accepts either a string query or a dictionary.
        """
        # If the input is a dictionary, extract context and question
        if isinstance(query_input, dict):
            query_text = query_input.get("question", "")
        else:
            query_text = query_input  # Default case for string input

        # Ensure we pass only the query text to the vector store
        results = self.collection.query(
            query_texts=[query_text],  # Pass query string properly
            n_results=n_results
        )

        # Ensure documents are returned correctly
        if "documents" not in results or not results["documents"]:
            raise ValueError("No documents found in the query results.")

        return results
```

This class provides the core functionality needed to process, store, and retrieve information from uploaded documents, making it a crucial component of the RAG pipeline.

6 Challenges and Bugs

Throughout the development process, we faced several challenging bugs that required careful troubleshooting and iterative problem-solving. One of the major hurdles was a fundamental misunderstanding in setting up the crew. To overcome this, we initially developed a simplified solution with similar functionality but without agentic AI. Specifically, we used a Streamlit interface that allowed users to upload a PDF and query its contents with assistance from Groq. This intermediate, function-oriented approach made it clearer how the overall task could be broken down and distributed among different agents.

Another recurring issue we encountered was hitting token rate limits. Initially, we took this as a signal to stop working for the day, but the frequency of these limits became an obstacle to progress. Fortunately, switching to a different Groq model resolved this issue and enabled smoother development.

A final significant challenge arose from managing various components of the codebase that required different input types. At first, we coped with this by repeatedly converting between datatypes, but this approach proved more complex than anticipated. The ultimate solution was to refactor the code to minimize the need for conversions. For example, we adjusted the prompt template used in the Rag class to ensure compatibility with other components.

7 Conclusion

This project demonstrated that building a personalized learning assistant using a combination of the Groq API, CrewAI, and Streamlit is both powerful and achievable. These tools, when integrated effectively, enable the creation of a custom LLM capable of document ingestion, retrieval, dynamic question answering, and much more.

However, working with the CrewAI framework posed significant challenges due to its recent introduction and limited available resources for troubleshooting and bug fixing. Despite these difficulties, the experience provided valuable insights into the process of building custom LLMs and revealed how accessible this technology can be with the right tools and frameworks.

Overall, this challenge has opened our eyes to the remarkable potential of generative AI and the relatively straightforward process of constructing a tailored language model. While the complexity of the technology is truly awe-inspiring, this project showed that creating a custom LLM is well within reach for developers willing to engage with cutting-edge frameworks and problem-solving.

A Generative AI

A.1 Emalisa Antonioli

I initially tried to use ChatGPT to help me better understand the concepts that are fundamental to our assignment. This was not useful as it fed me false information and only confused me further. I did end up using ChatGPT to help me understand any error messages I got, which allowed me to more easily debug the code.

A.2 Emma Schoofs

I used ChatGPT mainly to rewrite code and resolve errors in the rag file and the database file. Although his answers were not always valuable or often sent me in circles.

A.3 Jeffrey Stynen

Using ChatGPT was not as effective as I initially anticipated during this project. This was primarily because the *CrewAI* framework is relatively new. Nevertheless, I attempted to

resolve some errors with the assistance of ChatGPT, but these efforts were unsuccessful.

References

[1] A. Saxena. (2024) Documentor: Build a rag chatbot with ollama, chroma streamlit — real-world example. [Online]. Available: https://www.youtube.com/watch?v=3fvRdFuOkhY&ab_channel=AkanshaSaxena