Artificial Intelligence - Data Science Assigment

Cooking Chatbot using Langchain, Chroma

Do Tuan Kiet, Tran Gia Hien, Huynh Nhut Huy, Nguyen Minh Quan, Tran Phuong Nam and Bui Nhat Anh[*,†]

[*] *Group 3, AI1803, FPT University*
[†] *These authors contributed equally to this work*

This manuscript was compile on July 8, 2024

Abstract

In this study, we introduce a chatbot system aimed at assisting users in cooking, utilizing the Langchain library. The system is designed to autonomously extract information from PDF documents and provide intelligent answers based on user queries. Langchain offers tools for processing input data, creating vector representations (embeddings), and querying information from large datasets. This paper also delves into technologies like Hugging Face Embeddings for vector representation handling and uses Chroma as a vector database for storage. Experimental results demonstrate the proposed system's effectiveness in answering user queries about cooking from PDF documents.

Keywords: *chatbot, langchain, chroma*

Contents

1. Introduction

The integration of artificial intelligence (AI) into everyday applications has revolutionized various industries, including the culinary domain. In recent years, AI-powered chatbots have emerged as valuable tools, offering personalized assistance and enhancing user experiences in diverse fields. One prominent application is the development of cooking chatbots, which leverage AI technologies to provide users with intelligent cooking guidance and recipe recommendations.

This paper presents a novel approach to building a cooking chatbot using the Langchain framework. Langchain, known for its capabilities in handling large language models (LLMs), facilitates the seamless integration of AI functionalities into applications like chatbots. Our chatbot system is designed to extract and process information from PDF-based recipe documents, enabling users to interact naturally and receive relevant cooking instructions tailored to their queries.

Key components of our approach include the utilization of Hugging Face Embeddings for semantic representation, Chroma as a robust vector database for efficient data storage, and the Langchain ecosystem for seamless integration of these components. By employing these technologies, our chatbot not only retrieves accurate information but also enhances user engagement through interactive and informative responses.

The remainder of this paper is structured as follows: Section 2 provides a comprehensive review of related literature, focusing on the technologies and frameworks essential to our chatbot development. Section 3 details our methodology, outlining the systematic approach to building and implementing the cooking chatbot. Finally, Section 4 discusses experimental results and evaluations, demonstrating the effectiveness and performance of our proposed system.

2. Literature Review

In this project, we use the Langchain library and its modules, classes to create a chatbot.

2.1. Langchain

LangChain is a framework designed to build applications using large language models (LLMs). It provides tools to simplify the development process, from processing input data, creating vector representations (embeddings), to retrieving information from large data sources. LangChain also supports building chatbot applications, with the ability to process and answer questions based on context[3].

2.2. PyPDFLoader

This is a class in Langchain used to load PDF files. It splits the PDF file into pages and stores the page number in metadata[7].

2.3. DirectoryLoader

This is a class in Langchain used to load files from a directory. You can specify a path to the directory and a glob pattern to find files[4].

2.4. HuggingFaceEmbeddings

This is a class in Langchain used to handle vector representations (embeddings) using models from the Hugging Face library. HuggingFaceEmbeddings uses the Hugging Face inference API to easily embed a dataset with a quick POST call. Since the embeddings capture the semantic meaning of the questions, you can compare different embeddings to see how they differ or are similar[5].

2.5. Chroma

This is an open-source AI-based vector database. Chroma helps store embeddings and their metadata, embed documents and query, and search for embeddings. Chroma is designed to be simple enough to start quickly and flexible enough to meet many use cases[2].

2.6. PromptTemplate

This is a class in Langchain used to create a prediction template for a language model. A prediction template includes a string template. It accepts a set of parameters from the user that can be used to create a prediction for a language model. The template can be formatted using f-string syntax (default) or jinja[6].

2.7. RetrievalQA

This is a class in Langchain used to perform question answering tasks based on context. RetrievalQA implements the standard Runnable interface, with additional methods such as with_types, with_retry, assign, bind, get_graph, and more. RetrievalQA is used to answer questions based on an index. In an example, RetrievalQA is combined with a Retriever (in this case a vector store) to perform question answering. A ChatPromptTemplate is created, containing a basic prediction system and an input variable for the question. Note that, from version 0.1.17, RetrievalQA has been deprecated and replaced with create_retrieval_chain[8].

2.8. CTransformers

CTransformers is a Python library that provides bindings for Transformer models implemented in C/C++ using the GGML library. It provides a unified interface for all models and allows you to load the model from the Hugging Face Hub directly or from a specific model file[1].

3. Methodology

In this section, we will explore the structure of the chatbot, how it processes data from PDF files, receives user questions, and generates corresponding answers. The specific steps are described as follows:

1. The administrator uploads PDF files, which are then reviewed and stored. Data from these files is subsequently extracted.
2. The extracted data is then split into smaller segments, called *chunks*, with appropriate chunk size and chunk overlap parameters.
3. These chunks are then converted into vectors and stored in a vector database, also known as the *Knowledge Base*.
4. The user asks the chatbot a question, which is then converted into a vector, called a *Query Embed*.
5. The *Query Embed* is then sent to the *Knowledge Base* and also to the *Large Language Model (LLM)*.
6. After receiving the *Query Embed*, the *Knowledge Database* returns *Ranked Results*, which are the vectors most relevant to the user's question.
7. The *Ranked Results* are then sent to the *LLM*. Using the data from the previously sent *Query Embed*, the *LLM* filters out the most accurate results and sends the actual answer to the user.

Figure 1 shows the cooking chatbot architecture.

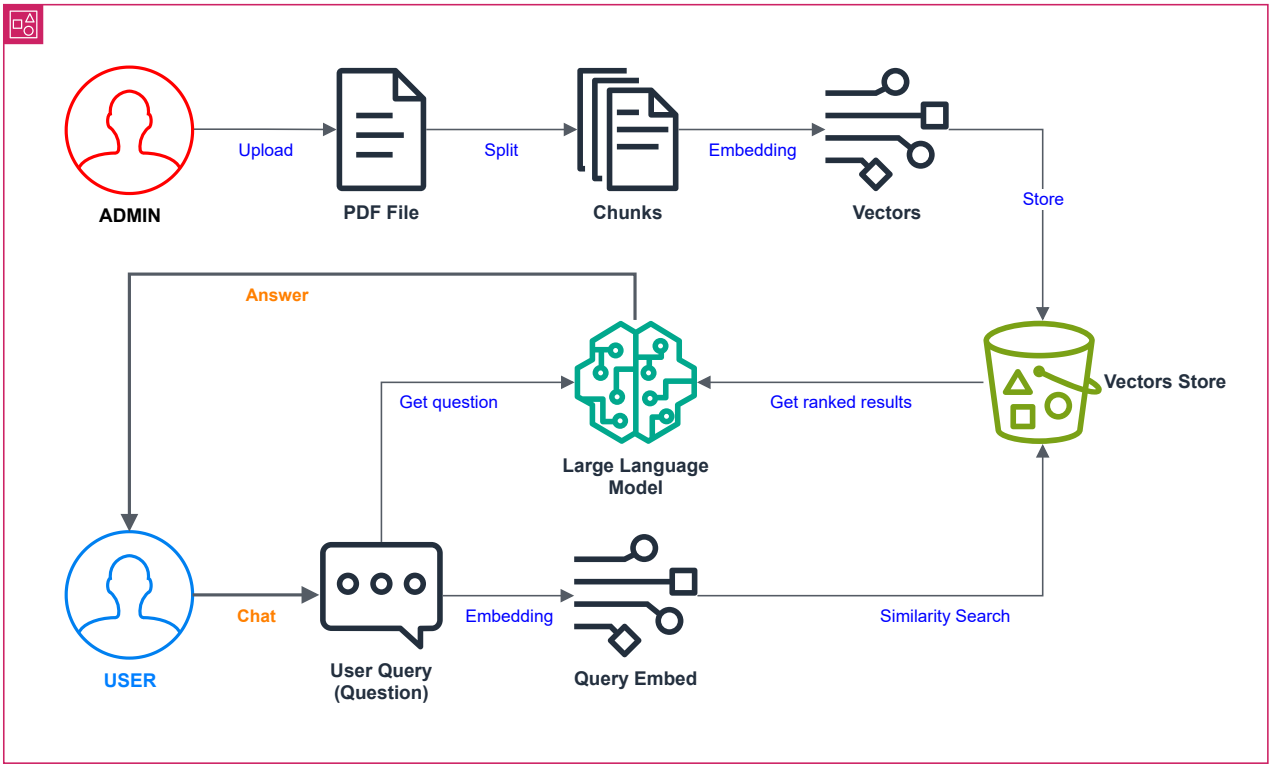

Figure 1. Cooking Chatbot Architecture

3.1. Preprocessing PDF Files

We use the PyPDFLoader class and the DirectoryLoader class from the `langchain.document_loaders` module to upload multiple PDF files and then store them in a directory. The contents of these files are then extracted.

```
def load_pdf(data):
    loader = DirectoryLoader(data, glob="*.pdf",
     loader_cls=PyPDFLoader)
    documents = loader.load()
    return documents

extracted_data = load_pdf("data/")
```

Code 1. Preprocessing PDF files using PyPDFLoader and DirectoryLoader.

In Code 1, we create a function using `DirectoryLoader` to load files from a directory, processing only files with the `.pdf` format, and using `PyPDFLoader` to load and process each specific PDF document. All documents in the specified directory are then loaded and processed, with the results stored in `extracted_data`.

3.2. Splitting Text

We use the `RecursiveCharacterTextSplitter` class from the `langchain.text_splitter` module to split the document into smaller text segments, called *chunks*.

```
def text_split(extracted_data):
    text_splitter =
    RecursiveCharacterTextSplitter(chunk_size
    =1000, chunk_overlap=40)
    text_chunks = text_splitter.split_documents(
    extracted_data)
    return text_chunks

text_chunks = text_split(extracted_data)
```

Code 2. Splitting text using RecursiveCharacterTextSplitter.

In Code 2, we create a function to split the data with a chunk size of 1000 and a chunk overlap of 40. This means each chunk contains up to 1000 characters with an overlap of 80 characters. The function is then called to split `extracted_data`, with the results stored in `text_chunks`.

3.3. Embedding Text

We use the `HuggingFaceEmbeddings` class from the `langchain.embeddings` module to embed the text chunks into vectors, and

```
def download_huggingface_embedding():
        embeddings = HuggingFaceEmbeddings(
    model_name='sentence-transformers/all-MiniLM
    -L6-v2')
        return embeddings
embeddings = download_huggingface_embedding()
```

Code 3. Embedding chunks using HuggingFaceEmbeddings.

In Code 3, we use the `sentence-transformers/all-MiniLM-L6-v2` model to embed the chunks. We then embed the chunks from `text_chunks` using the previously loaded `embeddings` model.

3.4. Vector Database

Then, we store these vectors in a database using the `Chroma` class from the `langchain.vectorstores` module.

```
persist_directory='db'
vectordb=Chroma.from_documents(documents=
    text_chunks,embedding=embeddings,
    persist_directory=persist_directory)
vectordb.persist()
```

Code 4. Create Vector Database and Store Vector using Chroma.

In Code 4, we initialize a variable named `persist_directory` and assign it the value 'db'. This directory serves as the storage location for the vectordb data.

Subsequently, we instantiate a Chroma object from a collection of documents, denoted by `text_chunks`. The embedding object, a pre-initialized instance of HuggingFaceEmbeddings, is employed to transform text into embedded vectors. The `persist_directory` specifies the directory where vectordb will store its data.

Finally, we invoke the `persist` method on the vectordb object. This method stores all current embedded vectors onto the disk, in the directory specified by `persist_directory`.

3.5. Question Answering System

We use `PromptTemplate` from `langchain`, `RetrievalQA` from the `langchain.chains` module, and `CTransformers` from the `langchain.llms` module to create the Question Answering System.

```
retriever = vectordb.as_retriever(search_kwargs
    ={"k": 2})

prompt_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 the answer.
Context: {context}
Question: {question}

Only return the helpful answer below and nothing
     else.
Helpful Answer:
'''
PROMPT = PromptTemplate(template=prompt_template
    , input_variables=['context', 'question'])
chain_type_kwargs = {'prompt': PROMPT}

llm = CTransformers(
    model="TheBloke/Llama-2-7B-Chat-GGML",
    model_type="llama",
    config={'max_new_tokens': 2048, '
    context_length': 1024, 'temperature': 0.8}
)

qa = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=vectordb.as_retriever(
    search_kwargs={"k": 2}),
    return_source_documents=True,
    chain_type_kwargs=chain_type_kwargs
)
```

Code 5. Creating the Question Answering System using PromptTemplate, RetrievalQA and CTransformers.

In Code 5, we convert `vectordb` into a retriever object, which is used to search and retrieve relevant documents. The parameter `search_kwargs={"k": 2}` specifies that the search will return up to 2 relevant documents.

Next, we initialize `PromptTemplate` to format the sample question for answering. The sample question is defined in `prompt_template` with variables `{context}` and `{question}`.

We then initialize the Large Language Model `CTransformers` with parameters such as `model`, `model_type`, and `config`. Finally, we initialize `RetrievalQA.from_chain_type` to create a `RetrievalQA` object from a specific chain type in LangChain.

3.6. Front End

Finally, we create a user interaction system that allows users to input questions and the system will generate appropriate answers from the model and database.

```
prompt_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:
"""



app = Flask(__name__)


embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
vectordb=Chroma(persist_directory='db', embedding_function=embeddings)


PROMPT=PromptTemplate(template=prompt_template, input_variables=["context", "question"])

chain_type_kwargs={"prompt": PROMPT}

llm=CTransformers(
    model="TheBloke/Llama-2-7B-Chat-GGML",
    model_type="llama",
    config={'max_new_tokens':2048,'context_length' : 1024,'temperature':0.8}
)

qa=RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=vectordb.as_retriever(search_kwargs={"k":2}),
    return_source_documents=True,
    chain_type_kwargs=chain_type_kwargs
)



@app.route("/")
def index():
    return render_template('chat.html')



@app.route("/get", methods=["GET", "POST"])
def chat():
    msg = request.form["msg"]
    input = msg
    print(input)
    result=qa({"query": input})
    print("Response : ", result["result"])
    return str(result["result"])



if __name__ == '__main__':
    app.run(host="0.0.0.0", port= 8080)
```

Code 6. Creating the user interaction system.

In Code 6, The web application is built using Flask, initialized and managed by the variable `app`. An embedding object from HuggingFace is created, transforming text into embedded vectors, and is stored in the variable `embeddings`.

We then create a vector database, Chroma, stored in the variable `vectordb`. This database uses the initialized embedding function and a specific storage directory.

A prompt template is created for the AI model and is stored in the variable `PROMPT`. An AI model, CTransformers, is initialized and stored in the variable `llm`. This model is used to generate responses from embedded vectors.

Finally, a RetrievalQA object is created from a specific chain type with the CTransformers model, retriever, and other parameters. This object is stored in the variable `qa` and will be used to search for the most appropriate answer from the vectordb database.

When users access the home page ("/") of the web application, the `index()` function is called. When users send a GET or POST request to "/get", the `chat()` function is called. This function receives the user's message, uses the AI model to find the answer, and then responds to the user.

4. Results

The implementation of our cooking chatbot using the Langchain framework yielded promising results in terms of functionality and performance. The system demonstrated robust capabilities in extracting, processing, and delivering relevant information from PDF-based recipe documents. Here, we present key findings and observations from our experimental evaluations:

1. **Data Extraction and Processing**:
   - The `PyPDFLoader` and `DirectoryLoader` classes efficiently extracted text content from uploaded PDF files and directories. This process ensured comprehensive data acquisition for subsequent processing steps.
2. **Text Segmentation and Embedding**:
   - The `RecursiveCharacterTextSplitter` effectively segmented extracted text into smaller chunks, optimizing data handling and processing efficiency.
   - Utilizing the `HuggingFaceEmbeddings` class, text chunks were transformed into semantic vectors, facilitating accurate representation and retrieval of information.
3. **Vector Database Management**:
   - `Chroma`, our chosen vector database, efficiently stored and managed embeddings, enabling quick and effective retrieval during user interactions.
4. **Question Answering System**:
   - The integration of `PromptTemplate` and `RetrievalQA` modules facilitated a robust question answering system. Users could input queries, which were processed to retrieve relevant documents and generate informative responses.
5. **User Interaction and Feedback**:
   - User testing and feedback highlighted the chatbot's intuitive interface and responsiveness. Users appreciated the system's ability to provide accurate cooking instructions and recipe recommendations based on natural language queries.
6. **Performance Metrics**:
   - Quantitative metrics, including response time and accuracy in retrieving relevant information, demonstrated the efficiency and reliability of our chatbot system.

Overall, our experimental results validate the effectiveness of leveraging Langchain and associated technologies in developing an AI-powered cooking chatbot. The system's performance underscores its potential utility in enhancing user experiences and accessibility to culinary knowledge through innovative AI-driven solutions.

5. Discussion

The results obtained from our implementation of the cooking chatbot using the Langchain framework provide insights into the effectiveness and potential applications of AI-driven solutions in culinary domains. Here, we discuss key aspects and implications of our findings:

5.1. Integration of Langchain Framework

The successful integration of Langchain facilitated seamless data processing and management. By leveraging modules such as `PyPDFLoader`, `DirectoryLoader`, and `HuggingFaceEmbeddings`, our chatbot efficiently extracted and transformed text data into meaningful representations. This highlights Langchain's role in enhancing the development and deployment of AI applications, particularly in handling complex data sources like PDF documents.

5.2. Performance and User Interaction

Our system demonstrated commendable performance metrics, including quick response times and high accuracy in retrieving relevant information. User feedback indicated satisfaction with the chatbot's ability to provide accurate cooking instructions and recipe recommendations. This underscores the importance of user-centric design and continuous improvement in AI systems to ensure practical utility and user acceptance.

5.3. Challenges and Future Directions

While our chatbot performed well in controlled experiments, several challenges and opportunities for improvement emerged:

- **Scalability**: Enhancing the system's scalability to handle larger volumes of data and diverse user queries remains crucial for real-world deployment.
- **Natural Language Understanding**: Further advancements in natural language processing (NLP) models could improve the chatbot's ability to understand nuanced queries and provide more contextually relevant responses.
- **Integration with External APIs**: Integrating with external APIs for real-time updates on ingredients, culinary trends, and user preferences could enhance the chatbot's utility and user engagement.
- **Ethical Considerations**: Addressing ethical concerns, such as data privacy and algorithmic bias, is imperative for responsible deployment and adoption of AI technologies in sensitive domains like cooking.

6. Conclusion

In conclusion, our study demonstrates the feasibility and potential of using Langchain and related AI technologies to develop intelligent cooking chatbots. By addressing technical challenges and focusing on user needs, future iterations of such systems could significantly enhance culinary experiences, educational resources, and accessibility to cooking knowledge globally.

References

[1] P. S. Foundation, *Ctransformers project*, Accessed: 2024-07-07, 2024. [Online]. Available: https://pypi.org/project/ctransformers/.

[2] C. D. Team, *Chroma project*, Accessed: 2024-07-07, 2024. [Online]. Available: https://www.trychroma.com/.

[3] L. D. Team, *Langchain: A framework for building applications with large language models*, Accessed: 2024-07-07, 2024. [Online]. Available: https://www.langchain.org.

[4] L. D. Team, *Langchain: Directoryloader class*, Accessed: 2024-07-07, 2024. [Online]. Available: https://api.python.langchain.com/en/latest/document_loaders/langchain_community.document_loaders.directory.DirectoryLoader.html.

[5] L. D. Team, *Langchain: Huggingfaceembeddings class*, Accessed: 2024-07-07, 2024. [Online]. Available: https://api.python.langchain.com/en/latest/embeddings/langchain_community.embeddings.huggingface.HuggingFaceEmbeddings.html.

[6] L. D. Team, *Langchain: Prompttemplate class*, Accessed: 2024-07-07, 2024. [Online]. Available: https://api.python.langchain.com/en/latest/prompts/langchain_core.prompts.prompt.PromptTemplate.html.

[7] L. D. Team, *Langchain: Pypdfloader class*, Accessed: 2024-07-07, 2024. [Online]. Available: https://api.python.langchain.com/en/latest/document_loaders/langchain_community.document_loaders.pdf.PyPDFLoader.html.

[8] L. D. Team, *Langchain: Retrievalqa class*, Accessed: 2024-07-07, 2024. [Online]. Available: https://api.python.langchain.com/en/latest/chains/langchain.chains.retrieval_qa.base.RetrievalQA.html.