Building a Document Chatbot Using LangChain

2023-10-08

Building a Document Chatbot Using LangChain

In recent times, numerous AI products have emerged that enable users to interact with their private PDFs and documents. But how do these products actually work? And how can you develop your own? Surprisingly, the process is relatively straightforward.

Let's delve into it!

We'll begin with a simple chatbot that can interact with a single document and then progress to a more advanced chatbot capable of interacting with multiple documents and document types. Additionally, it will maintain a record of chat history, allowing users to ask questions in the context of previous conversations.

How Does It Work?

At a fundamental level, how does a document chatbot operate? Essentially, it functions similarly to ChatGPT. With ChatGPT, you can input a block of text as a prompt and then ask ChatGPT to summarize the text or generate responses based on it.

Interacting with a single document, such as a PDF, Microsoft Word, or text file, follows a similar approach. We extract all the text from the document, input it as a prompt to an LLM (Large Language Model) such as ChatGPT, and then pose questions about the text. This process is akin to the aforementioned ChatGPT example.

Interacting with Multiple Documents

Things become more interesting when dealing with very large documents or multiple documents. Sending all the information from these documents to an LLM request is often infeasible due to size restrictions (token limits). Attempting to include excessive information would result in failure.

To overcome this challenge, we must selectively send only relevant information to the LLM prompt. But how do we identify the relevant information within our documents? This is where embeddings and vector stores come into play.

Embeddings and Vector Stores

We require a method to send only the pertinent information from our documents to the LLM prompt. Embeddings and vector stores provide a solution.

If you're unfamiliar with embeddings, don't worry. They may seem unfamiliar at first, but with a little explanation and practical application, their usage will become clearer.

Embeddings allow us to organize and categorize text based on its semantic meaning. We divide our documents into smaller text chunks and utilize embeddings to characterize each chunk according to its semantic meaning. An embedding transformer is employed to convert text chunks into embeddings.

An embedding assigns a vector (coordinate) representation to a piece of text. Consequently, vectors that are close to one another represent pieces of information with similar meanings. The embedding vectors, along with their corresponding text chunks, are stored within a vector store.

Once we have a prompt, we can employ the embeddings transformer to match it with the most semantically relevant text chunks. This allows us to identify related text chunks from the vector store. In our case, we utilize the OpenAI embeddings transformer, which leverages cosine similarity to calculate the similarity between documents and a question.

With a smaller subset of information relevant to our prompt, we can query the LLM using our initial prompt and provide only the pertinent information as the context for the prompt.

This approach circumvents the size limitations of LLM prompts. By utilizing embeddings and a vector store, we selectively pass only the relevant information related to our query, enabling the LLM to provide accurate responses.

So, how do we achieve this using LangChain? Fortunately, LangChain offers this functionality out of the box. With just a few simple method calls, we can get started. Let's begin!

Coding Time!

Interacting with a single pdf

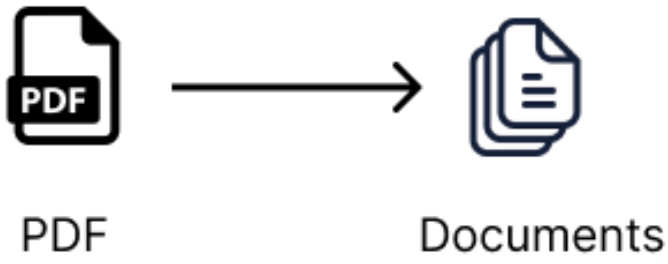
Let us commence by undertaking the processing of a solitary PDF, with subsequent plans to address the processing of multiple documents in due course.

The initial stride entails the formation of a `Document` derived from the PDF. Within LangChain, a `Document` serves as the fundamental class employed by chains to engage with information. Upon examining the class delineation of a `Document`, one observes a rather uncomplicated structure. It encompasses solely a `page_content` method, which facilitates access to the textual content enclosed within the Document.

```
library(reticulate)
use_python("/Users/Shared/anaconda3/envs/py11/bin/python")
```

```
class Document(BaseModel):
  """Interface for interacting with a document."""
  page_content: str
  metadata: dict = Field(default_factory=dict)
```

We use the DocumentLoaders that LangChain provides to convert a content source into a list of `Documents`, with one`Document`per page.



For example, there are DocumentLoaders that can be used to convert pdfs, word docs, text files, CSVs, Reddit, Twitter, Discord sources, and much more, into a list of`Document\'s`which the LangChain chains are then able to work. Those are some cool sources, so lots to play around with once you have these basics set up.

First, let's create a directory for our project:

```
mkdir multi-doc-chatbot
cd multi-doc-chatbot
touch single-doc.py
mkdir docs
# lets create a virtual environement also to install all packages locally only
python3 -m venv .venv
. .venv/bin/activate
```

Then download the sample pdf from here, and store it in the docs folder.

Let's install all the packages we will need for our setup:

```
pip install langchain pypdf openai chromadb tiktoken docx2txt
```

Now that our project folders are set up, let's convert our PDF into a document. We will use the `PyPDFLoader` class. Also, let's set up our OpenAI API key now. We will need it later.

```
import os
from langchain.document_loaders import PyPDFLoader

os.environ["OPENAI_API_KEY"] = "sk-*" # Your openAI API key here

pdf_loader = PyPDFLoader('./docs/NIPS-2017-attention-is-all-you-need-Paper.pdf')
documents = pdf_loader.load()
```

This returns a list of`Document's, one`Document`for each page of the pdf. In terms of Python types, it will return a`List`Document. So the index of the list will correspond to the page of the document, e.g.,`documents[0]`for the first page, `documents[1]`for the second page, and so on.

The simplest Q&A chain implementation we can use is the `load_qa_chain`. It loads a chain that allows you to pass in all of the documents you would like to query against.

In the following sections we will be using openAI API for our tasks. Alternatively you can use any open source model from `HuggingFaceHub`. If so you need to install HuggingFaceHub by running command from terminal:

```
pip install huggingface-hub
```

Then set your HUGGINGFACE_HUB_TOKEN environment variable with python code:

```
import os
os.environ["HUGGINGFACE_HUB_TOKEN"] = "your_api_token"
```

Thn you replace `chain = load_qa_chain(llm=OpenAI())` below with

```
chain = load_qa_chain(llm=HuggingFaceHub(repo_id="google/flan-t5-xxl", model_kwargs={"temperature":0.5,
```

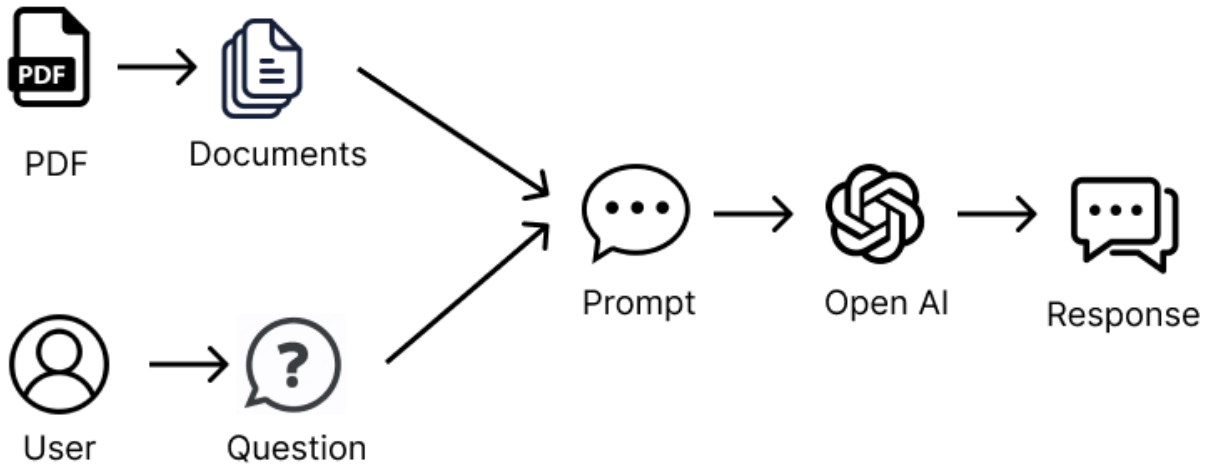
```
from langchain.llms import OpenAI
from langchain.chains.question_answering import load_qa_chain

# we are specifying that OpenAI is the LLM that we want to use in our chain
chain = load_qa_chain(llm=OpenAI())
# Alternatively you can use open source model from HuggingFaceHub
# If so you need to uncomment below:
# chain = load_qa_chain(llm=HuggingFaceHub(repo_id="google/flan-t5-xxl", model_kwargs={"temperature":0.


query = 'What is the pdf about?'
```

```
response = chain.run(input_documents=documents, question=query)
print(response)
```

Now, run this script to get the response:

```
$ python3 single-doc.py
The PDF is titled "Attention Is All You Need" and it is a research paper from the 31st Conference on Neu
```

What is actually happening in the background here is that the document's text (i.e., the PDF text) is being sent to the OpenAPI Chat API, along with the query, all in a single request.



passing all of the text from our source documents into the LLM prompt

Also, the `load_qa_chain` actually wraps the entire prompt in some text, instructing the LLM to use only the information from the provided context. So the prompt being sent to OpenAI looks something like the following:

```
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.

{context} // i.e the pdf text content

Question: {query} // i.e our actualy query, 'What is the pdf about?'
Helpful Answer:
```

That is why if you try asking random questions, like "Where is Paris?", the chatbot will respond saying it does not know.

To display the entire prompt that is sent to the LLM, you can set the`verbose=True`flag on the`load_qa_chain()`method, which will print to the console all the information that is actually being sent in the prompt. This can help to understand how it is working in the working in background, and what prompt is actually being sent to the OpenAI API.

```
chain = load_qa_chain(llm=OpenAI(), verbose=True)
```

As we mentioned at the start, this method is all good when we only have a short amount of information to send in the context. Most LLMs will have a limit on the amount of information that can be sent in a single request. So we will not be able to send all the information in our documents within a single request.

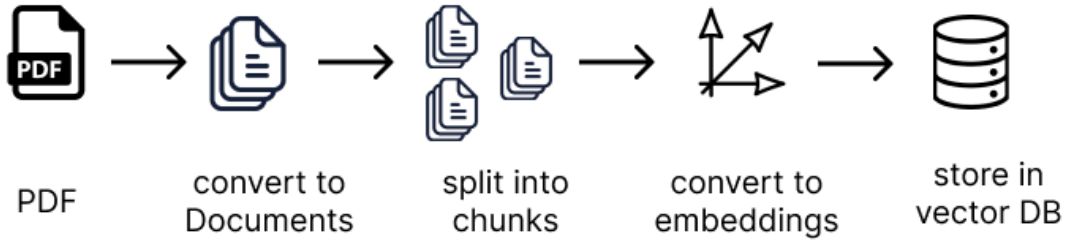
To overcome this, we need a smart way to send only the information we think will be relevant to our question/prompt.

Interacting With a Single PDF Using Embeddings

Embeddings to the rescue!

As explained earlier, we can use embeddings and vector stores to send only relevant information to our prompt. The steps we will need to follow are:

- Split all the documents into small chunks of text
- Pass each chunk of text into an embedding transformer to turn it into an embedding
- Store the embeddings and related pieces of text in a vector store



Let's get to it!

To get started, let's create a new file called‘single-long-doc.py, to symbolise this script can be used for handling PDFs that are too long to pass in as context to a prompt

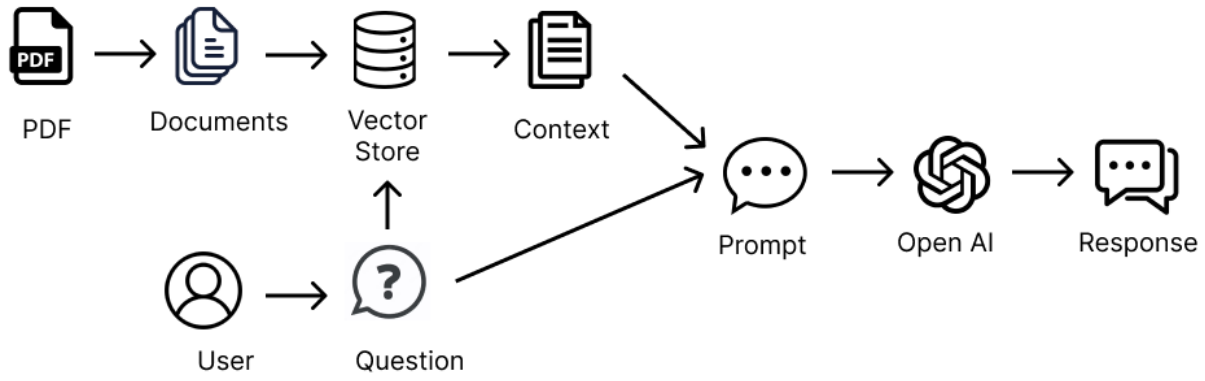
```
touch single-long-doc.py
```

Now, add the following code to the file. The steps are explained in the comments in the code. Remember to add your API key.

```
import os
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings

os.environ["OPENAI_API_KEY"] = "sk-"

# load the document as before
loader = PyPDFLoader('./docs/RachelGreenCV.pdf')
documents = loader.load()

# we split the data into chunks of 1,000 characters, with an overlap
# of 200 characters between the chunks, which helps to give better results
# and contain the context of the information between chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
documents = text_splitter.split_documents(documents)

# we create our vectorDB, using the OpenAIEmbeddings tranformer to create
# embeddings from our text chunks. We set all the db information to be stored
# inside the ./data directory, so it doesn't clutter up our source files
```

```
vectordb = Chroma.from_documents(
  documents,
  embedding=OpenAIEmbeddings(),
  persist_directory='./data'
)
vectordb.persist()
```

Once we have loaded our content as embeddings into the vector store, we are back to a similar situation as to when we only had one PDF to interact with. As in, we are now ready to pass information into the LLM prompt. However, instead of passing in all the`documents`as a source for our context to the chain, as we did initially, we will pass in our vector store as a source, which the chain will use to retrieve only the relevant text based on our question and send that information only inside the LLM prompt.



building a prompt using only relevant information from our document sources

We will use the`RetrievalQA`chain this time, which can use our vector store as a source for the context information.

Again, the chain will wrap our prompt with some text instructing it to only use the information provided for answering the questions. So the prompt we end up sending to the LLM something that looks like this:

```
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.

{context} // i.e the chunks of text retrieved deemed to be moset semantically
          // relevant to our question

Question: {query} // i.e our actualy query
Helpful Answer:
```

So, let's create the`RetrievalQA`chain, and make some queries to the LLM. We create the`RetrievalQA`chain, passing in the vector store as our source of information. Behind the scenes, this will only retrieve the relevant data from the vector store based on the semantic similarity between the prompt and the stored.

Notice we set`search_kwargs={'k': 7}`on our retriever, which means we want to send seven chunks of text from our vector store to our prompt. Any more than this, and we will overuse the OpenAI prompt token limit. But the more info we have, the more accurate our answers will be, so we do want to send in as much as possible. This article provides some nice information on tuning parameters for doc chatbot LLMs.

```
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI
```

```
qa_chain = RetrievalQA.from_chain_type(
    llm=OpenAI(),
    retriever=vectordb.as_retriever(search_kwargs={'k': 7}),
    return_source_documents=True
)

# we can now execute queries against our Q&A chain
result = qa_chain({'query': 'Who are authors of this paper?'})
print(result['result'])
```

Now, run the script, and you should see the result.

```
$ python3 single-long-doc.py
The authors of the paper "Attention Is All You Need" are as follows:

Ashish Vaswani (Google Brain)
Noam Shazeer (Google Brain)
Niki Parmar (Google Research)
Jakob Uszkoreit (Google Research)
Llion Jones (Google Research)
Aidan N. Gomez (University of Toronto)
Łukasz Kaiser (Google Brain)
Illia Polosukhin (Independent researcher, formerly affiliated with Google Research)

Please note that the authors' affiliations mentioned here are based on the information provided in the c
```

Awesome! We now have our document reader and chatbot working using embeddings and vector stores!

Notice we set the`persist_directory`for the vector store to be`./data`. So this is the location where the vector database will store all of its information, including the embedding vectors it generates and the chunks of text associated with each of the embeddings.

If you open the`data`directory inside the project root folder, you will see all of the DB files inside there. Cool right! Like a MySQL or Mongo database, it has its own directories that store all of the information.

If you change the code or documents stored, and the chatbot responses started looking strange, try deleting this directory, and it will recreate on the next script run. That can sometimes help with strange responses.

Adding Chat History

Now, if we want to take things one step further, we can also make it so that our chatbot will remember any previous questions.

Implementation-wise, all that happens is that on each interaction with the chatbot, all of our previous conversation history, including the questions and answers, needs to be passed into the prompt. That is because the LLM does not have a way to store information about our previous requests, so we must pass in all the information on every call to the LLM.

Fortunately, `LangChain` also has a set of classes that let us do this out of the box. This is called the `ConversationalRetrievalChain`, which allows us to pass in an extra parameter called `chat_history`, which contains a list of our previous conversations with the LLM.

Let's create a new script for this, called `multi-doc-chatbot.py` (we will add multi-doc support a bit later on).

```
touch multi-doc-chatbot.py
```

Set up the PDF loader, text splitter, embeddings, and vector store as before. Now, let's initiate the Q&A chain.

```
from langchain.chains import ConversationalRetrievalChain
from langchain.chat_models import ChatOpenAI

qa_chain = ConversationalRetrievalChain.from_llm(
    ChatOpenAI(),
    vectordb.as_retriever(search_kwargs={'k': 6}),
    return_source_documents=True
)
```

The chain run command accepts the`chat_history`as a parameter. So first of all, let's enable a continuous conversation via the terminal by nesting the`stdin`and`stout`commands inside a`While`loop. Next, we must manually build up this list based on our conversation with the LLM. The chain does not do this out of the box. So for each question and answer, we will build up a list called`chat_history`, which we will pass back into the chain run command each time.

```
import sys

chat_history = []
while True:
    # this prints to the terminal, and waits to accept an input from the user
    query = input('Prompt: ')
    # give us a way to exit the script
    if query == "exit" or query == "quit" or query == "q":
        print('Exiting')
        sys.exit()
    # we pass in the query to the LLM, and print out the response. As well as
    # our query, the context of semantically relevant information from our
    # vector store will be passed in, as well as list of our chat history
    result = qa_chain({'question': query, 'chat_history': chat_history})
    print('Answer: ' + result['answer'])
    # we build up the chat_history list, based on our question and response
    # from the LLM, and the script then returns to the start of the loop
    # and is again ready to accept user input.
    chat_history.append((query, result['answer']))
```

Delete the`data`directory so that it recreates on the next run. The responses can act strange sometimes when you change the chains and code setup without deleting the data created from the previous setups.

Run the script,

`python3 multi-doc-chatbot.py`

and start interacting with the document. Notice that it can recognise context from previous questions and answers. You can submit`exit`or`q`to leave the script.

```
multi-doc-chatbot python3 multi-doc-chatbot.py
Prompt: What is the pdf about?
Answer: The PDF is titled "Attention Is All You Need" and it is...
Prompt: What are authors of the paper?
Answer: The authors of the paper "Attention Is All You Need" are...
```

So, that's it! We have now built a chatbot that can interact with multiple of our own documents, as well as maintain a chat history. But wait, we are still only interacting with a single PDF, right?

Interacting With Multiple Documents

Interacting with multiple documents is easy. If you remember, the`Documents`created from our PDF Document Loader is just a list of `Documents`, i.e., a `List[Document]`. So to increase our base of documents to interact with, we can just add more`Documents`to this list.

Let's add some more files to our`docs`folder. You can copy the remaining sample docs from the GitHub repository docs folder. Now there should be a .pdf, .docx, and .txt file in our`docs`folder.

Now we can simply iterate over all of the files in that folder, and convert the information in them into`Documents. From then onwards, the process is the same as before. We just pass our list of`documents'to the text splitter, which passes the chunked information to the embeddings transformer and vector store.

So, in our case, we want to be able to handle pdfs, Microsoft Word documents, and text files. We will iterate over the`docs`folder, handle files based on their extensions, use the appropriate loaders for them, and add them to the'documentslist, which we then pass on to the text splitter.

```python
from langchain.document_loaders import Docx2txtLoader
from langchain.document_loaders import TextLoader

documents = []
for file in os.listdir('docs'):
    if file.endswith('.pdf'):
        pdf_path = './docs/' + file
        loader = PyPDFLoader(pdf_path)
        documents.extend(loader.load())
    elif file.endswith('.docx') or file.endswith('.doc'):
        doc_path = './docs/' + file
        loader = Docx2txtLoader(doc_path)
        documents.extend(loader.load())
    elif file.endswith('.txt'):
        text_path = './docs/' + file
        loader = TextLoader(text_path)
        documents.extend(loader.load())

text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=10)
chunked_documents = text_splitter.split_documents(documents)

# we now proceed as earlier, passing in the chunked_documents to the
# to the vectorstore
# ...
```

Now you can run the script again and ask questions about all the candidates. It seems to help if you delete the `/data` folder after adding new files to the `docs` folder. The chatbot doesn't seem to pick up the new information otherwise.

```
python3 multi-doc-chatbot.py
```

So there we have it, a chatbot that is able to interact with information from multiple documents, as well as maintain a chat history. We can jazz things up by adding some colour to the terminal outputs and handling empty string inputs. Here is a full copy of the script below:

```
import sys
import os
from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import Docx2txtLoader
from langchain.document_loaders import TextLoader
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.text_splitter import CharacterTextSplitter
from langchain.prompts import PromptTemplate

os.environ["OPENAI_API_KEY"] = "sk-XXX"

documents = []
for file in os.listdir("docs"):
    if file.endswith(".pdf"):
        pdf_path = "./docs/" + file
        loader = PyPDFLoader(pdf_path)
        documents.extend(loader.load())
    elif file.endswith('.docx') or file.endswith('.doc'):
        doc_path = "./docs/" + file
        loader = Docx2txtLoader(doc_path)
        documents.extend(loader.load())
    elif file.endswith('.txt'):
        text_path = "./docs/" + file
        loader = TextLoader(text_path)
        documents.extend(loader.load())

text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=10)
documents = text_splitter.split_documents(documents)

vectordb = Chroma.from_documents(documents, embedding=OpenAIEmbeddings(), persist_directory="./data")
vectordb.persist()

pdf_qa = ConversationalRetrievalChain.from_llm(
    ChatOpenAI(temperature=0.9, model_name="gpt-3.5-turbo"),
    vectordb.as_retriever(search_kwargs={'k': 6}),
    return_source_documents=True,
    verbose=False
)

yellow = "\033[0;33m"
```

```
green = "\033[0;32m"
white = "\033[0;39m"

chat_history = []
print(f"{yellow}----------------------------------------------------------------------------------------------------")
print('Welcome to the DocBot. You are now ready to start interacting with your documents')
print('------------------------------------------------------------------------------------------------')
while True:
    query = input(f"{green}Prompt: ")
    if query == "exit" or query == "quit" or query == "q" or query == "f":
        print('Exiting')
        sys.exit()
    if query == '':
        continue
    result = pdf_qa(
        {"question": query, "chat_history": chat_history})
    print(f"{white}Answer: " + result["answer"])
    chat_history.append((query, result["answer"]))
```

The LangChain repository

To understand what is happening behind the scenes a bit more, I would encourage you to download the LangChain source code and poke around to see how it works.

`git clone https://github.com/hwchase17/langchain`

If you browse the source code using an IDE like PyCharm (I think the community edition is free), you can hot-click (CMD + click) into each of the method and class calls, and it will take you straight to where they are written, which is super useful for clicking around the code base to see how things work.

Improvements

When you start playing around with the chatbot, and see how it responds to different questions, you will notice is only sometimes gives the right answers.

Our current method does have limitations. For example, the OpenAI token limit is 4,096 tokens, which means we cannot send more than around 6–7 chunks of text from the vector DB. This means we may not even be sending information from all our documents, which would be important if we wanted to know, for example, a piece of specific information across all documents. For example, one of the documents could be totally missed out, and so we would miss a crucial piece of information.

And we have only a few documents here. Imagine having 100s. At some point, a token limit of just 4096 is just not going to be enough to give us an accurate answer. Maybe you would need to use a different LLM, for example, other than the OpenAI one, where you can have a higher token limit, so you can send more context in it. One of the features you are hearing about these days is LLMs with greater and greater token limits.

If the document source size is too large, maybe training an LLM on your data is the way to go, instead of sending the information via prompt contexts. Or maybe some other smart tweaks could be made to chain parameters or vector store retrieval techniques. Maybe smart prompt engineering or some kind of agents for recursive lookups would be the way to go. This article gives some idea of how you can tweak prompts to give you better responses.

Likely it would be a combination of all these, and the answer might also vary depending on the types of documents you would want to parse. For example, if you choose to focus on a specific document type, for example, CVs, user manuals, or website scrapes, there might be certain optimisations more suited for specific types of content.

Overall, to get a multidocument reader that works well, I think you need to go a little beyond the surface parts of just getting it to work and start figuring out some of these enhancements that could make it a much more capable and useful chatbot.

Summary

So, that's it. We built a single-document chatbot and finished with a multi-document chatbot that remembers our chat history. Hopefully, the article helped to take some of the mystery out of embeddings, vector stores, and parameter tuning on the chains and vector store retrievers.