Build your own RAG and run it locally: Langchain + Ollama + Streamlit

Let's simplify RAG and LLM application development. This post guides you on how to builc your own RAG-enabled LLM application and run it locally with a super easy tech stack.

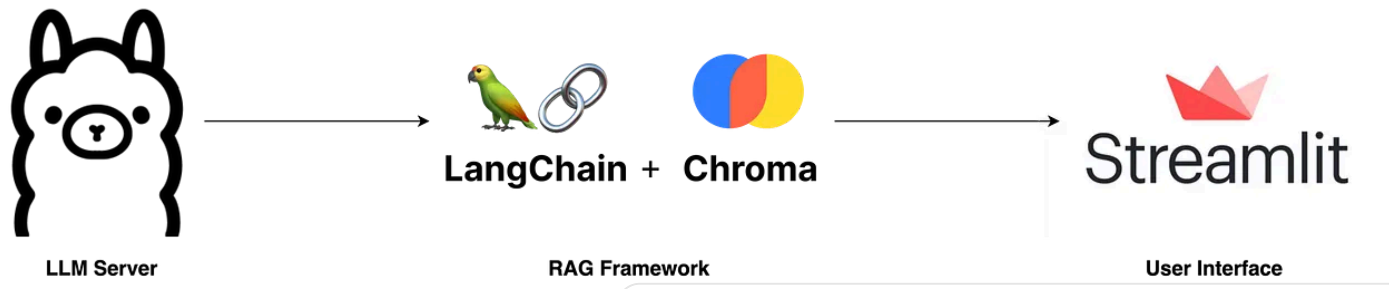
DUY HUYNH
DEC 01, 2023

2 1 Share

With the rise of Large Language Models and its impressive capabilities, many fancy applications are being built on top of giant LLM providers like OpenAI and Anthropic. The myth behind such applications is the RAG framework, which has beei thoroughly explained in the following articles:

To become familiar with RAG, I recommend going through these articles. This post, however, will skip the basics and guide you directly on building your own RAG application that can run locally on your laptop without any worries about data privacy and token cost.

We will build an application that something similar to [ChatPDT]() but simpler. Where users can upload a PDF document and ask questions through a straightforward UI. Our tech stack is super easy with Langchain, Ollama, and Streamlit.



- **LLM Server**: The most critical co to Ollama, we have a robust LLM laptop. While llama.cpp is an opti and run.
- **RAG**: Undoubtedly, the two leadi and LLamIndex. For this project, with it from my professional expe framework is vector storage. We'l with Langchain.
- **Chat UI**: The user interface is als many technologies available, I pre of mind.

Okay, let's start setting it up

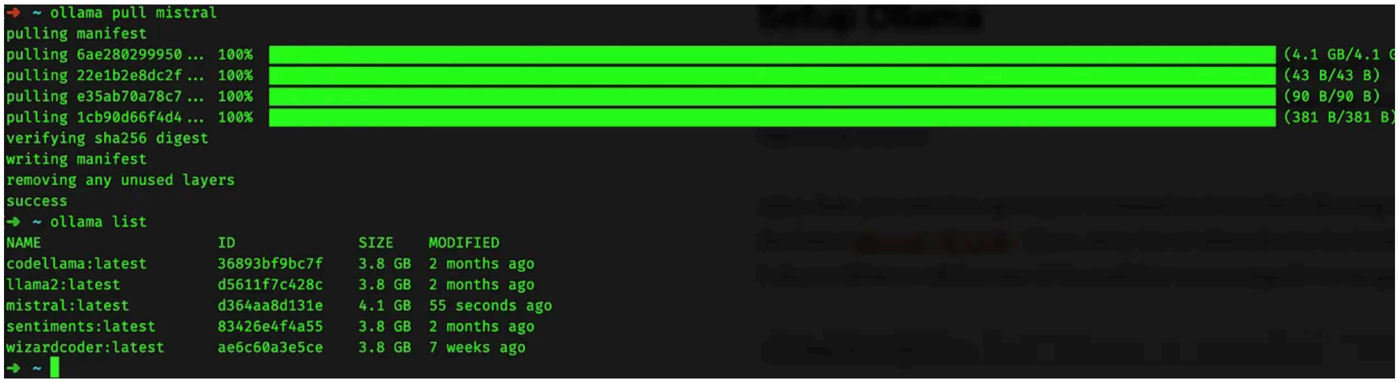
Setup Ollama

As mentioned above, setting up and running Ollama is straightforward. First, visit ollama.ai and download the app appropriate for your operating system.

Next, open your terminal and execute the following command to pull the latest Mistral-7B. While there are many other LLM models available, I choose Mistral-7B fo its compact size and competitive quality.

```
ollama pull mistral
```

Afterward, run `ollama list` to verify if the model was pulled correctly. The terminal output should resemble the following:



```
➜ ~ ollama pull mistral
pulling manifest
pulling 6ae280299950... 100% ████████████████████████████████ (4.1 GB/4.1 G
pulling 22e1b2e8dc2f... 100% ████████████████████████████████ (43 B/43 B)
pulling e35ab70a78c7... 100% ████████████████████████████████ (90 B/90 B)
pulling 1cb90d66f4d4... 100% ████████████████████████████████ (381 B/381 B)
verifying sha256 digest
writing manifest
removing any unused layers
success
➜ ~ ollama list
NAME                ID              SIZE    MODIFIED
codellama:latest    36893bf9bc7f    3.8 GB  2 months ago
llama2:latest       d5611f7c428c    3.8 GB  2 months ago
mistral:latest      d364aa8d131e    4.1 GB  55 seconds ago
sentiments:latest   83426e4f4a55    3.8 GB  2 months ago
wizardcoder:latest  ae6c60a3e5ce    3.8 GB  7 weeks ago
➜ ~
```

Now, if the LLM server is not already running, initiate it with `ollama serve`. If you encounter an error message like `"Error: listen tcp 127.0.0.1:11434: bind: address already in use"` it indicates the server is already running by default, and you can proceed to the next step.

Build RAG pipeline

The second step in our process is to build the RAG pipeline. Given the simplicity of our application, we primarily need two methods: `ingest` and `ask`. The `ingest` metho accepts a file path and loads it into vector storage in two steps: first, it splits the document into smaller chunks to accommodate the token limit of the LLM; second, it vectorizes these chunks using Qdrant FastEmbeddings and store into Chroma. The `ask` method handles user queries. Users can pose a question, and then the RetrievalQAChain retrieves the relevant contexts (document chunks) using vector similarity search techniques. With the user's question and the retrieved contexts, we can compose a prompt and request a prediction from the LLM server.

Source code: [Local rag example (github.com)]

```
from langchain.vectorstores import Chroma
from langchain.chat_models import ChatOllama
from langchain.embeddings import FastEmbedEmbeddings
from langchain.schema.output_parser import StrOutputParser
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.schema.runnable import RunnablePassthrough
from langchain.prompts import PromptTemplate
from langchain.vectorstores.utils import filter_complex_metadata


class ChatPDF:
    vector_store = None
    retriever = None
    chain = None

    def __init__(self):
        self.model = ChatOllama(model="mistral")
        self.text_splitter = RecursiveCharacterTextSplitter(chunk_size=1024,
chunk_overlap=100)
        self.prompt = PromptTemplate.from_template(
            """
            <s> [INST] You are an assistant for question-answering tasks. Use
the following pieces of retrieved context
            to answer the question. If you don't know the answer, just say that
you don't know. Use three sentences
             maximum and keep the answer concise. [/INST] </s>
            [INST] Question: {question}
            Context: {context}
            Answer: [/INST]
            """
        )

    def ingest(self, pdf_file_path: str):
        docs = PyPDFLoader(file_path=pdf_file_path).load()
        chunks = self.text_splitter.split_documents(docs)
        chunks = filter_complex_metadata(chunks)

        vector_store = Chroma.from_documents(documents=chunks,
embedding=FastEmbedEmbeddings())
        self.retriever = vector_store.as_retriever(
            search_type="similarity_score_threshold",
            search_kwargs={
                "k": 3,
                "score_threshold": 0.5,
            },
        )

        self.chain = ({"context": self.retriever, "question":
RunnablePassthrough()}
                      | self.prompt
                      | self.model
                      | StrOutputParser())

    def ask(self, query: str):
        if not self.chain:
            return "Please, add a PDF document first."

        return self.chain.invoke(query)

    def clear(self):
        self.vector_store = None
        self.retriever = None
        self.chain = None
```

The prompt is sourced from the Langchain hub: Langchain RAG Prompt for Mistral. This prompt has been tested and downloaded thousands of times, serving as a reliabl resource for learning about LLM prompting techniques. You can learn more about LLM prompting techniques here.

More details on the implementation:

- `ingest`: We use PyPDFLoader to load the PDF file uploaded by the user. The RecursiveCharacterSplitter, provided by Langchain, then splits this PDF into smaller chunks. It's important to filter out complex metadata not supported by ChromaDB using the `filter_complex_metadata` function from Langchain. For vector storage, Chroma is used, coupled with Qdrant FastEmbed as our embedding model. This lightweight model is then transformed into a retriever with a score threshold of 0.5 and k=3, meaning it returns the top 3 chunks with th highest scores above 0.5. Finally, we construct a simple conversation chain using LECL.

- `ask`: This method simply passes the user's question into our predefined chain anc then returns the result.
- `clear`: This method is used to clear the previous chat session and storage when a new PDF file is uploaded.

Draft simple UI

For a simple user interface, we will use Streamlit, a UI framework designed for fast prototyping of AI/ML applications.

Source code: local-rag-app.py (github.com)

```
import os
import tempfile
import streamlit as st
from streamlit_chat import message
from rag import ChatPDF

st.set_page_config(page_title="ChatPDF")


def display_messages():
    st.subheader("Chat")
    for i, (msg, is_user) in enumerate(st.session_state["messages"]):
        message(msg, is_user=is_user, key=str(i))
    st.session_state["thinking_spinner"] = st.empty()


def process_input():
    if st.session_state["user_input"] and
len(st.session_state["user_input"].strip()) > 0:
        user_text = st.session_state["user_input"].strip()
        with st.session_state["thinking_spinner"], st.spinner(f"Thinking"):
            agent_text = st.session_state["assistant"].ask(user_text)

        st.session_state["messages"].append((user_text, True))
        st.session_state["messages"].append((agent_text, False))
```

```
def read_and_save_file():
    st.session_state["assistant"].clear()
    st.session_state["messages"] = []
    st.session_state["user_input"] = ""

    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.session_state["ingestion_spinner"], st.spinner(f"Ingesting
{file.name}"):
            st.session_state["assistant"].ingest(file_path)
        os.remove(file_path)


def page():
    if len(st.session_state) == 0:
        st.session_state["messages"] = []
        st.session_state["assistant"] = ChatPDF()

    st.header("ChatPDF")

    st.subheader("Upload a document")
    st.file_uploader(
        "Upload document",
        type=["pdf"],
        key="file_uploader",
        on_change=read_and_save_file,
        label_visibility="collapsed",
        accept_multiple_files=True,
    )

    st.session_state["ingestion_spinner"] = st.empty()

    display_messages()
    st.text_input("Message", key="user_input", on_change=process_input)


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

Run this code with the command `streamlit run app.py` to see what it looks like.

Okay, that's it! We now have a ChatPDF application that runs entirely on your laptop. Since this post mainly focuses on providing a high-level overview of how to build you own RAG application, there are several aspects that need fine-tuning. You may consider the following suggestions to enhance your app and further develop your skills:

- **Add Memory to the Conversation Chain**: Currently, it doesn't remember the conversation flow. Adding temporary memory will help your assistant be aware o the context.
- **Allow multiple file uploads**: it's okay to chat about one document at a time. But imagine if we could chat about multiple documents – you could put your whole

bookshelf in there. That would be super cool!

- **Use Other LLM Models**: While Mistral is effective, there are many other alternatives available. You might find a model that better fits your needs, like LlamaCode for developers. However, remember that the choice of model depends on your hardware, especially the amount of RAM you have 💵
- **Enhance the RAG Pipeline**: There's room for experimentation within RAG. You might want to change the retrieval metric, the embedding model,.. or add layers like a re-ranker to improve results.

Finally, thank you for reading. If you find this information useful, please consider subscribing to my Substack or my personal blog. I plan to write more about RAG and LLM applications, and you're welcome to suggest topics by leaving a comment below. Cheers!

Full source code: vndee/local-rag-example: Build your own ChatPDF and run them locally (github.com)

2 Likes

← Previous Next →

Discussion about this post

Comments Restacks

Write a comment...

AjaySh 14 abr 2024

Hi, I have tried this demo, but the search is struck when searching something taking more time is th
any way to this just "loading .... :(.

LIKE REPLY S