Developing a Custom AI Assistant for Secure and Intelligent Document Processing

By Lars Vonk (336028)

Introduction

Over the past few years, the development of generative AI has advanced significantly. Tools like ChatGPT have become commonplace, with many people using them to answer questions based on existing data from public archives. These tools work well for processing vast amounts of information from the internet, which, as we all know, is an enormous and diverse resource. This ability has shown the practical value of large language models in answering questions and generating text based on their training data. The potential use cases are huge. It can answer questions, help you understand new topics by steering the model to your level of understand, it can quickly transform rough, unpolished writing into something more structured and clearer, allowing users to easily refine their thoughts into coherent text and much more.

It has also become quite easy to configure your own model using open-source resources. One example is the Python ecosystem which offers libraries like LangChain for this purpose. The open-source community has contributed significantly, with projects like Meta's LLaMA providing a performant large language model that can be copied, run locally and changed in any way the user wants.

This got me thinking. For several years, I've been using tools often described as "second brain" applications. Platforms designed to simplify the process of writing down and organizing knowledge. I find this idea fascinating because, while the human brain is great at creativity, it isn't always reliable when it comes to holding onto the details of that creativity. Over the past three to four years, I've been taking extensive notes, and an idea struck me: how incredible would it be to access my notes conversationally using a large language model?

I decided to make it happen. I exported my notes into markdown and text files, configured LangChain to match my system's specifications and my informational needs, and configured it to interact with my notes. In this essay, I will explain how I set this up, how it works and what the results are.

Contents

Configuring your own large language model

As mentioned in the introduction, tools like LangChain make it simple to set up and customize your own large language model. LangChain supports various large language model technologies, either through APIs or by running them locally with tools like Ollama. For this project, I decided to run the models locally. This is my preferred option because my notes might include private information, and it's important to ensure that this data isn't shared with external parties. However, using an API can also be a valid option if the stored information isn't sensitive or if your system doesn't have the capacity to run smaller models (which most systems can handle).

The basics

The LangChain website offers many guides covering a wide range of use cases. Setting up a simple model can take fewer than 10 lines of code. For example, one of the guides demonstrates how to configure a model to generate jokes using the ChatGPT API and a preconfigured prompt.

The first step in setting up a model is selecting a model and setting up a prompt, this prompt can then be injected with input from the user (or not). These prompts are also referred to as templates. These templates can take various forms and refining them by changing the text can significantly improve the quality and accuracy of the model's responses.

Selecting a model

One of the easiest ways to run a large language model locally is by using Ollama. Their website (https://ollama.com/search) lists a huge variety of models. The best method I've found is to compare different models using online resources that evaluate performance across various benchmarks and then select the one that fits both your system's specifications and your use case. For example, an 11-billion-parameter model may be excessive for a simple use case or a system without a graphics card, while a model with 1.5 billion parameters could work just fine. It's also a good idea to test a few different models and compare their performance and accuracy to ensure the responses are consistent and coherent.

Loading the documents

After the initial setup, the next step is loading the documents. To provide the correct context, a large store of small pieces of text is accessed. This vast collection of text fragments was initially created by using a text splitter or a recursive character text splitter, which breaks the documents into smaller, manageable pieces (the default characters for recursive splitting are spaces and new lines). These text splitters also use a predefined overlap between each piece of text. This overlap ensures that no pattern is lost by some less-than-ideal splits. It's important to note that this text splitting is done during the initial model creation, not when a question is asked. The splitting process ensures that multiple documents are divided into relevant sections ahead of time. Since the context window is finite, the model can then focus on the most relevant sections when generating responses. This process applies to all the documents in the knowledge base that the model can access at any time.

Vectorisation

Large language models rely on converting text into numerical vectors to better identify patterns and similarities, as this vectorized form is more efficient than processing raw text. This approach not only captures the words themselves but also highlights certain word patterns and categories that are likely to be of interest. To achieve this, both the input and the documents are transformed into vectors using an embedding model. The model recognizes patterns based on its training, identifying combinations of words that frequently appear together in specific contexts. For example, the words "neural," "network," and "artificial" in the same sentence likely point to a topic about AI and machine learning. These recognized patterns are then used to match relevant context from the text to help answer the question.

To match these patterns to the correct context, the previously split text is also converted into vectors using the same logic. These vectors are then stored in a vector store, which functions as a small, locally running, efficient database. The use of database technology is key here, as databases are optimized for quickly querying large datasets. When dealing with numerous small text fragments, the database's ability to efficiently search and retrieve relevant information makes it the ideal tool for matching patterns and providing accurate context.

Second brain applications

Second brain applications address one of the brain’s main limitations: memory. Over time, much of what we learn fades because it isn’t used regularly. While this knowledge can often be recalled by revisiting and reviewing it, it no longer feels immediately accessible. Second brain applications are particularly useful in this context, as they make it simple to jot down notes in a clear and organized way. These notes serve as a personalized reference, allowing you to quickly refresh your understanding when needed. Applications like Notion, Obsidian, Roam Research, and Evernote are good examples of tools that support this approach.

Taking notes has been repeatedly shown to enhance memory, with traditional paper methods often outperforming digital alternatives for retention. However, digital notetaking offers more convenience. The ability to access all your notes on any device and search through them instantly makes it an appealing choice for long-term use. Also, it doesn’t require a huge stack of paper after at least a year of note taking.

Features like cloud synchronization, tagging, and multimedia integration further enhance digital alternatives.

Second brain large language model

These two technologies work together perfectly. The knowledge base created by using a note taking application is a great resource for a large language model set up to run locally. It can use the parts of documents provided to answer questions and provide helpful insights.

This combination brings something new to large language models: the ability to provide responses and advice perfectly tailored to your specific use cases and knowledge level. While there have been attempts at this before, having all your notes as a reference allows the model to truly understand your expertise, what you already know, and what you might need help recalling. For example, if you want to revisit something you previously wrote down, the model can quickly summarize it for you, making it a powerful and personalized tool.

Results

After following the steps and doing my own debugging, I have developed a minimal model that can effectively recite my notes, provide examples I’ve written earlier, and more. While the current implementation is not very user-friendly, I believe improvements can be made. For instance, fetching the document context from the note-taking application via an API could eliminate the need to import all notes into the application. Running the model as a web application would also allow for easier access through the web without requiring advanced setup or running anything locally. The idea shows promise, and I plan to explore running it on a private server in my home network in a future project. This approach would also allow the model to be extended to interact with local devices and process your previously written instructions.
All these steps are included in the model configuration, which can be used through a desktop application built with Electron or called via a shell script I’ve written that requires the web server to run locally. You can find the details on my GitHub.

Source

- https://python.langchain.com/
- https://ollama.com/
- https://www.notion.so/
- https://www.amplenote.com/