AI Project Final Report

Dukarei Abbott

docQuery

Fall 2024

Introduction

I would be willing to wager that a large amount of what we call coding time is not spent on coding. That opening line sounds like the same type of line that gets swallowed up and regurgitated by everyone riding this new LLM wave and claiming that human-dominated coding jobs are over. I believe that for the foreseeable future, even with a rapid advancement in progress toward molding modern AI into such applications, humans will still be critical in oversight and ‘code-manicuring’ roles.

I believe that reading through a good bit of the documentation is a solid approach to getting started on a project. But, for many short-term projects or quick-fix solutions within a broader issue you are working on, it is useful to have a tool that can enhance your knowledge of the docs. That is the idea behind this project. A stand-in for short-term knowledge, and ultimately a gateway toward acquiring that long-term knowledge slightly faster and with less stack overflow headaches.

But as anyone who has tried to have an LLM pre-generate a large chunk of code will know, most of the time you end up slowly re-writing the entire thing yourself. So, for the time being I believe the use of an LLM is to point you in the right direction regarding a particular function or some other small utility, rather than as a ‘plan-of-attack’ tool. Obviously, this type of implementation would work better for this use-case as well(when compared to using a typical chat-trained model), I believe it could be of legitimate use in coding workflows.

At inception, I had planned to use python's web crawling libraries to steal away the code guides from microsoft.learn, then upload them live into the LLM. This would be inconvenient, as it is much better to train your model (embed the vector database), and then keep it that way to be searched repeatedly for any given query.

This is still viable, even with the given method, by simply using a separate model to iteratively search and download the various relevant C/C++ web pages into pdf form for the embedding model to break down and finally for the primary model to search the indexed database. This would basically constitute matching up two separate projects together, which I felt somewhat outside of the scope of what I can do within the spare time I have for this project.

The Project

One peg of my intended fix will be running a somewhat 'locally-tailored' LLM, which is made simplest and easiest using the ollama platform. Ollama allows you to use a single 'model file' to encapsulate all of the necessary environment details required to run a model locally, smoothing the overall process greatly and enabling easy switching back and forth between models for different testing purposes.

Ollama on its own is a really fun and interesting tool, enabling you to pull and run LLMs within your own command line as a REPL. I will be using it in conjunction with python however, to allow 'compile-time' operation of prompts as opposed to 'run-time'.

The purpose of ollama is to locally host your own ‘llm server’. This enables us to run multiple models, and feed their inputs to each other in a chain using our python code. The exact structure of model-chaining I plan to use for this project is known as Retrieval Augmented Generation, or RAG for short. Any basic RAG model is composed of two overarching sub-processes and five individual sub-steps. First step is indexing, where data is loaded, parsed, split into chunks, encoded as vectors and stored into a database for retrieval. This process creates our model’s knowledge reserve. Next up is analysis, where the retriever collects user input and allocates relevant data chunks from the vector database. The LLM is then fed this retrieved data and the input question where it will come up with its best possible response. RAG in a nutshell.

Indexing Systems

-1:Load and parse data to train AI with

-2:Split text into chunks for indexing

-3:Encode text chunks into vectors and store in DB for retrieval

Analysis Systems

-4:Retriever to read user input and collect pertinent data chunks from DB

-5:The phase wherein an answer is generated by feeding the LLM the retrieved data

So, initially I wanted to use spyder for its jupyter notebook-type functionality, allowing you to selectively run certain sections of the code. That would be useful in the scope of this project as I have not currently set up the database to persist, so it is trained and then disposed of after a given number of queries. This would also be useful to allow multiple queries on one run-through of the training, as re-running the program re-embeds the full database from all the pdf files iterated within the given repository.

The original run-through utilized the full python documentation downloaded into pdf form, running from within spyder. This took a full day to train on my CPU as I have an AMD gpu that turned out to be quite difficult to get to work with AI computations. AMD has a software fix in the works, so I checked out the HIP SDK they have released for windows thus far, to no avail.

Apparently you can configure LMStudio to use an AMD gpu locally on windows, but that doesn't exactly work for the RAG/actual developing applications at hand. But, AMD generally has better linux support and this holds true with rocm, as they have released full support and a pytorch package for most of their GPUs over there. So, I moved to WSL to get to work.

During the rocm install process, issues came up with my packages and I was forced to reinstall my wsl to have a fresh start. From here, a few minor linux issues came up but I was overall able to get everything successfully installed. However, when running

ollama models from command prompt I could still only barely see gpu usage reaching three percent. So, I moved to colab. The only slight hurdle in this is getting ollama to run. Simply run all the pip installs as usual but with an exclamation point beforehand, then run the commands below:

Pip install colab-xterm

%load-ext collabxterm

%xterm

Xterm runs a terminal within colab, allowing you to run ollama serve, enabling you to download models and use them in other cells within the colab. I have split the code into sections, making it easy to upload whatever pdf files you want parsed into the home directory on colab, run the first cell which encapsulates steps 1-3 of the RAG pipeline.

Steps 4-5 can be iteratively run after all embedding and chunking is complete, the reason for separating cells after the chromaDB is fully vectorized. I would call that the step-by-step process for running, although you could also simply install ollama locally, pull the models, and run from there if you have a sufficient gpu.

So for a proper step-by-step,

1. Install ollama
2. Run ollama pull for llama 3.1:8b and nomic-text-embed:latest, although any llm and text-embed model will do as long as you replace them in the models section of the code.
3. Pip install the following packages: Lanchain, langchain_core,

langchain_community, pypdf, ollama, chromadb, -qU langchain-ollama

4. Input your chosen pdfs into the same file that is assigned to the sourceDirectory string variable within the code

5. Run the code down to the question bit

6. Iteratively change questions to query the model

You could also optionally set up an environment package and workspace folder if already amidst a complex python ecosystem. Takes about 20 minutes to embed a decently sized pdf running with a T4 gpu on google colab.

Closing Thoughts

I would say the project is a success. General-purpose functionality is cool, but of course less specialized by nature. The axe is never as good as the hammer, until you need an axe. I would say interesting extensions would be saving the databases and allowing separate ones to be pre-trained and loaded up locally, as the embedding process is the most lengthy and computationally demanding.

Additionally, for end-user usage it would be useful to build up an app, but if it were being used in a live workflow its better off in the code editor. Perhaps some command line tooling to allow ‘python3 fileName.py -q “question here”, to input the prompt from there. Many directions to go with future development, but an improved code help buddy was the goal and that is absolutely what I present here.