Carson Crockett: A doc_reader.py Story

Originally copied from a Teams post

This is the reporting I have done on ‘doc_reader.py’ and my attempts at generating supplementary documentation for the RAG pipeline. Below are my tales of woe and success. I encourage you to learn from my failings and revel in my victories. All the files are available in the GitHub, CarsonDev branch

In the same vein as my previous post, I have been working on a method to generate supplementary documentation for our RAG pipeline. I have created a python script to read the COBOL files in a directory and generate a key value pair list of variables from each file with usage summaries for each variable. The end goal for this is an optimistic leap from where the script is now but I hope to improve it throughout the course of this week and later on to help the end product have a better understanding of the data it manipulates. At the moment I have been testing with the "Small.zip" provided at the start of the project to limit the size of the computation.

I've attached the current version of the script for document generation. This is not the script we are using for RAG but hopefully will be incorporated in the future. I'll give a brief walkthrough of the code:

It locates a (currently) hard-coded directory based on the structure of our GitHub to a folder containing COBOL code to be analyzed. It runs a for loop for each file in the directory. For each file we read and copy the content of the file to a local variable. We define a prompt to provide to our model, LLaMa3.2:1b, our prompt currently is "I am going to give you a COBOL file, it's context starts after the character '*'. I want you to look at each section and describe it's functionality to me. If there are any variables tell me how they are manipulated. At the end tell me what you think the purpose of the file is in no more than 10 sentences.\n*\n{content}." In previous versions I have tried to enforce output format but removed that from this prompt to compare the difference. We then use a subprocess to run a command line argument in our environment. This command calls ollama to interface with the model with the provided prompt, replacing {content} with the file content. We then store the result from this command in a new text file and save it.

This goal is to generate new documentation from COBOL code, not exactly a code interpretation but the AI's understanding of the code. As we spoke about earlier I think this approach is essential to having the RAG pipeline better understand each file and be able to retrieve files more accurately. We hope to further develop and break down this logic in the future to fine-tune the output.

Originally I started by having a prompt to extract the variable from the code, this was my original prompt ""Generate a list of all variables used in this COBOL file:\n {content}". This failed in a number of ways. It seemed to not even attempt to use the formatting provided and instead created an output similar to what you would expect normally from an LLM like GPT or LLaMA. It also had several hallucinations when extracting variable names, some were partially correct and some were completely made up. I originally thought the issue was with the prompt so I tried a more complex version: Generate a list of each variables referenced in the following COBOL file. Use the following format (Variable name) | (One sentence summary of variables use): \n {content}". This prompt was slightly better but suffered the same issues. From here I decided the prompt formatting wasn't the issue as much as the task. I assumed that the scope of the prompt was too large to do in one step, similar to what we discussed today, and decided to first have it describe the sections of the COBOL code. This created a much better output describing the major section breakdown of the file, for the smaller files this worked amazingly, relatively speaking at least. Next week I'm going to continue working on this to potentially have multiple prompts be sent sequentially to build off of each other and see what quality output we can generate. Ideally once it is sufficient I will go back to the RAG pipeline and test it's output with these files.

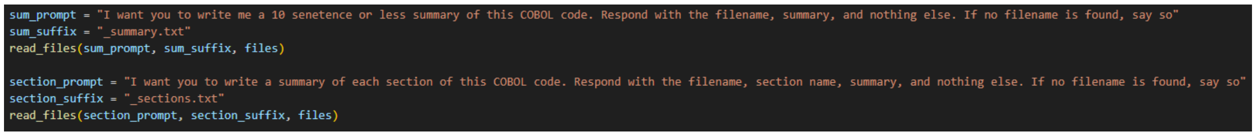
Phase 2

We have moved on to using a HuggingFace model for this code rather than Ollama. This means we have moved up from a 1B to 3B (LLaMa 3.2-3B) model which has generated significantly better results. The output of this model are saved as gen2 on the github and this version of the code is called doc_reader_gen2.py. This method uses panda dataframes and LangChain with HuggingFace to pass a prompt to a model and uses a custom OutputParser to extract the LLM's generated output.

The problem with this model is the computational requirements. I run it using 32G of RAM and 2 A100's and it still takes several minutes to compile 6 output files using medium.zip. This is partially because of the size of the files in that zip but that will be a problem across multiple possible inputs when working with COBOL. This processing time is much to high to use live with a GPT style chatbot to generate outputs but may be usable somewhere on the backend before the user begins querying to give it time to process. I'll continue working to make it run smoother.

Phase 3

I've begun breaking the reader into two parts. I now have it run the reading process twice with two prompts to create two types of file. Either a total file summary or a section summary of the file.



```
sum_prompt = "I want you to write me a 10 senetence or less summary of this COBOL code. Respond with the filename, summary, and nothing else. If no filename is found, say so"
sum_suffix = "_summary.txt"
read_files(sum_prompt, sum_suffix, files)

section_prompt = "I want you to write a summary of each section of this COBOL code. Respond with the filename, section name, summary, and nothing else. If no filename is found, say so"
section_suffix = "_sections.txt"
read_files(section_prompt, section_suffix, files)
```

My hope was to refine the output for each file but this method has made both the total summary and section summary worse or sometimes completely wrong. The output of this version is in the gen3 sub-directory

I am going to work on making the output better, I'm not sure at the moment why it began deteriorating when I explicitly hoped for the opposite of that to happen but hopefully gen4 will be more refined.

Phase 4
In phase 4 I was able to increase performance by separating doc_reader into 2 files. One to make file summaries and one for section summaries. I was also able to use LLaMa 3.1 8B with these files on 32G of RAM and 2 A100's with a bearable runtime.

The output of the summary file was fine. It had a semi-consistent format with decent data. I won't know exactly how correct until benchmarking and comparison which I double I will even be able to do. The section file gave pretty poor output. I'm going to try and read up on prompt engineering and see if I can adjust the prompts being used to create better output

Phase 5
In phase 5 I made some small changes. I recombined it into one file which runs twice. I removed a flag to limit CPU usage to try and improve the quality of outputs. I needed to start using 3 A100's for this change. I also implemented a Torch dataset and dataloader to improve the hardware consumption while running

I also adjusted the prompts to fit more into the LLaMa prompt structure with the system prompt being an instruction as to what the LLM will be doing with queries and the user prompt telling it specifically what to do with each file providing the context

Finally I used the AWS Card Demo file which contains a large amount of smaller COBOL files than I've been working with. The execution time per file was obviously much shorter but the time it took to complete the entire job was 15-20 minutes due to the volume. This is good news however as I thought this would not even be able to complete at all (I did run it with 2 A100's and it still worked. What this means is that the problem we've been encountering is not just hardware issues but its hardware capabilities above a certain file size.

Conclusion

The state the file is in now is about as good as I expect I can get it before the end of the semester. There are currently two files. AWS_doc_reader and doc_reader. The AWS is only different in that it reads files from another directory containing the COBOL code for the AWS carddemo (https://github.com/aws-samples/aws-mainframe-modernization-carddemo). This is a large open source COBOL project AWS released which we used to test our outputs against a system we can understand rather than the black-box that is Michelin's internal mainframe.

The purpose of this file is to generate summary documentation before the RAG pipeline is queried. This documentation is stored where the RAG pipeline can retrieve it during it's retrieval to provide easier access to summary information about certain files. This way it does not have to try and interpret this while also answering the user-query. This is all in an effort to provide more context to the RAG model when answering questions about COBOL when there is no supplementary information.

In the future I would recommend adjusting the hardcoded prompts to create different styles of summary file which may be more useful as we determine what the LLM needs to accurately answer questions. It also works slowly with large files and can sometimes run out of VRAM if a passed file is too large. If there was a way to break a file into chunks and send them as several smaller queries the resource draw would be lower. Maybe you could have one phase break it into sections and create a summary file for each section.