Joyce (Jialing) Pan

CS224G Personal Insight Report

Throughout the project, we developed a Python-oriented assistant which is a LLM-oriented platform that can help Python beginners or non-beginners in Python code generation, summarization, and translation to other programming languages.

Reflecting on our journey, we adhered our sprint goals closely except the lack of resources to fine-tune LLM, and our skills improved along the way. Specifically, we gained proficiency in streamlit which is a user-friendly UI platform and learned how to add backend LLM namely GPT/Claude in the platform using their APIs. We learned different prompt engineering techniques and learned how to implement Retrieval Augmented Generation (RAG) functionality. In addition, we also delved into the structure of LLM architecture and used efficient way to implement an LLM in low-resource setting.

Among all the skills we learned described above, two of the most important are prompt engineering techniques and skills to build a RAG system. Prompt engineering is essential for ensuring stable outputs from LLM. By incorporating step-by-step reasoning, LLM can better understand and deliver the specific answers we seek. This approach is particularly beneficial for unified assistant, tasked with code generation, summarization, and translation. We guided the LLM to determine which task the user wants to perform according to user's prompt and add specifically "please perform {user intent} regarding this prompt: " to the original prompt before feeding into LLM. This not only ensures that the py-assistant is being utilized correctly by the user but also steers the LLM towards generating the expected response. As a result, we apply similar prompt engineering principles across the board, enhancing both our unified chatbot and the individual pages dedicated to specific tasks.

We also explored a lot from Retrieval Augmented Generation (RAG). From research studies, RAG can be used for in-context learning and can produce better output. In essence, RAG operates by sifting through a backend document database, utilizing LLM embeddings to identify important information, which is then fed to the LLM to formulate responses. From code summarization, the backend files are langchain instructions and the relevant information can be retrieved with a high degree of relevant data retrieval accuracy. However, our attempts at code generation encountered challenges. We crawled the codebase from Google Cloud Platform as the backend document database. Code generation with RAG here does not work well here possibly because of three reasons: first, the codebase we crawled is a noisy database with different programming languages and different focuses (such as SQL and multi-modal models etc.), second, the embedding process may not be optimally aligned our purposes, and lastly, Python is one of the mostly used programming language, it is already extensively covered in existing LLMs. Going forward, we will consider testing alternative embedding types and curating a better codebase dedicated to Python.

In summary, we have successfully achieved our main functions of Py-Assistant and learned along. These learnings are foundational and will significantly shape our future projects.