Assignment 3

Lucas Waddell

Part A: Understanding LLM APIs

Q1: What are the advantages of using LLM APIs instead of training your own model?

There are numerous benefits that could come from using an LLM API rather than training and running inference with your own model. The first of course is cost, for both development and infrastructure. Building a state of the art LLM requires significant compute, energy, and infrastructure. For an average person, it is not really feasible to train your own massive model.

Another advantage is that an API allows much easier access to a model that someone else has already put the money into developing. This also means you do not need to worry about any of the maintenance or efforts that are required for running a model with high availability and performance.

Q2: Explain the differences between NLU, NLP, and NLG. How do these capabilities enhance an LLM-powered application?

Natural Language Understanding (NLU): This is how well the LLM can pull meaning from the input. Better NLU means the model can better understand the context and intent behind the query to give improved output.

Natural Language Processing (NLP): This refers to the LLMs ability to analyze and process the textual input. Better NLP means that the model would be better at things like cleaning and processing the input text.

Natural Language Generation (NLG): NLG is the ability of the LLm to produce fluent, relevant and meaningful responses to a query. High quality NLG means the model can produce more engaging output that will absolutely enhance the application.

Q3: Describe three key security best practices when integrating an LLM API into an application.

There are many possible security practices to be considered when integrating an LLM API into an application. The first is the API key. You should treat this key as if it is the most precious secret on earth that cannot be exposed by any means. You should be using proven techniques such as environment variables and strong passwords on any accounts that can generate keys. The keys should be rotated semi-frequently to ensure nothing has been leaked.

The second consideration would be user input. It should be assumed that input could be malicious, and therefore it should be cleaned before it is fed into the model. You can check prompt similarity against known malicious prompts, and always ensure your model is not executing any user code that could have side-effects.

Lastly, you must realize that if you are sending your data to a third-party LLM through an API, it is possible that your data will not be secure. You do not have guarantees as far as exactly what is happening to your data. You can use secure transport protocols to safeguard your data through transmission to the LLM, but once it gets there, anything could happen. Always sanity check to ensure that you are not sending confidential or sensitive information to the model.

Q4: What is rate limiting, and why is it important when working with LLM APIs?

Rate limiting is the practice of enforcing certain limits for the number of actions a user can take in a given time period. For example, we could say that users are only allowed to make 10 queries per minute. We could track this based on account or request IP, and then keep count of what request the user has made in a timeframe.

This of course is not an infallible strategy. Users could have multiple accounts or send requests through proxies or any other number of malicious workarounds.

Q5: How can caching help optimize API usage and reduce costs when using an LLM API?

Caching can help prevent similar queries being sent to the LLM. For example, two separate users could ask what color the sky is. Rather than asking the LLM the same question twice, we could let one user ask it, and save (or cache) the result so that when the second user asks the question we read from our saved queries rather than ask the LLM. There are many ways to implement this. A simple one would be to tokenize and then store the queries and responses you get from the LLM. When sufficient similarity is achieved between a cached tokenized prompt and a new one, we can default to the cached result.

Part C: Reflection

I would not say I faced any major challenges. I checked OpenAI and Anthropic to see if they had a free tier before I settled on Google Gemini. Getting an API key was extremely easy, and the documentation for the new ‘google-genai’ library was decent. It took me a moment to read the code to determine what kind of errors it generated, but overall it was not too difficult.

To ensure security I saved the API key in a .env file, and used ‘python-dotenv’ to load it. Network traffic is already secure due to the TLS provided by the library.

The script already uses streaming to make it seem like the response is being generated sooner. The next optimization I could think of would be implementing a prompt cache layer. With this, I would not need to query the model if there were two input texts to be summarized that were adequately similar. That being said, it would add a pretty significant layer of complexity to this otherwise small project.