NLP (CSC 306)
Winter 2025

Project 3 – Large Language Models

Due: Monday, 2/24, end of the day (Code)
Tuesday, 2/25, end of the day (Report)

In this project, you will make use of a large language model (LLM) server through its public API only. It is possible to build very capable systems by leaning on this resource, especially if you are thoughtful about how you make use of the API to solve your problem. After some exploration of accessing an LLM through its API, you will build a chatbot that talks to people about controversial topics with a goal of broadening their minds.

1 Learning Objectives

- Become familiar with some common design patterns and prompting techniques in black-box use of LLMs.
- Get an introduction to model-based evaluation.
- Learn about some issues in natural language discourse, such as turn-taking, conversational style, the relationships among claims, and properties of successful conversations.

2 Things to Download

Download the starter code from Nexus and unzip it. The resulting folder contains a Jupyter notebook `project_3_llm.ipynb` as well as a number of additional Python files and a `data` directory containing text files.

3 Overview

Open the *folder* `project_3_llm` in VS Code, then have a look at `project_3_llm.ipynb`. The notebook will guide you through the project. Make sure to read all explanations and instructions carefully.

The notebook has seven sections (the first two of which you have already seen in class last Wednesday).

1. Getting started
2. Dialogues and dialogue agents
3. Model-based evaluation
4. Reading the starter code
5. Similarity-based retrieval: Looking up relevant responses
6. Retrieval-augmented generation (Aragorn)

7. Victor

You have to complete seven tasks.

1. Define a simulated human character
2. Evaluate Alice and Airhead
3. Evaluate Akiko
4. Create Aragorn
5. Evaluate Aragorn
6. Create Victor
7. Evaluate Victor

4 Possible strategies for your Victor

4.1 Prompt engineering

A good first thing to do is to experiment with Alice's prompt. The wording and level of detail in the prompt can be quite important. Often, NLP engineers will change their prompt to try to address problems that they've seen in the responses.

Because it's "just" text editing, this won't get full credit by itself unless you make a real discovery. But it requires intelligence, care, experimentation, and alertness to the language of the responses and the language of the prompts. And you'll develop some intuitions about what helps and what doesn't. It is certainly worthwhile.

Of course, people have tried to develop methods to search for good prompts automatically, or semi-automatically with human guidance. So you could additionally try out SAMMO or DSPy – both have multiple tutorials and are downloadable from github.

If you try this, what worked well for you?

4.2 Chain of thought / Planning

The evaluation functions in `evaluate.py` asks each `EvaluationAgent` a "warmup question" before continuing with the real question. That is an example of chain-of-thought (CoT) reasoning, where the LLM is encouraged to talk through the problem for a few sentences before giving the answer. CoT sometimes improves performance.

Instead of using one prompt, could you help an `LLMAgent` argubot (like Alice) do better by having think aloud before it gives an answer? For example, each time the human speaks, your argubot (Victor) could prompt the LLM to think about the human's ideas/motivations/personality, and to come up with a plan for how to open the human's mind.

For example, you might structure this as a `Dialogue` among three participants, like this:

> Victor (to Eve): Do you think COVID vaccines should be mandatory?
>
> Eve: Have you ever gotten vaccinated yourself?
>
> Victor (private thought): I don't know Eve's opinions yet, so I can't push back. Eve might be avoiding my question because she doesn't want to get into a political argument. So let's see if we can get her to express an opinion on something less political. Maybe something more personal ... like whether vaccines are scary.

Victor (to Eve): In fact I have, and so have millions of others. But some people seem scared about getting the vaccine.

One way to trigger this kind of analysis is to present a `Dialogue.script()` to Victor (or to an observer), and ask an open-ended question about it. Or you could ask a series of more specific questions. That is basically what `eval_by_participant` and `eval_by_observer` do. But here the argubot itself is doing it, rather than the evaluation framework.

Eve would be shown only the turns that are spoken aloud. However, when analyzing and responding, Victor would get to see Victor's own private thoughts as well.

4.3 Few-shot prompting

In this homework, often an agent prompted a language model only with instructions. Can you find a place where giving a few examples would also improve performance? You will have to write the examples, and you will have to add them to the sequence of messages that your agent sends to the OpenAI API. See the sentence-reversal illustration earlier in this notebook.

One good opportunity is in the query formation step of RAG. This is a tricky task. The LLM is supposed to state the user's implicit claim in a form that looks like a Kialo claim (or, more precisely, a form that will work well as a Kialo query). It probably doesn't know what Kialo claims look like. So you could show it by way of example. This would also show it what you mean by the user's "implicit claim."

4.4 Parallel generation

The chat completions interface allows you to sample n continuations of the prompt in parallel, as we saw with "the apples, bananas, cherries ..." example. This is efficient because it requires only 1 request to the LLM server and not n. The latency does not scale with n. Nor does the input token cost, since the prompt only has to be encoded once.

Perhaps you can find a way to make use of this? For example, the query formulation step of RAG could generate n implicit claims instead of just one. We could then look for claims in the Kialo database that are close to any of those implicit claims.

Another thing to do with multiple completions is to select among them or combine them. For example, suppose we prompt the LLM to generate completions of the form (s, t, r) where s is an answer, t evaluates that answer, and r is a numerical score or reward based on that evaluation. ("Write a poem, then tell us about its rhyme and rhythm problems, then give your score.") If we sample multiple completions $(s_1, t_1, r_1), ..., (s_n, t_n, r_n)$ in parallel, then we can return the s_i whose r_i is largest.

4.5 Dense embeddings

BM25 uses sparse embeddings – a document's embedding vector is mostly zeroes, since the non-zero coordinates correspond to the specific words (tokens) that appear in the document.

But perhaps dense embeddings of documents would improve Aragorn by reading the text and abstracting away from the words, in a way that actually cares about word order. So, try it!

How? You can use the pre-trained embeddings we have used in class. Or you can get embeddings from Open AI[1]. Or you could figure out how to use OpenAI's knowledge retrieval API[2].

[1]https://platform.openai.com/docs/guides/embeddings
[2]https://platform.openai.com/docs/assistants/tools/file-search

5 Deliverables and Submission

Submit the following files on Gradescope.

- `argubots.py`
- `project_3_llm.ipynb`

If your implementation relies on any additional files that are not part of the starter code, please submit those on Gradescope as well.

Submit your report on Nexus.

- Report in PDF format.
- The LaTeX sources.

Acknowledgments

This assignment is adapted from one developed by Jason Eisner with help from Camden Shultz and Brian Lu.