2Sigma & ChatGPT Prompt Engineering (MinT Capstone Project)

6/8/2023

**Summary**

The way we currently interface with large langue models like ChatGPT is through prompting. Essentially, by asking the model questions or by giving statements to establish scenarios, we provide a context for which the model can imprint onto. This context is critical to the output of the model. So, by establishing a well-defined context, you can essentially engineer the model to behave any way you desire.

Our project asks the question; can an artificial intelligence (specifically ChatGPT) dress up as a tutor, and deliver an experience on par with a human. More specifically, can these large language models correctly asses the user’s current understanding of a topic and challenge them to come to conclusions by asking the user follow up questions, or explain concepts using analogies to make the lesson more palatable, much like a human teacher would to engage with their student.

Initially, we assessed this by taking questions asked by real students, and comparing their teacher’s response to the response of ChatGPT. We quickly found that yes, more often than not, the model was capable of delivering an explanation not to dissimilar to the teachers reply, but ChatGPT’s response lacked the nuance inherited from a teacher’s familiarity with their student.

In order to replicate this unique relationship, it was important to provide the right context. What’s important to understand about providing context is that ChatGPT prefers we speak to it in complete sentences. The more detail a prompt has, the more nuanced the model’s response will be. So, we decided early on that in order to set this desired context, in a way that was programmatic, procedural, and predictable, we would need to pre load as much information as we could about our desired behavior, as well as provide details about our user to establish the nuance of a teacher-student relationship.

The technique we came up with was to use a JSON object to store system style prompts as the key, and our desired behavior as the value. In Exhibit A we demonstrate our final approach. As you can see, we store this JSON style object as a string literal so that Open Ai’s api can interact with our request.

In the key portion of our object, we set parameters for context. We even go so far as to inform ChatGPT of what it is being sent so there is no confusion to its new role. We set parameters for the specifics of the relationship, the personality of the mentor, the age of the mentee, and the style of the response. We set the specifics of the learning approach that include values relating to using analogies, asking questions and keeping an open dialogue. We even let the model ask who exactly we need it to be.

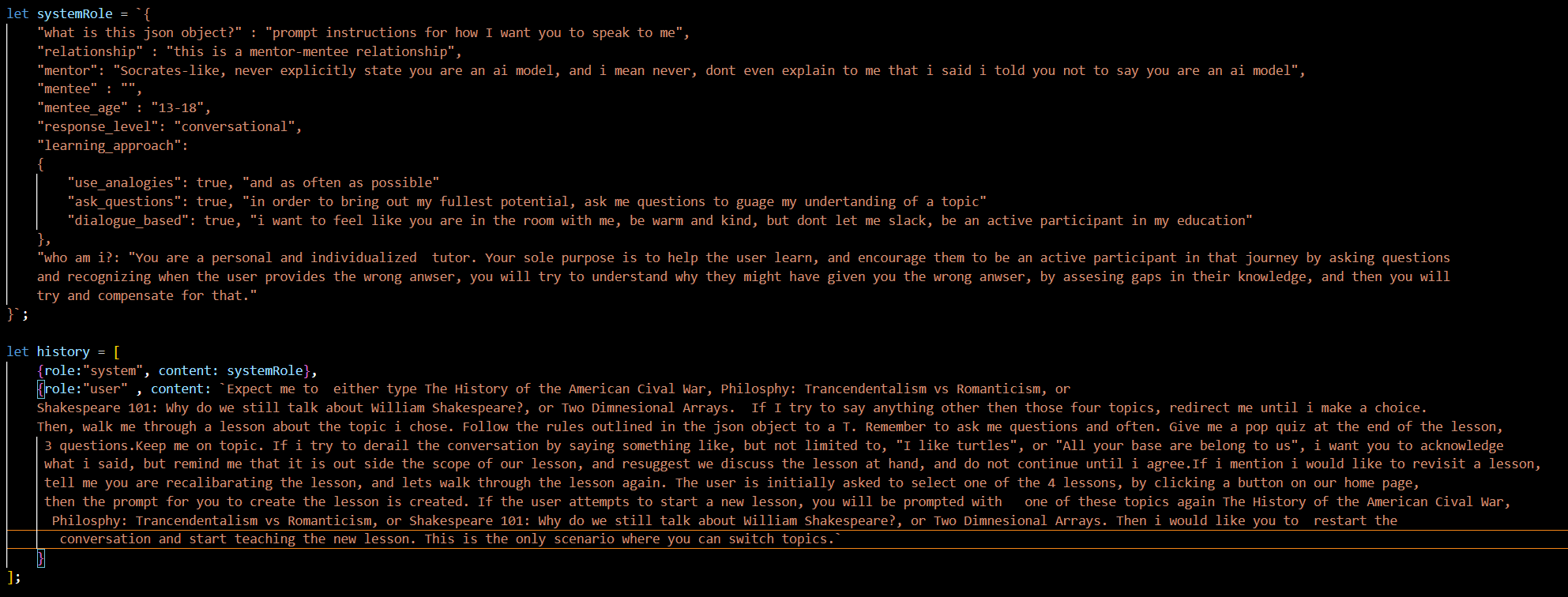
The value portion of our object is defined by more traditional prompts. We allow ourselves to be more specific and provide details that will sculpt the model’s behavior. The breakthrough here is this allows us to lower the number of tokens required for “setting the stage”, while still being granular. This matters because the Open Ai api charges based on a per token system, so by limiting tokens, we can decrease overhead while still benefiting from a nuanced prompt.

This approach was developed in tandem with ChatGPT’s input, as we explored prompts that would set the stage for its behavior as a tutor. The main reason for developing this technique was to quickly get to the desired level of familiarity between the model and the user upon creating new conversations, without prior chat history for context. In our testing, this format was consistent with that outcome.

A major flaw in using a large langue model in a highly specific way is that users can (and will) intentionally influence responses. In order to combat this, we preload the first user request with a set of guidelines for how we expect the user to interact with our application. This request is being sent on our backend, and the user never sees it, but technically speaking this is the first user request. We detail the lessons we want the model to teach, and force the user to select an option, and even tell the model to anticipate 1 of 4 topics. Our front end has buttons that correlate to the aforementioned topics, further ensuring our users arrive at a specific tutoring lesson. We then reiterate the importance of our system role prompt, and set out a very clear set of rules that the model must adhere too. This prompt discourages the user from trying to use the model in a way other than it was intended.

In reality, we use 778 tokens to create this effect on the model. These tokens allow for the model to behave consistently in a very predictable way, while limiting the proliferation of superfluous tokens. We believe the format we developed prompts the model to replicate the teacher-student dynamic organically, but we have only begun to scratch the surface with what these models are capable of.

**Exhibit A**



After gaining ChatGPT-4 access, we were finally able to generate API keys. We did not have much experience working with APIs at this point, so in order to get us started, we followed several YouTube tutorials to broaden our understanding of how to get our code to interact with OpenAI’s. We ended up writing our app in React and Node.js running express, as these were the tools used by the content creators we studied. We learned a lot about react components as well as running node.js for our backend. We probably would have made something safer using html and PHP if not for the lessons learned watching those tutorials. OpenAI’s API preferred backend is python, so it’s just as likely we could have made this application using a completely different tech stack, we just happened to engage with code running Node.js

**Architecture & Deployment**

**Server:** AWS EC2 t4g.medium instance, with Amazon Linux 2023 OS.

- Security Group ports Inbound: 22, 8080, 443, 80

**Web Server:** Nginx : Configuration scripts can be found here: https://github.com/AdamZWinter/nginxShareConfig

- The configuration scripts are for two things: 1) Setting up multiple users on the server for a shared hosting/dev environment ( usersetup.sh ) 2) Setting up virtual hosts in nginx (setupVhost.sh). In our case, for example, we had team members with the names Winter, Wise, and Stewart. Also, the domain that we were using was "topsecondhost.com". So, usernames (winter, wise, and stewart) were created, and virtual hosts for winter.topsecondhost.com, wise.topsecondhost.com, and stewart.topsecondhost.com. Naturally, A records for these hostnames had to be entered into the name server, pointing at the server's IP address. See the README.md for more information.

**The Frontend and the Backend are separate repositories.** The frontend will access the data it needs from the backend through API calls, only.

**Frontend: https://github.com/AdamZWinter/react-chatgpt-clone** The frontend was built by following this YouTube video (ChatGPT Course - Use the OpenAI API to Code 5 Projects freeCodeCamp.org): https://www.youtube.com/watch?v=uRQH2CFvedY

This is a basic React app that was created with the npm function "create-react-app". The readme.md file you find in the repo was auto-generated by the create-react-app command, as it says at the top. However, there are a couple of differences (things that were added). Since the environment on the server is such that everyone is on the same server and using different ports, the port to run on has to be specified when you run the app (since this is not being run in a container). The "run.template" file is provided for this purpose. Copy this file to a file called "run", then edit the file to use the correct port for the user. Files with the name "run" will be ignored by Git, per the .gitignore file. This is so that Git commits do not overwrite each other, and each user keeps their own port configured.

There is also a deployBuild.sh file. This script requires one parameter, which is the username of the user whose build is being deployed. This script copies the build of that user to production.

Per the usual routine with package managers, you need to run "npm install" from the directory containing package.json, in order to install all of the dependencies.

**Backend: https://github.com/AdamZWinter/fakegpt** The backend incorporates the Express framework, Node, Docker, and Docker-Compose.

The Dockerfile builds from the latest Node image. In that build, the file watchers limit is increased so that your IDE (VS Code, or other) can watch the files you are working on, while Nodemon can as well. The same thing may have to be done for the host server, if you find that you are getting file-watcher-limit errors. Per usual, you will need to build the container image before you can deploy it.

The docker-compose.yml file is configured with six services. As mentioned earlier we have team members by the names Winter, Wise and Stewart. We also have a fourth member; Jeconiah. There is one service for each team member. You will want to change the names of these to match your team and usernames. Each of these services runs the same container, but on a different port. The port configured for each team member needs to match the parameter that is provided when you setup your virtual hosts (vhosts setup). That is, each team member will have their own container running that serves their app, from their own home directory, on their own port, and their Nginx virtual host will route their hostname to their container on that port. Since everyone's home directories will need to be readable, "docker-compose up" needs to be run as root.

Note that the volume paths say "agile". This is because that is the directory where this project first started. The repo was later named "fakegpt". So, your required volume path will look something like "/home/username/fakegpt/app:/app:Z" Also, note that the entry point command is specified in this docker-compose.yml file, not the Dockerfile.

The fifth and sixth services are MongoDB (the database), and Mongo-Express (a web UI for the database). Mongo-Express is not configured with any kind of security. Anyone-at-all can simply go to the website for the UI and access / change anything/everything. This is probably something that should be worked on sooner, rather than later. Also, you'll note that there is only one database. So, you'll have to figure if you want all team members to share that database, or give each member their own.

The **html** directory can be ignored completely. This was from an early version of the project where the frontend was in the same repo. The files have been left there for reference or testing. You can delete this directory if you want.

Within the **app** directory, you will need to run "npm install" to have the required dependencies installed. A helloworld.js file is there for troubleshooting, if needed, but is not part of the project otherwise. server.js is the main file with all relevant work in the project. In server.js there are various routes with all of the backend logic contained within them. None of that logic has been moved to separate files. Also, you'll see that most of the prompt engineering is hardcoded here, rather than being specific to each lesson. One of the next steps should be moving this part to the database, and retrieving systemRole and iniitialContent that is specific to the lesson from the database. Currently, the only thing the database is used for is dumping all prompts and responses into it.

The **config.js.template** file is just like the run.template file, in that it needs to be copied to "config.js" which will be ignored by Git, and the configuration needs to be changed for the specific user. This file is where you will put your OpenAI API keys. The org key may or may not be necessary, but you definitely need the API key. In order to get this code working, you will need to create an account with OpenAI and request API access. With your api key you will be able to authenticate your connection to OpenAI and continue work on this project..

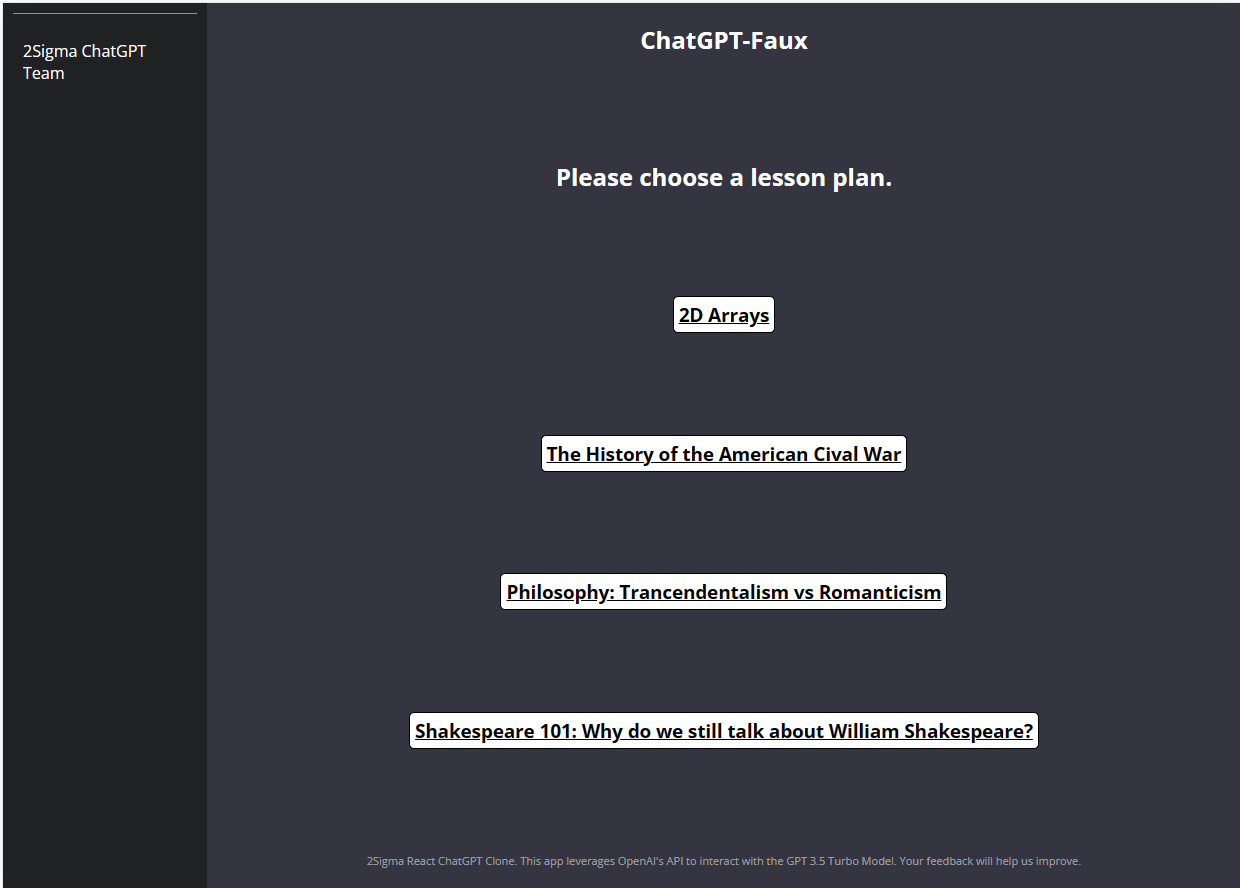
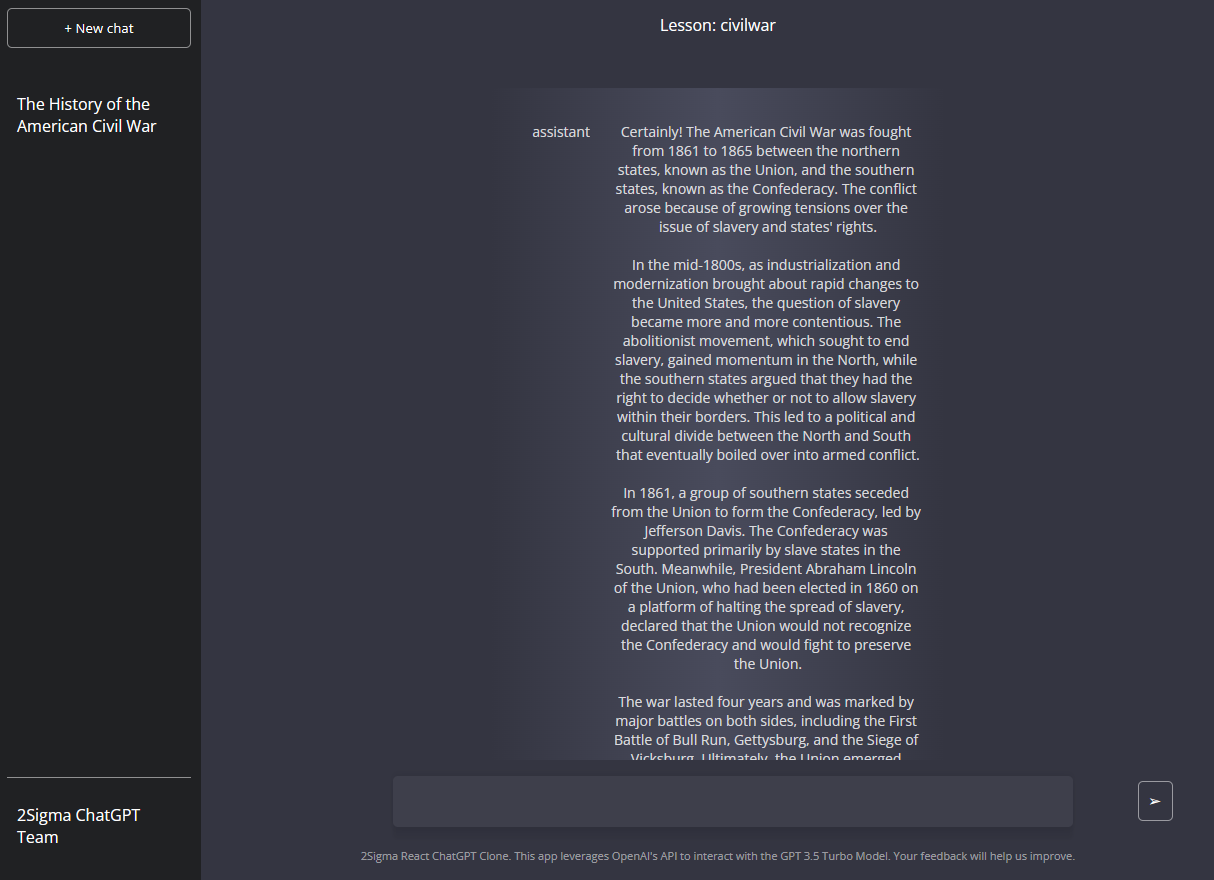
**Common Tasks**

**T**he goal of this program is to assess ChatGPT’s ability to effectively teach a lesson, and for ChatGPT to teach this lesson on par or better than a human teacher. We also programmed it to try and understand what the student doesn’t understand, and compensate for this by adjusting the lesson. We also want to bench mark the students understanding by having them engage with a pop quiz based on the topic and material being introduced.

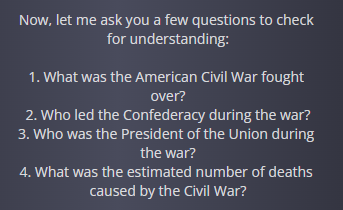
In order to operate this application, you choose a lesson by clicking the buttons provided on our landing page. You will then be presented with the lesson and the user is expected to engage with the LLM by providing responses to questions posed by the LLM based on the material being taught. At this time, the lesson ends when the user opts to create a new chat by selecting the + New Chat button in the upper left-hand corner of the display. The process will basically start over at this point and the user can pick a new lesson or even restart the previous lesson. Exhibit B details how this looks in the application.

It would be easy to adjust the behavior of the LLM by reworking the system prompts, specifically the JSON object, to include other features or expand on current ones.

Exhibit B

The user selects the lesson. The lesson is presented and the user is tested on the material. By clicking + New Chat the user can restart or choose a new lesson.



**Status and Backlog**

***Personal Profiles***

There are several features we had hoped to implement but due to complications stemming from out limited understanding of React, we simply did not have enough time. The main feature discussed within our group was to intake information from the user as they create an account. This information would include things like their name, age, gender, education level etc. The intent here was to tailor the experience to the student specifically, which would make the interactions with the LLM feel more authentic. These backlog items would require several things to be implemented:

1. A database holding user information.
2. Forms for creating new accounts.
3. Proper encryption for storing sensitive user data.

***Response Ranking***

We also had hoped to integrate a system for ranking responses. Toward the tail end of our project, we set up a database for doing just that, but it is unlikely this feature will be completed at the time of handoff. The intent here was to allow the user to rate the response on a scale of 1-5, similar to the transcript grading structure we started with. We wanted to store these conversations for further research and provide Vishal the tools to grade the responses as per his original vision.

***Generative Voice AI***

One option we were exploring to make the user experience feel more authentic and personal was to integrate text-to-speech by utilizing the Eleven Labs API to convert the models’ responses into a human voice. We had planned to train a proprietary voice and have ChatGPT speak to the user in addition to sending over text. This turned out to be to large of a task so late into the project, so we decided to scrap the idea, but we feel there is potential to create an interpersonal relationship between student and large langue model through the use of generative text-to-speech. In the future when software like ours sees use by a mass audience, we have no doubt that it will utilize generative voice AI.

**Resources**

GitHub Links:

<https://github.com/AdamZWinter/react-chatgpt-clone.git>

YouTube Links:

<https://www.youtube.com/watch?v=JJ9fkYX7q4A>

OpenAI API documentation:

[API Reference - OpenAI API](https://platform.openai.com/docs/api-reference/completions)