

AI-Powered Text Completion: A Capstone Project

Introduction

This project aimed to acquire experience with Generative AI by building and experimenting with a simple text completion app using Python and an API call. Currently, Generative AI models answer questions and even finish our sentences, capabilities that are already reshaping content creation. I built a lightweight text-completion application in Python that calls OpenAI's GPT-3.5 model via its API to understand how Generative AI processes input prompts to generate coherent and relevant text for the user. Through experimentation with prompt design and model parameters to evaluate output quality, I created an application to finish phrases using the user prompts.

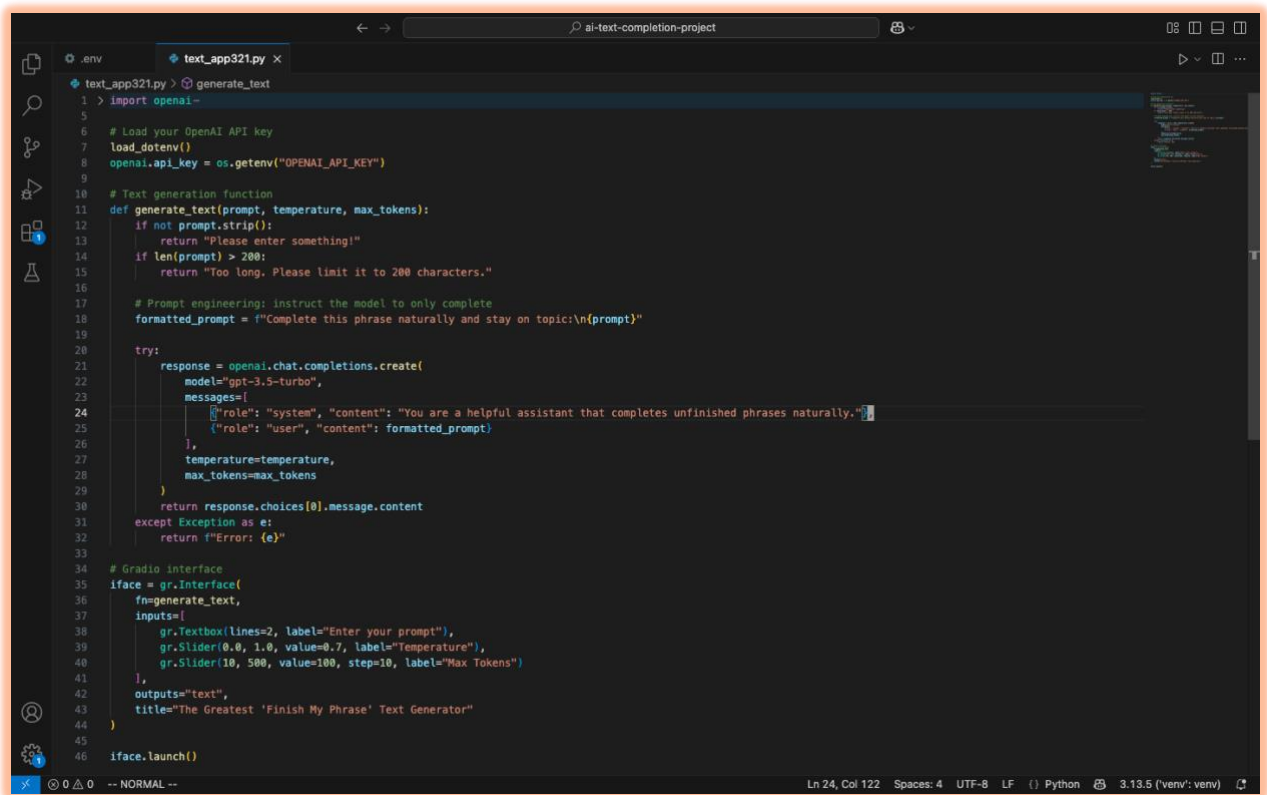
The most intriguing and challenging problem in building this application was using an API to call the information from the model directly. After this experiment, I saw that those different models, including deprecated older models, have their uses and applications to real business problems. Considering the capabilities and limitations of the AI model, it may not be in their best interest to use the newest or fastest models available due to processing, storage, and resource availability. Experimenting with current and deprecated models revealed that "older" architectures can solve many business problems while reducing cost and compute overhead. The main technical hurdle was managing API requests and responses.

Experiment Summary

To explore how they work under the hood, I built a lightweight text-completion application in Python that calls OpenAI's GPT-3.5 model via its API. My goals were threefold: (1) learn how prompt engineering and parameter tuning affect output quality, (2) evaluate trade-offs among model versions, and (3) practice integrating an external AI service into code.

The development of this AI text completion application proved to be a significant exercise in iterative troubleshooting, particularly concerning API integration. My initial

expectation was a relatively straightforward process, as I am familiar with building Python applications in controlled local environments (virtual or base). However, incorporating an external Generative AI API, especially within the constraints of free-tier services, introduced a complex layer of challenges.



```
1 > import openai-
2
3 # Load your OpenAI API key
4 load_dotenv()
5 openai.api_key = os.getenv("OPENAI_API_KEY")
6
7 # Text generation function
8 def generate_text(prompt, temperature, max_tokens):
9     if not prompt.strip():
10         return "Please enter something!"
11     if len(prompt) > 200:
12         return "Too long. Please limit it to 200 characters."
13
14     # Prompt engineering: instruct the model to only complete
15     formatted_prompt = f"Complete this phrase naturally and stay on topic:\n{prompt}"
16
17     try:
18         response = openai.chat.completions.create(
19             model="gpt-3.5-turbo",
20             messages=[
21                 {"role": "system", "content": "You are a helpful assistant that completes unfinished phrases naturally."},
22                 {"role": "user", "content": formatted_prompt}
23             ],
24             temperature=temperature,
25             max_tokens=max_tokens
26         )
27         return response.choices[0].message.content
28     except Exception as e:
29         return f"Error: {e}"
30
31 # Gradio interface
32 iface = gr.Interface(
33     fn=generate_text,
34     inputs=[
35         gr.Textbox(lines=2, label="Enter your prompt"),
36         gr.Slider(0.0, 1.0, value=0.7, label="Temperature"),
37         gr.Slider(10, 500, value=100, step=10, label="Max Tokens")
38     ],
39     outputs="text",
40     title="The Greatest 'Finish My Phrase' Text Generator"
41 )
42
43 iface.launch()
```

Most of the project's development time was dedicated to debugging and adjusting the code to achieve successful API communication. A recurring theme was the relentless errors when generating text using various free-tier Hugging Face models and their Inference API. I systematically explored several options:

Classic Hugging Face Inference API (Free Tier): This was the first avenue, aiming for direct access to publicly hosted models. Despite adhering to the documentation, consistent errors, including frequent 404 "Not Found" responses, indicated server issues, rate limiting, or more subtle mismatches in API endpoint behavior for specific models.

tiiuae/falcon-rw-1b: This model I tried early in my journey due to its size and accessibility. However, it presented integration difficulties, likely stemming from its specific API endpoint requirements or the parameters it expected for text generation via direct HTTP calls. I wasn't sure if I had the proper libraries installed in my environment. So, I then started over with yet another model.

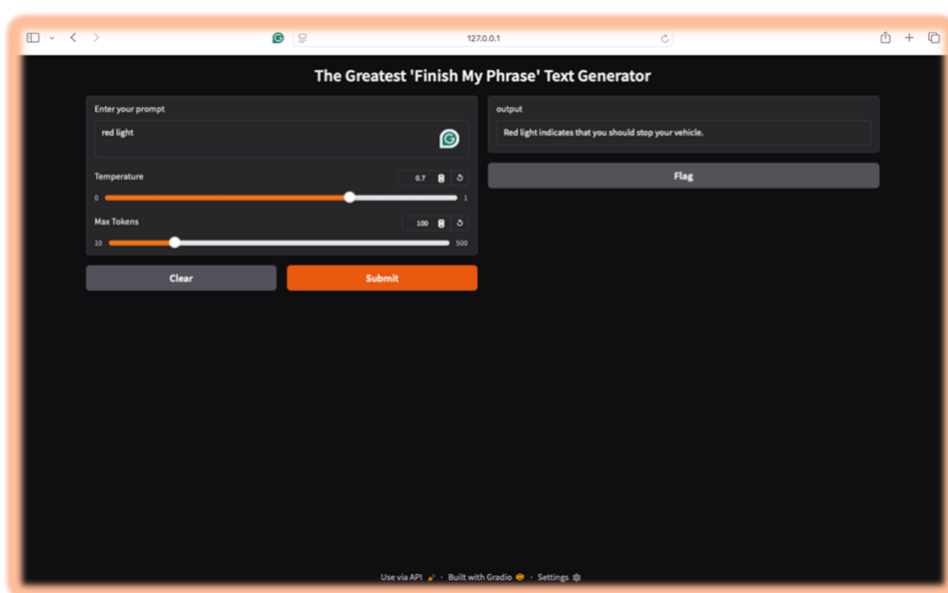
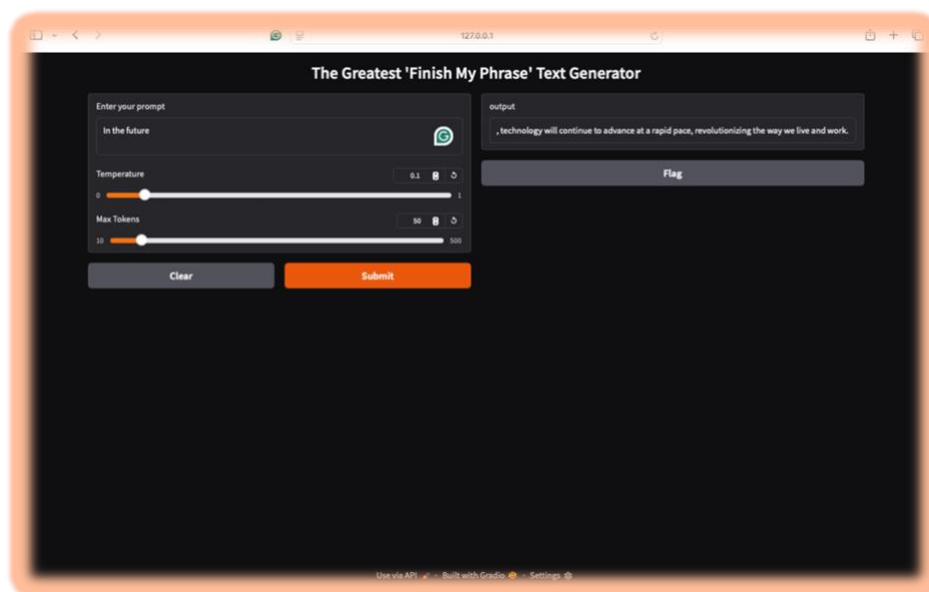
mistralai/Mixtral-8x7B-Instruct-v0.1: Transitioning to this larger, instruction-tuned model via `huggingface_hub's InferenceClient.chat.completions.create` method seemed promising for its capabilities. Yet, this path led to a persistent and unexpected `TypeError` indicating an unrecognized `max_new_tokens` argument. The error persisted despite confirming that the `huggingface_hub` library was updated and experimenting with parameter naming (`max_new_tokens` vs. `max_tokens`). This suggests an incompatibility or a particular parameter expectation for Mixtral on the public Inference API when used in this context.

bloom-560m: While a specific attempt with `bloom-560m` struggled with various Hugging Face models underscored the unreliability and complexity faced across the free tier.

Each error required investigation into API documentation, community forums, and a troubleshooting approach to parameter adjustments. The sheer volume and variety of errors, ranging from authentication failures to unexpected keyword arguments and general service unavailability, created a significant debugging time allotment.

After extensive and unsuccessful attempts to achieve reliable text generation with the free-tier options, a decision was made: to subscribe to and utilize OpenAI's API with paid credits. This change immediately resolved the persistent API communication issues. The code, which had been simplified for structure and logic, functioned once connected to OpenAI's service. This painstaking process, though challenging, provided invaluable hands-on experience with the practical limitations of free-tier AI services, the nuances of API documentation, and the critical importance of reliable infrastructure when building real-world applications.

I tried a variety of prompts, which made it easy to test different scenarios, gather results, and display how the model would behave. Some were creative, like "In the future..." or "Continue the story of a lost teddy bear..." Others were informative or instructional, such as "twinkle twinkle little star" and abstract like "red light". I also tried a few fun prompts like "When to travel on the sunny side of the street". Each text change gave a different response style, which showed how the model adjusts based on input and parameters like temperature and tokens.



127.0.0.1

The Greatest 'Finish My Phrase' Text Generator

Enter your prompt

Continue the story of a lost teddy bear

Temperature 0.93

Max Tokens 100

Clear Submit

output

The lost teddy bear was last seen sitting on a bench at the park, looking forlorn as children played around him. His fluffy fur was starting to get dirty from being outdoors for so long, but he held onto hope that his owner would come back to find him. Little did he know, a kind-hearted old man had noticed the lonely teddy bear and decided to take him home, where he would be cleaned up and given a special place of honor on the man's shelf, surrounded

Flag

Use via API · Built with Gradio · Settings

127.0.0.1

The Greatest 'Finish My Phrase' Text Generator

Enter your prompt

twinkle twinkle little star

Temperature 0.7

Max Tokens 100

Clear Submit

output

How I wonder what you are.

Flag

Use via API · Built with Gradio · Settings

Observations

When comparing different outputs, I noticed differences in the prompt type and temperature setting. I also had to add some prompt engineering to my Python script for the responses from the chat API to be more specific towards sentence completion rather than a friendly chatbot. For example, the sentences flowed into a paragraph, or the program repeated a nonsensical response on length and temperature settings; the explanation of gravity was educational and straightforward, and the story prompt had more imagination but less structure. At lower temperatures, like 0.2, the responses were more predictable. At higher temperatures, like 0.8 or 1.0, the reactions were more unique but sometimes less logical or focused.

Reflection

Although I had many trials while working on this project, I learned immensely about code behavior and how that code may behave with external elements and virtual environments. I had a chance to explore Google Colab and Google AI Studio and realized that even my favorite Jupyter notebooks are not the best fit for every use case. I'm proud that I stayed persistent until I made a working prototype.