OpenAI APIs 101

Yogev Shani, MTA

Agenda

- Explore OpenAI APIs
  - Chat completion
  - Image Generation
  - Speech to Text
  - Fine-Tuning
  - Moderation
- Deep Dive into Chat completion API & Function Calling feature
- Costs considerations

OpenAI API endpoints

- **Chat completion (Python method :** openai.ChatCompletion.create**)**
  - **Purpose:** Given a series of messages, generate a conversational response.
- **Image Generation (Python method:** openai.Image.create**)**
  - **Purpose:** Generate images from a textual description.
- **Speech to text (Python method :** openai.Audio.transcribe**)**
  - **Purpose**: Convert audio to text.
- **Fine tuning (Python method :** openai.FineTune.create**)**
  - **Purpose**: Customize an AI model with specific training data.
- **Moderation (Python method :** openai.Moderation.create**)**
  - **Purpose**: Detect and filter harmful or inappropriate content.

Getting started with OpenAI API &

- **Install Python library**
  - *pip install openai*
- **Get your API key**
  - Login at OpenAI developer platform
  - Go to API keys
  - Create a new secret key

Ensure successful installation

- **Verify installation**

```
import openai

print( openai.__version__)
```

- **And get:**
  - 1.58.1

Chat completion API

- **Purpose: Generates responses based on a conversation (list of messages)**
- **Key Parameters:**
  - **Model: Choose the LLM based on needs:**
    - gpt-3.5-turbo: Faster and cheaper.
    - gpt-4.0: Better quality, supports larger inputs, fewer errors.
  - **Temperature: Controls response creativity:**
    - **Range: 0.0 (deterministic) to 2.0 (creative).**
  - **Messages: Represents the conversation:**
    - **Each message includes:**
      - Role (e.g., user, system, assistant).
      - Content (text of the message).

messages: Structuring the Conversation for the Model.

```
messages = [
   {"role": "system", "content": "You are an assistant that talks to a 15 yo and you should speak the same"},
   {"role": "user", "content": "Should I use goto statements?"},
   {"role": "assistant", "content": "No bro, that's bad practice duh 😳"},
   {"role": "user", "content": user_input},
]
```

- The first 3 elements of the list are the "context"
- The last is the user's input we want the model to respond to
- system role: High-level instructions for the conversation
- assistant role: The model's "ideal" (or previous) response
- user role: The user's input

Full API code

```
from openai import OpenAI

client = OpenAI(
   api_key="XXX")

messages = [
   {"role": "system", "content": "You are a python assistant that talks to a 15 yo and you should speak the same"},
   {"role": "user", "content": "Should I use goto statements?"},
   {"role": "assistant", "content": "No bro, that's bad practice duh 😳"}]

user_prompt = input("Enter your prompt (or 'exit' to quit): ")

messages.append({"role": "user", "content": user_prompt})

chat_completion = client.chat.completions.create(
      messages=messages,
      model="gpt-3.5-turbo",
      max_tokens=50,  # Limit the number of tokens in the reply
      temperature=0.6
   )

ai_reply = chat_completion.choices[0].message.content
print("AI:", ai_reply)
```

Running it…

- User Input:
  - *Enter your prompt (or 'exit' to quit): what is an integer?*
- Open AI API Response:
  - *An integer is like a whole number without any decimal points, like 1, 5, -3, or 1000. It's just a number you can count with, ya know?*

system prompt

- What is a System Prompt?
  - ***Purpose**: The system prompt sets the rules or context for the conversation before the model responds to user input.*
  - *Role: "role": "system" specifies that this message is meant to define how the assistant should behave.*
- Examples:

```
{"role": "system", "content": "You are a polite customer support agent for a tech company."}
{"role": "system", "content": "You are a Python tutor who provides code snippets and explanations."}
{"role": "system", "content": "You are a casual assistant who uses humor and informal language."}
{"role": "system", "content": "You are an assistant who provides general health information but not medical advice."}
```

temperature values

- The temperature parameter controls the "creativity" or randomness of the model's responses. It affects how the language model decides between different plausible continuations of text.
- Temperature Range
  - Allowed Range: 0.0 to 2.0
  - Default Value: Typically, 0.7
- How It Works
  - Low Temperature (0.0): The model prioritizes the most likely outcomes. Use this for factual and deterministic tasks.
  - High Temperature (1.0 or higher`): The model explores less likely outcomes, making responses more creative or unexpected.

Temperature examples

- Prompt : "What is the capital of France?"?
  - Temperature = 0.0 (Deterministic):
    - Response: "The capital of France is Paris."
  - Temperature = 1.0 (Creative):
    - Response: "Paris, the city of lights, is the capital of France."
- Prompt: "Write a creative opening line for a story."
  - Temperature = 0.2 (Simple and straightforward):
    - Response: "Once upon a time, there was a small village."
  - Temperature = 0.7 (Moderately creative):
    - Response: "In the shadow of the towering cliffs, a secret lay buried for centuries."
  - Temperature = 1.5 (Highly creative and random):
  - Response: "Under the moon's mischievous grin, the talking rabbits hatched their daring plan."

temperature recommendations

- Low Temperature:
  - Reliable and deterministic outputs (good for serious tasks).
- High Temperature:
  - Creative and varied outputs (good for brainstorming or storytelling).
- Adjust temperature based on the task to balance creativity and accuracy!
  - Factual Q&A 0.0 - 0.2
  - Code Generation 0.2 - 0.5
  - Creative Writing 0.7 - 1.0
  - Brainstorming Ideas 1.0 - 1.5

OpenAI API Function Calling

- Function calling in OpenAI API enables the model to interact with external systems or processes by "calling" predefined functions based on user input or the conversation's context. This feature allows for integrating the language model with real-world applications like fetching data, automating tasks, or controlling external systems.

Function Calling – How it works?

- Define Functions:
  - Define a set of functions that the model can "call" when appropriate.
  - Each function is described with its name, description, and expected parameters.
- Provide Functions to the API:
  - Pass the functions as part of your `ChatCompletion` API request.
- Model Decides:
  - Based on the conversation and provided functions, the model decides if a function call is necessary and which function to use.
- Response Includes Function Call:
  - If the model calls a function, the response will contain: The function's name & A JSON string of arguments to pass to the function.
- The Application Executes the Function:
  - Parse the function call and execute it within your application.
  - Optionally, return the results back to the model for further conversation.

Function calling – Example 1

- The python function:

```
def call_staff():
    print("Calling staff to table")
```

- functions parameters :

```
{
    "name": "call_staff",
    "description": "Call a member of staff to the table",
    "parameters": {"type": "object", "properties": {}},
},
```

Function calling – Example 2

- The python function:

```
def place_order(name, size, take_away=False):
    place = "take away" if take_away else "eat in"
    print(f"Placing order for {name} pizza, {size} to {place}")
```

- functions parameters :

```
{
  "name": "place_order",
  "description": "Place an order for a pizza",
  "parameters": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "The name of the pizza, e.g. Pepperoni",
      },
      "size": {
        "type": "string",
        "enum": ["small", "medium", "large"],
        "description": "The size of the pizza. Always ask for clarification if not specified.",
      },
      "take_away": {
        "type": "boolean",
        "description": "Whether the pizza is taken away. Assume false if not specified.",
      },
    },
    "required": ["name", "size", "take_away"],
  },
},
```

function parameter schema breakdown

- **name**: The name of the function to call
- **description**: A description of the function, helps the model choose
- **parameters/type**: Always "object" for now
- **parameters/properties**: Empty if no function parameters
  - <param>/type: "string", "boolean", "int“
  - <param>/enum: Optional list of allowed values
  - <param>/description: Description of parameter, helps the model
- **parameters/required**: List of required parameters

system role prompt

```
{
   "role": "system",
   "content": "Don't make assumptions about what values to put into functions. "
   + "Ask for clarification if you need to.",
}
```

- **Tip**: Keep low temperature to avoid 'hallucinations' and to stick to the provided functions.

API response

If the model determines that a function call is required, the response will include a function_call property, and the content field will be null.

```
{
    "role": "assistant",
    "content": null,
    "function_call": {
        "name": "place_order",
        "arguments": "{\n\"name\": \"Olives\",\n\"size\":
\"large\",\n\"take_away\": true\n}"
    }
}
```

Parsing the response and calling the function

```
def get_chat(    openai_client, messages=None, model="gpt-4", temperature=0.2, functions=None ):
    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
        temperature=temperature,
        functions=functions,
    )
    message = response.choices[0].message
    if message.function_call:
        f = globals()[message.function_call.name]
        params = json.loads(message.function_call.arguments)
        f(**params)
    else:
        print(message.content)

    return message.function_call is not None
```

Conversation history

To enable the model to use previous inputs as context, the user's input and the model's reply must be appended to the messages list.

AI: How may I help you?
User: I'd like to order some pizza please
AI: Of course, I'd be happy to assist you with that. Could you please specify the name of the pizza you'd like to order and the size you prefer? Also, would this be for take away or are you dining in?
AI: I'd like a large one
User: Sure, could you please specify the type of pizza you would like to order?
AI: A Margherita please
User: Placing order for Margherita pizza, large to eat in

Take aways…

- Use Chat completions API For Text generation
  - Enhance it with functions calling
- Start with **gpt-3.5-turbo** continue to **gpt-4** as needed.
  - Function calling require **gpt-4**
- Python library can be used for getting started quickly
- Use OpenAI API Playground to experiment
- Full "pizza bot" example: plat.is/openai-pizza

Cost Considerations When Using OpenAI API

- **OpenAI API usage incurs costs -** The service is not free, and charges are based on token usage.
- **Token breakdown -** On average, 1 token equals approximately 4 characters in English.
- **Language differences -** Some languages may require more tokens, increasing costs.
- **Costs during development vs. production -** Expenses can grow significantly when scaling to production environments.
- **Choose cost-effective models -** Start with a model like gpt-3.5-turbo and explore cheaper options if possible.
- **gpt-3.5-turbo vs. gpt-4 -** While gpt-3.5-turbo is affordable, gpt-4 offers better performance at a higher cost.
- **High-cost APIs -** APIs for speech-to-text (e.g., Whisper) and image generation (e.g., DALL-E) are relatively expensive.
- **Local alternatives -** Consider running models like Whisper locally to reduce recurring expenses.

Fine tuning

- Fine-tuning is the process of taking a pre-trained model and further training it on a custom dataset to optimize its performance for a specific task.
  - It is useful when your prompts require many examples to achieve the desired output
  - Example: A general language model can be fine-tuned to specialize in legal, medical, or technical content.
- Fine tuned model can be used as a model with OpenAI APIs
- Fine-tuning offers precise control but can be expensive and might not always be cost-effective compared to well-crafted prompts.
- You incur costs for both the fine-tuning process and for using the fine-tuned model in production.
- Avoid fine-tuning unless you've exhausted prompt engineering techniques and need consistent specialized outputs.