Learning Objectives

Learners will be able to...

- **Generate a Response Using `openai.Completion.create`**
- **Define Endpoints and Embeddings Model**
- **Learn about Different Models**
- **Run Different Endpoints with OpenAI**

info

Make Sure You Know

You are familiar with Python.

Limitations

This is a gentle introduction. So there is a little bit of Python programming. The information might not be the most up to date as OpenAI releases new features.

Endpoints

So far, we have interacted with the GPT-3 model by using the official OpenAI playground, a text file for our prompts, or an IDE with some Python code. The user interfaces for these methods of interaction are all different. However, all of these interactions are making use of API endpoints.

An endpoint for the OpenAI API is a uniform resource identifier (URI) that is used to access data or services provided by the company. Each endpoint represents a different way of interacting with OpenAI. Send information to a specific endpoint, and OpenAI sends back a response based on your initial query.

All of our examples up until now have used the Completions endpoint when generating a response to our prompts. This is one of several endpoints available to users. Here is the full list of available endpoints:

1. **List Models** - also known as the metadata endpoint; returns a list of models as well as some metadata about each model
2. **Retrieve Model** - returns detailed metadata about the specified model
3. **Completions** - most popular endpoint; returns a response to a prompt
4. **Semantic Search** - allows you to semantically rank documents with natural language
5. **Files** - upload and manipulate files in OpenAI storage
6. **Classification** - lets you classify a query without the need for finetuning or hyperparameter tuning
7. **Answers** - takes a question and returns an answer based on provided information (files or training examples)
8. **Embeddings** - returns an embedding based on information sent to the API

We will be exploring some of these endpoints throughout this assignment. OpenAI is actively working on new engines and updating existing ones. Over time, there might be some changes to the endpoints we see now.

Models

Endpoint Signatures

Each endpoint has a signature. This means either a `GET` or `POST` request followed by the URI. Below is an example of the Completions endpoint signature. It uses a `POST` method. Other signatures may require a parameter. These will be identified by curly brackets `{ }`.

info

Completions Endpoint Signature

```
POST https://api.openai.com/v1/completions
```

Endpoint signatures are provided in the examples below to give you a better idea of how the OpenAI is doing behind the scenes.

List Model Endpoint

info

List Model Endpoint Signature

```
GET https://api.openai.com/v1/models
```

This code prints out a long list of all of the available models and accompanying metadata. You can see why this is sometimes referred to as the metadata endpoint.

```
models = openai.Model.list()
print(models)
```

In a previous discussion, we talked about the four different types of models: `davinci`, `curie`, `babbage`, and `ada`. Each of these models varies in terms of capabilities, speed, and cost. However, the list of available models is greater than four. We will see later how we can use models like `text-search-babbage-query-001`.

Printing all of the models looks like it may be a JSON object or a dictionary. In reality, the return type from the API call is `OpenAIObject`. However, we can treat it like a dictionary. The code below prints out the first model and its metadata.

```
models = openai.Model.list()
print(models["data"][0])
```

challenge

Try this variation:

- Create a list of all the model IDs without any other metadata.

```
models = openai.Model.list()
model_ids = [model["id"] for model in models["data"]]
print(model_ids)
```

Retrieve Model Endpoint

info

List Engines Endpoint Signature

```
GET https://api.openai.com/v1/models/{model}
```

Instead of parsing the list of models, you can retrieve metadata about a specific one if you know the `id` for the model. The code sample below prints out the metadata for the `text-davinci-002` model.

```
model = openai.Model.retrieve("text-davinci-002")
print(model)
```

Embeddings

Embedding Endpoint

info

Embeddings Endpoint Signature

```
POST https://api.openai.com/v1/embeddings
```

The API also has another experimental endpoint called *embeddings*. Embeddings are a representation of a given input as a vector of floating point numbers. Embeddings can be easily consumed by machine learning models and algorithms. They are often used to determine the semantic similarity between two texts. If two texts are similar, then their vector representations should also be similar.

Currently OpenAI offers three families of embedding models that allow for text search, text similarity and code search. Each family includes up to four models on a spectrum of capabilities:

- **Ada** (1024 dimensions),
- **Babbage** (2048 dimensions),
- **Curie** (4096 dimensions),
- **Davinci** (12288 dimensions).

These embedding models are specifically created to be good at a particular task. For example, `text-similarity-ada-001` is good at capturing semantic similarity between two or more pieces of text. Or `text-search-ada-doc-001` helps measure whether long documents are relevant to a short search query. For more information on embeddings, see the OpenAI documentation.

Creating an Embedding

In this example, we are going to use the `text-similarity-babbage-001` model. We also need to provide some text for the embedding. This is done with the `input` keyword argument. Both of these are required. The `user` keyword argument is optional. This unique identifier can be used to help monitor for abuse. The code sample below should print a long list of floating point numbers.

```
emb=openai.Embedding.create(
  model="text-similarity-babbage-001",
  input="You will rejoice to hear that no disaster has
        accompanied the")


print(emb)
```

Just like the list of models, the return object is of type `OpenAIObject`. If you want to access just the vector, change the print statement to the following:

```
print(emb["data"][0]["embedding"])
```

Edits

Edits

Given a prompt and an instruction, the model will return an edited version of the prompt.

`POST https://api.openai.com/v1/edits`

```
edi= openai.Edit.create(
  model="text-davinci-edit-001",
  input="What day was it? Wesdneday?",
  instruction="Fix the spelling mistakes"
)
print(edi['choices'][0]['text'].strip())
```

The Edit body

Similarly to our create call our edit calls can take a couple of arguments to help the user better control what is generated.

The must-have arguments are `model` and `instruction`.

```
edi= openai.Edit.create(
  model="text-davinci-edit-001",
  instruction="Captitalize the first letter of all the words
        in the sentence"
)
print(edi['choices'][0]['text'].strip())
```

For our purposes we are also going to make input as a default because it keeps it clean about what is the input we are interacting with.

`input` : is optional but defaults to an empty string when no argument is specified. It is used as a starting point for the edit.
`instruction`: is required and tell the AI how to edit the prompt

```
edi= openai.Edit.create(
  model="text-davinci-edit-001",
  input="What day was it? Wesdneday?",
  instruction="change wednesday to saturday."
)
print(edi['choices'][0]['text'].strip())
```

Feel free to try it more than once, there might be a chance the AI misinterprets the directions.
Similarly, to create a call there are a couple of other optional arguments. In this case, we could have added `n`, `temperature` and `top_p`.

`n`: defaults to 1 and tells how many edits to create from input and instruction
`temperature`: defaults to 1 , higher values mean the model will take more risks.
`top_p`: 0.1 means only the tokens comprising the top 10% probability mass are considered.

A Try It button is provided in case you wanted to try adding some of those variables.

Coding Exercise

Using the OpenAI Edit API, create a Python function `fix_spelling` that takes a string as input and returns a corrected version of the input with spelling mistakes fixed. The function should use the `text-davinci-edit-001` model. For a more effective output make `top_p`=0.1 as one of your arguments.