Bonus Lab 4: Using LLMs with OpenAI and Hugging Face with Python

Dr. Derrick L. Cogburn

2024-11-19

Table of contents

1 Lab Overview

1.1 Learning Objectives

1. Understand how to access and use Large Language Models (LLMs).
2. How to authenticate to open AI

Overview

In this final bonus lab we introduce the process of authenticating to OpenAI and using Large Language Models (LLMs) to generate text. We will use the OpenAI GPT-3 model to generate text based on a given prompt. We will also discuss the ethical considerations and potential biases associated with using LLMs.

1.2 There are two main parts to the lab:

Exercise 1: Authenticating to the OpenAI API

Exercise 2: Defining Helper Functions in OpenAI

Exercise 3: Prompting Exercises Using GPT-3.5 Turbo

Exercise 4: Defining Functions and Prompting in Hugging Face

2 Lab Instructions

Make sure you have the following Python packages installed in the environment you are using for the lab: OpenAI's `os`, `openai` package, and the Hugging Face `huggingface_hub` package, and the `transformers` package.

2.1 Exercise 1: Authenticating to the OpenAI API

You will first need to authenticate to OpenAI's API to access the GPT-3.5 Turbo model. You will need an API key to authenticate to the OpenAI API. Below is an example of how to authenticate to the OpenAI API using Python:

```
import os
import openai

from openai import OpenAI

os.environ['OPENAI_API_KEY'] = 'replacethiswithyourapikey'
```

Now, in a code chunk below, create a helper function to call one of the key OpenAI API endpoints. This helper function includes the get_completion() function.

```
import os
import openai

from openai import OpenAI
```

2.2 Exercise 2: Defining Helper Functions in OpenAI

```
client = OpenAI()

# This is a helper function to return what we want from the GPT 3.5 Turbo llm
def get_completion(prompt, model="gpt-3.5-turbo"):
    messages = [{"role": "user", "content": prompt}]
    response = client.chat.completions.create(
        model=model,
        messages=messages,
        temperature=0
    )
    return response.choices[0].message.content
```

A basic session with the GPT LLM has three key parts: (1) the text object, containing the text you are trying to act upon; (2) your prompt; and (3) the helper function containing the openai endpoint you are trying to use (in this case a get_completion() endpoint.)

Let's begin with creating a text object, using text = f" " "text" " ". Your text can contain any text you would like. We are going to begin by using the LLM to summarize this text. After you create the text object, you will create the prompt object using prompt = f" " "text" " ". Finally, we will call the endpoint helper function we created by creating an object called response = get_completion(prompt). Our final line of code in this chunk will be to print the response using the print(response) function.

2.3 Exercise 3: Prompting Exercises Using GPT-3.5 Turbo

```
text = f"""
You should express what you want a model to do by \
providing instructions that are as clear and \
specific as you can possibly make them. \
This will guide the model towards the desired output, \
and reduce the chances of receiving irrelevant \
or incorrect responses. Don't confuse writing a \
clear prompt with writing a short prompt. \
In many cases, longer prompts provide more clarity \
and context for the model, which can lead to \
more detailed and relevant outputs.
"""
```

```
prompt = f"""
Summarize the text delimited by triple backticks \
into a single sentence.
```{text}```
"""
```

```
response = get_completion(prompt)
print(response)
```

```
Providing clear and specific instructions to a model will guide it towards the desired output
```

Now, in the code chunk below, we will use the LLM to generate a set of steps based on some text provided.

```
text_1 = f"""
Making a cup of tea is easy! First, you need to get some \
water boiling. While that's happening, \
grab a cup and put a tea bag in it. Once the water is \
hot enough, just pour it over the tea bag. \
Let it sit for a bit so the tea can steep. After a \
few minutes, take out the tea bag. If you \
like, you can add some sugar or milk to taste. \
And that's it! You've got yourself a delicious \
cup of tea to enjoy.
"""
prompt = f"""
You will be provided with text delimited by triple quotes.
If it contains a sequence of instructions, \
re-write those instructions in the following format:

Step 1 - ...
Step 2 - …
…
Step N - …

If the text does not contain a sequence of instructions, \
then simply write \"No steps provided.\"

\"\"\"{text_1}\"\"\"
"""
response = get_completion(prompt)
```
```

```
print("Completion for Text 1:")
print(response)
```

```
Completion for Text 1:
Step 1 - Get some water boiling.
Step 2 - Grab a cup and put a tea bag in it.
Step 3 - Pour the hot water over the tea bag.
Step 4 - Let the tea steep for a few minutes.
Step 5 - Remove the tea bag.
Step 6 - Add sugar or milk to taste.
Step 7 - Enjoy your delicious cup of tea.
```

Now, let's try a second example.

```
text_2 = f"""
The sun is shining brightly today, and the birds are \
singing. It's a beautiful day to go for a \
walk in the park. The flowers are blooming, and the \
trees are swaying gently in the breeze. People \
are out and about, enjoying the lovely weather. \
Some are having picnics, while others are playing \
games or simply relaxing on the grass. It's a \
perfect day to spend time outdoors and appreciate the \
beauty of nature.
"""
prompt = f"""
You will be provided with text delimited by triple quotes.
If it contains a sequence of instructions, \
re-write those instructions in the following format:

Step 1 - ...
Step 2 - …
…
Step N - …

If the text does not contain a sequence of instructions, \
then simply write \"No steps provided.\"

\"\"\"{text_2}\"\"\"
"""
response = get_completion(prompt)
```

```
print("Completion for Text 2:")
print(response)
```

```
Completion for Text 2:
No steps provided.
```

Now, we will demonstrte how an LLM can be used to generate a conversation in a "consistent" style.

```
prompt = f"""
Your task is to answer in a consistent style.

<child>: Teach me about patience.

<grandparent>: The river that carves the deepest \
valley flows from a modest spring; the \
grandest symphony originates from a single note; \
the most intricate tapestry begins with a solitary thread.

<child>: Teach me about resilience.
"""
response = get_completion(prompt)
print(response)
```

```
<grandparent>: The tallest trees weather the strongest storms; the brightest stars shine in t
```

Now, the LLM will be used to generate a conversation in a "creative" style, but it will also be used to perform several key actions, including summarizing a text, translating the summary into French, listing each name in the French summary, and outputting a json object that contains the following keys: french_summary, num_names.

```
text = f"""
In a charming village, siblings Jack and Jill set out on \
a quest to fetch water from a hilltop \
well. As they climbed, singing joyfully, misfortune \
struck-Jack tripped on a stone and tumbled \
down the hill, with Jill following suit. \
Though slightly battered, the pair returned home to \
comforting embraces. Despite the mishap, \
their adventurous spirits remained undimmed, and they \
```

```
continued exploring with delight.
"""
# example 1
prompt_1 = f"""
Perform the following actions:
1 - Summarize the following text delimited by triple \
backticks with 1 sentence.
2 - Translate the summary into French.
3 - List each name in the French summary.
4 - Output a json object that contains the following \
keys: french_summary, num_names.

Separate your answers with line breaks.

Text:
```{text}```
"""
response = get_completion(prompt_1)
print("Completion for prompt 1:")
print(response)
```

```
Completion for prompt 1:
1 - Jack and Jill, siblings on a quest for water, face misfortune but remain adventurous.

2 - Jack et Jill, frère et sœur en quête d'eau, font face à la malchance mais restent aventu

3 - Jack, Jill

4 -
{
  "french_summary": "Jack et Jill, frère et sœur en quête d'eau, font face à la malchance ma
  "num_names": 2
}
```

Now, the LLM will be used to summarize a text, translate the summary into French, list each name in the French summary, and output a json object that contains the following keys: french_summary, num_names.

```
prompt_2 = f"""
Your task is to perform the following actions:
1 - Summarize the following text delimited by
```
```

```
  <> with 1 sentence.
2 - Translate the summary into French.
3 - List each name in the French summary.
4 - Output a json object that contains the
  following keys: french_summary, num_names.

Use the following format:
Text: <text to summarize>
Summary: <summary>
Translation: <summary translation>
Names: <list of names in summary>
Output JSON: <json with summary and num_names>

Text: <{text}>
"""
response = get_completion(prompt_2)
print("\nCompletion for prompt 2:")
print(response)
```

```
Completion for prompt 2:
Summary: Jack and Jill, two siblings, go on a quest to fetch water from a hilltop well but er

Translation: Jack et Jill, deux frères et sœurs, partent en quête d'eau d'un puits au sommet

Names: Jack, Jill

Output JSON:
{
  "french_summary": "Jack et Jill, deux frères et sœurs, partent en quête d'eau d'un puits au
  "num_names": 2
}
```

2.4 Exercise 4: Defining Functions and Prompting in Hugging Face

Now, we will take a look at an example from Hugging Face.

```
from transformers import pipeline, AutoTokenizer, AutoModelForCausalLM
import torch

def setup_model(model_name="facebook/opt-350m"):
```

```
    """Initialize model and tokenizer"""
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForCausalLM.from_pretrained(model_name)
    return model, tokenizer

def generate_response(prompt, model, tokenizer, max_length=100):
    inputs = tokenizer(prompt, return_tensors="pt")

    outputs = model.generate(
        inputs["input_ids"],
        max_length=max_length,
        temperature=0.7,        # Controls randomness
        top_p=0.9,             # Nucleus sampling
        do_sample=True,        # Enable sampling
        num_beams=4,           # Beam search
        no_repeat_ngram_size=2 # Avoid repetition
    )

    return tokenizer.decode(outputs[0], skip_special_tokens=True)

def main():
    # Example usage
    model, tokenizer = setup_model()

    prompt = "Explain what machine learning is:"
    response = generate_response(prompt, model, tokenizer, max_length=300)
    print(f"Prompt: {prompt}\nResponse: {response}")

if __name__ == "__main__":
    main()
```

```
Prompt: Explain what machine learning is:
Response: Explain what machine learning is:

Machine Learning is a set of algorithms that can be used to analyze data in a way that makes
```

This code demonstrates how to use a pre-trained language model from Hugging Face to generate text based on a prompt. The model is fine-tuned on a large dataset and can generate coherent and contextually relevant text based on the input prompt. This can be useful for various natural language processing tasks, such as text generation, summarization, and question answering.