gxdqkshk2

December 4, 2024

[15]:
```
# imports

import os
from dotenv import load_dotenv
from openai import OpenAI
import anthropic
from IPython.display import Markdown, display, update_display

import google.generativeai
```

[16]:
```
load_dotenv()
openai_api_key = os.getenv('OPENAI_API_KEY')
anthropic_api_key = os.getenv('ANTHROPIC_API_KEY')
google_api_key = os.getenv('GOOGLE_API_KEY')

if openai_api_key:
    print(f"OpenAI API Key exists and begins {openai_api_key[:8]}")
else:
    print("OpenAI API Key not set")

if anthropic_api_key:
    print(f"Anthropic API Key exists and begins {anthropic_api_key[:7]}")
else:
    print("Anthropic API Key not set")

if google_api_key:
    print(f"Google API Key exists and begins {google_api_key[:8]}")
else:
    print("Google API Key not set")
```

```
OpenAI API Key exists and begins sk-proj-
Anthropic API Key exists and begins sk-ant-
Google API Key exists and begins AIzaSyDX
```

[17]:
```
# Connect to OpenAI, Anthropic and Google
# All 3 APIs are similar
# Having problems with API files? You can use openai =␣
 ↪OpenAI(api_key="your-key-here") and same for claude
```

```
# Having problems with Google Gemini setup? Then just skip Gemini; you'll get␣
 ↪all the experience you need from GPT and Claude.

openai = OpenAI()

# claude = anthropic.Anthropic()
# in case if you are using claude API , uncomment the same


google.generativeai.configure()
```

Asking LLMs to tell a joke It turns out that LLMs don't do a great job of telling jokes! Let's compare a few models. Later we will be putting LLMs to better use!

What information is included in the API Typically we'll pass to the API:

The name of the model that should be used

A system message that gives overall context for the role the LLM is playing

A user message that provides the actual prompt

There are other parameters that can be used, including temperature which is typically between 0 and 1; higher for more random output; lower for more focused and deterministic.

[18]:
```
system_message = "You are an assistant that is great at telling jokes"
user_prompt = "Tell a light-hearted joke for an audience of Data Scientists"
```

[19]:
```
prompts = [
    {"role": "system", "content": system_message},
    {"role": "user", "content": user_prompt}

  ]

# put prompts in the list
```

[20]:
```
# GPT-3.5-Turbo

completion = openai.chat.completions.create(model='gpt-3.5-turbo',␣
 ↪messages=prompts)
print(completion.choices[0].message.content)
```

```
Why do data scientists prefer dark chocolate?

Because they like their data to be unambiguous!
```

[21]:
```
# GPT-4o-mini
# Temperature setting controls creativity
```

```
completion = openai.chat.completions.create(
    model='gpt-4o-mini',
    messages=prompts,
    temperature=0.7
)
print(completion.choices[0].message.content)
```

```
Why did the data scientist break up with the statistician?

Because she found him too mean!
```

[22]:
```
# GPT-4o

completion = openai.chat.completions.create(
    model='gpt-4o',
    messages=prompts,
    temperature=0.4
)
print(completion.choices[0].message.content)
```

```
Why did the data scientist bring a ladder to work?

Because they heard the project was going to the next level!
```

Run if this using Claude Anthropic API

[23]:
```
# Claude 3.5 Sonnet
# API needs system message provided separately from user prompt
# Also adding max_tokens

message = claude.messages.create(
    model="claude-3-5-sonnet-20240620",
    max_tokens=200,
    temperature=0.7,
    system=system_message,
    messages=[
        {"role": "user", "content": user_prompt},
    ],
)

print(message.content[0].text)



# Claude 3.5 Sonnet again
# Now let's add in streaming back results

result = claude.messages.stream(
```

```
    model="claude-3-5-sonnet-20240620",
    max_tokens=200,
    temperature=0.7,
    system=system_message,
    messages=[
        {"role": "user", "content": user_prompt},
    ],
)

with result as stream:
    for text in stream.text_stream:
            print(text, end="", flush=True)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[23], line 5
      1 # Claude 3.5 Sonnet
      2 # API needs system message provided separately from user prompt
      3 # Also adding max_tokens
----> 5 message = claude.messages.create(
      6     model="claude-3-5-sonnet-20240620",
      7     max_tokens=200,
      8     temperature=0.7,
      9     system=system_message,
     10     messages=[
     11         {"role": "user", "content": user_prompt},
     12     ],
     13 )
     15 print(message.content[0].text)
     19 # Claude 3.5 Sonnet again
     20 # Now let's add in streaming back results

NameError: name 'claude' is not defined
```

[24]:
```
# The API for Gemini has a slightly different structure

gemini = google.generativeai.GenerativeModel(
    model_name='gemini-1.5-flash',
    system_instruction=system_message
)
response = gemini.generate_content(user_prompt)
print(response.text)
```

```
Why was the data scientist sad?  Because they didn't get arrays.  (Get it?
A-rays… like sun rays…  never mind.)
```

[25]:
```
# To be serious! GPT-4o-mini with the original question

prompts = [
    {"role": "system", "content": "You are a helpful assistant that responds in␣
 ↪Markdown"},
    {"role": "user", "content": "How do I decide if a business problem is␣
 ↪suitable for an LLM solution? Please respond in Markdown."}
  ]
```

What if we want to stream results in markdown ?

[26]:
```
# Have it stream back results in markdown

stream = openai.chat.completions.create(
    model='gpt-4o',
    messages=prompts,
    temperature=0.7,
    stream=True
)

reply = ""
display_handle = display(Markdown(""), display_id=True)
for chunk in stream:
    reply += chunk.choices[0].delta.content or ''
    reply = reply.replace("```","").replace("markdown","")
    update_display(Markdown(reply), display_id=display_handle.display_id)
```

Deciding if a business problem is suitable for a Large Language Model (LLM) solution involves evaluating several factors. Here's a structured approach to help you make that decision:

0.0.1 1. Nature of the Problem

- **Text-Based Tasks**: LLMs are inherently designed to handle text-based tasks. Consider if the problem involves natural language understanding, generation, or transformation. Examples include content creation, summarization, translation, and sentiment analysis.
- **Complexity and Ambiguity**: LLMs excel at understanding context and handling ambiguous language, making them suitable for tasks that involve nuanced interpretation of text.

0.0.2 2. Data Availability

- **High-Quality Text Data**: Ensure that you have access to a large corpus of relevant, high-quality text data. LLMs require substantial data to perform well, especially if fine-tuning is necessary.
- **Diverse Data**: The data should be diverse enough to cover various contexts and nuances related to the problem.

0.0.3 3. Performance Requirements

- **Accuracy and Precision**: Determine if the LLM can meet the accuracy requirements for your task. While LLMs are powerful, they might not always provide the necessary precision for highly specialized tasks without fine-tuning.
- **Real-Time Processing**: Evaluate if the LLM can deliver responses within the required time frame, especially for real-time applications.

0.0.4 4. Cost Considerations

- **Computational Resources**: LLMs demand significant computational power. Consider whether you have the necessary infrastructure or budget to support these resources.
- **Training and Maintenance**: Assess the cost of training (if required) and ongoing maintenance of the LLM-based solution.

0.0.5 5. Ethical and Compliance Factors

- **Bias and Fairness**: LLMs can inherit biases present in the training data. Consider if the risk of biased outputs is manageable and acceptable for your application.
- **Privacy and Security**: Ensure that deploying an LLM solution complies with data privacy regulations and that sensitive information is protected.

0.0.6 6. Alignment with Business Goals

- **Strategic Fit**: Consider if the LLM solution aligns with your overall business strategy and goals. It should support or enhance your core functions or provide a competitive advantage.
- **Scalability and Flexibility**: Evaluate if the LLM solution is scalable and flexible enough to adapt to future business needs or changes in the problem scope.

0.0.7 7. Alternative Solutions

- **Comparison with Other Technologies**: Compare the potential LLM solution with other available technologies. In some cases, traditional machine learning models or rule-based systems might be more suitable.
- **Proof of Concept**: Consider developing a proof of concept to test the feasibility and effectiveness of using an LLM for your problem.

By carefully evaluating these factors, you can determine whether an LLM solution is appropriate for your business problem.

[ ]:

If using gradio UI

[27]:
```
import os
import requests
from bs4 import BeautifulSoup
from typing import List
```

```
from dotenv import load_dotenv
from openai import OpenAI
import google.generativeai
import anthropic

import gradio as gr
```

[28]:
```
# A generic system message - no more snarky adversarial AIs!

system_message = "You are a physicist. You are like Sheldon Cooper from Big␣
 ↪Bang Theory."
```

[29]:
```
# Let's wrap a call to GPT-4o-mini in a simple function

def message_gpt(prompt):
    messages = [
        {"role": "system", "content": system_message},
        {"role": "user", "content": prompt}
      ]
    completion = openai.chat.completions.create(
        model='gpt-4o-mini',
        messages=messages,
    )
    return completion.choices[0].message.content
```

[30]:
```
message_gpt("What is physics ? Is it the one with frogs ? ")
```

[30]:
```
"No, physics is not specifically about frogs. Physics is a branch of science
that deals with the study of matter, energy, and the fundamental forces of
nature. It aims to understand how the universe behaves, from the smallest
subatomic particles to the largest galaxies.\n\nFrogs might be studied within
the realm of biology, particularly in areas such as physiology or ecology, but
they don't form the core focus of physics. However, one could study aspects
related to frogs in a physics context, such as the biomechanics of how they jump
or the acoustics of their croaking. In summary, while there's an interface
between physics and biology, especially in fields like biophysics, physics
itself is much broader and doesn't revolve around any particular organism. If
you have more specific questions about physics or any related topics, feel free
to ask!"
```

User interface time

[31]:
```
# here's a simple function

def shout(text):
    print(f"Shout has been called with input {text}")
    return text.upper()
```

[32]:
```
shout("hello")
```

```
Shout has been called with input hello
```

[32]:
```
'HELLO'
```

[33]:
```
# Adding share=True means that it can be accessed publically
# A more permanent hosting is available using a platform called Spaces from␣
 ↪HuggingFace, which we will touch on next week

gr.Interface(fn=shout, inputs="textbox", outputs="textbox",␣
 ↪flagging_mode="never").launch(share=True)
```

```
* Running on local URL:  http://127.0.0.1:7860
Shout has been called with input jdnsaojdnsajdnsajndjasnjds

Could not create share link. Missing file:
c:\Users\31837\AppData\Local\Programs\Python\Python311\Lib\site-
packages\gradio\frpc_windows_amd64_v0.3.

Please check your internet connection. This can happen if your antivirus
software blocks the download of this file. You can install manually by following
these steps:

1. Download this file: https://cdn-media.huggingface.co/frpc-
gradio-0.3/frpc_windows_amd64.exe
2. Rename the downloaded file to: frpc_windows_amd64_v0.3
3. Move the file to this location:
c:\Users\31837\AppData\Local\Programs\Python\Python311\Lib\site-packages\gradio


<IPython.core.display.HTML object>
```

[33]:

[34]:
```
# Adding inbrowser=True opens up a new browser window automatically

gr.Interface(fn=shout, inputs="textbox",
             outputs="textbox",
             flagging_mode="never").launch(inbrowser=True)
```

```
* Running on local URL:  http://127.0.0.1:7861

To create a public link, set `share=True` in `launch()`.
```

```
<IPython.core.display.HTML object>
```

[34]:

Forcing dark mode Gradio appears in light mode or dark mode depending on the settings of the browser and computer. There is a way to force gradio to appear in dark mode, but Gradio recommends against this as it should be a user preference (particularly for accessibility reasons). But if you wish to force dark mode for your screens, below is how to do it.

[35]:
```
# Define this variable and then pass js=force_dark_mode when creating the␣
  ↪Interface

force_dark_mode = """
function refresh() {
    const url = new URL(window.location);
    if (url.searchParams.get('__theme') !== 'dark') {
        url.searchParams.set('__theme', 'dark');
        window.location.href = url.href;
    }
}
"""
gr.Interface(fn=shout, inputs="textbox", outputs="textbox",␣
  ↪flagging_mode="never", js=force_dark_mode).launch()
```

```
* Running on local URL:  http://127.0.0.1:7862

To create a public link, set `share=True` in `launch()`.
```

<IPython.core.display.HTML object>

[35]:

[36]:
```
# Inputs and Outputs

view = gr.Interface(
    fn=shout,
    inputs=[gr.Textbox(label="Your message:", lines=6)],
    outputs=[gr.Textbox(label="Response:", lines=8)],
    flagging_mode="never"
)
view.launch()
```

```
* Running on local URL:  http://127.0.0.1:7863

To create a public link, set `share=True` in `launch()`.
```

<IPython.core.display.HTML object>

[36]:

[37]:
```
# And now - changing the function from "shout" to "message_gpt"

view = gr.Interface(
```

```
    fn=message_gpt,
    inputs=[gr.Textbox(label="Your message:", lines=6)],
    outputs=[gr.Textbox(label="Response:", lines=8)],
    flagging_mode="never"
)
view.launch()
```

```
* Running on local URL:  http://127.0.0.1:7864

To create a public link, set `share=True` in `launch()`.
```

```
<IPython.core.display.HTML object>
```

[37]:

[38]:
```
# Let's use Markdown
# Are you wondering why it makes any difference to set system_message when it's␣
 ↪not referred to in the code below it?
# I'm taking advantage of system_message being a global variable, used back in␣
 ↪the message_gpt function (go take a look)
# Not a great software engineering practice, but quite sommon during Jupyter␣
 ↪Lab R&D!

system_message = "You are a helpful assistant that responds in markdown"

view = gr.Interface(
    fn=message_gpt,
    inputs=[gr.Textbox(label="Your message:")],
    outputs=[gr.Markdown(label="Response:")],
    flagging_mode="never"
)
view.launch()
```

```
* Running on local URL:  http://127.0.0.1:7865

To create a public link, set `share=True` in `launch()`.
```

```
<IPython.core.display.HTML object>
```

[38]:

[39]:
```
# Let's create a call that streams back results
# If you'd like a refresher on Generators (the "yield" keyword),
# Please take a look at the Intermediate Python notebook in week1 folder.

def stream_gpt(prompt):
    messages = [
        {"role": "system", "content": system_message},
```

```
        {"role": "user", "content": prompt}
      ]
    stream = openai.chat.completions.create(
        model='gpt-4o-mini',
        messages=messages,
        stream=True
    )
    result = ""
    for chunk in stream:
        result += chunk.choices[0].delta.content or ""
        yield result
```

[40]:
```
view = gr.Interface(
    fn=stream_gpt,
    inputs=[gr.Textbox(label="Your message:")],
    outputs=[gr.Markdown(label="Response:")],
    flagging_mode="never"
)
view.launch()
```

```
* Running on local URL:  http://127.0.0.1:7866

To create a public link, set `share=True` in `launch()`.
```

<IPython.core.display.HTML object>

[40]:

[41]:
```
def stream_model(prompt, model):
    if model=="GPT":
        result = stream_gpt(prompt)
    elif model=="Claude":
        result = stream_claude(prompt) # in case if using claude
    else:
        raise ValueError("Unknown model")
    yield from result
```

[42]:
```
view = gr.Interface(
    fn=stream_model,
    inputs=[gr.Textbox(label="Your message:"), gr.Dropdown(["GPT", "Claude"],␣
 ↪label="Select model", value="GPT")],
    outputs=[gr.Markdown(label="Response:")],
    flagging_mode="never"
)
view.launch()
```

```
* Running on local URL:  http://127.0.0.1:7867
```

```
To create a public link, set `share=True` in `launch()`.
```

```
<IPython.core.display.HTML object>
```

[42]:

[ ]:

[45]:
```
# A class to represent a Webpage

class Website:
    url: str
    title: str
    text: str

    def __init__(self, url):
        self.url = url
        response = requests.get(url)
        self.body = response.content
        soup = BeautifulSoup(self.body, 'html.parser')
        self.title = soup.title.string if soup.title else "No title found"
        for irrelevant in soup.body(["script", "style", "img", "input"]):
            irrelevant.decompose()
        self.text = soup.body.get_text(separator="\n", strip=True)

    def get_contents(self):
        return f"Webpage Title:\n{self.title}\nWebpage Contents:\n{self.
 ↪text}\n\n"
```

[46]:
```
# With massive thanks to Bill G. who noticed that a prior version of this had a␣
 ↪bug! Now fixed.

system_message = "You are an assistant that analyzes the contents of a company␣
 ↪website landing page \
and creates a short brochure about the company for prospective customers,␣
 ↪investors and recruits. Respond in markdown."
```

[48]:
```
def stream_brochure(company_name, url, model):
    prompt = f"Please generate a company brochure for {company_name}. Here is␣
 ↪their landing page:\n"
    prompt += Website(url).get_contents()
    if model=="GPT":
        result = stream_gpt(prompt)
    elif model=="Claude":
        result = stream_claude(prompt)
    else:
        raise ValueError("Unknown model")
    yield from result
```

[49]:
```
view = gr.Interface(
    fn=stream_brochure,
    inputs=[
        gr.Textbox(label="Company name:"),
        gr.Textbox(label="Landing page URL including http:// or https://"),
        gr.Dropdown(["GPT", "Claude"], label="Select model")], # in case if␣
 ↪using claude
    outputs=[gr.Markdown(label="Brochure:")],
    flagging_mode="never"
)
view.launch()
```

```
* Running on local URL:  http://127.0.0.1:7868

To create a public link, set `share=True` in `launch()`.
```

<IPython.core.display.HTML object>

[49]:

[ ]: