3

ACCEDIENDO A CHATGPT DESDE LA API DE OPENAI

En el capítulo 2, vimos cómo interactuar con ChatGPT a través de la interfaz gráfica provista por OpenAI, la manera más sencilla e intuitiva de utilizar ChatGPT. En este capítulo, abordaremos la interacción con los modelos de OpenAI a través de la API. Esta técnica nos permitirá un uso más versátil y automatizado de sus funcionalidades. A lo largo de este libro, exploraremos el acceso a estos recursos utilizando el lenguaje de programación Python pero existen módulos y métodos prácticamente análogos para hacerlo a través de Javascript o peticiones curl. Para profundizar en opciones para estos lenguajes te animo a consultar la documentación oficial de OpenAI.

3.1 VENTAJAS DE ACCEDER A CHATGPT DESDE LA API

El acceso a ChatGPT mediante la interfaz gráfica de OpenAI resulta muy sencillo e intuitivo y es la mejor opción a la hora de realizar consultas puntuales a ChatGPT, como por ejemplo, buscar información concreta, aprender sobre un tema determinado o experimentar con su creatividad. Sin embargo, existen muchas casuísticas en las que esta opción resulta demasiado limitada, por ejemplo, si deseamos integrar esa información dentro de una aplicación desarrollada por nosotros mismos. Imaginemos que estamos desarrollando una aplicación para facilitar el aprendizaje de matemáticas a los niños. En este caso, quizá queramos que la aplicación ofrezca distintas funcionalidades incluyendo un chat en el que el niño puede preguntar por conceptos matemáticos y que la aplicación se los explique de manera llana y sencilla sin entrar en tecnicismos que puedan dificultar la comprensión. Para poder integrar ChatGPT en nuestra aplicación y no obligar

a los usuarios a abandonar la aplicación cada vez que quieran utilizar el chat, podemos utilizar la API. La API de OpenAI facilita la realización de consultas de manera programática permitiendo el uso del conocimiento de los modelos GPT en nuestras propias aplicaciones e integrando las respuestas del modelo dentro de nuestra propia interfaz. Veamos algunas de las principales ventajas del acceso mediante la API frente al uso de la interfaz:

- **Integración en nuestros flujos de trabajo.** En el ejemplo anterior, veíamos una forma de poner a disposición del usuario la información de ChatGPT pero en ocasiones podemos desear que esta interacción sea "invisible" para el usuario. Pensemos en un gestor de correo electrónico en el que desarrollamos un pequeño programa que procesando los asuntos de los correos sea capaz de clasificarlos en distintas categorías como "personal", "trabajo" o "publicidad". Con el enfoque de aprendizaje automático tradicional debería entrenarse un modelo de clasificación con miles de mails correctamente clasificados como ejemplo. ChatGPT nos permite utilizar su conocimiento para ahorrarnos esta tarea. Para ello podríamos crear un mensaje que sería algo como: *vas a recibir asuntos de correos electrónicos y tienes que clasificarlos entre las siguientes categorías*... Utilizando Python y la API podemos hacer un programa en el que cada vez que llega un correo, de manera automática se envía esa consulta junto con el asunto del correo recibido. Una vez obtenida la respuesta el correo es recolocado en la carpeta que le corresponde. En este caso, la acción de ChatGPT se encuentra totalmente integrada en nuestra aplicación de manera que el usuario ni siquiera sabría que ChatGPT está siendo utilizado dentro del programa.

- **Personalización de los flujos de trabajo.** Cuando utilizamos la interfaz gráfica, los modelos vienen con una serie de parámetros preajustados de antemano para optimizar el uso generalizado de la aplicación. Al emplear la API tenemos la posibilidad de modificar dichos parámetros ajustando aspectos clave del modelo como su estabilidad o su creatividad. Además, podemos añadir instrucciones clave que afectarán a todas las interacciones con el modelo, pudiendo prefijar aspectos como el idioma, el tono o la finalidad del modelo. De la misma manera, al estar integrando la interacción con los modelos en un flujo de código, podemos realizar labores de procesamiento tanto del mensaje enviado a ChatGPT como del mensaje devuelto por este, para adaptarlo a determinados formatos (truncar su longitud, evitar enviar ciertos tipos de datos o informaciones a ChatGPT…).

- **Automatización y escalabilidad.** La API facilita la creación de procesos automatizados (como el que veíamos en el primer ejemplo) y escalables (permite mandar varias peticiones a la vez pudiendo dar soporte a muchos usuarios al mismo tiempo). Es importante entender que, en el desarrollo de aplicaciones, la escalabilidad es un factor de vital importancia. Las aplicaciones deben estar preparadas para recibir grandes cantidades de peticiones en un momento determinado siendo capaces de escalar y solventarlas, manteniendo siempre un tamaño adecuado para el tráfico de peticiones recibidas. No podemos tener una aplicación que se satura cuando más de cinco usuarios la utilizan a la vez pero tampoco podemos poner grandes cantidades de recursos a disposición de la aplicación si sus recursos solo se van a utilizar de manera masiva durante momentos puntuales.

- **Integración con otros sistemas y fuentes de datos.** Mediante la API podemos enriquecer los conocimientos de ChatGPT con nuestros propios datos almacenados, por ejemplo en ficheros, permitiendo respuestas mucho más adaptadas y satisfactorias. Si pensamos en un chatbot de asistencia al cliente en una página web de compra de ropa, podemos hacer que ChatGPT tenga acceso a la información de los pedidos del usuario para ayudarle a gestionar problemas que pueda haber tenido con ellos o para hacerle recomendaciones personalizadas de cosas que pueden gustarle basado en aquello que ha comprado previamente.

- **Control de formatos y estructuras de salida.** La API de OpenAI permite la adaptación de la salida a ciertos formatos, por ejemplo, JSON lo que nos facilitará aún más la integración de esta información dentro de nuestras propias aplicaciones. Además, frente a la información que la interfaz nos presenta de manera visual al interactuar con ella, la información devuelta por la API es fácilmente manipulable para su uso dentro de nuestro flujo de programación pudiendo volcarse a distintos elementos. Por ejemplo en Python, listas o diccionarios para su mejor manejo. Incluso es posible procesar esta información con otros procesos o modelos antes de presentársela al usuario. A lo largo de este capítulo veremos cómo utilizar estas estructuras para procesar, manipular y almacenar la información obtenida durante las llamadas a la API y en el capítulo 5 estudiaremos las salidas estructuradas que nos permiten volcar la información de las consultas directamente en diccionarios en formato JSON.

Estas son las principales ventajas que presenta el uso de la API de OpenAI frente a su interfaz gráfica. Realmente no existe una forma mejor que otra de interactuar con los modelos de ChatGPT sino que según la finalidad de nuestra tarea elegiremos una u otra. Para consultas rápidas, interacciones individuales y un uso más básico emplearemos la interfaz gráfica por su buen diseño y facilidad de uso. Por el contrario, cuando requiramos una mayor versatilidad o una integración de estas interacciones dentro de un flujo de trabajo escalable y automatizado, utilizaremos la API.

En el capítulo 2 ya vimos cómo interactuar utilizando la interfaz así que a continuación vamos a ver cómo podemos realizar llamadas a la API de OpenAI y cómo interpretar y procesar las respuestas de la API.

3.2 REALIZANDO NUESTRA PRIMERA LLAMADA

Para realizar nuestra primera llamada, lo primero que debemos hacer es instalar el módulo de OpenAI (en caso de que aún no lo tengamos instalado). Para ello podemos instalarlo usando pip desde la terminal o directamente desde un cuaderno de Jupyter de manera prácticamente análoga usando la instrucción:

```
!pip install openai
```

Fragmento de código 3.1. Instalación del módulo openai

> **NOTA**
>
> Si ejecutamos en la terminal deberíamos eliminar el signo de admiración inicial.

Una vez instalado el módulo debemos importarlo para poder trabajar con sus funciones:

```
import openai
```

Fragmento de código 3.2. Importación del módulo openai

A continuación, vamos a almacenar nuestra clave API en una variable. Si no has creado aún tu clave puedes revisar los pasos para hacerlo en el apartado 2.4.1 del segundo capítulo.

```
mi_clave = “inserta aquí tu clave entre comillas”
```

Fragmento de código 3.3. Introducción de la clave API

Una vez hecho esto debemos generar nuestro cliente. Como vimos en el capítulo 2 el cliente es la parte de una API que demanda información y la procesa. Para crear el nuestro, usaremos el módulo de OpenAI que automatiza la construcción de dicho cliente. Para ello solo debemos darle nuestra clave API. Esta clave permitirá a OpenAI monitorizar nuestras interacciones con los modelos para poder cobrar nuestras interacciones así como evitar usos inadecuados. Para crear este cliente utilizamos el código:

```
cliente = openai.OpenAI(api_key=mi_clave)
```

Fragmento de código 3.4, Generación del cliente autenticado

Con estos sencillos pasos ya hemos sido capaces de establecer nuestra conexión con la suite de OpenAI desde Python. Vamos a comprobar que esta conexión funciona de manera correcta haciendo nuestra primera llamada a ChatGPT. Para ello usaremos el siguiente fragmento de código:

```
primera_llamada = cliente.chat.completions.create(
     model=”gpt-4o-mini”,
     messages =[
        {“role”: “system”, “content”: “Eres un ayudante personal”},
        {“role”: “user”, “content”: “¿Podrías decirme la distancia entre Madrid
y París?”}
    ]
```

Fragmento de código 3.5. Primera llamada a GPT

En la primera línea estamos usando el cliente que hemos generado (y que contiene nuestra clave API) para conectarse a los modelos relacionados con la generación de texto contenidos en la clase `chat`. Dentro de esta accedemos a `completions` pues lo que deseamos es enviar un mensaje y que el modelo nos dé la respuesta. A esta función le vamos a pasar dos argumentos: por una parte tenemos `model` en el que indicaremos con qué modelo de todos los disponibles en la suite de OpenAI queremos trabajar y por otra en el argumento “`messages`” introduciremos la lista de mensajes que el modelo va a procesar.

A fecha de edición de este libro los modelos de generación de texto disponibles desde la API de OpenAI son los reflejados en la tabla 3.1.

Modelo	Descripción	Ventana de contexto	Datos de entrenamiento	Versiones
GPT-4o	Modelo GPT más avanzado con capacidades omnicanal y una mayor eficiencia	128.000 tokens	Hasta octubre de 2023	gpt-4o gpt-4o-2024-11-20 gpt-4o-2024-08-06 gpt-4o-2024-05-13
GPT-4o-mini	La versión mini de GPT-4o es el modelo más barato disponible.	128.000 tokens	Hasta octubre de 2023	gpt-4o-mini gpt-4o-mini-2024-07-18
o1	Modelos entrenados para realizar razonamientos lógicos complejos	128.000 tokens	Hasta octubre de 2023	o1-preview o1-preview-2024-09-12 o1-mini o-1 mini-2024-09-12
GPT-4	Modelo de generación de lenguaje especializado en la interacción en formato conversacional.	128.000 tokens (versión turbo) 8.192 tokens (versión base)	Hasta diciembre de 2023 (versión turbo). Hasta septiembre 2021 (versión base)	gpt-4-turbo gpt-4-turbo-2024-04-09 gpt-4-0125-preview gpt-4

Tabla 3.1. Modelos de generación de texto disponibles en la API.

La documentación oficial[25] de OpenAI recoge una lista de todos los modelos disponibles actualizada. Esta actualización implica tanto la aparición de nuevos modelos como la eliminación de algunos que se consideran obsoletos.

La estructura de cada mensaje es un diccionario con dos pares clave-valor. Se utiliza el primer par para definir el rol del emisor del mensaje. Los modelos actuales permiten cuatro roles:

- **System.** El rol `system` se utiliza para dar al modelo unas instrucciones generales que se deben respetar durante toda la interacción. Es una forma de configurar los mensajes con el modelo. Por ejemplo, si le decimos *Eres un ayudante personal que solo responde en inglés*, el chatbot siempre

25 https://platform.openai.com/docs/models#model-endpoint-compatibility

responderá en inglés incluso si el usuario le solicita que le responda en español. Es el rol con mayor jerarquía.

- **Assistant.** Aparecerá con rol `assistant` todos los mensajes devueltos por el modelo. Esto nos permitirá diferenciarlos de aquellos enviados por el usuario y aquellos que hemos fijado como configuración del sistema.
- **User**. El rol `user` engloba todos los mensajes enviados por los usuarios.
- **Tool.** El rol `tool` aparecerá cuando llamemos a alguna herramienta desde nuestra interacción con ChatGPT. Esto nos ayudará a distinguir la información que el asistente devolvería de manera directa, que llevaría el rol `assistant,` de aquella generada utilizando una herramienta. En el capítulo 5, veremos el uso de herramientas con ChatGPT.

Los roles pueden parecer poco prácticas en un principio pero resultan de gran utilidad en algunas aplicaciones en la que puede interesarnos, por ejemplo, medir el consumo de los usuarios y el tipo de consultas que suelen formular sin importarnos las respuestas provistas por el asistente. El rol `tool` es especialmente relevante porque en ocasiones podemos querer aplicar un preprocesamiento especial a los resultados venidos de una herramienta por lo que esta sería una forma sencilla de distinguirlos del resto de respuestas.

En el otro par clave-valor del diccionario que conforma un mensaje tendremos la clave `content` y como valor el mensaje que queramos enviar. Como puedes observar, podemos enviar mensajes con rol `system` para configurar nuestra interacción, pero también podríamos enviar mensajes con rol `assistant` para dar ejemplos del tipo de respuestas que queremos obtener por parte del asistente.

Así en el fragmento de código 3.5, simplemente lo que hemos hecho es llamar al modelo gpt-4o-mini configurándolo como un ayudante personal y preguntándole la distancia entre Madrid y París.

La ejecución del fragmento de código 3.5, nos devolvería una respuesta similar a la siguiente:

```
ChatCompletion(id='chatcmpl-AZcyhVatILIpY6jNWtUdiu4nWggLo',
choices=[Choice(finish_reason='stop', index=0, logprobs=None, message=ChatCompl
etionMessage(content='La distancia entre Madrid y París es de aproximadamente
1,050 kilómetros (alrededor de 650 millas) si se mide en línea recta. Sin
embargo, la distancia por carretera puede ser mayor, alrededor de 1,250
kilómetros (aproximadamente 780 millas), dependiendo de la ruta que se
tome.', refusal=None, role='assistant', audio=None, function_call=None, tool_
calls=None))], created=1733054675, model='gpt-4o-mini-2024-07-18', object='chat.
```

```
completion', service_tier=None, system_fingerprint='fp_0705bf87c0', usage=Comple
tionUsage(completion_tokens=67, prompt_tokens=30, total_tokens=97, completion_
tokens_details=CompletionTokensDetails(accepted_prediction_tokens=0, audio_
tokens=0, reasoning_tokens=0, rejected_prediction_tokens=0), prompt_tokens_detai
ls=PromptTokensDetails(audio_tokens=0, cached_tokens=0)))
```

Fragmento de código 3.6. Objeto ChatCompletion

Las respuestas de la API de OpenAI suelen ser objetos construidos específicamente por el módulo OpenAI y en este caso nos encontramos ante un objeto `ChatCompletion`. Veamos, parte por parte, los atributos más relevantes de este tipo de objeto:

- El `id` es un identificador único de nuestra interacción. No existirán otras interacciones con el mismo identificador dentro de nuestros proyectos.
- El atributo `choices` almacena toda la información relacionada con la respuesta.
- El atributo `created` indica la fecha y hora de emisión de la respuesta en formato timestamp.
- El atributo `model` indica el modelo usado para la generación de esta respuesta.
- El atributo `usage` monitoriza el consumo del modelo y utiliza como unidad de medida los tokens procesados. Un poco más adelante veremos qué es exactamente un token, pero de momento se puede pensar como una unidad equivalente a una palabra.

Los atributos `choices` y `usage` tienen a su vez distintos estamentos de información en su interior como veremos a continuación.

- Accediendo a `choices` mediante el operador`.` podemos comprobar que es una lista de objetos `Choice` construidos específicamente por OpenAI. Dentro de estos objetos encontramos los siguientes atributos:
- La `finish_reason` indica cómo ha finalizado la comunicación con el modelo. Tomará el valor "stop" cuando la petición haya terminado y hayamos obtenido una respuesta, "length" si alcanzamos el máximo número de tokens permitidos a la petición, "content_filter" si se omitió contenido porque hayamos añadido filtros de contenido a nuestra petición o "tool_calls" si nuestra petición desencadenó una llamada a alguna

herramienta. En el tercer apartado del capítulo 5, veremos cómo podemos invocar herramientas desde los modelos GPT.

- El atributo `message` es a su vez un objeto de OpenAI que tiene como propiedades: el "content" que almacena el contenido del mensaje, "refusal" que contiene el mensaje de rechazo en caso de que el modelo no haya aceptado la petición, "tool_calls" con las llamadas realizadas a herramientas y el "role" con el rol del autor de dicho mensaje que tomará alguno de los valores mencionados previamente (assistant, user o tool).

Por otra parte, accediendo a `usage` podemos entender mejor los recursos medidos como tokens que hemos consumido en cada petición. Dentro de este atributo los tokens consumidos se separan en:

- `Completion_tokens` son aquellos empleados por el modelo para dar respuesta a nuestra consulta.
- `Prompt_tokens` son aquellos empleados en el prompt que enviamos al modelo.
- `Total_tokens` es la suma de los `completion_tokens` y los `prompt_tokens`.

Existen otros atributos relacionados que exploraremos más adelante pero por el momento iremos trabajando con estos que son los más utilizados en el día a día en la interacción con modelos GPT.

Antes de continuar con la exploración de las respuestas de ChatGPT merece la pena detenerse brevemente en el concepto de token.

Def. Un **token** es una unidad de texto que se utiliza para procesar el lenguaje.

Dependiendo del idioma los tokens pueden ir desde una palabra completa (en idiomas con palabras cortas), una parte de una palabra (sufijos, prefijos…) o incluso un caracter individual (por ejemplo, signos de puntuación).

En el caso del español, dado que es un idioma rico en conjugaciones, sufijos y prefijos, una palabra suele representar entre 1.5 y 2 tokens de media. En inglés, sin embargo, una palabra suele representar entre 1 y 1.5 tokens.

Los tokens son una herramienta fundamental dentro del procesamiento de lenguaje natural ya que permiten modelar algo tan complejo como son los textos en pequeñas unidades sobre las que se establece una correspondencia numérica. De esta

manera los modelos pueden procesar caracteres especiales y dividir los textos de una manera mucho más granular que si se utilizaran palabras, mejorando el rendimiento y la eficiencia de estos modelos.

Hasta aquí hemos visto cómo utilizar la API de OpenAI en Python para realizar llamadas a modelos GPT y obtener una respuesta. Sin embargo, como hemos visto, esta respuesta viene dada en un formato muy específico y no tan fácilmente legible o incluso usable, por lo que a continuación vamos a construir funciones que nos ayuden a procesar, almacenar y visualizar esta información.

ⓘ NOTA

En esta sección, a fin de no hacer la lectura demasiado tediosa, se han abordado sólo los atributos y parámetros más relevantes. A lo largo de este libro, y a medida que otros parámetros o atributos se vayan utilizando, se procederá a su explicación detallada. Puedes encontrar la información referente a todos los parámetros en la documentación oficial de OpenAI[26].

3.3 FUNCIONES DE PYTHON PARA INTERPRETAR LOS RESULTADOS

Tras realizar nuestra primera llamada al modelo GPT y obtener un resultado tan complejo como el que vimos en el fragmento de código 3.6, a continuación vamos a ver cómo explotar esta salida. Para ello usaremos la siguiente función:

```
from datetime import datetime
def procesar_respuesta(respuesta, respuesta_completa=False):
   “””” Función que procesa la respuesta de ChatGPT generando un diccionario con
la información más relevante y un mensaje legible para el usuario”””
   diccionario_informacion = {}
   contenido = respuesta.choices[0].message.content
   diccionario_informacion[“mensaje”] = contenido
   motivo_fin = respuesta.choices[0].finish_reason
   rol = respuesta.choices[0].message.role
   diccionario_informacion[“rol”] = rol
   diccionario_informacion[“motivo_fin”] = motivo_fin
    fecha_creacion = datetime.fromtimestamp(respuesta.created).strftime(“%Y-%m-
%d %H:%M:%S”)
   diccionario_informacion[“fec_creacion”] = fecha_creacion
   modelo = respuesta.model
```

26 https://platform.openai.com/docs/api-reference/chat/object

```
    diccionario_informacion[“modelo”] = modelo
    tokens_usados = (respuesta.usage.prompt_tokens, respuesta.usage.completion_
tokens, respuesta.usage.total_tokens)
    diccionario_informacion[“tokens”] = tokens_usados
    if respuesta_completa:
        mensaje = f”{contenido} \nEl mensaje ha sido emitido por {modelo} a fecha
{fecha_creacion}. El mensaje ha consumido un total de {tokens_usados[2]} tokens,
siendo usados {tokens_usados[0]} en nuestro prompt y {tokens_usados[1]} en la
respuesta.”
    else:
        mensaje = contenido
    if motivo_fin!=”stop”:
        mensaje = mensaje + “ La ejecución ha sido interrumpida de manera
inesperada.”
    return diccionario_informacion, mensaje
```

Fragmento de código 3.7. Función que procesa las respuestas del modelo GPT

Esta función recibe como parámetros el objeto respuesta de la llamada al modelo y un booleano (puesto por defecto a False). El booleano respuesta_completa nos permite modular el grado de detalle que deseamos obtener como respuesta de nuestra función. La función nos devolverá un diccionario con toda la información de manera estructurada y accesible y un string con el mensaje de respuesta formateado.

La función comienza creando un diccionario vacío y va almacenando en dicho diccionario los distintos atributos vistos en la sección anterior mientras los formatea de manera que se facilite su comprensión y posterior utilización. Por ejemplo, transforma la fecha timestamp en una fecha comprensible para el usuario. Además en el string mensaje, almacena de manera legible información sobre la llamada así como el contenido si el parámetro respuesta completa es fijado como True y solo el contenido de la respuesta si está fijado a False. Por último, si la `finish_reason` no es “stop” se lanza un aviso para el usuario dentro del mensaje.

La ejecución de la función sobre la respuesta obtenida previamente (fragmento de código 3.8) devuelve dos objetos como hemos visto; un diccionario y un string:

```
dic_primera_llamada, mensaje_primera_llamada = procesar_respuesta(primera_
llamada, True)
```

Fragmento de código 3.8. Ejecución de la función para procesar la respuesta

Explorando un ejemplo de diccionario vemos cómo la información más relevante ha quedado claramente estructurada en pares clave-valor en el que las claves tienen nombres útiles e intuitivos:

```
{'mensaje': 'La distancia entre Madrid y París es de aproximadamente 1,050
kilómetros (alrededor de 650 millas) si se mide en línea recta. Sin embargo,
la distancia por carretera puede ser mayor, alrededor de 1,250 kilómetros
(aproximadamente 780 millas), dependiendo de la ruta que se tome.',
'rol': 'assistant',
'motivo_fin': 'stop',
'fec_creacion': '2024-12-01 12:04:35',
'modelo': 'gpt-4o-mini-2024-07-18',
'tokens': (30, 67, 97)}
```

Fragmento de código 3.9. Diccionario con información de respuesta estructurada

Por otra parte, el string, como la llamada se ha hecho con el indicador True contendrá un mensaje completo que resume la interacción:

```
La distancia entre Madrid y París es de aproximadamente 1,050 kilómetros
(alrededor de 650 millas) si se mide en línea recta. Sin embargo, la distancia
por carretera puede ser mayor, alrededor de 1,250 kilómetros (aproximadamente
780 millas), dependiendo de la ruta que se tome.
El mensaje ha sido emitido por gpt-4o-mini-2024-07-18 a fecha 2024-12-01
12:04:35. El mensaje ha consumido un total de 97 tokens, siendo usados 30 en
nuestro prompt y 67 en la respuesta.
```

Fragmento de código 3.10. Contenido del mensaje de la respuesta

Esta pequeña función (fragmento de código 3.7) nos permite hacer la información mucho más fácil de procesar con el diccionario y de comprender con el mensaje. Si quisiéramos montar un flujo, por ejemplo, para pedir que una petición se reenvíe en caso de que la respuesta no se haya devuelto correctamente, podríamos acceder a esta información de manera muy intuitiva utilizando el diccionario y la clave "motivo_fin" y simplemente usar un condicional `if` para realizar de nuevo la llamada cuando el motivo sea distinto de stop.

La función `procesar_respuesta` es una función destinada a facilitar los códigos para el trabajo a lo largo de este libro que aborda solo los atributos más útiles. La lógica de esta función sin embargo es aplicable a cualquier atributo contenido en la respuesta generada por OpenAI por lo que si consideras relevante otra información de la respuesta, puedes modificar la función en los cuadernos para añadirla como un nuevo atributo al diccionario de salida.

3.4 EL PROBLEMA DEL CONTEXTO

Uno de los grandes problemas que tendremos que abordar cuando decidamos utilizar la API para la construcción de asistentes y chatbots es la gestión del contexto. Cuando mantenemos una conversación con ChatGPT a partir de la interfaz de OpenAI, ChatGPT es capaz de mantener el hilo conversacional de manera que se puede producir un diálogo como el que sigue:

Usuario: ¿Cuál es la capital de Francia?
ChatGPT: París
Usuario: ¿Y la de Italia?
ChatGPT: Roma

El asistente, al igual que si habláramos con un ser humano, mantiene el contexto de la conversación y usa esta información para construir su respuesta por lo que no necesitamos preguntarle "¿Cuál es la capital de Italia?" para que entienda que eso es lo que queremos saber con nuestra segunda interacción.

Sin embargo si realizamos estas dos preguntas como consultas desde la API no será capaz de entender la segunda y no nos podrá dar una respuesta satisfactoria. Puedes ver el ejemplo en el repositorio de código en el cuaderno de Jupyter: 4.2. Construyendo nuestra propia interfaz para ChatGPT.

Para resolver este problema lo que debemos hacer es asegurarnos de que al realizar las llamadas a la API se le cargue todo el contexto, tanto las respuestas dadas previamente por el modelo como las consultas enviadas por el usuario. De esta manera cuando el modelo se ponga a inferir la respuesta contará con toda la información disponible para construir la mejor réplica posible. El siguiente código muestra una manera en la que podríamos construir nuestra variable mensajes que posteriormente utilizaremos en la llamada a la API:

```
mensajes = [
         {"role": "system", "content": "Eres un ayudante personal"},
         {"role": "user", "content": "¿Cuál es la capital de Francia?"}
     ]

llamada_ciudad_1 = cliente.chat.completions.create(
     model="gpt-4o-mini",
     messages =mensajes
 )
diccionario_ciudad_1, mensaje_ciudad_1 = procesar_respuesta(llamada_ciudad_1)
print("Mensaje 1:", mensaje_ciudad_1)
mensajes.append({"role" : "assistant", "content" : diccionario_
```

```
ciudad_1["mensaje"]})
mensajes.append({"role" : "user", "content" : "¿Y la de Italia?"})

llamada_ciudad_2 = cliente.chat.completions.create(
     model="gpt-4o-mini",
     messages =mensajes
 )
diccionario_ciudad_2, mensaje_ciudad_2 = procesar_respuesta(llamada_ciudad_2)
print("Mensaje 2:", mensaje_ciudad_2)
```

Fragmento de código 3.11. Envío de mensajes con distintos roles

Como vemos una vez recibida la primera respuesta y tras imprimir el mensaje 1 lo que hacemos es añadir a la lista de `mensajes` tanto la respuesta del modelo como nuestra nueva consulta. Así al llamar a la función por segunda vez el modelo tiene acceso a toda la información y devuelve la respuesta correcta.

Una vez vista una primera aproximación para la resolución del problema del contexto vamos a construir nuestro propio prototipo de interfaz para interactuar con ChatGPT capaz de abordar este problema.

3.5 CONSTRUCCIÓN DE UN PROGRAMA PARA INTERACTUAR CON CHATGPT

3.5.1 Construcción de una interfaz básica

En esta última sección del capítulo, vamos a poner en práctica todo lo aprendido hasta ahora construyendo nuestra propia función de Python que simula un interfaz de interacción con ChatGPT. Este tipo de programas pueden ser especialmente útiles cuando queremos incluir en una aplicación o página web, desarrollada por nosotros mismos, un chatbot apoyado en la potencia de los modelos GPT. De esta forma podríamos introducir unos primeros prompts con instrucciones y la información necesaria para el modelo que serían invisibles para el usuario pero gracias a la conservación del contexto configurarían las respuestas del chatbot. Por ejemplo, si tenemos una web de viajes podríamos añadir un prompt inicial que dijera algo como “Eres un chatbot en una agencia de viajes virtual” y posteriormente añadir una serie de instrucciones como: “Pregúntale al usuario por sus preferencias en viajes”, “Sugiérele al usuario destinos que se adapten a dichas preferencias”, “Plantea al usuario cuáles son buenas fechas para visitar estos destinos y por qué”. Mediante estos sencillos pasos estaríamos construyendo una primera versión de un

chatbot en código Python que luego podríamos conectar a nuestra página web o nuestra app. ¡Veamos cómo llevar esta idea a la práctica mediante código!

En este caso, para recibir los mensajes del usuario en un notebook de Python usaremos la función `input()`. Cada vez que esta función se active se abrirá un espacio dentro del cuaderno o terminal en el que estamos ejecutando para que el usuario pueda escribir y se pausará el flujo de ejecución hasta que el usuario introduzca la información pertinente. Una vez el usuario envíe el mensaje, el flujo de ejecución se reanudará. A continuación se presenta el código de dicha función:

```
def mi_propio_ChatGPT(cliente, mensajes_iniciales, modelo="gpt-4o-mini",
monitorizar_uso=True):
 """Función que genera un pequeño interfaz para interactuar con ChatGPT"""
 print("Estoy listo para hablar contigo.")
 numero_tokens = 0
 while True:
   mensaje_usuario = input("Usuario: ")
   if mensaje_usuario=="q":
     break
   mensajes_iniciales.append({"role" : "user", "content" : mensaje_usuario})
   llamada_modelo = cliente.chat.completions.create(
       model=modelo,
       messages = mensajes_iniciales
   )
   diccionario_respuesta, mensaje_respuesta = procesar_respuesta(llamada_modelo)
   print(mensaje_respuesta)
   mensajes_iniciales.append({"role" : diccionario_respuesta["rol"], "content" :
mensaje_respuesta})
   numero_tokens = numero_tokens + diccionario_respuesta["tokens"][2]
   if monitorizar_uso:
     mensaje_uso = f"Hasta ahora se han procesado un total de {numero_tokens}
tokens en esta interacción."
     print(mensaje_uso)

 print("La función se ha detenido.")
 print(f"En total durante la conversación se han consumido {numero_tokens}
tokens.")
```

Fragmento de código 3.12. Versión inicial de nuestro propio ChatGPT

La función toma cuatro parámetros: `cliente` que es la conexión a la API de OpenAI que utilizaremos, `mensajes_iniciales` que son los mensajes iniciales en los que configuraremos las instrucciones del asistente, `modelo` en el que indicaremos el modelo GPT en el que queremos que se apoye nuestro chatbot y `monitorizar_uso`

configurado a True por defecto que indicará con cada interacción el gasto de tokens realizado. La función no devuelve ningún objeto ya que el objetivo es simular por pantalla la interacción con un chatbot. Los únicos outputs que devuelve la función serán todos los mensajes que se le presentan al usuario en la pantalla mediante el método `print()`. Más adelante, veremos cómo modificar esta función para almacenar la conversación en un objeto que luego pueda ser reutilizable.

La función comienza emitiendo un mensaje indicando que ya está en funcionamiento e inicializando la variable `numero_tokens` a 0. Usaremos esta variable para controlar el gasto de tokens en las interacciones. Tras ello entramos en un bucle potencialmente infinito y abrimos el canal de comunicación para el usuario. El flujo de ejecución se detiene hasta que el usuario introduce un mensaje: si ese mensaje es el comando "q" el bucle se rompe y se informa mediante los prints de las dos últimas líneas que la ejecución se ha detenido y el número de tokens enviados y recibidos a través de la API de OpenAI. En caso contrario este mensaje se añade a la lista de mensajes asignándole el rol de usuario.

Tras esto se realiza la llamada al modelo asignado mediante el parámetro `modelo` enviando todos los mensajes almacenados en la lista hasta el momento y se almacena la respuesta en `llamada_modelo`. Procesamos esta respuesta mediante nuestra función `procesar_respuesta` (fragmento de código 3.7) y obtenemos por una parte el diccionario con información y por otra el string con el contenido del mensaje que mostramos en pantalla para que el usuario pueda conocer la respuesta del asistente. Para lograr la conservación de contexto de la que hablábamos en el apartado anterior añadimos el mensaje del asistente a la lista de mensajes extrayendo la información de rol del diccionario. Usamos también el diccionario para incrementar el número de tokens usando los tokens totales que incluyen los de la consulta del usuario así como los de la respuesta del modelo.

ⓘ NOTA

Es importante entender que como cada vez le estamos enviando toda la conversación al modelo para que pueda tener el contexto, el crecimiento del número de tokens será exponencial a medida que la conversación se vaya alargando.

Si el indicador de `monitorizar_uso` está en True mostramos también por pantalla en cada iteración el número de tokens procesados. Llegados a este punto el bucle se reinicia y es de nuevo el turno del usuario para enviar una consulta o el comando de cierre "q".

Para inicializar el chatbot con la configuración de la agencia de viajes debemos introducir las instrucciones mediante nuestra lista de mensajes iniciales:

```
mensajes_iniciales_agencia_viajes = [{"role" : "system", "content" : "Eres un
chatbot en una agencia de viajes virtual"},
                        {"role" : "system", "content" : "Pregúntale al usuario
por sus preferencias en viajes"},
                        {"role" : "system", "content" : "Sugiérele al usuario
destinos que se adapten a dichas preferencias"},
                        ]
```

Fragmento de código 3.13. Mensajes de configuración para el agente de viajes

¡Si ejecutamos la función con esta lista de mensajes iniciales ya tenemos en marcha nuestro asistente de viajes! Puedes consultar en el repositorio de código asociado al libro el cuaderno 3.2. "Construyendo nuestra propia interfaz para ChatGPT" para ver un ejemplo de interacción en el que el chatbot sigue nuestras instrucciones recomendando destinos y fechas específicas (Figura 3.1).

3.5.2 Almacenando conversaciones

Ya tenemos un chatbot funcionando capaz de utilizar la potencia de los modelos GPT para comprender los mensajes de los usuarios y emitir respuestas coherentes configuradas por nuestras propias instrucciones. Para completar esta interfaz podemos añadir, como ya existe en la interfaz de OpenAI, una funcionalidad que almacene las conversaciones mantenidas por el usuario previamente. Esto permitirá a los usuarios retomar conversaciones donde lo dejaron y a nosotros como administradores analizar y tener una trazabilidad sobre las interacciones de nuestros usuarios lo que nos permitirá medir la calidad de nuestro chatbot así como encontrar puntos de mejora o casos que se deben corregir mediante una configuración más adecuada. Para ello lo que vamos a hacer es construir una versión mejorada (ver fragmento de código 3.13) en la que añadimos un nuevo parámetro `ruta_conversacion`. Este parámetro será una ruta al lugar donde se almacenan las conversaciones, si la ruta existe, es decir, si el usuario está accediendo a una conversación previa, se cargarán los mensajes de la conversación anterior. Por el contrario, si la ruta no apunta a un archivo existente, se iniciará una nueva conversación. Enviamos, en ambos casos, un mensaje por pantalla al usuario para que sepa si está interactuando sobre una conversación previa o empezando una nueva.

En esta implementación, hemos decidido almacenar las conversaciones en formato json. Este formato nos permite almacenar la información en listas de pares de clave-valor por lo que se adecúa especialmente al almacenamiento de estas conversaciones. Si se deseara se podría almacenar en cualquier otro formato (.txt, .csv…) pero en nuestro caso, el uso de archivos json nos permitirá evitar

procesamientos y preprocesamientos para el guardado y lectura de las conversaciones simplificando significativamente la complejidad de estas funciones.

Como vemos en el código usaremos el método `json.load()` para cargar la información almacenada en archivos json en un diccionario de Python. Complementariamente, utilizaremos el método `json.dump()` para volcar la información de un diccionario a un archivo. Los argumentos `encoding` y `ensure_ascii` nos permiten guardar los archivos tolerando caracteres que en ocasiones son problemáticos para escribirse y leerse en Python como las tildes o los signos de abrir interrogación. El `indent=4` garantiza una mejor legibilidad de los archivos json cuando los abrimos para leerlos desde un editor de texto.

```
import json
import os
def mi_propio_ChatGPT_con_memoria(cliente, configuración, ruta_conversacion,
modelo="gpt-4o-mini", monitorizar_uso=True):
 """Función que genera un pequeño interfaz para interactuar con ChatGPT"""
 print("Asistente: Estoy listo para hablar contigo.")
 numero_tokens = 0
 if os.path.exists(ruta_conversacion):
   print("Cargando mensajes anteriores")
   with open(ruta_conversacion, "r", encoding="utf-8") as archivo_lectura_
conversacion:
     mensajes_almacenados = json.load(archivo_lectura_conversacion)
 else:
   print("Iniciando una nueva conversación")
   mensajes_almacenados = configuracion.copy()
 while True:
   mensaje_usuario = input("Usuario: ")
   if mensaje_usuario=="q":
     break
   mensajes_almacenados.append({"role" : "user", "content" : mensaje_usuario})
   llamada_modelo = cliente.chat.completions.create(
       model=modelo,
       messages = mensajes_almacenados
   )
   diccionario_respuesta, mensaje_respuesta = procesar_respuesta(llamada_modelo)
   print("Asistente: ", mensaje_respuesta)
   mensajes_almacenados.append({"role" : diccionario_respuesta["rol"], "content"
: mensaje_respuesta})
   numero_tokens = numero_tokens + diccionario_respuesta["tokens"][2]
   if monitorizar_uso:
```

```
      mensaje_uso = f"Hasta ahora se han procesado un total de {numero_tokens}
  tokens en esta interacción."
      print(mensaje_uso)

   print("La función se ha detenido.")
   print(f"En total durante la conversación se han consumido {numero_tokens}
  tokens.")
   try:
     with open(ruta_conversacion, "w", encoding="utf-8") as archivo_conversacion:
       json.dump(mensajes_almacenados, archivo_conversacion, ensure_ascii=False,
  indent=4)
     print(f"Conversación almacenada en {ruta_conversacion}")
   except:
     print("No hay datos para almacenar")
```

Fragmento de código 3.14. Versión mejorada de nuestro propio ChatGPT

Al ejecutar el código del fragmento 3.14, se abrirá un terminal en el que podremos simular nuestra conversación obteniendo respuestas directas de ChatGPT. En la figura 3.1. se presenta el resultado de una simulación de conversación con esta función:

```
Asistente: Estoy listo para hablar contigo.
Iniciando una nueva conversación
Usuario: Quiero irme de vacaciones a una gran ciudad
Asistente:  ¡Claro! Las grandes ciudades ofrecen una gran variedad de cultura, gastronomía y actividades. Para poder recomendarte el mejor destino, ¿tienes alguna preferencia en cuanto a:

1. **Continente o región**: ¿Buscas algo en Europa, América, Asia, etc.?
2. **Tipo de actividad**: ¿Te interesa la historia, el arte, las compras, la vida nocturna o un mix de todo?
3. **Época del año**: ¿Hay alguna fecha en particular en la que planeas viajar?
4. **Duración del viaje**: ¿Cuánto tiempo planeas estar de vacaciones?

Con esta información, podré sugerirte algunas grandes ciudades que se adapten a tus gustos.
Hasta ahora se han procesado un total de 1142 tokens en esta interacción.
Usuario: Me gustaría algo en Asia con muchos museos y arte. Mi idea es estar unas dos semanas
Asistente:  ¡Excelente elección! Asia ofrece algunas de las grandes ciudades más vibrantes y ricas en cultura y arte. Aquí tienes algunas recomendaciones que podrían interesarte:

### 1. **Tokio, Japón**
   - **Museos**: El Museo Nacional de Tokio y el Museo de Arte Contemporáneo de Tokio son imperdibles.
   - **Arte**: Puedes explorar el barrio de Harajuku para arte urbano y el famoso parque Ueno, que alberga varios museos.
   - **Experiencia**: La combinación de tradición y modernidad es fascinante, ¡no te pierdas el cruce de Shibuya!

### 2. **Seúl, Corea del Sur**
   - **Museos**: El Palacio de Gyeongbokgung y el Museo de Arte Moderno y Contemporáneo son destacados.
   - **Arte**: Visita el área de Hongdae, conocida por su vibrante escena artística y cultural.
   - **Experiencia**: La mezcla de cultura tradicional y contemporánea, además de su deliciosa gastronomía.

### 3. **Pekín, China**
   - **Museos**: La Ciudad Prohibida y el Museo Nacional de China son imprescindibles.
   - **Arte**: El Distrito 798 es un área artística que alberga galerías contemporáneas y modernas.
   - **Experiencia**: Además de explorar los museos, puedes visitar la Gran Muralla y degustar la famosa Pato Pekinés.

### 4. **Bangkok, Tailandia**
   - **Museos**: El Museo de Arte de Bangkok y el Museo Nacional son muy interesantes.
   - **Arte**: La zona de Chatuchak tiene un mercado de arte donde puedes encontrar piezas únicas.
   - **Experiencia**: La vida nocturna y los mercados flotantes son atracciones adicionales a la cultura.

### 5. **Singapur**
   - **Museos**: El Museo de Arte e Historia y el ArtScience Museum son altamente recomendados.
   - **Arte**: El barrio de Kampong Glam y el área de Little India ofrecen murales y arte callejero vibrante.
   - **Experiencia**: Disfruta de una ciudad moderna con una mezcla de culturas, y no te pierdas los Jardines de la Bahía.

### Sugerencias para tu viaje:
- **Transporte**: Investiga sobre el transporte público, ya que muchas de estas ciudades tienen sistemas de metro muy eficientes.
- **Turismo cultural**: Muchos museos ofrecen días con entrada gratuita o a precios reducidos, así que planifica tus visitas.

¿Te gustaría más información sobre alguno de estos destinos, o necesitas ayuda con otros aspectos de tu viaje?
Hasta ahora se han procesado un total de 2873 tokens en esta interacción.
Usuario: q
La función se ha detenido.
En total durante la conversación se han consumido 2873 tokens.
Conversación almacenada en ./conversaciones/conversacion_1.json
```

Figura 3.1. Simulación de una conversación con el interfaz de ChatGPT implementado.

3.6 RECAPITULACIÓN

A lo largo de este capítulo hemos visto cómo el uso de la API de OpenAI nos permite conectar con los modelos GPT y recibir respuestas sin salirnos de Python en ningún momento. Aunque para la emisión de consultas y la búsqueda de información la interfaz de OpenAI resulta mucho más cómoda y visual, el uso de la API nos permite procesar esta información de manera programática, customizar la salida para el usuario o almacenar parte de esta información en nuestros sistemas. Hemos llegado a generar una función capaz de conectarse a los modelos GPT y simular una interfaz, resolviendo de manera algo rudimentaria el problema del contexto (en el capítulo 8 veremos técnicas más elaboradas para abordar este problema) e incluso almacenando a posteriori las conversación y permitiendo su carga para seguir interactuando en un mismo hilo.

En el siguiente capítulo, pondremos el foco en la ingeniería de prompts. El objetivo es establecer procesos que nos permitan mejorar nuestras consultas logrando resultados que se adecúen mejor a nuestros requerimientos y las respuestas que estamos esperando. Se explicarán distintas técnicas y buenas prácticas que nos permitirán explotar toda la potencialidad de los modelos GPT (y en general, cualquier gran modelo de lenguaje) a través de nuestras consultas.