SISTEMAS COMPUTACIONAIS AVANÇADOS (SISTCA) ADVANCED COMPUTING SYSTEMS

Degree in **T**elecommunications and **I**nformatics **E**ngineering

Instituto **S**uperior de **E**ngenharia do **P**orto, **P**olitécnico do **P**orto

Lab Classes Script:
OpenAI

Version	Date	Authors	Update information
V1.0	April 2024	Patrícia Sousa (1210713), Carlos Alves (1211604), José Leal(1211066), Tiago Ribeiro (1210924) Supervisor: Paula Viana (pmv)	Original version

Table 1: Version Control

Contents

List of Tables

List of Figures

1 Introduction

1.1 Context

In this script we will present an API developed by openAI, one of the leading organisations in artificial intelligence (AI) research and development. Launched in 2020, it represents a significant milestone in accessing advanced AI technology. It offers an accessible and simplified interface for developers to integrate AI capabilities into a variety of applications, from text analysis to content generation.

The OpenAI API is based on recent advances in machine learning, in particular, deep neural network architectures, such as GPT (Generative Pre-trained Transformer) language models. These models have revolutionised the way machines understand and generate natural language, enabling tasks such as automatic translation, text summarising, question answering, and content generation.

GPT is trained with a vast amount of textual data from the internet, allowing it to capture the nuances and complexities of human language. Through the pre-training process, the model learns to represent knowledge in a general way, without the need for specific training for a particular task.

The OpenAI API leverages these pre-trained language models, providing a simple and effective interface for developers to take advantage of their capabilities. This means users can easily integrate advanced AI capabilities into their applications, without needing to understand complex implementation details or model training.

Additionally, it is designed with a focus on security and ethics, including measures to prevent malicious or harmful use of AI technology. This reflects OpenAI's commitment to promoting the responsible development of artificial intelligence and ensuring that its benefits are extremely accessible and used for the good of society.

Lastly, for a more interactive experience for the user, this script is available in another format, jupyter notebook. A free open-source software launched in 2014 by IPython that provides a notebook, this is, a shareable document that combines computer code, plain language descriptions, data, rich visualizations like 3D models, charts, graphs and figures, and interactive controls in one file, creating a fast and easy way to prototype and explain code, visualize data, and share ideas [1].

1.2 Motivation

One of the main motivations for choosing this topic lies in the importance of artificial intelligence in society today, some of whose main impacts include the automation and efficiency of tasks, more accurate medical diagnoses, learning adapted to the needs of each student, as well as the existence of virtual assistants.

By integrating artificial intelligence into various aspects of society, it is possible to solve complex problems, improve quality of life and drive innovation. However, it is essential that the development and implementation of AI is carried out in an ethical and responsible manner, taking into account privacy and security issues to ensure that the benefits are widely distributed and the risks mitigated.

In choosing this subject, it is recognised that there is a need to become familiar not only with the fundamental concepts of artificial intelligence, but also with the practical tools for its implementation. To this end, it is necessary not only to master theoretical principles, but also to explore programming languages and accessible platforms, such as the OpenAI API.

Whether in the SISTCA course or in other contexts, the OpenAI API offers a versatile and robust environment for exploring the potential of artificial intelligence.

1.3 Objectives

The main objective of this script is to explore the various features offered by the OpenAI API and learn how to integrate them, promoting a comprehensive and practical understanding of the platform's capabilities, providing users with a solid foundation to explore and use the following available tools (2.2):

- **Chat Completions**, to understand the generic implementation of chat GPT based assistant;
- **Assistants API + Tools** To create a program that provides the user with media suggestions such as books, films or TV shows;

- **Embeddings** Learn how to represent words, sentences, paragraphs, or entire documents in a continuous vector space;
- **Vision** The use of Vision functionality to extract information from an image.
- **Image Generation/DALL-E** Learn how to generate AI created images.
- **TTS (Text to Speech)** Turn strings into a robotised speech using AI;
- **Whisper** Capture an audio and translate it to different languages;
- **Moderation** Explore OpenAI's Moderation functionality to detect inappropriate content.

In addition, two practical exercises will be developed to apply the acquired knowledge and a challenge will be proposed to test the reader's understanding and ability to use the OpenAI API.

1.4 Document Structure

This tutorial is organised into seven chapters with the following structure: Introduction, Theoretical, Setup/Installation, Tutorial/Functionality, Exercises followed by a Challenge and ends with a closing chapter.

In the Introduction, some concepts about the OpenAI API will be presented that will be deepened throughout the script.

The theoretical part, will introduce the State-of-the-art and detail the API's Features.

Then, the third chapter focuses on Setup/Installation, explaining how to create an open AI account and set up a development environment.

In the Tutorial/Functionality part, a detailed tutorial of the available features will be provided.

After all these topics, two exercises will be developed and their corresponding resolution will be presented and a final challenge to test the knowledge acquired by the users.

After the tutorials two exercises are proposed, which are followed by a final challenge.

Finally, we conclude by discussing future improvements to the script.

2 Theoretical (scientific/technological background)

Before diving into specific AI models it's important to have a general understanding of how AI generally works.

Most AI solutions in natural language processing are based on broader **Large language models (LLMs)** these are algorithms that have been trained on vast amounts of data with the purpose of understanding and generating human like text. LLMs make use of the **transformer** type of architecture, a deep learning technique that, in short, represents text via numerical representations known as tokens and gives them different weights so as to be able to contextualise words and find similarities [2].

One example of a transformer based model is the **Generative pre-trained transformer (GPT)**, an AI model that has been pre-trained on large sets of data via the use of the transformer architecture for general purpose tasks. Furthermore, these models can be fine tuned to achieve greater performance in more specific tasks, such is the case of image generation models [3].

Lastly, in order to make use of these models it's essential to understand the concept of prompts. That is the name given to the textual inputs given to the model. These inputs are then broken down into the aforementioned tokens via a process we call tokenization, which facilitates the use of the models language structure.

2.1 State-of-the-art

AI, Artificial Intelligence, refers to a simulation of human intelligence in machines programmed to mimic human cognitive processes and actions. The concept of AI is not new, it has been around since the mid-20th century, but in the last few years there have been significant advances for AI. With these advances, due to its potential, AI has become increasingly important across various industries such as healthcare, finance, manufacturing, education, amongst others. Nowadays, one of the most famous companies developing AI products is OpenAI, which we are going to base our work on. However, just like any other business, OpenAI has its competitors. Some of them currently only dispose of a chat bot, and even that, at the time, are not available in Portugal. Listed below are some companies in the AI business.

AI	**OpenAI[4]**	**X[5]**	**Anthropic[6]**	**Deepmind[7]**	**Cohere[8]**
ChatBot	ChatGPT	Grok	Claude	Gemini	Coral
API	Yes	No	Yes	Yes	Yes
Inputs	Text, Audio, Image, Video		Text	Text, Image	Text
Outputs	Text, Audio, Image, Video		Text	Text	Text
ChatBot and API Availability in Portugal	Both	None	API Only	ChatBot Only	Both

Table 2: AI Companies.

2.2 OpenAI Features

OpenAI's API offers a vast array of cutting-edge AI models based on deep learning and natural language processing techniques. These models have been trained on vast datasets and fine tuned to fulfil a variety of tasks such as text and image generation, audio and text conversions, amongst other things.

2.2.1 GPT

The GPT (Generative Pre-trained Transformer) series is OpenAI's main set of large language models. These models are trained to understand and generate natural language text based on contextual inputs so as to better communicate with humans. The most widespread version, GPT-3.5 currently serves as the model that powers the free version of ChatGPT. Its understanding of human language allows for coherent conversations which makes it a suitable chat bot. GPT-4 improves upon its predecessor with a smarter and more knowledgeable model that provides greater accuracy across various tasks. In particular, GPT-4 introduces Vision as a new feature, that enables it to process image inputs, making it useful in a wider range of applications.

Function calling

The GPT based models are capable of calling previously specified functions in response to user actions or prompts by calling external APIs to retrieve data or to automate procedures like sending an email or extracting and sorting data from a document.

Assistants

The AI Assistants functionality leverages the use of the GPT models alongside function calling and other tools like file retrieval and code interpreter to allow users to create custom assistants that fulfil more specific tasks based on the provided instructions.

2.2.2 Other models

DALL-E

The DALL-E model is capable of generating images from natural-language text descriptions, as well as modifying existing ones by feeding the model instructions from a text prompt.

TTS

The Text-To-Speech model is capable of converting text into a natural sounding speech. Note that it currently only supports the English language.

Whisper

Whisper does the opposite of the TTS model: it takes an audio input and then transcribes it into text. Unlike text to speech, Whisper is capable of understanding multiple languages, as such it can be used to identify the input language and translate the contents of the speech into English.

Embeddings

Text embeddings are vectorial representations of strings of text, such as words or phrases. By comparing two or more vectors we can infer their similarity. This mechanism is highly useful in applications such as search engines or product recommendations due to it's ability of evaluating similarity between text strings.

Moderation

OpenAI's Moderation model is designed to verify if a certain piece of text includes any content that could be classified as hateful, violent, sexual, harmful or otherwise inappropriate. Whilst OpenAI's own use of the model aims to ensure that content complies with their usage policies [4], this model is suitable for any application that aims to ensure a safe digital environment.

3 Setup/Installation

3.1 Creating an OpenAI Account

In this section,we will guide you through the process of setting up an OpenAI account. Whether you're a developer, researcher, or simply curious about AI, having your own account opens the door to the vast possibilities of artificial intelligence.

Firstly, navigate to the OpenAI website [4] to create or log into your account.

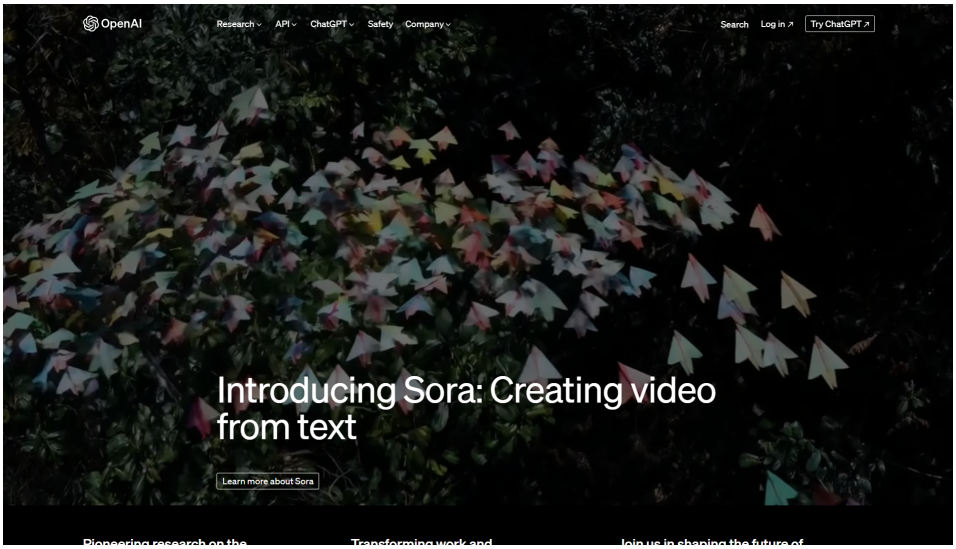

Figure 1: OpenAI Website Access

Upon logging in with your email, you will be presented with two options: ChatGPT and API. Select the API option to access the documentation.

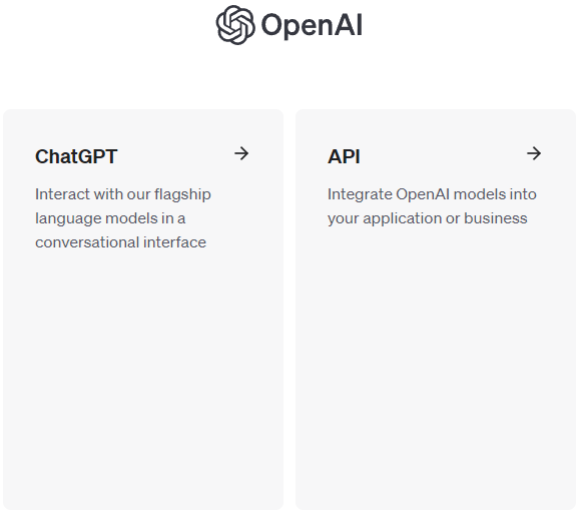

Figure 2: Select the API Option

Congratulations! You have now successfully created an operational OpenAI account.

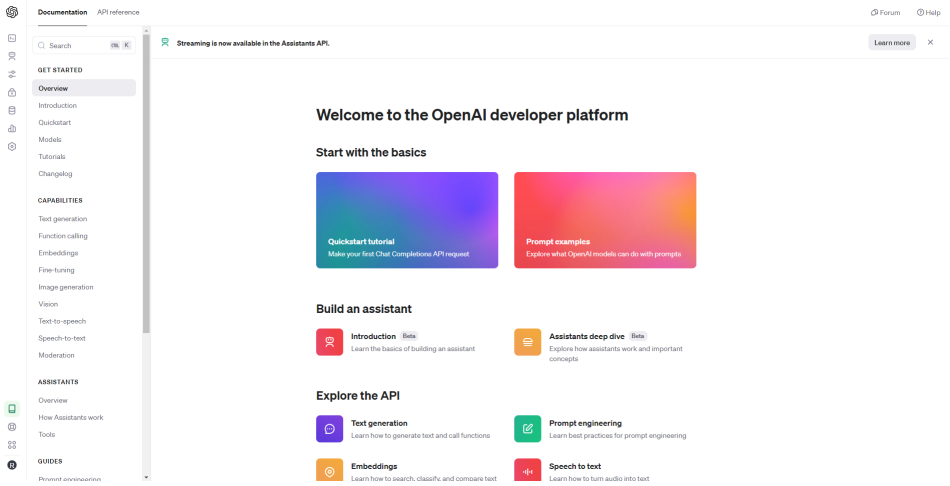

Figure 3: OpenAI Account Creation Confirmation

3.2 Setting up Your Development Environment

Setting up a proper development environment is crucial for working efficiently with AI applications. Ensure that you have the necessary tools and libraries installed on your system. For most AI development tasks, Python is the recommended programming language due to its extensive ecosystem of AI and machine learning libraries. So, that is exactly what we are going to help you with in this sub-section.

3.2.1 Windows

The first step is accessing python's official website and downloading it. In case you are not sure if you have already installed Python in the past, just type "cmd" in your search-bar and then type "python". If you are having trouble installing, maybe try checking Python's beginners guide [9].

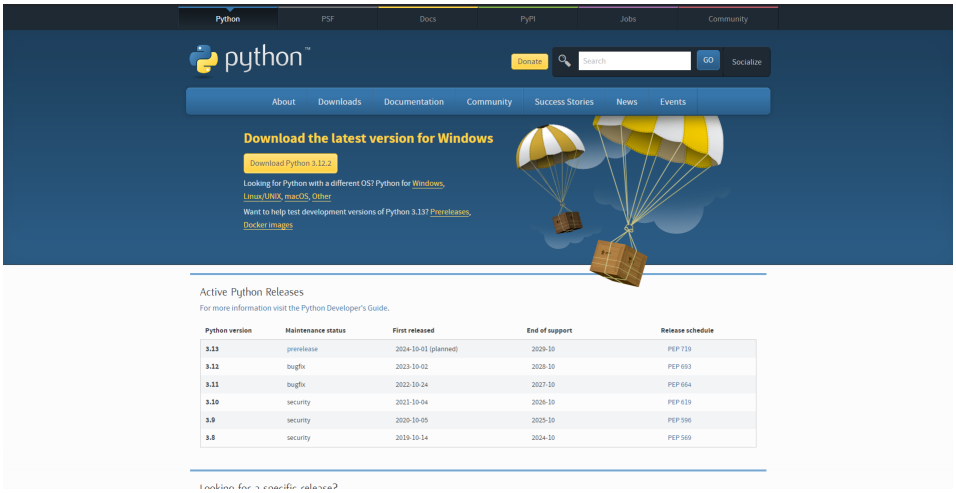

Figure 4: Download Python

Once installed, you are going to create a virtual environment, as it is good practise to avoid conflicts with other installed libraries.

Type this command in your command prompt:

```
'python3 -m venv openai-env'
```

Now, after creating the virtual environment, you need to activate it:

```
'openai-env\Scripts\activate'
```

After this step you should be able to see "openai-env" to the left of the cursor input section.

Once you have Python installed and (optionally) set up a virtual environment, the OpenAI Python library can be installed. From the command prompt, run:

```
'pip install --upgrade openai'
```

Once this completes, running 'pip list' will show you the Python libraries you have installed in your current environment, which should confirm that the OpenAI Python library was successfully installed.

Now we are going to setup your API key. If you don't have an API key yet then you'll have to follow the instruction in API section

Open your command prompt and then insert the following command:

```
'setx OPENAI_API_KEY "your-api-key-here'
```

In order to make this key setup permanent, you ought to access Environment Variables, for that you just need to search for it in your windows search bar. Click on "New" and then set

`"OPENAI_API_KEY"`

as the variable name and your API key as the value.

3.2.2 Linux

Firstly, open your terminal and introduce the following command in order to download python:

In case you are working on Debian or Ubuntu:

```
'apt install python3 python3-dev'
```

In case you are working on Red Hat, CentOS, or Fedora:

```
'dnf install python3 python3-devel'
```

If you are having trouble installing, maybe try checking Python's beginners guide [9].

Once installed, you are going to create a virtual environment, as it is good practise to avoid conflicts with other installed libraries.

Type this command in your terminal:

```
'python3 -m venv openai-env'
```

If you can't use none of these commands above because of this error: "The virtual environment was not created successfully because ensurepip is not available" then, try using the following command (after this command you have to re-insert one of the commands above):

```
'sudo apt install python3.10-venv'
```

Now, after creating the virtual environment, you need to activate it:

```
'source openai-env/bin/activate'
```

After this step you should be able to see "openai-env" to the left of the cursor input section.

Once you have Python installed and (optionally) set up a virtual environment, the OpenAI Python library can be installed. From the terminal, run:

```
'pip install --upgrade openai'
```

Once this completes, running 'pip list' will show you the Python libraries you have installed in your current environment, which should confirm that the OpenAI Python library was successfully installed.

Now we are going to setup your API key. If you don't have an API key yet then you'll have to follow the instruction in API section

Go to OpenAI website and access the "API keys" section, there you are going to retrieve your API key or create one in case you do not already have.

Then, open your terminal and type the following command:

```
'export OPENAI_API_KEY='your-api-key-here''
```

To save just press Ctrl+O.

If you want to check whether it is setup correctly, type

```
'echo $OPENAI_API_KEY'
```

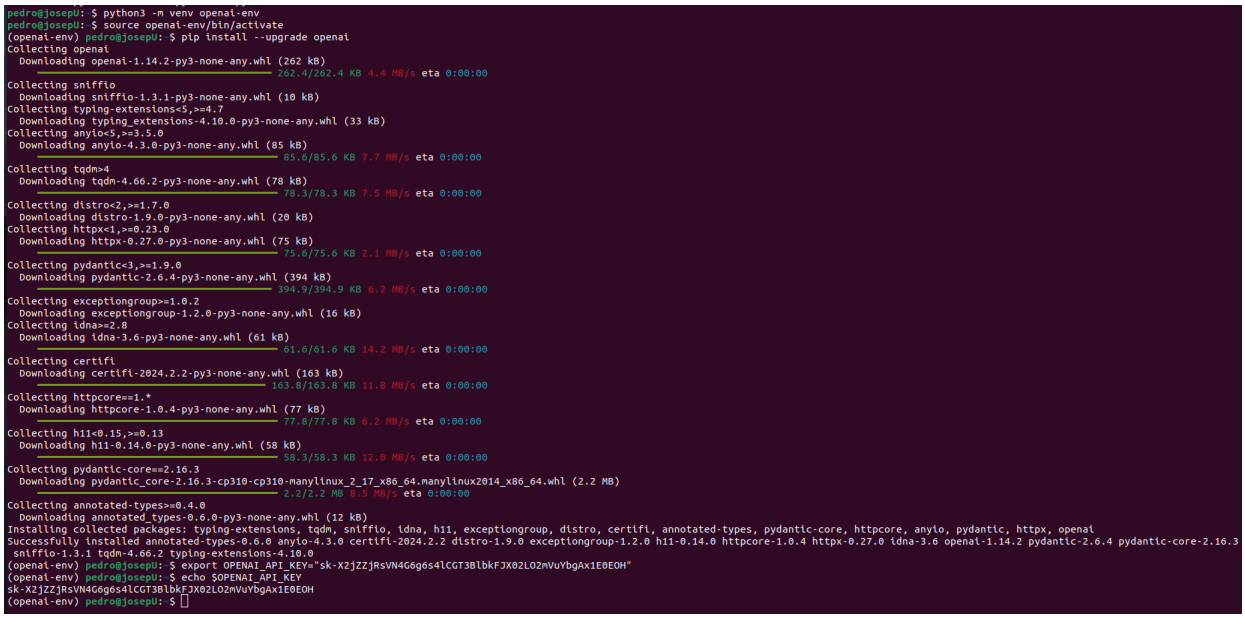

Figure 5: Setup Python

3.2.3 MacOS

Firstly install Brew, if not already installed:

```
'/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"'
```

Now that you already have Brew, open your terminal and introduce the following command in order to download python:

```
'brew install python'
```

If you are having trouble installing, maybe try checking Python's beginners guide [9].

Once installed, you are going to create a virtual environment, as it is good practice to avoid conflicts with other installed libraries.

Type this command in your terminal:

```
'pip install virtualenv'
```

Create venv:

```
'virtualenv openai-env'
```

Now, after creating the virtual environment, you need to activate it:

```
'source openai-env/bin/activate'
```

After this step you should be able to see "openai-env" to the left of the cursor input section.

Once you have Python installed and (optionally) set up a virtual environment, the OpenAI Python library can be installed. From the terminal, run:

```
'pip install --upgrade openai'
```

Once this completes, running 'pip list' will show you the Python libraries you have installed in your current environment, which should confirm that the OpenAI Python library was successfully installed.

Now we are going to setup your API key. If you don't have an API key yet then you'll have to follow the instruction in API section

Go to OpenAI website and access the "API keys" section, there you are going to retrieve your API key or create one in case you do not already have.

Then, open your terminal and type the following command:

```
'export OPENAI_API_KEY='your-api-key-here''
```

To save just press Ctrl+O.

If you want to check it did get setup correctly, type

```
'echo $OPENAI_API_KEY'
```

3.3 Installing Jupyter

The tutorials in this script have been designed to be used with Jupyter [10], an interactive python environment that will ease your first steps into AI.

You can install JupyterLab, the web-based interface, by running the following command:

```
'pip install jupyterlab'
```

You can launch the interface with:

```
'jupyter lab'
```

If you are using an IDE such as *VS Code* or *Jetbrain's PyStorm* simply install the available jupyter notebook extension/plugin.

3.4 Installing required packages

Before starting the tutorials, install all of the necessary python packages using pip:

```
'pip install -r <path_to_requirements.txt>'
```

4 Tutorial/Functionality

In order to work with OpenAI's APIs you must first import it's library.

Listing 1: Import OpenAI

```
from openai import OpenAI
```

To use this library you need to instantiate a client. To do this you are going to need a key. If no key is indicated in the constructor, OpenAI will default to the environment variable `OPEN_AI_KEY` value.

Listing 2: Create Client

```
# API_KEY -> is retrieve from the environment variable
client = OpenAI()

# API_KEY -> is indicated by the user at the moment of creation
client = OpenAI(api_key="{YOUR_OPEN_AI_KEY}")
```

For good practice, we advise that you use a *.env* file to store your private information, like your key(see *.env.example*).

Listing 3: Create Client wiht .env file

```
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
```

4.1 Chat Completions

Chat Completions is probably one of the simplest OpenAI´s capabilities. It simply takes a list of messages as input and returns an answer.

1 - First import and create an OpenAI client

Listing 4: Chat Completion step 1

```
from openai import OpenAI
client = OpenAI()
```

2 - Structure our response, passing the gpt model and a few messages to give a brief context to our chatbot

Listing 5: Chat Completion step 2

```
response = client.chat.completions.create(
  model="gpt-3.5-turbo",
  messages=[
    {"role": "system", "content": "You are a helpful football assistant."},
    {"role": "user", "content": "Who won the Euro back in 2016?"},
    {"role": "assistant", "content": "Portugal won the World Cup in 2016."},
    {"role": "user", "content": "Where was it played, what was the score of the final and
        who scored in that game?"}
  ]
)

```

```
print(response.choices[0].message.content)
```

4.2 Assistants

The Assistants API makes it possible to create AI assistants within applications. An Assistant has instructions and can use models, tools and files to answer the user's questions. The Assistants API currently supports three types of tools: Code Interpreter, File Search and Function Calling.

4.2.1 Assistant

In this example, an Assistant is created as a personal maths tutor.

Initially, an Assistant is created, which represents an entity that can be configured to respond to a user's messages using various parameters such as a , **'name'**, **'instructions'** and **'model'**.

Listing 6: Assistant step 1

```
from openai import OpenAI
from runners.standard_run import std_run
from runners.streaming_run import streaming_run


client = OpenAI()

assistant = client.beta.assistants.create(
    name="Math Tutor",
    instructions="You are a personal math tutor. Write and run code to answer math
        questions.",
    model="gpt-3.5-turbo",
)
```

Next, a Thread is created that represents a conversation between a user and one or more Assistants. A Thread can be created when a user (or their AI application) starts a conversation with their Assistant.

Listing 7: Assistant step 2

```
thread = client.beta.threads.create()
```

A message is added to the Thread, the content of the messages that users or applications create is added as objects to the Message Thread.

Listing 8: Assistant step 3

```
message = client.beta.threads.messages.create(
    thread_id=thread.id,
    role="user",
    content="I need to solve the equation: 4a - 13 = 7‘. Can you help me?"
)
```

When the user has added all of the necessary information/messages to the thread you must run the thread to provide them with results.

You can run a thread using its id as well as the assistant's id, like this:

Listing 9: Assistant step 4

```
    run = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=assistant.id,
        #model = "gpt-3.5-turbo" # By default the model defined in the assistant will be
            used. You can, however, overwrite it here.
        instructions = "Please address the user as SISTCA student." # This parameter is
            optional
    )
```

During its life cycle, a run can havemultiple states [4]:

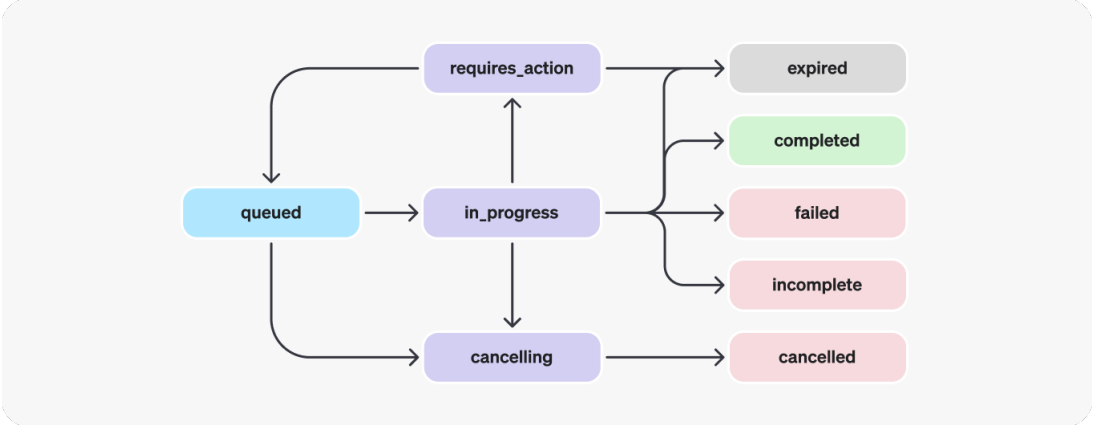

Figure 6: Run status flow diagram .

Therefore, we must check for its completion.

Listing 10: Assistant step 5

```
    if run.status == 'completed':
        messages = client.beta.threads.messages.list(
        thread_id=thread.id
        )
        print(messages)
    else:
        print(run.status)
```

The result is then returned by printing the messages returned by the run.

Listing 11: Assistant step 6

```
    messages = client.beta.threads.messages.list(thread_id = thread.id)
    print(messages.data[0].content[0].text.value)
```

In order to yield any results, an assistant's thread must be run. This is to say that whenever you're working with the assistant functionality you must have a runner function.

Due to the fact that we will be exploring other types of assistants, and as a way of not repeating this code multiple times, we will export as the following function:

Listing 12: Assistant step 7

```
    def std_run (thread_id, assistant_id, client):
        run = client.beta.threads.runs.create_and_poll(
            thread_id = thread_id,
            assistant_id = assistant_id,
            instructions = "Please address the user as SISTCA student."
```

```
    )

    if run.status == 'completed':
        messages = client.beta.threads.messages.list(thread_id = thread_id)
        print(messages.data[0].content[0].text.value)
    else:
        print(run.status)
```

If you have used ChatGPT before you may have noticed that the output is not returned all at once, like it is here, but rather gradually, in real time.

We can achieve this same effect by using a different type of runner, the streaming run.

In this run, the streaming_run function is defined, which is responsible for starting the streaming with the assistant. Within this, the **'EventHandler'** class is defined, which is inherited from **AssistantEventHandler**. This class contains the methods responsible for dealing with the different events that occur during streaming:

- **'on_text_created'**: Called when a new text is created by the assistant. This prints **'assistant'** to indicate that the assistant is responding.
- **'on_text_delta'**: Called for each text update. This prints the part of the text that was generated (delta.value).
- **'on_tool_call_created':** Called when the wizard calls a tool. This prints the type of tool call (tool_call.type).
- **'on_tool_call_delta':** Called for tool call updates, specifically for the code_interpreter. This prints the code_interpreter input and any outputs, especially logs.

Finally, the API client's create_and_stream method is used to start streaming, passing the **'EventHandler'** as the event handler.

- **'thread_id'** and **'assistant_id'** are the required identifiers.
- **'instructions'** defines specific instructions for the assistant, in this case asking it to address the user as a SISTCA student.
- **'event_handler'** is the instance of the **'EventHandler'** class.

The **'with'** block ensures that streaming continues until it is finished (stream.until_done()).

Listing 13: Assistant step 8

```
# With streaming

from typing_extensions import override
from openai import AssistantEventHandler


def streaming_run (thread_id, assistant_id, client):

    # First, we create a EventHandler class to define
    # how we want to handle the events in the response stream.

    class EventHandler(AssistantEventHandler):
        @override
        def on_text_created(self, text) -> None:
            print(f"\nassistant > ", end="", flush=True)

```

```
            @override
            def on_text_delta(self, delta, snapshot):
                print(delta.value, end="", flush=True)

            def on_tool_call_created(self, tool_call):
                print(f"\nassistant > {tool_call.type}\n", flush=True)

            def on_tool_call_delta(self, delta, snapshot):
                if delta.type == 'code_interpreter':
                    if delta.code_interpreter.input:
                        print(delta.code_interpreter.input, end="", flush=True)
                if delta.code_interpreter.outputs:
                    print(f"\n\noutput >", flush=True)
                    for output in delta.code_interpreter.outputs:
                        if output.type == "logs":
                            print(f"\n{output.logs}", flush=True)


        # Then, we use the `create_and_stream` SDK helper
        # with the `EventHandler` class to create the Run
        # and stream the response.

        with client.beta.threads.runs.stream(
            thread_id=thread_id,
            assistant_id=assistant_id,
            instructions="Please address the user as SISTCA student.",
            event_handler=EventHandler(),
        ) as stream:
            stream.until_done()

    print(streaming_run(thread.id, assistant.id, client))
```

4.2.2 Function Calling

Function Calling is one of Assistants API's tools, which in simple terms, invokes predefined functions in order to respond to a certain prompt. As you will verify in just a moment, in this tutorial we asked our weather assistant to give us the maximum and minimum temperatures and rain probability in a certain location.

By using functions we allow the assistant to call additional methods as it sees fit. Assistants will analyse the user's message and decide if there is a function that can be used to answer the request and, if so, a request to call that function will be made to the runner.

For instance, in this case we want to make it so that when asking for weather data for a Portuguese city or region the data should come from IPMA.

By using IPMA'S API we can check the weather data for a specific location using it's ID.

We start by defining the functions the assistant may call, alongside their parameters.

Listing 14: Function Calling step 1

```
assistant = client.beta.assistants.create(
instructions="You are a weather bot. Use the provided functions to answer questions.",
model="gpt-3.5-turbo",
tools=[
    {
        "type": "function",
        "function": {
            "name": "get_weather_data",
            "description": "Get the current weather forecast for the specified
```

```
                location",
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "City or region for which to get the weather
                            forecast",
                    }
                },
                "required": ["location"]
            }
        }
    }
]
)
```

Then, we must define actual functions to fetch the weather data.

To get weather data for a city or region we must first know its global ID. We can do this by accessing the list of identifiers and extracting the **'globalIdLocal'** for the desired city (**'location_name'**).

By visiting the list of identifiers endpoint, this *json* is returned:

Listing 15: Function Calling - IPMA´s json response - step 2

```
{
"owner": "IPMA",
"country": "PT",
"data": [
    {
        "idRegiao": 1,
        "idAreaAviso": "AVR",
        "idConcelho": 5,
        "globalIdLocal": 1010500,
        "latitude": "40.6413",
        "idDistrito": 1,
        "local": "Aveiro",
        "longitude": "-8.6535"
    },
    {
        "idRegiao": 1,
        "idAreaAviso": "BJA",
        "idConcelho": 5,
        "globalIdLocal": 1020500,
        "latitude": "38.0200",
        "idDistrito": 2,
        "local": "Beja",
        "longitude": "-7.8700"
    },
[...]
```

Knowing the *json* structure, define a function to get the **globalIdLocal** for the matching city:

Listing 16: Function Calling step 3

```
import requests

def get_location_id(location_name):
    url = "https://api.ipma.pt/open-data/distrits-islands.json"
    try:
        response = requests.get(url)
```

```
            if response.status_code == 200:
                data = response.json()
                for location in data["data"]:
                    # Check if the location name matches the input by converting them to
                        lower case, so that i.e. "Porto" == "porto"
                    if location["local"].lower() == location_name.lower():
                        return str(location["globalIdLocal"])
                return None # Location not found
            else:
                print("Failed to fetch data:", response.status_code)
                return None
        except Exception as e:
            print("An error occurred:", e)
            return None
```

Test the function:

Listing 17: Function Calling step 4

```
location_id = get_location_id("Porto")
print(location_id)
```

The next function, **'get_weather_data()'**, uses the **'location_id'** returned by **'get_location_id()'** to access the actual weather data of the required location, which we then return.

Listing 18: Function Calling step 5

```
def get_weather_data(location_id):
url =
    f"https://api.ipma.pt/open-data/forecast/meteorology/cities/daily/{location_id}.json"
try:
    response = requests.get(url)
    if response.status_code == 200:
        data = response.json()
        forecast = data["data"][0]
        return forecast
    else:
        print("Failed to fetch data:", response.status_code)
        return None
except Exception as e:
    print("An error occurred:", e)
    return None
```

Test the function:

Listing 19: Function Calling step 6

```
weather_data = get_weather_data("1131200")
print(weather_data)
```

Having created the necessary functions we can now try running our assistant by creating a thread and adding a message requesting a weather forecast.

Listing 20: Function Calling step 7

```
thread = client.beta.threads.create()

#Define the location name
location_name = "Porto"

message = client.beta.threads.messages.create(
```

```
        thread_id=thread.id,
        role="user",
        content=f"How is the weather today in {location_name}?",
    )

    print(message.content[0].text.value + "\n")
```

Now, initiate a run to yield a response.

Listing 21: Function Calling step 8

```
    run = client.beta.threads.runs.create_and_poll(
        thread_id = thread.id,
        assistant_id = assistant.id
    )

    if run.status == 'completed':
        messages = client.beta.threads.messages.list(
        thread_id=thread.id
        )
        print(messages)
    else:
        print(run.status)
```

This time the run status returns a different value - **'required_action'**. This likely means that the assistant has called a function and is waiting for a response.

Let's check that:

Listing 22: Function Calling step 9

```
    # Check if a run is trying to call a tool and if so, act accordingly
    for tool in run.required_action.submit_tool_outputs.tool_calls:
        if tool.function.name == "get_weather_data":

            # let's start by checking the content of the tool call
            print(tool.function)
```

The assistant wants to call the **get_weather_data** tool using the value "Porto" as the location argument.

While the assistant is smart enough to realise that it should use the **'get_weather_data'** tool to answer the user's question it does not know how to proceed, because we never linked that tool with any of our two previously defined functions.

The set of instructions to be performed when an assistant calls a function are defined in the runner. Let's do that:

Listing 23: Function Calling step 10

```
import json

# Check if a run is trying to call a tool and if so, act accordingly
for tool in run.required_action.submit_tool_outputs.tool_calls:
    if tool.function.name == "get_weather_data":

        # First, call the get_location_id function to get the location ID for the location
            argument in the tool call
        location_id = get_location_id(json.loads(tool.function.arguments)["location"])

        # Now, knowing the location ID, we can call the get_weather_data function to get
            the weather data for the location
```

```
        forecast = get_weather_data(location_id)
```

Forecast returns a *json* similar to this:

Listing 24: Function Calling step 11

```
{'precipitaProb': '99.0',
 'tMin': '11.5',
 'tMax': '16.8',
 'predWindDir': 'W',
 'idWeatherType': 6,
 'classWindSpeed': 2,
 'longitude': '-8.6294',
 'forecastDate': '2024-05-18',
 'classPrecInt': 2,
 'latitude': '41.1580'}
```

To provide the user with relevant weather data we shall return the minimum and maximum temperature as well as the rain probability.

Listing 25: Function Calling step 12

```
print(f"Rain probability: {forecast['precipitaProb']}, Max temperature:
    {forecast['tMin']}, Min temperature: {forecast['tMax']}")
```

To output this data we first define an array in which to store the tool outputs. In this case we only have one single tool, but this would be especially useful if we were dealing with multiple tools.

Listing 26: Function Calling step 13

```
# Create an array to store the tool outputs
tool_outputs = []
```

Finally, we submit the output of the **'get_weather_data'** function to the run by appending an "output" key to the output array.

Listing 27: Function Calling step 14

```
tool_outputs.append({"tool_call_id": tool.id,
                     "output": f"Rain probability: {forecast['precipitaProb']}, Max
                         temperature: {forecast['tMin']}, Min temperature:
                         {forecast['tMax']}"
                     })
```

Having defined the tool outputs we must now add them to the runner so that our assistant can use them to answer our initial question.

Listing 28: Function Calling step 15

```
if tool_outputs:
    try:
        run = client.beta.threads.runs.submit_tool_outputs_and_poll(
            thread_id=thread.id,
            run_id=run.id,
            tool_outputs=tool_outputs
        )
        print("Tool outputs submitted successfully.")
    except Exception as e:
        print("Failed to submit tool outputs:", e)
else:
```

```
    print("No tool outputs to submit.")
```

We then check for the run completion and display the output message:

Listing 29: Function Calling step 16

```
if run.status == 'completed':
    messages = client.beta.threads.messages.list(thread_id=thread.id)
    print(messages.data[0].content[0].text.value)
else:
    print(run.status)
```

The updated runner with the tool call handling code will look like this:

Listing 30: Function Calling step 17

```
def std_run (thread_id, assistant_id, client):
    run = client.beta.threads.runs.create_and_poll(
        thread_id = thread_id,
        assistant_id = assistant_id,
        instructions = "Please address the user as SISTCA student."
    )

    if run.status == 'completed':

        messages = client.beta.threads.messages.list(
            thread_id = thread_id
        )

        # This code acts upon the presence of image files when using code interpreter
        if messages.data[0].content[0].type == 'image_file':
            return(messages.data[0].content[0].image_file.file_id)
        else:
            print(run.status)

        return(messages.data[0].content[0].text.value)

    # The following code is used to handle tool/function calls

    # Preparing tool outputs
    tool_outputs = []
    for tool in run.required_action.submit_tool_outputs.tool_calls:
        if tool.function.name == "get_weather_data":
            location_id = get_location_id(json.loads(tool.function.arguments)["location"])
            forecast = get_weather_data(location_id)
            tool_outputs.append({"tool_call_id": tool.id,
                                 "output": f"Rain probability: {forecast['precipitaProb']},
                                     Max temperature: {forecast['tMin']}, Min temperature:
                                     {forecast['tMax']}"
                                 })

    # Submitting tool outputs and polling for completion status
    if tool_outputs:
        try:
            run = client.beta.threads.runs.submit_tool_outputs_and_poll(
                thread_id=thread_id,
                run_id=run.id,
                tool_outputs=tool_outputs
            )
            print("Tool outputs submitted successfully.")
        except Exception as e:
            print("Failed to submit tool outputs:", e)
```

```
    else:
        print("No tool outputs to submit.")

    # Retrieving messages after tool output submission
    if run.status == 'completed':
        messages = client.beta.threads.messages.list(thread_id=thread_id)
        print(messages.data[0].content[0].text.value)
    else:
        print(run.status)
```

Using the same logic we can also stream the response in the runner. However, since function calling uses different events that can conflict we the previously defined ones, instead of updating the **'streaming_run'** function we will create a new one.

Listing 31: Function Calling step 18

```
def streaming_run_fc (thread_id, assistant_id, client):

    class EventHandler(AssistantEventHandler):
        @override
        def on_event(self, event):

            # Retrieve events that are denoted with 'requires_action'
            # since these will have our tool_calls

            if event.event == 'thread.run.requires_action':
                run_id = event.data.id # Retrieve the run ID from the event data
                self.handle_requires_action(event.data, run_id)

        def handle_requires_action(self, data, run_id):
            tool_outputs = []

            for tool in data.required_action.submit_tool_outputs.tool_calls:
                if tool.function.name == "get_weather_data":
                    location_id =
                        get_location_id(json.loads(tool.function.arguments)["location"])
                    forecast = get_weather_data(location_id)
                    tool_outputs.append({"tool_call_id": tool.id,
                                         "output": f"Rain probability:
                                             {forecast['precipitaProb']}, Max temperature:
                                             {forecast['tMin']}, Min temperature:
                                             {forecast['tMax']}"
                                         })

            # Submit all tool_outputs at the same time
            self.submit_tool_outputs(tool_outputs, run_id)

        def submit_tool_outputs(self, tool_outputs, run_id):
            # Use the submit_tool_outputs_stream helper
            with client.beta.threads.runs.submit_tool_outputs_stream(
                thread_id=self.current_run.thread_id,
                run_id=self.current_run.id,
                tool_outputs=tool_outputs,
                event_handler=EventHandler(),
            ) as stream:
                for text in stream.text_deltas:
                    print(text, end="", flush=True)
                print()


    with client.beta.threads.runs.stream(
        thread_id=thread_id,
        assistant_id=assistant_id,
```

```
        event_handler=EventHandler()
    ) as stream:
        stream.until_done()
```

4.2.3 Code Interpreter

The Assistant's Code Interpreter makes it possible for assistants to write and run python code to answer user questions and analyse files.

The Code Interpreter works similarly to the tools in the previous tutorial as it too is a tool.

In this tutorial we will attempt to portray Code Interpreter's usefulness for data analysis by providing it with a file containing the data of 100 customers of a fictitious organisation.

Listing 32: Code Interpreter step 1

```
file = client.files.create(
    file=open("resources/customers-100.csv", "rb"),
    purpose='assistants'
)
```

To use code interpreter it must be passed as a tool parameter of the assistant object. We will also be adding our previously defined file as a resource.

Listing 33: Code Interpreter step 2

```
assistant = client.beta.assistants.create(
    instructions="Your purpose is to analyze the provided documents and to provide
        answers based on them.",
    model="gpt-3.5-turbo",
    tools=[{"type": "code_interpreter"}],
    tool_resources={
        "code_interpreter": {
        "file_ids": [file.id]
        }
    }
)
```

Next, we create a thread and will try asking the assistant for the customer ID of Sheryl Baxter, the customer in the first row of the provided file. If it works, the value **'DD37Cf93aecA6Dc'** should be returned.

Listing 34: Code Interpreter step 3

```
thread = client.beta.threads.create()

message = client.beta.threads.messages.create(
    thread_id=thread.id,
    role="user",
    content="What is the customer Id for Sheryl Baxter?"
)

print(message.content[0].text.value + "\n")
```

This time will simply call our previously defined runner instead of writing the runner code all over again.

Listing 35: Code Interpreter step 4

```
print(std_run(thread.id, assistant.id, client))
```

Now, for something more complex, let's ask the assistant to create a graph using the provided file.

Listing 36: Code Interpreter step 5

```
message = client.beta.threads.messages.create(
    thread_id=thread.id,
    role="user",
    content="Create a graph of the number of customers per country."
)

print(message.content[0].text.value + "\n")
```

This action should return a file with the attached graph. When an assistant return a file it is marked as an **'image_file'**. To access it we must get its id.

This can be achieved by checking for the presence of an **'image_file'** in the return messages of a completed run:

Listing 37: Code Interpreter step 6

```
if run.status == 'completed':

    messages = client.beta.threads.messages.list(
        thread_id = thread.id
    )

    # This code acts upon the presence of image files when using code interpreter
    if messages.data[0].content[0].type == 'image_file':
        print(messages.data[0].content[0].image_file.file_id)
    else:
        print(run.status)
```

The updated runner with this code looks like this:

Listing 38: Code Interpreter step 7

```
def std_run (thread_id, assistant_id, client):
    run = client.beta.threads.runs.create_and_poll(
        thread_id = thread_id,
        assistant_id = assistant_id,
        instructions = "Please address the user as SISTCA student."
    )

    if run.status == 'completed':

        messages = client.beta.threads.messages.list(
            thread_id = thread_id
        )

        # This code acts upon the presence of image files when using code interpreter
        if messages.data[0].content[0].type == 'image_file':
            return(messages.data[0].content[0].image_file.file_id)
        else:
            print(run.status)

        return(messages.data[0].content[0].text.value)

    # The following code is used to handle tool/function calls

```

```
    # Preparing tool outputs
    tool_outputs = []
    for tool in run.required_action.submit_tool_outputs.tool_calls:
        if tool.function.name == "get_weather_data":
            location_id = get_location_id(json.loads(tool.function.arguments)["location"])
            forecast = get_weather_data(location_id)
            tool_outputs.append({"tool_call_id": tool.id,
                                 "output": f"Rain probability: {forecast['precipitaProb']},
                                     Max temperature: {forecast['tMin']}, Min temperature:
                                     {forecast['tMax']}"
                                 })

    # Submitting tool outputs and polling for completion status
    if tool_outputs:
        try:
            run = client.beta.threads.runs.submit_tool_outputs_and_poll(
                thread_id=thread_id,
                run_id=run.id,
                tool_outputs=tool_outputs
            )
            print("Tool outputs submitted successfully.")
        except Exception as e:
            print("Failed to submit tool outputs:", e)
    else:
        print("No tool outputs to submit.")

    # Retrieving messages after tool output submission
    if run.status == 'completed':
        messages = client.beta.threads.messages.list(thread_id=thread_id)
        print(messages.data[0].content[0].text.value)
    else:
        print(run.status)
```

Store the file ID:

Listing 39: Code Interpreter step 8

```
file_id = std_run(thread.id, assistant.id, client)
```

To retrieve the file we must call the **'files_endpoint'** using the **'file_id'**.

Listing 40: Code Interpreter step 9

```
image_data = client.files.content(file_id=file_id)
image_data_bytes = image_data.read()
```

Save and display the result:

Listing 41: Code Interpreter step 10

```
import IPython

image_name = "graph.png"

image_path = os.path.join("resources", image_name)

with open(image_path, "wb") as image_file:
    image_file.write(image_data_bytes)

IPython.display.Image(image_name)
```

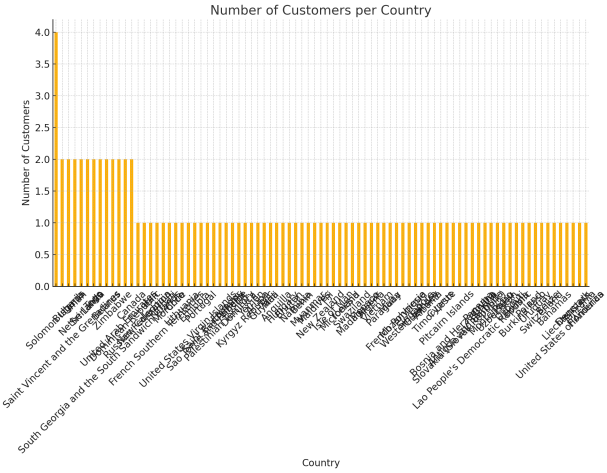

Figure 7: Resulting graph

4.2.4 File Search

File search is a tool that complements the Assistants API. It allows users to send their documents and question the assistant about them. The documents sent by the user are transformed in embeddings and stored in vector stores. File search uses both keyword and vector search to retrieve relevant information.

This feature implements good retrieval practices to extract the necessary data to answer the user query:

- Rewrites user queries to optimise them for search.
- Breaks down complex user queries into multiple searches that can run in parallel.
- Runs both keyword and semantic searches across both assistant and thread vector stores.
- Reranks search results to pick the most relevant ones before generating the final response.

File search supports a variety of different file formats. For this tutorial, a simple txt containing some wikipedia articles was the source of the information.

- .c
- .cs
- .cpp
- .doc
- .docx
- .html
- .java
- .json
- .md
- .pdf
- .php

- .pptx
- .py
- .rb
- .tex
- .txt
- .css
- .js
- .sh
- .ts

Vector Store

A vector store is a digital storage system that holds vectors. Adding a file to a vector store automatically parses, chunks, embeds and stores the file in a vector database that is capable of both keyword and semantic search. Adding a file or files to a vector store is an *async* operation. The maximum file size you can store 512 MB and each file can not contain more that 5,000,000 tokens, this is computed automatically when a file is attached.

In this tutorial we will provide the assistant with a text file containing wikipedia articles about musical artists. We will then have the assistant search this file to answer questions.

Listing 42: File Search step 1

```
assistant = client.beta.assistants.create(
  name="Music Information Assistant",
  instructions="You are an expert music analyst. Use your knowledge base to answer
      questions regarding musical artists and their work.",
  model="gpt-3.5-turbo",
  tools=[{"type": "file_search"}],
)
```

Create a vector store to store your information

Listing 43: File Search step 2

```
vector_store = client.beta.vector_stores.create(name="Music Information")

# Add the desired files. It is possible to add multiple files
file_paths = ["resources/file_search_information.txt"]
file_streams = [open(path, "rb") for path in file_paths]


# upload the files and add them to the vector store
file_batch = client.beta.vector_stores.file_batches.upload_and_poll(
  vector_store_id=vector_store.id, files=file_streams
)
```

Then, update the assistant with the newly created vector store.

Listing 44: File Search step 3

```
assistant = client.beta.assistants.update(
  assistant_id=assistant.id,
  tool_resources={"file_search": {"vector_store_ids": [vector_store.id]}},
)
```

```
5
6  # upload the user provided file to OpenAI
7  message_file = client.files.create(
8    file=open("resources/file_search_information.txt", "rb"), purpose="assistants"
9  )
```

Create a thread and ask the assistant a question whilst attaching the file to the input.

Listing 45: File Search step 4

```
1  thread = client.beta.threads.create(
2    messages=[
3      {
4        "role": "user",
5        "content": "How many artists do you know about, and what are their albums?",
6        "attachments": [
7          { "file_id": message_file.id, "tools": [{"type": "file_search"}] }
8        ],
9      }
10   ]
11 )
```

Lastly, run the thread and print the response:

Listing 46: File Search step 5

```
1      print(std_run(thread.id, assistant.id, client))
```

4.3 Embeddings

OpenAI's text embeddings transforms strings to numbers, that allows to measure the relatedness of text strings. Embeddings are commonly used for:

- Search (where results are ranked by relevance to a query string)
- Clustering (where text strings are grouped by similarity)
- Recommendations (where items with related text strings are recommended)
- Anomaly detection (where outliers with little relatedness are identified)
- Diversity measurement (where similarity distributions are analyzed)
- Classification (where text strings are classified by their most similar label)

An embedding is a vector (list) of floating point numbers. The distance between two vectors measures their relatedness. Small distances suggest high relatedness and large distances suggest low relatedness.

Like mentioned before, Embeddings have a lot of uses but this tutorial will only focus on how to make the request and a show simple comparison between two strings.

Models

Right now there are three available models for Embeddings. The ones with "-3" on the name are third generation models.

- text-embedding-3-small

- text-embedding-3-large
- text-embedding-ada-002

1 - Import the necessary libraries and start the client

Listing 47: Embeddings tutorial step 1

```
from openai import OpenAI
client = OpenAI()
```

2 - Retrieve the embeddings

model - changes the model use to retieve information *input* - string message you want to retrieve the embeddings from

Listing 48: Embeddings tutorial step 2

```
response1 = client.embeddings.create(
    input="We are testing to see if this string has any similarities to another one.",
    model="text-embedding-3-small"
)

embeddings1 = response1.data[0].embedding

print(f"Embeddings Example - {embeddings1[0]}, {embeddings1[1]}")
```

Compare two strings using embeddings and cosine similarity

1 - Import numpy and cosine_similarity

Listing 49: Embeddings Compare Two Strings step 1

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
```

2 - Retrieve the embeddings for the second string

Listing 50: Embeddings Compare Two Strings step 2

```
response2 = client.embeddings.create(
    input="We're experimenting to determine if this string bears resemblance to another.",
    model="text-embedding-3-small"
)

embeddings2 = response2.data[0].embedding
```

3 - Convert the embeddings to numpy arrays

Listing 51: Embeddings Compare Two Strings step 3

```
embeddings1 = np.array(embeddings1).reshape(1, -1)
embeddings2 = np.array(embeddings2).reshape(1, -1)
```

4 - Calculate the similarity

The closer the score is to 1 the similar it is.

Listing 52: Embeddings Compare Two Strings step 4

```
similarity_score = cosine_similarity(embeddings1, embeddings2)

final_score = float(format(similarity_score[0][0], ".2f"))

print(f"Similarity: {final_score}")
```

4.4 Vision

Vision API from OpenAI, based on GPT-4 Turbo, is an enormous jump in AI capability. This functionality brought AI the capability to take pictures as input, comprehend them and answer questions about them. This development guarantees to revolutionize the way we interact and use AI.

Vision can accept images through links or even by passing its base64 encoded image. Unfortunately, this feature is only available to ChatGPT-4, so we will be addressing this tutorial at the end of our article as an extra.

1. Firstly, in order to start using this functionality, you need to import the necessary packages. In this case, we only need to import the **'openai'**.

```
        from openai import OpenAI
```

2. Now it´s time to create a **'client'** in order to request the **'response'** from **'openai'**.

```
        client = OpenAI()
```

3. Last but not least, the **'response'** is structered, passing the role, content and the image.

```
    response = client.chat.completions.create(
      model="gpt-4-turbo",
      messages=[
        {
          "role": "user",
          "content": [
            {"type": "text", "text": "What's in this image?"},
            {
              "type": "image_url",
              "image_url": {
                "url":
                    "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisconsin-madison-
            },
          },
        ],
      }
      ],
      max_tokens=300,
    )

    print(response.choices[0].message.content)
```

4.5 Image Generation (DALL-E)

The open AI image API provides three methods for interacting with images, namely creating images from scratch using a text prompt (DALL-E 3 and DALL-E 2), creating edited versions of

images by having the model change some areas of a pre-existing image based on a new text prompt (DALL-E 2 only) and creating variations of an existing image (DALL-E 2 only).

This guide covers the basics of the API methods with useful code examples.

4.5.1 Image Generation

The image generation method allows you to create an original image with a text prompt.

Initially, it is necessary to import the display and Image functions from the IPython.display module, in order to display the image inside a Jupyter notebook.

It also imports os, used for operating system operations, as explained earlier for creating a venv environment, and imports the OpenAI class from the openai module, an interface for using the API.

The code creates an instance of the OpenAI() client. A request is made to the API to generate an image, with the parameters:

- **model:** specifies the model used to generate the image, in this case "dall-e-2", which specialises in generating images based on textual descriptions;
- **prompt:** descriptive text that will serve as input for the model to generate the image, in this example, "A cat inside a car";
- **size:** specifies the size, 1024x1024 pixels;
- **quality:** sets the quality, "standard". When using DALL-E 3 it is possible to set quality: "hd", i.e. fine detail. However, standard quality square images are generated more quickly;
- **n:** defines the number of images generated.

After the API generates the image, the image URL is extracted from the response and stored in the *image_ url* variable.

Finally, the image URL is printed and the image is displayed using the display function with the Image class, passing the URL as a parameter and setting the image width to 500 pixels.

```
from IPython.display import display, Image

from openai import OpenAI

client = OpenAI()

response = client.images.generate(
  model="dall-e-2",
  prompt="A cat inside a car",
  size="1024x1024",
  quality="standard",
  n=1,
)

image_url = response.data[0].url
print(image_url)
display(Image(url=image_url, width=400))
```

Figure 8: Image generated

4.5.2 Edit (DALLE 2 only)

Also known as "inpainting", the image editing endpoint allows you to edit an image by loading an image and a mask indicating the areas that should be replaced, you can use tools such as [GIMP] or , to erase a certain area of the image. The transparent areas of the mask indicate where the image should be edited, and the prompt should describe the complete new image, not just the deleted area.

The image and mask sent must be square PNG images of less than 4 MB and must also have the same dimensions. The non-transparent areas of the mask are not used to generate the output, so they don't necessarily have to match the original image as in the following example, which shows the original image, the mask and the resulting image after the editing process.

(a) Original (b) Mask (c) Output

Figure 9: Input/Output from the edit endpoint

```
from openai import OpenAI
client = OpenAI()
from IPython.display import display, Image

response = client.images.edit(
  model="dall-e-2",
  image=open("original.png", "rb"),
  mask=open("mask.png", "rb"),
  prompt="A hand holding a sandwich",
  n=1,
  size="1024x1024"
```

```
    )
    image_url = response.data[0].url

    print(image_url)
    display(Image(url=image_url, width=400))
```

4.5.3 Variations (DALL·E 2 only)

The image variations endpoint allows you to generate a variation of a given image.

Similar to the edits endpoint, the input image must be a square PNG image less than 4MB in size.

```
    from IPython.display import display, Image
    from openai import OpenAI
    client = OpenAI()

    response = client.images.create_variation(
      model="dall-e-2",
      image=open("original.png", "rb"),
      n=1,
      size="1024x1024"
    )

    image_url = response.data[0].url

    print(image_url)
    display(Image(url=image_url, width=400))
```

The images below correspond to a possible example of the endpoint variations.

(a) Original (b) Variation

Figure 10: Input/Output from the variation endpoint

4.6 TTS

In this tutorial you will learn how to integrate text-to-speech feature from open-ai in your projects.

Voice only changes the tone and the "person" who is speaking. TTS only produces english audio files.

There are two models available tts right now: **tts-1** and **tts-1-hd**. If you want lower latency **tts-1** is recommended, but it comes with lower quality than **tts-1-hd**.

Available voices	Output Formats
Alloy	**mp3** - default format
Echo	**opus**
Fable	**aac**
Onyx	**flac**
Nova	**waw**
Shimmer	**pcm**

Table 3: TTS Information Table

1- Importing the necessary libraries and creating the client

pathlib offers classes that represent a filesystem path.

Listing 53: TTS tutorial step 1

```
from pathlib import Path
from openai import OpenAI

client = OpenAI()
```

2 - Create a path to save the audio file

In here you can choose the format you want your audio file to be in.

Listing 54: TTS tutorial step 2

```
speech_file_path = Path(f"Tutorials/TTS/tts_audio.mp3")
```

3 - Generate the audio

The endpoint will receive a model, a voice and your input.

Listing 55: TTS tutorial step 3

```
response = client.audio.speech.create(
  model="tts-1",
  voice="shimmer",
  input="Hey, I'm a student in Lincenciatura de Engenharia de Telecomunicacoes e
      Informatica in Instituto Superior de Engenharia do Porto, and I'm doing a tutorial
      on how to use open-a.i in my projects!"
)

response.write_to_file(speech_file_path)
```

4.7 Whisper

In this tutorial you will learn how to use Whisper to transcribe text from audio files as well translate it into English.

```
    import os
    from openai import OpenAI

    client = OpenAI()
```

Load the audio file

Feel free to try different audio files as well as add/record your own. Files must be of one of these types: mp3, mp4, mpeg, mpga, m4a, wav, and webm.

Note that files greater than 25MB will need to be segmented using additional libraries

```
audio_file = open("audio.wav", "rb")
```

1. **Transcribe an audio file**

   The transcription endpoint will take the input audio and transcribe it into text.

```
transcription = client.audio.transcriptions.create(
model="whisper-1",
file=audio_file,
response_format="text"
)

print(transcription)
```

   Notice how **response_format=̈text¨**? To get additional information to get additional information try changing it to **verbose_json**.

   You should now receive a json response with additional parameters. One of which, the **language** parameter, includes the detected language from the input file.

   Note: If the language is not being properly detected, which may negatively impact transcription, you can add an additional parameter stating it according to the ISO-639-1 format.

```
(
    model="whisper-1",
    file=audio_file,
    response_format="text"
    language="..."
)
```

   We can modify our code to reflect this:

```
transcription = client.audio.transcriptions.create(
    model="whisper-1",
    file=audio_file,
    response_format="verbose_json"
)
print(f"Detected language: {transcription.language}")
print(transcription.text)
```

2. **Translation**

   Using the translation endpoint we can translate the contents of the audio file to English (currently this the only available language for translation).

```
translation = client.audio.translations.create(
    model = "whisper-1",
    file = audio_file,
    response_format="text"
)
print(translation)
```

4.8 Moderation

The moderation endpoint is a tool you can use to check whether text is potentially harmful. It can be used to identify content that could be harmful and take action.

The templates classify the following categories:

- **Hate** - Content that expresses or promotes hatred based on race, gender, ethnicity, religion, nationality, sexual orientation, disability status or caste. Hateful content directed at unprotected groups constitutes harassment.
- **Hate/Threatening** - Hateful content that also includes violence or serious harm towards the targeted group based on race, gender, ethnicity, religion, nationality, sexual orientation, disability status, or caste.
- **Harassment** - Content that expresses, incites, or promotes harassing language towards any target.
- **Harassment/Threatening** - Harassment content that also includes violence or serious harm towards any target.
- **Self-harm** - Content that promotes, encourages, or depicts acts of self-harm, such as suicide, cutting, and eating disorders.
- **Self-harm/Intent** - Content where the speaker expresses that they are engaging or intend to engage in acts of self-harm, such as suicide, cutting, and eating disorders.
- **Self-harm/Instructions** - Content that encourages performing acts of self-harm, such as suicide, cutting, and eating disorders, or that gives instructions or advice on how to commit such acts.
- **Sexual** - Content meant to arouse sexual excitement, such as the description of sexual activity, or that promotes sexual services (excluding sex education and wellness).
- **Sexual/Minors** - Sexual content that includes an individual who is under 18 years old.
- **Violence** - Content that depicts death, violence, or physical injury.
- **Violence/Graphic** - Content that depicts death, violence, or physical injury in graphic detail.

To obtain a classification for a piece of text, a request is made to the moderation endpoint, as shown in the following code fragment. Firstly, the OpenAI class from the openai module is imported, then an instance of the class is created and assigned to a variable, configuring the client to interact with the OpenAI API.

With the **'moderations.create()'** method, the text is sent for moderation.

The response from the API will be a JSON object with information about the moderation of the text.

```
from openai import OpenAI

client = OpenAI()

response = client.moderations.create(
    input="I own a lot of guns which I intend to use, in order to harm people"
)

output = response.results[0]
print(output)
```

Below is an example of the endpoint response structure. It returns the following fields:

- **flagged**: Set to true if the model classifies the content as potentially harmful, false otherwise.
- **categories**: Contains a dictionary of violation flags for each category. The value will be true if the model flags the corresponding category as violated, false otherwise.
- **category_scores**: Contains a dictionary of raw scores per category issued by the model, denoting the model's confidence that the input violates the OpenAI policy for the category. The value is between 0 and 1, with higher values denoting greater confidence. The scores should not be interpreted as probabilities.

```
{
    "id": "modr-XXXXX",
    "model": "text-moderation-007",
    "results": [
        {
            "flagged": true,
            "categories": {
                "sexual": false,
                "hate": false,
                "harassment": false,
                "self-harm": false,
                "sexual/minors": false,
                "hate/threatening": false,
                "violence/graphic": false,
                "self-harm/intent": false,
                "self-harm/instructions": false,
                "harassment/threatening": true,
                "violence": true
            },
            "category_scores": {
                "sexual": 1.2282071e-6,
                "hate": 0.010696256,
                "harassment": 0.29842457,
                "self-harm": 1.5236925e-8,
                "sexual/minors": 5.7246268e-8,
                "hate/threatening": 0.0060676364,
                "violence/graphic": 4.435014e-6,
                "self-harm/intent": 8.098441e-10,
                "self-harm/instructions": 2.8498655e-11,
                "harassment/threatening": 0.63055265,
                "violence": 0.99011886
            }
        }
    ]

}
```

5 Exercises

5.1 Exercise A

Embeddings have a lot of uses, and when combined with other APIs, they can do even more. One example is using embeddings with chat completion to extract information from a PDF and then creating a function to ask anything about the document.

In the following exercise, you will create a program that retrieves information from a PDF and answers questions about it. In order to achieve this, you must:

- Convert a PDF file to embeddings and save them in a CSV file.
- Use embeddings to search a user query in the CSV file.
- Send that information to chat completions.

The PDF used in this exercise will be `LETI_SISTCA_2023_24_Team2_OpenAI.pdf`.

1. **Start by importing the requiring dependecies and initialize the client and creating some constants**

Listing 56: Exercise A - Part 1 Step 1

```
from openai import OpenAI
import pandas as pd
import re
import tiktoken
import PyPDF2

import ast
from scipy import spatial

client = OpenAI()


SECTIONS_TO_IGNORE = [
    "Contents",
    "List of Tables",
    "List of Figures",
    "References",
]

MAX_TOKENS = 1600
BATCH_SIZE = 1000

# TODO #1: Create consts for GPT_MODEL and EMBEDDING_MODEL (small)
```

2. **Simple Logic to Extract the information from the pdf**

   This is a simple logic to extract the necessary information from the pdf we are going to use.

Listing 57: Exercise A - Part 1 Step 2

```
def extract_text_from_pdf(pdf_path):
    with open(pdf_path, 'rb') as file:
        reader = PyPDF2.PdfReader(file)
        text = ''
        for page_num in range(len(reader.pages)):
            text += reader.pages[page_num].extract_text()
    return text
```

```
8
9       def split_sections_from_pdf(pdf_text):
10          title = []
11          text = []
12          ignore = True
13          current_section = "I.N.I.T.I.A.L-V.A.L.U.E"
14          for line in pdf_text.split('\n'):
15              line = line.strip()
16
17              if not line:
18                  continue
19              if is_new_section(line):
20                  ignore = ignore_section(line)
21                  if ignore:
22                      continue
23                  title.append(line)
24                  if current_section != "I.N.I.T.I.A.L-V.A.L.U.E":
25                      text.append(current_section)
26                  current_section = ""
27              else:
28                  if not ignore:
29                      current_section += " " + line
30          if current_section:
31              text.append(current_section)
32
33
34          sections = [(title),(text)]
35          return sections
36
37      def ignore_section(line):
38          if any(section in line for section in SECTIONS_TO_IGNORE):
39              return True
40          return False
41
42      def is_new_section(line):
43          pattern = r"\d+\.\d+(?:\.\d+)? [A-Z].*?"
44
45          if line.strip().count('.') > 7:
46              return False
47          if re.match(pattern, line.strip()):
48              return True
49          if any(section in line for section in SECTIONS_TO_IGNORE):
50              return True
51
52          return False
53
54      def clean_section(section):
55
56          titles = section[0]
57          text = section[1]
58
59          for line in text:
60              line = re.sub(r"\[\d+\]", "", line)
61              line = re.sub(r"\[\d\d+\]", "", line)
62              line = line.strip()
63
64          return (titles, text)
65
66
67      def num_tokens(text, model = GPT_MODEL):
68          encoding = tiktoken.encoding_for_model(model)
69          return len(encoding.encode(text))
70
```

```
    def halved_by_delimiter(string, delimiter = "\n"):
        chunks = string.split(delimiter)
        if len(chunks) == 1:
            return [string, ""]
        elif len(chunks) == 2:
            return chunks
        else:
            total_tokens = num_tokens(string)
            halfway = total_tokens // 2
            best_diff = halfway
            for i, chunk in enumerate(chunks):
                left = delimiter.join(chunks[: i + 1])
                left_tokens = num_tokens(left)
                diff = abs(halfway - left_tokens)
                if diff >= best_diff:
                    break
                else:
                    best_diff = diff
            left = delimiter.join(chunks[:i])
            right = delimiter.join(chunks[i:])
            return [left, right]


    def truncated_string(string, model, max_tokens, print_warning = True,):
        encoding = tiktoken.encoding_for_model(model)
        encoded_string = encoding.encode(string)
        truncated_string = encoding.decode(encoded_string[:max_tokens])
        if print_warning and len(encoded_string) > max_tokens:
            print(f"Warning: Truncated string from {len(encoded_string)} tokens to
                {max_tokens} tokens.")
        return truncated_string


    def split_strings_from_subsection(title, text, max_tokens = 1000, model =
        GPT_MODEL, max_recursion = 5):

        string = "\n\n".join(title + text)
        num_tokens_in_string = num_tokens(string, model)

        if num_tokens_in_string <= max_tokens:
            return [string]

        elif max_recursion == 0:
            return [truncated_string(string, model, max_tokens)]

        else:
            for delimiter in ["\n\n", "\n", ". "]:
                left, right = halved_by_delimiter(text, delimiter=delimiter)
                if left == "" or right == "":

                    continue
                else:

                    results = []
                    for half in [left, right]:
                        half_strings = split_strings_from_subsection(title,
                            half,max_tokens,model,max_recursion - 1)
                        results.extend(half_strings)
                    return results

        return [truncated_string(string, model, max_tokens)]
```

3. **Call all functions to retrieve the clean pdf sections to then split it into strings**

Listing 58: Exercise A - Part 1 Step 3

```
1
2     # TODO #2: Call the previous created functions
3     pdf_text = ...
4     pdf_sections = ...
5     cleaned_sections = ...
6
7     MAX_TOKENS = 1600
8     strings = []
9     titles = cleaned_sections[0]
10    texts = cleaned_sections[1]
11    for i in range(len(titles)):
12        strings.extend(split_strings_from_subsection(titles[i], texts[i],
              max_tokens=MAX_TOKENS))
```

4. **Transforming the information to embeddings and saving it to a CSV file**

Listing 59: Exercise A - Part 1 Step 4

```
1
2
3     embeddings = []
4     for batch_start in range(0, len(strings), BATCH_SIZE):
5         batch_end = batch_start + BATCH_SIZE
6         batch = strings[batch_start:batch_end]
7
8         # TODO #3: Make a request to the embeddings API with the batch as input
9
10
11        for i, be in enumerate(response.data):
12            assert i == be.index
13        batch_embeddings = [e.embedding for e in response.data]
14        embeddings.extend(batch_embeddings)
15
16
17    df = pd.DataFrame({"text": strings, "embedding": embeddings})
18
19
20    SAVE_PATH = "SISTCA_TEAM2.csv"
21    df.to_csv(SAVE_PATH, index=False)
```

5. **The first part is done, now we need to create a function so GPT can awnser anything about the pdf using the saved embeddings**

   Change the Embedding Model and read the CSV file

Listing 60: Exercise A - Part 2 Step 1

```
1
2     # TODO #4: Change the EMBEDDING_MODEL (ada)
3
4
5     # TODO #5: Create a variable with the CSV file path
6
7
8     df = pd.read_csv(embeddings_path)
9     df['embedding'] = df['embedding'].apply(ast.literal_eval)
```

6. **Functions to compare the relatedness off the strings with the query**

Listing 61: Exercise A - Part 2 Step 2

```
1
```

```
    def strings_ranked_by_relatedness(query, df , relatedness_fn = lambda x, y: 1 -
        spatial.distance.cosine(x, y), top_n = 100) :

        # TODO 6: Make a request to the embeddings API with the query as input


        query_embedding = query_embedding_response.data[0].embedding
        strings_and_relatednesses = [
            (row["text"], relatedness_fn(query_embedding, row["embedding"]))
            for i, row in df.iterrows()
        ]
        strings_and_relatednesses.sort(key=lambda x: x[1], reverse=True)
        strings, relatednesses = zip(*strings_and_relatednesses)
        return strings[:top_n], relatednesses[:top_n]

    strings, relatednesses = strings_ranked_by_relatedness("open ai", df, top_n=5)


    def num_tokens(text, model = GPT_MODEL):
        encoding = tiktoken.encoding_for_model(model)
        return len(encoding.encode(text))

    def query_message(query,df, model, token_budget):
        strings, relatednesses = strings_ranked_by_relatedness(query, df)
        introduction = 'Use the below articles on the document about OpenAI made by
            Team 2, composed by Patricia Sousa, Carlos Alves, Jose Leal and Tiago
            Ribeiro, for SISTCA to answer the subsequent question. If the answer
            cannot be found in the articles, write "Sorry, the information you seek
            cannot be found in the document in question."'
        question = f"\n\nQuestion: {query}"
        message = introduction
        for string in strings:
            next_article = f'\n\nPDF article section:\n"""\n{string}\n"""'
            if (
                num_tokens(message + next_article + question, model)
                > token_budget
            ):
                break
            else:
                message += next_article
        return message + question
```

7. Create the final function

If the chat completions do not have information to answer the query, as defined, it will say *"Sorry, the information you seek cannot be found in the document in question."*

Listing 62: Exercise A - Part 2 Step 3

```

    def ask(query, df = df, model = GPT_MODEL, token_budget = 4096 - 500,
        print_message = False):
        message = query_message(query, df, model=model, token_budget=token_budget)
        if print_message:
            print(message)
        messages = [
            {"role": "system", "content": "You answer questions about the document
                made by Team2 for SISTCA about OpenAI."},
            {"role": "user", "content": message},
        ]

        # TODO 7: Make a request to the Chat Completions API


```

```
        response_message = response.choices[0].message.content
        return response_message
```

8. **Test the ask function to verify it**

   Because every time you run ask you give a new prompt to chat completions the answers may very for one attempt to another.

Listing 63: Exercise A - Part 2 Step 4

```

    print(ask("Scientific/technological background"))
    print(ask("Give me the authors"))
    print(ask("What can you tell me about the document"))
    print(ask("Give me the document structure"))
```

The solution for this exercise is available here on github.

5.2 Exercise B

Despite, in it's current state, only allowing English translation, we can increment the Whisper API with other APIs so as to be able to translate audio into other languages. By using the standard GPT model in the Chat Completions API we can translate text to and from any language.

In the following exercise you will create a program that translates an audio message into any other language. In order to achieve this you must:

- Use Whisper to transcribe the original audio;
- Translate the resulting transcription into any language using Chat Completions;
- Return the translated audio using Text-To-Speech.

1. **Start by importing the required dependencies and initializing the client**

```
            import os
            from dotenv import load_dotenv
            from openai import OpenAI

            load_dotenv()

            api_key = os.getenv("OPENAI_API_KEY")

            client = OpenAI(api_key=api_key)
```

2. **Use Whisper to transcribe the original audio**

```
            # TODO #1: load the audio file


            # TODO #2: transcribe contents and detect the input language
```

3. **Translate the transcribed text into any language using Chat Completions**

```
            # TODO #3: translate the transcription to another language using
                chatCompletions
```

4. **Return the translated audio using Text-To-Speech**

```
            # TODO #4: output the translated audio file
            
            
            
            
            
            # Save the output audio file
            output_file_path = Path(__file__).parent / f"translation.mp3"
            translated_audio.write_to_file(output_file_path)
```

```
            # Test the output
            import IPython.display as display

            print(f"Detected language: {detected_language} \n")
            print(f"Transcribed text: {transcribed_text} \n")

            print(f"Translated text: {translated_text} \n")
            print(f"Translated audio file saved to {output_file_path}")


            display.Audio(speech_file_path)
```

The solution for this exercise can be found here.

6 Challenge

With all tutorials done, you should now be able to answer correctly to this final challenge. In this section we propose you the following exercise: You have to create an assistant which will be capable of analysing and answering questions about films, series and books. The average question should look like this, 'Give me the top 10 best films of all time.'.

We advise that you follow this steps:

- Create an Assistant with function calling.
  - For a better flexibility you should retrieve from the user query the type (movie, series, book), theme(comedy, thriller...) and quantity.
- Create a function to retrieve information from Chat Completions.
  - You can force Chat Completions to send the information in a specific format, like JSON.
- Show the response in a bullet list.

In our github you can find a Flask template where you can visually test the results of your challenge. To have a compatibility with the template, make sure that your final function, the one that retrieves the assistant message, follows this format:

Listing 64: Challenge Flask Function

```
def get_cinema_info(type_filter, genre_filter, quantity_filter):
    ...
```

7 Future Work

The OpenAI API is still an in development effort this means that everything is prone to change. In fact, some of it has changed during the development of this script. As previously noted before some of the functionalities that have been addressed is not yet available for the explorer tier of the API, which includes the "GPT-3.5-Turbo" model. At the time of writing this, the launch of a new flagship model is on the horizon - "GPT-4o" [11]. This should make previously tier-locked APIs such as Vision accessible to everyone, as well as potentially added new features.

Despite the rather wide scope of this script some of OpenAI's tools haven't been explored due to them being unavailable in the API. Such is the case for Sora [12], a video generation model, or the research publications like CLIP [13, 14], Jukebox [15, 16] or Point-E [17, 18]. Another feature unable to be explored, despite being compatible with "GPT-3.5-Turbo", was fine-tuning. This feature allows to train "GPT" with your own dataset [19].

For a future update of this script or development of a new one with the same topic you can explored this features.

8 Appendix

8.1 Exercise A - Solution

Solution for Exercise A, subsection 5.1.

Listing 65: Exercise A Solution

```
from openai import OpenAI
import pandas as pd
import re
import tiktoken
import PyPDF2

import ast
from scipy import spatial

client = openai.OpenAI()

SECTIONS_TO_IGNORE = [
    "Contents",
    "List of Tables",
    "List of Figures",
    "References",
]

MAX_TOKENS = 1600
BATCH_SIZE = 1000

# TODO #1: Create consts for GPT_MODEL and EMBEDDING_MODEL (small)
EMBEDDING_MODEL = "text-embedding-3-small"
GPT_MODEL = "gpt-3.5-turbo"

def extract_text_from_pdf(pdf_path):
    with open(pdf_path, 'rb') as file:
        reader = PyPDF2.PdfReader(file)
        text = ''
        for page_num in range(len(reader.pages)):
            text += reader.pages[page_num].extract_text()
    return text

def split_sections_from_pdf(pdf_text):
    title = []
    text = []
    ignore = True
    current_section = "I.N.I.T.I.A.L-V.A.L.U.E"
    for line in pdf_text.split('\n'):
        line = line.strip()

        if not line:
            continue
        if is_new_section(line):
            ignore = ignore_section(line)
            if ignore:
                continue
            title.append(line)
            if current_section != "I.N.I.T.I.A.L-V.A.L.U.E":
                text.append(current_section)
            current_section = ""
        else:
            if not ignore:
                current_section += " " + line
```

```
    if current_section:
        text.append(current_section)


    sections = [(title),(text)]
    return sections

def ignore_section(line):
    if any(section in line for section in SECTIONS_TO_IGNORE):
        return True
    return False

def is_new_section(line):
    pattern = r"\d+\.\d+(?:\.\d+)? [A-Z].*?"

    if line.strip().count('.') > 7:
        return False
    if re.match(pattern, line.strip()):
        return True
    if any(section in line for section in SECTIONS_TO_IGNORE):
        return True

    return False

def clean_section(section):

    titles = section[0]
    text = section[1]

    for line in text:
        line = re.sub(r"\[\d+\]", "", line)
        line = re.sub(r"\[\d\d+\]", "", line)
        line = line.strip()

    return (titles, text)


def num_tokens(text, model = GPT_MODEL):
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))

def halved_by_delimiter(string, delimiter = "\n"):
    chunks = string.split(delimiter)
    if len(chunks) == 1:
        return [string, ""]
    elif len(chunks) == 2:
        return chunks
    else:
        total_tokens = num_tokens(string)
        halfway = total_tokens // 2
        best_diff = halfway
        for i, chunk in enumerate(chunks):
            left = delimiter.join(chunks[: i + 1])
            left_tokens = num_tokens(left)
            diff = abs(halfway - left_tokens)
            if diff >= best_diff:
                break
            else:
                best_diff = diff
        left = delimiter.join(chunks[:i])
        right = delimiter.join(chunks[i:])
        return [left, right]

```

```
118
119 def truncated_string(string, model, max_tokens, print_warning = True,):
120     encoding = tiktoken.encoding_for_model(model)
121     encoded_string = encoding.encode(string)
122     truncated_string = encoding.decode(encoded_string[:max_tokens])
123     if print_warning and len(encoded_string) > max_tokens:
124         print(f"Warning: Truncated string from {len(encoded_string)} tokens to
              {max_tokens} tokens.")
125     return truncated_string
126
127
128 def split_strings_from_subsection(title, text, max_tokens = 1000, model = GPT_MODEL,
      max_recursion = 5):
129
130     string = "\n\n".join(title + text)
131     num_tokens_in_string = num_tokens(string, model)
132
133     if num_tokens_in_string <= max_tokens:
134         return [string]
135
136     elif max_recursion == 0:
137         return [truncated_string(string, model, max_tokens)]
138
139     else:
140         for delimiter in ["\n\n", "\n", ". "]:
141             left, right = halved_by_delimiter(text, delimiter=delimiter)
142             if left == "" or right == "":
143
144                 continue
145             else:
146
147                 results = []
148                 for half in [left, right]:
149                     half_strings = split_strings_from_subsection(title,
                          half,max_tokens,model,max_recursion - 1)
150                     results.extend(half_strings)
151                 return results
152
153     return [truncated_string(string, model, max_tokens)]
154
155
156 # TODO #2: Call the previous created functions
157 pdf_text = extract_text_from_pdf("LETI_SISTCA_2023_24_Team2_OpenAI.pdf")
158
159 pdf_sections = split_sections_from_pdf(pdf_text)
160
161 cleaned_sections = clean_section(pdf_sections)
162
163 MAX_TOKENS = 1600
164 strings = []
165 titles = cleaned_sections[0]
166 texts = cleaned_sections[1]
167 for i in range(len(titles)):
168     strings.extend(split_strings_from_subsection(titles[i], texts[i],
          max_tokens=MAX_TOKENS))
169
170
171
172 embeddings = []
173 for batch_start in range(0, len(strings), BATCH_SIZE):
174     batch_end = batch_start + BATCH_SIZE
175     batch = strings[batch_start:batch_end]
176     # TODO #3: Make a request to the embeddings API with the batch as input
```

```
    response = client.embeddings.create(model=EMBEDDING_MODEL, input=batch)
    for i, be in enumerate(response.data):
        assert i == be.index
    batch_embeddings = [e.embedding for e in response.data]
    embeddings.extend(batch_embeddings)


df = pd.DataFrame({"text": strings, "embedding": embeddings})


SAVE_PATH = "SISTCA_TEAM2.csv"
df.to_csv(SAVE_PATH, index=False)

# TODO #4: Change the EMBEDDING_MODEL (ada)
EMBEDDING_MODEL = "text-embedding-ada-002"

GPT_MODEL = "gpt-3.5-turbo"

# TODO #5: Create a variable with the CSV file path
embeddings_path = "SISTCA_TEAM2.csv"

df = pd.read_csv(embeddings_path)
df['embedding'] = df['embedding'].apply(ast.literal_eval)

def strings_ranked_by_relatedness(query, df , relatedness_fn = lambda x, y: 1 -
    spatial.distance.cosine(x, y), top_n = 100) :

    # TODO 6: Make a request to the embeddings API with the query as input
    query_embedding_response = client.embeddings.create(
        model=EMBEDDING_MODEL,
        input=query,
    )

    query_embedding = query_embedding_response.data[0].embedding
    strings_and_relatednesses = [
        (row["text"], relatedness_fn(query_embedding, row["embedding"]))
        for i, row in df.iterrows()
    ]
    strings_and_relatednesses.sort(key=lambda x: x[1], reverse=True)
    strings, relatednesses = zip(*strings_and_relatednesses)
    return strings[:top_n], relatednesses[:top_n]

strings, relatednesses = strings_ranked_by_relatedness("open ai", df, top_n=5)

def num_tokens(text, model = GPT_MODEL):
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))

def query_message(query,df, model, token_budget):
    strings, relatednesses = strings_ranked_by_relatedness(query, df)
    introduction = 'Use the below articles on the document about OpenAI made by Team 2,
        composed by Patrcia Sousa, Carlos Alves, Jose Leal and Tiago Ribeiro, for SISTCA
        to answer the subsequent question. If the answer cannot be found in the articles,
        write "Sorry, the information you seek cannot be found in the document in
        question."'
    question = f"\n\nQuestion: {query}"
    message = introduction
    for string in strings:
        next_article = f'\n\nPDF article section:\n"""\n{string}\n"""'
        if (
            num_tokens(message + next_article + question, model)
            > token_budget
        ):
```

```
            break
        else:
            message += next_article
    return message + question


def ask(query, df = df, model = GPT_MODEL, token_budget = 4096 - 500, print_message = False):
    message = query_message(query, df, model=model, token_budget=token_budget)
    if print_message:
        print(message)
    messages = [
        {"role": "system", "content": "You answer questions about the document made by Team2 for SISTCA about OpenAI."},
        {"role": "user", "content": message},
    ]

    # TODO 7: Make a request to the Chat Completions API

    response = client.chat.completions.create(
        model=model,
        messages=messages,
        temperature=0
    )
    response_message = response.choices[0].message.content
    return response_message

print(ask("Tell me about whisper or tts"))
print(ask("Give me the authors"))
print(ask("What can you tell me about the document"))
print(ask("Give me the document structure"))
```

8.2 Exercise B - Solution

Solution for Exercise B, subsection 5.2.

Listing 66: Exercise B Solution

```
from pathlib import Path
from openai import OpenAI

client = OpenAI()


# load the audio file
audio_input = open("Exercises/Exercise_B/audio.wav", "rb")


# transcribe contents and detect the input language
transcription = client.audio.transcriptions.create(
    model="whisper-1",
    file=audio_input,
    response_format="verbose_json"
)

detected_language = transcription.language


# translate the transcription to another language using chatCompletions
output_language = "french"

```

```
translation = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "user", "content": f"Translate the following text to {output_language}:
            {transcription.text}"}
    ]
)

translated_text = translation.choices[0].message.content


# output the translated audio file
translated_audio = client.audio.speech.create(
  model="tts-1",
  voice="shimmer",
  input=translated_text
)


output_file_path = Path(__file__).parent / f"translation.mp3"

translated_audio.write_to_file(output_file_path)


# test the output

print(f"Detected language: {detected_language} \n")
print(f"Transcribed text: {transcription.text} \n")

print(f"Translated text: {translated_text} \n")
print(f"Translated audio file saved to {output_file_path}")
```

References

[1] "Project Jupyter Documentation — Jupyter Documentation 4.1.1 alpha documentation." [Online; accessed 2024-05-18].

[2] IBM, "What is an ai model?," *IBM*. Accessed: 2024-03-28.

[3] R. Merritt, "What is a transformer model?," *NVIDIA Blog*, mar 2022. (Accessed: 2024-03-28.

[4] "Openai." Available at: https://openai.com/. Accessed: 2024-03-27.

[5] xAI, "xai grok." Available at: https://grok.x.ai/. (Accessed: 2024-03-27).

[6] Anthropic, "Anthropic home." Available at: https://www.anthropic.com/. (Accessed: 2024-03-27).

[7] Google, "Deepmind technologies." Available at: https://deepmind.google/technologies/. (Accessed: 2024-03-27).

[8] Cohere, "Cohere home." Available at: https://cohere.com/. (Accessed: 2024-03-27).

[9] "Python." Available at: https://wiki.python.org/. (Accessed: 2024-03-29).

[10] "Jupyter." Available at: https://jupyter.org/. (Accessed: 2024-05-18).

[11] "Hello gpt-4o." Available at: https://openai.com/index/hello-gpt-4o/. (Accessed: 2024-05-18).

[12] "Sora." Available at: https://openai.com/index/sora/. (Accessed: 2024-05-18).

[13] "Clip: Connecting text and images." Available at: https://openai.com/index/clip/. (Accessed: 2024-05-18).

[14] "Clip on github." Available at: https://github.com/openai/CLIP. (Accessed: 2024-05-18).

[15] "Jukebox sample explorer." Available at: https://jukebox.openai.com/. (Accessed: 2024-05-18).

[16] "Jukebox on github." Available at: https://github.com/openai/jukebox. (Accessed: 2024-05-18).

[17] "Point-e: A system for generating 3d point clouds from complex prompts." Available at: https://openai.com/index/point-e/. (Accessed: 2024-05-18).

[18] "Point-e on github." Available at: https://github.com/openai/point-e. (Accessed: 2024-05-18).

[19] "Fine-tuning." Available at: https://platform.openai.com/docs/guides/fine-tuning. [Online; accessed 2024-05-18].