CUSTOM CHATBOT USING LLAMA_INDEX AND OPENAI API

Anne-Sophia LIM

Supervisor: Dr. Edwin FOO

May - July 2023

NANYANG THE INNOVATIVE POLYTECHNIC

ESIEE PARIS

Acknowledgement

NANYANG THE INNOVATIVE POLYTECHNIC

ESIEE PARIS

Table of contents

Introduction

Open ai api

My data

Chatbot my test files

Speech to text / text to speech

Introduction

According to the website Statista.com, the volume of data available in 2020 was about 64 zettabytes, which means 64 trillion gigabytes and this number should triple by 2025. This incredible amount of information is so much to manage and to sort that people can struggle looking for information efficiently. [1] That is why more and more automated help service agents are created on websites and that eventually led to the creation of chatbot like the very well-known ChatGPT. But the problem is that the latest data trained on are from September 2021. One solution to have the chatbot have access to newest data is to train our own chatbot, but to optimize computational resources, minimize financial costs, and reduce training time, it should be trained using targeted data.

This project is about a custom chatbot using the OpenAI API which data comes from the Nanyang Polytechnic website. It can answer questions about two specific courses: Robotics and Mechatronics and Engineering with business. This should have been implemented into a robot but I only worked on the python code.

Introduction of the custom chatbot

The chatbot was created using the OpenAI API and the OpenAI's GPT-3.5 model. This model (and its other versions) is commonly used in tasks involving natural language processing and have shown remarkable performance in various language-related tasks.

Definition of NLP:

First of all, we should define what NLP is. It stands for 'Natural Language Processing' and is a branch of computer science and in particular, the branch of Artificial Intelligence. NLP encompasses a broad range of tasks like text classification, sentiment analysis, machine translation, question answering, and more, using machine learning and deep learning models to deal with data. It can be encountered in digital assistants, speech-to-text dictation softwares or customer service chatbots. Firms also use this more and more to help employees to increase their productivity.

[https://www.ibm.com/topics/natural-language-processing#:~:text=the%20next%20step-,What%20is%20natural%20language%20processing%3F,same%20way%20human%20beings%20can.] [https://www.coursera.org/articles/computational-linguistics]

Definition of LLM:

A Large Language Model is a trained deep learning model that can process and understand a large amount of texts and generate a response in a human-like manner. They learnt from a massive number of data during pre training. They are built on the transformer architecture, which is a neural network architecture. Examples of large language models include OpenAI's GPT models, or BERT (developped by Google).

We can now talk about OpenAI, which is a key player in the world of Artificial Intelligence, contributing significantly to its advancements. Its development has influenced various AI applications, such as natural language processing, computer vision, and reinforcement learning, shaping the cutting-edge research in the AI community.

What is OpenAI ?

OpenAI is a private laboratory founded in 2015 in San Francisco, some of its founders are Ilya Sutskever (CSO), Greg Brockman (the actual chairman and president of openAI), Sam Altman (current CEO) and Elon Musk (before he leaves 3 years later due to potential conflict of interests because of his status as the CEO of Tesla). OpenAI aims to develop safe artificial intelligence to the "benefit of all humanity".
In 2018 was released GPT-1, the first large language model by openAI. Two years later, GPT-3 was released, which is a model of a pretrained language model on large dataset. The

next year was the release of Dall-E that generates unique and creative images from textual descriptions. Then came ChatGPT in late 2022, a language model chatbot built on top of GPT-3 and 4 months ago GPT-4 was released. It can now understand visual input and push the limitation of the token limits to 8 192 tokens (GPT 3.5 limitation was up to 4 096 tokens)

What is the OpenAI API?

The OpenAI API (application programming interface) allows developers to access and integrate OpenAI's language models and natural language processing capabilities for different use cases. There are several

- See what are the base model davinci, curie, ada…

è openAI FineTuning, what is it ?

here is a website that talks about fine tuning and chat completion pretty well (HOW TO FINE TUNE !! Ive never done it the right way, fine tuning is with questions/answers and we can rank them with his method!!) :

https://betterprogramming.pub/openai-api-fine-tuned-models-vs-chat-completion-a-case-study-e3774fadc8c7

The training process and ongoing usage determine pricing.

For example, if you are training "Curie" with a dataset of 1000 items, each consisting of 500 tokens, and the training process takes four cycles (more on that later), training will use 500*1000*4 = 2,000,000 tokens. Resulting in a cost of (2,000,000/1,000) * $0.0030 = $6. Once the model is trained, you pay for ongoing usage and do not need to pay for training it again unless you want to train it further, which is possible.

Model	Maximum text length
gpt-3.5-turbo	4,096 tokens (~5 pages)
gpt-4	8,192 tokens (~10 pages)
gpt-4-32k	32,768 tokens (~40 pages)

OpenAI Fine-Tuning Pricing (June 1st, 2023) above

Model	Training	Usage
Ada	$0.0004 / 1K tokens	$0.0016 / 1K tokens
Babbage	$0.0006 / 1K tokens	$0.0024 / 1K tokens
Curie	$0.0030 / 1K tokens	$0.0120 / 1K tokens
Davinci	$0.0300 / 1K tokens	$0.1200 / 1K tokens

Difference between GPT openai, huggingface and transformers

Why did I not use other language model?

Simple Transformers Questionaswering : not useful because it used json files for training and in each file, there should be only a single list of dictionaries: Each such dictionary contains two attributes, the "context" and "qas".

- context: The paragraph or text from which the question is asked.
- qas: A list of questions and answers.

The list of questions and answer is not possible to give because there could be hundreds of different questions, plus, we should provide the answers ourselves, which is not what we want to do.

link : https://towardsdatascience.com/question-answering-with-bert-xlnet-xlm-and-distilbert-using-simple-transformers-4d8785ee762a

The greatest tuto for chatbot but it is in javascript…

https://www.youtube.com/watch?v=Ix9WIZpArm0

My chatbot code

I tried several tutorials to start my project. Some of them start from scratch, meaning that we had to train the chatbot to write the basic sentences, but as we cannot train the chatbot as much as ChatGPT did, it will never be as accurate and precise in its response so this was not the more optimal way to start. Most of the tutorial began with a pretrained model, like the one I used.

(If I want to stream part by part like ChatGPT : https://github.com/openai/openai-cookbook/blob/main/examples/How_to_stream_completions.ipynb)

Different types of Openai API:

For those with the pretrained model, there are some difference between the different library used. I first tried with the openai library. It allows us to use Openai.Completion or Openai.Chat.Completion API. Both return answers from a given input. They need different parameters. For example, the Chat.Completion API needs a model that will be used and a list of messages that will give an example of how it should respond, with the different roles. Here is an example:

```
{
  "model": "gpt-3.5-turbo",
   "messages": [{"role": "system", "content": "You are a helpful assistant."}, {"role": "user", "content": "Hello!"}]
}
```

The message is generally a list of message role/content, that begins with the system, followed by an exchange between the user and the assistant.
Other parameters can be set up, such as a function created by itself or implemented by us that can be called when it is relevant to use it in the response, and a temperature that will create a more or less random or deterministic response, which I will explain below.

For the Completion API, the main difference is that we can enter a prompt instead of messages. The model will then return one or more predicted completions.
“https://betterprogramming.pub/openai-api-fine-tuned-models-vs-chat-completion-a-case-study-e3774fadc8c7” pour cette partie, fine tuning incroyable!

Problem of number of tokens:

However, there is a limitation of tokens taken for each prompt. Depending on the model, for example, if we use the “text-davinci-003” or “chat-gpt-3.5” model, the limitation is 4096 tokens per request. But for the “chat-gpt-4” the limitation reach 8192 tokens per request.

The problem here is that the number of tokens for the file for Robotics & Mechatronics has each time exceeded the maximum token size allowed and there was still the Engineering

and Business file to take into account. I first tried to put a text file converted from the html file of the website "https://www.nyp.edu.sg/schools/seg/full-time-courses/robotics-and-mechatronics.html" without having properly processed the data. The HTML file contained the code used to properly display the information on the website in HTML, there were a lot of unnecessary lines so I removed them from the text file. This one works but some of the data were not accurately processed (The chatbot cannot differentiate the different elective programmes of the third year) so I decided to create a JSON file containing only the useful information to structure the data. However, I came back to the original problem, which was the number of tokens exceeded as shown below.

```
InvalidRequestError: This model's maximum context length is 4097 tokens, however you requested 6504 tokens
```

I decided to try with the Llama_index library, which is more flexible when it comes to data integration.

What is Llama_index?

Llama_index is a data framework that help us connect our LLM with external data. It provides tools such as data connectors that combine datasets from any source. They can be structured (from Excel or SQL for example), unstructured (like raw text files or PDFs) or semi-structured (these datasets come from APIs like Slack for example). It can also index our data. Indexing data allows a more efficient search over the dataset by creating a data structure optimized for retrieval. Even if the volume of data increase, it can manage the scalability and maintain a high search performance. Plus, it can be used to create embeddings to help on relatedness of texts. Finally, It can supplies a retrieval and query interface and can be integrated with other frameworks like LangChain or ChatGPT. With an input prompt over our data, it can return a knowledge-augmented response.

With Lama_index, I chose to use the "text-davinci-003" model. To do so, I had to put an openAI API key. This key allows us to enter prompts and get back a response with taking into account the rate limits but not the maximum number of tokens. This limit is used to regulate the number of times the server can be accessed within a precise period of time (contrary to the maximum number of token which limit, as said, the number of token per prompt). The reasons for limitation is variable, it ranges from protecting the server against a misuse or abuse of the API to regulate the flow in order to give a more fair access to other people using it without any slow service or interruption. They are measured in two ways: RPM (requests per minute) and TPM (tokens per minute). The table below shows the maximum amount of RPM and TPM allowed. We can see that the amount differs according to the different model used.

	TEXT & EMBEDDING	CHAT	EDIT	IMAGE	AUDIO
Free trial users	3 RPM 150,000 TPM	3 RPM 40,000 TPM	3 RPM 150,000 TPM	5 images / min	3 RPM
Pay-as-you-go users (first 48 hours)	60 RPM 250,000 TPM	60 RPM 60,000 TPM	20 RPM 150,000 TPM	50 images / min	50 RPM
Pay-as-you-go users (after 48 hours)	3,500 RPM 350,000 TPM	3,500 RPM 90,000 TPM	20 RPM 150,000 TPM	50 images / min	50 RPM

image : https://platform.openai.com/docs/guides/rate-limits/overview (used this website for info)

Moreover, the number of token is not equivalent regarding the old models. In the below table we can see that for example, 1 TPM is equal to 25 TPM for the curie model.

TYPE	1 TPM EQUALS
davinci	1 token per minute
curie	25 tokens per minute
babbage	100 tokens per minute
ada	200 tokens per minute

I start my code with a function called createVectorIndex() that is used for embeddings.
The first part of this function is creating a prompt helper with parameters.

```
def createVectorIndex():
    max_input = 4096
    tokens = 256
    max_chunk_overlap = 20
    chunk_size = 600

    prompt_helper = PromptHelper(max_input, tokens, max_chunk_overlap, chunk_size_limit=chunk_size)
```

The PromptHelper class helps us to split the dataset into smaller chunks according to the limitations we put as hyperparameters.

The second part is the creation of LLM, based on an openai model.

```
#define LLM
llmPredictor = LLMPredictor(llm=OpenAI(
    temperature=0.5,
    model_name="text-davinci-003",
    max_tokens=tokens
```

The LLMPredictor class is wrapper around an LLMChain from LangChain. It handles the conversion from the prompt input it has received to a string that can be understood by the LLM put inside. The hyperparameter temperature here controls the randomness and thus the creativity of the response of the LLM. The higher the temperature is, the more random the response will be.

For example, I ask the chatbot “what are the elective programmes of robotics?”, this is the response I got with a temperature of 0.2:
“The two elective programmes of robotics are Automation & Robotics Technology and Wafer Fabrication Technology. ”

For the temperature of 0.8:
“The two elective programmes of robotics are Automation & Robotics Technology and Aerospace Technology.”
We can see that the first response is correct whereas the second one is more random (and half false).
For our case, as we want an accurate answer, we will put a low temperature.

This is where we choose the LLM, here it is the openAI’s “text-davinci-003” model.

The function createVectorIndex() then loads the documents from a folder with the SimpleDirectoryReader() function, creates indexes from these documents (with GPTVectorStoreIndex()’s from_documents() function) and sets the index id (thanks to set_index_id()).

The second function of my code answerMe(vectorIndex) is used for saving the indexing to the storage with the storage_context.persist() function, it allows the LLM to load and have access when creating the query engine with the as_query_engine() function. The user can interact with the chatbot through the input which will send the query to the query engine, that will then deliver the response.

(https://levelup.gitconnected.com/connecting-chatgpt-with-your-own-data-using-llamaindex-663844c06653 site incroyable qui explique le fonctionnement de mon code en llama_index avec des explications détaillées: connecting your chatbot with your own data using llama-index)

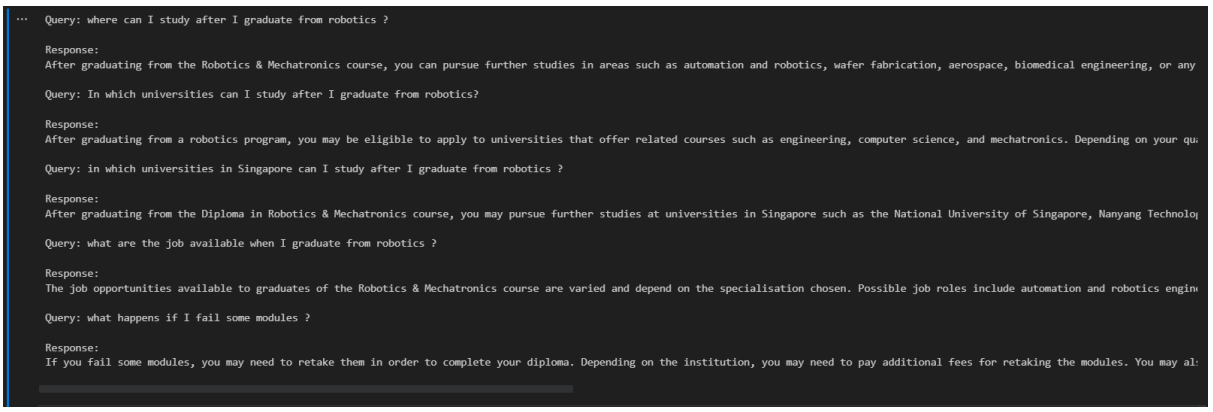
- **llama_index** is a project that provides a central interface to connect your LLM's with external data.
- **LangChain** is a framework for developing applications powered by language models.

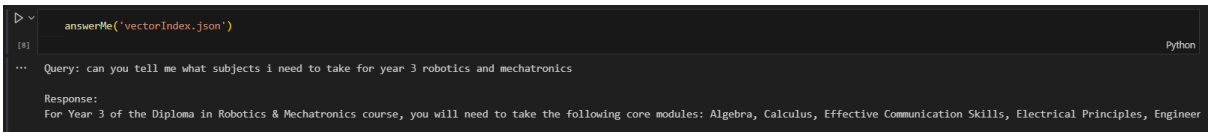
```
def index_documents(folder):
max_input_size = 4096
num_outputs = 512
max_chunk_overlap = 20
chunk_size_limit = 600

prompt_helper = PromptHelper(max_input_size,
num_outputs,
max_chunk_overlap,
chunk_size_limit = chunk_size_limit)

llm_predictor = LLMPredictor(
llm = ChatOpenAI(temperature = 0.7,
model_name = "gpt-3.5-turbo",
max_tokens = num_outputs)
)

documents = SimpleDirectoryReader(folder).load_data()

index = GPTVectorStoreIndex.from_documents(
documents,
llm_predictor = llm_predictor,
prompt_helper = prompt_helper)
index.storage_context.persist(persist_dir=".")    #    save    in    current
directory
```

Here's what the function is doing:

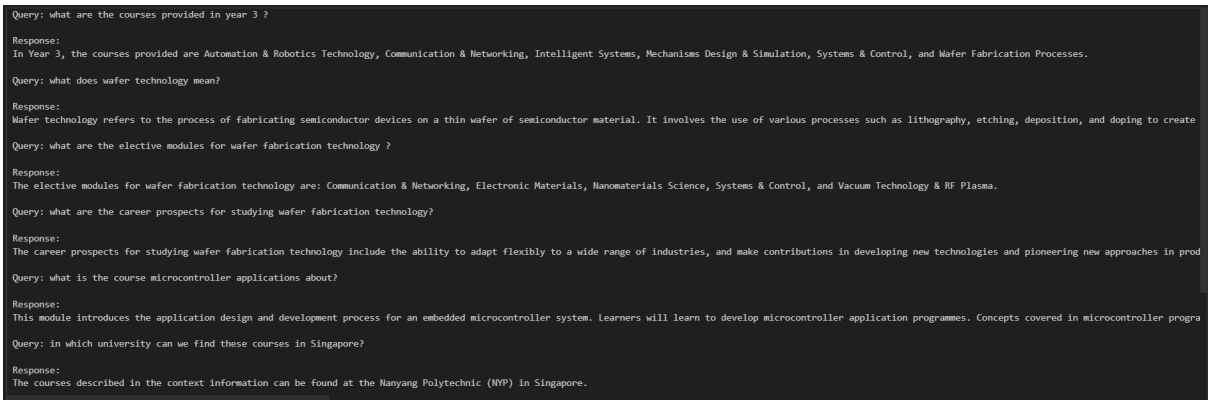
- Once the indexing is done, you persist it to storage using the `storage_context.persist()` function. Persisting the index allows you to query the LLM at a later time without spending time performing the indexing again. By default, the index is saved to a file named **vector_store.json**.

Here is some tests I made by asking the chatbot some questions:





here I used the file .txt, the accuracy of the response is not precise



the code is working but still not very accurate

My data

JSON files

I got my data from two websites:

- NYP Engineering with business:

https://www.nyp.edu.sg/schools/seg/full-time-courses/engineering-with-business.html

- NYP Robotics and Mechatronics:

https://www.nyp.edu.sg/schools/seg/full-time-courses/robotics-and-mechatronics.html

First simple json files

I first simply scrapped these whole websites (one json for each website) with beautifulsoup:

```
url = "https://www.nyp.edu.sg/schools/seg/full-time-courses/robotics-and-mechatronics.html"
response = requests.get(url)
soup = BeautifulSoup(response.content, "html.parser")
```

With this code, it sends a request to get the url using the requests library. It then creates a BeautifulSoup object by parsing the HTML content of the response. This allows us to navigate and extract data from the webpage using BeautifulSoup's methods.
The main problem is that (as said above) the files' size exceeded the maximum number of tokens allowed and while most the file was correctly sorted, a small part was not. Especially the third year of Robotics and Mechatronics which has two elective programmes: Automation & Robotics Technology and Wafer Frabrication Technology: when the website is converted to a text file, each line was written one below the other and with that, the model cannot properly handle the information.

What the website looks like when we pull down the menu of year 3:

Elective Programmes

(Choose one specialisation)

Automation & Robotics Technology

Core Modules

- Automation Systems Design
- Communication & Personal Branding
- Motion Control & Drives
- Semestral Project
- General Studies

Elective Modules (Choose two)

- Communication & Networking
- Intelligent Systems
- Mechanisms Design & Simulation
- Systems & Control
- Wafer Fabrication Processes

Wafer Fabrication Technology

Core Modules

What it looks like once as a text file:

```
Year 3
Elective Programmes
(Choose one specialisation)
Automation & Robotics Technology
Core Modules
Automation Systems Design
Communication & Personal Branding
Motion Control & Drives
Semestral Project
General Studies
Automation Systems Design [60 hours]
This module aims to provide essential concepts and principles t
```

When I tested my code with a part of the file (only year 3), with this question:

```
ans = mychatbot('with the given information only, what are the different specialisation for year 3?')
```

I get this response: 'The specialisations for Year 3 are Automation & Robotics Technology, Communication & Networking, Intelligent Systems, Mechanisms Design & Simulation, Systems & Control, and Wafer Fabrication Processes.'

But when I try:

```
ans = mychatbot('what are the different specialisation for year 3?')
```

This is what I get: 'Year 3 offers students four specialisation options: Automation & Robotics Technology, Wafer Fabrication Technology, Biomedical Engineering, and Mechatronics &

Automation Technology. Each specialisation offers its own sets of core and elective modules.'

We can see here that the model is more accurate (still false because two of the specialisation have been removed since then) with the second question, probably because it has checked its own information instead of the file I provided.

I wanted to see if I could get better results for the first question with a proper json file, where the information would be sorted.

More sophisticated json files

I will only talk about the json file for Robotics and Mechatronics as it was more arduous than the one for Engineering with Business, but they both have the same pattern overall.

Most of the code for Robotics and Mechatronics follows the same scheme, especially for year 1 and year 2 but year 3 was more challenging as it has two elective programmes with core modules and elective modules themselves. I first did not know how to sort the json file or what should be nested first, if I should separate by core and elective modules or by years… After some tests it appears that organizing by years was more efficient.

I created a function called creation_df_years(soup, years) that returns a dataframe for each year containing the name of the module, the number of hours required and its description.
The second function called split_title_description(df_year) splits the dataframe into three parts: title, hours and description that are converted to lists.
I then adjusted these lists by removing or adding some lines following the modules because some lines should be concatenated while some parts should be splitted.

Here is what I did for year 1 (the year 2 behaves the same way):

```
# Year 1
title_tab_1 = split_title_description(df_year_1.iloc[:,0])[0]
hours_tab_1 = split_title_description(df_year_1.iloc[:,0])[1]
description_tab_1 = split_title_description(df_year_1.iloc[:,0])[2]
description_tab_1 = concat_lines(5,6,description_tab_1) # append the note that was below the description
description_tab_1 = concat_lines(len(description_tab_1) - 2, len(description_tab_1), description_tab_1)
```

I splitted and saved the dataframe of the year into three different variables. I had to concatenate two parts of the table description_tab (because some lines have been separated while they should not).

When I print df_year_3, this is what I get:

We can see that the two elective programmes are not distinguishable. To separate automatically the two parts and their core and elective modules, I looked for the indexes of the lines containing ‘core modules’ ([2, 26]) and ‘elective modules’ ([14, 38]).

	Year 3
0	Elective Programmes\n(Choose one specialisation)
1	Automation & Robotics Technology
2	Core Modules
3	Automation Systems Design [60 hours]
4	This module aims to provide essential concepts...
5	Communication & Personal Branding [30 hours]
6	In today’s competitive environment, a strong p...
7	Motion Control & Drives [60 hours]
8	This module equips learners with practical kno...
9	Semestral Project [60 hours]
10	This module develops learners’ abilities in ap...
11	General Studies
12	To provide you an all-rounded education, NYP o...
13	To learn more about the GSMs offered, click here
14	Elective Modules (Choose two)
15	Communication & Networking [60 hours]
16	This module acquaints learners with the fundam...
17	Intelligent Systems [60 hours]
18	This module provides students with essential k...
19	Mechanisms Design & Simulation [60 hours]
20	This module aims to equip learners with knowle...
21	Systems & Control [60 hours]

The end of the first table ‘core modules’ is when begins the table ‘elective modules’ (index of the elective module - 1), I applied this principle for the different tables.

```
index = 0
indexes_core_modules = df_year_3.index[df_year_3['Year 3'].str.contains('Core Modules')] # get the indexes of df_year_3 that
indexes_elective_modules = df_year_3.index[df_year_3['Year 3'].str.contains(r'Elective Modules.*', regex=True, case=False)]

if indexes_core_modules[index] != indexes_core_modules[-1]:
  core_modules_EP_1 = df_year_3[indexes_core_modules[index]:indexes_elective_modules[index]-1]
  elective_modules_EP_1 = df_year_3[indexes_elective_modules[index]:indexes_core_modules[index+1]-1]
  name_EP_1 = df_year_3.iloc[indexes_core_modules[index]-1]
  index=+1
core_modules_EP_2 = df_year_3[indexes_core_modules[index]:indexes_elective_modules[index]-1]
elective_modules_EP_2 = df_year_3[indexes_elective_modules[index]:]
name_EP_2 = df_year_3.iloc[indexes_core_modules[index]-1]
```

As the variables here are still pandas objects, I had to convert the names into a string to retrieve them.

```
# get the names of the elective programmes
name_EP_1 = ''.join(name_EP_1)
name_EP_2 = ''.join(name_EP_2)
```

Finally, I separated each modules tables into the three new variables (title, hours, description).

```
# elective programme 1 : automation & robotics technology core module and elective module
title_tab_3_EP_1_CM = split_title_description(core_modules_EP_1.iloc[1:,0])[0]
hours_tab_3_EP_1_CM = split_title_description(core_modules_EP_1.iloc[1:,0])[1]
description_tab_3_EP_1_CM = split_title_description(core_modules_EP_1.iloc[1:,0])[2]

title_tab_3_EP_1_EM = split_title_description(elective_modules_EP_1.iloc[1:,0])[0]
hours_tab_3_EP_1_EM = split_title_description(elective_modules_EP_1.iloc[1:,0])[1]
description_tab_3_EP_1_EM = split_title_description(elective_modules_EP_1.iloc[1:,0])[2]

# elective programme 2 : Wafer Fabrication Technology core module and elective module
title_tab_3_EP_2_CM = split_title_description(core_modules_EP_2.iloc[1:,0])[0]
hours_tab_3_EP_2_CM = split_title_description(core_modules_EP_2.iloc[1:,0])[1]
description_tab_3_EP_2_CM = split_title_description(core_modules_EP_2.iloc[1:,0])[2]

title_tab_3_EP_2_EM = split_title_description(elective_modules_EP_2.iloc[1:,0])[0]
hours_tab_3_EP_2_EM = split_title_description(elective_modules_EP_2.iloc[1:,0])[1]
description_tab_3_EP_2_EM = split_title_description(elective_modules_EP_2.iloc[1:,0])[2]
```

I extracted other relevant information from the website such as the contact information of the course coordinators and manager, or the description of the career prospects.

I insert all the information into my json object, here is an extract:

The json object is then saved into a new json file named ‘R&M.json’.

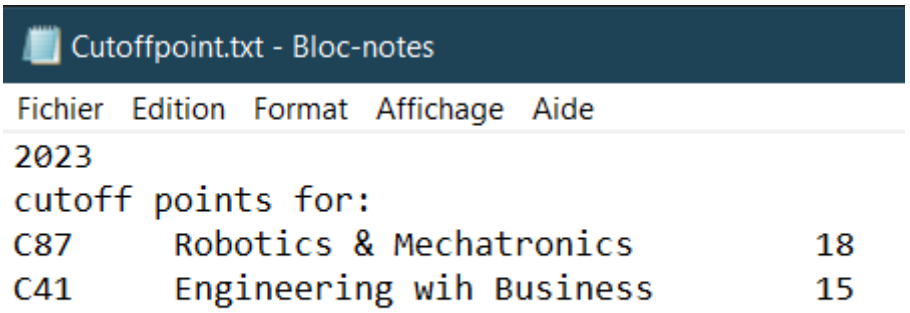
```
data = {
    "fullTimeCourses": [
        {
            "name": full_time_course_name,
            "about the course": info,
            "years": []
        }
    ],
    "career Prospects": description_career_prospects,
    "further Studies": description_further_studies,
    "contact Us": {
        "course Manager": {
            "name": contact[1],
            "telephone Number": contact[2],
            "email": contact[3]
        },
        "course Coordinator": [
            {
                "name": coord[1],
                "telephone Number": coord[2],
                "email": coord[3]
            },
            {
                "name": coord[5],
                "telephone Number": coord[6],
                "email": coord[7]
            }
```

```
with open('R&M.json', 'w') as json_file:
    json.dump(data, json_file, indent=1)
```

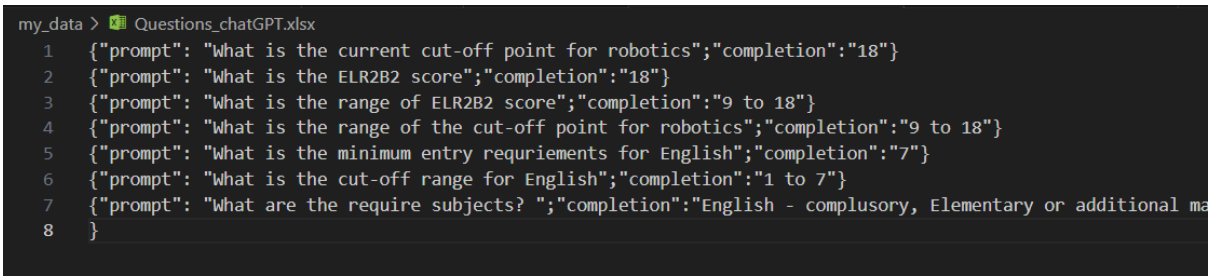
I did the same for Engineering with Business, and saved it into a json file named “EWB.json’.

XLSX files and TEXT file

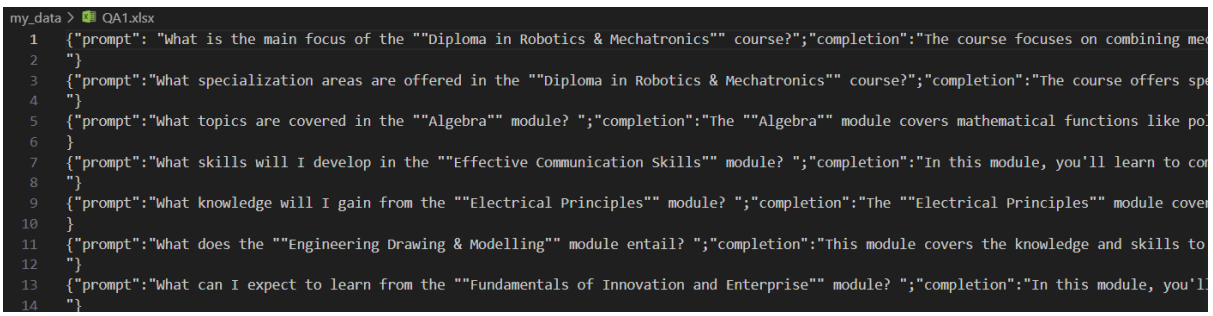
I then added two xlsx files, and one text file. Two were given by my supervisor, to test if the data is actually taken into account or not (which globally is) and the third file was created by me, to enhance the accuracy of the response for several questions that were first not well answered by the chatbot.



First text file



First xlsx file



Extract of the second xlsx file

This type of files can enhance the chatbot rapidly but their are two main problems. The first one is that if we ask a question that is not inside the file, the robot cas loose some accuracy.

To fix that, we should put thousands of prompt/completion but here is the second problem. It would be not optimum to write manually all the cases it can encountered.

Speech-to-text and text-to-speech

This part of the code was to enhance the chatbot in order to recreate a more human-like chatbot.

Speech-to-text

The speech-to-text code is used when the user wants to ask a question orally.
I used the SpeechRecognition library.

```
import speech_recognition as sr
```

This is a Python library that can integrate speech recognition functionality such as handling audio inputs and saving them, sending them to an API or retrieving and also translating the recognized text.

I created a function called “speech-to-text(timeout=10)” which will recognize the speech input within 10 seconds after running the function (the duration of the parameter can be adapted).
I initialized an instance of the Recognizer class from the SpeechRecognition library, which handles speech recognition operations.

```
# Initialize recognizer class (for recognizing the speech)
recognizer = sr.Recognizer()
```

```
for i in range(3): # will retry 3 times
    # Record Audio
    with sr.Microphone() as source:
        print("Say something!")
        # Optionally, can add this line to adjust for ambient noise
        # recognizer.adjust_for_ambient_noise(source) #listen for 1 seco
        # Listening to the microphone
        audio = recognizer.listen(source, timeout=timeout)
        try:
            # Store the result in a variable
            text = recognizer.recognize_google(audio, language='en-US')
            return text # if speech recognition was successful, return t
        except sr.UnknownValueError:
            print("Sorry, I did not understand, can you repeat please?")
        except sr.RequestError as e:
            print("Request Failed; {0}".format(e))

print("Sorry, I still did not understand what you said.")
return None
```

I then set up a loop that will try three times to recognize the speech, is case it does not understand what the user said. The microphone of the computer is used thanks to the Microphone class from the library to detect and record the audio, which is passed to the the “recognize_google()” function. This function is used to understand the the speech and convert it into a text stored in a variable “text”.
The function is created to properly handle errors, otherwise, it would crash each time there is an error.

https://www.activestate.com/blog/how-to-build-a-digital-virtual-assistant-in-python/ good website for speech to text

Text-to-speech

This part is used to convert a text input to spoken words. The library used here is pyttx3.

```
import pyttsx3
```

This library allows us to customize the voice output, such as the language of the voice or its speed.

```
def text_to_speech(mytext):
    # The text that you want to convert to audio
    engine = pyttsx3.init()
    engine.setProperty('voice', 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Voices\Tokens\TTS_MS_EN-US_ZIRA_11.0')
    engine.setProperty("rate", 187) # Integer speech rate in words per minute. Defaults to 200 word per minute.
    engine.say(mytext)

    return engine.runAndWait()
```

I created a function called text_to_speech(mytext), the parameter is the string that will be spoken. The engine that is initialized is responsible for generating the speech audio.
I then customize some parameters. As my laptop language is in French, the default language used there is in French, which means that even if the sentence is in english, the voice would say it with a french accent, so I had to manually set the voice to an english voice.
The speech rate is also changed and is set to slightly slow it down.
The “engine.say(mytext)” line tells the engine to convert the specified text into speech. Finally, “engine.runAndWait()” is called to run the text-to-speech engine and wait for the speech to complete. It ensures that the program doesn't proceed until the speech is finished.

Fine tuning

Try to fine tune chatbot 2:
https://betterprogramming.pub/openai-api-fine-tuned-models-vs-chat-completion-a-case-study-e3774fadc8c7 check this website, it is very complete and useful if you want to fine tune !
https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset this one from openai to prepare your dataset from the terminal (tried just below but this does still not work)

pip install --upgrade openai

export OPENAI_API_KEY="<OPENAI_API_KEY>" but set instead of export bc we are on windows and not on Mac os.

pip install openpyxl if I want to fine tune through a xlsx file
https://www.youtube.com/watch?v=2SW6_3p37pE tuto to text to speech but 1h (dont 40min de son site internet et 20min de vrai truc utile, mais le son prend le text dans le cadre)https://www.codingem.com/python-text-to-speech/ text to speech simple mais enregistre dans un fichier mp3 avant de pouvoir le lire donc pas ouf

https://www.youtube.com/watch?v=_RTN8CWFUsc&t=357s dont know what this video is about but Im halfway so maybe should finish it?

Graphical user interface with Streamlit

Streamlit is an open-source Python library that allows developers to create web applications for data science and machine learning projects easily. It is designed to help developpers to convert data scripts into interactive web applications that can be access through a web browser. Streamlit's user-friendly nature make it an easily accessible and approachable framework for beginners.

I created a new python file called app.py that includes almost all the code from the previous file but in order to avoid problems between the front-end and the back-end, I change some parts of the code, for example the three-times trial from the speech-to-text function.

However, I added a voice trigger function that will detect a specific phrase before actually process the speech, so the speech detechtion can then be run in the background non-stop. Here is my code:

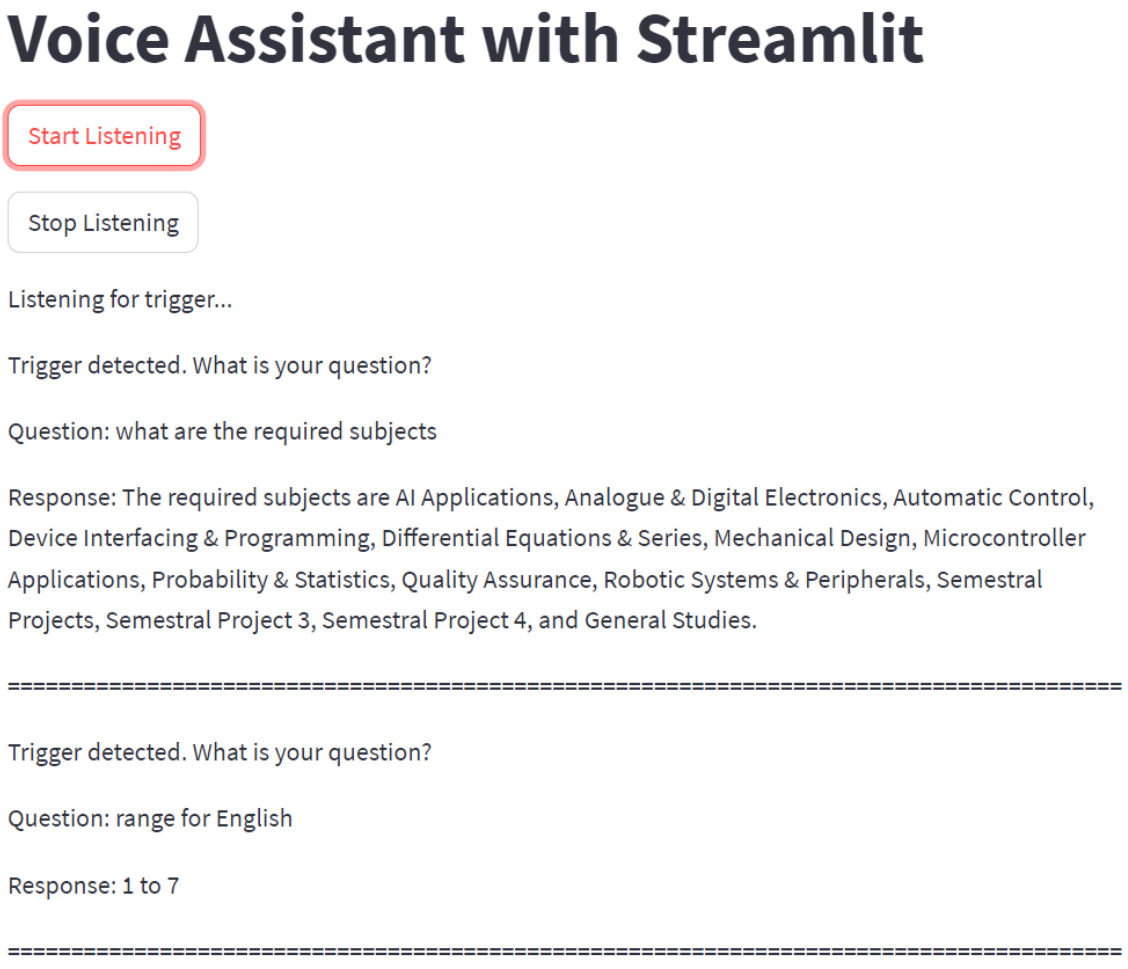
```python
# Voice trigger function
def listen_for_trigger():
    recognizer = sr.Recognizer()
    mic = sr.Microphone()
    trigger_phrase=["hey assistant", "thank you"]
    print("Listening for trigger...")
    with mic as source:
        while True:
            audio = recognizer.listen(source)
            try:
                trigger = recognizer.recognize_google(audio).lower()
                if trigger_phrase[0] in trigger:
                    print("Trigger detected.")
                    return trigger_phrase[0]
                if trigger_phrase[1] in trigger:
                    print("You're welcome, bye!")
                    return trigger_phrase[1]
            except sr.UnknownValueError:
                pass
            except sr.RequestError:
                print("Sorry, speech recognition service is currently unavailable.")
                break
```

There a two triggers “hey assistant” and “thank you”. When the recognizer will identify one of them, the latter will be returned.

Here is the main function:

```python
def main():
    trigger_phrase = ["hey assistant", "thank you"]
    st.title("Voice Assistant with Streamlit")
    start_listening = st.button("Start Listening")
    stop_listening = st.button("Stop Listening")

    def response():
        st.write(f"Question: {prompt}")
        response = answerMe(prompt)
        st.write(f"Response: {response}")
        text_to_speech(response)
        return

    if start_listening:
        st.write("Listening for trigger...")
        while True:
            if listen_for_trigger() == trigger_phrase[0]:
                text_to_speech("How can I help you?")
                st.write("Trigger detected. What is your question?")
                prompt = speech_to_text(timeout=30)
                response()
            if listen_for_trigger() == trigger_phrase[1]:
                st.write("You're welcome!")
                text_to_speech("You're welcome!")
                break
            if stop_listening:
                break
```

The website is built so there are only the title and the two buttons “start listening



References

[1] https://www.statista.com/statistics/871513/worldwide-data-created/