NYU OGS Chatbot

Nuo Lei
nl2581@nyu.edu

Tianyu Du
td2418@nyu.edu

Yong Zhao
yz6433@nyu.edu

Chuan Shi
cs5526@nyu.edu

Luyi Ge
lg3846@nyu.edu

Abstract

This paper focused on the NYU OGS Chatbot, which is designed to assist users in accessing information related to international affairs at New York University. The chatbot offers quick and accurate responses to user queries. Optimization strategies, including parallel processing methods, enhance efficiency in data collection and analysis. The chatbot's scalability and potential for broader deployment make it a valuable tool for the NYU community.

1 Background

NYU OGS (Office of Global Service) is the department dealing with international affairs for the entire university. Especially, OGS handles immigration matters for all students, faculty, and staff seeking immigration and visa support when traveling to NYU locations in the US and around the world.[1] Following the rules of the government is crucial for international students and staff to keep their lawful status. Nevertheless, the information listed on the OGS website is kind of overwhelming and sometimes can be confusing in details for students and staff to figure out. This encourages us to build a chatbot to help the users quickly identify the needed information. Students and staff could get immediate and precise information in interaction with the chatbot before they seek help from a real officer from OGS. The chatbot would increase efficiency and would benefit the whole NYU community.

2 Problem Setup

The work to build a chatbot can be separated into three main parts.
1. Collect all the information needed from the OGS website by web scraper.
2. Using the information we collected to build a chatbot.
3. Generate website for our chatbot so that users can interact with the chatbot.

As in this project, we select several webpages with FAQs to ensure the quality of the chatbot. The typical webpage can refer to the following link: FAQ. We identify the expandable singleton in the webpages and collect all the FAQs as our dataset. We do statistical analysis on our collected dataset. Then, we use the dataset to build on the LLM through retreival-augmented generation (RAG) using LangChain to get a chatbot. Finally, we apply Gradio to generate the webpage for our chatbot.

After that, we use different methods to optimize our code to improve the efficiency for future works.

Preprint. Under review.

3 Methodology

Here below is the pseudo code for our project.

Algorithm 1: Chatbot Pseudo Code

Data: FAQs from OGS websites
Result: Webpage of Chatbot that can interact with users to answer questions regarding NYU international affairs.

Step 1: Do web scraper and collecting data
for *each website URL listed* **do**
    Extract question and answer pairs in expandable singleton from the website;
Step 2: Get statistical analysis of the FAQ dataset
Step 3: Train the chatbot using LangChain
Template prompt: <s>[INST] Instruction: You're an expert in international student and immigration services at NYU.
Use only the chat history and the following information $\{context\}$
to answer in a helpful manner to the question. If you don't know the answer - say that you don't know. Keep your replies short, compassionate and informative.
$\{chat\_history\}$
Question: question [/INST]
Step 4: Generate webpage for interaction using Gradio.

4 Initial Code

We create separate py files for each part of the work. The initial base code follows the problem setup above to achieve our goals on the chatbot. In the web scraper stage, we apply **Beautifulsoup** to identify the FAQs from the webpage. In the statistical analysis stage, we calculate the distribution of word counts in sentences and do necessary data clearning for the following work. Then, we apply **LangChain** to build the chatbot, and use **Gradio** to generate the webpage for our chatbot. During designing our codes, we keep in mind on its scalability. We hope the new data could be easily added and collected in order to enlarge the knowledge of our chatbot.

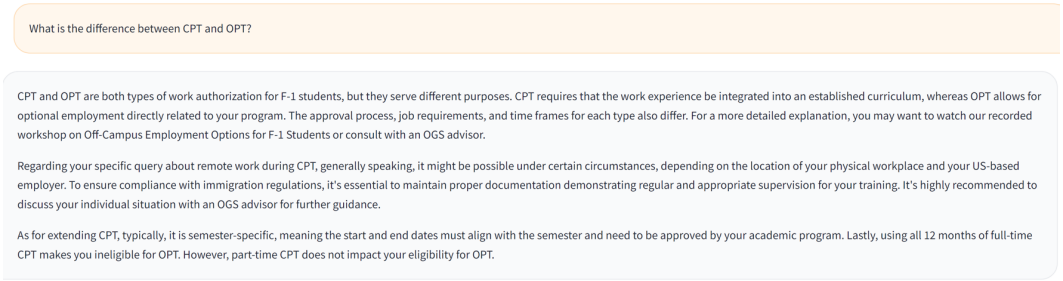

Figure 1: Demo for chatbot webpage

Figure 2: Demo for chatbot webpage

What worth to notice here is the demo figure of our model. From the figures above, we can see that the chatbot could answer relevant questions correctly, while also identifying irrelevant questions.

5 Optimization

We focused our optimization on the web scraper stage and statistical analysis stage. In the optimization process, we employ **time** and **line_profiler** to analysis the run time. For the web scraper part, we noticed that we can deal with each webpage separately. So we apply **multiprocessing** and **threading** to do parallel operation. For the statistical analysis part, we apply **concurrent** and **Numba** to speed up the processing time.

5.1 Web Scraper Optimization

In this section we apply optimization on our Web Scraper code. First, we used **line_profiler** on our base code and discovered that the request to visit the URL requires most of the time processing. Therefore, we decided to use parallel processing methods to optimize our code. We tried **multiprocessing** and use **pool.map** to allocate the task and aggregating the results.

We also tried **threads**. Here, we create threads for each URL, using **thread.start** to start each thread task, and use **thread.join** to ensures that the main thread waits for each thread to complete its execution before proceeding further. Finally, we aggregate the results by collecting the data obtained from different threads and combining them into a single output.

5.2 Statistical Analysis Optimization

In this section, we apply optimization on our Statistical Analysis code. We first tried **concurrent** on our code. Here, we use **concurrent.futures.ThreadPoolExecutor** to manage a pool of worker threads. This executor allows concurrent execution of tasks in multiple threads. Inside the context manager of the **ThreadPoolExecutor**, tasks are submitted for processing FAQ items using the **map** method. Another **ThreadPoolExecutor** context manager is used to calculate the word count for each sentence in parallel.

We also tried **Numba** to optimize our code. By decorating the **word_count** function with **@nb.jit**, it instructs Numba to compile the function for better performance. When the **word_count** function is called, **Numba** compiles it to machine code, potentially providing significant speedup compared to the standard Python implementation, especially for numeric and array-based computations.

6 Results

Metric	Baseline	Multiprocessing	Multithreading
Wall Time	3.102s	0.634s	0.735s
CPU Time	1.040s	0.047s	0.880s

Table 1: Comparison of wall time and CPU time for different methods in web scraper stage

Metric	Baseline	Concurrent	Numba
Wall Time	0.011s	0.047s	5.091s
CPU Time	0.010s	0.045s	4.529s

Table 2: Comparison of wall time and CPU time for different methods in statistical analysis stage

The above tables are the wall time and cpu time in different stagees and for different methods. We can see that most of the optimization has significant impact on the processing time. In the web scraper stage, Both multiprocessing and multithreading significantly reduce both wall time and CPU time compared to the baseline. We can conclude that it is efficient to use parallel processing methods to deal with requesting URL tasks.

Unlike the web scraper stage, the baseline method in the statistical analysis stage has remarkably lower wall time and CPU time.Both **concurrent** and **Numba** optimization methods show an increase in both wall time and CPU time compared to the baseline, with **Numba** exhibiting significantly

higher values. This might because our dataset is too small so that the optimization method is inferior by its higher starting time.

7 Conclusion

In this project, we successfully built a chatbot that is able to answer related OGS questions under good interaction with users. We used the optimization strategy learned in the class to speed up the Web Scraper and Statistical Analysis processes. The result reflects the effectiveness of parallel processing during the request visiting to URL.

Optimization plays a key role in our project because the chatbot is highly scalable. Unlike finetuned models, our retrieval-augmented generation (RAG) model benefit directly from a larger knowledge base and new information, with little to none additional work such as retraining/finetuning. Therefore, the more efficient we can add information to the database, the more knowledgeable the chatbot would be. By utilizing optimization techniques, we make sure that the data scraping step will not become a bottleneck.

8 Further Work

There are several things we could do in our further work. First, we can enlarge our dataset for the chatbot to contain more information, including non-FAQ content on the OGS website as well as other NYU sites. This could enable our chatbot to answer more questions related to NYU. Second, we could consider the specific metrics used to evaluate the performance of the chatbot as quantitative analysis. Third, we could refine our website or even develop an app and make it public for the users. We would offer long-term maintenance and support of the chatbot, including ongoing updates, bug fixes, and adaptation to changes in user needs. The optimization we learned in the course would definitely help in this process once the project turns out to be large.

9 References

[1] OGS webpage `https://www.nyu.edu/students/student-information-and-resources/student-visa-and-immigration.html`