CS 242 Project Report Part B

Group Number: 20

Group Members: Yifan Yu, Tianyang Li,

Qian Xiang, Xinle Chen, Jiang Zhu

Date: Mar 17, 2023

## 1. Collaboration Details: Description of the contribution of each team member.

* Yifan Yu collected data into JSON by using Tweepy, designed frontend and backend architecture, wrote code, made architecture flowcharts, wrote shell scripts, and wrote reports.
* Xinle Chen initialized the Jupyter environment, wrote the initial and standard Lucene indexing program, implemented the MultiFieldQueryParser hashtag solution, integrated Lucene engine into web application and made comparison conclusions for Lucene and BERT with concrete experimental results in the project.
* Qian Xiang, Tianyang Li, and Jiang Zhu together tested and implemented additional indexing features, implemented BERT engine and debugged issues with application, collected, plotted and analyzed test results for both parts and wrote the rest of the report.

## 2. Overview of system, including (but not limited to):

### Architecture

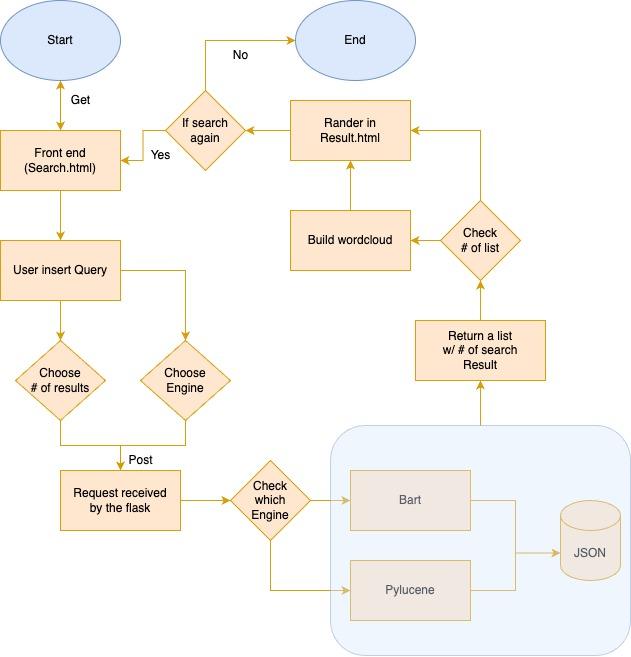


Fig 1. General workflow chart of the whole system

In this part, we mainly have three components.

* search.html: This is the user interface component of the system. It provides a search box where the user can enter a query, select the search engine to use (either PyLucene or BERT) and specify the top-k number return. When the user submits the form, the query and search engine selection are sent to the app.py backend for processing.
* app.py: This is the backend component of the system. It receives the user's query and search engine selection from search.html and processes the query using the selected search engine. If the user selects PyLucene, the system retrieves the top k documents that match the query using an existing Lucene index (which we have generated before in phase 1). If the user selects BERT, the system generates a summary of the query and retrieves the top k documents that are most similar to the summary using cosine similarity. The search results are then passed to results.html for display.
* results.html: This is the component that displays the search results to the user. It shows the top k documents that match the user's query. It also displays a word cloud of the search results, which provides a visual representation of the most common words in the documents. If there are no search results, the word cloud is not displayed.

### Details of how BERT was used (model choice, indexing schema).

We chose distilbert-base-nli-stsb-mean-tokens as our model instead of all-distilroberta-v1. Because the distilbert-base-nli-stsb-mean-tokens model is a good choice for tasks related to natural language inference and semantic similarity, as it was trained specifically for these tasks. Since the tweets we crawled are all about job or interview, using such a model for indexing and retrieving is able to perform latent semantic similarity comparisons between different pieces of text. Compared to the all-distilroberta-v1 model, which is a larger and more powerful language model and suitable for the dataset contains a wide range of topics and concepts beyond job/interview-related tweets, distilbert-base-nli-stsb-mean-tokens is faster and more memory-efficient while still maintaining strong performance on semantic similarity tasks. This makes it a suitable choice for indexing and retrieving large volumes of tweets.

When indexing, we use dense embeddings generated by the above mentioned pre-trained neural network model, the embeddings are then processed using mean pooling to generate a single vector encoding for each sentence. Then we tokenized each tweet using the tokenizer and encoded them as input\_ids and attention\_mask tensors. These tensors are stored in a dictionary called "tokens". Then we use the model to compute the embeddings of the tokens and the embeddings are stored in a PyTorch tensor called "last\_hidden\_state". We performed mean pooling on the embeddings to produce a single vector for each tweet, which represents its sentence embedding. Finally, we used FAISS to create an index of the sentence embeddings, and saved the resulting index to disk which can be used to perform nearest neighbor searches later.

### Explain how you use the BERT index to do the ranking

We used the BERT index to find the most similar tweets to a user’s query, based on their semantic meaning. Here’s the process:

1. The user’s query is passed through the BERT model to obtain a dense embedding representation of the query.
2. Load the BERT index from the index file, which contains a set of dense embeddings representing a pre-processed set of tweets.
3. Compare the query embedding to each document embedding in the index using cosine similarity.
4. The most similar tweets (with the highest cosine similarity scores) are returned as the search results, with the number of tweets determined by the "top\_k" parameter specified by the user.
5. Finally the search results are rendered as an HTML page using the Flask web framework.

### Explain how you use the Lucene index to return results. E.g., what options did you use with the QueryParser? How were multiple fields in the Lucene index used with the QueryParser?

The "QueryParser" is used to create a query for each field, using "StandardAnalyzer" to tokenize the query string and perform other text processing tasks like removing stop words. Then the resulting queries parsed by "QueryParser" are added to the "BooleanQuery" builder using "BooleanClause.Occur.SHOULD", indicating that tweets that match either of the sub-queries should be included in the results.

We used two more fields in addition to "Context" which are "Hashtags" and "UserID". We gave hashtags a boost of 2 and username a boost of 2.5 because we thought during a query, when the hashtag of a tweet matches the query, that tweet should have a greater chance of containing the relevant information that the user wants. Moreover, as we explained earlier, username is even more important than hashtags in a lot of scenarios where the query is a specific name other than a random word. Therefore, we set the username to have a boost of 2.5.

#### 

### Report the run time of the BERT index construction. This should be comparable with the report on the run time of the Lucene index construction from report A. There should also be a discussion explaining the differences between the run times.

Plots of Runtime Analysis:

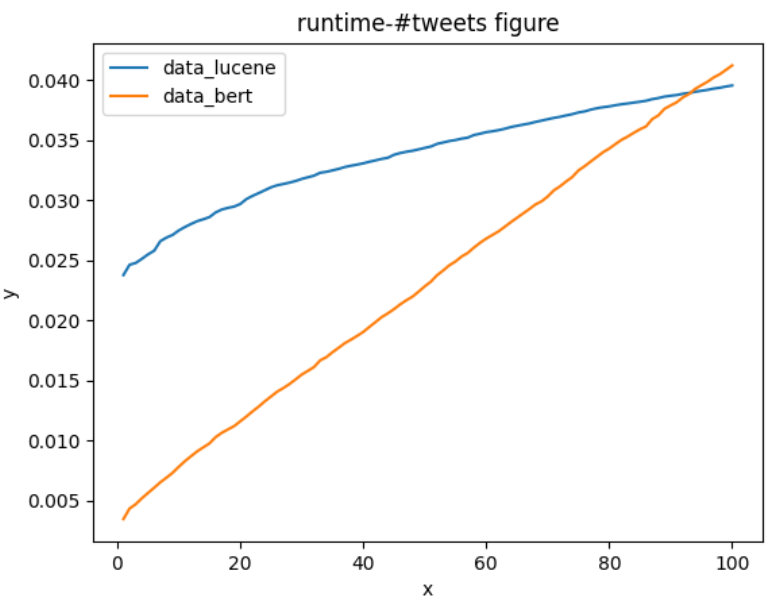
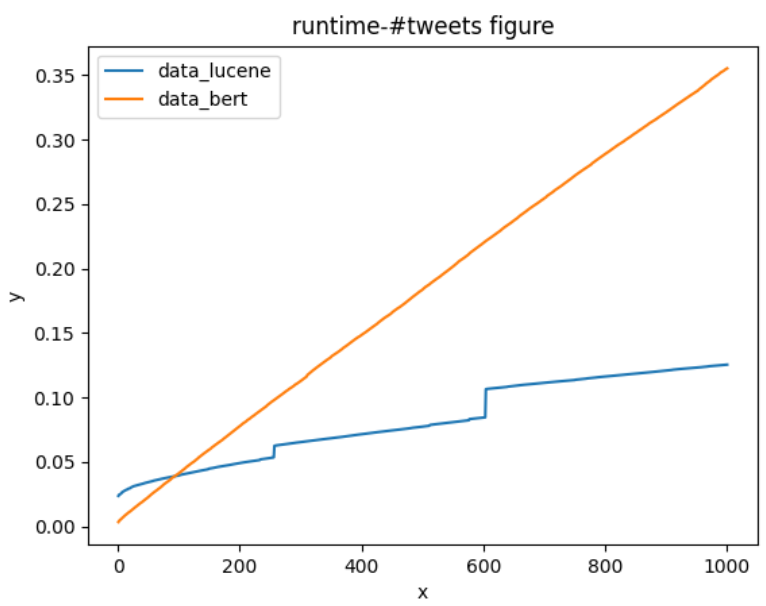
 

Fig 1. Indexing 100 tweets from a 500mb file Fig 2. Indexing 10000 tweets from a 500mb file

### 

Fig 3. Indexing 1 million tweets from 500mb Fig 4. Indexing all 1.43 million tweets from 500mb

These four figures are the graphs plotted to analyze the runtime between the Lucene and the BERT index creation process. Before the first index has been created, there is a time gap after the program starts. This is due to the fact that our program takes about that amount of time to read the entire 500mb of data, which is pretty enormous. Therefore we started timing after the data was read, only to compare the run time between the Lucene and BERT index construction process.

As we can see from these figures, Lucene runs longer than BERT runs before 90 indexing , and the gap narrows. With indexing at the intersection of 90 indexings, BERT starts to beat Lucene and the gap widens. BERT has a time advantage when dealing with small amounts of data, and it spends more time than Lucene indexing with large amounts of data.

Moreover, BERT's figure is obviously smoother and more linear than Lucene's. As can be clearly seen from Figure 2, Lucene's figure has a stair-shaped pattern. For about every 250-300 indexing, there is a time gap of about 0.02 seconds. In Figure 1 and 3, we can still notice a slight and subtle stair-shaped pattern in the figure, but this feature is almost invisible in Figure 4.

For 100 indexes, the index creation process of Lucene and the BERT took about 0.0395 seconds and 0.0412 respectively. When it comes to 1000 indexes, they took about 0.1255 seconds and 0.3552 respectively. For 10000 indexes, it took about 0.3563 seconds and 3.7564 seconds respectively. For the entire 1.43 million indexes, it took about 40 seconds and 911 seconds respectively.

| # of index | Time(s) | | Indexes per second | |
| --- | --- | --- | --- | --- |
|  | Lucene | BERT | Lucene | BERT |
| 100 | 0.0395 | 0.0412 | 2,531 | 2,427 |
| 1000 | 0.1255 | 0.3552 | 7,968 | 2,815 |
| 10000 | 0.3563 | 3.7564 | 28,066 | 2,662 |
| ~1.43 million | 40 | 911 | 35,750 | 1,569 |

Table 1. Runtime and Performance of Lucene VS BERT

From the table above where the indexes per second for each trial is calculated, we can see that when indexes exceed 1000, the efficiency of Lucene is much higher than BERT. Lucene and BERT are almost equally efficient when only 100 indexes are processed.

### Compare the quality of the rankings of Lucene and BERT. Show examples where one is doing better than the other and explain.

In terms of ranking quality, BERT has been shown to outperform Lucene in several tasks, especially in understanding the context and intent of the query. BERT can handle complex queries that involve multiple concepts or entities and can understand the nuances of language, such as sarcasm or ambiguity. For example, in our project if we use the query “iHerb” which is an unfamous company name to retrieve the result by using Lucene, the retrieve result will be empty as no one of our 500MB tweets contains this keyword. However, if we use BERT with the same query “iHerb” to retrieve the results, it does return the results and the results tweets contain the keyword “job” which is our crawling keyword. The BERT can understand “iHerb” is a company name and a company name is relevant to the keyword “company” and the keyword “company” is relevant to “job”. So we think the result is relevant to the query. This example embodies the power of BERT of understanding the context and intent of the query which Lucene does not have.

Lucene has some advantages over BERT in certain scenarios, such as queries with only a few keywords or queries where the exact matching of terms is important. The main characteristic of these scenarios is simplicity, which is also the characteristic of Lucene. Lucene is simple, however it is efficient, effective and space friendly. For example, in our project if we use the same size data to build the index, Lucene will be much faster than BERT. In terms of space, we use 500MB tweets data to build the Lucene index and the size of the index is only 4KB. However, when we use a small subset of the tweets data to build the BERT index, the size of the index is 6MB. That is a significant increase.

In conclusion, BERT is fit for a complex and flexible query system and has very high hardware resource requirements. So Google is a very proper representation. Lucene is fit for some internal simple search system of organizations and has much lower costs of resources.

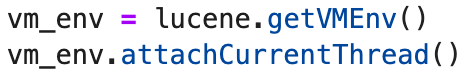
## 3. Limitations of system

1. For the data we stored in JSON during the last phase, we didn’t think of doing much data cleaning. Therefore, the search results now will show the same text from different Twitter users when a query is searched multiple times and this may lead to unattractiveness to our app users.
2. It’s difficult to see how many requests can be handled by the BERT or Pylucene at the same time.
3. The indexing process of BERT takes a significant amount of time and that results in an extremely long waiting time every time we try to get the results of searching using BERT. Therefore, instead of indexing every time the user searches using BERT, we did the indexing of the entire tweets file ahead of time and saved it so that now when a search is needed, our app can skip this process and return the results much faster. However, this leads to the limitation that our BERT search engine is only efficient for existing tweets data. When new data comes in, the indexing preparation is needed.
4. We specified the port number for our application to run to be 8888. However, when that port is being used, we cannot run the application anymore. This happens when we try to open Jupyter and the application at the same time.

## 4. Obstacles and solutions

We implemented a feature on the web application that displays a word-cloud on the right side of the results webpage. However, when the search engine could not find any relevant result, the word-cloud still shows some words that are not related to what we searched for, which is actually the results it generated previously. That is because we didn't clear the cache of the last search results, thus leading to this confusing situation. To solve this problem, we made changes to the code of the results webpage to when there is no result found, instead of displaying the most recent word-cloud, we display a presetted image showing “no results found”. A screenshot of our app in this situation could be found in the later section.

The second hard problem we encountered was that when we were using PyLucene as the search engine for our web app, we could only search for once. When we searched for the second time, the app crashed. The main reason we can not search for the second time is that we init lucene VM in the retrieve function so each time we search by using lucene will repeatedly init the lucene VM. So the app will crash because of the “JVM is already running” exception. Then we move the Lucene initVM function to global to prevent duplicate initialization. However a new issue occurs. Seems the Lucene will disconnect the thread after the searching engine finishes its work. So the same issue when we search the second time, the app will crash still exists, which is caused by the disconnected thread. We think the reason why Lucene disconnects the thread after the search engine finishes searching is that Lucene wants to save the thread resources. It is fair to disconnect the thread after Lucene finishes its work. After finding the root cause of this issue, we activate the Lucene searching thread each time when users do Lucene searching. The fixing code is shown below.



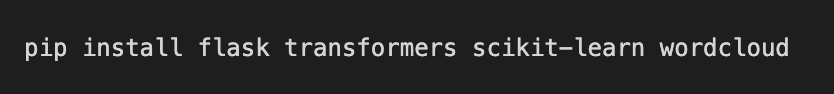
The third hard problem we encountered is that when we integrate Lucene into our web app, the web app gets stuck at some point of the running process and it can not totally be started. But the web app works properly when we remove the Lucene logic. After deep contemplation, we think we should not include the whole Lucene logic in the web app. The Lucene logic can be separated into two parts which are indexing and retrieving. Actually in the web app we just need the retrieving logic which is relatively simple to the indexing logic. So we decided to just include the retrieving logic in the web app. There are two main advantages of the separation. The first one is that the web app will be much faster as it just cares about retrieving and uses the index which is already built by the separated index process. The second advantage which is also the solution of the app running issue is that we can clean out a lot of indexing code so that the code in the web app is much more clean and concise. By using the clean code in the web app, the web app can run appropriately when we integrate the Lucene into it.

Moreover, another obstacle we had was that our BERT search engine was extremely slow and it took about 30 seconds to return the search results. The solution is similarly to the third problem. We separated the search and index process. The index process actually takes much more time than the search. So we first built the index and then the web app just uses the indexing which is already built by the index process to retrieve the result and return it to the front-end. After the separation, Bert returns the result in just 2 seconds, which is a significant improvement.

## 5. Instructions on how to deploy the system

### Ideally, you should include an indexer.bat (Windows) or indexer.sh (Unix/Linux) executable file that takes as input all necessary parameters e.g. Example: [user@server] ./indexer.sh <input-dir> <output-dir>

Before starting, we need to make sure that the environment is ok. The main library we are using is the Pytorch and flask. Pylucene is already on the school server. Then we have to install using the following command:

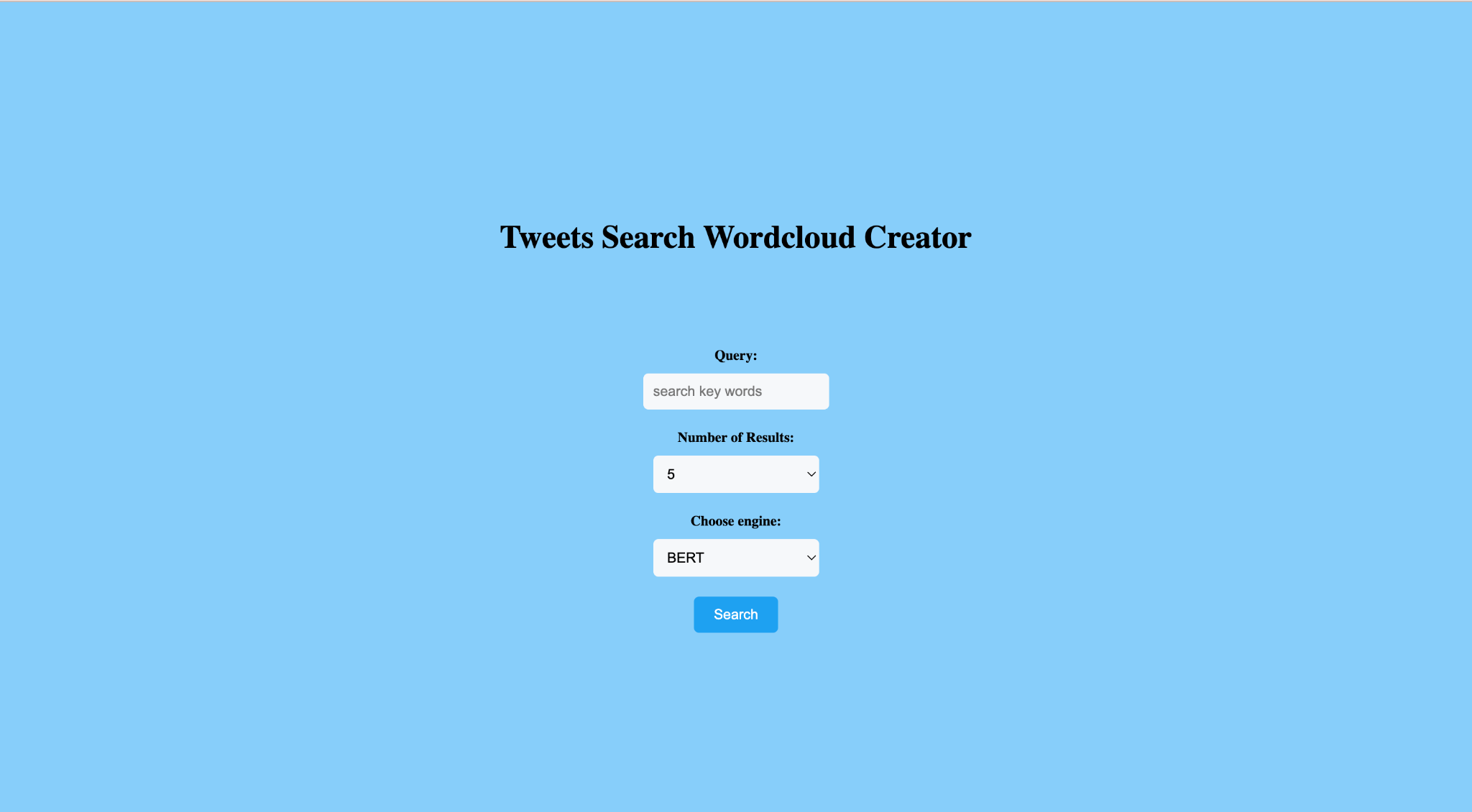


After that, we just need to run the start.sh script as we already have built the Lucene and BERT index, and upload them with the source code. If the index files do not work properly, we can run “python3 my\_lucene.py” to build the Lucene index and run “python3 my\_bert.py” to build the BERT index, and then run start.sh to start the web application.

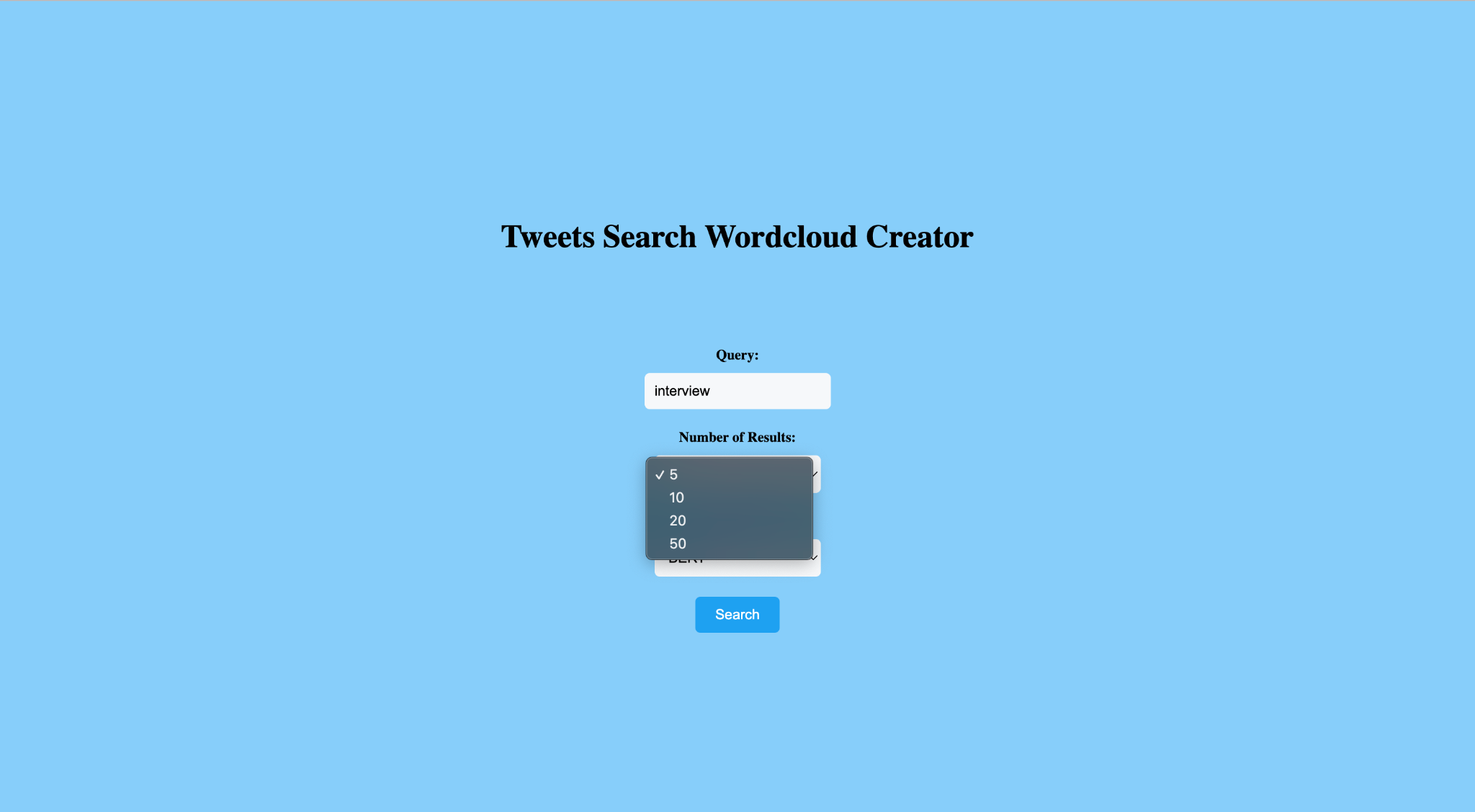
### Create a web-application (e.g. Web Archive) that can be deployed to a web server like Tomcat.

We assume the web server will be deployed on the school server, thus we set the port 8888 by default. And after waiting for the flask to start, the user can access the web.

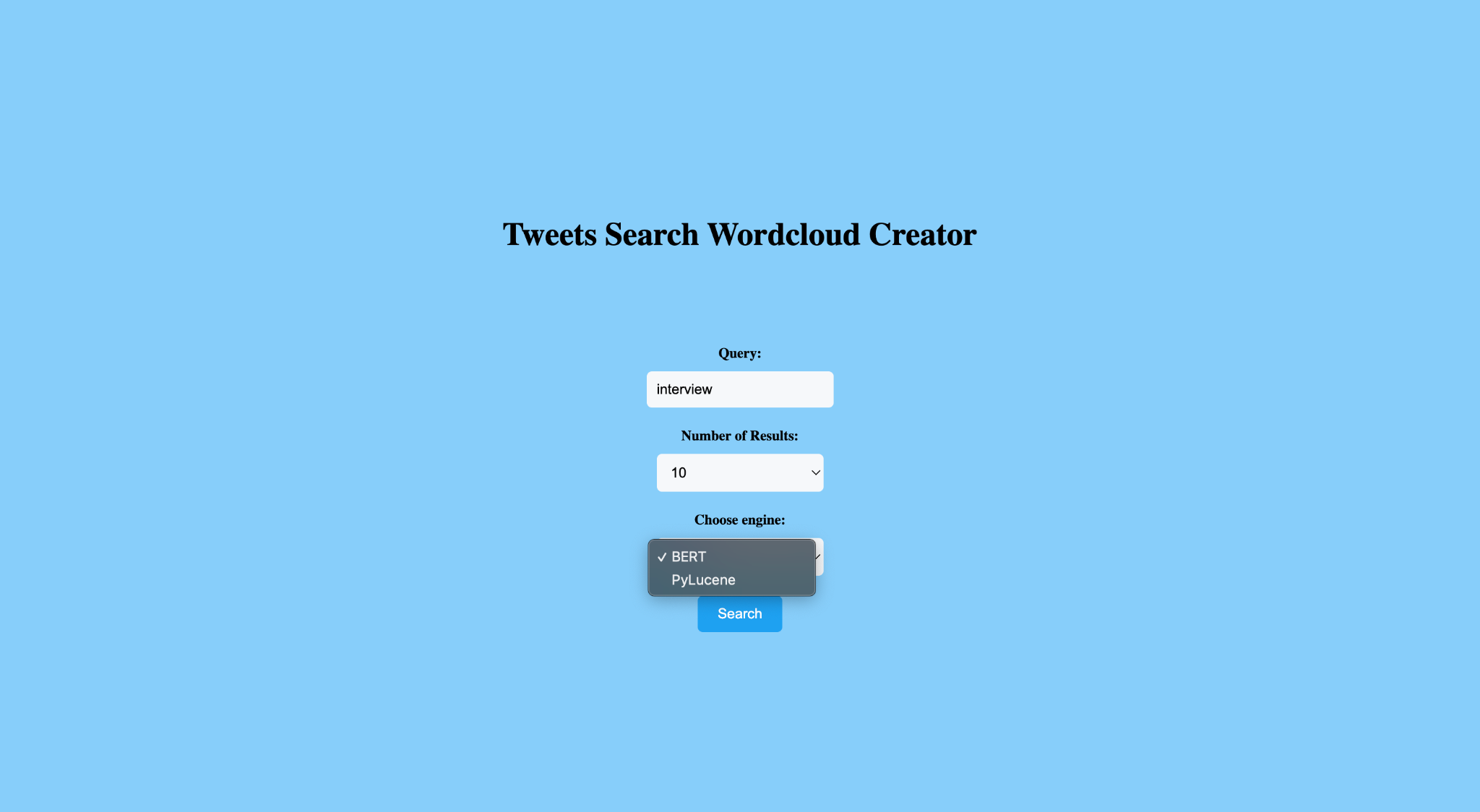
## 6. Screenshots showing the system in action



Screenshot 1. Homepage



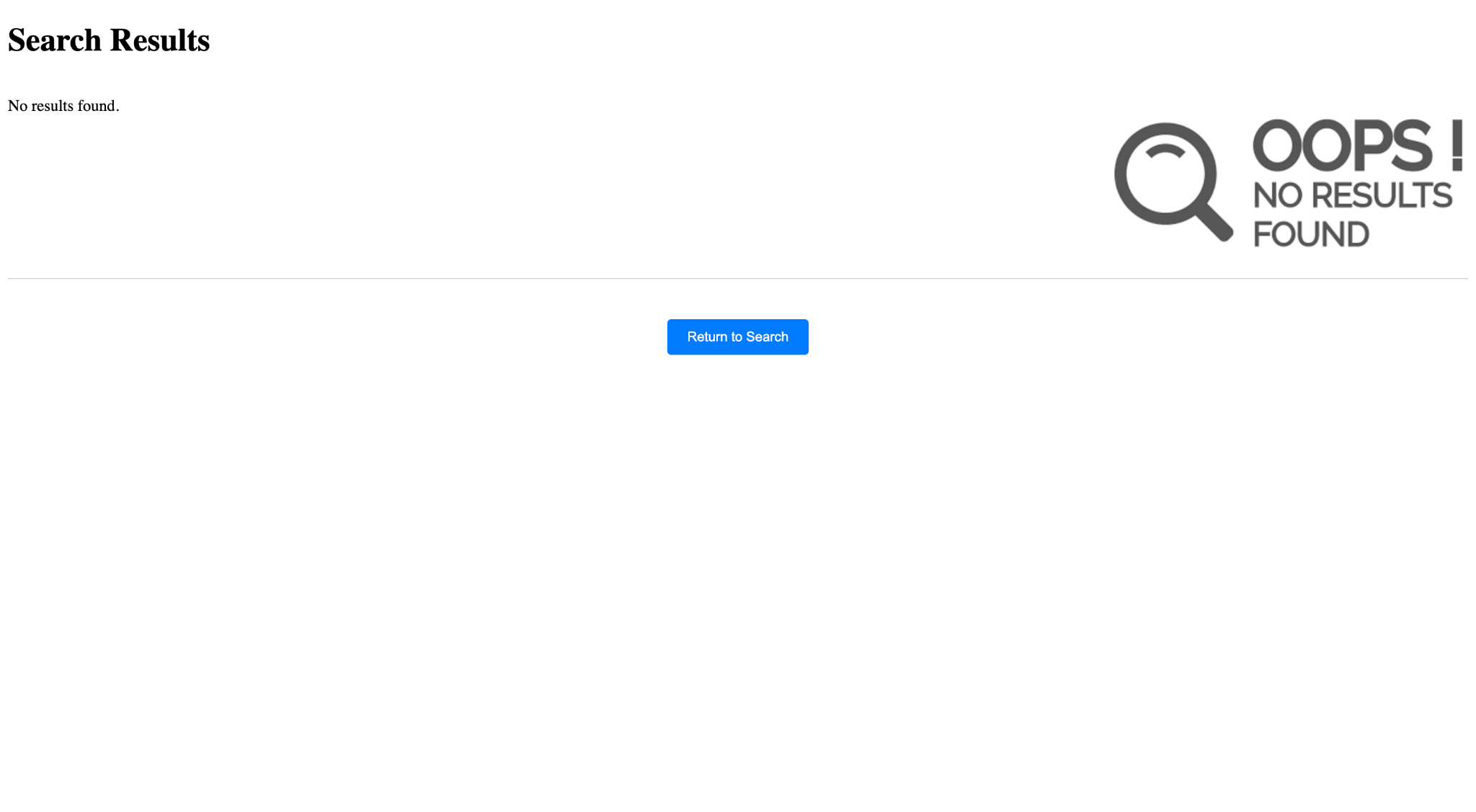
Screenshot 2. Showing number of results dropdown menu



Screenshot 3. Showing the engine choices dropdown menu



Screenshot 4. The results page of searching “interview” using BERT



Screenshot 5. The results page of searching a query that has no result