Annie He

Literature Organization Report

For this project, the setting included the COVID-19 Open Research Dataset (CORD-19), a collection of over 33,000 scientific articles related to the novel COVID-19 coronavirus. Each scientific article has key properties, including a unique paper ID, title, abstract, and body text. Our goal was to implement natural language processing and information retrieval techniques to design a search engine that can select the top one hundred articles that are most relevant to the query.

In order for the user to receive articles in a short time after entering the query, we must first preprocess and organize the articles. The first step in preprocessing is the parsing and extraction of any information from the articles. At first, we wanted to extract the body text, but the runtime was too long (almost half a day). But after going through some of the actual files, we believed that the scientists placed the most important keywords in the titles and abstracts. Therefore, we have chosen to extract only the paper IDs, titles, and abstracts. We parsed the text by removing punctuation and stop words, and stemming each word, in this order. By removing punctuation and stop words first, stemming the remaining words was more efficient. We also used the nltk package (stopwords, PorterStemmer) to help us with parsing. Furthermore, I suggested to use dataclass for articles because this decorator would allow us to keep track of the properties and create the final extracted article. Lastly, we noticed many documents were not in English, were improperly formatted, or were lacking components. Thus, we decided to exclude non-English documents from preprocessing, as well as documents without a title and abstract. Surprisingly, this decision cut out 13% of the documents!

The next step is the organization of the collection. After gathering the necessary parsed information from each article, we planned to calculate the TF-IDF scores of each word in the corpus and create an inverted index of these TF-IDFs. To determine the TF-IDF score of a specific word in a specific document, we first count the number of times this term appears in the document (TF) and then divide by the total number of terms in the document. Next, we compute the inverse document frequency, which measures how important a term is. The rarer a term is, the higher its IDF score is. By multiplying these two scores together, we have the term's retrieval status value for one document.

$$DF(t, D) = \sum_{d \in D} \begin{cases} 1 & \text{if } t \in d \\ 0 & \text{else} \end{cases}$$

Figure 1. Document frequency

$$IDF(t, D) = \log \frac{|D|}{DF(t, d)}$$

Figure 2. Inverse document frequency

$$RSV(Q, d) = \sum_{q \in Q} TF(q, d) * IDF(q)$$

Figure 3. Retrieval status value

The titles and abstracts would each have their own vectorizer to calculate TF-IDF. Then, we would sum the TF-IDFs together, with the titles' TF-IDFs weighted more than the abstracts'. At first, we used sklearn's package to obtain these TF-IDF values in a matrix form (code attempt: `demo_vsm.py`). However, creating the inverted index (a nested dictionary) of keys being the words and the values being a dictionary of document IDs to TF-IDF scores had a long runtime, just for abstracts alone. We were also stumped by how the inverted index would come into play with a query in the retrieve method. How would we use this mapping of words to their documents and TF-IDF scores to find the most relevant articles? So we decided to try another approach. Instead of sklearn, we discovered the rank-bm25 package. Based on a probabilistic relevance model, Okapi BM25 not only handles the TF-IDF scoring, but also accounts for term saturation and length normalization. We believed this model was more robust in scoring documents relating to a query than the vector space model was. We also combined weighted repeats of the title with the abstract into one string for each article, which is similar to the oversampling technique. Oversampling the title made the title words more significant than the abstract words.

The final part is to design the search engine with the retrieve function. In `demo_vsm.py`, we used the vectorizer to transform the query into a TF-IDF document-term matrix, and then computed the cosine

similarity between the query vector and each document vector. The smaller the cosine value, the closer in relation the two vectors are, which means the more relevant this document is to the query. However, in our BM25 attempt, the model quickly collected the top 100 scores, and then we sorted them in descending order. Finally, after obtaining the paper IDs from extract.csv, we printed 100 document paper IDs for each query.

$$k(x, y) = \frac{xy^\top}{\|x\|\|y\|}$$

Figure 5. Cosine similarity equation

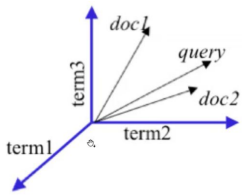

Figure 6. Vector space model

$$RSV(Q, d) = \sum_{q \in Q} IDF(q) * \frac{TF(q, d) * (k + 1)}{TF(q, d) + k * (1 - b + b(\frac{L(d)}{\bar{L}}))}$$

Figure 7. BM25 equation

I believe the most important decision we made for this assignment was the use of csv and pickle files to store information. At first, we were storing the extracted information of articles into a huge dictionary in memory. From only preprocessing the titles and abstracts, extraction took hours to complete. After discussing with the professor and TAs, they suggested writing to an output file after preprocessing each document. With their advice, we wrote the parsed information (paper ID, title, abstract) to a csv file after every file iteration. In the organize method of `demo_vsm.py`, we also stored the vectorizer and the TF-IDF matrix into separate pickle files. In the organize method of `demo.py`, we stored the BM25 model in a pickle file. With all the data stored into files, it was possible to run extract and organize once because we would simply open these files in retrieval. This method reduced our runtime from many hours to half an hour, along with retrieval only taking a few seconds! All files (some are zipped) are included in our GitHub. We also submitted our code to Kaggle (https://kaggle.com/joshreitz/biol1595-assign2/).

Our literature organization and search engine model, with weighting and writing to output files, performed quite well. We manually compared our search engine results to those from the CORD-19 collection website (https://cord-19.apps.allenai.org/). We counted the number of matches between their top 100 results and our top 100 results for each query. To increase the number of matches, we tweaked the ratio of title repeats to abstract repeats. For the vector space model, the best ratio of 5:1 resulted in about a 25% match to CORD-19's top 100. Meanwhile, the BM25 model yielded a maximum of 48% match rate with a 3:1 ratio for the query "coronavirus response to weather changes", even though many of CORD-19's top 100 articles did not exist in our collection. Therefore, we ultimately decided to consider the BM25 model as our best design for a search engine.

Query	Title weight	Abstract weight	# of VSM Matches	# of BM25 Matches
coronavirus origin	3	1	18	38
coronavirus origin	5	1	21	34
coronavirus origin	8	1	22	33

Table 1. Number of matches based on different weight ratios between the VSM and BM25 models for an example query

Future directions would include cross language support, enhancing the usability, and including other article elements (body text, authors, etc.). Translation of the non-English documents may expand our results but would require additional preprocessing. Enhancing the usability could include displaying the links to the articles.