TREC-COVID: Information Retrieval System for COVID-19 Related Documents

1. Motivation

Our goal is to develop an information retrieval system to find relevant documents for each given topic in the dataset.

- The timeliness of this topic.
- How helpful it is for the public to know reliable information about the virus. To help mitigate the information crisis regarding COVID-19.
- Clinicians and clinical research during the COVID-19 pandemic: It is also helpful for researchers to study and manage the rapidly growing COVID-19 corpus.
- We hope that our study will contribute to the global effort in fighting the COVID-19 pandemic and any future events.

2. Dataset

4.1 The CORD-19

TREC-COVID dataset consists of the Round 5 document set. We used a dataset provided by Kaggle that contains the May 19, 2020, version of CORD-19 documents and metadata (Round 3).

- Data
    - JSON files
        - PDF documents in JSON format
        - PMC(PubMed Central) documents in JSON format
    - Metadata.csv
    - Qrels.csv (used for evaluation)

4.2 Topics

The round-3 Topics data file (topics-rnd3.csv) contains the topic-id, query, question, and narrative for 40 topics.

1. query: a short keyword query
2. question: a more precise natural language question
3. narrative: a longer description that further elaborates on the question, often providing specific types of information that would fall under the topic score

We used this dataset to create multiple variations of input queries. We evaluated our initial results with the following three input query schemes.

1. Query
2. Query+Question
3. Query+Question+Narrative

```
[45] topics_df = pd.read_csv('/content/topics-rnd3.csv')
     topics_df.head()
```

	topic-id	query	question	narrative
0	1	coronavirus origin	what is the origin of COVID-19	seeking range of information about the SARS-Co...
1	2	coronavirus response to weather changes	how does the coronavirus respond to changes in...	seeking range of information about the SARS-Co...
2	3	coronavirus immunity	will SARS-CoV2 infected people develop immunit...	seeking studies of immunity developed due to i...
3	4	how do people die from the coronavirus	what causes death from Covid-19?	Studies looking at mechanisms of death from Co...
4	5	animal models of COVID-19	what drugs have been active against SARS-CoV o...	Papers that describe the results of testing d...

Figure 2: Snippet of the Topics data file

3. Methodology

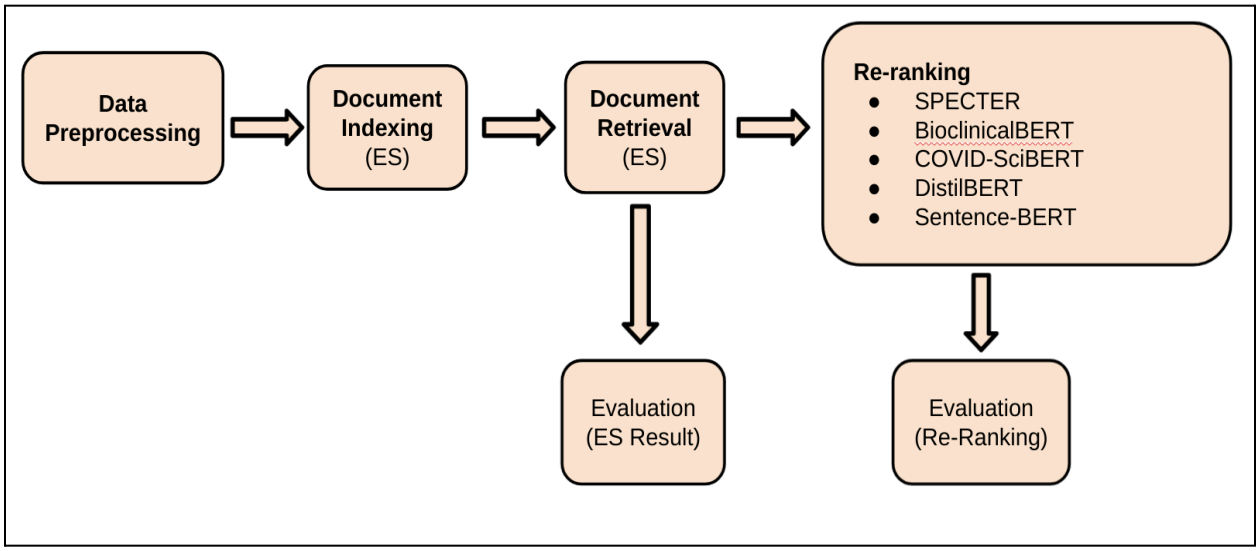

Figure 1: Proposed System Architecture

1. Data Pre-processing:
   - Process raw documents into a searchable format.
2. Document Indexing:
   - Store processed documents into an ES index.
3. Document Retrieval
   - Retrieve top n relevant documents from the index for each query.
4. Re-ranking:
   - Apply re-ranking algorithms using BERT-based models to boost the ranking of relevant documents.

3.1 Data Pre-processing

We primarily use Metadata.csv, which contains cord_uid, title, abstract, and other fields for 128K research papers. The cord_uid is a unique ID for each document (research paper). We dropped the 330 duplicated cord_uid entries in the dataset. We extracted cord_uid, title, and abstract from this dataset and used this for our initial testing.

Data parsing:
We parse the JSON files to extract the introduction section for each paper. We used the metadata file to get 128K cord_uids. For each cord_uid, we accessed the JSON mentioned under the 'pdf_json_files' field. We used the JSON file to access the full text of the paper and extracted the introduction sections if available. We combined the multiple introduction sections into a single section.

```
[27] newdata_df.head()
```

	cord_uid	title	abstract	introduction
0	ug7v899j	clinical features of culture-proven mycoplasma...	objective: this retrospective chart review des...	mycoplasma pneumoniae is a common cause of upp...
1	02tnwd4m	nitric oxide: a pro-inflammatory mediator in l...	inflammatory diseases of the respiratory tract...	since its discovery as a biological messenger ...
2	ejv2xln0	surfactant protein-d and pulmonary host defense	surfactant protein-d (sp-d) participates in th...	surfactant protein-d (sp-d) is a member of the...
3	2b73a28n	role of endothelin-1 in lung disease	endothelin-1 (et-1) is a 21 amino acid peptide...	
4	9785vg6d	gene expression in epithelial cells in respons...	respiratory syncytial virus (rsv) and pneumoni...	rsv and pvm are viruses of the family paramyxo...

```
[28] newdata_df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 128162 entries, 0 to 128491
Data columns (total 4 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   cord_uid      128162 non-null  object
 1   title         128162 non-null  object
 2   abstract      128162 non-null  object
 3   introduction  128162 non-null  object
dtypes: object(4)
memory usage: 4.9+ MB
```

Figure 3: Dataset with cord_uid, title, abstract, and introduction for 128K research papers.

Concatenation:
We combined the title, abstract, and introduction columns into a single text column. We used this dataset for document embedding.

```
data.head()
```

	cord_uid	text
0	ug7v899j	clinical features of culture-proven mycoplasma...
1	02tnwd4m	nitric oxide: a pro-inflammatory mediator in l...
2	ejv2xln0	surfactant protein-d and pulmonary host defens...
3	2b73a28n	role of endothelin-1 in lung disease endotheli...
4	9785vg6d	gene expression in epithelial cells in respons...

Figure 4: The title, abstract, and introduction columns are concatenated into a single text column. The text column was used to create embeddings for each document.

3.2 Document Indexing & Retrieval using Elasticsearch

Elasticsearch is an open-source, broadly distributable, readily scaled, enterprise-grade search engine. Our research uses Elasticsearch for information retrieval since this software deploys the BM25 algorithm and is scalable for many records. Queries are significantly faster with Elasticsearch. Elasticsearch is built on top of Apache Lucene, an information retrieval software library that is fast and open source. It uses an inverted document index for indexing and searching. Apache Lucene allows us to do tokenizing, stemming, filtering stop words and HTML tags, bounding box searches, relevance scoring, and provides advanced search options like synonyms, based on similarity proximity.

3.2.1 Indexing Pipeline

This is the indexing pipeline of Elasticsearch. The documents are fed into the indexing pipeline. The task of the analyzer is to create tokens using a tokenizer and/or apply filters. Additionally, each field can define an analyzer at index or query time or both. Finally, the output tokens are fed into the document writer. It is also possible to have multiple threads in our indexing process, each with its own document writer; it is called concurrent indexing, which is another feature of Lucene.

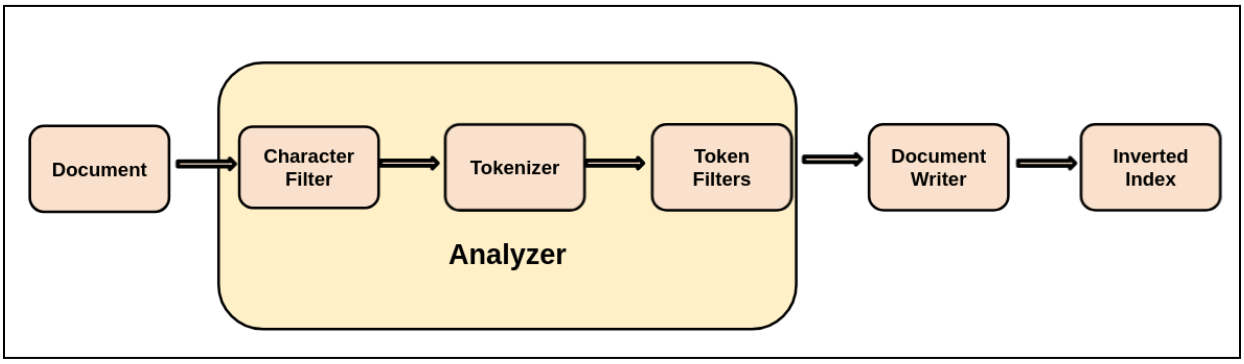

Figure 5: Indexing Pipeline in Elasticsearch

In order to set up Elasticsearch, we have used the open-source version of the Elasticsearch package. We have tried several methods of getting the Elasticsearch setup, and due to the unsupported product issue that we faced, we decided to use a version of Elasticsearch prior to 7.14. Hence we have used Elasticsearch version 7.13, we downloaded and set up a working instance of ES at localhost port 9200 (http://localhost:9200/). By CURLing into it, we have confirmed that the Elasticsearch instance is up and running.

In our work, we used the “Python Elasticsearch Client” to carry out the indexing and querying using ES. We installed it using pip and imported the Elasticsearch object from the “Elasticsearch” module (ex: *from Elasticsearch import Elasticsearch*) for initializing it.

```
[ ] %%bash

    wget -q https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.13.4-linux-x86_64.tar.gz
    wget -q https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.13.4-linux-x86_64.tar.gz.sha512
    tar -xzf elasticsearch-7.13.4-linux-x86_64.tar.gz
    sudo chown -R daemon:daemon elasticsearch-7.13.4/
    shasum -a 512 -c elasticsearch-7.13.4-linux-x86_64.tar.gz.sha512

    elasticsearch-7.13.4-linux-x86_64.tar.gz: OK

[ ] %%bash --bg

    sudo -H -u daemon elasticsearch-7.13.4/bin/elasticsearch

    Starting job # 0 in a separate thread.

[ ] # Sleep for few seconds to let the instance start.
    time.sleep(30)

[ ] %%bash
    curl -sX GET "localhost:9200/"

    {
      "name" : "4258361c2d5b",
      "cluster_name" : "elasticsearch",
      "cluster_uuid" : "De8Ki9TMQ1ucH7tjCm_R8Q",
      "version" : {
        "number" : "7.13.4",
        "build_flavor" : "default",
        "build_type" : "tar",
        "build_hash" : "c5f60e894ca0c61cdbae4f5a686d9f08bcefc942",
        "build_date" : "2021-07-14T18:33:36.673943207Z",
        "build_snapshot" : false,
        "lucene_version" : "8.8.2",
        "minimum_wire_compatibility_version" : "6.8.0",
        "minimum_index_compatibility_version" : "6.0.0-beta1"
      },
      "tagline" : "You Know, for Search"
    }
```

Figure 6: Setting up ES Instance

```
[ ] !python -m pip install elasticsearch

    Collecting elasticsearch
      Downloading elasticsearch-7.16.0-py2.py3-none-any.whl (385 kB)
         |████████████████████████████████| 385 kB 13.7 MB/s
    Requirement already satisfied: urllib3<2,>=1.21.1 in /usr/local/li
    Requirement already satisfied: certifi in /usr/local/lib/python3.7
    Installing collected packages: elasticsearch
    Successfully installed elasticsearch-7.16.0
```

Initialize Elasticsearch object

```
[ ] es = Elasticsearch(timeout=30, max_retries=30, retry_on_timeout=True)
    es.info()

    {'cluster_name': 'elasticsearch',
     'cluster_uuid': 'OWxlcdSQTLaYsn0yc3nVLw',
     'name': 'e120c0cfd613',
     'tagline': 'You Know, for Search',
     'version': {'build_date': '2021-07-14T18:33:36.673943207Z',
      'build_flavor': 'default',
      'build_hash': 'c5f60e894ca0c61cdbae4f5a686d9f08bcefc942',
      'build_snapshot': False,
      'build_type': 'tar',
      'lucene_version': '8.8.2',
      'minimum_index_compatibility_version': '6.0.0-beta1',
      'minimum_wire_compatibility_version': '6.8.0',
      'number': '7.13.4'}}
```

Figure 7: Installing the python Elasticsearch wrapper and Initializing the Elasticsearch object.

3.2.2 Create Index & Document Indexing

Elasticsearch offers the flexibility of creating your own index mappings. During indexing, Elasticsearch converts raw data (our pre-processed documents) into internal documents and stores them in a basic data structure similar to a JSON object. Each document is a simple set of correlating keys and values. In the figure below (Figure 8) is a simple code block that can be used to create an index named “esindex” and index the 128162 documents using a for a loop. The platform allows fast searches in large clusters by partitioning indexes into shards and replicating them across the cluster.

```
[91] print("creating the '{}' index.".format("esindex"))
     res1 = es.indices.create(index="esindex")
     print("Response from server: {}".format(res1))

     creating the 'esindex' index.
     Response from server: {'acknowledged': True, 'shards_acknowledged': True, 'index': 'esindex'}
```

```
[ ]  for i in range(0,len(metadata_records)):
      #print(i)
      es.index(index="esindex", doc_type="test-type", id=i, body=metadata_records[i])
```

Figure 8: Installing the python Elasticsearch wrapper and Initializing the Elasticsearch object.

3.2.3 Querying the Index & Creating the Output Submission File

Elasticsearch provides query DSL - a simple JSON style domain-specific language that enables users to execute search queries. A query will examine one or many target values and score each of the elements and results according to how closely they match the query focus. We used the “multi_match” keyword since it is a convenient way of running the same query against multiple fields. The “query” property specifies which query text to be used when querying. We tried three different query alternatives from the given query dataset: query, query+question, and query + question + narrative. The “fields" property specifies what fields to query against. In our case, we wanted to query against the title, abstract, and intro fields in the document. The “size” parameter helps us control how many documents we need to be retrieved. By default, searches return the top 10 matching hits, we have tried 10, 30, 100, 500, and 1000. Finally, the Dataframe is saved as a CSV file into our Google Drive to be evaluated. The tool used for evaluation and the process followed will be discussed in the final evaluation section.

```
submission = []
for _, row in topics_df.iterrows():
    result_search = es.search(index='trec-covid-esindex2', size = 30, query={
        'multi_match': {
            'query': row['text2'],
            'fields': ['title', 'abstract', 'introduction']
        }
    })
    result_search_df = Select.from_dict(result_search).to_pandas()
    submission.extend((row['topic-id'], cord_uid, score) for cord_uid, score in zip(result_search_df['cord_uid'],result_search_df['_score']))
submission_df2= pd.DataFrame(submission, columns=('topic-id', 'cord-id', 'score'))
```

Figure 9: Querying the Elasticsearch Index and Saving the output as a Pandas Dataframe

```
[ ] submission_df.to_csv('/content/drive/MyDrive/CS834/elastic_result_1_new.csv', index=False)
```

Figure 10: Saving the Pandas Dataframe as a CSV file into the Drive

3.3 Document Re-ranking

This is the document re-ranking pipeline. For document re-ranking, we used the results of the Elasticsearch and fed them to transformer language models. Re-rankers estimate the relevance of a document, d to a query, q by estimating a score, s. The score is a value that lies between 0 &1 and this score is a measure of the relevancy of a document, d to a query, q.

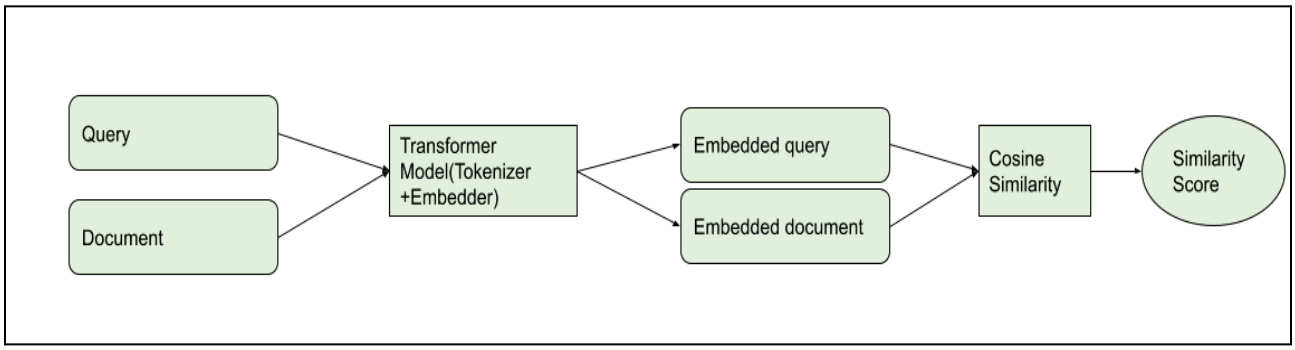

Figure 11: Document Re-ranking Pipeline

3.3.1 Document & Query Embedding:

In this phase, we used transformer language models to embed queries and documents. We took the top 30 relevant documents for each query from the results of Elasticsearch and then we embedded them. This is how the Elasticsearch result looks like:

```
submission_df2 = pd.read_csv('elastic_result_2.csv')
#submission_df2 = pd.read_csv('/content/drive/MyDrive/CS834//elastic_result_2.csv')
```

```
submission_df2.head(8)
```

	topic-id	cord-id	score
0	1	ab757i3f	55.514620
1	1	d6by9p41	55.514620
2	1	deajwhx0	47.746610
3	1	apqizr54	47.746610
4	1	k9lcpjyo	47.311020
5	1	dv9m19yk	47.058220
6	1	lbv69du4	46.163776
7	1	dtv7to3l	45.959045

Figure 12: Elasticsearch result

In the result, the “topic-id” represents the query and the “cord-id” column represents the documents. The score column is a measure of how relevant a query is to a particular document. For the re-ranking phase, we concatenate the “query”, “question”, and “narrative” columns from the query dataset. These concatenated strings represent queries. We took the “title” and “abstract” of the document dataset and concatenated them which represents documents. Then, we used transformer language models to embed the query and the documents. A transformer-based language model learns context-specific relations between words (or sub-words) in a text using an attention mechanism. The transformer consists of two mechanisms - an encoder that reads the text input and a decoder that produces a prediction for the task. We use the encoder mechanism in our task. BERT is a transformer-based language model which is a great breakthrough in the field of natural language processing. BERT's main technical innovation is its bidirectional training of Transformer, a popular attention model for natural language modeling. In our re-ranking phase, we used various BERT based language models as re-ranker:

1. BioclinicalBERT
2. COVID-SciBERT
3. SPECTER
4. DistilBERT
5. Sentence-BERT

BioclinicalBERT: BioclinicalBERT was initialized from BioBERT & trained on all MIMIC notes. In our code, we loaded the model via the transformer library.

```
from transformers import AutoTokenizer, AutoModelForMaskedLM
bio_tokenizer = AutoTokenizer.from_pretrained("mrm8488/bioclinicalBERT-finetuned-covid-papers")
bio_model = AutoModelForMaskedLM.from_pretrained("mrm8488/bioclinicalBERT-finetuned-covid-papers")
```

COVID-SciBERT: The original SciBERT model is an expansion of the BERT model and is trained on papers from the corpus of semanticscholar.org. COVID-SciBERT, an
expansion of SciBERT, and is done using the papers present in the COVID-19 Open Research Dataset Challenge (CORD-19). This model was loaded using the following commands from the transformer library.

```
from transformers import AutoTokenizer, AutoModelForMaskedLM, AutoModel
model = transformers.AutoModelWithLMHead.from_pretrained('lordtt13/COVID-SciBERT')
tokenizer = transformers.AutoTokenizer.from_pretrained('lordtt13/COVID-SciBERT')
```

SPECTER: SPECTER uses citation graphs based on pretraining a transformer language model to construct document-level embeddings for scientific documents. It incorporates inter-document context into the transformer language models to learn document representations. SPECTER was loaded using the following commands from the transformers library.

```
from transformers import AutoTokenizer, AutoModel
specter_tokenizer = AutoTokenizer.from_pretrained('allenai/specter')
specter_model = AutoModel.from_pretrained('allenai/specter')
```

DistilBERT: DistilBERT is a light, fast, and cheap transformer model that is trained from BERT. It is a distilled version of BERT. In our code, we loaded the model via the transformer library.

```
import transformers as ppb
model_class = ppb.DistilBertModel
tokenizer_class, pretrained_weights = (ppb.DistilBertTokenizer, 'distilbert-base-uncased')
```

Sentence-BERT: In Sentence-BERT (SBERT), siamese and triplet network structures are used to derive semantically meaningful embeddings of a sentence that can be compared using cosine-similarity.

```
from sentence_transformers import SentenceTransformer
embedder = SentenceTransformer('bert-base-nli-mean-tokens')
```

3.3.2 Computing cosine similarity:

We use the query and document embedding from transformers to compute the cosine similarity between each query and document pair.

```
  query_embeddings = tokenize_mask_train(tokenized_query,tokenizer,model)
  corpus_embeddings = tokenize_mask_train(tokenized_test,tokenizer,model)

  print("query_embeddings\n\n")
  print(query_embeddings)
  print("====================")
  print("corpus_embeddings:\n\n")
  print(corpus_embeddings)
```

```
query_embeddings


[[ -4.611788   -10.0737295   -9.397793   ...  -3.4378543   -2.8042665
   -0.31411657]]
====================
corpus_embeddings:


[[ -6.3581166  -12.232764   -11.7638855  ...  -4.8635635   -3.7523048
   -1.4132162 ]
 [ -6.3402815  -12.414287   -11.820553   ...  -4.8806405   -3.7209344
   -1.3749849 ]
 [ -6.6121893  -10.582827   -10.170077   ...  -2.1952105   -1.538495
   -0.99576056]
 ...
 [ -8.777595   -10.474017   -10.563586   ...  -2.6131158   -1.7548635
   -0.44237462]
 [ -5.742764   -10.050857   -11.160933   ...  -2.846777    -1.3007455
   -0.04962496]
 [ -5.686034   -10.232718   -11.210122   ...  -2.9830966   -1.2939622
   -0.08194567]]
```

Figure 13: Query and document embedding

On the basis of the computed cosine similarity score, we rank the documents for each query.

```
print("Re-ranking Result using COVID-Sci-BERT")
print(dataframe.head(6))
```

```
Re-ranking Result using COVID-Sci-BERT
   topic_id    cord_id     score
0         1   dv9m19yk  0.988361
1         1   k9lcpjyo  0.984367
2         1   xummm9xu  0.983956
3         1   lqkwsh6a  0.981658
4         1   ectly1v4  0.981259
5         1   dtv7to3l  0.980925
```

Figure 14: Re-ranking result using COVID-SciBERT

4. Results & Evaluation

In order to evaluate the documents retrieved for each topic using our Information Retrieval System, we used Trec_eval (TREC evaluation tool). Trec_eval is the standard tool used by the TREC community for evaluating an ad hoc retrieval run, given the results file and a standard set of judged results. Kaggle provided us with judge results in the form of qrels.csv (Figure 15). The qrel file has the relevant judgments for documents that have been evaluated in the previous two rounds. The relevance judgments have been made by human annotators that have biomedical expertise.

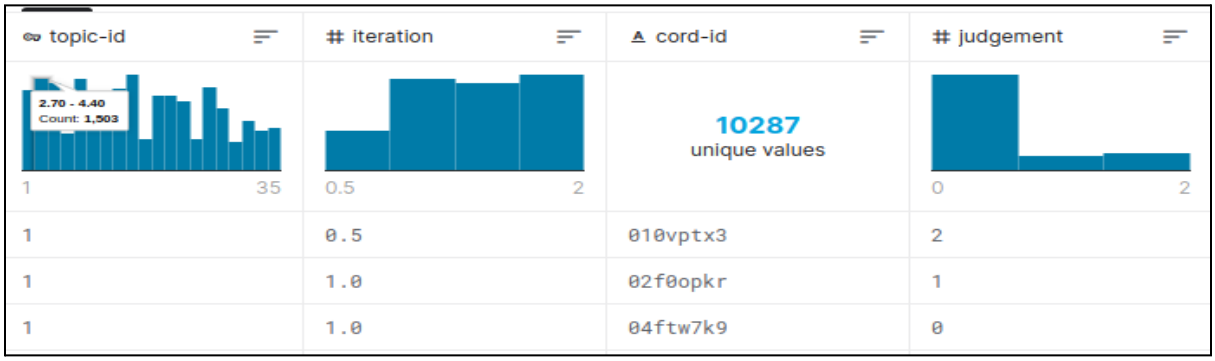

Figure 15: qrels file contains a standard set of judged results.

The Trec_eval tool provides multiple measures for evaluation. We reported the following measures for our models:

- MAP - Mean Average Precision
- GMAP - Geometric mean of (per-topic) average precision
- NDCG - Normalized Discounted Cumulative Gain

4.1 Elasticsearch results

We evaluated our Elasticsearch models with multiple parameters to fine-tune our results. Our first parameter was the input query. We used the topics data file to concatenate the query, question, and narration fields and use them as our input queries. We evaluated our initial results with three input query schemes. Table 1 shows the values of map, gm_map, and NDCG for three input query schemes. We received the best results while using the concatenate of all three fields (query + question + narrative) as input query. We used this input query for our later models as well.

Input query	map	gm_map	NDCG
query	0.0097	0.0020	0.0457
query + question	0.0128	0.0037	0.0559
query + question + narrative	0.0157	0.0043	0.0695

Table 1: map, gm_map and NDCG for 3 Variations of Input Query. The result is based on Top 10 relevant documents (n=10) for given 40 topics.

Elasticsearch by default retrieves the top 10 relevant documents for each topic. For example, we received 400 relevant documents for 40 topics. We tested the Elasticsearch results by increasing the number of top relevant documents retrieved for each topic (At different document cutoff levels, size=n). Figure 16 shows the linear relation between the number of top relevant documents retrieved for each topic and the map value. We got the best result on retrieving the top 1000 relevant documents for each topic:

- **Mean average precision (MAP) -0.0765**
- Geometric mean average precision (gm_map) - 0.0608
- NDCG - 0.2948

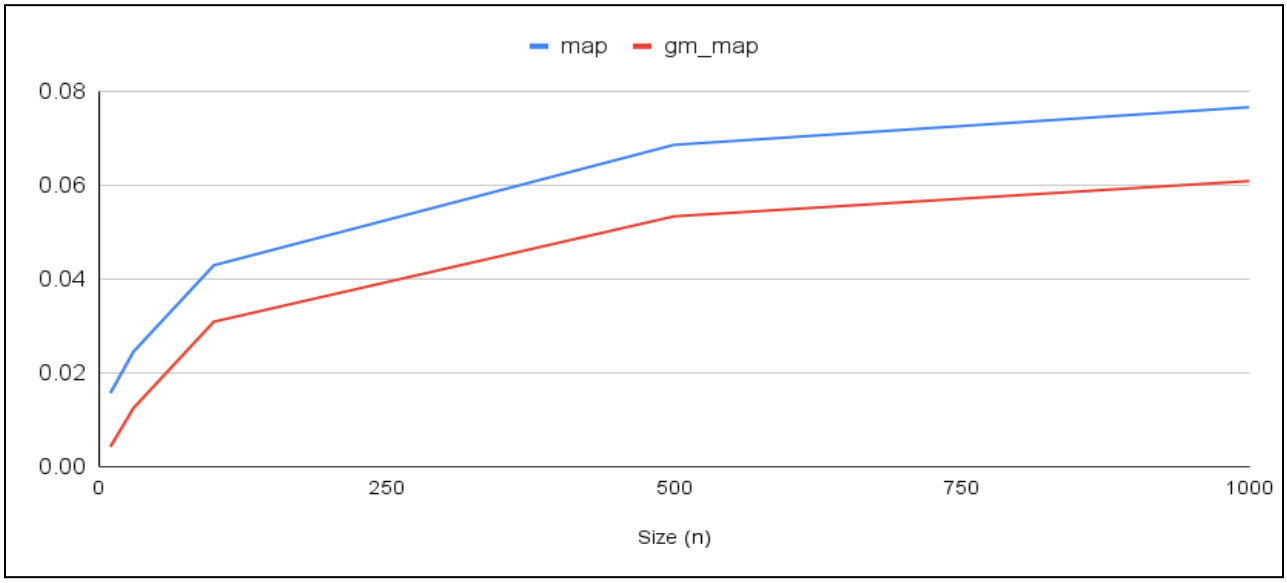

Figure 16: map and gm_map values for different sizes (number of top relevant documents retrieved for each topic).

4.2 Reranking results

Model	map	gm_map	NDCG
BioclinicalBERT	0.0097	0.0035	0.0454
COVID-SciBERT	0.0136	0.0031	0.0591
SPECTER	0.0109	0.0042	0.0514
DistilBERT	0.0131	0.0034	0.0581

Sentence-BERT	0.0094	0.0036	0.0454

Table 2: Re-ranking results for top 10 relevant documents

Initially, we focused on the top 10 relevant documents for each query we received from Elasticsearch. We reranked them using 5 different language models: BioclinicalBERT, COVID-SciBERT, SPECTER, DistilBERT, Sentence-BERT. From the comparison, we see that COVID-SciBERT produced the highest map and NDCG. Then we decided to rerank the top 100 documents from Elasticsearch results based on their relevance to each query using only COVID-SciBERT. COVID-SciBERT took approximately 8 hours to embed all 100 relevant documents for each query during the re-ranking phase. For the top 100 relevant documents, we achieved:

- Mean average precision (MAP) - 0.0362
- Geometric mean average precision (gm_map) - 0.0265
- NDCG - 0.1562

Our experiment shows that increasing the number of relevant documents increases the NDCG, gm_map, and map of re-ranking results.

5. Discussion

In this research, we created an information retrieval system to retrieve relevant documents in the TREC CORD-19 dataset for each topic. We have first pre-processed the raw documents into a searchable format. We have then set up an Elasticsearch instance and we indexed the documents into the Elasticsearch index. The top n relevant documents were then retrieved from the index for each query. As the final step, we applied several re-ranking algorithms using BERT-based models to boost the ranking of relevant documents and compared the results for different re-ranking models. The best result was a mean average precision of 0.0765 when 1000 documents were retrieved for each topic. We received our best result (top 100) using the COVID-SciBERT model for re-ranking and it was a mean average precision of 0.0362.