CMS Semantic Search

Ammar Hani A Almarzooq, Dmytro Kovalskyi, German Giraldo, Hasan Ozturk

August 2024

1 Motivation

Research on the CMS experiment produces a large volume of literature, and it's important for people working on the experiment to be able to locate specific publications from such corpus. Current tools created to help with this task are limited–their search capacity manifests in matching exact substrings. This tool is insufficient especially given recent technological developments, particularly in the field of artificial intelligence and transformer-based language models.

This project aims to explore the capabilities of the state-of-the-art methods in conducting semantic search on the scientific publications of CMS.

1.1 Problem Formalization

This is the method using which we are able to evaluate a given model. We start with some terminology:

1. Query: the string a user may provide to the search engine
2. Target: the publication that the user is looking for given the user query.
3. Rank: the rank of the target publication among all other publications in the output of sorted publication, provided by a model when given a query by the user. A rank of one indicates the model rated the target as the most similar publication.

It can be seen that low ranks (closer to 1) suggest that the model is well performing. A way to evaluate the model is to use a set of queries as inputs and have a predetermined target per query. The evaluation metric could be the median rank across all the queries, and the set of query-target tuples can be called the test set.

> The efficacy of a model can be defined as how low the median rank is.

Of course, the bigger the test set, and the closer the queries to actual use, the more informative the evaluation is. Other performance metrics are the mean, lower and upper quartiles, each given some insight into the performance.

The above definitions assume that there is only one target per query, which is not necessarily always the case. A way to work around this is to include multiple tests where the query is the same but the targets are different and then decide a way to aggregate them together (minimum rank, average, or median).

2 Findings

Here is a run down of the different experiments we carried out: We used different models: BERT, SciBERT, T5, and GPT Neo. We explored different pre-processing approaches and parameters.

There are 2 publication sets that we used for results: one with 612 publications, and the other with 1333 publications. This is important to mention because the results with tests using the latter are expected to be worse since there are more competing articles.

Here are the benchmark prompts:

- **Basic Search**
  - "Bs to Mu Mu"
    BPH-21-006, BPH-16-004, BPH-13-007, BPH-13-004, BPH-11-020, BPH-11-002
  - "Latest B to dimuon"
    BPH-21-006
  - "Tracking performance"
    TRK-10-001, TRK-11-001
  - "Latest Higgs to WW"
    HIG-22-008, HIG-20-013, HIG-19-002
- **Advanced**
  - "Higgs discovery"
    HIG-12-028
  - "Higgs mass"
    HIG-19-004, HIG-14-042, HIG-14-009

2.1 SciBERT

2.2 Benchmark

We started with basic tests to some already existing search engines to use as benchmarks. The inputs to those are 15 queries: 5 queries are exact duplicates of the titles of ransom publications, 5 queries are title paraphrasings based on random publications. The other 5 are based on substrings of the paraphrased titles.

2.2.1 Google

Google Key	Raw	Paraphrased	Substring
1: 276	1st Result	Not in top 10	1st Result
2: 451	1st Result	5th Result	Not in top 10
3: 433	1st Result	3rd Result	4th Result
4: 178	1st Result	7th Result	4th Result
5: 15	1st Result	1st Result	1st Result

Table 1: Google Search Results

2.2.2 Perplexity

Perplexity is an AI-powered search engine that acts as a chatbot to answer questions and provide information directly, similar to how search engines work but with a conversational interface.

Perplexity Key	Raw	Paraphrased	Substring
1: 276	Fail	Fail	Fail
2: 451	Fail	Fail	Fail
3: 433	Fail	Fail	Fail
4: 178	Success	Success	Fail
5: 15	Success	Fail	Fail

Table 2: Perplexity Search Results

2.2.3 ChatGPT-4o

ChatGPT-4o Key	Raw	Paraphrased	Substring
1: 276	Success	Fail	Success
2: 451	Success	Fail	Fail
3: 433	Success	Fail	Fail
4: 178	Success	Fail	Success
5: 15	Success	Success	Success

Table 3: ChatGPT-4o Search Results

2.2.4 Titles Test

Showing stats for a sample size of 13 test queries, 612 sorted titles:
Mean: 213.23
Median: 213
Best: 3
Worst: 480
Lower Quartile: 126.5
Upper Quartile: 267.5
Percent of matches in top 3: 7.69%
Percent of matches in top 5: 7.69%
Rankings: [196, 133, 3, 283, 114, 480, 212, 218, 252, 329, 120, 219, 213]

2.2.5 Abstracts Test

Showing stats for a sample size of 12 test queries, 612 sorted abstracts:
Mean: 282.67
Median: 269.5
Best: 16
Worst: 551
Lower Quartile: 89.5
Upper Quartile: 486.0
Percent of matches in top 3: 0.00%
Percent of matches in top 5: 0.00%
Rankings: [329, 99, 214, 284, 524, 68, 80, 515, 551, 255, 457, 16]

2.2.6 Title and Abstract Test

Showing stats for a sample size of 12 test queries, 612 sorted titles-abstracts:
Mean: 275.17
Median: 233.0
Best: 70
Worst: 599
Lower Quartile: 139.0
Upper Quartile: 408.0
Percent of matches in top 3: 0.00%
Percent of matches in top 5: 0.00%
Rankings: [187, 112, 201, 372, 265, 85, 166, 510, 599, 291, 444, 70]

2.2.7 Chopped Embeddings

Showing stats for a sample size of 12 test queries, 612 sorted titles-abstracts:
Mean: 224.75
Median: 208.0
Best: 15
Worst: 592
Lower Quartile: 102.0
Upper Quartile: 278.5
Percent of matches in top 3: 0.00%
Percent of matches in top 5: 0.00%
Rankings: [89, 264, 216, 25, 115, 190, 241, 457, 592, 15, 200, 293]

2.2.8 Keywords

Showing stats for a sample size of 7 test queries, 800 sorted titles:
Mean: 299.14
Median: 386
Best: 25
Worst: 586
Lower Quartile: 41
Upper Quartile: 524
Percent of matches in top 3: 0.00%
Percent of matches in top 5: 0.00%
Rankings: [386, 41, 586, 25, 524, 405, 127]

2.3 T5

Showing stats for a sample size of 13 test queries, 612 sorted abstracts:
Mean: 307.00
Median: 284
Best: 2
Worst: 618
Lower Quartile: 142.0
Upper Quartile: 497.0

Percent of matches in top 3: 7.69%
Percent of matches in top 5: 7.69%
Rankings: [474, 112, 618, 363, 613, 76, 520, 246, 172, 284, 191, 2, 320]

2.4 GPT Neo

Showing stats for a sample size of 12 test queries, 1333 sorted titles:
Mean: 296.25
Median: 204.5
Best: 14
Worst: 773
Lower Quartile: 64.5
Upper Quartile: 552.0
Percent of matches in top 3: 0.00%
Percent of matches in top 5: 0.00%
Rankings: [131, 773, 697, 596, 219, 69, 190, 14, 508, 29, 269, 60]

3 Project Description

This section discusses the details of the repository and projects, which should provide a comprehensive tour of the project aspects, and can help readers contribute to the project.

3.1 Set Up

Here are steps to set up the app in development mode. Deployment steps can be found in their respective sections.The repository can be found in https://gitlab.cern.ch/cms-semantic-search/cms-semantic-search. One may use:

```
git clone https://gitlab.cern.ch/cms-semantic-search/cms-semantic-search.git
```

Install the python requirements:

```
pip3 instal -r ./backend/requirements.txt
```

3.2 The Backend

The backend refers to the part of the repository which contains the actual search engine that is in use, as well as some test files, datasets and corpora, and some utilities. The Backend is deployed in vocms0103.cern.ch.

3.2.1 Flask App

The backend is served using a flask app. The app exists in bert/scibert_express.py (this file name is as of the time of writing the report and may have changed). The app contains a number of endpoints, namely a default endpoint to confirm it's running, a search post request that

takes a query and returns the sorted list of publications, and various endpoints that collect user feedback.

To run the flask app locally, simply navigate to ./backend and run

```
python3 bert/scibert_express
```

3.2.2 Test files

There are various test files that may exist in different branches of the repository. They are similar in implementation, however. They generally involve the following:

1. Loading a pretrained model.
2. Loading the corpus of publication-related texts (e.g. all titles, abstracts, etc.).
3. Taking test file from standard input.
4. Import embeddings from a text file.
5. Create an FAISS index.
6. Get the embedding of a query.
7. Calculate the distances from each publication embedding to the embedding of the query.
8. Find the target through the sorted publications and return its rank.
9. Repeat last three steps for other queries.

An example of running the file is:

```
cat text_inputs/test.json | python3 bert/scibert_titles_test.py
```

This will return a list of the ranks of each of the targets in the test.json file.

The publication-related embeddings are generated via the create_embedding files. Here is the general code of the files:

1. Loading a pretrained model.
2. Loading the corpus of publication-related texts (e.g. all titles, abstracts, etc.).
3. Calculate the embedding for each of the items in the text corpus.
4. return the embeddings

An example of running the file is:

```
python3 bert/scibert_create_embeddings.py > title_embeddings.txt
```

The test input files consist of two json lists. The first list is a list of queries, and the second is a list of targets. It's crucial that each target corresponds to its query by having the same index.

3.2.3 Utilities

The utility functions can be found in the utilities directory. Here are the two most notable ones.

1. stats.py can be used to turn the output of the test programs into a form more interpretable, providing the mean, median, and other metrics. To use, simply pipe the output of test programs into the stats.py file.
2. create_corpus.py creates a corpus by scanning other directories and gathering desired text from each of the publications (title, abstract...)

3.3 The Frontend

This section contains the user interface for the running model, as well as features that allow sending feedback.

The frontend is a react application with default CSS. Deploying the app is explained in the frontend readme markdown file.

3.3.1 Tokens

Feedback collection creates an interest in gathering information about the users. As a quick solution to this problem, instead of implementing SSO, a simple system was created. Users of interest can be sent a custom URL: https://cms-semantic-search.app.cern.ch/<token>. This URL redirects them to https://cms-semantic-search.app.cern.ch/token/<token>. Any feedback collected from this route will also save the token, allowing the developer to keep track of user-specific feedback. Normal users may be sent to https://cms-semantic-search.app.cern.ch/public

3.3.2 Redirect Work-around

https://cms-semantic-search.app.cern.ch/<token>is in fact a backend endpoint. All it does is redirect to https://cms-semantic-search.app.cern.ch/token/<token>, which is the front end route that contains the search functionality and UI. When users access the backend endpoint for the first time, they might be asked to authorize the certificate. Otherwise, the front end application may not be able to receive a response from the backend server.