S.E.M. Readme
COSI132A Final Project

Samantha Richards, Emily Fountain Molly Moran

May 12, 2020

Description

We've titled our project the Subject Equivalency Measure, or S.E.M. As the name suggests, our project is focused around implementing a system which would allow researchers to quickly find related papers. Additionally, we did a lot of data cleaning, made functionality for researchers to search via the CORD-NER entity annotations, and added several useful UI elements to facilitate easy searching.

Group members

Samantha Richards, Emily Fountain, and Molly Moran

Code submitted by

Emily Fountain

Dependencies

First, there are a handful of files that are necessary to run this code. The first three files we provide in our submission; the fourth requires some work from the individual building the index.

- The CORD-NER list of entities as provided by the University of Illinois research group (a json file). [1]
- The metadata file that CORD-NER is based off of (a csv file). This metadata file is different from the current metadata file that's in the CORD-19 dataset. The reason why this file is required is that the CORD-NER entity list does not hold all the necessary indexing information on its own; it relies on the ordering of the metadata in this file.
- A file which we have created that cross references the entities from the CORD-NER list with the actual items from the metadata file (a json file).
- The CORD-19 data. This data will need to be laid out in a different way from it is in the original Kaggle download.

  You only have to lay out the data in this different way if you want to create the index from scratch. As outlined in the "Build" section of our paper, we've given you the files you need to run our `query.py` function without using your own version of the data.

  If you want to create the index from scratch, create a directory that only houses files which hold data you'd like to be indexed; you'll pass this to our index build method. (Although doing this may be annoying, it is necessary, since Kaggle regularly changes how the data looks when downloaded).

Second, there are several software packages that are necessary to run our code. We have enumerated them in `requirements.txt` for ease of installation, but for completeness, we will also list them here:

```
certifi==2019.11.28, chardet==3.0.4, Click==7.0, elasticsearch==7.5.0,
elasticsearch-dsl==7.1.0, Flask==1.1.1, html2text==2020.1.16, idna==2.8, itsdangerous==1.1.0,
Jinja2==2.11.1, lxml==4.4.2, MarkupSafe==1.1.1, mkl-fft==1.0.15, mkl-random==1.1.0,
mkl-service==2.3.0, nltk==3.4.5, numpy==1.18.1, pycurl==7.43.0.4, python-dateutil==2.8.1,
requests==2.22.0, six==1.14.0, urllib3==1.25.8, Werkzeug==1.0.0, Wikipedia-API==0.5.3,
wptools==0.4.17, networkx, langid, pandas, jsonlines
```

Build instructions

After starting ElasticSearch on your machine, for the quickest set up with querying, you can run `quick_startup.sh`, which will set up the module, build the index, and start up the web search. For the script to work, you will need to have your data in a directory called `data` one level below `cord_19_ems`.

To do this without the startup script, do the following:

First, navigate to the `cord-19-ems` outer directory and run

```
$ python setup.py develop
```

which will set up all the dependencies and the necessary folder structure. Then run

```
$ pip install -r requirements.txt
```

to be sure all the requirements have been downloaded. Next, navigate to `cord_19_ems/es_module`. From here, call

```
$ python index.py --index_name=INDEX NAME --data_dir=PATH TO DATA DIRECTORY
```

Please note: If you are calling `index.py` from *anywhere* else in the project structure, you can still do so, but for added safety you will also need to pass in the path to `es_module` and the paths of three files included in `data_extras` (a metadata file from 3-13-2020, the NER file downloaded from the University of Illinois CORD-NER project, and the output file generated from these first two). You can check the bash script quick_startup.sh for a guide to how to formulate these arguments.

Finally, in order to start up web search, run

```
$ python query.py --index_name=INDEX NAME
```

with the name matching the name you passed in when you created the index. Then go to `http://127.0.0.1:5000` in your web browser to start searching.

Functionalities

In this section, we will try to enumerate all the functionalities that we implemented in our search engine. This includes UI improvements and work done on the corpus itself.

Results page

- The entities that are in each paper are listed, and are clickable so a researcher can quickly find other papers which also have that entity in them.
- "Find more articles" button. This will either find articles with similar citations, or articles with similar topics. This button returns a results page that is ranked by similarity in the chosen aspect.

Details page

- Sections of the paper are listed on the sidebar and are clickable.
- Provided a list of where this paper was cited. We also provide some context from the referencing paper and have clickable links to the referencing paper.
- References that link to other papers that are in the corpus are clickable.
- Visual UI improvements and addition of more data in easier-to-read format.

Other

- Articles not in English can now be removed from or included in search results easily.

Modules

Index

Index.py builds the ElasticSearch index from the CORD-19 dataset, cross-referencing the raw data with several additional metadata sources in the process.

Overview of Actions in Index.py

- Loads several crucial pieces of metadata with respect to the corpus: a citation graph (for generating pagerank scores), a dictionary of anchor text associated with each article, and counts of entities occurring in each article.
- Iterates through each document in the corpus and fills out the appropriate metadata fields.

"Article" Object and Fields

Our search index is made up of custom Article objects, each with the following fields:

- id_num: text field, a unique identifier for the document
- authors: a Nested list of Name objects
- title: text field, title of the article
- abstract: text field, abstract of the article
- citations: Nested list of Citation objects
- pr: float field, pagerank score of article
- cited_by: a Nested list of AnchorText objects
- anchor_text: text surrounding citations of this article in the corpus
- publish_time: int field, year of publication
- ents: text field, list of entities in the article
- in_english: boolean field, whether the article is in English

Details on Custom Object Fields

- Authors: Authors are indexed as custom objects with a "first" and "last" name field, for more advanced searching.
- Body: The "Body" text field is indexed as a list of custom "Section" objects. Each "Section" has a name and corresponding text. (i.e. "Results", "Discussion", etc.) This allows for easier display and navigation of article sections on the document display page.
- AnchorText: Each article is indexed with an "anchor_text" field, corresponding to the inbound links to the paper. The field is a list of AnchorText objects. AnchorText objects contain an id number (for fast retrieval of the article citing it) and a text field with the raw anchor text. (Note that the in-bound links only correspond to in-corpus articles.)

- Citation: Each article is also indexed with a "citations" field, which corresponds to the outbound links of the paper. Note that the outbound links may contain papers not in the corpus. Each Citation object is associated with metadata including authors, title and year of publication, which are used for display purposes. They are also indexed with a numeric field called "in_corpus", which determines whether the cited article is in the current corpus. If so, our interface ensures that the citation links directly to the corresponding article page.

Query

`query.py` starts up the web interface that we can use to query over our index.

Overview of actions in `query.py`

- Starts up and controls the web application
- After a searcher types in a query, searches over the index and returns ranked responses
- When a searcher finds an article they'd like to know more about, lets them to navigate to a page where they can read the paper
- Allows a searcher to do "more like this" queries, also ranked

Methods of interest

- `main()`: Starts up web application.
- `results()`: The powerhouse of the `query.py` module. This method renders results pages for both normal search and "more like this" search.
- `more_like_this_ents()`: Queries for papers with similar entities, then renders a results page.
- `more_like_this()`: Queries for papers with similar citations, then renders a results page.
- `filter_for_authors()`: Filters an existing search object for documents that match the author query.

More about `results()`

As mentioned above, `results()` does most of the heavy-lifting in our query system. First, the method finds out whether this query is a normal query or a "more like this" query. Depending on the answer, the type of page that is returned varies. If the query is of the "more like this" type, the method sends the relevant information to the `more_like_this` methods, which query, rank, and render the results page. If the query is instead a standard search, the method will query over the index (either conjunctive or disjunctive, depending on the user input). It also queries language, publish time, and authors at this point. Finally, after the method has retrieved the matching documents, it renders the results in ranked order.

Extras

`extras.py` contains all the helper functions used to build the index, called from `index.py`. This includes `all_ner_metadata_cross_reference()`, the function which cross-references entities against article ids, generating a json that maps article ids (shas) to entities found within those articles, `load_dataset_to_dict()`, which loads json files from the data directory into a single python dictionary, `filter_entities()`, which performs post-filtering of entities to remove likely false positives, `get_anchor_text()`, which extras text peripheral to citations of within-corpus articles, `get_entity_counts`, which collects the counts of all entities so that they can be further post-filtered based on frequency, `generate_citation_graph`, which creates a graphical representation of the citations in the dataset, `get_year()`, which uses a regex to pull just the year from a date string, and `untokenize()`, which takes a list of entities, joins any multi-word entities with underscores, and joins the entire list with spaces so that it can be indexed as a text field while preserving the original tokenization scheme.

Example queries

In the spirit of the CORD-19 task we originally took on ("What do we know about virus genetics, origin, and evolution?"), our test queries are based around knowledge needs for this task.

Let's consider the following information need. "I want to get information about livestock and animals carrying disease". I'll type in "animal host livestock" and choose conjunctive search.

From here, I find several useful papers. I'll read "A Strategy To Estimate Unknown Viral Diversity in Mammals". From here, I can see the topics (entities) that occur in this paper. This could help me understand more about what's in the paper. If I find something I'm interested in, I can click on it and be taken to a search page with that entity as the query (in this case, maybe "fecal contamination").

If I find this paper particularly helpful, I can use the "Find more articles" button to search for more articles that have similar citations or similar topics. Similarly, if I click on the paper, I can see in the sidebar "See this article cited in context", where I can look at other articles which cite this paper, and if I want to, click on them.

Finally, I can look at the references portion of this paper and if there are papers listed there that are also in the corpus, I can click on a link to take me to them.

References

[1] Xuan Wang et al. "Comprehensive Named Entity Recognition on CORD-19 with Distant or Weak Supervision". In: (2020). arXiv: `2003.12218 [cs.CL]`.