**IR Competition**

**Description**

This is the final course project for ***CS 410 - Text Retrieval & Mining*** course at the University of Illinois at Urbana-Champaign.

For the project our team decided to participate in the CORD-19 Open Research Dataset IR Competition. The CORD-19 dataset contains over 57000 scholarly articles available for the global research community. Our goal is to build a live system that supports search on this massive dataset.

**Project Team:**

Nathaniel Rupsis (NetID: nrupsis2) – Team Captain

Hoa Le (NetID: hle30)

**Video Presentation**

Link to our video presentation for this project can be found below: <https://mediaspace.illinois.edu/media/t/1_cgdhmw3n>

**Prerequisites**

**Packages**

* Metapy

We use Metapy to build and evaluate our search engine.

# Ensure your pip is up to date

pip install --upgrade pip

# install metapy!

pip install metapy pytoml

* argparse

The argparse module makes it easy to write user friendly command line interfaces.

argparse should work on Python 2.3 and newer.

# Try one of the following installation methods:

pip install argparse

python setup.py install

easy\_install argparse

putting argparse.py in some directory listed in sys.path should also work

**Running the program**

Directory:

| - /test

\- line.toml

... data

| - /train

\- line.toml

... data

A test and train directory need to exist with a line.toml file which specifies the metaData

// line.toml

type = "line-corpus"

metadata = [{name = "uid", type = "string"},

{name = "title", type = "string"},

{name = "abstract", type = "string"},

{name = "introduction", type = "string"}]

To build the index, simply run:

// remove the index else the evaluator will use a cached version

rm -rf idx/ && python3 build\_courpse.py train

where the first argument is either "test" or "train" (need to have corresponding test / train files for MetaPy to work)

To run the evaluator:

python search\_eval.py config\_train.toml

where the config argument is either config\_train.toml or config\_test.toml

**File & Code Walkthrough**

**search\_eval.py**

This is our main file containing most of the crucial functions used for creating ranking algorithms and searching an index as well as evaluate and writing the results.

Below is the list of functions in the search\_eval.py file:

expand\_query(query):

* Description: This function is a simple approach to query expansion. It expands on some important keywords such as covid-19, coronavirus, test, mark, spread, etc… It also contains some pre-processing methods such as removing punctuation, converting to lower case, filtering out common words, etc…
* Parameter: query
* What the function returns: return list of query words containing new keywords.

load\_queries():

* Description: This function retrieves query from the xml file.
* Parameter: None
* What the function returns: return tuple array of queryID and query pairs.

load\_ranker(cfg\_file):

* Description: This function returns the Ranker object to evaluate
* Parameter: cfg\_file is the path to a configuration file used to load the index.
* What the function returns: Ranker object.

saveResults(prediction\_results):

* Description: This function writes the results to “predictions.txt” file.
* Parameter: prediction\_results (a list of 3 elements returned from running ranking algorithms)
* What the function returns: return list of query words containing new keywords.

runQueries(queries)

* Description: This is our main function for creating inverted index and ranking algorithms.
* Parameter: it takes the output which is the query from load\_queries function as input.
* What the function returns: The function saveResults mentioned above is called to store the output.

**build\_courpse.py**

This function contains code to extract information such as title, abstract, introduction, etc… from metadata.csv using example code. It also writes the data to .dat file and create metadata file for metapy.

**config.toml**

We have two separate config file : config\_test and config\_train. Each file contains general settings that deal with paths related to the its corresponding dataset. We specified some details about the corpus and its analyzer. For example:

dataset = "train" # name of the dataset

corpus = "line.toml" # type of corpus

query-judgements = "qrels.txt" # path to query judgement file

[[analyzers]]

method = "ngram-word" # set of co-occurring words

ngram = 1 # specified to one word aka unigram

filter = "default-unigram-chain"

**stopwords.txt**

A text file contains list of common English words that do not add much meaning to a sentence. For example: “the”,”a”,”an”,etc…They can be safely ignored, and this is part of pre-processing to filter out useless information.

**train & test folders**

Contain the cord-19 dataset and metadata.csv file. The train data also contains query relevance file.

**predictions.txt**

This is a text file that has the same format as relevance judgment file which is (query\_ID doc\_uid relevance\_score). It contains our final top 1000 documents per query and is used for computing nDCG score on the leaderboard.

**Implementation Overview**

**Task Definitions**

1. Collecting data
   * 1. Unzip and extract data from the provided train.zip
     2. Collect and store information from metadata.csv such as title, abstract, introduction, etc…
2. Building a corpus
   * 1. Create train/test.dat file. Each line in .dat file represents one document.
     2. Create metadata.dat file containing the collected information from step 1 part ii.
     3. Create line.toml configuration file containing corpus input formats.
3. Implementing simple query expansion
   * 1. Collect query from xml file. Query are stored in a tuple array (queryID, query)
     2. Create query expansion function with some preprocessing methods.
4. Create an inverted index
   * 1. Set up the config file. This file contains fields that need to be specified and makes references/pathing to several files.
     2. Write code to create inverted index.
5. Searching the index
   * 1. Create bm25 ranker.
     2. Run query to search the index and returns top 1000 documents per query as sorted vector pairs (doc\_id, double).
     3. Store the query-ids, results and the corresponding uid (document\_id retrieved from metadata file) and write to predictions.txt file.
6. Optimize nDCG score
   * 1. Submit predictions.txt file and check the nDCG score on the leaderboard. Address potential problems and identify methods to improve nDCG score.

**Potential Issues Encounter**

If UnicodeEncodeError: 'charmap' codec can't encode characters error appears, running python3 seems to fix that. One solution which we will add to our code is to write (encoding = “utf-8”) when reading .json or .dat file.

We ran into some disk problem when implementing BERT-large method for reranking. We decided to exclude it from our final code to avoid this potential issue.

**Key Findings & Potential Improvements**

Even though we made it to leaderboard. The result was not what we expected. Below is what we identify that might help with improving the result:

1. Add more information to the index (actual documents)
2. Weight the keywords better, since relevancy will definitely be based on the keywords (covid, alcohol, spread, African American, health, etc…)
3. Further expanding on query. Query expansion has helped enhancing our result greatly.
4. Implementing BERT large for reranking. We ran into disk problems when trying to implement it.

**Contact**

[nrupsis@gmail.com](mailto:nrupsis@gmail.com)

[hle027@gmail.com](mailto:hle027@gmail.com)