“*HW1 - Search Engine*”

CSE 272: Information Retrieval

Spring 2023

Name

Rohan Ghosalkar (rghosalk@ucsc.edu)

SID: 2005624

Instructor

Yi Zhang

Link to GitHub Repository: https://github.com/rohan527/cse272_HW1_search-engine

Table of Contents

Introduction

In this assignment, we have to create a search engine or, in other words, do batch retrieval on a dataset (OHSU88-91). To do this, we first create an inverted index on the dataset. This is followed by using algorithms such as Boolean Algorithm, Term Frequency (tf), Term Frequency - Inverse Document Frequency (td-idf), and a custom algorithm we create ourselves. Finally, we create a log of our results which contains rank, document id, and a score of the top 50 documents that were retrieved.

Software

For data processing, we don't use any special software software. Since I use Mac OS, I thought it better not to use Lucene (also because I am not familiar with Java). Instead, for indexing and searching purposes, I used Whoosh. The Whoosh Query Parser software is flexible and can be customized to handle specific query types or data formats. I have included a link to the Whoosh software in this section. The main difference between Whoosh and Lucene is that the previous one is written in Python, and the latter is in Java. Therefore, it was more flexible and more accessible for me to use Whoosh than Lucene. Other than this, we also use Numpy and Pandas for data handling whenever necessary.

Architecture/Data Structures

My entire code is written in a colab notebook. However, I have also added a Python file that can be used to execute the code on a local machine. In the colab notebook, we first install and import the necessary libraries. This is followed by dataset/query parsing. After this, we introduce the function for each search algorithm, and in the main code, we call these functions and generate our log files.

Major Issues

The primary issue I faced was understanding how to use the Whoosh software. Data parsing and preprocessing were pretty straightforward since we could use the NLTK sources for removing stopwords and cleaning the data of redundant information. I was also able to understand how to do this when trying to use the Whoosh library, as it allowed us to use the uncleaned data and then remove the stopwords etc., while we input the data in the search.

Data Parsing

The input dataset includes two types of ID (.I & .U), source (.S), mesh terms (.M), title (.T), abstract (.W), publication type (.P), and authors (.A). In comparison, the query dataset (query.ohsu.1-63) contains queries that have been included between <top> and </top>, the <num> section which describes the corresponding number of disease codes, the <title> section which includes its title, and the <desc> section that contains either one question or a detailed description.

Dataset Parsing

The dataset is parsed as we call the function with the search algorithms. The read_documents function parses the dataset and returns every document in a dictionary with the document details (ID, abstract, authors, etc.). This is used to create an inverted index of the document. The working of the create index function is written below.

Query Parsing

The query data is first converted into a list called queries which contains a dictionary with keys {'num' , 'title' , 'description' }. This list is later used when we try to use the search algorithms.

Algorithms

- boolean

Boolean search or inverted index search helps us identify the document's relevancy. After we have indexed all of our documents, we do an inverted index search based on either AND/OR/NOT. The output is a match based on the search terms and how much of a match it is with the corpus or input set.

- tf

Term frequency (TF) search is a technique used by search engines to rank search results based on the frequency of the search terms in each document. The idea behind TF search is that documents containing a higher frequency of search terms will likely be more relevant to the user's query.

- tf-idf

Term frequency-inverse document frequency (TF-IDF) search is a technique used by search engines to rank search results based on the relevance of the search terms to the documents in a collection. TF-IDF search considers the frequency of the search terms in each document and the rarity of the search terms across the entire collection.

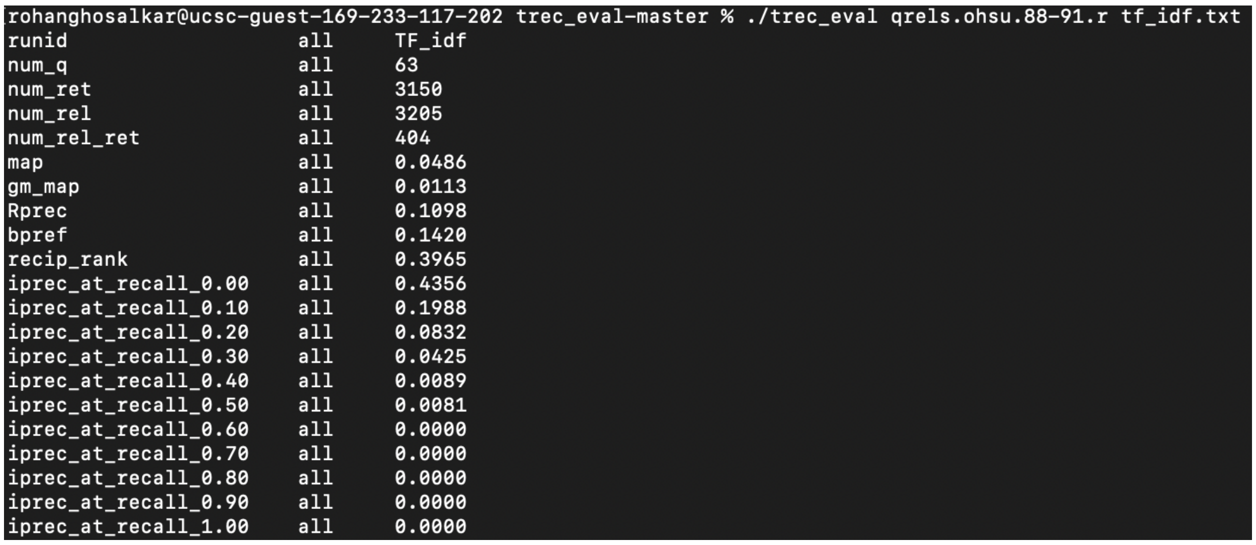
Custom Algorithm

The score is defined as the sum of the product of the weight and the number of occurrences of each query term in the title and abstract fields, plus a constant factor. Weights are assigned based on the field: 5 for the title field, and 2 for the abstract field. The maximum weight is used if a query term occurs in both document fields.

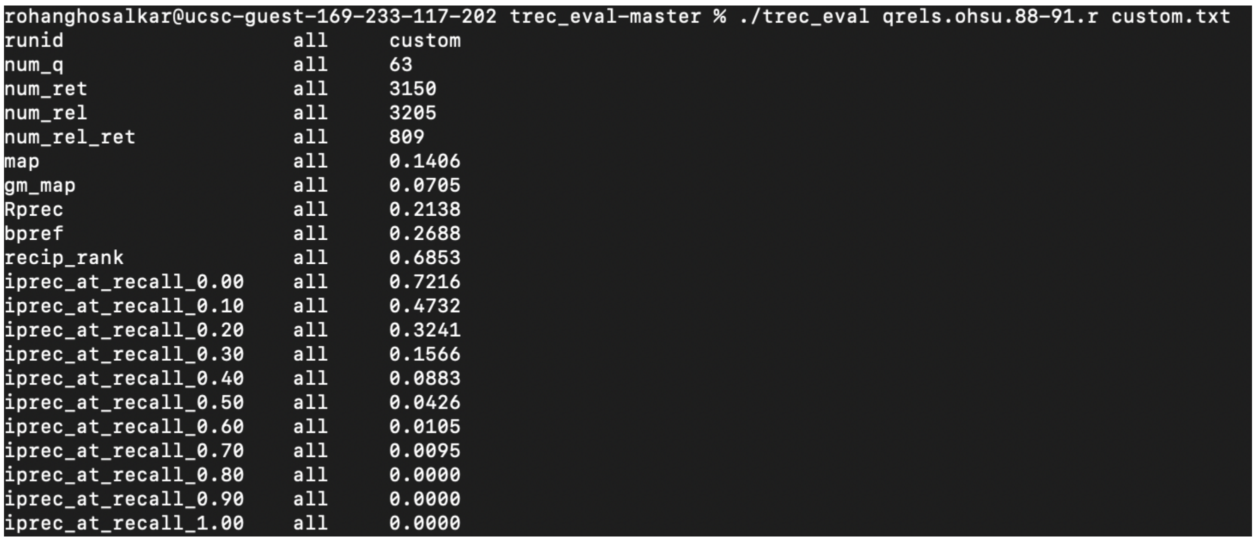
In the code, we define a custom scoring function, custom_score_fn, that calculates the score for each document based on the number of occurrences of each query term in the title and abstract fields. We assign weights based on the field and add a constant factor to the final score. We then use this function with ix.searcher(weighting=custom_score_fn) as s: line to apply it during the search.

Experimental Results

I found that the tf-idf search algorithm worked better than the other methods. Since the tf-idf search module is already in the Whoosh library, I used it. My custom algorithm worked better than expected, giving good results too.



```
rohanghosalkar@ucsc-guest-169-233-117-202 trec_eval-master % ./trec_eval qrels.ohsu.88-91.r tf_idf.txt
runid                   all     TF_idf
num_q                   all     63
num_ret                 all     3150
num_rel                 all     3205
num_rel_ret             all     404
map                     all     0.0486
gm_map                  all     0.0113
Rprec                   all     0.1098
bpref                   all     0.1420
recip_rank              all     0.3965
iprec_at_recall_0.00    all     0.4356
iprec_at_recall_0.10    all     0.1988
iprec_at_recall_0.20    all     0.0832
iprec_at_recall_0.30    all     0.0425
iprec_at_recall_0.40    all     0.0089
iprec_at_recall_0.50    all     0.0081
iprec_at_recall_0.60    all     0.0000
iprec_at_recall_0.70    all     0.0000
iprec_at_recall_0.80    all     0.0000
iprec_at_recall_0.90    all     0.0000
iprec_at_recall_1.00    all     0.0000
```



```
rohanghosalkar@ucsc-guest-169-233-117-202 trec_eval-master % ./trec_eval qrels.ohsu.88-91.r custom.txt
runid                   all     custom
num_q                   all     63
num_ret                 all     3150
num_rel                 all     3205
num_rel_ret             all     809
map                     all     0.1406
gm_map                  all     0.0705
Rprec                   all     0.2138
bpref                   all     0.2688
recip_rank              all     0.6853
iprec_at_recall_0.00    all     0.7216
iprec_at_recall_0.10    all     0.4732
iprec_at_recall_0.20    all     0.3241
iprec_at_recall_0.30    all     0.1566
iprec_at_recall_0.40    all     0.0883
iprec_at_recall_0.50    all     0.0426
iprec_at_recall_0.60    all     0.0105
iprec_at_recall_0.70    all     0.0095
iprec_at_recall_0.80    all     0.0000
iprec_at_recall_0.90    all     0.0000
iprec_at_recall_1.00    all     0.0000
```

Conclusion

In conclusion, I have learned the fundamentals of search algorithms and search engines. I have also learned how to implement boolean, tf, and tf-idf search algorithms for document retrieval. In essence, I feel that this is quite useful considering the importance of document retrieval and search engines these days with the rise of AI-based conversational models and large language models, which will require finding the correct response to user data.