SEARCHIVE

TTDS coursework 3 - Search Engine

Lawrence He - s1659436
Moy Yuan - s1652665
Ling Lu - s1926829
Jiening Li - s1891132
Luqi Li - s1743112

Contents

1 Introduction

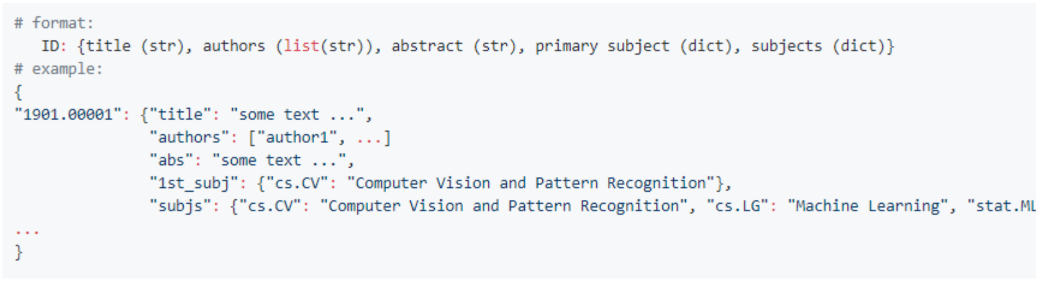
Our search engine product, **Searchive**, is specially designed for scholar search. We collect our data from the open-access archive (arXiv) of 1,657,730 scholarly articles [1]. In this report, we will explain our implementation of the search engine in several parts: *Data Collection, Preprocessing, Indexing, Searching, Ranking* and *Server*. Furthermore, we will specify the individual work and contributions.

The motivation behind this project is that arXiv is an excellent archive of the scholar articles but its on-site search does not perform well. We want to create an efficient and multi-featured scholar search for these valuable scholar data.

2 Data Collection

The scholar articles in arXiv have 8 categories and each one has several sub-domains. For instance, the class *Computer Science* has a sub-class *Computation and Language* with the corresponding tag, *cs.CL*. Each paper is identified by a unique ID with the format of `yymm.xxxxx`, e.g. the ID `2002.06191` means that this paper was submitted on Feb 2020, and it is the 6191st paper to be submitted that month.

We use `scrapy`[2] (based on Python) to build the web crawler, gathering data from arXiv export, and storing it in the local database. Each data entry has a document ID as the `key`, and a `dictionary` that contains the corresponding **title, author(s), abstract, primary subject** and **secondary subject(s)** as `value` (see an example below).



```
# format:
   ID: {title (str), authors (list(str)), abstract (str), primary subject (dict), subjects (dict)}
# example:
{
"1901.00001": {"title": "some text ...",
               "authors": ["author1", ...]
               "abs": "some text ...",
               "1st_subj": {"cs.CV": "Computer Vision and Pattern Recognition"},
               "subjs": {"cs.CV": "Computer Vision and Pattern Recognition", "cs.LG": "Machine Learning", "stat.ML
...
}
```

Moreover, we do not store the link to the submitted pdf because it can be reconstructed from the document ID. The data points with the same ID prefix (i.e. `yymm`) are kept in a single `json` file. Each `json` file contains approximately 11,000 entries on average. Because of the limited computational power, we decide to use only one million data points.

The primary assumption of our work is that the abstracts of the scholar articles are highly informative. Therefore, we do not need to parse all the content of the paper. And in fact, words in non-abstract sections may be superfluous to the search request. This decision makes our searching results more accurate and our data storage lighter.

To keep live data storage, we set up two web crawlers that work continuously. We use the first crawler to

[1]According to the statistics from the website: https://arxiv.org/
[2]Available at: https://scrapy.org/

collect new data, while the second to collect the old data (re-submissions of the papers are allowed in arXiv). The collection is saved in temporary folder and we run the update daily. The indexing is then updated correspondingly.

3 Preprocessing

As mentioned in section 2, the scrapped data contain various attributes. For the diversity of the search choices, we create separate indexing (see section 4) for each attribute except that the contents of abstract and subjects are combined as a whole. The preprocessing is done with a `Normaliser` class, where we have the following cases that use different normalisation methods:

- Abstract + Subjects: Some abstracts contain formulas in Latex format that is largely irrelevant to the search query. We remove anything between two \$ symbols as to clean the Latex text. Next, we apply the routine of "Tokenisation → Casing (lower) → Stopping → Stemming" to generate the final tokens.
- Title: We do the same as Abtract + Subjects except that we do need to clean the Latex content in advance.
- Authors: Here the Stopping and Stemming are removed from the routine because there is no point to do this for people's names.

The tokenisation method we use is simply looking for every word character (English alphabets and numbers) in the text because the valid part of the search query mostly contains the word characters only. We use the stop-word list from `nltk.corpus` and the English stemmer from `nltk.stem.snowball`.

4 Indexing

4.1 Initialising index

We are using positional inverted index in this project since the system allows query search and phrase search. For each preprocessed term, we recorded its document frequency (df), term frequency (tf) and also every existing position in each document containing the term. Compared to cw1, we made an advancement in data structure for the index to improve the search speed: we are now using two separate types of dictionaries to store the index information. The first type of dictionary stores the following information:

```
term:
    df: xx (df)
    docID1: xx (tf in doc1)
    docID2: xx (tf in doc2)
    ...
```

The second type of dictionary stores the following information:

```
term:
    docID: [pos1, pos2, ...]
    docID: [pos1, pos2, ...]
    ...
```

What we are using in cw1:

```
term:
    df: xx (df)
    docID1: tf: xx (tf in doc1), poslist: [pos1, pos2, ...]
    docID2: tf: xx (tf in doc2), poslist: [pos1, pos2, ...]
    ...
```

We are using two separate dictionaries here rather than just using one in cw1 because the position information is only needed in phrase search for the specific document containing the term. While doing normal query search, we only need tf and df for the specific document containing the term. So the first dictionary is used for query search and the second dictionary is used for phrase search specifically. In this way, the process of querying the dictionary to retrieve tf, df and positions will speed up with a simpler dictionary structure.

Since the system supports querying authors, titles, and abstracts of the articles respectively, we created index respectively for each of the three fields. [3] Also, an extra dictionary will record parameters needed for BM25 (see 6.1 part) which are total number of document, total length of all abstracts, total length of all titles, total length of all authors. So we have seven dictionaries in memory in total after initialising.

4.2 Saving and reading index

Each dictionary listed above is saved into a pickle file respectively rather than json file because pickle file can maintain datatype such as integer, while loading json file back in memory will return everythin as string which needs further processing. We considered pickle more convenient so we used pickle as the saving file format. Everytime index is needed, the system will read the pickle file into the memory as a dictionary.

We have also implemented methods to save the index to a cloud database (Google Firebase) and a local database (MongoDB) and query the index given a word from them, but we considered saving the index data in pickle files and read them back into dictionaries in memory for query while searching as the most efficient way after experiments and comparisons.

4.3 Updating index

Compared to cw1 which used fixed collections and fixed index, we improved our system by enabling updating index while some documents in the collection have been changed. Once a document has changed, we will

[3] The subject information of the article is added at the end of abstracts.

read in the original index files, the original document and the new document. We will delete all the positions recorded under each term in the original document and then add in all the positions under each term in the new document in the second type of dictionaries mentioned above (see 4.1). Tf and df will be updated during this process in the first type of dictionaries mentioned above (see 4.1). Also, total length of all abstracts, total length of all titles, total length of all authors will be updated since the length of the documents might have some changes.

5 Searching

5.1 Implementation

In search module, it allows query search and phrase search without the limitation of word length which is improved compared to CW1. The search module implemented the function of returning the document IDs that match the search query entered by the user. There are 4 search modes, **abstract**, **title**, **author**, and **general** search.

The first step of search module is to determine the search mode. In the `mode_select()` method, there are two categories. One is the independent search for **abstract**, **title** or **author**. In this type of search mode, determine whether the input query is a normal query or a phrasal query with quotation marks. The normal query search uses the `query_search()` method and the phrasal search uses the `phrase_search()` method. The other is the **general** search, which searches for relevant document IDs from title, author, and abstract collectively. The general search mode also needs to determine the type of the input query (normal or prhasal). The results of the general search are the union of the three independent search modes.

The next step is pre-processing the search query, which is same as the pre-processing in Indexing. If the search mode is author, we used a specific normalisation method for author, which is different from other modes (see section 3). After pre-processing, we get the cleaned tokens that can be used to find relevant document IDs from the index.

5.2 Optimization

To reduce the search time, we create different indexing dictionary formats for the two search modes. For the normal query search, we use the truncated indexing (to save memory) that does not include the corresponding positions for each token(see 4.1). For the phrasal search, we use the original indexing (see 4.1 for two types of the dictionary formats).

We implement `query_search()` function by taking the union containing the document IDs for each term in search query and return the list of document IDs. If there is no result that meets the requirement, return None.

For `phrase_search()` function, firstly, determine whether the first term and last term of the phrase are in the same document at the same time. If no such document exists, simply return None to reduce search time. If the first word and the last word are in some documents, determine whether the distance between the two words is equal to the length of the query minus 1. If not, skip this document ID and proceed to the next file. Next, we need to compare the distance between the other terms in the query. If the distance between each subsequent term and the previous term in this query is 1, then store this document ID in the list `Docid_phrase` that will be returned and sent to the Ranking module.

6 Ranking

6.1 TF-IDF and BM25 Algorithms

Compared to coursework 1, we not only use the **tf-idf** [1] but also **bm25** [5] algorithms to calculate the importance score of query according to the documents, which provides users with multiple choices. tf-idf , short for *Term Frequency-Inverse Document Frequency*, and bm25 , short for *Best Matches 25*, are commonly used in information retrieval and text mining algorithms. In this project, these two algorithms are used to calculate the importance of query to different documents, so as to achieve the purpose of returning highly relevant documents to users.

For tf-idf, the algorithm mainly uses **TF** (normalized word frequency) and **IDF** (inverse document frequency) to calculate the weight of each term in query and then sum them up to get the final score; for bm25, the final score is determined by the weighted sum of the importance score between each term in the query and document. Among them, the method of calculating correlation involves 3 parameters, which are **k1**, **k2**, and **b**. According to [4], k1 is used to control the sensitivity of the score to tf. The smaller k1 is, the less sensitive it is to tf. b is used to control the degree of punishment of the document length weight: b is 0, the document length has no effect on the weight; b is 1, the document length has a full penalty effect on the weight. In this project, 2, 0.5 and 0.75, were selected for these parameters respectively.

We also used a new method which mixes tf-idf and bm25 by adding them with different weights. After testing with a small number of samples, it was finally determined that when the weights are 0.7 and 0.3 (see Table 2), user satisfaction was higher after evaluating the system by our group.

6.2 Implementation

we have 4 modes for searching, **general**, **author**, **title** and **abstract**. For each of them, we used the relative data which was saved in different dictionaries: **ab**, **au**, **ti** and **param** standing for abstract, author, title and parameter (used by bm25 algorithm to calculate the relevance). we also have 3 methods, tfidf, bm25 and mix, which uses tfidf algorithm, bm25 algorithm and sum of the result of tfidf and bm25 algorithms with different weights, respectively.

For ranking part, the corresponding python file is `ranking_local.py`. Functions, such as `mode_select()` and `preprocess_squery()`, are called from **Searching** part to get pre-processed term list from the query and the corresponding documents ID list of those terms. Other functions such as `read_index_file_no()` and `read_index_file()` are also called from **Indexing** part to load the dictionaries, **ab**, **au**, **ti** and **param**, and the dictionaries with term position, **ab_pos**, **au_pos**, **ti_pos** into the local memory. In `ranking_local.py`, `rank()` function is used to calculate the score under different modes and return the list of id-score pairs, and `search_for_detail()` function is used to return the ranked dictionaries with details of those documents under different modes. This program first gets the related documents list and cleaned query list from search part, and then, calculate and sorted the score of each document from related documents list. Finally, according to id of those documents, it returns the dictionary which contains the detail, such as abstract, subjects, etc..

6.3 Optimization

The speed of the query is one of the keys to the search system. In this part, three aspects were optimized to improve the speed of ranking. The first one is to reduce the number of file reads. By reading data into memory at the beginning of the program, the number of subsequent file reads is reduced. The second one is to limit the number of output files to 500, which reduces the search time for details. The last one is to merge the collections of documents into one json file (**merge.json**) and save it in local disks. Each time the program is started, the ranking time is greatly reduced by initializing merge.json file into memory, because the file is read only once.

7 Server

Implementing a nice interface for query submission and results display is indispensably for an exciting project. While terminal is technically capable of accepting queries and representing entries of documentations, we decided to build a website as our user interface. Based on our experience, we chose Flask[4] as our micro web framework because of its simplicity, community support and light-weight structure. Initially, we had limited resources for Database (Firebase) and Server (AWS EC2 with Mongo Database). Nevertheless, because of the limited free ram-usage, we eventually decided to run the server on local laptops thus website will be hosted on local-host. Our system can be host online if provided with server and domain. However for this coursework, we will demonstrate our system in person.

7.1 Web Page Structure

With the aids of HTML, CSS, and JavaScript we built a google-style web page that allows users to type their query and submit it for the results. The main CSS style is from bootstrap[5] and cloudflare for rendering Latex text[6]. To manage the templates, a layout template with a navigation bar and all other web-page templates extend the layout template. For the result web page specifically, a card style layout is applied to show individual document for the legibility and a side bar is provided with additional filters. In short, the philosophy of the design is to create a simple, clean look (screenshots of the website are included in the appendix page A B). While maintaining a stealth appearance, there are a good amount of features included for this project as we wish to offer as many choices as possible to the users.

7.2 Basic Features

The basic features are to enhance user's experience with our website.

- Users have the ability to customize what type of content they want to search for (e.g. abstract).

[4]https://flask.palletsprojects.com/en/1.1.x/
[5]https://www.bootstrapcdn.com
[6]https://www.cloudflare.com

- Users have the ability to customize what ranking metric they prefer (e.g. tfidf).
- Paging function is provided for better Easy of Use and the number of results per page can be selected.
- Advanced search with additional requirements such as sorting method (e.g. Newest to Oldest).

7.3 Auto-correction

Misspelling is a common issue when searching especially for non-native speakers. In addition to the built-in spellcheck plug-in in most web browsers, we implemented an auto-correction functionality with `pyspellchecker` [7]. Once the user submit a query, the spellchecker will firstly identify the misspelled word(s) and replace them with the correct spelling before pre-processing them in the next step of searching. By default, the search results of the corrected query are displayed and the users are notified with message about the misspelling. The auto-correction can also be disabled if it is a false alarm or the user simply insists to search the original query.

7.4 Language Model (deprecated)

A well-pretrained language model can be much more powerful than traditional rule-based methods. The calculation time, however, can be unsatisfying long without GPUs. In our case where the machines running the server does not provide industrial level of computation ability, we prioritised time complicity over functionality as information retrieval is all about speed. Nevertheless, a language model that captures information about semantics is worth mentioning. We used pytorch[8] to build a LSTM[2] based model with pre-train GloVe[3] embeddings. Our system is capable of predicting the most likely next word given the (partial) input query and finding semantically similar words that can be related to the users' input.

8 System Evaluation

8.1 Efficiency Testing

We have developed three generations of the system (by using different methods to load and save the index) and the efficiency is improved dramatically. We run a efficiency testing using the two-word, five-word and nine-word queries, and the search mode is "general search". We report the average time spent by the system in Table 1. The differences among the three generations are:

- First generation: All index saved in `mongodb` [9], a local database.
- Second generation: Index with term positions saved in `monbodb`; Index with `tf` and `df` saved in the local disk (see section 4 for the details of these two types of indexing).
- Third generation: All index saved in the local disk, and read in memory; Merge the data collection (`json` files) for faster retrieval.

[7] https://pypi.org/project/pyspellchecker
[8] https://pytorch.org/
[9] https://docs.mongodb.com/manual/

	2 words		5 words		9 words	
	query search	phrase search	query search	phrase search	query search	phrase search
1st gen.	30.98s	24.42s	87.23s	72.93s	182.64s	120.35s
2nd gen.	1.85s	14.34s	3.65s	60.56s	6.54s	95.47s
3rd gen.	1.76s	1.18s	3.54s	2.70s	6.49s	4.33s

Table 1: Time Usage

8.2 User Satisfaction

We invited five students to complete the evaluation of the search engine. The evaluation is divided into three aspects: user interface, content relevance (mix method) and search speed. The score is an integer from 1 to 10, where 1 means completely dissatisfied and 10 means completely satisfied. The table 2 shows the degree of satisfaction of each student in different aspects and the average score of each aspect.

score	UI	Relevance	Search Speed
Student 1	9	7	9
Student 2	7	7	9
Student 3	9	6	10
Student 4	8	8	10
Student 5	8	8	9
Average	8.2	7.2	9.6

Table 2: Scoreboard of UI, Relevance and Search Speed

9 Individual Contribution

9.1 Jiening Li

I was directly responsible for indexing. The work consists of creating index, effectively storing the index, and updating the index while some documents in the collection have changed.

- Creating index: The final version of indexer is stated above (see 4). Apart from that, I made different attempts to improve the efficiency and compress the size of indexer, including using delta encoding for index compression (which was found to need more time while initialising index and thus deprecated), and also re-designing the structure of the dictionary to increase the speed of ranking (see **??**) .

- Saving and reading index: I implemented four ways of saving index data which are (1) saving them in Google Cloud Firestore Database [10], (2) local MongoDB database on my own machine, (3) MongoDB database implemented on a AWS EC2 server [11] and (4) writing them to a local pickle file. I compared them and found that: the first method has limit on read/write requests for free each day which is not realistic for large data collection; the second method is very slow while initialising index, and querying from database is much slower than querying from dictionary in memory directly; the limit of the third method is that AWS EC2 only provides 1GB memory for free which is not enough for our data storage and hosting an web application. Therefore, I selected the most efficient and usable one which is the fourth

[10] https://firebase.google.com/docs/firestore
[11] https://aws.amazon.com/ec2/?nc1=h_ls

method (see 4.2 for implementation details). This method speed up the following searching and ranking module significantly. I also write the function to read the index pickle file back into memory and return the doc id and relevant information needed given a word in the query. I spent a lot of time learning to use MongoDB and AWS EC2 server here.

- Updating index: an advancement compared to cw1. See 4.3 for implementation details.
- Others: I wrote the report for the indexing part.

9.2 Luqi Li

I was responsible for the implementation of the search module and the searching part of this report. I started with query search. Compared to CW1, this query_search() method did not limit the length of user's input search query. In CW1, phrase search only has a length of 2 query, so the implementation is simple. However, in practice, we can not limit the length of the user's query when they search, so I expanded the length of phrase search from 2 to n. The detailed implementation of the code has been written in the report. I modified search modules based on the different index storage modes(see 9.1 part). The reason is that we need to reduce the search time, we tried several different index storage methods and dictionary structures. The time to run the search module was reduced from around 1s to 0.06s. In the process of testing, we also found some bugs. I need to consider some situation where the term entered by the user may not exist at index, the index may not exist after update index, or the phrase may not exist.

9.3 Ling Lu

I was mainly responsible for writing codes and report of the ranking part. Compared to coursework 1, I added the bm25 algorithm and mixed it with tf-idf algorithm, which gives users multiple choices. The entire ranking code has undergone three generations of changes with the change in the way the index is stored (see 9.1 part). These changes are based on improving the query speed. All documents collections are merged into one file called merge.json and saved in local disks. Under the joint improvement of the index storage method, data reading and saving method, the speed of the ranking part changed from 30 seconds to 0.6 seconds. I also assisted in the integration of the code later and the test of the amount of data.

9.4 Moy Yuan

My responsibility was to build the front and back end of the server from the web design to implementation. Since creating a web server was both new to this coursework and an unexplored area for all of us, I spent a significant amount of time learning from online tutorials. The major features of the website has been thoroughly introduced thus not repeated here. However, in terms of visuals and usability along with some miscellaneous will be better presented in the demonstration. Furthermore, as I am the only member in the group with access to machine that is capable of running the server because of the huge memory requirement, I was also responsible for most large scale testing of indexing, searching, ranking and the combined overall prototype web application. Lastly, the corresponding part of report was produced by me.

9.5 Lawrence He

My personal contribution is summarised as follows:

- Contact person and project manager: Decide the project's objectives and make plans for each of them. Construct the project pipeline and use the GitHub private repository to manage the version control. The README file in our repository is used for public announcement and timeline arrangement, this simple approach keeps the team integrated.
- Data collection: Learn the web crawler techniques and develop the code that collects the arXiv data (see section 2); Design the data structure for the scrapped data.
- Preprocessing: Write the code for preprocessing that meets the needs of every subsequent module (see section 3).
- Report: Create the structure of the report and write the relevant parts of my work.

Although each of us in the team is responsible for different modules in the pipeline, we actively interact with each other to ensure the integrity of the system. The ultimate system we submit is capable of handling a large data collection while maintaining the efficiency and effectiveness. This is an achievement of us as a whole.

References

[1] Lukáš Havrlant and Vladik Kreinovich. A simple probabilistic explanation of term frequency-inverse document frequency (tf-idf) heuristic (and variations motivated by this explanation). *International Journal of General Systems*, 46(1):27–36, 2017.

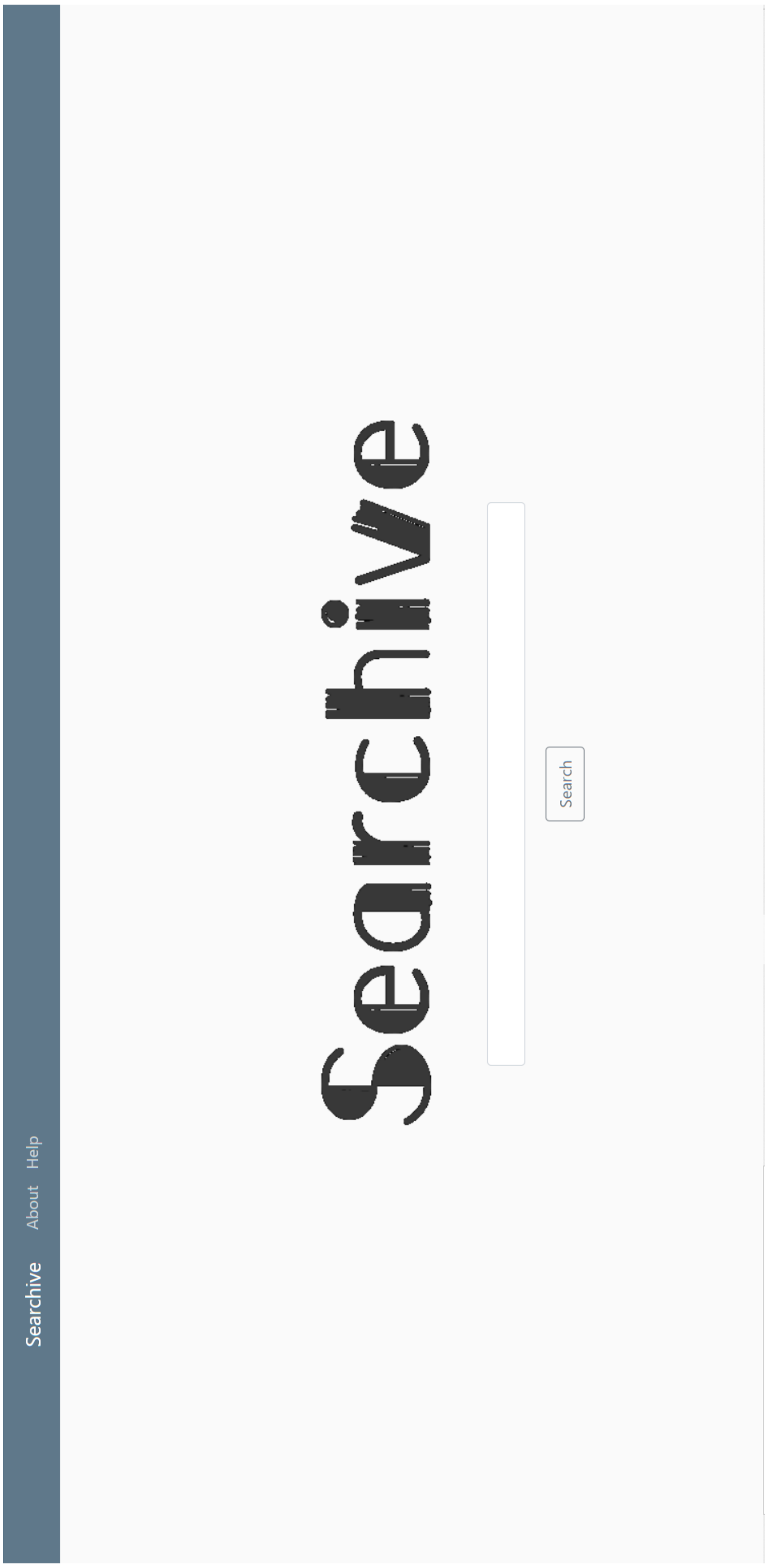
[2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[3] RichardSocher JeffreyPennington and ChristopherD Manning. Glove: Global vectors for word representation. Citeseer.

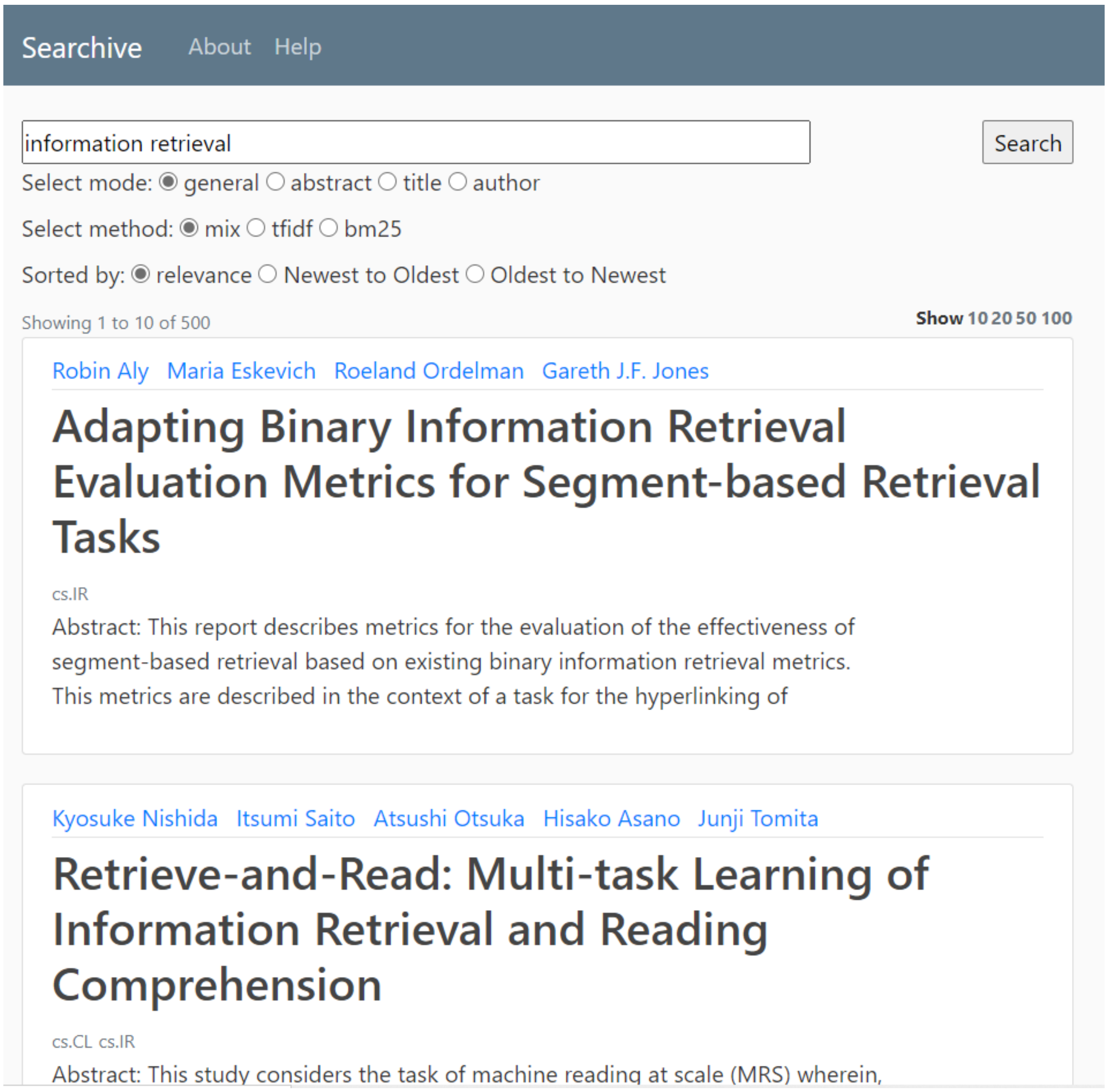
[4] Yuanhua Lv and ChengXiang Zhai. Adaptive term frequency normalization for bm25. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1985–1988, 2011.

[5] Xapian. The bm25 weighting scheme. https://xapian.org/docs/bm25.html. Accessed February 27, 2020.

Appendix A The screenshot of the website



Appendix B The screenshot of sample results



information retrieval

Select mode: [✓ general] abstract title author

Select method: [✓ mix] tfidf bm25

Sorted by: [✓ relevance] Newest to Oldest Oldest to Newest

Showing 1 to 10 of 500

Show 10 20 50 100

Robin Aly Maria Eskevich Roeland Ordelman Gareth J.F. Jones

Adapting Binary Information Retrieval Evaluation Metrics for Segment-based Retrieval Tasks

cs.IR

Abstract: This report describes metrics for the evaluation of the effectiveness of segment-based retrieval based on existing binary information retrieval metrics. This metrics are described in the context of a task for the hyperlinking of

Kyosuke Nishida Itsumi Saito Atsushi Otsuka Hisako Asano Junji Tomita

Retrieve-and-Read: Multi-task Learning of Information Retrieval and Reading Comprehension

cs.CL cs.IR

Abstract: This study considers the task of machine reading at scale (MRS) wherein,