

Article retrieval system

<https://github.com/Rebryk/SPbAU-IR>

Vsevolod Stepanov

Yurii Rebryk

tehnar5@gmail.com

y.a.rebryk@gmail.com

ABSTRACT

This is our report for Information Retrieval project.

1 PROJECT DESCRIPTION

We are developing a system for searching scientific articles.

The data for our search engine is collected from online libraries like arxiv.org and from publishers that have information about their articles available online (like springer.com).

Firstly, we crawled 20k pages from these sources and saved them for future processing.

Then, we filtered irrelevant pages (those that doesn't contain an article description) and processed the rest of the documents. For each relevant page, we extracted article's title, abstract, list of authors, date of publishing, etc. Also, we store a link for a PDF version of an article, if it's available.

After that, we are going to implement a search engine for searching among these articles. **We will discuss it later.**

The key feature of our service will be creating some kind of graph over authors/articles and their references to each other. **We will discuss it later.**

2 SYSTEM DESIGN

We use Python 3 for development and PostgreSQL database.

Articles, processed articles' abstracts and inverted index are stored in a filesystem. Articles' metadata and PAF are stored in the database.

3 DATA ACQUISITION

For data acquisition we implemented a web crawler.

The key details of our implementation are:

(1) Politeness policy

We follow the constraints defined in `robots.txt`. We do not visit excluded pages, do not store page if there is a `noindex` meta tag, and of course, do not spam a website with a lot of queries. Even if a delay is not specified in `robots.txt` we use value defined in the configuration file (default delay is 50ms).

(2) Distributed crawling

The problem with Python's multithreading is that only one thread can be run at a time because of GIL, so we use multiprocessing instead. Each crawler runs in its own process and several crawlers can be run simultaneously.

Crawlers do not exchange URLs. This solution is suitable in our case because:

- (a) There are not so many data sources with scientific articles so we can just list them.
- (b) We are not interested in links leading to another domain, because, most likely, this domain will not contain any relevant documents.

Also it gives us a few advantages:

- (a) Simplicity of implementation. It is more difficult to make a mistake.
- (b) Easy to run on different computers, because communication is not needed.
- (c) Each crawler has its own set of allowed hosts (we define these hosts in the configuration file). It prevents the same page to be downloaded several times.

We store each HTML page we crawled in a filesystem and put some page's metadata (like its URL, hash of page content, date of last page modify) in database. Duplicated pages (with equal hashes) are ignored.

4 DATA PROCESSING AND STORAGE

As we have a few data sources and we need a specific information extracted from crawled documents (like article's title, abstract and list of authors), we implemented a separate data processor for each data source.

Data processing is done as follows:

- (1) Firstly, we filter out pages that doesn't contain an article.
- (2) Then, we extract relevant parts of remained pages and transform it to a plaintext, removing all the HTML tags.
- (3) After that, article's abstract is processed for future indexing, stemming its words and removing stopwords and punctuation

Raw and processed article's abstract are stored in the filesystem, other information (like title, list of authors, link to PDF if available) is put into database

We use BeautifulSoup for webpage parsing, and NLTK for abstract processing.

4.1 Indexing

4.1.1 Page attribute file. For each article, we store its attributes: the title, the path to file with abstract, the path to file with processed abstract, the number of words, the link to pdf and so on.

This information we store in the database.

4.1.2 Inverted index. We built an inverted index. For each word, we store a list with documents that contain this word. And for each

document, we store the number of occurrences of this word in the document with positions of these occurrences.

The document lists are sorted by the number of occurrences. Also, we use gap values to store positions of these occurrences for better compression.

This index can be build by several processes simultaneously. We split all articles by their id into groups, and each group is processed by a separate process. After that, all indices are merged into one index.

We store this index in a filesystem. We use gzip for compression.

5 RANKING

We implemented TF-IDF with cosine similarity for ranking.

We used inverted index constructed previously to calculate TF-IDF.

To improve search performance, for each document its vector is constructed once and then stored on disk. These vectors are stored in several files, each file contains a batch of vectors. To decrease memory usage and improve search performance even further, vectors are stored in a sparse form.

When answering a query, we retrieve only documents that contain at least one term from the query and then sort them.

6 FEATURES

6.1 Article similarity graph

We want to plot articles as points on 2D plane in a way that articles that are “similar” would be close to each other as points on plane.

One has an opportunity to see the given documents on this map and click on neighbor articles to open them.

To measure “similarity” of articles we’re going to use doc2vec. Unfortunately, training doc2vec takes some time, so for now we implemented more simple approach. We use pretrained word2vec to get a vector for each term in an article and then use mean value of these vectors as a document’s vector.

Later, we’re going to compare this two approaches (but most likely, trained doc2vec would be better)

6.2 Authors graph

The same feature will be implemented for authors.

For each author we’ll calculate mean vector of his articles and then visualize it the in same way as for articles.

7 INTERFACE

We use Flask, Angular and Bootstrap to visualize search results (fig. 1).

7.0.1 Additional filters. Users have an opportunity to search articles not only by content, but also they can specify the date range. In this case, the system shows only that articles which were published in the given date range.

7.0.2 Articles map. We project all articles into 2-dimensional space and draw this space. On the map, one can see the extracted documents and click on neighbor articles to open them. It’s an additional opportunity to discover similar articles.

Article Retrieval System

[Universal gradient descent](#)
 Alexander Gasnikov 01 Nov 2017
 Relevance: ☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4
 In this small book we collect many different and useful facts around gradient descent method. First of all we consider gradient descent
[+ Show More](#)

[Local Gyrokinetic Study of Electrostatic Microinstabilities in Dipole Plasmas](#)
 Zi-cong Huang, Wei-ke Ou, Bo Li, Hua-sheng Xie, Yi Zhang 31 Oct 2017
 Relevance: ☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4
 A linear gyrokinetic particle-in-cell scheme, which is valid for arbitrary perpendicular wavelength $k_{\perp} \rho_{\perp}$ and includes
[+ Show More](#)

[Near-Optimal Straggler Mitigation for Distributed Gradient Methods](#)
 A. Salman Avestimehr, Mahdi Soltanolkotabi, Songze Li, Seyed Mohammadreza Mousavi Kalan 27 Oct 2017
 Relevance: ☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4
 Modern learning algorithms use gradient descent updates to train inferential models that best explain data. Scaling these approaches
[+ Show More](#)

[Effect of Composition Gradient on Magnetothermal Instability Modified by Shear and Rotation](#)
 Sagar Chakraborty, Anya Chaudhuri, Himanshu Gupta, Shubhadeep Sadhukhan 27 Oct 2017
 Relevance: ☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4

Figure 1: Search page

[Universal gradient descent](#)
 Alexander Gasnikov 01 Nov 2017
 Relevance: ☐ 0 ☐ 1 ☒ 2 ☐ 3 ☐ 4
 In this small book we collect many different and useful facts around gradient descent method. First of all we consider gradient descent
[+ Show More](#)

Figure 2: Feedback tool

7.0.3 Feedback. We decided to add an additional bar with buttons for each retrieved document. Users can use these buttons to mark how relevant the given document is. This feature is a tool for evaluating the performance of our system on the next project stage (fig. 2).

8 EVALUATION

We asked 8 assessors to help us evaluate our search service.

8.1 Offline evaluation

We provided them with 10 search queries listed below in the table and asked them to rank search results on a five-grade scale from 0 (irrelevant) up to 4 (vital document).

Assessors submitted their assessments via search page using evaluation buttons on a search result snippets.

Query	DCG	MAP	RR
gradient descent	28.078	0.833	1.000
neural networks	39.942	0.917	1.000
clusterization	19.989	0.639	0.500
graph theory	23.480	0.450	0.500
linear differential equations	15.442	0.639	0.500
numerical analysis	11.924	0.325	0.250
statistics	48.747	1.000	1.000
linear programming	11.905	0.250	0.250
object detection	26.698	0.700	1.000
music generation	24.721	0.583	0.500

Table 1: Search evaluation

After that, we cleaned up their assessments, removing extra queries and filtering out duplicate assessments for the same query and document rank. We calculated several metrics based on cleaned assessments: DCG, MAP, RR. We treated document as a relevant if it had average score no less than 2.

8.2 Online evaluation

With our evaluation buttons, users can give us feedback about search quality while using our service.

We store all assessments in the database. So we can calculate different metrics easily.

8.3 User study

We asked users about their experience with our search system.

Advantages

- Quick article search.
- Simple, but comfortable UI.
- Convenient evaluation buttons, which allow accessors to do their job without a headache.
- Filters work perfectly.

Disadvantages

- Keywords are not highlighted.
- No additional snippets.
- Some articles are in German.

REFERENCES