Articlix: search through tons of articles

IR Term Project Report

Stanislav Belyaev
Saint-Petersburg Academic University
Saint-Petersburg
stasbelyaev96@gmail.com

Ekaterina Noskova
Saint-Petersburg Academic University
Saint-Petersburg
katenos823@gmail.com

ABSTRACT

Through the years, the problem of searching teaching materials on the particular topic has been very acute. In addition to finding series of educational articles, you need to be able to get a grasp of what exactly author telling. That's why we design Articlix - a search system that fits your knowledge level.

1 INTRODUCTION

The project is developed in several stages. First of all, we need to crawl data from the Web. Then, process it and build the index. Then, build a search system on top of that. Then, make a nice web-interface to run Articlix. And, last but not least, evaluate search precision.

The main project feature, as we suggest, will be adjusting to user interests and knowledge. The main part of the query is simple text, but alongside with it, the user can also pass some additional information or prior knowledge. Examples are "learn C++ | Java, Python". The second part of the query is prior knowledge, that helps Articlix do ranking more accurately. User define prior knowledge before run queries. Prior knowledge can be written in any form, as far as it represents a simple string or bunch of strings.

2 IMPLEMENTATION STEPS

We will go through each major step, describing each in detail.

2.1 Crawler

For the data sources, we gonna use blog posts platforms like medium and similar, focusing on tech-related topics. A typical medium-like article focusing on one small piece of knowledge, like some language feature or model architecture, so it makes it perfectly good for our task.

For seeds pages, we use some predefined set of tech-related tags, in which medium group blogs with a similar topic. The examples are: machine-learning and data-science. We obtained tags list from medium tags embeddings, finded on the Internet. Embedding helps us organize a clusters with similar meanings, easily fetching list of 200 tech-related tags. This tags used as a seed points to start searching from and as a set of valid topics to check if particular article (which annotated which tags) suitable for us.

Another data source is separate blog sites, building in the medium ecosystem. Example is hackernoon.com. For these, we first fetch raw html content of page, and trying to find medium badge, which indicates that particular site suitable for us. This needs slightly different approach in crawling, so here we do a medium source generalization. Essentially, this is just an abstract class to inherit from, with ability to override valid url, page, site and content checking.

More detailed: we provide validator interface for each source class, that has validate method for URL, site, and page. For example, some medium articles can be found at the separate site for the author, so we need to fetch page content first and see if it has the medium badge as HTML node. Rules for other data source are simpler: we just check if regexp (see next) matches URL. Here, we use slightly different approaches for seeds initializing and new URLs validating, so it multiple data source crawling.

One of the major problems with the medium for crawling - it's a lot of garbage links (twitter, youtube, etc.) which easily leads any breadth-first walking of the topic. For solving this, we use set of predefined constraints on valid URLs to walk. Essentially, this is just regexps, that match all valid posts and skip unnecessary links. Example is

```
r'https:\/\/medium.com\/(?!tag|p)[@A-Za-z0-9_-]+\/\Z'
```

for medium articles. We also break walking if 2 pages in a row doesn't suit us. That's help us not to walk out of tech articles far away.

We implement crawler using *Python3* for the sake of code simplicity. Nevertheless, this comes with one major problem - GIL. That's why we use multiprocessing distributed crawling, instead of multithreading one. This surely gives us major overhead, still not of a big deal though. The processes communicate through one big process-safe queue, which gives tasks and accepts new URLs to add. We don't processing the same URL twice, check if new URL to add in the queue already was processed sometime before. For all of this tasks, we use process-safe implementations of data structures in Python multiprocessing package.

Crawler fully satisfies politeness policies, including page permissions, robots.txt and crawl delay (waiting for 5t if none). This part needed to be done carefully, because medium politeness policies are rather strict.

For any stored page, we also fetch some meta information, like number of likes, comments, estimate time to read (if any), publisher, publish date, author, etc. This information will be used in future to do ranking more accurately. After page fetching, we store its content with additional meta-info at PostgreSQL database. Here, we also add some UNIQUE and NOT NULL constraints (on URL and text, particularly). Futher, we will move data to zipped csv file, as it is easier to get it from python.

It takes approximately 1 hour to store about 10k unique URLs. For now, we fetch about 70k pages and check content manually for some of them (not founding a garbage pages, so that's could be a good data to work with).

https://github.com/stasbel/articlix

Overall, the hardest part of crawling was writing with process-safe code and dealing with garbage URLs.

2.2 Index building

After crawling we've got raw HTML text. Articlix catchs block "div" that corresponds to article content and extracts text from title and basic text in order to get useful representation. All HTML tags are removed in order to get plain text (html2text and BeautifulSoup packages). This text with some meta information about parent page (published date, estimated time to read article, number of comments and likes) are inserted as raw in created data base.

As soon as Articlix stopped crawling and filled data base, it begin to build reversed index for fast search. We used multiprocessing to build it. Each process can work with one document — title text and base text of one article.

Each text is preprocessed: punctuation is deleted, text is separated into words, stop-words are removed, remaining words are stemmed (all is done using nltk package). Moreover Articlix deals with phrases: it extract bigrams from text and stems them too.

Then Articlix build reversed index from list of words and phrases. For each word index include documents' ids, where word appeared, its positions in document and variables that corresponds to part of document, where the word was observed: title or base text. Index is implemented as dictionary for faster search.

As already said, one process build index for one document. When small indexes are built for all of documents, they are combined into final common. As indices are dictionaries, their union is rather simple. This index is serialized into file for further work with ujson library. It is also possible to update index, when new document appears. Ujson helps to make index fast.

After index was built we print it to check. It was valid: most frequent words were in a lot of documents and special only in some.

During task there were some troubles in building index. Firstly, we have to change our data base, that we had after crawling: because new columns appeared. Secondly, when multiprocessing was used index was build too long: the problem was in shared between processes multiprocessing-save dictionary-index. Since we use Python, this dictionary was from Manager class (multiprocessing module) and this class have some delay in working with its structures. That's why we aren't using shared memory and each process returns its own small index.

Besides pages, we also store bunch of meta information as page features inside db as separate columns.

2.3 Search

For now, we have data with meta info and index, built on that data. Our next task will be making a search system, which operates on top of data and index. Some implementation details are: search system will be separate class, which takes bunch of parameters and provide find method to do the querying. We do ranking queries, resulting in list of urls for relevant articles.

Core idea of our search system is, of course, BM25. Here, you can control which tf and which idf calculating method to use, passing appropriate arguments as search system parameters (you can also add smoothing, which is 0.5 by default). U can also pass k_1 and b consts, which are 2 and 0.75 by default.

First feature is prior information about user. You can pass it as bunch of strings to search system, which then calculate bm25 score per document vector for each prior string and that sum them element-wise, resulting in prior document relevance score. This score goes in weighted sum with query bm25 score. Weights are p and $1 - p$, where p parameter allows us to control how much we want to take prior information into account (default $p = 0.25$).

Second feature is spell checking. For given query q, besides basic bm25 score, we also calculate bm25 score with corrected version of q with spell checker (word-wise). We then take max of this two score, results in final score for document. This make sense, assuming that most of a queries either correct or consist one grammar mistake, so our way covered most of the real cases. It's planned that user will be somehow warned (some mark in graphical interface?) if spell checker suggest some word to fix.

After obtained score for each document, Articlix find n topmost articles (you can of course pass n as a parameter) according to scores (using argpartiton alg, so it O(n)). Then, Articlix can additional sort n relevant articles according to some order, which is also controlled by user (for example, you may want to see articles with more comments first). Articlix using small score treshold=0.05 to filter non-relevant data, which helps, for example, not to get query result, when user passed random combindation of letters (all bm25 scores will be 0).

Overall, system runs pretty fast, because we use as many vector operations (numpy) as possible. We also do caching for last 1024 querries, tf and idf calculatings, which give a little speed-up, assuming most of the users will do search of things, which were querried sometime before (maybe we need to adjust cache size, but that's optional too).

2.4 Web interface

One of the most important parts of making your own search tool is making web interface. It should be useful, intuitive and should reflects the purpose of the tool.

We developed interface for Articlix. It is simple HTML pages.

Home page includes one field, that user should fill: his personal information, that search tool will use. He can't leave field empty, Articlix will insist on filling it.

As soon as you fill information about yourself and press button "Continue", you will see the main search page. There are more features for search inference here. First of all query window is on the top. Under it there is personal information field, that is already filled. Next there is select box with different sorts: by relevance (scores include information about user), by published date, by likes, by comments, by estimate time to read article. First sort is default. Also user can choose how many results he want to see.

Under everything fields there is magic button "Search". When you press it, page is refreshed and search result appear. It is a table with references to articles. You can also see additional information about every article: published time, estimate time to read it, number of likes and comments. Every article is shown as reference to it with its title as text and there is snippet of its content under title.

Words from main query are highlighted on title and snippet. Positions for highlighting are taken from index: we search positions for every token in query in index. Snippets are chosen to maximize number of highlighted words in it. For it we slide window of fixing size over article content and calculate number of positions, that corresponds to query tokens' positions, that are inside this window. We take window with maximum value.

Articlix also make spell check: if score of fixed query is higher that not fixed, it will ask user: "May be you mean: **fixed** query". Text that was fixed in fixed query is highlighted too.

Moreover we use bootstrap4 for nice and clear interface theme.

2.5 Evaluation

The main idea behind evaluation of Articlix search precision is DCG. We will access each search answer with score from 0 to 4, showing how good this answer related to initial query intention.

For this, we need predefined query tasks (10 for each assessor, failrly enough, because it can be time consuming), ground truth scores (put by ourself, maximum score by default) and assessors scores. The last one we obtained from server log csv after all assessment sessions were over. There were around 5 assessors, each got 10 query tasks and small and simple instruction, finishing with offering to additionally evaluate overall user experience with simple google form including questions about web interface convenience and features usefulness. Each task consist of query to run, prior to pass in to "information about yourself", sorting order and result size upper bound.

For the sake of better user experience while evaluating, we run our web interface on the server using apache2. We also add assessment column for users for do the evaluation right into the Articlix. This is simple and convenient solution, which helps make the assessment less painful. Assessments results saved at server as csv with "timestamp, ip, query, prior, sort, topn, pos, score" columns, which helps easily parse the results into python code and do the calculating. The Articlix can still be accessed at http://35.227.117.218/.

We calculate normalized dcg version and get around 0.67, which is not that bad, assuming relatively small dataset size (for covering all tech related topics), rather complex tasks to evaluate and the fact is we don't check queries outcome, when define a tasks (we don't want to overfit on validate, right?). For example, one of the task was query "best programming language" with "hope for the best" prior information, which can be yield controversial results, with big dispersion.

2.6 Helping

We helped 3 teams with their projects assessment and overall user experience review. Each team provided small description of how to run, use and evaluate their systems. We gave some additionally comments on what we like and what we don't, which, hopefully, help other to better understand and find project problems.

Generally, out experience with other teams was nice and productive.

3 REFLECTION AND FURTHER WORK

Sure, Articlix has some drawbacks that can be fixed afterwards. One of the most important is rather small search space (200 tech related tags, which is, as it turned out, insufficient to cover all tech-related topics), that doesn't cover all aspects of our life. Also one of our features - spell checker - doesn't work pretty good. It was taken from PyEnchant library and can be upgraded. Another thing that we notice is that some snippets in web interface contain information in another encoding.

The future work will be dealing with drawbacks and revision of ranking method. The reason why it needs to be refined is because blog writing time is major feature, which certainly need to be taken into account somehow (this comes from the blogs nature - they don't last long and losing value exponentially with time). It was empirically deduct from experience in using medium search system in compassion with Articlix one. The medium tends to offer more fresh content, filtering out outdated blogs (which is why they could be non-relevant for us, even if they cover query intention).

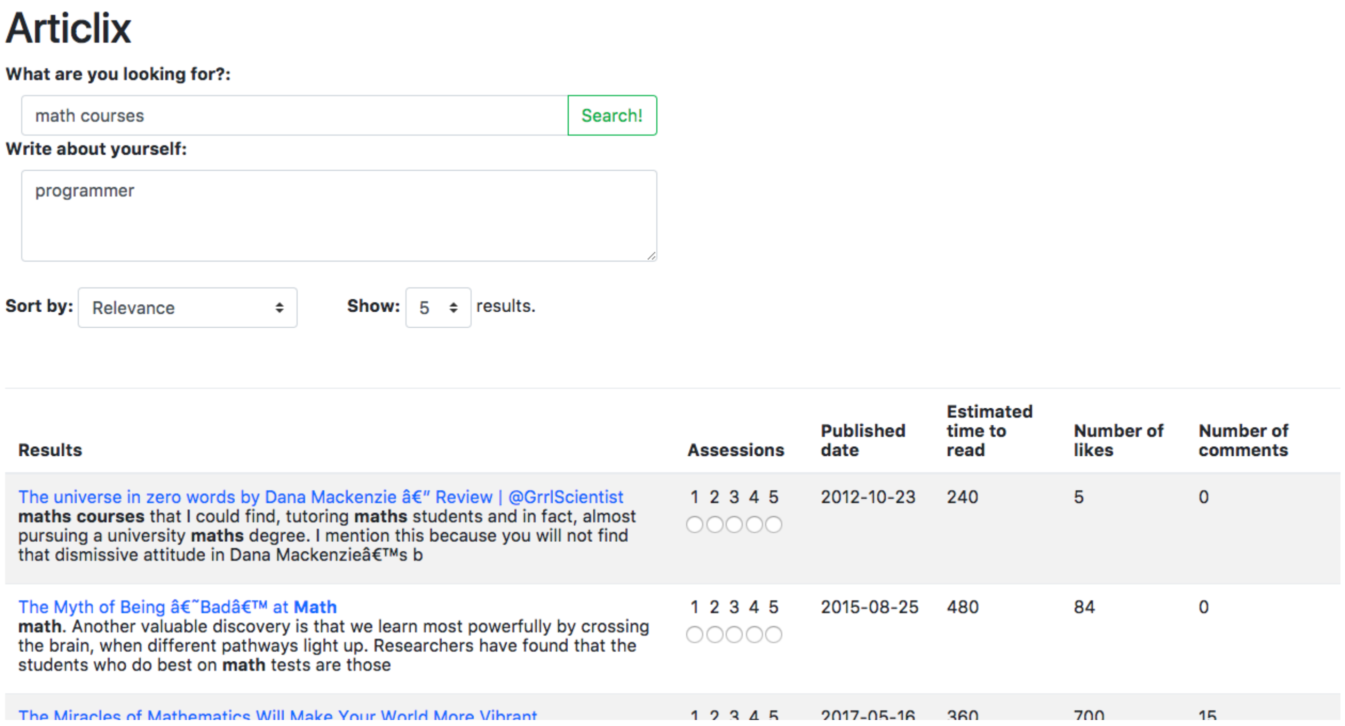
4 CONCLUSIONS

So, at the end of the day, we have developed, tested and evaluate search system with nice interface, fast and relevant results.

One thing we learned for sure is importance of nature of data your work with. It turned out, that blog itself is rather volatile and temporary object to work with, so it needs to be take into account when do rankings (writing and fetching timestamps). This problem needs further investigation, on which we don't have time to.

Second thing we want to emphasize is different steps complexity. The biggest problem was, of course, to implement good, polite and fast crawler, which is seems to be trivial at first glance, but turned out to be rather painful task to do. Other steps are more clear (so, maybe time and work proportion need to be refined).

Despite the minor drawbacks, we managed to make a complete search system, which has a bit of a sense (well, it's not useless enough) and we hope you'll enjoy using it!



Articlix

What are you looking for?: math courses

Write about yourself: programmer

Sort by: Relevance

Show: 5 results.

Results	Assessions	Published date	Estimated time to read	Number of likes	Number of comments
The universe in zero words by Dana Mackenzie â€" Review | @GrrlScientist **maths courses** that I could find, tutoring **maths** students and in fact, almost pursuing a university **maths** degree. I mention this because you will not find that dismissive attitude in Dana Mackenzieâ€™s b	1 2 3 4 5 ○○○○○	2012-10-23	240	5	0
The Myth of Being â€˜Badâ€™ at **Math** **math**. Another valuable discovery is that we learn most powerfully by crossing the brain, when different pathways light up. Researchers have found that the students who do best on **math** tests are those	1 2 3 4 5 ○○○○○	2015-08-25	480	84	0
The Miracles of Mathematics Will Make Your World More Vibrant	1 2 3 4 5	2017-05-16	360	700	15

Figure 1: This is a screen-shot of web-interface