Code-related text search engine

Report

Yaveyn Anna
Simiyutin Boris

1 DESCRIPTION

Our project is a developer-oriented search engine. The main goal is to help people with technical problems, in a way similar to StackOverflow. Here is the link to project's repository.

2 ARCHITECTURE AND DESIGN

We have selected Kotlin as our main programming language. Although, we keep our system modular and use other languages when it is convenient. For instance document processing and inverted index building is implemented in Python.

For parallel crawling we use Akka actor system. Also we use Apache Spark framework for faster index building.

2.1 Data acquisition

2.1.1 General info. Data acquisition in our system is performed by crawler, which collects data from the Internet, starting from popular programmer-oriented sites, such as www.linux.org, stackoverflow.com, cppreference.com and docs.oracle.com. We use single configuration for every seed.

Our program meets politeness requirements for a web-crawler. We use open source RobotsTxt library for parsing robots.txt files and follow request rate conventions by 20 sec timeouts.

During development of crawler we faced performance issues, which were solved by making it multithreaded. We also had some troubles with intense memory usage. We fixed them by setting maximum total number of processed pages by each worker, which limited maximum size of it's page queue.

2.1.2 Concurrency. We decided to perform data acquisition in parallel and we chose an actor-based concurrency model to do so. The main advantage of using actors is that you can write highly concurrent, distributed and fault-tolerant code without thinking about synchronization between different threads. One of the most popular toolkit for actors in JVM is Akka - a cross-platform open-source framework written in Scala. We use it because it is a well-known and powerful tool with comprehensive documentation and large community.

The crawler consists of one manager actor and ten worker actors. Worker actors perform the actual crawling and the manager actor handles the communication between them. Each worker is responsible for it's own range of host names and holds it's own queue of Urls. When worker comes across a host name which it is not responsible for worker sends request to manager and manager redirects the Url to corresponding worker. The manager is also responsible for assigning workers to new host names.

2.1.3 Summary. The resulting program runs pretty fast and uses network channel rather efficiently. At first stage we crawled 103000 documents. However, we have some problems with rate limits at multi-domain resources e.g. stackexchange.org which we are going to fix on the next stage of the development.

2.2 Data storage

2.2.1 General info. This part of project is written in Python as there is a wide variety of python modules for text processing.

2.2.2 Preprocessing. At first, we parse html code of pages and extract headers and body text for weighted ranking. Then, we remove punctuation, tokenise sentences, filter words from English dictionary using *enchant* module, remove stop words and perform stemming using *nltk* module.

2.2.3 Indexing. For each word and each file we compute the number of occurrences of this word in each header and in the body of a web page. For faster computation we decided to use map-reduce technique. At first, we tried to use Apache Spark for this purpose, but we faced some problems with tuning it in proper way, so we switched to pure python and used *multiprocessing* module. As a result we have an inverted index which takes into account the fact that words from different parts of HTML document should be counted separately.

We keep map from document id to its URL in a text file to reduce index file size.

Index is stored as serialized python dictionary. We expect full index file to be less than 600 MB so it's OK to store it in RAM. Search will take constant time due to default python implementation of dictionary.

2.2.4 Stackoverflow Database. As we use posts from stackoverflow.com to aid in our search, we also conduct an indexing of these posts. The initial data is acquired as Microsoft SQL Server Database which is distributed free of charge and is available via Torrent. Due to the excessive size of the database we have to filter these posts out with accordance to the score of the corresponding accepted answer. We take 30,000 posts with the highest score. Furthermore we do not consider answers other than accepted one because they often contain duplicate information and are less comprehensive.

To sum up, for each of 30,000 most popular posts we take theirs titles, bodies, tags and texts of the corresponding accepted answers and conduct an indexing process in the same way we did for the crawled pages. The only difference is that we process different parts of each post (title, tag, body of the question and body of the answer) separately instead of separating results for different parts of an HTML document as it was with crawled pages.

2.2.5 Summary. We ran our indexer on subset of crawled data consisted of 75k pages. It took 1 hour and 10 minutes to process

https://www.brentozar.com/archive/2015/10/how-to-download-the-stack-overflow-database-via-bittorrent/

them and index size is 500 MB. Processing full 100K dataset is going to take no much longer.

2.3 Searching

2.3.1 General info. Our system uses data both from crawled pages and from Stackoverflow database and mixes it to show in one ranked list of results. As ranking method we use weighted BM25.

2.3.2 Basic search. Our index is stored in chunks to conserve RAM and reduce garbage collector load on indexing stage. Each chunk contains info about 20K documents. Basic search is implemented in two stages.

- The first one is preprocessing stage:
  - We merge all index chunks into one, and collapse term frequences with weights corresponding to source of terms, e.g. headers have greater weight than body for web pages, question body has greater weight than answer body for SOVF posts.
  - We create a new index file. For each pair (term, document) we calculate BM25 score with parameters $k = 1.2, b = 0.75$ When all data is processed, index file is stored on disk.
- Second stage is fast online query processing and results retrieval without heavy computations. As soon as our index in rather compact and fits in memory, we load it fully as a python dictionary.

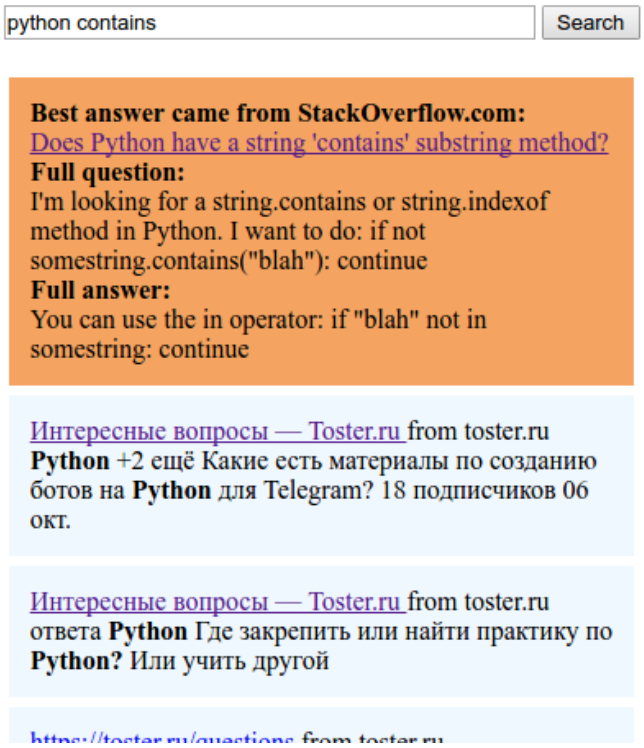
2.3.3 Stackoverflow DB. As the only difference between indices built over the crawled pages and over the stackoverflow database is different categories the text is split into (HTML headers "h1" - "h5" and "body" for crawled pages and "title", "tags", "question" and "answer" for stackoverflow), the process of assembling the final index and the searching algorithm used to deal with the data from stackoverflow are essentially the same as described in the previous paragraph.

2.3.4 Special features.

- We combine search results of two different types (from crawled pages and from stackoverflow database) into one list. In order to do this it is enough to empirically choose weights for corresponding labels on stage of BM25 score preprocessing. At current stage we used coarse estimation of them and going to finely tune weights at the stage of evaluation.
- If the score of the best post from stackoverflow exceeds a certain threshold we regard it as a perfect result for given query and show the answer from this post as a card on top of all results. The threshold is linearly dependent on the query size and is estimated based on results of several test queries.

2.3.5 Summary. Our search system produces reasonable results for most of queries. Although we think we will need to recrawl our document set to fit data restrictions more precisely. E.g. the first dataset lacks considerable amount of pages we are interested in and contains certain amount of surely irrelevant pages such as from Youtube or non-english Wikipedia.

2.4 Interface



2.4.1 General info. We made web interface for our search system. It has query box and search button. When button is pressed, results are obtained asynchroniously via AJAX GET request and represented as tiles. Each tile has the link to the document shown as its title and a small snippet of text with query terms highlighted.

2.4.2 Web server. We use Flask as a framework for Python REST web server, as it is super easy to use and fits our task perfectly.

2.4.3 Features. Our first special feature - showing post from StackOverflow as a card if it is substantially better than other results - needs special interface, when our second feature - smart mixing results from different sources - doesn't.

2.4.4 StackOverflow cards. When system provides result from StackOverflow which matches query very well, it is shown on top of all other results in a separate block. In this block the full question, answer and link to original post are given.

2.4.5 Snippets extracting. To obtain snippet of document that matches query well, we split all document in successive patches and count number of query terms in preprocessed version of each patch. Best patch is looked through one more time and all terms from query are highlighted.