Information Retrieval

Project report

Mikhail Koltsov
Andrey Kravtsun

1. INTRODUCTION

There is a problem with searching shoes when you need to buy one. We decided to implement a search engine for simplifying this task so you won't need to search on your favourite sites selling shoes and some clothes stuff.

Desired functionality: given a query with shoes description, like *"туфли спортивные с закрытым носом"*, show only relevant e-shop pages.

Features (may change later during development):

1.1. Source code

We have our repository at GitHub[1]. Code is in Python 3.5 and package requirements are stored in `requirements.txt`

(1) system understands queries like *"туфли с закрытым носом"* and *"туфли на танкетке"* as different, and results in precisely relevant pages with user-specified filters applied;
(2) filtering on preferred e-shops;
(3) filtering and sorting by price.

2. DATA ACQUISITION

2.1. Architecture

We aimed for the distributed crawling from the start, so our architecture was largely influenced by it. Also, we wanted components to be abstracted from each other. For those interested package diagrams were generated and put into docs[2].

2.1.1. Main package.

- `Page`[3] - entity that represents one page. It knows whether the page was downloaded, what are its children (sites that this page refers to inside its domain) and its last fetch time.
- `DomainQueue`[4] - shared queue of seed domains. All workers read from it. Each worker gets one domain, removes it from the queue and starts breadth-first search (BFS). Access to this queue is not protected by mutexes, because it is very rare: one worker spends most of its time doing BFS rather than reading new domains. So, we treat the situation when two workers are reading from DomainQueue simultaneously as low-probable.
- `CrawlQueue`[5] - worker-private queue of pages. Each worker performs Breadth-First Search from the received seed domain. Each worker operates in Firewall-mode: only crawls pages from one domain at a time.

2.1.2. Package `'storage'`.

- `BasicStorage`[6] - abstraction of storage. It has its implementations as:
- `LocalStorage` and
- `GDriveStorage`.

Along with page, meta-information is stored: URL, size, path to file. Storage also filters duplicate pages by content using MD5 hash (it is sufficiently long to not bother about hash collisions, also it is reasonably fast).

2.1.3. Package `'robots'`.

- `RobotsProvider`[7] - static entity that knows about `robots.txt` of certain domains.
  Functionality of `RobotsProvider` is implemented through usage of:
- `MemberGroupData`, which corresponds to rule for specific member group at `robots.txt`), and
- `RobotsParser` that generally converts raw text from `robots.txt` into viable parameters.

2.2. Politeness

Crawlers can retrieve data much quicker than human searchers, so they can influence badly on a site performance. In order not to bother any site with our crawler's activity and have no impact on its performance and throughput we use several commonly used guidelines.

2.2.1. Robots exclusion protocol.

- Our crawler does not query a page's children if it contains `nofollow` word in tag's `<meta name="robots">` key `content`.
- We neither fetch nor store pages with `noindex` in the tag mentioned above.

2.2.2. Server-wise exclusion.

- We do not request any pages declared under `Disallow` directive in `robots.txt` for common member group (`user-agent: *`).

[1] https://github.com/ItsLastDay/FindMyShoes
[2] https://github.com/ItsLastDay/FindMyShoes/tree/master/docs
[3] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/crawler/page.py#L19
[4] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/crawler/queues.py#L9
[5] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/crawler/queues.py#L51
[6] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/crawler/storage/basic_storage.py#L9
[7] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/crawler/robots/provider.py#L13

IR'17, Fall 2017, Saint-Petersburg, RU
.

- We respect site performance diversity and use `'crawl-delay'` parameter in `robots.txt` for setting pause between consecutive queries. If it is not present, we pause for 1 second between queries.

2.2.3. Crawler identification.

- Our crawler declares itself as `'findmyshoes_bot'` in HTTP requests to server.

2.3. Flow

There can be many participating processes in our crawling. Each computer must have a list of seed domains assigned and placed into `domain_queue.txt` file next to `crawler.py`. One can start arbitrary number of workers with command:

```
python3 crawler.py
```

Then, firewall strategy will took place and every domain will be assigned to one specific worker (so that one worker still can crawl several domains if it's fast enough).

Workers have an algorithm, as follows:

(1) read line from `domain_queue.txt` and erase that line from the queue. If the queue is empty, exit.
(2) do a BFS on this domain, maintaining delays and storing pages with `storage.BasicStorage` instance. Each page is checked whether it can be crawled or stored.
(3) return to step 1

2.4. Problems

- Google Drive for storing pages was overall a bad idea. It works slower than storing pages locally, because each store is >= 1 HTTP-requests. Also, GDrive API has rate limits. We managed to store about 9.000 pages in 6 hours, whereas storing pages locally could achieve > 100.000 in one night run.
- Moreover, there were problems with storing Russian letters through GDrive API, so we stored base64-encoded pages. And the API seems not to be well-documented itself.
- We tried to use standard solutions for `robots.txt` parser, like `urllib.robotparser`. But they surpisingly failed even on first test page[8], which is allowed to be crawled in corresponding `robots.txt`[9] but unfetchable as those parsers say.

2.5. Results

At night 115793 pages were downloaded while running 3 crawlers on several different domains, as our expert colleague proposed:

- https://www.bonprix.ru/
- https://www.lamoda.ru/
- http://www.asos.com/
- http://www.ecco-shoes.ru/
- https://respect-shoes.ru/
- https://ru.antoniobiaggi.com/
- http://www.rieker.com/russisch
- https://www.net-a-porter.com/ru/en/

Our crawler proves itself to be polite and distributed. Besides, we crawled more than 100k documents. Therefore, we hope for an excellent mark.

3. DATA PROCESSING AND STORAGE

3.1. Overview

During this stage we have to extract meaningful parts from raw data, so we can use it in our search engine later. Our raw data is a HTML page from an online shopping site corresponding to one pair of shoes. We assume that this page describes all there is to know about this particular shoes.

Conceptually, we divide all information on the page in two parts:

- shoes' attributes, such as its brand, color, type, etc. We can use these attributes as filters, because data is highly structured;
- description of the shoes and user reviews. This is unstructured, free-text, opinionated data that can be used in text search.

All our code for this part can be found in the repository[10], as well as README files on how to run the indexer. We provide sample raw HTML pages, so that every viewer can run our index builder and make queries to it.

3.2. Data storage

Two data storage methods are used.

3.2.1. Indexing unstructured data. We compute *inverse index* from our unstructured data. When you have several documents, *forward index* is just these documents: for each document we know set of all words contained in it.

Inverse index is a similar thing: for each word in our corpus[11] there is a list of documents containing this word. Actually, there can be several lists: list of documents, list of term frequencies[12] or word positions in these documents.

We currently compute two lists: documents that contain this word and term frequencies in documents. But our architecture is extensible.

Inverted index data is stored partially in text, partially in binary format. There is an entity *dictionary*, which is a text file in JSON[13] format with the following example structure:

```
{
  "words": {
    "хотел": {
      "global_count": 38,
      "df": 38,
      "offset_weight": 0,
      "offset_inverted": 0
    },
    "подкладка": {
```

[8] https://www.google.com/maps/reserve/partners
[9] https://www.google.com/robots.txt
[10] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing
[11] set of all documents
[12] i.e. number of occurrences
[13] JavaScript Object Notation

```
        ...
    }
    ...
}
```

Word-specific fields consist of:

- `global_count` is the total number of occurrences of this word in our corpus;
- `df` is document frequency, i.e. number of documents this word appears in;
- `offset_inverted` is the offset (in bytes) in inverted list binary file, at which the inverted list for this word begins. Notice that inverted list has the same size as `df`, so we need to store only one offset. This would not be true for list of word positions in documents;
- `offset_weight` has the same meaning as `offset_inverted`, but for the list of term frequencies.

The inverted list itself is stored in a binary file (term frequencies list is similar, so we'll only describe one of them). It contains only integers of fixed width (4 bytes). For each word, its inverted list is a continuous segment of integers. There are no separators, because we know all extents of any word's list using the *dictionary*. In order to build this index, we use two-pass algorithm (meaning that all documents are read twice):

(1) compute the total size of inverted list (that is, sum of all term frequencies), compute offsets of each word's inverted list part
(2) fill the index with actual data

Code for the index building can be found in Index Builder script[32].

In order to read from or write to these binary files we use `mmap`[14] library. It is a low-level feature that allows to map the whole file into the address space of a process. This file appears as a continuous segment of memory, so we are able to do random reads and writes (as opposed to sequential reading from regular files). Moreover, mmap supports laziness: if we do not touch some part of the file, it won't be even loaded from disk to memory.

Example of reading from index can be found in repo[15].

3.2.2. Database for structured data. We use MongoDB[16] for storing all structured data in the way described below.

3.2.3. Structured attributes.

- `name` as shown by e-shop (e.g. `"Ботинки ECCO HOLTON 621174/01001"`)
- `brand` (e.g. `"ECCO"`)
- `type` (e.g. `"ботинки"`)
- `colors` list of colors (e.g. `"белый"`, `"чёрный"`)
- `price` (in rubles, e.g. `"11000"`)
- `sizes` list of sizes in the Russian system (e.g. `39`, `42`)
- `gender` expected gender of a shoe wearer, as shown by e-shop (e.g. `"женская обувь"`, `"мужская"`)

3.2.4. Unstructured.

- `description` shoes description, as shown by e-shop
- `reviews` list of user review texts
- `attributes` dictionary of attributes that do not fall in "structured" category (or, at least, we don't know how to extract them from a certain e-shop). For example, those attributes can be one of the following:
  -- 'подкладка', e.g. 'мех'
  -- 'страна', e.g. 'Китай'
  -- ...

Example document of MongoDB[16] collection where we store extracted data can be found in our repository[17]. MongoDB uses JSON format for all data.

3.3. Data extraction

Each e-shop website has its own way of structuring HTML. For example, the name of the product (i.e. shoes) can be written in `<div id ="product_name">` tag. One can make queries like *"get all tags that has <p> as parent"* using CSS selectors[18], in this case `"p  div"`. CSS selectors also support id's and other attributes of HTML tags.

We analyzed each e-shop independently and came up with CSS selectors that refer to the meaningful information on the page. In general, it was structured like this:

- predefined places for major pieces of information, e.g. "div h1.product-name" selector for product name
- lists of variable size for minor attributes (like type of clasp), e.g. "strong.product-attribute-name" for all attribute names

Interestingly, this information was slightly different for different e-shops: for instance, one website shows "Country" in a predefined position, another writes it in a list of various attributes.

You can explore all our CSS selectors here [19].

3.4. Data processing

Structured information of a concrete product page are stored in the database as is. But unstructured data has to be preprocessed with the following pipeline:

- tokenization;
- cleaning;
- stop-word removal;
- stemming.

3.4.1. Tokenization. In order to extract separate words from our reviews and a good's description we use the algorithm:

(1) Divide text into sentences with regular expressions matching one or more sentence ending as `'.'`, `'!'`, `'?'`.
(2) Tokenize obtained sentences with `ToktokTokenizer`[20] from NTLK library[1].

[14] https://docs.python.org/3.5/library/mmap.html
[15] https://github.com/ItsLastDay/FindMyShoes/blob/indexer_report/src/indexing/index_reader.py#L17
[16] https://www.mongodb.com/
[17] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/data/mongo-document-sample.json
[18] https://www.w3schools.com/cssref/css_selectors.asp
[19] https://github.com/ItsLastDay/FindMyShoes/blob/indexer_report/src/indexing/domain_specific_extractor.py#L151
[20] http://www.nltk.org/api/nltk.tokenize.html#nltk.tokenize.toktok.ToktokTokenizer

3.4.2. Cleaning. Here we filter out non-words tokens, e.g. stray punctuation. We preserve digits, because shoe sizes and parameters like heel size can be significant when choosing the right shoes.

3.4.3. Stop-word removal. We use dictionary-based stop-word removal, filtering out words from the following list:

- 'на',
- 'и',
- 'в',
- 'не',
- 'очень',
- 'но',
- 'с',
- 'как',
- 'для',
- 'по',
- 'а',
- 'из',
- 'от',
- 'без',
- 'у',
- 'к',
- 'что',
- 'обувь'.

Those words were chosen from the current set of most frequent words in our documents, as these words are thought to be meaningless for our search engine.

3.4.4. Stemming. Our stemming stage is algorithmic-based, using Porter stemmer `RussianStemmer`[21] from NLTK library [1]. The stemmer's formal description is provided in [2].

3.5. Architecture

In this stage we decided to use action-driven approach, splitting all our tasks into several scripts.

3.5.1. Data Extractor. [22] Small routine that reads all of our HTML files in parallel, extract meaningful information using Domain-Specific Extractor and stores it in JSON files.

3.5.2. Domain-Specific Extractor. [23] Provides abstract class `AbstractDataExtractor`[24] for parsing product data from a page's HTML code. Central method `parse_html`[25] performs some HTML preprocessing, such as:

- stripping JavaScript code;
- leaving page structure as it is, i.e. <head>, <html>, <title>;
- removing CSS styling.

`parse_html` follows the *Template method* pattern. It calls all other methods that populate dictionary in strict order. Descendants can override all methods. It is meaningful for some site-specific processing. Otherwise, descendants should only override CSS selectors. Currently, there are several implementation for `AbstractDataExtractor`[24] as follows:

- `BonprixExtractor`[26] for www.bonprix.ru.
- `RespectExtractor`[27] for respect-shoes.ru
- `AntonioExtractor`[28] for ru.antoniobiaggi.com
- `AsosExtractor`[29] for www.asos.com
- `LamodaExtractor`[30] for www.lamoda.ru
- `EccoExtractor`[31] for www.ecco-shoes.ru

Obviously not all attributes can be extracted for each site, so our implementations provides only those virtual methods that apply.

3.5.3. Index Buider. [32] Simple class with two key methods that correspond to two passes of indexing algorithm as described in 3.2.1. Stores *dictionary* as an instance variable, so that preserving data between passes is easy.

3.5.4. Database storer. [33] Currently, it is a bash script for loading all JSON[13] files created with Data Extractor.

3.5.5. Text utilities. [34] Implements `TextExtractor` class for getting preprocessed words from raw text, as described in sections 3.2, 3.4.

3.6. Problems

- shoes description on www.ecco-shoes.ru is inside <noindex> tag. If we followed this restriction, we would not be able to search effectively from this website;
- ru.antoniobiaggi.com has awful HTML markup. For instance, they have several tags with id `"color_to_pick_list"`, whereas in "good" HTML you should have zero or one tags with particular name. That makes parsing the site harder;
- we looked at the pages we got from the crawling stage. It appeared that about 10% of them were actual shoes. So we had to rethink some parts of our crawler, so that it only stores "relevant" pages (where "relevance" is measured by matching URL's by regular expressions) and traverses pages in right direction. For example, on lamoda.

[21] http://www.nltk.org/api/nltk.stem#nltk.stem.snowball.RussianStemmer
[22] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/data_extractor.py
[23] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py
[24] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py
[25] https://github.com/ItsLastDay/FindMyShoes/blob/master/src/indexing/domain_specific_extractor.py#L60
[26] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py#L151
[27] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py#L182
[28] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py#L201
[29] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py#L230
[30] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py#L251
[31] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/domain_specific_extractor.py#L269
[32] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/index_builder.py
[33] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/database_storer.sh
[34] https://github.com/ItsLastDay/FindMyShoes/tree/indexer_report/src/indexing/text_utils.py

ru there are many pages, about 2 million, whereas only 25k of them are shoes, so we had to start from shoes catalogue instead of main page;
- we still need unification of several enumerable attributes, stated differently on different sites, i.e. `gender` can be represented as *"женская обувь"* in one place, and as *"для женщин"* in another. The same applies to attributes' names, i.e. *"Страна"* vs. *"страна-производитель"*, and so on;
- in a certain moment we both come to conclusion that we need a more thorough mutual code review, as there was a misunderstanding using the other person's code that lead to bugs. We have no opportunity of pair programming, so we need to be more descriptive in our documents and gradually come to unified coding style;
- we had a hot discussion on how to do words filtering: do we need to leave numbers or words with both alphabetical and numerical characters? As we do not use any complex data processing techniques now (such as n-gram[35]), we decided to postpone solving these ambiguities.

REFERENCES

[1] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".

[2] Martin Porter. 2007. Russian stemming algorithm. (April 2007). Retrieved November 3, 2017 from http://snowball.tartarus.org/algorithms/russian/stemmer.html

[35] https://en.wikipedia.org/wiki/N-gram