Project Report

Information Retrieval

Albert Cerfeda
Alessandro Gobbetti

This project was realized in the span of about 2 weeks (14 days).
Developed as part of the group project for *Information Retrieval SA 2021-2022*, part of BSc INF at Università della Svizzera Italiana (USI) in Lugano.

Contents

16.12.2022
Università della Svizzera italiana
Faculty of Informatics
Switzerland

1 Introduction

Banana™[1] aims to be a search engine to discover and fund your favourite content creators.
We put efforts into making our system sustainable and self-improving, without requiring us to start processes or intervene manually. We faced quite a lot of challenges, mostly regarding Solr [See Sec.2.2] and its very steep learning curve.
Nonetheless, we are satisfied enough with the current state of the project. There are still many improvements to be done, but considering the time on our hands this is good enough.

2 Tech stack

In our system there are dfferent components that operate independently from one another. It's important to gain understanding of the moving pieces that make our system work.

2.1 Scrapy

Scrapy is an open source Python framework for crawling and scraping websites [2]. Our backend invokes the Scrapy process for crawling/scraping a set of artists, categories or search queries. The scraped data output is then directed into a `.json` file, ready to be read and indexed by Solr.

2.2 Solr

Solr is a popular enterprise search platform built on top of *Apache Lucene*[3]. It includes features like distributed indexing, replication and load-balanced querying, and automatic recovery.
It is responsible for indexing and querying the collection. It retrieves the artists most pertinent to the user's search query and filters.

2.3 NodeJS + MongoDB

For our backend we chose NodeJS `v. 19.2.0`. It handles user queries, relevance feedback and coordinates and interacts with all the other components of our stack.
It also handles the scraping and indexing process, as it periodically provides the Solr and Scrapy processes with the authors/tags/queries to scrape and index. The MongoDB database serves for the purpose of storing and remembering the index timestamp for each author/tag/query.
The user has the option to label as "unsatisfying" the search results for a query: the backend will then instruct Scrapy to scrape the search results of each site for the provided user query.
It performs additional computation on the user's query such as pagination, query reranking, sorting, "more like this" similarity feature and input sanification.
It also adopts some security measures like using a rate limiter to protect from DDOS attacks. See Section 4 for a more detailed explanation of how the backend works.

2.4 Vue.js 3

3 Retrieving and indexing documents

Retrieving and indexing documents is the core of any search engine. In this section we will describe these two processes in detail.

[1] Banana is not an actual trademark. We thought it was funny xd
[2] Scrapy: See website
[3] Solr: See website

3.1 Crawling process

Retrieving documents from the web is a complex process. The crawling process is used to retrieve documents from the web. This is done by a crawler (also called spider), that automatically visit web pages and read their content. When a crawler visits a web page, it extracts the links to other pages and adds them to the list of pages to visit. This process is repeated until the crawler has visited all the pages it can find. In the case of our project, we used the Scrapy framework to crawl the web pages.
In order to properly crawl a website we need to understand how it is structured. For this reason, this was a fundamental part in the project to decide which sites to crawl. Some websites are not meant to be crawled easily.
An ideal website to crawl would have a list of links to all the pages to crawl (all the creators using the site). SubscibeStar is a perfect example of this: the whole website is static and the creators are listed. The only thing to do is visit every site in the list on every page and extract the information. This is not the case for most websites, and we had to find a way to crawl them.
Ko-fi, for example, does not list all the creators, so we had to find another way to crawl it. Since creators have tags associated with them, we decided to use them and search by category.
Patreon does not even have categories, so we decided to query the site using random super-generic keywords (like "music" or "art"). This way we are not sure to find all the creators, but at least we can crawl a lot of them.
The crawling process is not always straightforward, and we had to make some compromises. When we crawl a website we are sending thousands of requests to a webserver, and we need to be careful not to overload it or get banned. Some websites have a limit on the number of requests per second. Between the three websites we crawled, hrefhttps://www.ko-fi.com/Ko-fi has the most restrictive policy.
We used some good practices to safely crawl the websites. To avoid sending too many requests at the same time, we used a delay between requests. For the case of hrefhttps://www.ko-fi.com/Ko-fi, we set the `DOWNLOAD_DELAY` to 3 seconds, which means that the crawler will wait a random interval between $0.5 \cdot 3$ and $1.5 \cdot 3$ seconds before sending the next request since the `RANDOMIZE_DOWNLOAD_DELAY` setting is enabled (by default).
Another good practice is to use a *user agent* to identify the crawler. This is a string that is sent with every request, and it is used by the webserver to identify the client. The *user agent* contains information about the client, like the operating system, the browser, and the version. To be more clear, an example of user agent is: `Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0 Safari/537.36`.
At every request, our crawler uses a random user agent thanks to the `scrapy.downloadermiddlewares.useragent.UserAgent` middleware.
Another problem we faced is that some websites are javascript-heavy, so we had to wait some time for the page to load. To solve this problem, we used the `scrapy-playwright` library, which is a middleware that uses the Playwright library to interact with the browser. It simulates a real user using a real browser, so it can wait for the page to load and interact with it. We used it to wait until a css selector was present on the page, but it can do much more, like clicking on buttons or filling forms.
The spiders we coded have a common structure, they can take as parameter a list of tags to search, a list of urls of creators pages, or a list of queries to perform on the website. The default parameters are used when we want to crawl the whole website. The spiders have a method `parse` that is called on the pages listing artists. This method extracts the urls of the artists pages and calls the `parse_artist` method on them. It also extracts the urls of the next pages and calls itself on them. The `parse_artist` method is the one that extracts the information from the artist page. All the information are stored in the same format tanks to the `artist_dict.make` method. The info we save for each artist are:

- `site`: the website we crawled (ko-fi, patreon, subscribestar)
- `page_link`: the url of the artist page
- `artist_name`: the name of the artist
- `artist_image`: the url of the artist image
- `artist_banner`: the url of the artist banner

- `bio`: a short description of the artist
- `bio_long`: a more detailed description of the artist
- `amount_post`: the number of posts the artist has
- `amount_subs`: the number of subs the artist has
- `price_tiers_title`: a list of all the price tiers titles
- `price_tiers_monthly`: a list of strings of all the price tiers amounts per month
- `price_tiers_monthly_chf`: a list of all the price tiers amounts converted to CHF
- `price_tiers_description`: a list of all the price tiers descriptions
- `tags`: a list of tags associated with the artist
- `socialmedias`: a list of social media links
- `indexDate`: the date of the crawling

All the crawled data can be outputted by scrapy in a json file, that we use in the next section to index the data.
To run the spiders, navigate to the `crawler/crawler` folder and run the following command:

```
scrapy crawl <spider-name> [OPTIONAL PARAMETERS] -o outputfile.json
```

where `<spider-name>` is the name of the spider you want to run, and `outputfile.json` is the name of the file where the data will be saved. The optional parameters can be passed as `-a tags=[...] -a artists=[...] -a searches=[.`

3.2 Indexing process

Once the data is crawled, we need to index it to then be able to search it. We use the Apache Solr search engine to do so.

4 The glue that keeps everything together: the backend

4.1 Automatic scraping

4.2 Handling user queries

5 Presenting documents to users: the frontend

6 User evaluation: la merda

7 Conclusion