Information Retrieval **December 11th 2023**

Students: Kyla Kaplan, Elvira Baltasar

Contents

1. Introduction

Throughout this report, we will share in detail the objectives outlined, the methods implemented, the challenges overcome and, above all, the knowledge and skills acquired during the implementation of this project.

2. KYBE

In our project we provide the possibility of searching for articles related to artificial intelligence. With recent significant increased interest in AI it is quite valuable to have an efficient search engine for compiling key articles related in the field of AI. Special attention has also been paid to the aesthetic and visual presentation of the project. The name "KYBE" is made with the union of some letters of the names of the two members of this project, Kyla and Elvira.

3. Tech stack

1. Scrapy

In the development of our project we have used the powerful tool Scrapy, an open source framework in Python designed specifically for efficient data extraction from websites. We have used it to collect data from three pages, "AI TIME JOURNAL", "Analytics Vidhya" and "MIT News", then this information has been stored in different .jsonl files.

2. PyTerrier

PyTerrier is an open source library in Python designed specifically to simplify the development of information retrieval (IR) systems and search engines. Built on the Terrier platform, PyTerrier offers an easy-to-use interface that allows developers to quickly implement various IR functionalities, such as indexing and document retrieval. We used it to index the data and process queries.

3. Vue.js / Flask / MongoDB

The visual part of the project has been one of our main concerns during this process as it conditions the user experience and provides a good understanding of the use of the page. It is implemented with Vue.js, a progressive JavaScript framework used to build interactive, single-page user interfaces (SPA). It provides an organized structure for the development of reusable components, facilitates the creation of efficient and maintainable web applications by providing tools and patterns for state management, declarative DOM manipulation, and two-way data linking. In addition to Vue, we have included a npm library called Axios to catch the links/requests between the back-end and front-end.

Flask is a lightweight and flexible web development framework for Python, designed to facilitate quick and easy creation of web applications. In our search engine project, we use Flask to build the user interface and manage client requests to the server. Flask provides a minimalist yet powerful architecture that allows us to focus on the specific logic of the search engine without worrying too much about low-level details.

For our database, we have chosen to use MongoDB, which is a document-oriented NoSQL database mangement system. In terms of our search engine, MongoDB is used as an interim storage solution between going through the different steps of crawling, indexing and clustering our collected data. It's perfect for this use, as it allows for "un-organized" data, that may not often have all of its respective fields.

4. The fundamental processes of information retrieval: crawl and index

1. Crawling

This process is the first fundamental step in building a search engine. It consist of exploring and collecting information from several websites. A "crawler" or "spider" is an automated program responsible for browsing web pages, following links and downloading the content for further processing. During this process, relevant data such as text, links, images, (and more), are extracted and stored in a file for its analysis and classification.

In our project, first we needed to scrape the links of the articles of each page, store them in one .jsonl depending on which page they are from and then get into every link to get all this data for every article:

- `first_paragraph`
- `article_tittle`
- `author`
- `date_published`
- `tags`
- `article_url`
- `article_image`

As an example, here is part of the spider for the MIT Artificial Intelligence website where we get the data mentioned before:

```
def parse(self, response):
        #Starting point
        page_body = response.xpath('.//div[@class="page--section--inner"]')
        #...
        # Aggregate article title
        article_title = page_body.xpath('.//span[@itemprop="name headline"]/text()').get()
        #...
        # Get tags
        tags = []
        tag_body = response.xpath('.//ul[@class="news-article--topics-list"]/li')
        for t in tag_body:
            tag = t.xpath('.//a/text()').getall()
            tags.extend(tag)

        #...
        yield{"text": first_paragraph, "title": article_title, "author": author, "date":
            date_published, "tags": tags, "article_url": article_url, "img_url":
            article_image}
```

As seen in the final *yield*, we return the necessary fields for our data indexing and final search engine. Then, we store it in three seperate .jsonl files.

2. Indexing

The next step after crawling is organizing and structuring the information previously collected. The data extracted is now processed and stored in an index that works as a ¨jumbo-database¨ for efficient and fast search via an ´inverted index table´ provided by the PyTerrier library. This indexes all of our data, and can be (partially) found in the following file in our project:

```
'../indexing/db/output.json'
```

In this file, we have a data frame with two fields, `docno` and `text`. We finished the crawling step with three output jsonl, the first step now is to merge all of them in just one.

Some of the data fields we collected are not worth indexing them, for example, images and/or urls, we just join every other field into one called text and make a unique docno for every entry:

```
    for entry in data:
        text = "{} {} {} {} {} {}".format(
            entry["text"],
            entry["title"],
            entry["author"],
            entry["tags"],
            entry["year"],
            entry["month"])
        text = re.sub(r'[^a-zA-Z0-9]', ' ',text.replace("None", " "))
        text = re.sub(r'\s{2,}', ' ', text)
        entry["text"] = text

    return data
```

Now is the moment of indexing the returned data:

```
    indexer = pt.DFIndexer('../indexing/index_ai', overwrite=True)
    index_ref = indexer.index(preIndexTable["text"], preIndexTable["docno"])
    index_ref.toString()
    index = pt.IndexFactory.of(index_ref)
```

The indexed data now is written in this file *'../indexing/indexai'*.

5. Features

1. Simple features

1.1. Results snippets

As we have mentioned previously, one of the elements we crawled from the pages is the first paragraph of each news article:

```
    #...
    # Aggregating the first paragraph
    first_paragraph = ""
    for p in
        page_body.xpath('.//div[@class="news-article--content--body--inner"]/div/p'):
    # Concatenate the text content of the paragraph
    first_paragraph += p.xpath('string()').get()
    first_paragraph = strip_first_paragraph(first_paragraph)
    #...
```

In the context of search results, these first paragraphs represent the brief description that accompany each link and provides a preview of the content in a kind of "Google style".

1.2. Grid layout

We also implemented a view of the results with a grid layout, having three news "blocks" per row instead of one, as the list layout. You can see this feature in Figure 6

2. Complex features

2.1. Results clustering

Clustering is a method in computing that groups similar data sets together. Usually, clustering is performed against a user query, and modeled accordingly. However, for our project's use-case, we have implemented clustering s.t. it finds the most relevant/most common topics in our whole database, hence creating a *MostPopularTopics* list. This can be found in the image below, or alternatively by following the path *.../menus*.
If needed, we are able to render more topics.

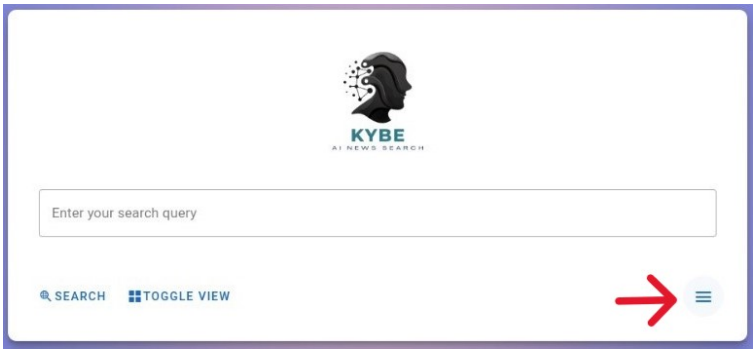

Figure 1: Topic menu location

6. The magic hidden: the backend

1. Queries

Once we got the user's query from the frontend, it is time to process it. This part is a bit technical unfortunately. First we call this function:

```
def getQueryResult(index, query, db_objs):
    bm25 = pt.BatchRetrieve(index, num_results=10, wmodel="BM25")
    queries = pd.DataFrame(
        query,
        columns=["qid", "query"],
    )
    #...
```

This function is located in the folder `../backend/controllers/indexing.py`, and the interim `results` it creates look like this (shortened for practicality, and it also prints in the terminal):

```
      qid  docid  docno  rank     score query
0  q1   4367  d4368     0  12.281181 lstm
1  q1   4664  d4665     1  12.001864 lstm
2  q1   2418  d2419     2  11.720492 lstm
3  q1   4573  d4574     3  11.709748 lstm
4  q1   5991  d5992     4  11.587412 lstm
5  q1   4196  d4197     5  11.390737 lstm
6  q1   4434  d4435     6  11.390737 lstm
7  q1   4822  d4823     7  11.390737 lstm
8  q1   5857  d5858     8  11.378843 lstm
9  q1   4142  d4143     9  11.200628 lstm
...
```

The query `lstm` was passed in the above example. The function itself uses the BM25 model (part of the PyTerrier library) to perform queries in a search engine. The results are formatted using the `retrieve_info()` function, which adds the necessary additional data from the database to the

information and returns it in a JSON format to be rendered. This can be seen in the following image, which is an interim screenshot from the terminal of the results (unformatted):

Figure 2: Screenshot from terminal in the backend

In this function, we also incorporate the use of a *threshold*. This *threshold* is applied by multiplying the score of the most relevant document (i.e. its score) for the query. Consequently, we keep only the data whose rank is within the range defined by this multiplication result and the rank of the first result. This might make the results seem shorter than they are, by only keeping the truly most relevant ones w.r.t. the query.

For the part on clustering, we have a file in the backend responsible for all of the clustering logic, and containing all the necessary methods `../backend/controllers/clustering.py`. Below is a screenshot from the terminal of the interim first few lines (head of the dataframe), and in the right-most column, you can see it being aggregated according to topic(s) which are most popular.

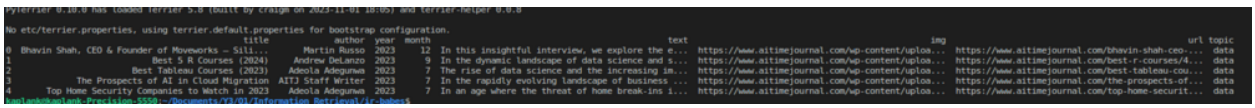

Figure 3: Screenshot of clustering from terminal in the backend

```
    #...
    threshold = thresholding(cdf, threshold=0.15)
    #...
    cdf_filtered = cdf[cdf["score"] >= threshold]

    if len(cdf_filtered) == 0:
        return "[]" # Return an empty array as a string
    elif len(cdf_filtered) == 1:
```

```
        # Return a single JSON object as a string
        js = cdf_filtered.to_json()
        print(js)
    #...
```

7. In everyone's sight: the frontend

In an IR system, the most important aspect is not just the search engine, but also a clear page design and UX/UI for the user to enter queries and retrieve/render the results respectively. You can see that in the following images of our website:

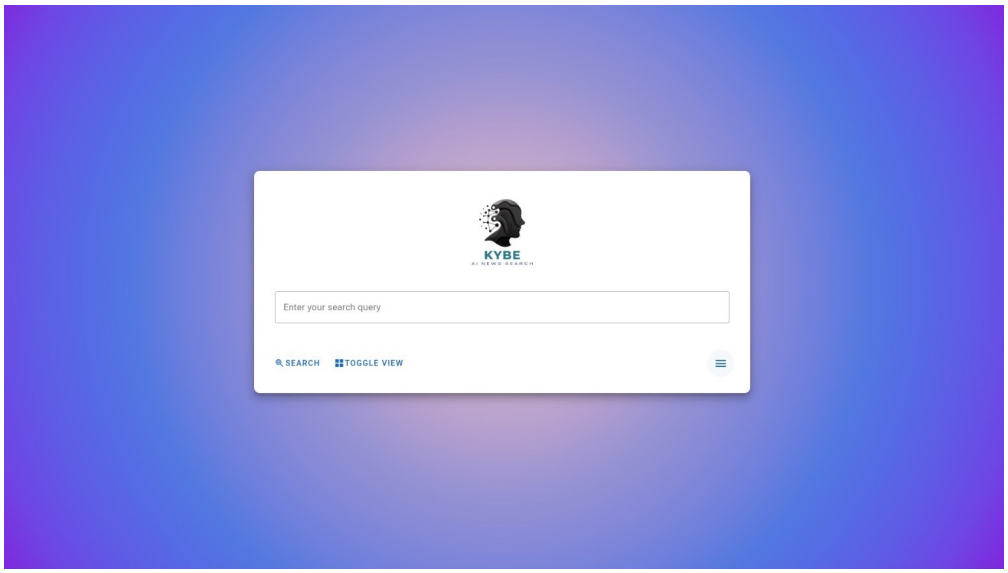

Figure 4: Main page

Here you can see that, in our frontend, we have a search bar where the user writes the query, a button to submit it (search), and the toggle button. In the right there is a third button that would lead to the screen in *Figure* 7 to see the most popular topics.

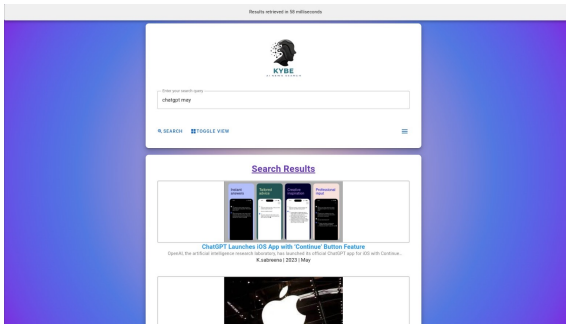

Figure 5: Results page

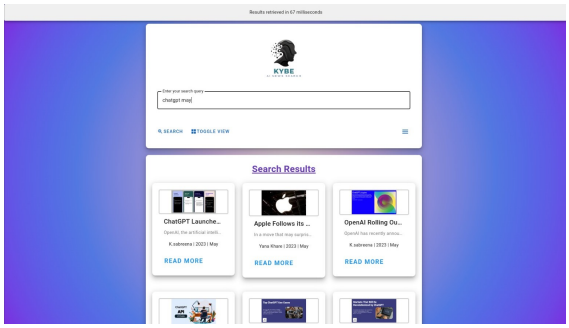

Figure 6: Results page in toggle form

As you can see in the *Figure* 5 once the query is submitted, another block will appear so the results are rendered there. Clicking in the button `"toggle view"` you will get the results rendered as in *Figure* 6, on a grid view.
All this functionality has been created using Vue.

Figure 7: Clustering page

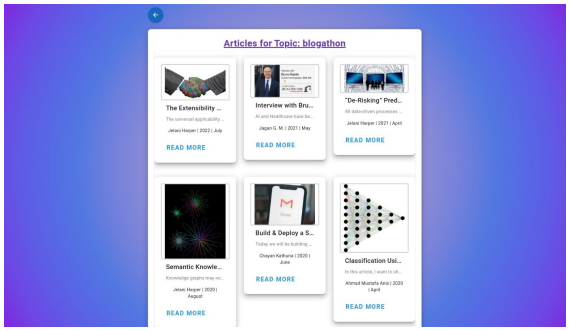

Figure 8: Clustering results page

Both the logo and the favicon were made by us:

Figure 9: Favicon icon

Figure 10: Custom logo

8. User evaluation

Having already built our machine we must evaluate it. As we are going to use an experimental evaluation, we will decide the aim of the evaluation: if the search results are satisfactory for the user and if the appearance is pleasant and attractive.

Now we must formulate our hypotheses:

- The interface of the website is intuitive and easy to use.
- Please look up the term "mark zuckerberg". How relevant were the search results?
- Please look up the term "chatgpt". How relevant were the search results?
- Please look up the term "chatgpt may". How relevant were the search results?
- The initial presentation of the search results is clear and easy to understand.
- The toggle (grid vs. list) view of the search results is clear and easy to understand.
- The clustering by most popular terms worked and clicking on the topics provided expected results.
- The website provides useful and reliable information.
- How likely would you recommend this page to a friend or colleague?

Having these statements we built a form in order to mark each of the statements from 0 to 10 being 0, "I don't agree at all", and 10, "I fully agree". Click here to be redirected to the form.

We asked people to complete the form and then we analyzed the results:

Figure 11: Average result score for each question

At the end of the form we also added an open question to get feedback from the users *"If there's one thing you could change or improve about the search engine, what would it be? Are there any additional features or functionalities you would like to see added?"*
And here is the feedback we got:

Figure 12: Additional feedback

9. Use of the program

In order to run the program, you will have to download a .zip version from GitHub (link can be found in the Appendix). Prior to running any part of the program, however, you will have to install all of the necessary packages, libraries, and modules. Here are some, but not all: MongoDB (with a server running on `localhost:27017`), Python3 latest version (not just Python), Flask, Yarn (and more).

In three terminals, launch the following:

1. The first terminal will have to run MongoDB: `mongod --dbpath mongodb`
2. The second terminal will have to run the backend. Firstly, one has to navigate to the folder `backend`, and from there run `flask run`. This command will also point out all the necessary libraries to install as an error message within the terminal.
3. The third terminal will run the frontend. Here, one will also have to navigate to the `frontend` folder, and from there run firstly `yarn install`, and then `yarn serve`.

Thus, the final website can be seen at `http://localhost:8080/search`. For any troubleshooting, please refer to the respective library documentations online.

10. Final thoughts

In summary, we have successfully developed a search engine for AI which is able to crawl from pages and then, based on a query gave by the user, render related documents. In addition the visual part is pleasant and clear.

1. Limitations and Suggestions for Improvement

Some of our current setbacks are, firstly in the clustering implementation, the time it takes to load (on average $\approx$ 20 seconds), secondly in the toggle view, where the images are too small to be properly seen. As potential future improvements, we could have a calendar picker so the user can pick the date they want the articles from. Also a more interactive clustering page such that is possible to filter the results, unlike now, as in Figure 8 where we currently just have all the articles being rendered. Another suggestion for future technical implementation would be a better connection between MongoDB and the back-end, as for the moment it is a bit shallow. Additionally, scraping more pages to make our website more powerful and make the images appearance better.

A. Appendix

Here is the link to our projects GitHub where the final code is.