TüR – Tübingen Retrieval

Josef Müller[1], Okan Coskun[1], Lukas Seehuber[1], and Alexander Phi. Goetz[1]

[1]University of Tübingen firstname.lastname@student.uni-tuebingen.de

November 14, 2024

1 Introduction

This report describes the development of our search engine **TüR – Tübingen Retrieval** for the course Modern Search Engines lectured by Prof. Dr. Carsten Eickhoff at the University of Tübingen. The goal was to build a search engine focused on Tübingen related content in the English language. Our additional design goals were a seamless user-experience in the front end, responsive information retrieval, as well as efficient computation during the web crawling stage and database handling. The following chapters give an overview of our implementation details and conclude how our design decisions contribute to our goals.

2 Implementation Details

Our Search Engines Repository[1] is split into the front end user experience part and the back end engine part. The first found in the repository's `client` and the latter in the `engine` folder, respectively. Another separation between these two are the technology stacks employed in each. The `engine` is written in Python 3.11 and the front end was built using `Vue.js` [1] and `Tailwind CSS` [2].

This section will first focus on the bulk of work in the `engine` folder and finish with the contents of the `client`. In the engine we find the *document processing pipeline*, the server's *JSON endpoints*, *summarization* capabilities and notably the modified BM25 *ranked document retrieval*.

2.1 Document Processing Pipeline

We implemented a two-fold document processing pipeline. One aimed to crawl and one to load and process already crawled documents, as shown in Figure 1. The pipeline that starts with crawling documents from the web is called *online* as it requires an internet connection, whereas the one called *offline* allows to re-process and re-index already crawled documents.

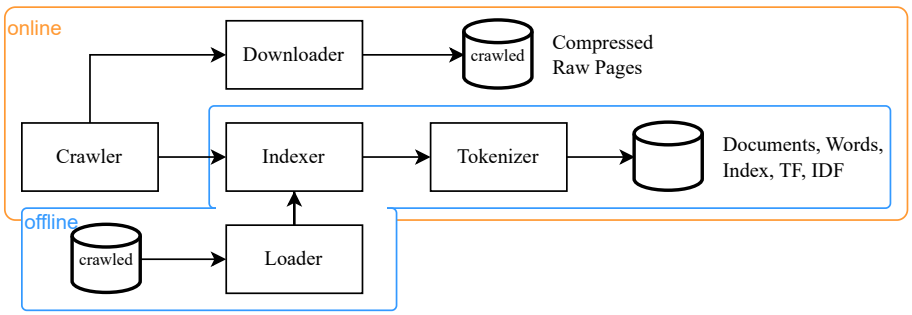

Figure 1: Document Processing Pipelines for on- and offline document processing.

Besides the separation and overlap of both pipelines, we want to highlight the role of the database used in this project. We chose `DuckDB` [3] as our storage solution, as it is a modern embedded

[1]`https://github.com/am9zZWY/TueR/tree/HODORv1`

analytical database that was built to be close to Python. For example, it allows for direct use of Pandas DataFrames inside queries and easily produces them as well.

The pipeline stages themselves are found in the files `crawl.py`, `download.py`, `index.py`, `tokenizer.py` and the base class in `pipeline.py`. The main goal of the design was to have a composable architecture of pipeline stages that process data and send their results to the next stage. All important computations happen in the `process` methods, which then pass their results to the next stage.

2.1.1 Web Crawling and Indexing

The `online` pipeline starts with our **Crawler**. Similar to the lecture, we start of with an initial frontier found in the global constant `SEEDS`, which is retrieved through `aiohttp` requests in parallel. A further optimization was to keep track of which domains are being crawled and limit the number of requests to any single domain. Each crawler then selects a new URL from the queue, prioritizing those from domains that are not currently being crawled.

This allows us to also set the headers and a rotating list of user agents referencing this course. Before any page content gets processed, we respect the `robots.txt` and check the language of the page to match whether it could be of interest. The last part is done by either checking the HTML language-tag and using `ELD` [4] as a fallback to detect the language of the content. We note all visited and about to be visited websites for further consideration. Many webpages also use client-side rendering, which requires a browser to execute the JavaScript code. Therefore we experimented with `Playwright` [5], a library to control a headless browser. This approach was discontinued because of several problems with our multithreaded crawling approach.

Next up is one of the following, either the **Downloader** or the **Indexer**. The first is used to store an encountered website in the database table `crawled` by compressing the raw page text. This can be used later on in the `offline` pipeline which would start with the **Loader**, extracting the compressed texts and handing them off to the next stage. The second, namely the **Indexer**, using `BeautifulSoup` [6], extracts the title and description and saves them in the database table `documents`. This will also be useful, when we want to present a page to the user.

2.1.2 Tokenizer and TF/IDF pre-computation

After including a website to the relevant documents, the **Tokenizer** pipeline step is executed. To access said content in a more targeted fashion, we check for HTML tags that suggest the main content we want to consider for further use. Moreover, we acknowledge that the site's title and description, as well as image alt texts, contain relevant tokens that may appear in later user queries. Next, the function `preprocess_text` runs our modular pipeline, cleaning up the document's content by converting Unicode to ASCII format and utilizing Python's regular expression capabilities to favor pure text.

Then we tokenize and lemmatize via SpaCy[7]. We use the same library to remove stop words, punctuation, and whitespace. These tokens are then stored in the database's `tf` table, where the Term Frequency is stored per document. This table also acts as our "inverted index" as thanks to the table's constraints and index structure we basically get the index structure discussed in the lecture for free.

At last, when all documents are tokenized, we compute the final Inverted Document Frequencies (IDF) for each token. This is done in the database directly and corresponds to the formula given in the lecture.

2.2 Query Processing and Ranking

In our project, we implemented a document ranking system that utilizes the well-known Okapi Best-Match Model Version 25 (BM25) algorithm, enhanced by incorporating word similarity. Following pre-processing, the query is expanded with similar words and the ranking is based on the document relevance regarding the expanded query. We also incorporated a weighting system to ensure that

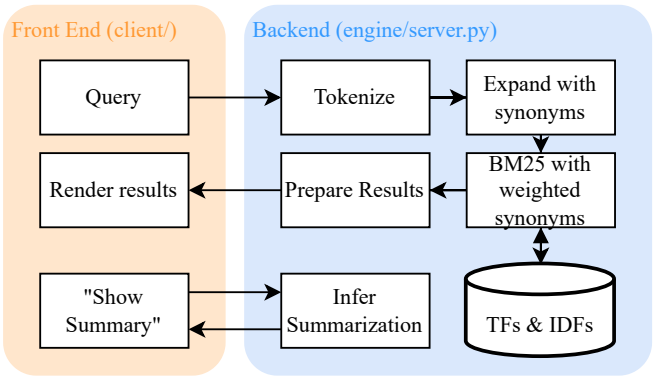

Figure 2: User Interaction

added words contribute less to the score than original query words. The ranking runs on a union of documents containing at least one of the pre-processed expanded query words. This ensures we run the computation on a subset of documents.

We employ an exponential decay function to determine the number of similar words to be generated,

$$N = M \cdot e^{-D \cdot |Q|},$$

where N is the number of similar words to use, M is the maximum number of similar words per word (default is 7), D is the decay rate (default is set to 0.08 to restrict the query length to approximately 40 words), and $|Q|$ is the query length after preprocessing.

Each token from the query is expanded with similar words using GloVe embeddings [8]. In our case, we use the `glove-wiki-gigaword-100` model, which is a GloVe model, pre-trained on the Gigaword 5 and Wikipedia2014 corpora, provided by the `gensim` library [9]. Words with similar contexts in the corpus will have similar vector representations. We only considered similarity scores of at least $min(similarity) = 0.7$ to mitigate the risk of topic drift. This was necessary as our process involves not just generating synonyms, but also words with contextual similarity.

We have adapted the BM25 formula to rank a document d given a query Q as follows:

$$\text{score(Q,d)} = \sum_{w \in \text{query}} \text{weight}_w \cdot \text{idf}(w) \cdot \frac{\text{tf}(w) \cdot (k_1 + 1)}{\text{tf}(w) + k_1 \cdot (1 - b + b \cdot \frac{L_d}{\tilde{L}})}.$$

The tf and idf values were pre-calculated and retrieved from the database described above, as shown in Figure 2. The parameters k_1, handling term repetition, and b are BM25-specific and set to defaults influenced by Schuth etal.[10]. L_d is the document length, and $\tilde{L}$ is the average document length, used together with b for document length normalization as discussed in the lecture. We calculate the BM25 score separately for the original query terms and the expanded query terms. Their sum respectively is then added together, which yields the final score of a document d given the query Q.

The weight for words in the original query is set to 1, unless we could not find any similar words for it, which means it is uncommon and most likely a "Tübingen"-related term. Given it also exists in our index, we increased the weight to be $\lfloor 7 \cdot min(similarity) \rfloor = 4$, which makes up for the missing similar words but also increases the importance of documents that contain such unique words. The `weight` for words of the expanded query, i.e., the similar words, is calculated by dividing their GloVe similarity value by the number of overall similar words in the expansion.

By combining the BM25 algorithm with a word similarity component, we tried to make sure that we also take documents into account, that might not perfectly fit to the query but might heavily relate to it and thus increase the recall of our search engine. However, we also made sure not to fall into the trap of topic drifting, which can happen by taking too many similar words into consideration such that the original query intent gets lost, losing heavily in precision of the search results.

2.3 Back and front end

We implemented the server back end using `Flask` [11], a micro web framework, which exposes endpoints to the front end to handle the search, preview, and summary of pages.

For the front end, we wanted to ensure that new users would find themselves comfortable by adopting a familiar layout while introducing new, unobtrusive features to improve the search experience. Those features encompass a preview of each webpage, a summary, and the option to use a dyslexic font, as can be seen in Figure 3.

For the preview, the server fetches a fresh site upon receiving a request and returns the HTML-Code. This was necessary, since many websites block the direct use of `<iframe>`-tags with a URL.

A similar approach was done for the summary. When the button is clicked, the server receives the page-id and pulls the website's content from the database. We then summarize the content using `pegasus-xsum` [12], a model optimized for generating abstract summaries. This method worked well for websites that pre-render most of their content on the server, whereas websites with client-side rendering provided less content for the summary, which caused the model to give inaccurate summaries.

We further improved the user experience by listing the last three searched queries, which can be inserted back into the search box after clicking on them. Additionally, we implemented a toggle for Comic Sans to improve the user experience for people suffering dyslexia [13].

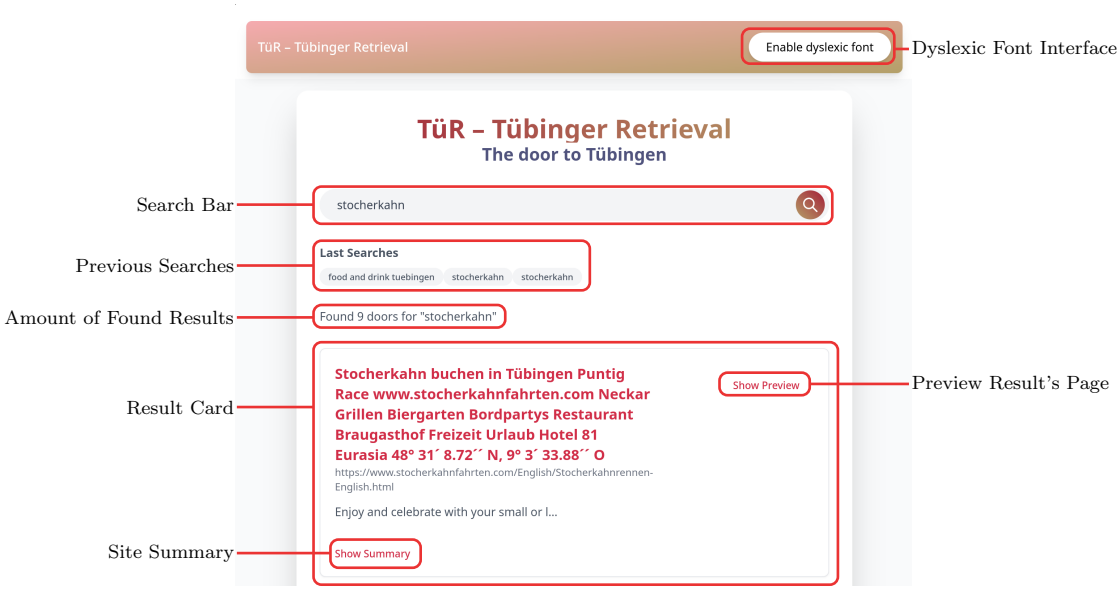

Figure 3: TüR's front-end experience

3 Conclusion

We achieved our aforementioned design goals by building a multithreaded crawler, the offline TF and IDF computation in the database and by expanding the BM25 with additional query terms utilizing not only synonyms but different similar words that we got through GloVe embeddings. This yields a higher recall in our results with the trade-off in precision, which we tried to counteract by limiting the amount and weight of additional terms. Here we see possibilities for improvement, as our current retrieval potentially runs on a large subset, when there are a lot of common query words. To remedy this issue, we could have implemented a more sophisticated document retrieval before ranking. As for the front end we had the ideas to include a more readable option for dyslexic users, the ability to let the search engine summarize the result page's content and preview the page itself.

During the project, we faced several issues. Initially, we inefficiently recrawled sites instead of downloading them. We overcame this obstacle by implementing a downloader that allowed us to post-process the downloaded pages in the `offline` pipeline. Furthermore, our multithreaded approach restricted the use of tools like Playwright, limiting access to client-side rendered content. Also, while

we strived for an independent, composable pipeline-architecture, we ended up having strong dependencies between some pipeline stages. But eventually, we achieved a fast crawler implementation, when set to the matching thread amount.

DuckDB was valuable for the analytical part of computing TF, IDF and ranking. However, it wasn't able to handle concurrent connections from different processes to the same database, which was problematic for transactional tasks like adding newly discovered pages to tables. On the other hand, we greatly benefited from its seamless integration of Pandas DataFrames.

Overall, we deepened our understanding of how modern search engines approach the plethora of difficult tasks and aimed at implementing straightforward yet effective solutions to our tasks. We also gained respect for those who research and improve search engines and adjacent tasks.

References

[1] Evan You and Vue.js Contributors. Vue.js, 2024. URL `https://github.com/vuejs/core`.

[2] Adam Wathan and Tailwind Labs. Tailwind css, 2024. URL `https://tailwindcss.com`. A utility-first CSS framework for rapidly building custom user interfaces.

[3] Mark Raasveldt and Hannes Muehleisen. DuckDB. URL `https://github.com/duckdb/duckdb`. `https://github.com/duckdb/duckdb`.

[4] Nito. Efficient language detector in python, 2024. URL `https://github.com/nitotm/efficient-language-detector-py`. Fast and accurate natural language detection. Detector written in Python.

[5] Microsoft. Playwright, 2024. URL `https://playwright.dev`. A framework for reliable end-to-end testing for modern web apps.

[6] Leonard Richardson. Beautiful soup, 2024. URL `https://www.crummy.com/software/BeautifulSoup/`. Library for parsing HTML and XML documents in Python.

[7] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. 2020. doi: 10.5281/zenodo.1212303.

[8] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL `http://www.aclweb.org/anthology/D14-1162`.

[9] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.

[10] Anne Schuth, Floor Sietsma, Shimon Whiteson, and Maarten Rijke. Optimizing base rankers using clicks. In *Proceedings of the 36th European Conference on IR Research on Advances in Information Retrieval - Volume 8416*, ECIR 2014, page 75–87, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 9783319060279. doi: 10.1007/978-3-319-06028-6_7. URL `https://doi.org/10.1007/978-3-319-06028-6_7`.

[11] Pallets. Flask documentation (3.0.x), 2024. URL `https://flask.palletsprojects.com/en/3.0.x/`.

[12] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization, 2019.

[13] British Dyslexia Association. Dyslexia friendly style guide, 2024. URL `https://www.bdadyslexia.org.uk/advice/employers/creating-a-dyslexia-friendly-workplace/dyslexia-friendly-style-guide`. Guidance on creating dyslexia-friendly written material and workplace environments.