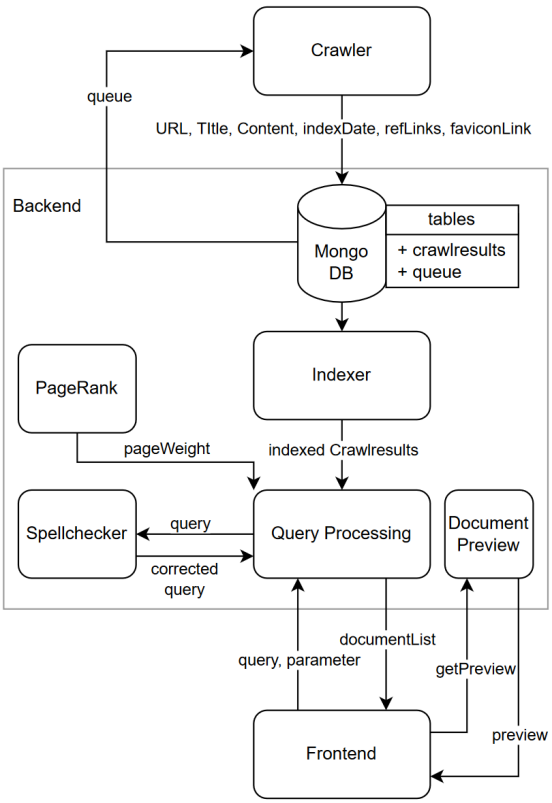
Project Report: Tü-be-fair MSE SS24

Jannik Brandstetter Hoang An Nguyen Nico Martin Julian Borbeck

Abstract

Introducing Tü-be-fair, a fast, high precision modern search engine designed to enable users with greater control over their query results. Key features include high-performance query search across approximately 18,000 Tübingen-specific websites, and fully customizable parameters such as diversity, fairness, and PageRank weight, which are easily adjustable by the user. Tü-be-fair also offers spellchecking and instant feedback suggestions for misspelled queries, favicon support, transparency through relevance score display, and loading of website preview via a backend proxy. These functionalities collectively enhance user experience by providing precise, relevant, and user-tailored search results.

1. Project Structure



2. Crawler

The following section details the data retrieval process and design principles underlying our web crawler.

2.1. Architecture

The crawler is constructed using python and includes multiple key components, each using specific libraries and serving a different purposes. The main component, crawler.py, fetches the content of web pages. For URL prioritization, url_ranker.py ranks the extracted links based on relevance to predefined keywords, in particular to rank Tübingen related content higher than other ”random” content.

To maintain the integrity of the extracted links and to remove unnecessary processing, link_checker.py validates discovered URLs and filters out specific data types and duplicates. The storage and management of crawled data are handled by db.py, which stores the extracted information in a MongoDB database.

Before data is stored, it undergoes preprocessing via pre_processing.py, which cleans and normalizes the extracted text. The Max Heap data structure, crucial for URL ranking, is implemented in maxHeap.py.

To perform the ranking process, the ranking_keywords.txt file contains the keywords used by the URL Ranker to score URLs. Additionally, the seed.txt file provides the initial set of URLs, ensuring the crawler begins with a broad coverage of Tübingen-related information.

2.2. Innovations

2.2.1. Tübingen Content Filtering

The crawler implements a content filtering mechanism by verifying the presence of the keyword ”Tübingen” within the fetched content. This process ensures that only pages containing a reference to Tübingen are retained. Consequently, this establishes a selection policy whereby websites must be at least loosely related to Tübingen to be deemed relevant and thus saved.

2.2.2. URL RANKING

The URL Ranker assigns scores to URLs based on predefined keywords related to Tübingen. The algorithm used to rank a link works as follows:

- url: the URL to be ranked
- depth (d): the depth of the URL
- keyword_score (α): the score increment for each keyword found in the URL, default value is 0.1
- K: the list of keywords
- L: length of the URL
- δ(k, url): indicator function that equals 1 if keyword k is in the URL, 0 otherwise

$$S = \frac{1 + \sum_{k \in K} \delta(k, \text{url}) \cdot \alpha}{L \cdot d}$$

$$\delta(k, \text{url}) = \begin{cases} 1 & \text{if } k \in \text{url} \\ 0 & \text{otherwise} \end{cases}$$

The algorithm ranks websites higher, which contain terms located in the keyword list, to maintain a rather "efficient" crawling. This minimizes the time the crawler takes to crawl "unnecessary" websites not related to any of the keywords.

2.2.3. LINK VALIDATION

In the first rounds of data crawling it became apparent, that many URLs were duplicates even though they had different "paths" such as anchortags or queries like "&page=x". There were also alot of pages which weren't reachable. As such we added a Link Checker which ensures that hyperlinks are valid and reachable before further processing, filtering out duplicates and specific data types such as URLs ending on ".mp3" or ".jpg" as we don't process them.

2.2.4. DATA PREPROCESSING

The preprocessing component cleans and normalizes text content (removing punctuation, lowercasing, lemmatizing etc..), removing HTML tags, JavaScript leftovers, and unnecessary stop words. It then returns the content as a list of tokens.

2.2.5. MAX HEAP FOR URL MANAGEMENT

The Max Heap data structure efficiently manages URLs based on their scores, ensuring the crawler processes the most relevant URLs first.

3. Query Processing

The query processing takes in a query, a document, the index and parameters to rank accordingly. It consists mainly out of the BM25 algorithm, but also incorporates Diversity and Fairness to rerank the results, to create a variety of different search results. Okapi BM25 can be summarised as seen in the lecture:

$$RSV(Q, d) = \sum_{q \in Q} IDF(q) * \frac{TF(q, d) * (k + 1)}{TF(q, d) + k * (1 - b + b(\frac{L(d)}{\overline{L}}))}$$

3.1. Innovations

3.1.1. DIVERSITY AND FAIRNESS

After computing the scores we rerank these scores based on diversity and fairness metrics. The general reranking algorithm we use is very similar to the one used in the lecture to rerank documents greedily with respect to diversity. We have extended this algorithm by extending the score to also include a per document metric for fairness.
Our Fairness metric is based upon the idea that for a ranking to be fair, the exposure for each item should have a linear relationship with its ranking score. As has already been established in the lecture, this is not the case. We have therefore decided to model the true exposure based upon rank position using an exponentially decaying function with parameters x : position in the ranking, $\lambda = 0.2$ decay rate and $e_0 = 1$: exposure at the first position. This yields the final following formula:

$$exposure(p) = e_0 \cdot e^{-\lambda \cdot p}$$

To measure the difference between our expected exposure based upon a linear relationship between ranking score, e_{expected} and exposure and the assumed real distribution function of our exposure based upon position in the ranking, e_{true}, in each step of the reranking we calculate a difference for each document $\Delta = \frac{e_{\text{expected}}}{e_{\text{true}}}$.

This difference is then normalized to a range of $[0, 1]$ using a sigmoid function.
The diversity for each document is also normalized to a range of $[0, 1]$ by scaling by the maximum possible diversity (the cardinality of a set of all possible words in all documents to be reranked).
The relevance is also normalized to a range of $[0, 1]$ by scaling relevance scores by the maximum relevance score.

This way we obtain per document per step of the reranking the normalized diversity $d(x)$, the normalized relevance $r(x)$ and the normalized fairness $f(x)$.
The final score for each document can then be calculated as a simple sum with the three weights α, β, γ, which are constrained, so that $\alpha + \beta + \gamma = 1$.

The final ranking score is calculated per document per reranking step as

$$s(x) = \alpha r(x) + \beta d(x) + \gamma f(x)$$

By tuning the parameters α, β, γ, the user can decide how much they would like to weight fairness, relevance and diversity per query.

3.1.2. Query Expansion

We initially implemented a Query Expansion Model but discarded it due to excessive padding and irrelevant term additions, such as "Taiwanese" and "Mongolian" for the query "Chinese." Instead, we opted for a high-precision search engine that returns results strictly containing the query terms.

4. Frontend

The frontend encompasses all elements visible to the user. This section outlines our user interface and the design decisions that informed its development.

4.1. Architecture

The frontend is a component-based application developed with Vue.js 3.0, utilizing Vuetify as the responsive UI library.

4.2. Innovations

4.2.1. User-friendly parameter tweaking

Users can easily adjust the parameters of our search engine through the settings button located to the right of the search bar. Here, they can customize options for Okapi BM25, diversification, fairness, and PageRank according to their preferences. The default parameters are set to strike a good balance between high-scoring results and diversity.

4.2.2. Pagination

When the user does their inital search, only the first 10 documents are retrieved at first to reduce the waiting time to a minimum. After that we load more documents in batches of the size 10 while the user scrolls using pagination.

4.2.3. Spellchecker

When the user queries the search engine we check the query for spelling mistakes. We do this by asynchronously sending a spellcheck request to the backend. When receiving confirmation and a correction for the query we still supply the user with the documents for their inital query but show them a correction suggestion that the user can accept or discard.

4.2.4. Favicons

When the backend responds with search query results, we asynchronously load the favicons for the provided sites using the crawled favicon URLs. Before making a direct request, we verify the existence of the resource. If the favicon cannot be found during crawling, we attempt to retrieve it from a default path. If the favicon is still unavailable, a default icon is displayed.

4.2.5. Preview and Search Term Highlighting

To display a preview of the provided sites, we asynchronously request a preview of the sites when displaying them through a backend request. This is not done from the frontend directly, but through a backend proxy to avoid CORS exceptions. When we get the preview, we highlight the search terms from our current query in the description by formatting them bold.

5. Backend

Our backend is built using the Python microframework Flask and encompasses all background processes, including those initiated when the application starts and those triggered by requests from the frontend.

5.1. Innovations

5.1.1. Spellchecker

The spellchecking is implemented using the Python library PySpellChecker, which compares the Levenshtein distance between each word in the query and words in a known dictionary. For enhanced accuracy, we have augmented the default English dictionary with all tokens from our inverted index.

5.1.2. Inverted Index

As a performance improvement to speed up retrieval time, we employewd an inverted index. This index is constructed as an object containing a dictionary of tokens, along with their posting list objects. We have implemented an AND based intersection search as described in the lecture, as well as a simple, set based or intersection search.
At the same time, as a performance improvement to the ranking algorithm, we also store the corpus size in the inverted index document, as well as the number of occurrences per token in each tokens posting list.
Additionally, to further improve performance, we have also implemented skip pointers per posting list, with the skip pointers being evenly spaced $\sqrt{|\texttt{posting list}|}$ apart.

5.1.3. User controlled Hyperparameters

In the query endpoint, the user can specify all ranking relevant hyperparameters, such as the BM25 hyperparameters, weights for the PageRank score and weights of fairness and diversity in the reranking.

5.1.4. PageRank

We have implemented the PageRank algorithm based upon a link matrix. Due to us being constrained by limited data quantities, we have decided to modify the PageRank algorithm to not give us a Rank per url, as we would not have found many links to one specific URL in our dataset. Instead, we deciced to use a per domain/ base url PageRank scheme by truncating all links to the base url before constructing the link matrix. This has yielded better results for scoring pages, but at the cost of a loss in resolution of the importance of single pages in the same domain.

5.1.5. Percentile scores

To give the user a better context of exactly how good each document was scored before reranking, and to enable the user to be able to better understand how reranking with different parameters might impact their search results, we also calculate the percentile for each document in the ranking before reranking the best n documents. This percentile is also visualized on the left side of the corresponding document.

6. Limitations and Further Improvements

The following sections discusses shortcomings of our project and suggest potential improvements that could mitigate or even resolve these issues.

6.1. Limitations of Data Crawling

6.1.1. Data Pre-Processing

During the HTML text pre-processing in the crawling step, we encountered significant challenges in removing cookie banners due to their lack of standardization. Consequently, queries containing words like ”cookies,” ”settings,” and ”accept” may yield unsatisfactory results, as these terms are commonly found on sites requesting cookie consent.

6.1.2. Javascript translation websites

Our crawler only scrapes pure HTML and doesn't execute javascript on websites. As such, some websites (even important ones like `https://www.tuebingen.de/`) which translate to english purely by javascript through a KI model couldn't be scraped by us. This leads to some important websites missing, as the english version is gate-kept behind javascript execution.

6.1.3. Data Size

Because our crawler is focussed on highly specific websites related to tübingen, we ”only” amassed a limited dataset of 18.000 sites, due to the time constraints. It is evident that a more extensive crawling process would yield a significantly larger set of potential results. Combined with our diversity and fairness tools, a larger dataset would enable the generation of more varied search results, thereby providing users with greater flexibility in tailoring the results to their preferences.

6.2. Limitations of Query Processing

6.2.1. Handling of no matching results

To enhance the speed of our query processing, we retrieve URLs from the database that contain all the words in the query. Currently, if a query yields no results, nothing is returned. A potential improvement is to implement an OR fetch as a fallback if the AND retrieval finds no URLs. This change could address the issue of our .tsv output only returning results for three of the five queries and could be easily resolved by applying the suggested solution.

7. Explanation of TSV Data

This section explains our TSV data, the reasons behind the results obtained, and potential fixes. Queries one, two, and four processed correctly. However, issues were encountered with queries three and five, which did not return any URLs. These problems can be explained as such:

Query 3 involved a simple spellchecking error. Our approach to handling such errors involves processing the query and returning zero results if nothing is found. Simultaneously, we check for misspellings in the backend, identify them, and send a signal indicating the misspelled term along with a suggested correction. The user can then easily search for the corrected string by clicking on the correction prompt displayed under the search bar. This design ensures that users can easily rectify and resubmit their queries for accurate results, but still maintain their original query if they deliberately chose so.

Query 5 presented a challenge due to our query processing strict requirement for AND matches between all words in a processed query. Consequently, no matches are found unless all words exactly match. An effective solution would be to initially perform an AND search, but if no results are returned, subsequently perform an OR search to yield partially relevant URLs. Additionally, implementing a ranking system for OR search results could prioritize the most relevant links, thereby enhancing the user experience by providing a broader range of useful information even when exact matches are unavailable.