Modern Search Engines
Group Project Report

Aline Breitinger, Moritz Christ, Lili Even, Tim Gerne and Jonathan Nemitz

July 21, 2025

---

1 Introduction

As part of the course Modern Search Engines by Prof. Dr. Carsten Eickhoff at the University of Tübingen, we were tasked with developing a search engine themed around the city of Tübingen delivering english content. Our work is explained in the following report. Our implementation can be found at the submitted GitHub link.

2 Web Crawling and Indexing

2.1 Web Crawling

The crawling process started with a frontier consisting of 145 pages related to Tübingen in different categories, such as tourist attractions, museums, restaurants, outdoor activities and the university and other educational centers. The initial frontier covered websites of concrete entities such as the city museum, to make sure that these will be present in the index. Furthermore, review and rating platforms such as "Tripadvisor" or "TheFork" were included, as these websites tend to include many links to relevant places and entities for a given city. We later discovered that such sites often restrict the majority of their content for small crawlers like ours, only allowing crawling by big companies.
The frontier of the crawler is a priority queue, where each element consists of three parts: a priority score indicating how relevant a website is for the crawling process, a depth score indicating how many websites lay between the website and the initial seed and the URL used for accessing the website. The crawling is done for one website at a time and starts by popping the frontier entry with the highest relevance score. The website undergoes several checks before being saved and indexed. It is first filtered by URL pattern to exclude unwanted file types. Then, it is verified whether the URL has been visited and whether access is allowed by robots.txt. For this RobotFileParser from the urllib module is used. Only English content is retained, and a SimHash algorithm checks for duplicates. The websites's content is retrieve using BeautifulSoup and language is detected using the langdetect module. For all websites the crawl delay is respected. These checks were implemented to retrieve websites that are relevant for our search engine while at the same time respecting the preferences of their owner. If all checks pass, the URL is saved, and its links are added to the frontier. The priority score for each of these links is calculated based on the website's depth and an added priority bonus if the link contains the word "Tübingen" in its URL or its associated anchor text. The depth of an entry is the depth of its parent entry increased by one.
As expected, a majority of the websites crawled with this process were found to have Tübingen in their URL. While this was desired, we wanted to see if it was possible to crawl a more diverse set of websites, while still focusing on Tübingen as a theme for the search engine. For this a second priority function was created. Additionally to the previous criteria it also increases the priority for websites with domains that have not been visited before. This bonus decays with the amount of visited URLs for that domain. Furthermore, each website inherits 20% of its parents priority score i.e. the website where its link was found. A small random bonus is added to enable more exploration. This bonus decays with increasing depth.

After every 50 crawling iterations the results were saved. This makes it possible to stop the crawling process at any time and then start it again from where it was stopped. This was especially useful in the beginning stages of the crawling process as some unexpected errors occurred that stopped the process (e.g. the appearance of special characters on a website). The code could then be adapted to handle these errors and the crawling could be continued without losing previously saved information.

2.2 Indexing

Once the URLs were crawled and stored, the next step involved extracting and preparing the content for indexing. For the current index, only the URLs crawled with the initial priority function were used, as they showed more promising results in our evaluation. For each crawled page, the raw HTML was parsed and non-informative elements such as layout structures, navigation bars, and repetitive content were removed to isolate the main textual content. This main content was then normalized through a preprocessing pipeline: tokenization, lowercasing, removal of stopwords and punctuation as well as stemming. The same normalization was later applied to incoming queries to ensure consistency.
Based on the cleaned and normalized content, two types of indices were constructed. First, a traditional inverted index was created, assigning each term to the documents it appeared in, along with the term frequency it appeared in each document. In addition to this lexical representation, a semantic index was built to enable vector-based retrieval. Using the all-MiniLM-L6-v2 model from SentenceTransformers, dense vector embeddings were generated for the extracted content of each document. These embeddings were indexed using FAISS(Facebook AI Similarity Search) [1], an open-source library optimized for fast similarity search. The FAISS index stores the embedding vectors in a structure that allows efficient nearest neighbor search based on L2 (Euclidean) distance. This index is stored locally (semantic_index.faiss). At query time, the same model is used to embed the input query, which is then matched against the index to retrieve the most semantically similar documents.
For our final implementation, we chose to continue exclusively with the semantic index due to its ability to capture contextual meaning more effectively.

3 Query Processing and Ranking

3.1 Evaluation

For evaluation, we tested all model configurations on a benchmark of 100 diverse queries (queries.txt) that we created in create_queries.py. For each of our test queries, we fetched the top search results from the Google Custom Search API (qrels.txt). This gave us a strong external benchmark for document relevance. Since not all URLs returned by Google were part of our own crawled dataset, we only retained those results that we had also discovered and indexed. We then used Precision@10, Recall@10 and NDCG@10 as our metrics and also tracked evaluation runtime. These metrics helped us compare the tradeoffs between accuracy and performance across all models, which ultimately guided our selection of the hybrid RRF model as the final choice.

3.2 Query Expansion

We tested multiple query expansion methods to determine whether they could improve retrieval quality. Using WordNet, we expanded queries with synonyms of the keywords. While this approach worked for simple queries, it frequently introduced unrelated concepts, leading to a slight drop in both NDCG@10 and precision.
In the pseudo-relevance feedback (PRF) method, we initially ran a BM25 query, selected the top 10 documents and extracted the most common terms to append to the original query. This method sometimes improved recall but often caused semantic drift, especially on short or vague queries. Lastly, we explored GloVe-based semantic expansion by identifying the most similar terms in the 300-dimensional GloVe space and including only those present in our corpus. We only enabled it for short queries (at most 3 terms). While this method performed better than WordNet, it still negatively affected precision in the final evaluation. As a result, we disabled all expansion techniques in our final model configuration.

[1] Douze et al. (2024)

3.3 Retrieval Models

As a baseline model, we implemented BM25 using a custom scoring function (BM25RetrievalModel) with configurable k1 and b values. The document index was constructed using a combination of normalize_and_tokenize() applied to both the main content and titles of the crawled documents. Query processing in this lexical approach included lowercasing, tokenization and optional query expansion using techniques such as WordNet, pseudo-relevance feedback (PRF) or GloVe.
In addition to lexical methods, we implemented dense retrieval to enable semantic search. For this purpose, we used the all-MiniLM-L6-v2 model from SentenceTransformers to generate embeddings. Each document was encoded into a dense vector based on its title for speed considerations. Queries were similarly embedded and matched against the FAISS index. Query processing in this approach can also be done with GloVe-based expansion.
To combine the strengths of both lexical and semantic retrieval, we developed two hybrid models. The first, Hybrid Alpha, computes a linear combination of BM25 and dense scores using a weighted sum: alpha · BM25 + (1 - alpha) · Dense, where alpha was optimized using a simple grid search. The second and final model, Hybrid RRF, uses a Reciprocal Rank Fusion to combine the rankings from the BM25 and dense models. In this case, the final score for each document is computed as $\frac{1}{(k+rank)}$ from the top 100 results from each model. We tuned the k parameter and selected k=20 based on validation performance. We ultimately chose this as our final model due to its balanced tradeoff between precision, recall and evaluation speed.
We also experimented with neural reranking, although this was not included in our final evaluation pipeline. We used cross-encoders such as stsb-TinyBERT and cross-encoder/ms-marco-MiniLM-L-6-v2 to re-encode each (query, document) pair from the top 100 hybrid results. These re-rankers provided a joint similarity score for more fine-grained relevance scoring. However, this significantly increased evaluation time (from roughly 2 seconds to over 190 seconds) and actually led to a slight drop in NDCG and precision, likely due to a domain mismatch and the zero-shot nature of our task. The neural reranking module remains implemented in the codebase as CrossEncoderReranker and can be enabled using —use_reranker.

4 User Interface

The user interface (UI) of our search engine was developed using Streamlit, a Python framework that enables rapid prototyping of web applications. Streamlit not only simplifies the creation of user interfaces, but also offers seamless deployment, allowing the application to be hosted on the web with minimal setup. One limitation of Streamlit is the restricted range of built-in UI components and customization options. However, these constraints can be addressed through custom components or by leveraging standard HTML and CSS.
As a web application, our system naturally separates into a client-server architecture. We designed the interface to be minimalistic and clean, while still providing a variety of useful features to support efficient and intuitive search behavior.
Navigation is facilitated via a search history sidebar on the left, allowing users to revisit past sessions. The central input area supports both direct text input and file uploads for query submission. At the bottom of the interface, users are always provided with contextual support, including tools for query refinement and website recommendation. By default, the engine retrieves the top 100 most relevant documents, while displaying the top 20, based on the assumption that users rarely explore beyond the first few results.
To improve efficiency, a performance mode is available that disables metadata loading. In this mode, predicted keywords, computed via TF-IDF on each webpage are hidden, reducing latency and resource usage. When enabled, these keywords offer a quick overview of each document's content. Given that users are often directed to lengthy webpages, we implemented a summarization feature that generates concise summaries using a large language model hosted by Cohere API. This helps users assess the relevance of a document at a glance.
Another notable feature is query term importance visualization. The system estimates Shapley values for each word in the query, highlighting them in varying shades of green based on their importance. These values are approximated via Monte Carlo estimation, limited to a maximum of 30 subsets to manage computational cost. We also integrated a conversational assistant, powered by Cohere API, which operates in two modes:

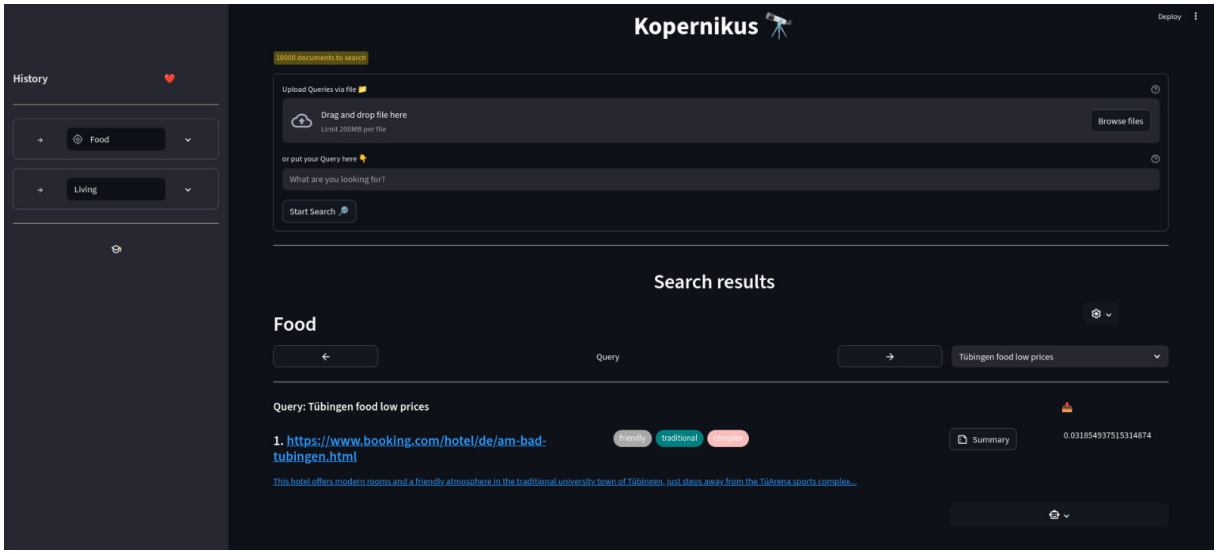

Figure 1: User Interface

- Query refinement: The assistant analyzes the query and user intent to suggest improvements that increase the relevance and reliability of retrieved results.
- Website recommendation: Based on the user's context and retrieved webpages, the assistant recommends a suitable website using a retrieval-augmented generation (RAG) approach. This leverages LLM capabilities without requiring heavy client-side computation.

These features demonstrate how the system could evolve beyond a university project into a production-ready application, given the necessary infrastructure and funding. By hosting our own compute and storage servers, we could eliminate the need for on-demand web scraping and store webpage metadata in advance. Additionally, precomputing summaries using models like DistillBART would reduce dependency on external APIs like Cohere, further improving efficiency and scalability.

References

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. (2024). The faiss library.