Programming Assignment 2

Filip Turk - Lan Biteznik - Tschimy Aliage Obenga

1. Introduction

In the second part of the assignment, we implemented a system for finding semantically similar content using vector embeddings. The main goal was to convert textual segments from web pages into dense vector representations, store them in a database, and allow efficient similarity search based on user queries. For development and testing, we initially used a partial version of the database created in the first part. This allowed us to test the performance and correctness of embedding generation and query functionality before running the full embedding process on the complete dataset.

2. Implementation Details

Data Preparation

Our work during this section was primarily focused on reviewing how the domain we scraped (https://www.fri.uni-lj.si/) structured its pages in HTML, we tried to find repetitive patterns that would make the page easier to segment and identify certain portions of the HTML page which we could automatically remove from the pool of segments (e.g. pages that always contained the same information no matter where on the domain the user was, headers, footers etc.). We found that the domain indeed had design rules it implemented which made it easier for us to segment and clean the data:

- The majority of content useful to us was always stored in the "class="block block-system" element, under the id="block-system-main" CSS identification, this HTML class always featured only content relevant to the page, it conveniently did not contain any info from the header or the footer.
- We also decided to include special handling of the "class="breadcrumbs" element, since we figured that having the breadcrumbs content of each page (that has a breadcrumbs section to begin with), formatted the same way at the very beginning of the resulting plaintext *cleaned_page* table entry would increase the accuracy of the final vector-based search function.

- We decided to separate our page content into segments by HTML elements, which we then separated by a marker that was set to default as ”<<<PARAGRAPH>>>”, but can be set by the user to be whatever they want, we simply picked an obvious marker that we assumed would never appear in the text and therefore would never run the risk of being confused for whitespace or common symbols appearing in the content itself. This type of segmentation meant that each relevant HTML parent element (as well as all of the child elements up to one layer below) became its own segment separated by the set marker in the pre-segmented plaintext output. We found that the page generally organized content by topic this way and this would therefore produce more accurate results than, say, simply segmenting the content by sentences or anything similar.

Choice of Embedding Model

We experimented with several models, prioritizing those that were free and could be run locally. We ultimately selected the **LaBSE** model from the `*sentence-transformers*` library. It was chosen for its multilingual capabilities (supporting English and Slovene), its ability to run locally, and its strong performance for semantic similarity tasks. The model is automatically downloaded and cached the first time it is used. From then on, it is reused from the local cache, avoiding repeated downloads.

Embedding Generation and Storage

We wrote a Python script that:

1. Reads text segments from the database

2. Uses **LaBSE** to generate embeddings

3. Stores the embeddings in the `embedding` column of the `*page_segment*` table

We processed the segments in batches to improve performance and reduce redundant operations.

Indexing and Similarity Search

`**pgvector**` was used to store vector data in PostgreSQL. To improve query performance, we implemented approximate nearest neighbor (ANN) indexing using the `IVFFlat` index type. This enabled fast similarity searches even with tens of thousands of segments.

3. Challenges and Solutions

One of the first challenges we encountered was how to divide the raw HTML content into meaningful segments. To solve this, we implemented a custom parser that performed structural segmentation based on HTML elements, preserving semantic cohesion across segments.

Another important consideration was the potential cost associated with generating vector embeddings. Rather than relying on paid APIs such as **OpenAI's** embedding models, we opted for the open-source **LaBSE** model from the sentence-transformers library, which we ran locally to avoid incurring usage fees.

To store and query the resulting vector embeddings efficiently, we integrated the **pgvector** extension in PostgreSQL and configured it to use the **IVFFlat** index. This enabled us to perform approximate nearest neighbor (ANN) searches with much greater speed.

Finally, to measure similarity between vectors and provide relevant search results, we adopted cosine similarity as our comparison metric, which is well-suited for high-dimensional embeddings like those generated by LaBSE.

4. Conclusion

We successfully completed the assignment with the following outcomes:

- All segments in the `*IEPSdb_partial*` database now have vector embeddings.

- We implemented a working similarity search system using local infrastructure.