Assignment 3: Semantic Search and Retrieval with Sentence-BERT

Stig Hellemans, Can Çetinsoy

Background

Introduction

In previous assignments, we explored techniques for syntactic search, which often fail to capture synonymy and other semantic nuances of text. To address this limitation, we implemented a semantic search and retrieval system using three frameworks: (1) a default **vector database** that performs an exhaustive search across all documents and retrieves those most similar to the query, (2) a **cluster database** that groups documents into clusters and compares queries to cluster centroids first, then searches within the top-matching clusters, and (3) a **hierarchical navigable small worlds (HNSW) database**, which uses navigable small-world properties for approximate and efficient search. The latter two methods provide faster, approximate search compared to the exhaustive approach of the first framework. We implemented each database and compared their performance with a third-party library, Facebook AI Similarity Search (FAISS) [1].

Vector Database

A vector database is a system designed for efficient storage, retrieval, and management of high-dimensional vector representations of data. In this implementation, the vector database leverages a text embedding model to encode text documents into dense vector embeddings, capturing semantic information such as synonymy and contextual meaning. These vectors enable similarity-based searches, allowing users to retrieve the most relevant documents for a given query by comparing their vector representations using cosine similarity. Since all embedding vectors are normalized to have a magnitude of one, cosine similarity simplifies to calculating the dot product between the document and query vectors. This approach also allows for the use of efficient linear algebra libraries. The system supports efficient batch encoding, enabling the computation of document vectors for both the small and large dataset. The large dataset comprises approximately 500,000 documents. Additionally, the translation of internal indices back to document IDs provides a general-use implementation, making it adaptable for integration with other databases. This approach ensures scalability and semantic-rich retrieval capabilities across varying dataset sizes. All other implementations are based on this default implementation.

Text Embedding Model

A core component of a vector database is the text embedding model, which transforms the entire text of a document or query into a dense vector embedding. To produce meaningful embeddings, the model must capture the semantic similarity between texts. While models like BERT are pre-trained using masked language modeling (MLM) objectives, this primarily yields token-level embeddings. Aggregating these token embeddings—such as by averaging— or using the [CLS] token can result in suboptimal representations, as it may not accurately reflect the varying contributions of individual tokens to the overall semantics. For instance, averaging the embeddings of the tokens [‘the’, ‘information’, ‘retrieval’] might overemphasize the significance of ‘the’ in the global text embedding.

To address this, a pre-trained BERT model can be fine-tuned to discern between semantically similar and dissimilar text pairs. This fine-tuning process often utilizes datasets that inherently encode similarity information, such as paraphrase datasets, manually scored semantic similarity datasets, or translations of the same text in different languages [2, 3]. Contrastive learning techniques, such as those utilizing triplet loss, are frequently employed during fine-tuning. These methods encourage embeddings of semantically similar texts to be closer together, while pushing embeddings of dissimilar texts further apart.

After testing the performance of several different sentence transformers, we chose the ‘all-MiniLM-L6-v2’ (see Evaluation). The foundation of this model is MiniLM-L6, a distilled 6-layer version of the BERT architecture, pre-trained using a masked language modeling (MLM) objective [4]. During its creation, MiniLM-L6 was trained to replicate the behavior of a larger and more complex teacher model through a knowledge distillation process. This approach reduced the model’s size and computational demands while preserving much of the original BERT model’s representational power. Next the model was fine-tuned on a diverse dataset comprising over 1 billion sentence pairs. These pairs were sourced from multiple datasets, including Reddit comments, S2ORC citation pairs, WikiAnswers duplicate questions, and more [5]. This extensive and varied training data enables the model to capture a wide range of semantic relationships.

Cluster Database

A clustering technique such as k-means will group document texts into k groups based on euclidean distance and thus similarity. Euclidean distance is interrelated with cosine similarity since all embedding vectors are normalized (Fig. 1). Although this additional step increases database construction, since the clustering takes computation, it reduces search time. This is done by first comparing the query vectors with the centroids (cluster means) and only subsequently searching through the top number of clusters. Since you don’t compare the query vector with each document embedding vector, it is an approximation and optimality of the retrieved documents is not ensured. You could choose to increase the number of top clusters but this in turn increases computational load and thus search time. In the evaluation section, we’ll go further in finding this balance.

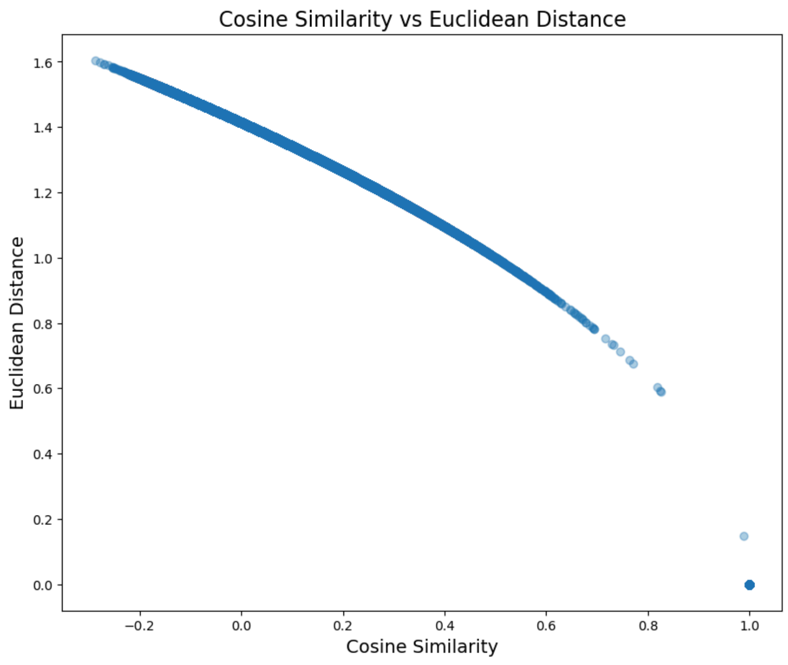

Figure 1: The figure illustrates an inverse relationship between cosine similarity and Euclidean distance of normalized document vectors. As cosine similarity increases, indicating higher angular similarity, Euclidean distance decreases, showing closer proximity in Euclidean space. The experiment was done using the small document dataset.

Hierarchical Navigable Small Worlds (HNSW) Database

Hierarchical navigable small worlds (HNSW) databases leverage the navigable small world property of well-connected networks. For example, in the Facebook user network, it is known that, on average, only approximately 3.5 intermediate friend hops are needed to connect any user to another [6]. This property is applied in the retrieval context to navigate toward document embedding vectors that are more similar to a query vector. To implement this efficiently, a hierarchical, layered approach is used. It begins by identifying the most similar vectors within a sparse subset of the document vectors (Fig. 2). These retrieved document vectors then serve as starting points for traversal in the next layer, which contains a denser set of nodes (document vectors). This hierarchical traversal allows the network to be navigated effectively while avoiding becoming stuck in local minima. The issue of local minima is further addressed by utilizing multiple starting probes during traversal [7]. Although this approach provides an approximate search and does not guarantee optimality, increasing the number of nearest neighbors per node enhances the network's navigability and improves retrieval performance.

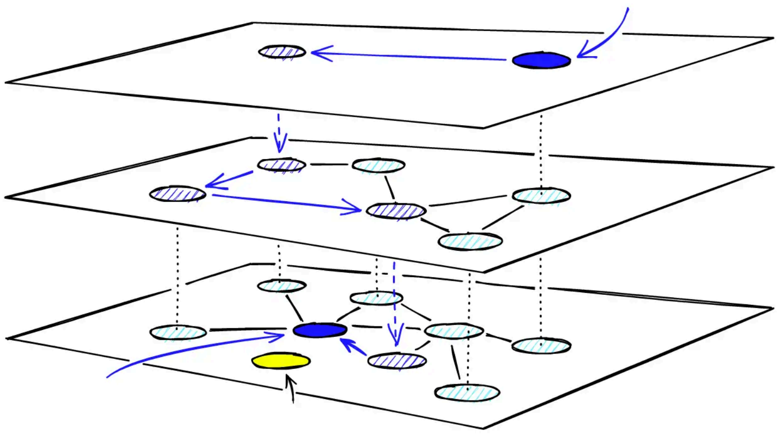

Figure 2: This figure illustrates the abstract representation of hierarchical navigable small worlds (HNSW). A sparse upper layer enables rapid traversal of the entire network to locate hubs of document vectors similar to the query. Moving down the layers allows for efficiently identifying document vectors that are increasingly similar to the query.

Implementation Details

Vector database

Our vector database implementation precomputes all document embedding vectors and stores them in a matrix, where each row represents a document vector. To map internal row indices to the actual document IDs, we maintain a mapping mechanism. Document IDs, which can be integers or strings, are specified in the document dictionary alongside their file paths when calling the *encode_docs* method. The method reads the text content from the provided file paths and transforms it into embedding vectors using batched processing for efficiency.

As a base class, this implementation supports additional training or refinement of the database via the *train* method. This includes precomputing data for clustering, Hierarchical Navigable Small Worlds (HNSW) graphs, or other approximation methods. Additionally, it offers functionality to save and load precomputed databases for reuse.
During inference, the primary operation is the *search* function in the *search* module. This function processes document vectors in batches to calculate and maintain the top k most similar documents and their corresponding cosine similarity scores. To optimize performance, the system aggregates

results incrementally when the stored data for top documents exceeds a predefined threshold. This approach avoids frequent sorting by using partitioning, but the final top k documents are sorted in the final step to ensure correct ordering of the final k retrieved documents.

Cluster database

The cluster database extends the functionality of the base *VectorDatabase* class by incorporating k-means clustering. The number of clusters is treated as a hyperparameter, and the database maintains an additional *doc_ids_in_cluster* list, where each index corresponds to a specific cluster. This enables quick access to document vectors within a given cluster.

During inference, in addition to the standard *search* function, the system restricts the search to only the top c clusters, determined by comparing query vectors to cluster centroids using cosine similarity. Cluster centroids act as the average representation of their respective clusters (Fig. 3). To facilitate parallel searching across multiple query vectors and clusters, the process iterates through all clusters. For each rank *i* in the *top_c* clusters, query vectors assigned to that cluster are collected and processed. For a given cluster, the system retrieves a fixed number of documents (*k*) for each query vector. The retrieved results, consisting of documents and their similarity scores, are stored in a result matrix of shape (*top_c* * *k*, *query_number*). Finally, the result matrix is reduced by aggregating and selecting only the top k documents based on the highest similarity scores, ensuring efficient and accurate retrieval. The *top_c* parameter serves as a second hyperparameter in the implementation.

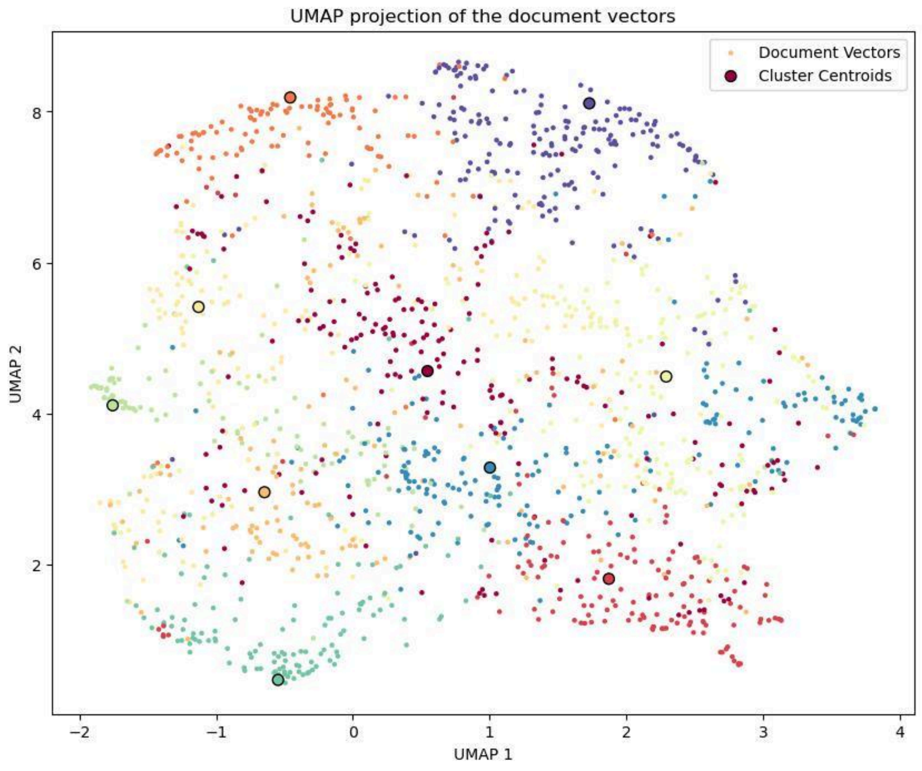

Figure 3: UMAP projection of document vectors, color-coded by cluster, with cluster centroids highlighted. This visualization illustrates the distribution of clustered embeddings and their centroids, which serve as average representations for efficient query-based retrieval.

Hierarchical Navigable Small Worlds (HNSW) Database

It is important to note that our original implementation failed to produce reasonable results. For this reason, we decided to omit those results and instead focus on a FAISS-based implementation, paying particular attention to hyperparameter tuning of the FAISS HNSW index. We have retained our own code primarily as a conceptual illustration, to help explain the underlying algorithm. Our code assigns each document vector to various levels at random (see *_assign_levels*), following an approach similar to FAISS [7]. Next, *_make_edges* connects each document vector to its nearest neighbors. During inference, the algorithm randomly selects several “probes” in the topmost layer. Each probe represents a document vector that serves as a starting point for the search. As the search proceeds, *_traverse_layer* uses similarity-based connections to allow probes to “hop” from one document vector to another that is more similar to the query vector. One challenge we encountered was that multiple probes often converged to the same endpoint. This issue made some of the probes redundant. We define an endpoint as a node where the probe cannot continue to any neighbor more similar than its current node. Additionally, we observed cyclic behavior, where a probe would repeatedly cycle through the same sequence of nodes. Once a probe reaches an endpoint at a given layer, it descends to the next layer and resumes traversal. Finally, at the last layer, *_get_top_k* collects the neighbors of the probes to expand the pool until at least k unique candidates are obtained. The similarities of these candidate vectors are then computed against the query vector to produce the top k results. This final stage was not derived from the FAISS algorithm; rather, it was introduced to address the issue of probe redundancy.

Facebook AI Similarity Search (FAISS) Implementations

We implemented the IndexFlat2D, IndexIVFFlat, and IndexHNSWFlat databases, corresponding to our default, clustering, and HNSW implementations, respectively. To ensure seamless integration with our existing codebase, we wrapped FAISS in a class that inherits from the *VectorDatabase* base class. Performance and hyperparameter tuning experiments showed similar trends to our self-implemented approaches, except for significantly reduced query times. These faster query times, expected due to FAISS's optimized C++ implementation, outperformed all other methods. This confirms that our code does not introduce bugs that substantially alter performance, apart from the case of HNSW. To illustrate these results, we provided plots analogous to Fig. 5 for FAISS in the supplementary materials (Fig. S1).

Our implementation can be found at this Github link.

Analysis

Sentence transformers

We evaluated the performance of several sentence transformers to make an informed decision on the most suitable text embedding model for the remainder of our assignment (Fig. 4). The compact *all-MiniLM-L6-v2* model emerged as one of the top performers, second only to the significantly larger *all-mpnet-base-v2* model. In information retrieval, computational efficiency is as crucial as retrieval performance. On our hardware, *all-mpnet-base-v2* required 8 times longer database construction times and 5.5 times longer inference times compared to *all-MiniLM-L6-v2*. Given this trade-off, we chose to proceed with the *all-MiniLM-L6-v2* model for its balance of performance and efficiency.

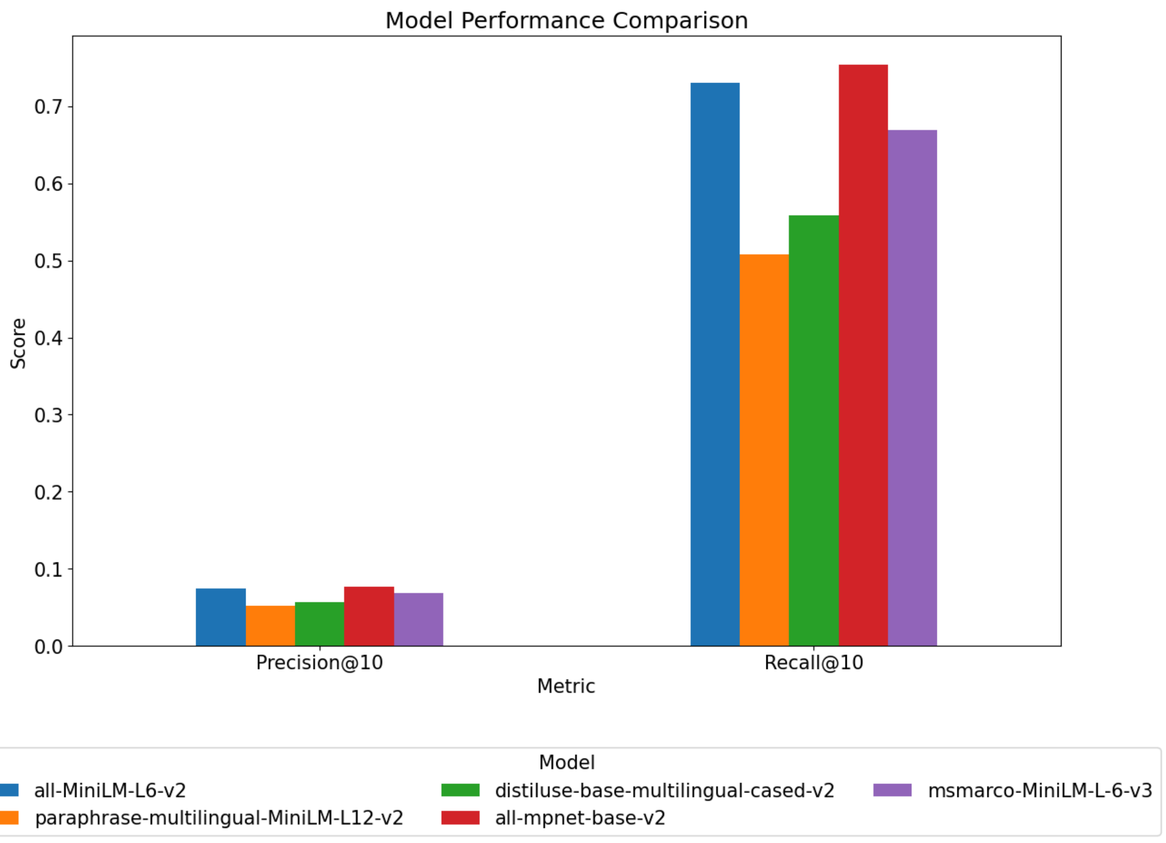

Figure 4: Comparison of Precision@10 and Recall@10 for various sentence embedding models, highlighting differences in retrieval performance across models.

Hyperparameters

Cluster Database

Two key hyperparameters influence the performance of the clustered database: the total number of clusters and the *top_c* ranked clusters. The total number of clusters determines how the documents are divided and directly impacts clustering time, as higher cluster counts lead to increased computational load (Fig. 5C). During inference, the parameter *top_c* determines the approximation level of the search, represented in Fig. 5A by the fraction of clusters searched, normalized by the total number of clusters. With more clusters, searching a smaller fraction achieves near-optimal Mean Average F1 Score at *k*=10 (MAF1@10), demonstrating that effective searches can be performed with a low top_c without significant performance degradation.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

However, this efficiency comes at a cost. Increasing the number of clusters also increases the computational overhead, since a larger number of clusters must be searched per batch of queries (Fig. 5B). Finding the right balance between these hyperparameters is thus essential. For our subsequent analyses, we determined that using 100 clusters and setting *top_c* = 12 provided an optimal configuration, as indicated by the red-circled points in the figures. By contrast, for the FAISS implementation, we adjusted our parameters to 200 clusters and *top_c* = 4 to achieve faster query times than the default settings.

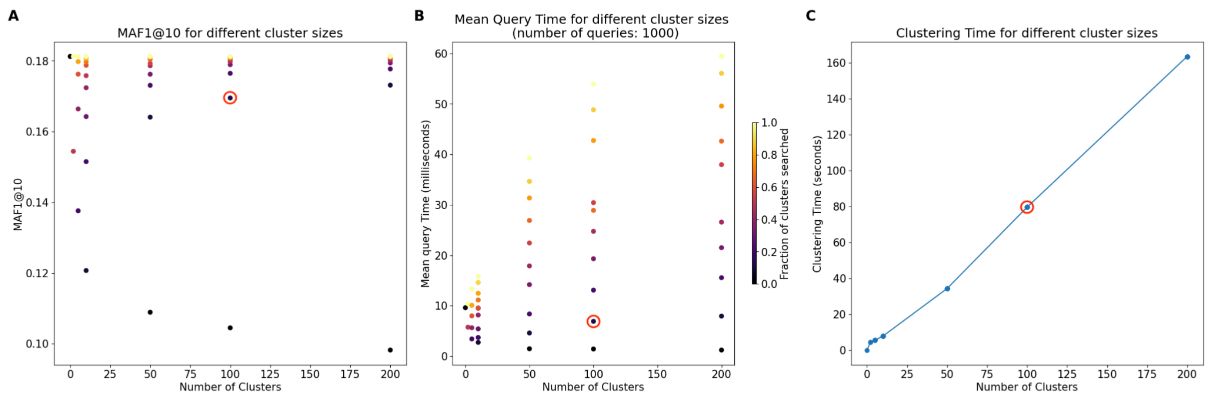

Figure 5: Performance metrics for varying cluster sizes. The black dot on the far left represents the default database without clustering. (A) MAF1@10 scores for different numbers of clusters, highlighting search accuracy across cluster sizes. (B) Mean query time (ms) for 1,000 queries, colored by the fraction of clusters searched, demonstrating the trade-off between search efficiency and fraction coverage. (C) Clustering time (seconds) as a function of cluster size, showing the computational cost of clustering increases linearly with the number of clusters. Red circles mark the selected optimal cluster size balancing performance and cost of 100 clusters and *top_c*=12.

HNSW (FAISS) Database

The HNSW index in FAISS has three key hyperparameters: *efSearch*, *efConstruction*, and *M*. *efSearch* controls the number of candidate neighbors explored during querying, balancing recall and speed, with higher values improving recall at the cost of slower queries. *efConstruction* determines the number of candidates considered during index building, where higher values enhance index quality but increase construction time and memory usage. *M* sets the maximum number of connections per node, affecting the graph's connectivity and recall; larger values improve accuracy but require more memory and longer build times. After tuning the hyperparameters, we selected the following values: *efSearch* = 100, *efConstruction* = 100, and *M* = 16 (Fig. S2).

Comparison

This table presents key evaluation metrics used to compare various implementations across different assignments for the large dataset. The main performance indicators are MAP@K and MAR@K, while additional metrics include index size, index construction time, vocabulary size, and query times.

	VSM (Self-implemented)	Lucene	Dense retrieval (Self-implemented)		Dense retrieval (FAISS)		
	VSM	Default	Vector Database	Cluster Database	Vector Database	Cluster Database	HNSW Database
Hyperparams				n_clusters=100 top_c=12		n_cluster=200 top_c=4	efSearch=100 efConstruction=100 M=16
MAP@1		90.0%	49.2%	34.4%	49.2%	42.8%	47.3%
MAR@1		6.1%	3.3%	2.3%	3.3%	2.9%	32.5%
MAP@3	25.6%	82.8%	36.9%	29.9%	36.9%	31.1%	35.5%
MAR@3	5.1%	16.8%	7.3%	5.9%	7.3%	6.1%	7.0%
MAP@5		76.3%	30.8%	26.8%	30.8%	25.4%	29.7%

MAR@5		25.5%	10.1%	8.8%	10.1%	8.3%	9.7%
MAP@10	21.3%	63.0%	23.0%	21.7%	23.0%	18.7%	22.1%
MAR@10	14.0%	41.6%	15.0%	14.1%	15.0%	12.1%	14.4%
Database size (original dataset: 3.51 GB)	1.15 GB	1.3 GB	824 MB	834 MB	824 MB	824 MB	791 MB
Index construction time	3.5 h	8 min 37 sec	1.5 h	1.5h	1.5 h	1.5 h	1.5 h
Query time (ms/query)	44	9.3	9.2	7.4	1.2	0.89	0.71
Vocab size (# terms)	7 180 128	21 411 542	/	/	/	/	/

For our dataset, syntactic retrieval with Lucene outperforms dense semantic retrieval methods, contrary to expectations from benchmarks like MS MARCO [8]. We suspect that crucial information is lost during document encoding, as our model only encodes the first 256 tokens, and 86.58% of documents exceed this limit (Fig. S3). To address this, we experimented with chunking on the small dataset. Chunking splits documents into shorter, overlapping segments, allowing more information to be captured by multiple embedding vectors. We adjusted the search process to ensure unique document rankings and selected a chunk length of 128 and a stride of 16 (Fig. S4). Although this approach significantly improved performance, it still lags behind Lucene (Fig. S5).

FAISS offers significantly faster query times, although generating embedding vectors takes considerably longer than constructing a Lucene index. This is likely due to running on an M1 MacBook Air with only 8GB of RAM. While dense vector databases tend to be smaller in size without document chunking, this may change when document chunking is applied.

References

[1] Douze, M. *et al.* The Faiss library. (2024).
[2] Reimers, N. & Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference* 3982–3992 (2019) doi:10.18653/v1/d19-1410.
[3] Reimers, N. & Gurevych, I. Making Monolingual Sentence Embeddings Multilingual using Knowledge Distillation. *EMNLP 2020 - 2020 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference* 4512–4525 (2020) doi:10.18653/v1/2020.emnlp-main.365.
[4] Wang, W., Bao, H., Huang, S., Dong, L. & Wei, F. MiniLMv2: Multi-Head Self-Attention Relation Distillation for Compressing Pretrained Transformers. *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021* 2140–2151 (2020) doi:10.18653/v1/2021.findings-acl.188.
[5] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[6] https://research.facebook.com/blog/2016/2/three-and-a-half-degrees-of-separation/
[7] https://www.pinecone.io/learn/series/faiss/hnsw/
[8] https://www.sbert.net/docs/pretrained-models/msmarco-v3.html

Supplementary

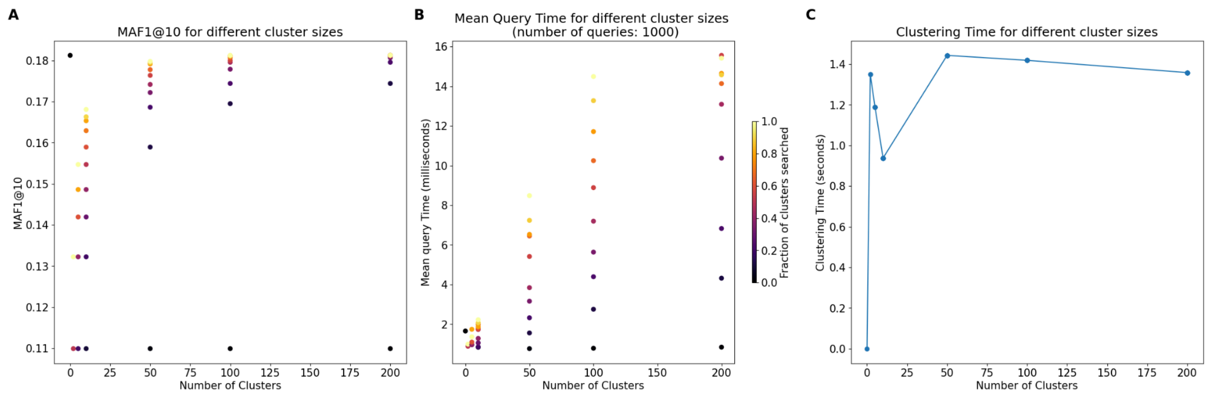

Figure S1: These experiments were conducted using the FAISS library. The black dot on the far left represents the default database without clustering. The results closely align with those of our self-implemented Cluster Database, with two key differences: (1) the clustering (construction) time in FAISS is more constant, whereas our implementation exhibits a more linear trend; and (2) including all clusters during inference does not fully converge to the exhaustive case, as seen in Fig. 5, likely due to the use of additional heuristics and advanced techniques in FAISS.

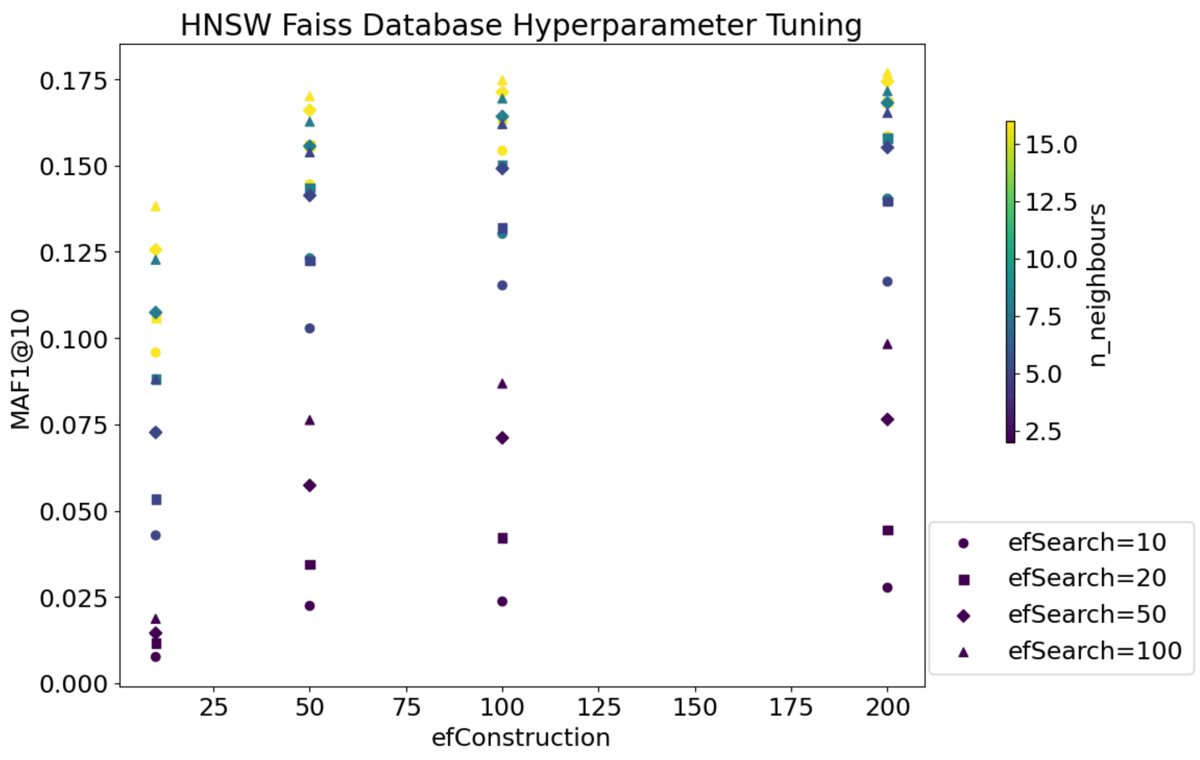

Figure S2: Hyperparameter tuning of the HNSW FAISS database. The final chosen hyperparameters are *efSearch* = 100, *efConstruction* = 100, and *M* = 16. The Mean Average F1 score at k=10 is used as the evaluation metric.

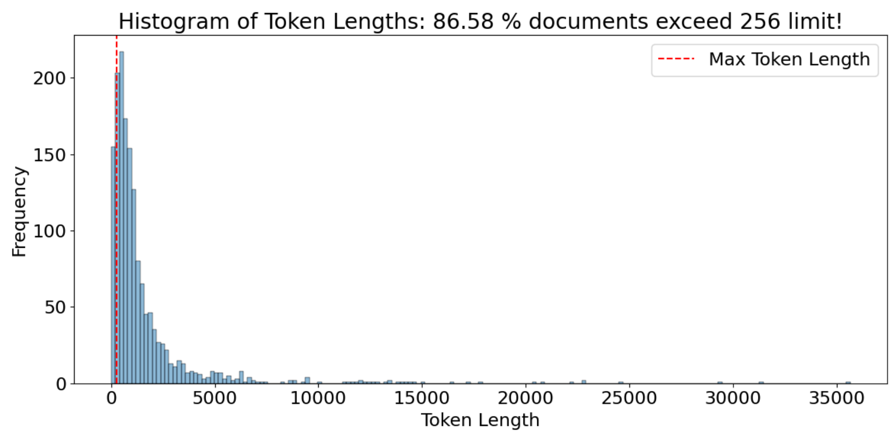

Figure S3: This histogram illustrates the distribution of token lengths across documents in the small dataset. The maximum token length of 256 corresponds to the limit imposed by our model (*all-MiniLM-L6-v2*).

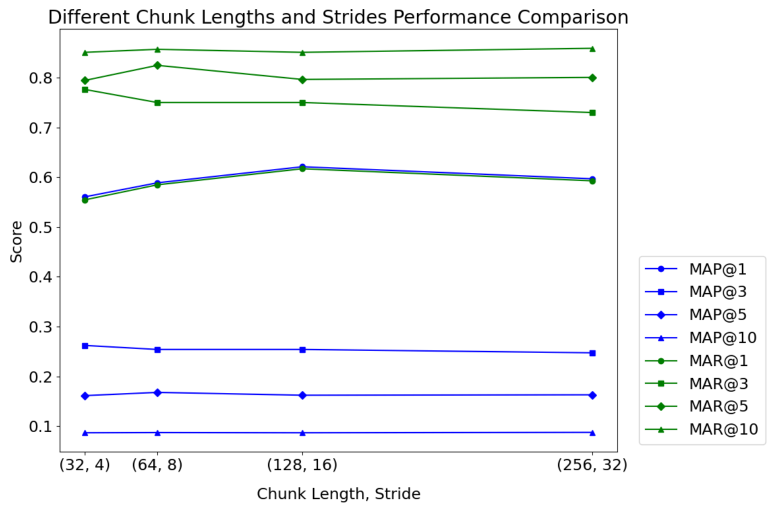

Figure S4: The figure presents the impact of various chunking and stride configurations on the small dataset, illustrating their corresponding mean average precision (MAP) and mean average recall (MAR) for k = 1, 3, 5, and 10.

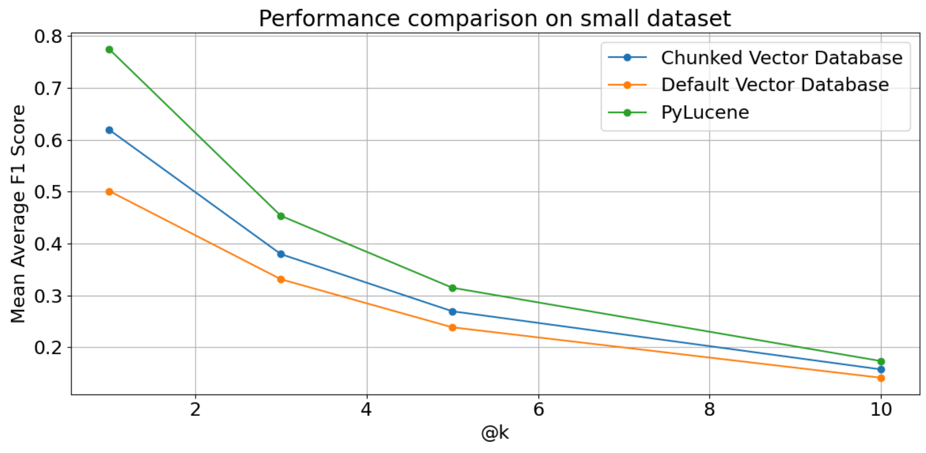

Figure S5: The figure presents the mean average F1 score (MAF1) for k=1, 3, 5, and 10 on the small dataset, comparing three configurations: the default vector database, a chunked vector database (chunk length = 128, stride = 16), and PyLucene.