Towards Lightweight and LLM-Free Semantic Search for mathlib4

Isaac (Rucheng) Li
University of Pittsburgh

Abstract

Finding the right theorem in `mathlib4` (a library for the Lean 4 proof assistant) can be surprisingly hard: everyday mathematical phrases (e.g., "a product is zero if and only if one factor is zero" or "a * b = 0 iff a = 0 or b = 0") rarely match the precise formal theorem names (e.g., `mul_eq_zero`). Recent semantic search engines either rely on large embedding models, raising latency and deployment costs, or adopt lightweight embedding models that can struggle with formula-heavy queries. We present a high-accuracy, fast, lightweight semantic search engine that can run entirely locally on a laptop-class CPU.

A public demo and reproducible notebooks are available at `https://github.com/IsaacLi74/Lightweight-and-LLM-Free-Semantic-Search-for-mathlib4`.

1 Introduction

Mathematical libraries in proof assistants have grown so large that keyword-based lookup often fails even for experienced users.

Before large language models, several content-based systems already supported searching formal theorems (e.g., Whelp [1]). These helped, but they often struggled with informal queries. This fact is a hurdle for newcomers who want to use proof assistants to learn mathematics.

More recent semantic search engines use embedding models [2] to embed a detailed informal paraphrase for each formal theorem as a search anchor, and at query time embed the user's text into the same vector space, ranking candidates by cosine similarity. For `mathlib4` [3], **LeanSearch** [4] builds a paired formal–informal corpus and optionally performs LLM-based query augmentation that rewrites the user's input into a more detailed description before dense retrieval. Although it performs well on reported benchmarks, it relies on the `E5-Mistral-7B` [5, 6] embedding model, which is difficult to deploy on resource-constrained devices (e.g., laptops). Another advanced engine, **LeanExplore** [7], adopts a similar embedding-based approach but uses a lightweight embedding model `bge-base-en-v1.5` [8] and a hybrid ranker over multiple text sources (primary declaration's name, docstring, informal descriptions...), supporting both local and hosted API modes.

While **LeanExplore** is lightweight, this design has trade-offs: in our tests, it struggles to align formula-heavy queries with the correct formal theorem. An obvious workaround is to invoke online LLM-based query augmentation to rewrite formula-heavy inputs into detailed natural-language queries. However, this adds query-time latency and cost, undermining the goal of lightweight local deployment. **LeanSearch** handles formula-heavy queries better but depends on a larger model (`E5-Mistral-7B`) and exhibits some variable-naming bias (e.g., searching `a + -b = a - b` returns correct results, whereas `m + -i = m - i` does not). Motivated by these limitations, we introduce a straightforward, laptop-runnable `mathlib4` search engine that supports formula-based queries, reduces naming bias, and achieves high retrieval accuracy without any need for online LLM calls at query time.

2 Method

We first describe how a user query is processed at runtime, and then explain how the search engine is constructed step by step.

At query time, we begin by normalizing the input query. Mathematical symbols are standardized (e.g., replacing "<=" with "≤"), and spaces are inserted around symbols so that tokens are clearly separated. The normalized query is then embedded using `Qwen3-Embedding-0.6B` [9] with a fine-tuned LoRA [10] adapter to produce a vector representation. Using this vector, we retrieve the 50 nearest pre-embedded search anchors by cosine similarity from a prebuilt `FAISS HNSW` [11, 12] (Hierarchical Navigable Small World graphs) index. Finally, we deduplicate by formal theorem names and return the top-10 results with their names and statements.

2.1 Data preparation

From a `mathlib4` snapshot (Lean 4.20.1), we extract **179,194** theorems with `LeanDojo` [13]. For each theorem, we use `DeepSeek-Chat` [14] API to produce three detailed queries — **q1** natural-language, **q2** formula-style, **q3** mixed — while wrapping every variable with & (e.g., `&a&`). We then normalize the statement, insert spaces around every variable and mathematical symbol so they tokenize separately, and mask variables left-to-right as $\S_1, \S_2, \ldots$ (e.g., `&a&` $\mapsto \S_1$).

2.2 Training set synthesis

To embed queries of the same theorem closer together while separating those of different theorems, we fine-tune the lightweight base embedding model on query pairs with the loss function `MultipleNegativesRankingLoss` [15] (MNRL).

Index the theorems by $i = 1, \ldots, N$ with $N = 179{,}194$. For each theorem i with detailed queries $(q_1^{(i)}, q_2^{(i)}, q_3^{(i)})$, we form three positive pairs $(q_1^{(i)}, q_2^{(i)})$, $(q_2^{(i)}, q_3^{(i)})$, and $(q_3^{(i)}, q_1^{(i)})$. Concatenating these across all theorems in a fixed order—first all (q_1, q_2), then all (q_2, q_3), and finally all (q_3, q_1)—yields $3N = 537{,}582$ positive pairs. For a batch of B pairs $\{(a_i, p_i)\}_{i=1}^{B}$ (where the first element is treated as the anchor and the second as its positive), the MNRL objective with cosine similarity $s(\cdot, \cdot)$ and scaling factor α is

$$\mathcal{L} = \frac{1}{B} \sum_{i=1}^{B} \left[-\log \frac{\exp\big(\alpha\, s(a_i, p_i)\big)}{\sum_{j=1}^{B} \exp\big(\alpha\, s(a_i, p_j)\big)} \right].$$

Because LLMs have finite context, semantically similar theorems may yield identical queries. Under MNRL, this creates false negatives if such duplicates appear in the same batch. To avoid this, we group duplicate query theorems and enforce a one-per-batch rule, ensuring that no two members of the same group occur together.

To further increase training hardness while avoiding noise, we construct three theorem sets by reordering the full theorem collection according to different batching strategies. These sets are later used to form the training sets as described above. The batching strategies are detailed below:

- **Low-clustered**: near-uniform random ordering of the full theorem set.
- **Medium-clustered**: formal theorem names sorted so theorems from the same area co-occur, increasing within-batch similarity.
- **High-clustered**: cluster-based batches via k-means on base model embeddings of formal theorem statements, yielding batches of closely related patterns.

2.3 Model adaptation and base index construction

We adopt `Qwen3-Embedding-0.6B` as the base embedding model and fine-tune it with a lightweight `LoRA` adapter over four epochs: one **Low-clustered**, two **Medium-clustered**, and one **High-clustered** as above. We then use the adapted embedding model to embed all detailed queries, and index them with `FAISS HNSW` to form the vector–to–theorem mapping of the base search engine. We choose `HNSW` because it is a state-of-the-art approximate nearest neighbor (ANN) method that significantly accelerates vector search while maintaining high recall. Similar ANN indexing is adopted in `LeanSearch` and `LeanExplore`.

2.4 Vague query harvesting and index finalization

For each theorem, we use `DeepSeek-Chat` to generate five vague queries (under the same normalization as detailed queries). We run each query against the base search engine. If the ground-truth theorem is missing from the top-10 results, we log the tuple (query, target formal statement, top-10 results) for adjudication. We then prompt `DeepSeek-Chat` to adjudicate these candidates, discarding queries that are unintelligible or too underspecified to retrieve the target theorem. Also, for highly generic patterns that match many declarations (e.g., $\S_1 + \S_2 = \S_2 + \S_1$ for many `add_comm` theorems), if the top-10 results are reasonable, we drop those rarely used variants (e.g., `UInt64.add_comm`). The accepted vague queries are then merged with the original detailed queries to form the final set of search anchors. We re-embed all search anchors and build the final `FAISS HNSW` index, yielding the final search engine.

3 Results

We evaluate on **2,400** queries derived from **300** formal theorems. Since LeanSearch is built on Lean 4.16.0, LeanExplore on Lean 4.19.0, and our model on Lean 4.20.1, we select a set of commonly used and important mathematical theorems as the evaluation anchors to ensure that all search engines can retrieve them. To preclude any leakage from training anchors, the 2,400 evaluation queries were independently regenerated from the original formal theorems using `DeepSeek-Chat`, rather than reusing any search anchors constructed during indexing. The queries are divided into four groups (detailed_nl, detailed_formula, short_nl, short_formula), each with two paraphrase variants (*pos 0* and *pos 1*). For formula-style queries, paraphrases mainly consist of renaming variables to different alphabetic symbols (e.g., $a \mapsto m$) without altering the mathematical content. The evaluation metric is the percentage of queries for which the correct formal theorem appears among the top-10 results (additionally, we also count as correct cases where the namespace is dropped in results; e.g., `dvd_mul_left` is a correct answer for `Nat.dvd_mul_left`).

Group (pos)	Ours	LeanExplore	LeanSearch	LeanSearch (aug.)
detailed_nl (0)	98.00%	80.67%	79.67%	—
detailed_nl (1)	97.67%	71.33%	77.33%	—
detailed_formula (0)	95.33%	69.67%	78.67%	—
detailed_formula (1)	94.33%	30.67%	68.67%	—
short_nl (0)	94.33%	71.00%	78.00%	—
short_nl (1)	86.67%	58.33%	71.00%	—
short_formula (0)	89.67%	64.33%	73.00%	—
short_formula (1)	84.67%	26.00%	55.33%	—
Overall	**92.58%**	**59.00%**	**72.71%**	—

Note: "aug." refers to LeanSearch with online LLM-based query augmentation enabled. Results for this setting have not yet been evaluated.

Efficiency. All timings are measured in Google Colab using the demo. End-to-end per-query latency is typically **0.5–1.0 s**, depending on input length, with average RAM around **7.2 GB**. *Session CPU: Intel(R) Xeon(R) CPU @ 2.20GHz; vCPUs: 2; RAM: 12.7 GB.*

4 Limitations

Although our results demonstrate strong performance, the current system still has several limitations.

Incomplete theorem extraction. Our theorem set (179,194 theorems from Lean 4.20.1) is incomplete, smaller than the LeanSearch theorem set by about 9,000 theorems. This is easy to address; we deferred it because our initial goal was to validate the overall design. The current set suffices for that purpose.

Over-separation in training. Our fine-tuning strategy draws queries of the same theorem closer together in the embedding space while pushing those of different theorems apart. From results, this strategy works out well. However, this contrastive setup neglects connections across distinct theorems. For example, `Nat.prime_def_lt` (if a number smaller than a prime p divides p, it must be 1) and `Nat.prime_def_lt'` (there are no divisors of p in the interval $(1, p)$) both express almost the same fact about primes, yet one is treated as a negative in the **Medium-clustered** dataset. After multiple rounds of clustered training, this may lead to overfitting. An obvious solution would be to construct an independent training set disjoint from the query set. However, we did not pursue this option due to funding constraints.

Dependence on anchor richness. The retrieval quality of our engine is closely tied to the richness of its search anchors. Abstract inputs such as "axioms of natural numbers" fail to retrieve the Peano axioms, because the generated anchors do not encode such conceptual-to-formal links. Larger embedding models may better capture these higher-level abstractions, but at the expense of higher computational cost. Feedback from the Lean Zulip community reported similar issues. For instance, the query "subgroup transitivity" failed to return `le_trans`. This reflects a broader limitation of embedding-based search engines: during search anchor construction, the LLM does not fully capture that general theorems such as `le_trans` support a wide range of applications. While improved prompting or the use of more advanced online LLMs (e.g., ChatGPT or Gemini) could potentially solve this problem, our choice of `DeepSeek-Chat` was primarily motivated by funding constraints.

Potential evaluation bias. Although we did not reuse search anchors when constructing the test set, generating both anchors and queries with the same LLM (`DeepSeek-Chat`) may introduce stylistic/distributional bias and inflate performance. This problem can be solved by evaluating on test sets generated by other LLMs and by human-written queries.

5 Future Work

In future work, we plan to progressively address the limitations discussed above. Although our current prototype is built for Lean, the proposed design is lightweight and generic, and could be adapted to other proof assistants with minimal changes. The system is also easy to maintain: when libraries such as `mathlib4` update, one only needs to add the updated formal theorems and related new search anchors. Furthermore, when a user query fails to retrieve relevant results, it can be converted into vague queries and periodically incorporated through the same *vague query harvesting and index finalization* procedure described in Section 2.4. In this way, the search engine can continuously improve over time while preserving its low-cost and locally deployable nature.

References

[1] A. Asperti, C. S. Coen, E. Tassi, and S. Zacchiroli. A Content Based Mathematical Search Engine: Whelp. In *Types for Proofs and Programs (TYPES 2004)*, LNCS 3839, pp. 17–32. Springer, 2006. https://doi.org/10.1007/11617990_2.

[2] N. Reimers and I. Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of EMNLP–IJCNLP*, pp. 3982–3992, 2019. https://aclanthology.org/D19-1410.

[3] The mathlib Community. The Lean Mathematical Library. In *CPP*, 2020. https://doi.org/10.1145/3372885.3373824.

[4] G. Gao, H. Ju, J. Jiang, Z. Qin, and B. Dong. A Semantic Search Engine for mathlib4. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 8001–8013. https://aclanthology.org/2024.findings-emnlp.470/.

[5] L. Wang, N. Yang, X. Huang, B. Jiao, L. Yang, D. Jiang, R. Majumder, and F. Wei. Text Embeddings by Weakly-Supervised Contrastive Pre-training. arXiv:2212.03533, 2022. https://arxiv.org/abs/2212.03533.

[6] L. Wang, N. Yang, X. Huang, L. Yang, R. Majumder, and F. Wei. Improving Text Embeddings with Large Language Models. arXiv:2401.00368, 2024. https://arxiv.org/abs/2401.00368.

[7] J. Asher. LeanExplore: A search engine for Lean 4 declarations. arXiv:2506.11085, 2025. https://arxiv.org/abs/2506.11085.

[8] S. Xiao, Z. Liu, P. Zhang, and N. Muennighoff. C-Pack: Packaged Resources To Advance General Chinese Embedding. arXiv:2309.07597, 2023. https://arxiv.org/abs/2309.07597.

[9] Y. Zhang, M. Li, D. Long, X. Zhang, H. Lin, B. Yang, P. Xie, A. Yang, D. Liu, J. Lin, F. Huang, and J. Zhou. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. arXiv:2506.05176, 2025. https://arxiv.org/abs/2506.05176.

[10] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685, 2021. https://arxiv.org/abs/2106.09685.

[11] M. Douze, J. Johnson, and H. Jégou. The FAISS Library. arXiv:2401.08281, 2024. https://arxiv.org/abs/2401.08281.

[12] Y. A. Malkov and D. A. Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020. https://doi.org/10.1109/TPAMI.2018.2889473.

[13] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *NeurIPS 2023 (Datasets and Benchmarks)*. https://papers.neurips.cc/paper_files/paper/2023/file/4441469427094f8873d0fecb0c4e1cee-Paper-Datasets_and_Benchmarks.pdf.

[14] DeepSeek-AI. DeepSeek-V3 Technical Report. arXiv:2412.19437, 2024. https://arxiv.org/abs/2412.19437.

[15] Sentence-Transformers Team. MultipleNegativesRankingLoss (documentation). https://www.sbert.net/docs/package_reference/losses.html#multiplenegativesrankingloss.

[16] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (system description). In *Automated Deduction – CADE-25*, LNCS 9195, pp. 378–388. Springer, 2015. https://doi.org/10.1007/978-3-319-21401-6_26.

[17] L. de Moura and S. Ullrich. The Lean 4 Theorem Prover and Programming Language (System Description). In *Automated Deduction – CADE 28*, LNCS 12699, pp. 625–635. Springer, 2021. https://doi.org/10.1007/978-3-030-79876-5_37.