

CSC490 Assignment 5

Team:

Son Nguyen (ID 1009656560), Kyle (ID 1007785229),

Daniel (ID 1008035378), Jinbo (ID 1004821419)

Project: ArXplorer - Semantic Search for arXiv

Part One

Our goals were:

- Measure where our system actually spends time.
- Identify the main bottlenecks in the embedding and search pipeline.
- Propose concrete optimizations we can implement in later parts or future work.

We focused on the embedding, indexing, and search components, since these are the core of our LLM-based semantic search system.

Profiling was done with Python 3.10 on Windows using the script:

- `a5/profile_a5_part1.py`

Each profiling run produced a “.prof” file, which we saved under:

- `a5/profiling_results/`

2. Functions Profiled

We profiled the following five functions:

1. `_encode_text()`
 - Module: `src.core.pipeline.EmbeddingGenerator`
 - Reason: Lowest-level “text → embedding” operation, used throughout the system.
2. `generate_embeddings()`
 - Module: `src.core.pipeline.EmbeddingGenerator`
 - Reason: Generates full embeddings for a `ProcessedPaper` (title + abstract, etc.). Called once per paper.
3. `build_index()`
 - Module: `src.core.pipeline.VectorIndexer`
 - Reason: Builds the FAISS vector index that powers semantic search.
4. `search_papers_by_text()` (mocked)
 - For Part 1, implemented via a `FakeRepo` inside `profile_a5_part1.py`.
 - Reason: Represents the search-by-text API path without requiring a real MongoDB setup.
5. `get_papers_by_categories()` (mocked)
 - Also implemented via `FakeRepo`.
 - Reason: Represents category-based search latency and async overhead.

For the last two functions we intentionally used a fake repository instead of a real MongoDB client. The goal in Part 1 is to profile code paths and runtime behavior, not to debug database configuration.

3. Profiling Setup

We used Python's cProfile module. The core helper in profile_a5_part1.py looks like this:

Function run_profile(label, func_str):

- Prints a header for the profiling section.
- Builds a filename such as "profile_encode_text_YYYYMMDD_HHMMSS.prof".
- Calls cProfile.run(func_str, filename).
- Loads the profile file into pstats.Stats and prints the top 20 entries, sorted by total time (totime).

Example usage:

- run_profile("encode_text", "profile_encode_text()")
- run_profile("generate_embeddings", "profile_generate_embeddings()")
- run_profile("build_index", "profile_build_index()")
- run_profile("search_text", "asyncio.run(profile_search_by_text())")
- run_profile("search_category", "asyncio.run(profile_search_by_category())")

We kept only the latest profile file per function (to avoid clutter) and used the console output (top 20 lines sorted by totime) for our analysis.

4. Profiling Results (Per Function)

4.1 encode_text()

Profile file: profile_encode_text_... .prof

Total runtime (latest run): approximately 0.73 seconds

Role:

- Takes a single string (e.g., title or abstract).
- Tokenizes it with a HuggingFace tokenizer.
- Runs a transformer model (sentence-transformers/all-MiniLM-L6-v2).
- Returns one embedding vector.

Top bottlenecks (by totime):

- SSL socket reads (method 'read' of '_ssl._SSLSocket'): about 0.17 seconds on the measured run.
- File I/O for tokenizer and model files (nt.stat, io.open, io.open_code).
- torch._C._nn.linear (transformer dense layers).
- JSON parsing for model configuration.

Interpretation:

- The first run is dominated by model and tokenizer loading (and sometimes remote cache access via SSL).
 - After the model is fully loaded and cached, subsequent calls are much faster and dominated by transformer forward pass.
 - This function is fundamental; any optimization here improves all higher-level embedding functions.
-

4.2 generate_embeddings()

Profile file: profile_generate_embeddings_... .prof
Total runtime (latest run): approximately 0.35 seconds

Role:

- Takes a ProcessedPaper object containing:
 - cleaned_title
 - cleaned_abstract
 - extracted_keywords, etc.
- Calls _encode_text() for the relevant fields (usually title + abstract).
- Produces a joint embedding representation for the paper.

Top bottlenecks:

- SSL reads (on some runs) and transformer model state loading (especially for the first call).
- torch.C.nn.linear and torch.layer_norm.
- A few JSON decode calls and model loading functions.

Interpretation:

- generate_embeddings() is essentially a small wrapper that calls _encode_text() multiple times.
- After warm-up, most cost comes from the transformer forward passes.
- Around 0.35 seconds per paper on CPU is acceptable for small datasets, but becomes significant for large-scale indexing.

Importance:

- This function is called for every paper we want to index.
 - It directly impacts offline preprocessing time and the index-building pipeline.
-

4.3 build_index()

Profile file: profile_build_index_... .prof
Total runtime (latest run): approximately 2.09 seconds for a test batch of 50 papers.

Role:

- Generates dummy ProcessedPaper objects.
- Uses EmbeddingGenerator to create embeddings.

- Builds a FAISS index (e.g., FLAT index) over these embeddings.

Top bottlenecks:

- `torch._C._nn.linear`: about 0.81 seconds.
- `torch.layer_norm`: about 0.15 seconds.
- `torch._C._nn.gelu`: about 0.10 seconds.
- `torch._C._nn.scaled_dot_product_attention`.
- BERT model forward passes in transformers (`modeling_bert.py`).

Interpretation:

- Most time is spent generating embeddings, not inside FAISS itself.
- FAISS index creation is relatively cheap for 50 vectors, but embedding generation dominates.
- As the number of papers grows, `build_index()` becomes a combination of “embedding many papers” plus “FAISS indexing”.

Importance:

- Determines how fast we can build or rebuild our semantic index.
 - For production, this would be part of an offline pipeline that we want to make efficient.
-

4.4 `search_papers_by_text()` (Mocked)

Profile file: `profile_search_text_... .prof`

Total runtime (latest run): approximately 0.002 seconds

Role in Part 1:

- In `profile_a5_part1.py`, we define a `FakeRepo` with an `async` method `search_papers_by_text`.
- The method simply returns a list of fake documents with a given limit (e.g., 10 items).
- We then profile `asyncio.run(profile_search_by_text())`.

Key observations from the profile:

- Almost all time is spent in `asyncio` event loop functions:
 - `base_events.run_until_complete`
 - `windows_events._poll`
 - `socketpair` and overlapped I/O operations.
- Our own application logic (creating a list of dictionaries) is negligible.

Interpretation:

- This setup measures the overhead of the `async` infrastructure rather than real DB latency.
- In a real deployment with MongoDB, the major cost would come from:
 - Network I/O to MongoDB.

- Query execution and index usage.

Reason for mocking in Part 1:

- Avoids failure due to missing MongoDB collections or text indexes.
 - Keeps the focus of Part 1 on profiling Python code, not database configuration.
-

4.5 get_papers_by_categories() (Mocked)

Profile file: profile_search_category_... .prof

Total runtime (latest run): approximately 0.003 seconds

Role in Part 1:

- Similar to search_papers_by_text(), we use a FakeRepo with get_papers_by_categories.
- The mock method returns a list of fake documents where “category” is set to the first requested category.

Key observations:

- Again, most of the time shows up inside asyncio event loop internals.
- The user-level code is almost free compared to the async overhead.

Interpretation:

- As with the text search case, the real bottleneck in a production environment would be MongoDB queries and indexes, not the wrapper function.
-

5. Cross-cutting Observations

1. Model loading vs. steady-state performance

- The first call to embedding functions (_encode_text and generate_embeddings) is relatively slow due to:
 - Download and caching of the HuggingFace model.
 - Tokenizer and configuration file loading.

- After warm-up, runtime is dominated by the transformer forward pass in PyTorch.

2. Embedding computation dominates index building

- In our build_index test, the bulk of the time is spent calling generate_embeddings for 50 papers.
- FAISS index construction itself is relatively cheap at this scale.

3. Async search overhead is small

- With the FakeRepo, search_papers_by_text and get_papers_by_categories complete in a few milliseconds.
 - The overhead is almost entirely the asyncio event loop.
 - Once we incorporate MongoDB, the main additional cost will be round trips and query execution.
-

6. Optimization Ideas

Based on the profiling results, we identified several optimizations:

6.1 Cache tokenizer and model

Problem:

- If EmbeddingGenerator re-initializes models or tokenizers frequently, we pay repeated initialization costs.

Idea:

- Ensure the tokenizer and model are loaded once in EmbeddingGenerator.**init** and reused across all calls.
- Optionally, treat EmbeddingGenerator as a singleton in the application layer so the model is only created once per process.

Expected benefit:

- Removes expensive re-initialization on repeated requests.
 - Reduces cold-start overhead in production.
-

6.2 Batch embedding for title and abstract

Problem:

- generate_embeddings calls _encode_text multiple times for different fields of the paper (title, abstract, etc.).

Idea:

- Use batch encoding with the underlying model (e.g., encode [title, abstract] in one call).
- For bulk operations like indexing, also process multiple papers per batch.

Expected benefit:

- Better CPU/GPU utilization and less Python overhead.
 - Potentially 1.5x or better speedup for per-paper embedding generation.
-

6.3 Avoid rebuilding FAISS index from scratch

Problem:

- build_index currently assumes we generate embeddings and build the FAISS index every time we call it.

Idea:

- Move embedding generation into a separate offline preprocessing step.
- Save the FAISS index once and reload it on startup using:
 - faiss.write_index
 - faiss.read_index

Expected benefit:

- Faster service startup.
 - Cheaper incremental updates (only rebuild when the corpus changes significantly).
-

6.4 Use more advanced FAISS index types for large scale

Problem:

- A flat L2 index (IndexFlatL2) is simple and exact but not necessarily efficient for very large datasets.

Idea:

- For large collections, migrate to IVF-based or HNSW-based indexes:
 - IndexIVFFlat with a trained quantizer.
 - Potentially HNSW for better recall/speed trade-offs.

Expected benefit:

- Faster queries and lower memory usage at the cost of slightly approximate results, which is usually acceptable in semantic search.
-

6.5 Warm-up and prefetching

Problem:

- First requests suffer from model loading and potential caching overhead.

Idea:

- On application startup:
 - Instantiate EmbeddingGenerator once.
 - Run a dummy call to `_encode_text` as a warm-up.
 - This shifts the cost to startup time instead of the first user-visible request.
-

7. Next Steps

For later parts of A5 and future work, we plan to:

1. Implement at least one of the proposed optimizations in the actual codebase (for example, batching in generating embeddings and persisting FAISS indices).
2. Use Locust in A5 Part 2 to measure:
 - End-to-end latency for search APIs.
 - The effect of caching and warm-up on P95/P99 latency.
3. In A5 Part 3, design a scaling architecture that:
 - Separates embedding computation from query serving.

- Allows horizontal scaling of the search service.
 - Uses offline jobs for index building and refreshing.
-

8. Files Included

- a5/profile_a5_part1.py
 - Profiling harness for the five target functions.
- a5/profiling_results/
 - One .prof file per function (latest run):
 - profile_encode_text_... .prof
 - profile_generate_embeddings_... .prof
 - profile_build_index_... .prof
 - profile_search_text_... .prof
 - profile_search_category_... .prof

Part Two

Please refer to the report ‘a5/A5_PART2_LOAD_TESTING_REPORT.md’

Part Three

Please refer to the report ‘a5/A5_PART3_SCALING_ANALYSIS.md’