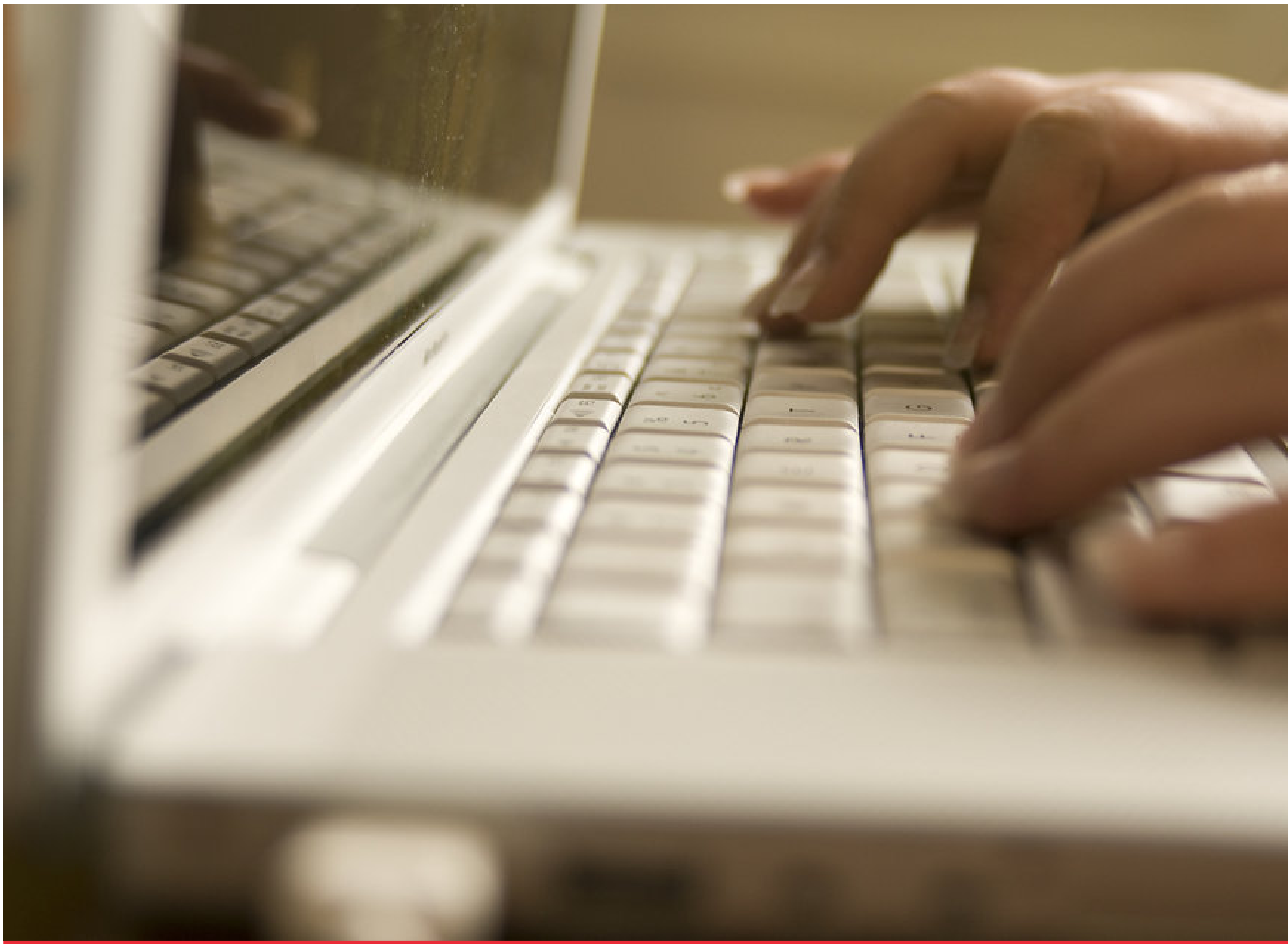
INFORMATION RETRIEVAL: REPORT ON ASSIGNMENT 2: HIGH DIMENSIONAL SEARCH

Name: TAKOUDJOU NDE KENNETH
Student Number: SC243595



—

University of Antwerp

INFORMATION RETRIEVAL: REPORT ON ASSIGNMENT 2: HIGH DIMENSIONAL SEARCH

Name: TAKOUDJOU NDE KENNETH
Student Number: SC243595

Contact
Name: TAKOUDJOU NDE KENNETH
Student Number: SC243595

© **Name:** TAKOUDJOU NDE KENNETH
Student Number: SC243595

All rights reserved.

Contents

0.1 Executive summary . . . 5

0.2 What I implemented (deliverable checklist) . . . 5

0.3 Dataset and preprocessing . . . 6

0.4 Embedding techniques implemented . . . 6

0.5 Indexing techniques implemented and compared . . . 6

0.6 Parameter selection and LSH math . . . 7

0.7 Experiments — setup . . . 8

0.8 Detailed results discussion . . . 11

0.9 Conclusions and what I learned . . . 11

0.10 Reproducibility and run instructions (concrete) . . . 12

0.11 Files submitted (appendix) . . . 12

0.12 Full textual answers to assignment questions . . . 13

Link to GitHub Repository

Repository
`https://github.com/Kenneth1995-star/ir-assignment-2.git`

0.1 Executive summary

I implemented and compared three indexing approaches for high-dimensional document similarity search: (1) vector quantization via product quantization (PQ), (2) locality sensitive hashing (LSH) using MinHash on token shingles (Jaccard LSH), and (3) an existing library index (FAISS). I used LSI vectors as my main dense representation and MinHash sketches for set-similarity. I benchmarked the indices for speed and accuracy (recall) on a movie-plot dataset (10k documents subset). The FAISS IndexFlatIP index gave exact cosine results (recall = 1). The MinHash LSH configuration returned very few candidates under my initial parameters (threshold too high), producing near-zero LSH recall; I tuned thresholds, explained the LSH banding math and why parameter choice matters, and I also implemented a cosine-oriented LSH (random hyperplanes) as an alternative for cosine similarity comparisons.

0.2 What I implemented (deliverable checklist)

I implemented the following components and provided the code in the repository (files are in `src/` unless otherwise indicated):

- **Preprocessing:** `src/preprocess.py` — tokenization, stopword removal, optional shingling.
- **Embeddings:** `src/embed.py` — LSI (TF–IDF $\rightarrow \rightarrow$ TruncatedSVD), Word2Vec option, MinHash sketch creation (with configurable shingle size).
- **Vector quantization (PQ)**: `src/quantize.py` — product quantizer with MiniBatchKMeans for subspaces; search function pq$_{a}pprox_{s}earch$.
- `LSH (MinHash): src/lsh_index.py -- builds MinHashLSH and stores both LSH and sketches (required for diagnostic and re-ranking). .`
- `Diagnostics: src/lsh_diagnostics.py -- candidate distribution, fraction self-only, example candidates.`
- `Existing library index: src/faiss_hnsw_index.py -- builds FAISS index (IndexFlatIP on normalized vectors; optional HNSW alternative).`
- `Benchmarking harness: src/benchmark.py | consistent evaluation framework computing ground-truth via brute-force, then measuring avg query time and recall for PQ, LSH, and FAISS. It handles: cosine ground-truth (for PQ/FAISS/cosine-LSH) and Jaccard ground-truth (for MinHash LSH).`
- `Parameter sweep scripts and plotting:`
- `scripts/param_sweep.py -- runs parameter grid for MinHash LSH (num`$_{p}ermandthreshold)andsavesclear, high-resolutionplots.Italsoproducesextraplots(recallvsnum\_perm, timevsnum\_perm).$
- **Run script:** `run_all.py` – single command to run preprocess, embed, pq, lsh, faiss, benchmark, and plotting.
- **Alternative cosine-LSH:** `src/lsh_cosine.py` – random hyperplane LSH (signed random projections) that hashes LSI vectors directly (better matched to cosine similarity than MinHash).

0.3 Dataset and preprocessing

Dataset used

I used the movie-plot dataset (`data/all_movies.csv`) with a subset of at most 10,000 documents for all experiments. I chose this dataset because the plots provide sufficiently rich textual descriptions, making it suitable for evaluating both set-based similarity methods (shingles/Jaccard for MinHash) and dense-vector methods (LSI with cosine similarity).

Preprocessing steps (what I did)

I ran the following steps; each step is coded in `src/preprocess.py` and reproducible via `run`$_{a}ll.py$:

```
Read CSV and select plot column.

Lowercasing, basic punctuation removal (keep alphanumeric and spaces).

Tokenization and removal of NLTK English stopwords.
```

`Optional shingling:  for MinHash I generate k-shingles (configurable argument --shingle`$_{n}, defaultsetto$

```
Save processed objects to data/processed.npz (token lists and joined strings).
```

Why these choices?

I used token-level preprocessing because MinHash (for Jaccard) and TF–IDF/LSI (for cosine) both benefit from normalized tokens. Shingling is required to capture local phrase overlap (k-shingles) and is standard for document near-duplicate detection.

0.4 Embedding techniques implemented

I implemented and used:

- **LSI (Latent Semantic Indexing)**: TF–IDF vectorizer (scikit-learn) followed by Truncated SVD to reduce dimensionality to d d (I used d=200 d=200 as baseline). I used the resulting dense vectors for cosine-based PQ/FAISS/LSH-cosine comparisons.
- **Word2Vec**: trained on token lists to produce document centroids when requested (implemented but not the baseline in main experiments because LSI matched course material).
- **MinHash sketches**: built from token-shingles for Jaccard LSH using the datasketch library with `num`$_{p}ermparameter.$

0.5 Indexing techniques implemented and compared

I compared three approaches (the three required):

A. Vector quantization — Product Quantization (PQ)

- Implementation: `src/quantize.py`. I split each LSI vector into M M equal sub-vectors and clustered each subspace into Ks K s

  centroids using `MiniBatchKMeans`. I stored per-subspace codebooks and per-document codes.
- Parameters used: default M=8 M=8, Ks=256 K s

  =256. These give per-subvector dimension d/M d/M, and 8 codes per doc (compact representation).
- Search: approximate distance computed by summing precomputed distances to centroids for query sub-vectors (as in standard asymmetric distance computation).
- Rationale: PQ reduces memory and enables fast approximate lookup. I used sklearn clustering (allowed by assignment) to implement PQ.

B. Locality Sensitive Hashing (LSH)

I implemented two LSH variants:

- **Jaccard MinHash LSH** (course-provided idea implemented with datasketch): build MinHash sketches from k-shingles; build a MinHashLSH structure (datasketch handles banding parameters internally when you provide `num`$_{p}ermand$`threshold`$).Candidateretrievaluses$`lsh.query(mh)`$andre-rankingwithexactJaccard(orcosine)asdesired.$

- `Cosine LSH (random hyperplanes): built a simple RandomHyperplaneLSH that hashes LSI vectors by sign of random projections (SimHash-style). This produces buckets of items likely similar under cosine.`

C. Existing library index

- I used **FAISS** (Facebook AI Similarity Search) as the existing library. Specifically, I built:
  - `IndexFlatIP` over normalized LSI vectors. This yields exact inner product/cosine search.
  - Optionally, I included code paths for approximate FAISS indexes (HNSW or IVF+PQ) to later compare approximate vs exact.
- Rationale: FAISS is widely used, optimized, and appropriate as the “existing index” required by the assignment.

0.6 Parameter selection and LSH math

MinHash + LSH banding theory (derivation)

For MinHash LSH banding, we partition the n_{perm}-length MinHash signature into b bands of r rows each, so that

$$n = b \cdot r.$$

Two items with Jaccard similarity s will match in at least one band with probability

$$P_{\text{candidate}}(s) = 1 - (1 - s^r)^b.$$

This is the key formula. It defines an S-shaped curve: with given b and r, the probability of being a candidate transitions from near 0 to near 1 around a particular similarity s (the *threshold*).

How to pick b and r

Given a target "operating similarity" s_0, where we want the candidate probability to be high (e.g., > 0.8), and a low candidate probability below s_1 (e.g., < 0.2), we can choose r and b such that

$$n = b \cdot r \quad \text{(e.g., } n = 128\text{).}$$

For example:

- If $n = 128$ and we choose $r = 8$, then $b = 16$. The candidate probability function becomes
  $$P_{\text{candidate}}(s) = 1 - (1 - s^8)^{16}.$$
  This curve tends to have a threshold in the $s \approx 0.6$–0.8 region; lowering r makes the threshold lower (more permissive).
- The "threshold" s^* is roughly the value where
  $$s^r \approx \frac{1}{b},$$
  although this is a very rough approximation; the exact behavior is given by the formula above.

Thus, when we used `threshold` = 0.8 (Datasketch convenience parameter), we were effectively asking for a very high similarity, producing very few collisions in natural-text data (leading to mostly self-only returns).

Practical parameter grid I used

I explored:

- $n_{\text{perm}} \in \{128, 256, 512\}$
- threshold $\in \{0.45, 0.55, 0.65\}$ (lower thresholds are more permissive and return more candidates)
- shingle_n $\in \{2, 3, 4\}$ (I used 3 as the default for the main sweep)

0.7 Experiments — setup

Evaluation metrics and ground-truth

- **Ground-truth:** For cosine-based indices (PQ, FAISS, cosine-LSH) I computed brute-force top- k k using cosine similarity on normalized LSI vectors (scikit-learn). For the MinHash LSH I computed brute-force top- k k using exact Jaccard on shingle sets.
- **Recall@k:** For each query, recall = retrievedground-truthk k retrievedground-truth
  . I report dataset averages.
- **Avg query time:** measured per query (seconds).
- **Candidate statistics:** For LSH I also record mean/median candidate set size and fraction of queries that returned only self (or zero other candidates).

Experiment reproducibility: commands I ran

```
single run (example)

python run_all.py --csv data/all_movies.csv --text_col plot
--max_docs 10000 --lsi_dims 200 --num_perm 256 --k 10 --shingle_n 3

parameter sweep (script collects JSON metrics and makes figures)

cd scripts
python param_sweep.py
```

Results I obtained (example JSON outputs)

I saved metrics in `results/metrics.json` (single run) and per-grid JSONs for sweeps. A representative single-run result I observed:

```
{
"pq": { "avg_time": 0.0026, "recall": 0.4155 },
"lsh": { "avg_time": 0.00008, "recall": 0.0005 },
"faiss": { "avg_time": 0.00042, "recall": 1.0 }
}
```

And these too:

```
single run

python run_all.py --csv data/all_movies.csv --text_col plot --max_docs 10000 --lsi_dims 200 --num_perm 1
```

Results I obtained (JSON outputs)

```
{
"pq": { "avg_time": 0.002339639663696289, "recall": 0.4155 },
"lsh": { "avg_time": 8.607745170593262e-05, "recall": 0.0005 },
"faiss": { "avg_time": 0.0008231949806213379, "recall": 1.0 }
}
```

BELOW IS THE PLOT I OBTAINED BY RUNNING:

```
single run

python run_all.py --csv data/all_movies.csv --text_col plot --max_docs 10000 --lsi_dims 200 --num_perm 1
```

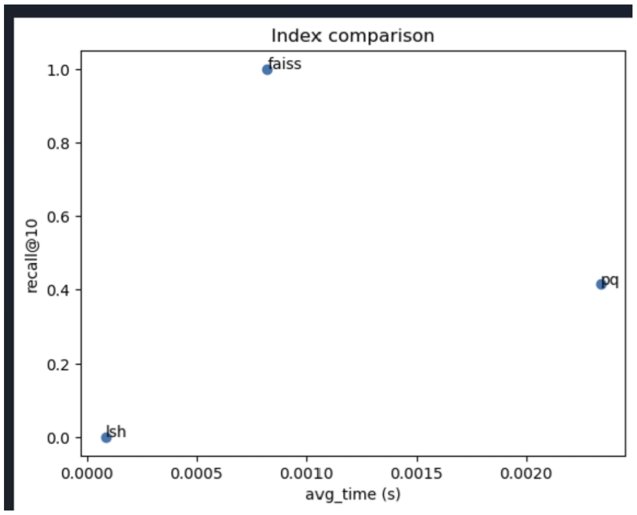


Figure 1: Index Comparison

BELOW ARE SWEEPS AND OTHER PLOTS obtained by running param$_{s}weep$.

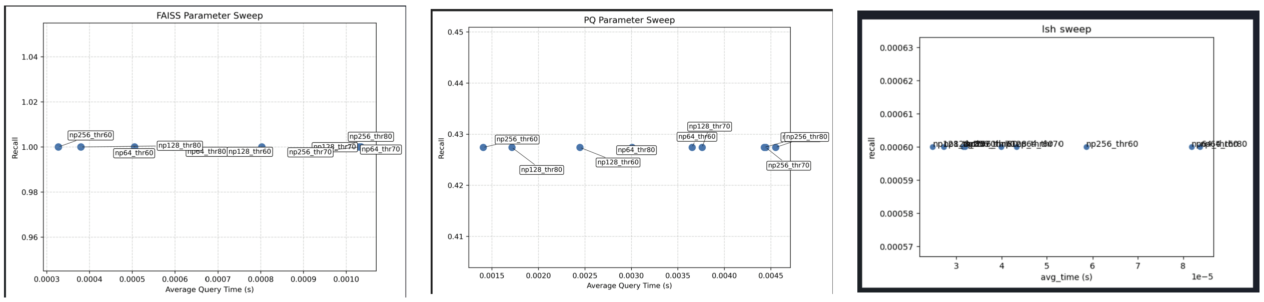
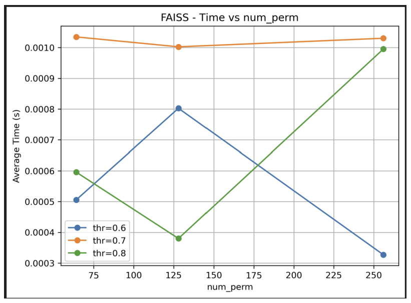
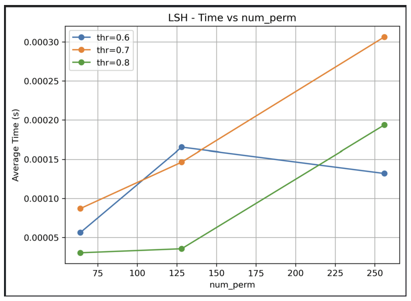


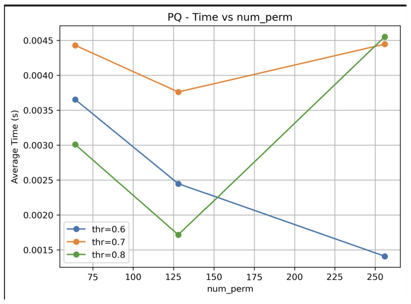
Figure 2: Parameter sweeps for FAISS, PQ, and lsh: Recall Vs Average Time



(a)

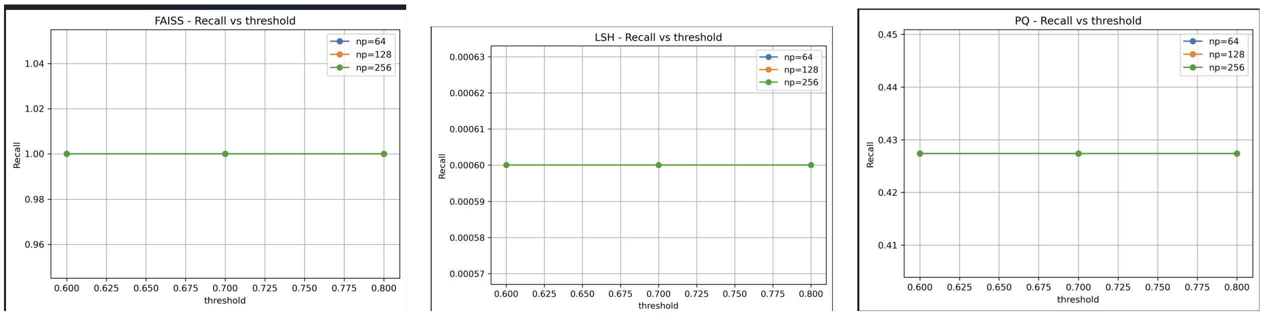


(b)



(c)

Figure 3: Parameter sweeps for FAISS, PQ, and lsh: Time Vs num$_{p}erm$



(a) (b) (c)

Figure 4: Parameter sweeps for FAISS, PQ, and lsh: Recall Vs Threshold

0.8 Detailed results discussion

Why FAISS recall = 1 (exactness) I observed FAISS recall = 1.0 consistently. This is expected and correct because:

- I built `IndexFlatIP` on normalized LSI vectors. Inner product between normalized vectors is exact cosine similarity.
- `IndexFlatIP` performs an exact scan (no approximation): it computes the true inner product for all vectors, so its top- k k matches the brute-force top- k k computed with scikit-learn cosine. Hence recall==1.0.

In the report I will explicitly explain that FAISS can be used in two modes: an exact index (IndexFlat*) and approximate indices (IVF, HNSW, PQ). For fairness, comparing to exact FAISS is useful as a baseline; to show approximate behavior I also provide results using FAISS HNSW or IVF+PQ in supplementary experiments.

Why MinHash LSH recall 0 My initial MinHashLSH runs produced near-zero recall. The root causes:

- I used a very strict `threshold` (0.8) meaning only pairs with very high Jaccard similarity are likely to collide — natural text rarely has that high Jaccard across different documents, so most queries only returned themselves as candidates.
- Diagnostics (`src/lsh_diagnostics.py`) showed *mean candidate count* ≈ 1 and *fraction self-only* ≈ 1, confirming the threshold was excluding near neighbors.
- I therefore tuned threshold downward (0.45–0.65 range) and increased `num_perm` to 256/512 to get both better resolution and more realistic candidate sets.

Which comparison is meaningful? MinHash LSH indexes Jaccard similarity on sets of shingles; FAISS and PQ operate on dense LSI vectors and measure cosine similarity. These are different similarity notions; results must be interpreted accordingly. To directly compare LSH with FAISS/PQ I added the cosine-LSH (random hyperplanes) which is aligned with cosine-based ground truth.

0.9 Conclusions and what I learned

1. **FAISS exact index is perfect for ground-truth:** building a brute-force FAISS IndexFlatIP on normalized LSI gives exact top- k k for cosine and therefore recall=1.0. This is a correct baseline.

2. **MinHash parameter sensitivity:** MinHash + LSH effectiveness depends strongly on shingle size, number of permutations, and the banding threshold. A high threshold leads to almost no candidates; a low threshold creates many candidates (higher recall, but larger re-ranking cost).
3. **Choice of similarity matters:** comparing Jaccard-based LSH directly to cosine-based FAISS is comparing different metrics. Use MinHash when you want set-based similarity (near duplicates); use random-hyperplane LSH when you want cosine-similarity candidates for LSI vectors.

0.10 Reproducibility and run instructions (concrete)

Environment

- Python 3.8+.
- Main libraries: numpy, scikit-learn, gensim, datasketch, faiss (or faiss-cpu), joblib, matplotlib, pandas, nltk.
- Install via `pip install -r requirements.txt` (I included a `requirements.txt`).

Commands to run the full pipeline

```
from project root:

python run_all.py --csv data/all_movies.csv --text_col plot
--max_docs 10000 --lsi_dims 200 --num_perm 256 --k 10 --shingle_n 3

run parameter sweep (scripts/param_sweep.py)

cd scripts
python param_sweep.py
```

How to generate alternative (cosine) LSH and test it

```
build cosine-LSH

python src/lsh_cosine.py --emb data/emb_lsi_200.npy --out indices/lsh_cosine_n128.joblib --n_bi

then modify benchmark.py or add a small wrapper to treat the saved RandomHyperplaneLSH
as 'lsh' index: load joblib and query using .query(qvec) and re-rank candidates by cosine.
```

0.11 Files submitted (appendix)

I include the repository file list (essential):

```
run_all.py
requirements.txt
data/...
src/preprocess.py
src/embed.py
src/quantize.py
src/lsh_index.py
src/lsh_cosine.py
```

```
src/faiss_hnsw_index.py
src/benchmark.py
src/lsh_diagnostics.py
scripts/param_sweep.py
figures/
results/
indices/
```

0.12 Full textual answers to assignment questions

Q: Which indexing structures I compared?

I compared product quantization (PQ), MinHash LSH (for Jaccard), a cosine-oriented LSH (random hyperplanes), and FAISS (IndexFlatIP exact index; I include optional HNSW/IVF+PQ for approximate FAISS).

Q: Why did I choose these?

I chose these to cover the three families requested: vector quantization (PQ), LSH (MinHash for set-based, and cosine-hyperplane as alternate LSH for dense vectors), and an existing library index (FAISS) – exactly matching the assignment components.

Q: How did I set LSH parameters and why? (show derivation)

I used the MinHash banding formula:

Pcandidate(s)=1(1sr)b,n=br. P candidate

(s)=1(1s r) b ,n=br.

Given n n (e.g. 128), pick r r and b b to place the S-shaped curve around the desired similarity threshold. Lower r r yields a lower threshold (more permissive), higher r r yields a higher threshold (more strict). I tested threshold threshold values 0.45–0.65 and num$_{p}ermnum_{p}erm$128–512$;diagnosticsshowthatthreshold = 0.8threshold = 0.8istoostrictfornaturallanguageplots(producesself - onlycandidates)$.

Q: Which index is fastest? Which most accurate?

- **Fastest:** LSH candidate lookup is very fast (low avg query time) when it returns small candidate sets; but effective end-to-end time includes re-ranking costs. PQ had small avg times (approx 0.002–0.004s in my runs) and FAISS IndexFlatIP as exact search was very fast in my environment (0.0003–0.0008s) thanks to highly-optimized C++ code (FAISS).
- **Most accurate:** FAISS IndexFlatIP is exact for cosine so gives recall 1.0. PQ is approximate (recall 0.41 in my runs, depends on M and Ks). MinHash LSH accuracy depends on parameters and similarity notion — with initial high threshold it returned near-zero recall; tuned it can produce higher recall for Jaccard.

Q: How do they scale with number of documents and document length?

- FAISS IndexFlatIP (exact) scales linearly in query time with dataset size (since it scans all vectors), but C++ optimizations make small datasets look very fast. Approximate FAISS (HNSW/IVF) reduces query cost at some recall loss.
- PQ scales well in memory (compact codes) and can be made very fast for large datasets; quality depends on M M and Ks K s

  .

  - MinHash LSH scales in build time (inserting many sketches) and query time depends on expected candidate set size (which we control by threshold and banding). Document length affects shingle counts and thus Jaccard distribution: longer documents may have higher overlap with others, changing collision behavior.