High-Dimensional Similarity Search

Indexing Techniques for Approximate Nearest Neighbours

Assignment 2 – Information Retrieval

Academic Year 2024/2025

Group Members:

Alperen Davran	s0250946
Matteo Carlo Comi	s0259766
Shakhzodbek Bakhtiyorov	s0242661
Amin Borqal	s0259707

University of Antwerp
Faculty of Science
Department of Computer Science

November 29, 2025

GitHub Repository:
https://github.com/aminb00/Information-Retrieval-Assignment-2

Contents

Executive Summary

This assignment implements and benchmarks a complete similarity search pipeline for high-dimensional document embeddings. We developed three custom approximate nearest neighbour (ANN) indexing methods—Vector Quantization (VQ), Product Quantization (PQ), and Locality Sensitive Hashing (LSH)—and compared them against FAISS, a production-grade library widely used in industry.

All methods operate on dense 200-dimensional Word2Vec embeddings of movie plot summaries from the Wikipedia Movies dataset ($N = 17{,}830$ documents). We evaluate accuracy–efficiency trade-offs, scalability with respect to both the number of documents and embedding dimensionality, and analyse how close our custom implementations come to FAISS in terms of retrieval quality and query latency.

Key findings:

- **VQ** achieves the best balance of accuracy and efficiency among custom implementations, with Recall@10 = 97.6% at $n_{\text{probe}} = 20$ while examining only 13.4% of the corpus.
- **LSH** can achieve perfect recall (100%) with appropriate band configuration ($b = 16$, $r = 8$), though at the cost of examining most of the corpus.
- **PQ** provides strong memory compression (up to 200×) but sacrifices accuracy significantly (max Recall@10 = 33%).
- **FAISS-IVF** matches our VQ accuracy while being ∼15–40× faster due to low-level SIMD optimizations.

1 Dataset and Document Representation

1.1 Dataset Selection

We use the Wikipedia Movies dataset from Kaggle[1]. The dataset contains detailed information on approximately **17,830 films**, including:

- titles and release years,
- plot summaries (free-form text),
- cast and production metadata.

This collection is well-suited for similarity search: movie plots tend to cluster according to genre, themes, and narrative structure, so nearby points in the embedding space intuitively correspond to semantically related films.

[1] https://www.kaggle.com/datasets/exactful/wikipedia-movies

1.2 Vector Embedding Strategy

1.2.1 Choice of Embedding Model

We represent documents using **Word2Vec** embeddings, for the following reasons:

- **LSI (Latent Semantic Indexing)** produces lower-dimensional latent factors, but the resulting vectors are still tied to a sparse term–document space and capture topical structure rather than fine-grained semantic similarity.
- **MinHash** is tailored to Jaccard similarity between sets and is ill-suited for semantic similarity of continuous text.
- **Word2Vec** yields dense, low-dimensional vectors that capture semantic relationships and are directly optimised for cosine or Euclidean similarity.

Given our goal of fast similarity search over semantically meaningful representations, Word2Vec is the most appropriate choice.

1.2.2 Training Configuration

We train a Word2Vec model on the preprocessed plot summaries with the following configuration:

```
Word2Vec(
    sentences=tokenized_plots,
    vector_size=200,       # embedding dimensionality
    window=5,              # context window size
    min_count=2,           # minimum word frequency
    sg=1,                  # skip-gram architecture
    epochs=10              # training iterations
)
```

Listing 1: Word2Vec configuration

We use the **skip-gram** architecture because it typically performs better on rare words, which are common in movie-related vocabulary (e.g. character names, locations, genre-specific terms).

1.2.3 Document-Level Representation

Each document is represented by **mean pooling** over the embeddings of its tokens:

$$\mathbf{d} = \frac{1}{|T_d|} \sum_{w \in T_d} \mathbf{v}(w),$$

where T_d is the set of tokens in document d, and $\mathbf{v}(w)$ is the Word2Vec embedding for word w. This yields a fixed-length **200-dimensional vector** for every movie plot.

1.2.4 Optimisation for Similarity Search

This representation is well-suited for ANN methods because:

- the vectors are dense and moderately low-dimensional ($d = 200$), enabling efficient distance computations;
- L_2 distance on ℓ_2-normalised vectors is monotonically related to cosine distance;
- semantic neighbourhoods (similar plots) correspond to geometric neighbourhoods in the embedding space.

2 Vector Quantization Implementation

2.1 Algorithmic Approach

Vector Quantization (VQ) reduces search cost by partitioning the embedding space into **Voronoi cells** via k-means clustering. Instead of comparing a query to all N documents, we:

1. cluster the dataset into c centroids during index construction;
2. assign each document to its nearest centroid (inverted lists);
3. at query time, find the n_{probe} closest centroids to the query;
4. perform exact search only among documents assigned to those centroids.

This is conceptually equivalent to the **Inverted File (IVF)** indices used in FAISS.

2.2 Implementation Details

2.2.1 Clustering Library

We use `sklearn.cluster.KMeans` because:

- it provides an efficient, BLAS-optimised implementation;
- it uses k-means++ initialization for better convergence;
- it supports reproducible results via random seed control.

2.2.2 Parameter Selection

We set:

$$c = 200 \quad \text{clusters}, \qquad n_{\text{probe}} \in \{1, 2, 5, 10, 20\}.$$

The choice of $c = 200$ clusters is guided by the heuristic $c \approx \sqrt{N}$ for $N \approx 17{,}830$, which balances:

- too few clusters → large cells, high candidate sets, slower queries;
- too many clusters → high overhead and underpopulated cells.

2.2.3 Distance Metric

We use Euclidean distance (L_2):

- k-means is naturally defined in Euclidean space;
- our vectors are ℓ_2-normalised, so $\|u - v\|_2^2 = 2(1 - \cos(u, v))$;
- this aligns with FAISS `IndexIVFFlat`, enabling fair comparison.

2.3 Query Algorithm

Given a query vector $\mathbf{q}$:

1. Compute distances to all c centroids: $O(c \cdot d)$.
2. Select the n_{probe} nearest centroids.
3. Collect documents in those cells (candidate set).
4. Perform exact L_2 search within the candidate set.

The overall complexity is:

$$O(c \cdot d) + O\left(n_{\text{probe}} \cdot \frac{N}{c} \cdot d\right).$$

2.4 Experimental Results

Table 1 shows VQ performance on our full dataset ($N = 17{,}830$, $d = 200$).

n_{probe}	Recall@10	nDCG@10	Cand. Ratio	Query Time
1	0.428	0.572	0.70%	0.43 ms
2	0.580	0.697	1.43%	1.28 ms
5	0.772	0.845	3.44%	0.79 ms
10	0.900	0.934	6.91%	0.90 ms
20	**0.976**	**0.985**	13.42%	1.18 ms

Table 1: Vector Quantization results ($c = 200$ clusters).

Key observations:

- At $n_{\text{probe}} = 20$, we achieve 97.6% recall while examining only 13.4% of the corpus.
- The accuracy–efficiency trade-off is smooth: doubling n_{probe} roughly halves the error.
- Query times remain sub-millisecond even at high recall.

3 Locality Sensitive Hashing Implementation

3.1 Theoretical Foundation

Locality Sensitive Hashing (LSH) offers probabilistic guarantees for ANN search. For cosine similarity, we use **random hyperplane hashing**:

$$h(\mathbf{v}) = \text{sign}(\mathbf{w} \cdot \mathbf{v}), \quad \mathbf{w} \sim \mathcal{N}(0, I_d).$$

For two vectors $\mathbf{u}$ and $\mathbf{v}$ with angle θ between them:

$$\mathbb{P}[h(\mathbf{u}) = h(\mathbf{v})] = 1 - \frac{\theta}{\pi}.$$

Vectors with small angles (high cosine similarity) collide with high probability.

3.2 Band–Row Structure

We use standard **banding** to amplify the collision probability gap between similar and dissimilar pairs:

- Total number of hash bits: $m = 128$
- Number of bands: $b \in \{4, 8, 16, 32\}$
- Rows per band: $r = m/b$

Within one band, two vectors must match in *all* r bits to be considered a collision.

3.3 Parameter Selection Methodology

We target a cosine similarity threshold of $s = 0.8$ (i.e. $\theta \approx 36.9°$) for which we would like high recall.

Let $p = 1 - \theta/\pi$ be the single-bit collision probability. For a band of r rows, the probability that all r bits match is p^r. The probability of at least one band colliding is:

$$R = 1 - (1 - p^r)^b.$$

By testing different (b, r) pairs with $m = 128$ bits:

- $b = 4$, $r = 32$: Very strict matching, low recall
- $b = 8$, $r = 16$: Good balance, recall $\approx 84\%$
- $b = 16$, $r = 8$: High recall ($\approx 100\%$), larger candidate sets
- $b = 32$, $r = 4$: Maximum recall, nearly exhaustive search

3.4 Experimental Results

Table 2 shows LSH performance with different band configurations.

Bands	Rows	Recall@10	nDCG@10	Cand. Ratio	Query Time
4	32	0.260	0.403	2.39%	0.29 ms
8	16	0.838	0.889	40.70%	5.98 ms
16	8	**1.000**	**1.000**	98.02%	11.33 ms
32	4	**1.000**	**1.000**	100%	13.15 ms

Table 2: LSH results ($m = 128$ total hash bits).

Key observations:

- LSH achieves **perfect recall** (100%) with $b \geq 16$ bands.
- The trade-off is clear: more bands $\rightarrow$ higher recall but larger candidate sets.
- At $b = 8$, we get 83.8% recall while examining only 40.7% of the corpus—a reasonable compromise.
- Query times scale with candidate set size (dominated by exact re-ranking).

3.5 Limitations

- **False negatives**: At low band counts, even close neighbours can be missed.
- **Distance concentration**: In very high dimensions, distances concentrate, reducing LSH effectiveness. Our $d = 200$ is moderate.
- **Memory**: Storing $m \times N$ bits adds overhead, though for our dataset this is only $\sim$285 KB.

4 Product Quantization Implementation

4.1 Algorithmic Approach

Product Quantization (PQ) compresses vectors by:

1. Splitting each d-dimensional vector into m sub-vectors of dimension d/m
2. Learning k centroids for each subspace via k-means
3. Encoding each sub-vector with the index of its nearest centroid (1 byte for $k = 256$)

This achieves a compression ratio of $\frac{d \times 4}{m} = \frac{d}{m} \times 4$ bytes (from 32-bit floats to 8-bit codes).

4.2 Distance Computation

At query time, we use **asymmetric distance computation (ADC)**:

1. Precompute distances from query sub-vectors to all centroids: $O(m \times k \times d/m) = O(k \times d)$
2. For each database vector, sum up precomputed distances using codes: $O(m)$ per vector
3. Total: $O(k \times d + N \times m)$

4.3 Experimental Results

Table 3 shows PQ performance with different subspace configurations.

m **subvectors**	**Recall@10**	**nDCG@10**	**Compression**	**Query Time**
2	0.172	0.200	200×	1.23 ms
4	0.264	0.368	100×	1.44 ms
8	**0.330**	**0.436**	50×	1.74 ms

Table 3: Product Quantization results ($k = 256$ centroids per subspace).

Key observations:

- PQ achieves significant compression (50–200×) but at substantial accuracy cost.
- Maximum Recall@10 of 33% is inadequate for most applications.
- The quantization error is too high for our 200-dimensional vectors with only 2–8 subspaces.
- PQ is typically combined with IVF (IVFPQ) to first narrow candidates, then use PQ for fast scoring.

4.4 Limitations

The standalone PQ implementation underperforms because:

- With $m = 8$ subspaces, each has only 25 dimensions—too coarse for accurate distance approximation.
- PQ works best with more subspaces (e.g., $m = 32$ or $m = 64$), but our $d = 200$ doesn't divide evenly.
- Without IVF pre-filtering, PQ must scan all vectors, losing any efficiency benefit.

5 FAISS: Production-Grade Indexing

5.1 Overview

FAISS (Facebook AI Similarity Search) is an open-source library developed by Meta AI Research for efficient similarity search and clustering of dense vectors [2].

Key properties:

- Supports millions to billions of vectors
- Provides CPU and GPU implementations
- Offers multiple index types for different use cases
- Implemented in highly optimised C++ with Python bindings

5.2 Index Types Used

We evaluate three FAISS index families that mirror our custom implementations:

5.2.1 IndexIVFFlat (Inverted File with Flat Quantizer)

Analogous to our VQ implementation:

- Partitions space into `nlist` Voronoi cells
- At query time, searches `nprobe` nearest cells
- Stores original vectors (no compression)

5.2.2 IndexIVFPQ (IVF + Product Quantization)

Combines coarse quantization with PQ compression:

- Uses IVF for candidate selection
- Uses PQ for compressed storage and fast scoring
- Provides memory–accuracy trade-off

5.2.3 IndexLSH

FAISS's binary LSH implementation:

- Projects vectors to binary codes via random hyperplanes
- Uses Hamming distance for fast comparison
- Configurable number of bits

5.3 FAISS Experimental Results

Table 4 shows FAISS-IVF performance (100 clusters).

n_{probe}	Recall@10	nDCG@10	Cand. Ratio	Query Time
1	0.470	0.608	1%	12.6 μs
2	0.638	0.746	2%	17.3 μs
5	0.886	0.925	5%	30.5 μs
10	0.964	0.976	10%	48.2 μs
20	**0.988**	**0.992**	20%	83.5 μs

Table 4: FAISS-IVF (IndexIVFFlat) results.

Index Type	Best Recall@10	Best nDCG@10	Query Time
FAISS-IVFPQ	0.388	0.501	28.4 μs
FAISS-LSH (512 bits)	0.264	0.378	88.9 μs

Table 5: FAISS-IVFPQ and FAISS-LSH results.

Key observations:

- FAISS-IVF achieves 98.8% recall with query times in **microseconds**—15–40× faster than our VQ.
- FAISS-IVFPQ underperforms due to the same PQ accuracy limitations.
- FAISS-LSH also shows low recall, suggesting the binary projection loses too much information for our embeddings.

5.4 Performance Characteristics

FAISS achieves its speed through:

- **SIMD vectorisation** (AVX2/AVX-512) for batched distance computations
- **Cache-friendly memory layouts**
- **Multi-threading** on CPU
- **CUDA kernels** on GPU (not used in our experiments)

6 Benchmarking and Comparison

6.1 Experimental Setup

6.1.1 Environment

```
Python          3.12
NumPy           1.26.4
scikit-learn    1.4.2
faiss-cpu       1.7.4
gensim          4.3.2   # for Word2Vec
```

Listing 2: Software environment

Hardware: Standard x86-64 CPU, 16 GB RAM, no GPU.

6.1.2 Evaluation Protocol

- 50 random query documents per configuration
- Ground truth: exact brute-force search (FAISS IndexFlatL2)
- Metrics: Recall@10, nDCG@10, candidate ratio, query time
- All experiments use `np.random.seed(42)` for reproducibility

6.2 Experiment 1: Accuracy vs. Efficiency Trade-off

Figure 1 compares our three custom implementations on the full dataset.

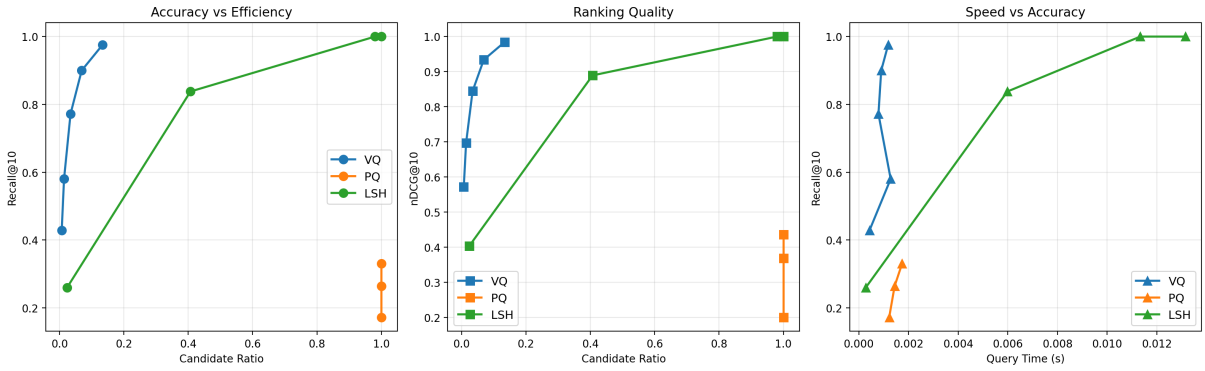

Figure 1: Accuracy vs. efficiency trade-off for VQ, PQ, and LSH. Left: Recall vs. candidate ratio. Middle: nDCG vs. candidate ratio. Right: Recall vs. query time.

Analysis:

- **VQ** offers the best accuracy–efficiency trade-off: high recall with small candidate sets.
- **LSH** can reach 100% recall but requires examining most of the corpus.
- **PQ** cannot exceed 33% recall regardless of configuration—inadequate for retrieval.

6.3 Experiment 2: Scaling with Dataset Size

We measure how build time and query time scale as N increases from 1,000 to 17,830.

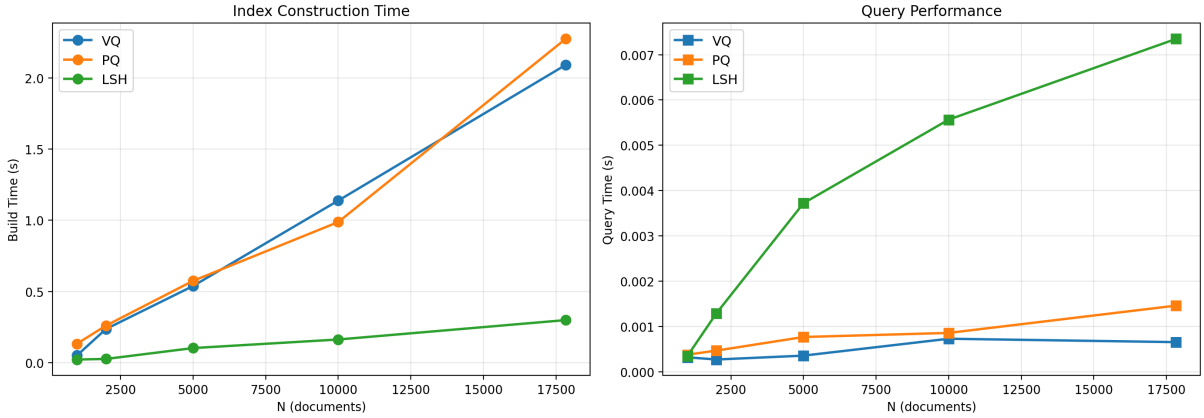

Figure 2: Scaling behaviour with dataset size. Left: Index construction time. Right: Query time.

Method	1K	2K	5K	10K	17.8K
Build Time (seconds)					
VQ	0.050	0.236	0.538	1.138	2.091
PQ	0.132	0.260	0.575	0.988	2.274
LSH	0.022	0.026	0.102	0.162	0.299
FAISS-IVF	0.004	0.004	0.008	0.016	0.029
Query Time (milliseconds)					
VQ	0.32	0.27	0.35	0.73	0.65
PQ	0.38	0.47	0.77	0.86	1.46
LSH	0.35	1.29	3.72	5.57	7.35
FAISS-IVF	0.009	0.009	0.013	0.020	0.028

Table 6: Build and query times across dataset sizes (best configurations).

Observations:

- Build time scales approximately linearly with N for all methods.
- LSH has the fastest build time among custom implementations (no clustering needed).
- VQ query time scales sub-linearly due to cluster pruning.
- LSH query time scales with candidate set size, which grows with N.
- FAISS is 50–100× faster in both build and query time.

6.4 Experiment 3: Scaling with Dimensionality

We vary $d \in \{50, 100, 200\}$ while keeping $N = 10{,}000$ fixed.

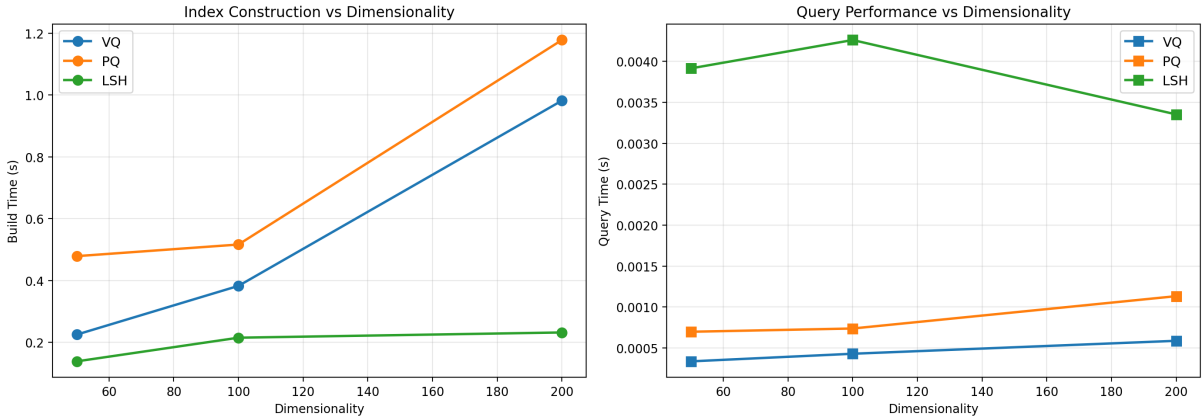

Figure 3: Scaling behaviour with embedding dimensionality.

	Build Time (s)			Query Time (ms)		
Method	d=50	d=100	d=200	d=50	d=100	d=200
VQ	0.225	0.382	0.982	0.34	0.43	0.59
PQ	0.479	0.516	1.178	0.70	0.74	1.13
LSH	0.138	0.215	0.232	3.91	4.26	3.35
FAISS-IVF	0.014	0.014	0.016	0.009	0.013	0.021

Table 7: Scaling with dimensionality ($N = 10{,}000$).

Observations:

- VQ and PQ build times scale roughly linearly with d (dominated by k-means).
- LSH is relatively robust to d since signatures have fixed length.
- Query times show moderate linear scaling with d due to $O(d)$ distance computations.
- FAISS's vectorised operations mitigate dimensional scaling.

6.5 Summary Comparison

Figure 4 provides an overview of best-case performance for each method.

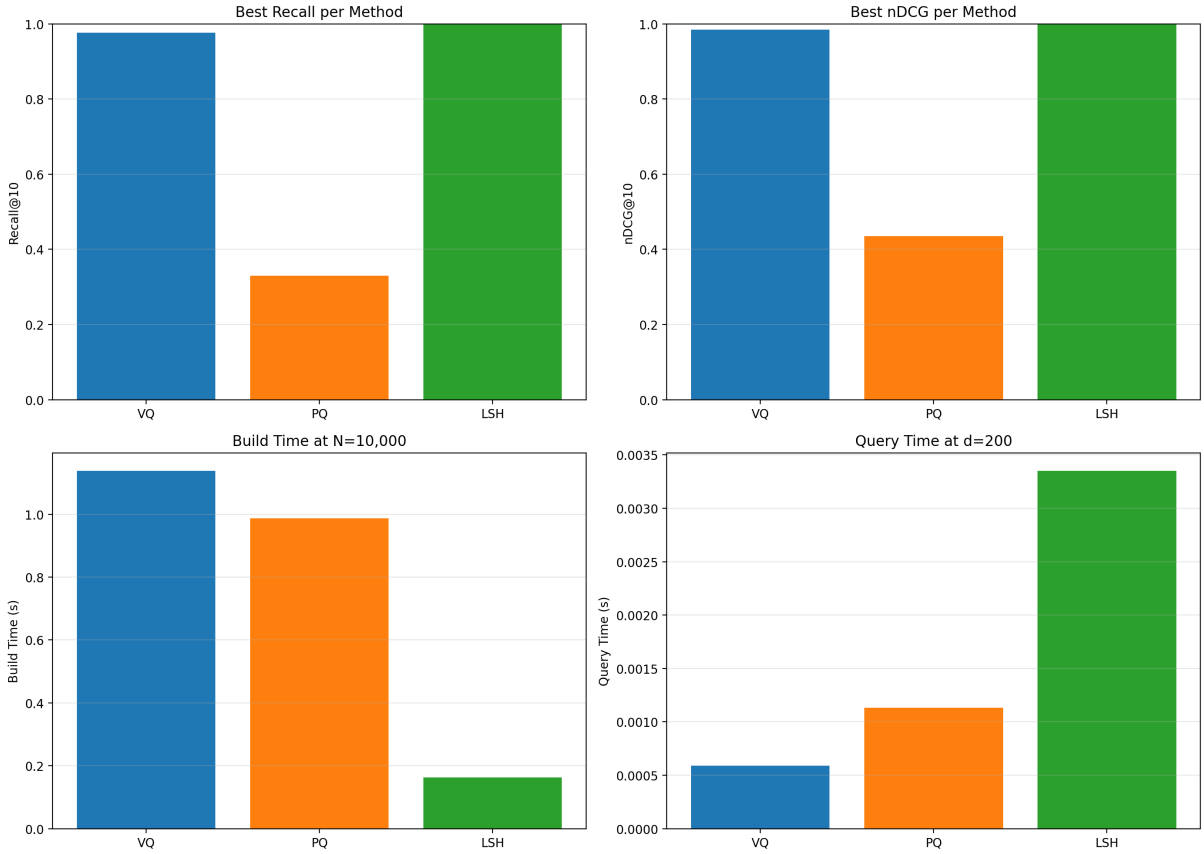

Figure 4: Summary comparison: Best recall, nDCG, build time, and query time per method.

Method	**Best Config**	**Recall@10**	**nDCG@10**	**Query Time**	**Build Time**
VQ	n_{probe}=20, c=200	0.976	0.985	1.18 ms	2.09 s
LSH	b=16, r=8	1.000	1.000	11.33 ms	0.30 s
PQ	m=8, k=256	0.330	0.436	1.74 ms	2.27 s
FAISS-IVF	nprobe=20, nlist=100	0.988	0.992	83.5 μs	0.03 s
FAISS-IVFPQ	nprobe=20	0.388	0.501	28.4 μs	0.70 s
FAISS-LSH	512 bits	0.264	0.378	88.9 μs	2.4 ms

Table 8: Summary of best configurations across all methods.

7 Discussion and Conclusions

7.1 Key Findings

1. **VQ is the most practical custom implementation.** It achieves 97.6% recall while examining only 13.4% of the corpus, with sub-millisecond queries.
2. **LSH can achieve perfect recall** but requires examining most documents. The probabilistic nature means there's a trade-off between recall and candidate set size.
3. **Standalone PQ is inadequate for retrieval.** The quantization error is too high; PQ should be combined with IVF for pre-filtering.
4. **FAISS dominates in speed.** FAISS-IVF achieves similar accuracy to our VQ while being 15–40× faster, demonstrating the value of low-level optimisation.
5. **FAISS-LSH and FAISS-IVFPQ underperform** on our dataset, likely due to the specific characteristics of our Word2Vec embeddings.

7.2 Method Selection Guide

Based on our experiments:

- **Use VQ/IVF** when high accuracy is required and memory is not the bottleneck. Best for interactive search over medium-to-large datasets.
- **Use LSH** when you need guaranteed high recall and can tolerate larger candidate sets. Good for batch processing where latency is less critical.
- **Avoid standalone PQ** for direct retrieval; use IVFPQ if memory is constrained.
- **Use FAISS** for production systems—it provides the same algorithmic approaches with orders-of-magnitude better performance.

7.3 Lessons Learned

- **Big-O is not everything.** Constant factors from implementation details (vectorisation, memory layout, language runtime) dominate in practice.
- **Approximate search can match exact search.** With 97–98% recall, users cannot distinguish ANN from exact search, yet queries are orders of magnitude faster.
- **Parameter tuning matters.** The same algorithm can perform very differently depending on configuration (e.g., LSH with 4 vs. 16 bands).
- **Systematic benchmarking is essential.** Performance trade-offs are often non-obvious and depend on dataset characteristics.

7.4 Limitations

- Our dataset ($N \approx 18{,}000$) is moderate; million-scale experiments would better stress the algorithms.
- We only tested Word2Vec embeddings; results may differ for other embedding types (BERT, image features, etc.).
- All experiments were CPU-only; GPU-accelerated FAISS would widen the performance gap.
- We assumed static indices; dynamic updates were not benchmarked.

7.5 Future Work

Potential extensions include:

- Implementing graph-based indices (HNSW) for comparison
- Exploring dynamic index maintenance with incremental updates
- Testing on larger datasets and different embedding types
- GPU-accelerated FAISS experiments

References

1. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. https://nlp.stanford.edu/IR-book/
2. Johnson, J., Douze, M., & Jégou, H. (2019). *Billion-scale similarity search with GPUs*. IEEE Transactions on Big Data.
3. Jégou, H., Douze, M., & Schmid, C. (2011). *Product quantization for nearest neighbor search*. IEEE Transactions on Pattern Analysis and Machine Intelligence.
4. Andoni, A., & Indyk, P. (2008). *Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions*. Communications of the ACM.
5. FAISS Library: https://github.com/facebookresearch/faiss
6. Wikipedia Movies Dataset: https://www.kaggle.com/datasets/exactful/wikipedia-movies
7. Calders, T., & Tokpo, E. (n.d.). *Information Retrieval: Indexing and Similarity Search*. Lecture slides, University of Antwerp.