IRRS Lab 6: Duplicate Detection using Simhash

Domenico Azzarito, Adrian Hagen

December 9, 2025

1 Introduction

The aim of this lab session is to implement the Simhash algorithm for near-duplicate detection within a corpus of scientific abstracts from arXiv.org. Unlike exact duplicate detection which can be solved with standard hashing (like MD5), near-duplicate detection requires Locality Sensitive Hashing (LSH). The goal is to map similar documents to the same buckets with high probability, calculating the Hamming distance between fingerprints to approximate Cosine Similarity.

2 Understanding the Simhash algorithm

The core idea of Simhash is to generate a fingerprint of b bits for each document such that the fingerprint preserves the cosine similarity of the original vectors.
The algorithm proceeds as follows:

1. Initialize a vector V of zeros of length b.
2. For each term t in the document with weight w_t:
   - Compute a hash h_t of b bits.
   - Convert bits to ± 1 (0 becomes -1).
   - Add $w_t \cdot h_t$ to V.
3. The final fingerprint is derived by setting the i-th bit to 1 if $V[i] \geq 0$, and 0 otherwise.

2.1 Manual Calculation (Exercise 1)

We verified the logic with a manual exercise. Given a document d with terms ‘bit‘ ($w = 0.4$) and ‘coin‘ ($w = 1.2$), and assuming $b = 4$:

- Hash(‘bit‘) $= 1111 \rightarrow [1, 1, 1, 1]$
- Hash(‘coin‘) $= 1001 \rightarrow [1, -1, -1, 1]$

The weighted sum calculation is:

$$V = 0.4 \cdot [1, 1, 1, 1] + 1.2 \cdot [1, -1, -1, 1]$$

$$V = [0.4, 0.4, 0.4, 0.4] + [1.2, -1.2, -1.2, 1.2] = [1.6, -0.8, -0.8, 1.6]$$

Converting signs to bits ($+ \rightarrow 1, - \rightarrow 0$):

$$\textbf{Simhash} = \textbf{1001}$$

3 Implementation Details

3.1 Simhash Generation

We implemented the `_simhash` function to process the 58,102 documents. To store the intermediate weighted sums, we use a `numpy.ndarray` We utilized the provided `_termhash` (based on MD5) to generate consistent random projections. The time complexity for generating fingerprints is linear with respect to the number of non-zero terms in the document collection.

3.2 LSH Indexing

To efficiently find candidates without comparing every pair ($O(N^2)$), we applied the standard LSH technique:

- The b-bit signature is split into m chunks of k bits each ($b = m \times k$).
- We maintain m hash tables, implemented as `list[dict[int, list[doc_id]]]` (one dict per table, mapping bucket_id to list of docs).
- A document ID is stored in the bucket determined by the integer value of the k-bit chunk for that specific band.

Two documents are considered **candidates** if they collide in *at least* one band. Candidate pairs are stored in a `set[tuple[doc_id, doc_id]]` data structure to deduplicate pairs from bucket collisions. This increases recall (probability of finding a similar pair) at the slight cost of precision.

4 Performance Analysis: M and K

The efficiency and accuracy of the LSH scheme depend heavily on the parameters m (number of bands) and k (bits per band). We benchmarked the system using various configurations, keeping m fixed to study k, and vice versa. We selected a threshold value of $\tau = 0.8$ for the cosine similarity on the original tf-idf representations of the abstracts. This means that every pair of documents with a cosine similarity larger than 0.8 would be considered an actual duplicate pair. Table 1 summarizes the results.

Table 1: LSH Simhash Benchmark Results (Similarity Threshold $\tau = 0.8$)

M	**K**	**Candidates**	**TP** (≥ 0.8)	**FP** (< 0.8)	**Precision**	**Time (s)**
3	12	1,392,508	17,878	1,374,630	0.013	165.00
3	18	40,276	16,244	24,032	0.403	110.46
3	22	17,038	15,492	1,546	0.909	126.43
4	18	48,027	16,824	31,203	0.350	139.29
5	18	56,020	17,420	38,600	0.311	164.70

4.1 Varying K (Bits per Band)

We fixed $m = 3$ and tested $k \in \{12, 18, 22\}$.

- **Low K (12):** Buckets are small (2^{12}), leading to many collisions. This results in huge candidate generation (1.4 million pairs) and extremely low precision (1.3%), as most collisions are accidental (False Positives).

- **High K (22):** Buckets are very specific (2^{22}). Precision jumps to 90.9%, but we risk missing true duplicates (lower recall) because documents must be almost identical to collide.

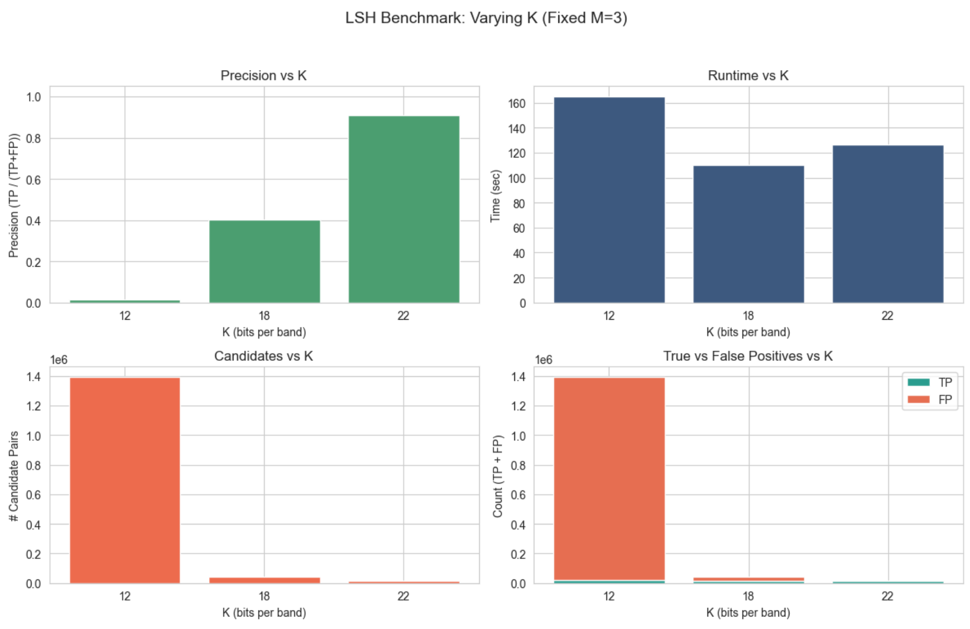

Figure 1: Impact of K on LSH Performance (fixed M=3)

4.2 Varying M (Number of Bands)

We fixed $k = 18$ and tested $m \in \{3, 4, 5\}$.

- Increasing m gives documents more "chances" to collide.
- As seen in Figure 2, raising m from 3 to 5 increased the number of True Positives found (from 16,244 to 17,420), improving recall.
- However, it also increased False Positives significantly, dropping precision from 40.3% to 31.1%.

5 Duplicate Detection Results

Using a similarity threshold of $\tau = 0.8$, together with the default values of $M = 3$ and $K = 18$, our system identified 16,244 True Positive pairs. The top duplicates found were identical or nearly identical abstracts, confirming the algorithm's correctness. For instance, document pairs like ('2938', '3016') and ('25608', '44584') had a cosine similarity of effectively 1.0.

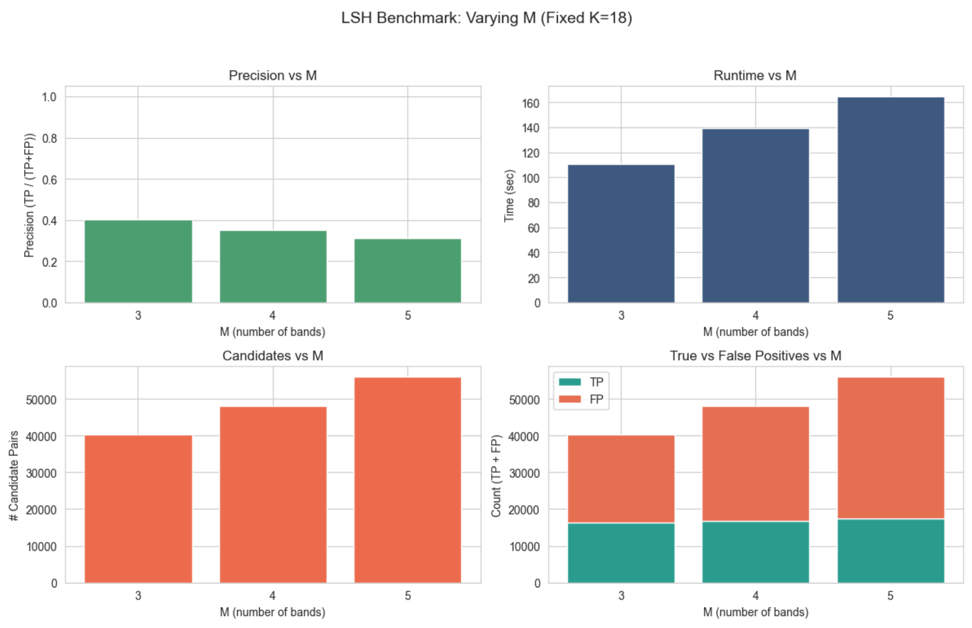

Figure 2: Impact of M on LSH Performance (fixed K=18)