In-Depth Analysis of Similarity Search and Locality Sensitive Hashing

Let me walk you through this fascinating topic as if we're having a tutorial session. I'll build from fundamentals to the sophisticated techniques described in this document.

Part 1: The Fundamental Problem

Why Do We Need Similarity Search?

Imagine you're Google. You have billions of web pages, and someone types a query. You need to find similar documents **fast**. The naive approach would be:

```
For each query:
    Compare query to all N documents
    Return K most similar ones
```

This is **O(N)** per query. With N = 10 billion documents, this is impossibly slow.

For **all-pairs similarity** (like finding duplicate web pages), it's even worse: compare each document to every other document = **$O(N^2)$** comparisons. For 10 billion documents, that's 10^{20} comparisons!

The Modern Twist: Dense Embeddings

Traditionally, documents were "sparse" - represented by which words they contain. Now, with neural networks, we represent documents as **dense vectors** in $\mathbb{R}^D$ (maybe D = 768 dimensions).

- Query $q \in \mathbb{R}^D$
- Documents $d \in \mathbb{R}^D$
- Need to find K "closest" documents quickly

This is where **Locality Sensitive Hashing (LSH)** becomes crucial.

Part 2: Traditional Hashing vs. LSH

Traditional Hash Functions: Dispersion is Key

In classic computer science, you want hash functions that **scatter** items uniformly:

Example: Storing a dictionary

Words → Hash function → Buckets [0, 1, 2, ..., M-1]

Goal: Minimize collisions. If "apple" and "orange" hash to the same bucket, that's BAD.

Weakly Universal Hash Family:

$$\Pr(h(x_1) = h(x_2)) \leq 1/M \text{ for all } x_1 \neq x_2$$

Strongly (2-Universal) Hash Family:

$$\Pr(h(x_1) = y_1 \wedge h(x_2) = y_2) = 1/M^2 \text{ for all } x_1 \neq x_2,\ y_1 \neq y_2$$

Example: `h(x) = (ax + b) mod p` (where p is prime) is weakly universal.

Perfect Hashing: The Gold Standard

For **static** datasets (no insertions/deletions), we can achieve:

- O(n) storage
- O(1) **worst-case** lookup time

How? Two-level hashing:

Level 1: Hash n items into n buckets using function f

- Keep choosing f until $\Sigma_i\ b_i^2 \leq 4n$ (where b_i = bucket size)
- Expected attempts: 2

Level 2: For bucket i with b_i items, allocate array of size ~b_i^2

- Find hash function g_i with no collisions within bucket
- Expected work: $O(b_i^2)$

Why b_i^2 space? By the **birthday paradox**, with ~b_i^2 slots, collision probability becomes tiny for b_i items.

Total storage: $\Sigma_i\ b_i^2 \leq 4n = O(n)$ ✓

Part 3: The LSH Philosophy - Collision is GOOD

The Paradigm Shift

In LSH, we want the **opposite** of traditional hashing:

> **Similar items SHOULD collide; dissimilar items should NOT**

This seems counterintuitive, but it's brilliant for search!

Different Distance Measures

Real-world applications need different notions of similarity:

1. **Hamming distance:** Number of differing bits (for bit vectors)
2. **L_1 distance:** $||a - b||_1 = \Sigma_j |a_j - b_j|$
3. **L_2 distance:** $||a - b||_2 = \sqrt{(\Sigma_j (a_j - b_j)^2)}$
4. **Cosine similarity:** $(a \cdot b)/(||a||_2||b||_2)$ [angle between vectors]
5. **Jaccard similarity:** $|A \cap B|/|A \cup B|$ [for sets]

Each needs a different LSH scheme!

Part 4: Hamming Distance LSH - The Master LSH

The Basic Idea

Setup: N-bit strings in {0,1}^N

Hash family F: $h_i(x) = x_i$ (the i-th bit of x)

Key Insight:

$$Pr(h_i(x) = h_i(y)) = 1 - ||x,y||\_H/N$$

If Hamming distance $||x,y||\_H \leq N/3$, then Pr(hash collision) $\geq 2/3$ If Hamming distance $||x,y||\_H \geq 2N/3$, then Pr(hash collision) $\leq 1/3$

Definition: Family F is **(c, r, P_1, P_2)-sensitive** if:

- Distance $\leq r \implies$ Pr(collision) $\geq P_1$
- Distance $\geq cr \implies$ Pr(collision) $\leq P_2$

With $c > 1$ and $P_1 > P_2$, we get **separation**.

Amplification: Making the Separation Stronger

One bit isn't enough discrimination. We **amplify**:

Step 1: Sample k bit positions → k-bit sketch **Step 2:** Repeat L times independently → L different k-bit sketches

Example: For 128-bit strings, might use k=10, L=20

Data Structure

```
For each item x:
    Compute g₁(x), g₂(x), ..., g_L(x)  [each is k bits]
    Store pointer to x in:
        - Slot g₁(x) of hash table 1
        - Slot g₂(x) of hash table 2
        - ...
        - Slot g_L(x) of hash table L
```

Each item appears in L places!

Query Algorithm

python

```
def query(q, r):
    candidates = set()
    for ℓ in range(1, L+1):
        slot = g_ℓ(q)
        for x in hashtable[ℓ][slot]:
            if hamming_distance(q, x) <= r:
                candidates.add(x)
            if len(checked) > 2L:
                break  # Stop if checking too many
    return candidates
```

Part 5: The Mathematical Beauty - Why It Works

Choosing k: Controlling False Positives

For a **far point** x (distance ≥ cr from query q):

$$\Pr(g_\ell(q) = g_\ell(x)) = P_2^k$$

We want this small! Set:

$$k = \log(n)/\log(1/P_2)$$

Then $P_2^k = 1/n$

Expected far collisions per table: $n \times (1/n) = 1$ **Expected far collisions over L tables:** L

By **Markov's inequality:**

$$\Pr(\text{far collisions} \geq 2L) \leq E[\text{far collisions}]/(2L) = L/(2L) = 1/2$$

So with probability ≥ 1/2, we check at most 2L far points. ✓

Choosing L: Ensuring We Find Near Points

For a **near point** x (distance ≤ r from query q):

$$\Pr(g_\ell(q) = g_\ell(x)) = P_1^k$$

Let's compute this:

$$
\begin{aligned}
P_1^k &= P_1^{(\log n / \log(1/P_2))} \\
&= \exp(\log P_1 \times \log n / \log(1/P_2)) \\
&= \exp(-\log(1/P_1) \times \log n / \log(1/P_2)) \\
&= n^{(-\rho)}
\end{aligned}
$$

where $\rho = \log(1/P_1)/\log(1/P_2)$

Key parameter ρ: Measures how much harder it is to hash similar items apart than dissimilar items together.

Probability near point collides in at least one table:

$$1 - (1 - n^{(-\rho)})^L$$

Set $L = n^\rho$, then this becomes:

$$1 - (1 - n^{(-\rho)})^{(n^\rho)} \to 1 - 1/e \approx 0.63$$

So we find near neighbors with 63% probability! (Can be improved by trying multiple times)

Part 6: From Master to Specific - Cosine Similarity

The Random Hyperplane Technique

For vectors $x, y \in \mathbb{R}^D$, cosine similarity is:

$$\cos(\theta) = (x \cdot y)/(\|x\|_2 \|y\|_2)$$

where θ is the angle between vectors.

Brilliant Idea: Use random hyperplanes!

1. Choose random unit vector u uniformly on unit sphere
2. Define $h_u(x) = \text{sign}(u \cdot x) \in \{-1, +1\}$

The Magic:

$$\Pr(h_u(x) = h_u(y)) = 1 - \theta/\pi$$

where θ is the angle between x and y!

Why? The hyperplane perpendicular to u separates x and y if u points into the wedge between them. This happens for fraction $\theta/(2\pi)$ of rotations on each side, total θ/π.

Generating Random Unit Vectors

Challenge: Sample uniformly from unit sphere in D dimensions.

Naive approaches fail:

- Uniform latitude + uniform longitude ≠ uniform on sphere (bunches at poles)

Correct approach: Use **multivariate Gaussian**!

Box-Muller Transform:

1. Generate U, V ~ Uniform[0,1]
2. Set $\Theta = 2\pi U$, $R = \sqrt{(-2\ln(1-V))}$
3. Then $X = R\cos(\Theta)$, $Y = R\sin(\Theta)$ are independent N(0,1)

Key insight: If $X_1, ..., X_D \sim N(0,1)$ independently, then:

$$u = (X_1, ..., X_D) / \|(X_1, ..., X_D)\|_2$$

is uniform on the unit sphere! This works because multivariate Gaussian is **spherically symmetric**.

Implementation

python

```python
import numpy as np

def random_hyperplane_hash(x, num_bits=64):
    """Hash vector x using random hyperplanes"""
    D = len(x)
    hash_bits = []

    for _ in range(num_bits):
        # Random unit vector
        u = np.random.randn(D)
        u = u / np.linalg.norm(u)

        # Hash bit
        hash_bits.append(1 if np.dot(u, x) >= 0 else 0)

    return tuple(hash_bits)
```

Now feed these N-bit hash codes into Hamming LSH!

Part 7: Dot Product Similarity - A Clever Reduction

Sometimes we want **dot product** $a \cdot b$ directly (not normalized).

The Trick: Transform to cosine similarity!

Given corpus vectors $x_n \in \mathbb{R}^D$ and query q:

1. **Scale:** Make $\max_n \|x_n\|_2 = 1$
2. **Transform:**

$$\hat{x}_n = (x_n, \sqrt{1 - \|x_n\|_2^2}) \in \mathbb{R}^{(D+1)}$$
$$\hat{q} = (q, 0)/\|q\|_2 \in \mathbb{R}^{(D+1)}$$

Check:

- $\|\hat{x}_n\|_2 = 1$ for all n ✓
- $\|\hat{q}\|_2 = 1$ ✓
- $\hat{x}_n \cdot \hat{q} = x_n \cdot q$ (the original dot product!) ✓

Now use cosine LSH on the augmented vectors!

Part 8: L_2 Distance LSH

The Random Projection Method

Setup: Points in $\mathbb{R}^d$, want to find near neighbors under Euclidean distance.

Idea:

1. Choose random line ℓ with random orientation
2. Partition ℓ into segments of width a
3. Project each point x onto ℓ
4. Hash value = bucket index where projection lands

Intuition:

- If points are **close** (distance $\ll a$), likely same bucket
- If points are **far** (distance $\gg a$), unlikely same bucket

Analysis Sketch

Consider points x, y:

- Let $d = \|x - y\|_2$
- Let θ = angle between line xy and random line ℓ

Case 1: $d \geq 2a$

- For collision, need projection difference $\leq a$
- Need $\theta \in [60°, 90°]$
- $\Pr(\text{collision}) \leq 1/3$

Case 2: $d \leq a/2$

- Projection difference $\leq a/2$ always
- $\Pr(\text{collision}) \geq 1/2$

This gives $(d_1 = a/2, d_2 = 2a, p_1 = 1/2, p_2 = 1/3)$-LSH.

Part 9: Jaccard Similarity and Min-Hash

The Problem

Jaccard similarity for sets A, B:

$$J(A,B) = |A \cap B|/|A \cup B|$$

Application: Document similarity when documents are sets of words/shingles.

Min-Hash: An Ingenious Solution

Setup: Sets $A, B \subseteq \{1, 2, ..., n\}$

The Hash Function:

1. Choose random permutation π: $\{1,...,n\} \to \{1,...,n\}$
2. $h_\pi(A) = \min\{\pi(a) : a \in A\}$

The Miracle:

$$Pr(\min \pi(A) = \min \pi(B)) = J(A,B)$$

Why Does This Work?

Intuitive proof:

The minimum of $\pi(A \cup B)$ must come from somewhere. What's the probability it comes from $A \cap B$?

$$Pr(\min \text{ in } A \cap B) = |A \cap B|/|A \cup B| = J(A,B)$$

But $\min(\pi(A)) = \min(\pi(B))$ exactly when the minimum is in both sets!

Formal counting proof:

Count permutations where $\min(\pi(A)) = \min(\pi(B))$:

- Must map some element from $A \cap B$ to the minimum position
- $|A \cap B|$ choices for which element
- $(|A \cup B| - 1)!$ ways to arrange the rest
- Total: $|A \cap B| \times (|A \cup B| - 1)! / (|A \cup B|!) = |A \cap B|/|A \cup B|$

Practical Implementation

Problem: n! permutations are too many! For n=1000, need $\log(1000!) \approx 8{,}000$ random bits per permutation.

Solutions:

1. **Min-wise independent families:** Smaller families that preserve the property
    - Still need e^n permutations (huge!)
2. **Approximate min-wise independence:** Allow small error ε
    - Family size $O(n^2/\varepsilon^2)$
    - Need only O(log n) random bits
    - Linear families: h(x) = ax + b mod p
3. **In practice:** Use good pseudorandom permutations

Estimation

Sample M permutations π_1, ..., π_M:

$$\hat{J}(A,B) = (\#\text{ of matches})/M$$

Variance decreases as ~1/M (like estimating coin bias).

Part 10: Shingling - From Words to Sequences

The Motivation

Word set overlap misses **ordering**:

- "Dog bites man" vs "Man bites dog" → same word set!
- Need to detect **plagiarism**, **mirrors**, **near-duplicates**

Shingling Algorithm

For text documents:

1. Convert to token sequence
2. Sliding window of size w (e.g., w=4)
3. Each window = one "shingle"

Example:

Text: "the quick brown fox jumps"
4-shingles:
- "the quick brown fox"
- "quick brown fox jumps"

For URLs:

www6.infoseek.com/cellblock16/inmates/dilbert/personal/foo.htm

Tokens: [www, infoseek, com, cellblock, inmates, dilbert, personal, foo]

Positional bigrams:
- (cellblock, inmates, 0)
- (inmates, dilbert, 1)
- (dilbert, personal, 2)
- (personal, foo, 3)

Position matters! "foo/bar" ≠ "bar/foo"

Why Shingles?

- **Robust:** Small changes → mostly same shingles
- **Fast:** Can use min-hash on shingle sets
- **Effective:** w=4 with 32-bit tokens → 128-bit shingles → 2^128 possible values

---

Part 11: All-Pairs Similarity at Web Scale

The Challenge

Given $n \approx 10^{10}$ web pages:

- For each, find 10 most similar
- Output size: 10n (manageable)
- Must run in $o(n^2)$ time!

The Algorithm

Step 1: Sketch Generation

```
For each document d:
    For each permutation π (or hash function):
        Compute sketch s = min π(shingles(d))
        Write (s, d) to file f_π
```

Step 2: Grouping

```
For each file f_π:
    Sort by sketch value s
    Documents with same s are now contiguous!

    For each group with same s:
        Output all pairs (d₁, d₂) in group to file g_π
```

Step 3: Counting

```
Merge all g_π files
Sort by document pair (d₁, d₂)
Count collisions: C[(d₁, d₂)] = # permutations where they matched

If C[(d₁, d₂)] ≥ threshold:
    Compute actual similarity
    Add to results if in top-K
```

Why This Works:

- **High recall:** If documents similar, they'll collide in many hash tables
- **Low false positives:** Dissimilar documents rarely collide in many tables
- **Efficiency:** Each document examined only L times, not n times!

Part 12: Graph Applications - Mirror Detection

Web Graph Contraction

Problem: Many pages are duplicates:

- http://www.yahoo.com vs http://dir.yahoo.com
- URL rewrites, virtual hosts, mirrors

Solution: Use LSH to find duplicates, then **contract** the graph

Algorithm:

1. Shingle all pages
2. Find similar pairs using LSH
3. Merge duplicate nodes
4. This can **cascade**: merging pages makes their parents look more similar!

Benefits:

- Save storage
- Better link analysis (PageRank, etc.)
- Avoid redundant crawling

During Crawling

Challenge: Only have URLs, not content yet!

Solution: URL shingling

- Tokenize URL
- Remove stopwords (htm, html, index, home, bin, cgi)
- Form positional bigrams
- Use min-hash to find candidate mirror hosts
- Then do detailed textual check

Part 13: Theoretical Nuances

The ρ Parameter

Recall: $\rho = \log(1/P_1)/\log(1/P_2)$

- **Small ρ:** Easier to distinguish similar from dissimilar
  - Need fewer tables (L smaller)
  - Faster queries
- **Large ρ:** Harder to distinguish
  - Need more tables
  - More storage

For Hamming with r = N/3, cr = 2N/3:

$P_1 = 2/3$, $P_2 = 1/3$
$\rho = \log(3/2)/\log(3) \approx 0.369$

The k-L Tradeoff

- **Larger k:** Fewer false positives per table, but need more tables L
- **Smaller k:** More false positives per table, but need fewer tables L

Optimal choice balances:

- Storage (L hash tables)
- Query time (check L tables, each with some false positives)

Triangle Inequality Check

Homework problem: Show 1 - J(A,B) is a metric.

Proof sketch: Let d(A,B) = 1 - J(A,B) = 1 - |A∩B|/|A∪B|

Need to show: $d(A,C) \leq d(A,B) + d(B,C)$

This is non-trivial! The Jaccard distance actually satisfies triangle inequality, making it a true metric.

Part 14: Implementation Considerations

Storage Overhead

Each of n items stored in L tables:

- Storage: O(nL)
- Each table has 2^k slots
- Might use ~nL/2^k items per slot on average

Tuning:

- If $2^k \ll n$: many collisions per bucket
- If $2^k \gg n$: most buckets empty (waste space)

Randomness Quality

Important: Hash functions must be truly random enough

Bad: Correlated hash functions → all tables similar → no amplification

Good:

- Use cryptographic PRNGs seeded differently
- For permutations: careful implementation
- Document /dev/random vs /dev/urandom trade-off

The 2L Probe Limit

Why stop checking after 2L candidates?

Answer: Expected false positives = L. By Markov, $\Pr(\geq 2L) \leq 1/2$.

If we keep going, might check many false positives. Better to:

- Stop early
- Report "not found" (but might exist)
- Or: try with different random hash functions

Part 15: Practical Wisdom

When to Use Each LSH Variant

Hamming LSH:

- Already have binary features
- Boolean data
- After applying another LSH (master LSH!)

Cosine LSH:

- Text documents (after TF-IDF)
- Neural embeddings
- When direction matters more than magnitude

Dot Product LSH:

- Recommendation systems (user·item ratings)
- When magnitude matters

L_2 LSH:

- Spatial data
- When absolute distances matter
- Computer vision features

Min-hash:

- Sets (words, shingles, user actions)
- Sparse high-dimensional data
- Classic document similarity

Hyperparameter Tuning

Start with:

- $k = \log(n)/\log(1/P_2)$
- $L = n^{\rho}$

Then tune based on:

- Recall requirements (higher L → better recall)
- Query time budget (lower L → faster)
- Storage constraints (lower L → less storage)

Combining with Exact Methods

Hybrid approach:

1. Use LSH to get candidates (high recall, some false positives)
2. Compute exact similarity for candidates
3. Return top-K by exact similarity

This is what the document describes for all-pairs!

Summary: The Big Picture

LSH is a fundamental paradigm shift:

Old paradigm: Hash to avoid collisions **New paradigm:** Hash to encourage collisions among similar items

The recipe:

1. Choose hash family matching your distance metric
2. Each hash has P_1 (near) vs P_2 (far) collision probabilities
3. Amplify by combining k hashes (AND operation → reduces both)
4. Amplify by using L tables (OR operation → increases near, keeps far low)
5. Tune k and L based on $\rho = \log(1/P_1)/\log(1/P_2)$

Why it works:

- Mathematical beauty of probability amplification
- Clever reductions (Hamming as master LSH)
- Practical algorithms that scale to billions of items

Applications:

- Web search and de-duplication
- Image similarity
- Recommendation systems
- Plagiarism detection
- Near-duplicate detection
- And many more!

This is truly one of the most elegant ideas in modern data structures and algorithms!