LSH Forest for approximate similarity search

Michael Bartoloni

November 2025

Contents

1 Introduction

1.1 Context

Similarity search is a classic problem with real world applications in many areas of computer science and data analysis: given a query object, retrieve items that are similar according to some similarity measure. Exact similarity search (for example, exact nearest neighbors under Jaccard, cosine, or L_2 distance) quickly becomes infeasible on large, high-dimensional datasets because naive indexing either requires prohibitive memory or degrades to linear scans as dimensionality grows.

To overcome this bottleneck a large body of work has developed *approximate* similarity search methods. These methods trade a small, controlled decrease in exactness for much faster query times and significantly reduced storage or indexing costs. One successful family of methods is Locality Sensitive Hashing (LSH). LSH maps objects to compact hash signatures such that similar objects collide (map to the same bucket) with higher probability than dissimilar ones. Using multiple independent hash tables or banding strategies, practitioners can obtain candidate sets that are then re-ranked by a (cheap) similarity estimator to return high-quality results quickly.

Several LSH variants exist to adapt to different similarity measures and operational constraints. In particular, this work employs *forest* structures of prefix tries built from LSH bit-strings (often called *LSH Forests*) which provide a compact index and allow flexible, level-driven candidate enumeration. Such structures compress long common prefixes and enable both fast per-table retrieval and coordinated multi-table search strategies.

1.2 Goal of the work

The aim of this project is to implement and experimentally evaluate LSH-based indexing methods following the Stanford's 2005 paper on the subject:

1. **Implement and test LSH**(K) (multiple independent LSH tables) and an **LSH Forest** implementation (prefix tries built from LSH bit-strings).

2. **Reproduce and extend the empirical tests reported in the paper** on LSH indexing. Specifically, we will reproduce the paper's experiments where possible (in particular, P2P experiments are out of scope for this project) and verify that the results hold.

2 Overview of the LSH frameworks

In this section, we recall the theoretical definitions and properties, along with some problematics, for both the LSH(K) index and the LSH forest. This will serve as a basis for the implementation of the structures.

2.1 Locality-Sensitive Hashing (LSH) and the LSH(K) index

2.1.1 Parameter legend (notation used in the paper)

n Number of points (documents) stored in the dataset (per tree indexing cost and storage scale).

ℓ Number of independent LSH Trees in the forest (the forest size). Each tree is constructed from an independent sequence of hash functions.

k In the basic LSH index, the concatenation length (number of hash digits per table). In the LSH Forest the explicit fixed k is replaced by variable-length labels (up to a maximum).

k_m Maximum allowed label length for a point in an LSH Tree (used to bound label growth). Labels are extended during insert until distinct or until k_m is reached.

c Small constant used to set an initial per-tree candidate budget; the paper often uses $c = 3$ in theoretical statements. In practice c appears in $M = c\ell$ or related budget choices.

M Total (raw) candidate budget aggregated across the forest during synchronous ascent. Typical choices include $M = c\ell$ or a dynamically-chosen M ensuring at least m distinct points.

m Number of desired (distinct) nearest neighbors to return (top-m query objective). The synchronous ascent stops when at least m distinct points have been collected (and other budget constraints).

N (When used) total dataset cardinality; in the paper n is typically used for dataset size—if you use N in your code, treat it as synonymous with n.

ε Approximation parameter for ε-approximate nearest neighbors used in theoretical definitions.

2.1.2 Definition

Locality-Sensitive Hashing (LSH) is based on families of hash functions that make similar items collide with higher probability than dissimilar ones. Formally, a family $\mathcal{H}$ of functions from a domain S to a range U is called $(r, \varepsilon, p_1, p_2)$-sensitive if for any $p, q \in S$:

$$\begin{aligned} \text{if } D(p,q) \le r \quad &\Rightarrow \quad \Pr_{h\in\mathcal{H}}[h(p) = h(q)] \ge p_1, \\ \text{if } D(p,q) > r(1+\varepsilon) \quad &\Rightarrow \quad \Pr_{h\in\mathcal{H}}[h(p) = h(q)] \le p_2, \end{aligned}$$

with $p_1 > p_2$. Intuitively, points at distance at most r collide with probability at least p_1, while sufficiently distant points collide with probability at most p_2.

2.1.3 The LSH(K)index

The basic LSH(K) index is built from repeated concatenations of independent LSH functions and multiple independent hash tables:

1. **Digit labels** Draw k hash functions $h_1, \ldots, h_k$ independently from $\mathcal{H}$. For each data point p compute the composite label (signature) $g(p) = (h_1(p), \ldots, h_k(p))$. If each h_i outputs a small alphabet (e.g., a single bit), $g(p)$ is a k-digit bucket label.
2. **Multiple tables** Repeat step (1) independently ℓ times to build ℓ separate hash tables with independent concatenated hash families $g_1, \ldots, g_\ell$. The key idea is to raise the chance that near neighbors collide in at least one table while keeping collisions with distant points unlikely.

Querying Given a query q, compute its k-digit signature under each of the ℓ tables and retrieve the points that fall into the same buckets. The retrieved set of candidates is then re-ranked using the true similarity function and the top results are returned. A typical operational choice is to examine a bounded number of candidates (for example $c\ell$ candidates total, or a fixed number per table) and then select the best m results from that candidate pool.

2.1.4 Index properties and guarantees

Theoretical guarantees. If $\mathcal{H}$ is an $(r, \varepsilon, p_1, p_2)$-sensitive family then, for appropriate choices of k and ℓ (as functions of r, ε, n), the LSH index can be shown to return an ε-approximate nearest neighbor with sublinear expected query cost and sub-quadratic storage overhead in n. These classical guarantees rely on knowing the distance scale r used when selecting k, ℓ.

Practical properties. In practice:

- ℓ (number of tables) is relatively forgiving: once ℓ is large enough (e.g., $\ell \approx 10$) adding more tables yields diminishing returns.
- k (the number of concatenated hashes per table) is critical: small k produces many false positives (distant points collide), large k reduces collisions even among true near neighbors and makes buckets too sparse (insufficient candidates). The optimal k depends strongly on the dataset, the query distribution, n, and the desired output size m.

2.1.5 Difficulties and limitations

The paper highlights several theoretical and practical problems of the basic LSH index:

1. **Parameter dependence and retuning.** The parameters k and ℓ (and the implicit radius r used to reason about collisions) depend on the dataset size n, the distance scale of interest r, and the approximation parameter ε. If the dataset or its distribution changes, the index may need re-tuning or rebuilding for good performance.
2. **Coverage vs. storage trade-off.** Guaranteeing ε-approximate results for *all* queries (with different nearest-neighbor distances) requires either many differently-tuned indices (one for each relevant scale r) or very large tables; both approaches increase storage and processing cost (storage can blow up roughly proportional to $1/\varepsilon$ in the extreme theoretical constructions). In practice, a "good" value of r is chosen and a single index is used, at the cost of tuning optimally yet another parameter

These limitations motivate the LSH Forest construction, which attempts to remove the need for manual tuning by using variable-length labels and synchronous, level-driven candidate enumeration across multiple prefix trees.

2.2 LSH Forest

2.2.1 Basic idea and motivations

The LSH Forest replaces fixed-length concatenated signatures and fixed hash-table parameters with a forest of prefix trees built from variable-length LSH labels. Each tree stores the dataset points indexed by progressively longer prefixes of their LSH bit-strings (or general small-alphabet signatures). The motivation is to remove the need to tune k for the index, since the optimal value is very dependent on the number of data points and their distribution.

2.2.2 Index construction

- **Forest of tries.** The index consists of ℓ prefix trees (a forest). Each tree is built from the same underlying MinHash/LSH family but uses an independent randomness source so that labels are independent across trees. For each data point the algorithm computes an LSH signature (a bit-string or small-alphabet string) and inserts the point into every tree by walking the tree along the prefix of the signature, until a leaf is reached. Internal nodes represent paths and do not hold any point, but their number can grow rapidly. To address this, the paper proposes a compression scheme, which is not implemented here, for reasons that will be explained when talking about the dataset.

- **Variable-length labels.** Rather than concatenating a fixed number of hash functions, each tree stores progressively longer prefixes as needed: nodes correspond to prefixes of varying length and are created only when the insertion path diverges.

It's worth noting that, in reality, we actually choose a k (larger is better), and once reached the k-th level, we aggregate all points that share the label in a single leaf. The value of k is less impactful, as a large k can only improve the queries' results, but in large datasets the trees need compression to not blow up the number of internal nodes.

2.2.3 Query strategy (level-driven candidate enumeration)

The paper proposes a two-phase query strategy that leverages the forest structure:

Descent (per-tree). For each tree, descend along the tree following the query's signature until no further full-edge match is possible [Source code 1]. The leaf reached is the *stopping node* for that tree. It represents the best match for the label on the tree.

```
descend(q, xi, s)
    args: query q at level xi on node s
    if (s is leaf):
        return (s, xi)
    else:
        y = xi + 1
        Evaluate gi(q, y)
        t = Child node from branch labeled hy(q)
        (p, z) = DESCEND(q, y,t)
        return (p, z)
```

Source code 1: Descend algorithm

Synchronous ascent across the forest. First, collect candidates from each tree's stopping-node. If more candidates are required, starting from the deepest level reached among the trees as the current level, raise the level and collect all points in the reached subtree at the current level. This process continues until enough candidates are retrieved [Source code 2].

```
sync_ascend(x[1,. . . , l], s[1, . . . , l])
    args: xi reached level for each tree
          si leaf node for each tree
    level = max(x)
    P = {}
    while (level > 0 and (|P| < cl or |distinct(P)| < m))
        for (i = 1 ; i ≥ l ; i + +):
            if (x[i] == level):
                P = P ⋃ Descendants(s[i])
                s[i] = Parent(s[i])
                x[i] = x[i] − 1
        level = level − 1
    return P
```

Source code 2: Synchronous ascend algorithm

Asynchronous ascent across each tree. Instead of checking across all trees for the number of candidates, each tree performs the ascent independently and collects its portion of the candidates. Note that the number is at least equal to the number of points requested by the query [Source code 2]. Its main usage is in a hard disk implementation, but it's still valuable to experiment on this since we're relaxing the query mechanism and then potentially underperforming with candidate similarity.

```
async_ascend(xi, si)
    Args: xi level and corresponding leaf node si
    P = {}
    while (xi > 0 and |P| < max(c, m))
        P = P ⋃ Descendants(si)
        si = Parent(si)
        xi = xi − 1
    return P
```

Source code 3: Aynchronous ascend algorithm

Main theoretical guarantee. Under the paper's strengthened sensitivity condition, and with a sufficiently large number of trees (the paper analyzes ℓ proportional to $n\,f(\varepsilon)$ in the formal proof), the LSH Forest returns ε-approximate nearest neighbors with constant probability for queries whose nearest-neighbor distances lie in the specified range. The practical interpretation is that the forest removes the need to choose a single k: the index adaptively chooses effective prefix lengths per query. The paper also notes that the theoretical constants are conservative and that, in practice, modest ℓ (e.g., $\ell \approx 10$) suffices on real datasets. In summary, the LSH Forest trades the single-parameter tuning burden of standard LSH for a slightly more complex index and query protocol that adaptively discovers an appropriate prefix length per query. The design aims to improve robustness across heterogeneous queries while keeping the candidate enumeration effort controllable via a global expansion schedule.

3 Implementation of the indexes

3.1 Hash functions

The hash families use a min-hashing for a locality-sensitive Jaccard distance measure. Chosen family H of random linear functions (linear min-wise independent) of the form h(x) = (ax + b) mod p, each term's ID in the document vector is hashed using a function h drawn from H, to obtain $h(x_i)$. Let $y = min_i(h(xi))$, that is, y is the minimum of the different IDs hashes. The value y is then hashed down to a single bit using a second hash function h' that maps each of 0, 1, 2, . . . $(p - 1)$ to 0 or 1.

3.2 Structures

In this section, the classes are briefly described, mainly describing what purpose they serve. For the actual implementations, they can be found here.

- **Minhash**: A simple helper class creates k hash functions with random sampling, and performs the lsh indexing on demand, given a document.
- **LSHTable**: A basic lsh table in which the received document gets indexed with its minhash signature and assigned to a bucket. This is the scheme for both insertion and queries
- **LSHIndex**: Leverages ℓ LSH tables. On each insertion, it puts the document in each of the tables. For candidate searching, it retrieves $c\ell$ candidates from the tables' buckets (c for each table). If the number of candidates is more than c for a table, it randomly takes c candidates.
- **Node**: Represents three possible states for tree nodes. Root has no parent, internal nodes have no points but at least one child, leaves have no children and store points.
- **LSHTree**: Inserts points by descending the prefix tree bit by bit following the point's label until it finds a leaf or the max depth k_m is reached. At the stopping node, there are 3 possible cases. When the reached leaf is at max depth, the point is stored in the leaf among the other points that share the label (if there are any). If the leaf is not at max depth, the labels of both the leaf and the inserted point are checked until divergence. If they diverge at some point, the two leaves are put as children of the last common internal point, otherwise they are aggregated in a single leaf, as they are indistinguishable. Queries are described in the previous section, and can be both synchronous and asynchronous.
- **LSHForest**: Leverages ℓ LSH trees. Can choose to query its trees in synchronous or asynchronous mode.

4 Experiments and results

In this sections the setup for both the dataset and experiments are described, along with the results of the runs.

4.1 Setup

4.1.1 Dataset and preprocessing

The paper uses a subset of the Reuters corpus, consisting of ∼100K documents. This project uses the Reuters-21578 from the NLTK python library, consisting of ∼10K documents. Since the dataset is 10 times smaller than the one used in the original experiments, the results are not expected to fully adhere to the paper's results. As mentioned when talking about compression, this work does not implement it, and the main reason is that the dataset is small, so the maximum number of leaves does not blow up, and actually the results will show that the number of points per leaf drops dramatically when the label lengths grow. Following the original setup stop-word elimination and stemming are performed on the documents. Then, LogTF is calculated on each document's term, leaving a weighted version of their bag-of-words representation, effectively representing points in a vectorial space.

4.1.2 Similarity metric

The project leverages the weighted Jaccard similarity, that is defined between two non-negative vectors

$$J_w(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{d} \min(x_i,\, y_i)}{\sum_{i=1}^{d} \max(x_i,\, y_i)}.$$

for $\mathbf{x}, \mathbf{y} \in R^d_{\geq 0}$.

4.1.3 Algorithms

The tests require the algorithms to perform top-m queries: given q, a query document chosen at random from the dataset, find m points that are similar to q. In particular, the methods tested are:

- **LSH(K)**
- **LSH Forest sync**
- **LSH Forest async**
- **Random** - This just selects randomly for candidate points

All experiments are averaged on 100 runs for better stability of the plots. The main metric for all the experiments is the average similarity, But there are other interesting parameters to look at in the algorithms.

4.2 Tuning the LSH index

The first experiment aims to find the optimal value of the LSH(K) algorithm, and involves answering a top-5 query for the index on an increasing value of k. Two variants of the index are evaluated, as well as a random baseline:

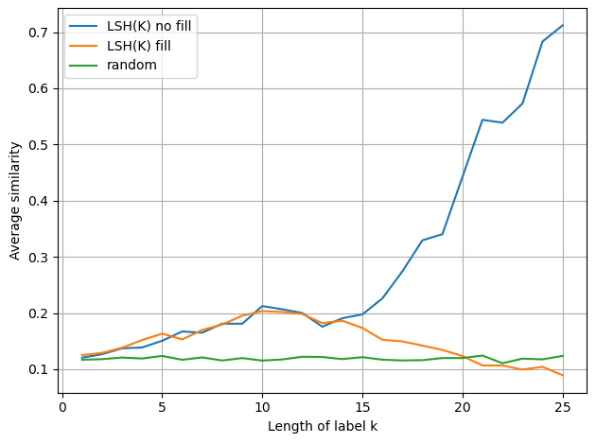

Figure 1: Average similarity of top-5 queries for the LSH(K) index with 25 candidates

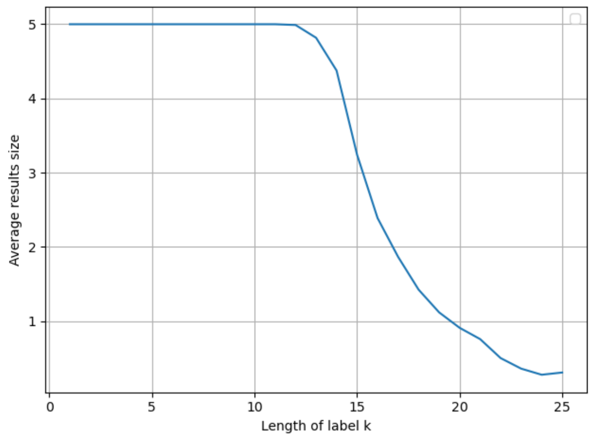

Figure 2: Average size of the returned results of top-5 queries for the LSH(K) no fill index at different label lengths

- **Fill**. This version adds random points to the candidates if less than m distinct points are retrieved from the queries results.
- **No fill**. This version returns only the top-m distinct points from the candidate set, even when the number of candidates retrieved is less than m.

Intuitively, fill and no fill will both increase the average similarity initially, as a short label leads to fewer buckets, but at some point they are going to diverge, one dropping performance towards random, the other performing better with higher k. The optimal k is the best-performing label length that preserves equal performance. For this experiment, $\ell = 5$, and c is set to 5. This means that each table in the index is retrieves 5 candidates, for a total of 25 candidates per query. Figures 1, 2 show the results of the experiment. As expected, at k≈13 the two LSH strategies diverge, but the best performance is around 10/11. Just to be more conservative, we set the k value to 10 the dataset. It's worth noting that the dataset used in this project has a much lower average similarity between documents, sitting at just above 0.1. This is due to the fact that the number of very similar documents is less, as a good similarity score is reached only with a long label, so much that on some queries nothing other than the queried document is returned.

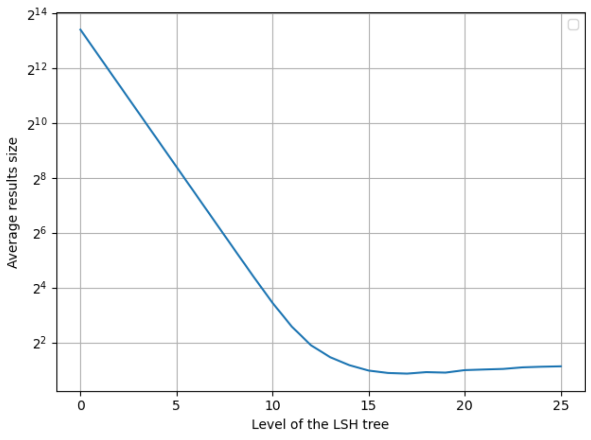

Figure 3: Distribution of the documents across the LSH Tree levels

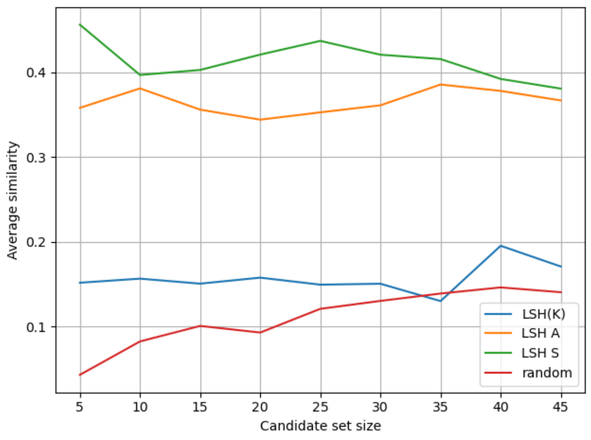

Figure 4: Average similarity measures for LSH Forest and basic LSH for top-m queries

4.3 LSH Forest

4.3.1 Point distribution in the tree

This experiment measures how many points are at each level of the tree or below at the given depth. Figure 3 shows that the trees are quite balanced, as the drop in the average number of documents is exponential as the depth increases.

4.3.2 Tuned LSH(K) vs LSH Forest

This experiment compares the optimal LSH index with LSH forest. In this setting $\ell = 5$, $k = 11$, $M = 2m$. The experiment variates on both desired results and candidate size, calculating the average similarity of the queries results. Figures 4, 5 show the results of the experiment. Varying the candidate set size does not affect performance dramatically for LSH Forest, as the top-m results are usually the first retrieved. Still, as we had anticipated, the relaxed asynchronous search loses some performance over the synchronous version, as it takes a more local approach on the candidate retrieval, without leveraging all trees at once, but rather collecting the best candidates for each tree. LSH(K) performs worse, and by a great margin, and increasing the candidate size does not affect them at all. Random candidate search, instead, shows some improvement. This is to be expected,

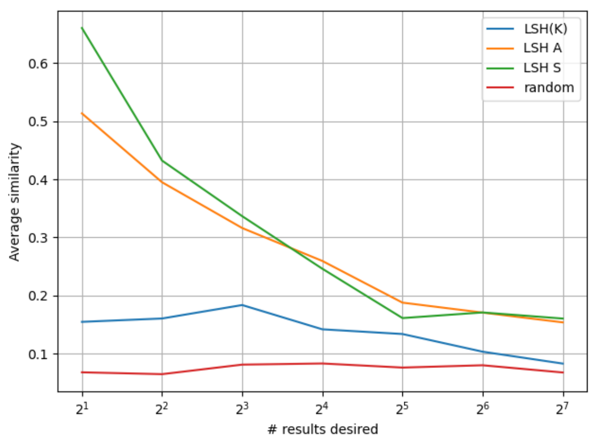

Figure 5: Average similarity of top-5 queries with varying candidate set sizes

since with more candidates the probability of choosing good ones increases, ultimately closing the gap with LSH(K). Varying the results desired, instead, produces different outcomes. Performance drops for all the algorithms, as expected from an increasing number of desired results from a dataset which does not have a lot of similar points per query.

5 Conclusions

The results, even if very different from the paper's results, still verify the theoretical properties and benefits of the algorithms:

- LSH(K) suffers from generality on different types of queries, and lower average similarity per result, with the need of tuning for a specific dataset.
- LSH Forest does not need tuning, and outperforms the basic index in all experiments, both in synchronous and asynchronous version, offering a good tradeoff between complexity and efficiency. Also, the synchronous version confirms itself to be a valid alternative for parallel execution in particularly high computation times settings. The data may be the weakest point in this analysis, as points in the dataset have low average similarity, meaning that it's difficult to find good candidates for the LSH(K) index, even for good values of k. However, this is yet another proof of LSH Forest's adaptability: with Forest, we always return the most relevant results wrt the sampled hash family, as shown by the good results for low m queries.