Locality-Sensitive Hashing (LSH)

Diogo Pinto - 110341

University of Aveiro
Mining Large Scale Datasets (2024/2025)

May, 2025

1 A brief explanation of what LSH is and how it works

Locality-Sensitive Hashing (LSH) is a technique designed to efficiently identify similar items in large datasets by reducing the dimensionality of the data while preserving the notion of similarity. Traditional similarity search algorithms often require computing the distance or similarity between all pairs of items, which becomes computationally expensive at scale. LSH addresses this challenge by using a family of hash functions with the key property that similar inputs are more likely to be hashed to the same value. [3]

The core concept behind LSH is probabilistic hashing. Instead of assigning completely random or uniformly distributed hash values, LSH uses hash functions that preserve proximity. That is, the probability that two items are mapped to the same "bucket" is higher if the items are similar according to a chosen similarity measure (e.g., **Jaccard, Cosine, or Euclidean distance**).

To implement LSH, multiple hash functions are used to generate compact signatures or fingerprints for each item. These signatures are then grouped into bands (in the case of **MinHashing**) or compared based on their binary forms (in the case of random projections). Only items that collide in at least one band or under one of the hash functions are considered "candidate pairs" and are subjected to more precise similarity computations. [2]

This dramatically reduces the number of comparisons needed. For example, instead of checking all pairs of n^2 items in a data set, LSH narrows the search to a smaller subset of candidates with a high probability of similarity, resulting in a more scalable and efficient process for similarity search. This conceptual framework makes LSH particularly attractive in applications such as recommendation systems, near-duplicate detection, plagiarism detection, and large-scale machine learning pipelines.

2 How LSH Can Be Used to Speed Up Similarity Computation Between Users/Items

In large-scale recommender systems, identifying similar users or items is a fundamental step. However, as the number of users and items grows into millions, computing pairwise similarities becomes prohibitively expensive. LSH offers an efficient solution by significantly reducing the number of comparisons needed.

Rather than exhaustively comparing every user or item with all others, LSH hashes users/items into buckets such that similar entities are likely to end up in the same bucket. Then, similarity computations are performed only among items within the same bucket.

For example, in a user-based collaborative filtering setup, each user might be represented as a high-dimensional binary vector indicating their interactions. Using a technique like **MinHashing** (relevant for **Jaccard similarity**), these vectors are compressed into compact signatures. LSH is then applied to group similar users, allowing the system to focus only on comparing users within the same buckets.

Similarly, when random projection-based LSH is used for **cosine similarity**, items are hashed into buckets based on their angular proximity. This is particularly useful in modern systems where users and items are represented by dense vectors learned from models. [1]

3 Specific techniques relevant for user/item-based filtering.

LSH can be adapted to different types of similarity measures. The two most common approaches are:

3.1 MinHashing + Jaccard Similarity

MinHashing is an LSH technique tailored for set-based data, particularly when the similarity between entities is measured using the Jaccard index. The Jaccard similarity between two sets A and B is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Computing this directly for all user pairs in a large system is computationally expensive. MinHashing offers a way to approximate the Jaccard similarity by generating compact signatures for each set.

The idea is to apply multiple independent hash functions to the elements of a set and record the minimum hash value under each function. These values form a fixed-length signature vector for the set. The probability that two sets have the same hash value for a given function is equal to their Jaccard similarity. Therefore, the fraction of matching values across all hash functions serves as an estimator of the true similarity.

Locality-Sensitive Hashing step: These signatures are then partitioned into bands. Each band is hashed again and if two users share the same hash in at least one band, they are considered candidate pairs.

Use case:

- Each user is represented as a set of item IDs they interacted with (e.g., purchased, rated, viewed).
- MinHash transforms each user set into a compact signature (e.g., 100 values per user instead of thousands).
- LSH groups users whose signatures match in at least one band, drastically reducing the number of comparisons.

This approach is particularly effective when data is sparse and binary (e.g., implicit feedback systems), making Jaccard similarity a natural fit. [1]

3.2 Random Projection + Cosine Similarity

Random projection-based LSH is designed for dense vector spaces, where cosine similarity is a common metric. Cosine similarity between two vectors $\vec{u}$ and $\vec{v}$ is given by:

$$\cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|}$$

To approximate cosine similarity efficiently, LSH uses a technique based on random hyperplanes. Each hash function corresponds to a random vector $\vec{r}$, and the sign of the dot product $\vec{u} \cdot \vec{r}$ determines whether the hash bit is 0 or 1. If two vectors are similar (i.e., small angle between them), they are more likely to fall on the same side of many hyperplanes, resulting in similar binary signatures.

Locality-Sensitive Hashing step: These binary signatures are stored in hash tables. When querying, only items with matching or similar bit signatures are considered candidates.

Use case:

- Each item (e.g., product, song, article) is represented as a dense embedding vector (from models like Word2Vec, BERT, or collaborative filtering).
- Random projection hashes each item into a binary signature that preserves angular proximity.
- Cosine-based LSH retrieves items with similar embeddings efficiently, allowing scalable nearest-neighbor search in high-dimensional spaces.

This method is especially useful in systems where item or user representations are learned through deep learning or matrix factorization and stored as high-dimensional float vectors. [1]

3.3 Other Variants

- **Euclidean LSH:** Useful when absolute distance matters.
- **Hamming LSH:** For binary feature vectors.

4 An Example of LSH in a Recommender Context

A concrete example of **Locality-Sensitive Hashing** applied in a recommender system is the project *"Conference Paper Recommendation System using LSH"* [5]. The goal of this system is to recommend similar academic papers based on their textual content using efficient approximate similarity techniques.

Data representation: Each document is converted into a set of *k-shingles*, which are contiguous substrings of length k extracted from the text (e.g., from the title or abstract). This turns the recommendation task into a set similarity problem, making the **Jaccard similarity** a natural choice.

System workflow:

1. **Shingling:** The text of each paper is processed to generate a set of k-shingles.
2. **MinHashing:** A set of hash functions is applied to the shingles of each document to produce a compact signature (a vector of integers) that approximates the Jaccard similarity between sets.

3. **LSH with Banding:** The MinHash signatures are divided into *bands* and hashed into *buckets*. Documents that land in the same bucket in at least one band become candidate pairs.

4. **Filtering and Recommendation:** Only candidate pairs are then compared using the true Jaccard similarity, and the most similar documents are recommended.

Advantages:

- **Computational efficiency:** Avoids exhaustive pairwise comparisons between all documents.
- **Scalability:** Suitable for large documents.
- **Simplicity:** Relies on hashing and set operations, without requiring complex embeddings.

Limitations:

- The choice of parameters (shingle size, number of bands, number of hash functions) significantly impacts performance.
- It does not capture deeper semantic relationships between documents (unlike embedding-based models).

This case study demonstrates how LSH, when combined with MinHashing, can be effectively used to build content-based recommendation systems for large-scale textual data.

5 Trade-offs in Using LSH for Recommender Systems

While **Locality-Sensitive Hashing** (LSH) offers clear advantages for scaling similarity-based recommendations, it also comes with trade-offs that must be considered:

Advantages:

- **Efficiency:** LSH allows for approximate nearest neighbor search in sub-linear time, which is critical for systems operating at web scale.
- **Scalability:** It handles high-dimensional data and massive datasets efficiently, making it suitable for real-time recommendation in production systems.
- **Simplicity:** Conceptually simpler to implement than more complex deep learning-based retrieval systems.

Limitations:

- **Approximation vs Accuracy:** LSH prioritizes speed over precision. It might miss some true nearest neighbors or include false positives, potentially affecting recommendation quality.

- **Parameter Tuning:** Performance depends heavily on the number of hash functions, hash tables, and other hyperparameters, tuning them is not always straightforward.
- **Static Index:** Traditional LSH indexing is not as dynamic, adding or removing items requires rehashing or rebuilding parts of the index, making it less flexible for rapidly evolving systems.
- **Cold Start:** LSH is less helpful for cold-start problems (e.g., new users or items with little interaction history), where hybrid or model-based approaches might be better.

In summary, LSH is ideal for approximate, fast, and scalable similarity search but may need to be combined with other techniques for high-accuracy, real-time personalization.

6 Most Interesting or Useful Aspect for Large-Scale Systems

One of the most compelling aspects of LSH for large-scale recommender systems is its ability to reduce computational complexity without sacrificing much performance. This makes it particularly attractive for real-time applications like search autocomplete, "you might also like" modules, or content deduplication.

What stands out is the elegance of the hash-based approach: instead of calculating distances across millions of vectors, LSH probabilistically narrows the search space to only the most promising candidates almost like a pre-filtering stage. This is especially powerful when paired with learned representations (e.g., embeddings from deep learning models) and a fast indexing tool like FAISS.

Moreover, LSH's compatibility with parallel and distributed systems makes it well-suited for use in cloud-scale architectures, where minimizing latency is as important as maximizing relevance.

References

[1] Rajaraman, A., & Ullman, J. D. (2012). Mining of Massive Datasets. http://infolab.stanford.edu/~ullman/mmds/ch3n.pdf

[2] Pinecone. Locality-Sensitive Hashing with FAISS. https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/

[3] Joshi, S. (2022). Understanding Locality Sensitive Hashing (LSH). https://medium.com/@sarthakjoshi_9398

[4] Y. Wang, et al. (2022). Locality-sensitive hashing for fast similarity search: A review. https://www.sciencedirect.com/science/article/abs/pii/S0020025522003644

[5] Parineeta Dey (2021). Conference Paper Recommendation System using LSH. https://github.com/parineeta16/Conference-Paper-Recommendation-System-using-LSH