COP 6731 Advance Database Systems

Project Presentation Report

Anmol Sureshkumar Panchal; UID: 4446829

Topic: Locality Sensitive Hashing (LSH)

Objective:

To study LSH and algorithms to improve its performance to process large datasets of information retrieval, data mining, image processing and so on.

Introduction:

A similar search problem like KNN queries are often aimed at collection of objects (e.g., documents, images) that are characterized by a collection of relevant features and represented as points in a high dimension attribute space. Given that those queries are in the form of points in the space, we are required to find the nearest (most similar) objects to the queries. The interesting and well explored case is d-dimensional Euclidean space. This problem is of major importance to a variety of applications like data compression, databases and data mining, information retrieval, image and video databases, data analysis. In these areas, the problem that usually arises is arrival of vast amount of data and of high dimensions, which we also call "curse of dimensionality". When given a collection of points and a distance function such as hamming distance function or Euclidean distance function between them, the nearest search is one of the most important forms in similarity search. Nearest neighbor search has been applied in image processing, local match feature etc.

The key insight behind the technique of LSH is that the hash functions should map adjacent points to the same buckets with higher probability, but map points far from each other to the same buckets with lower probability. The basic idea of LSH is to hash the points from the database to ensure that the probability of collision is much higher for those points that are close to each other than for those that are far apart. LSH is an indexing technique that makes it possible to search efficiently for nearest neighbors amongst large collections of items, where each item is represented by a vector of some fixed dimension. The algorithm is approximate but offers probabilistic guarantees i.e. with the right parameter settings the results will rarely differ from doing a brute force search over your whole collection. The search time will certainly be different though: LSH is useful because the complexity of lookups becomes sublinear in the size of the collection. Now later of this proposal we will see the techniques proposed in the three papers selected to present and will analyze and conduct the comparisons to see which method is more suitable for LSH implementation.

Method 1: An Improved Algorithm for Locality-Sensitive Hashing

An important technique to solve NN problem is locality sensitive hashing or LSH. Its idea is to connect the possibility of collision in the same bucket between points p and point q with their distance. That is, if some point p and point q are close to each other, then there is greater probability they collision; but instead if the greater distance between p and q, the more smaller probability they collision.

An LSH family need to satisfy the two conditions as follows:

• If d (q, v) <= d1, then the probability h(q) = h(v) is at least p1.

• If d (q, v) > d2, then the probability h(q) = h(v) is at most p2.

Where d (q, v) denotes the distance between q and v, d1 < d2, h(q) and h(v) represents the hash value of q and v. If hash functions satisfy the two conditions above, we can call it (d1, d2, p1, p2) sensitive. Then we can get one or more hash tables when hashing for data through one or more hash functions of (d1, d2, p1, p2) sensitive, this process is known as Locality-Sensitive hashing.

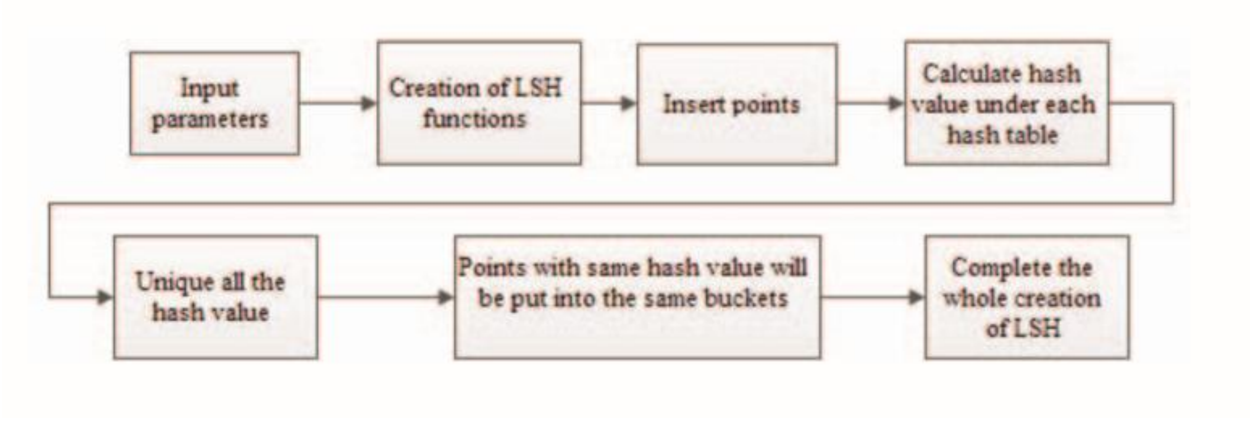

Input parameters
Creation of LSH functions
Insert points
Calculate hash value under each hash table
Unique all the hash value
Points with same hash value will be put into the same buckets
Complete the whole creation of LSH

Fig 1: Creation model of Locality-Sensitive hashing.

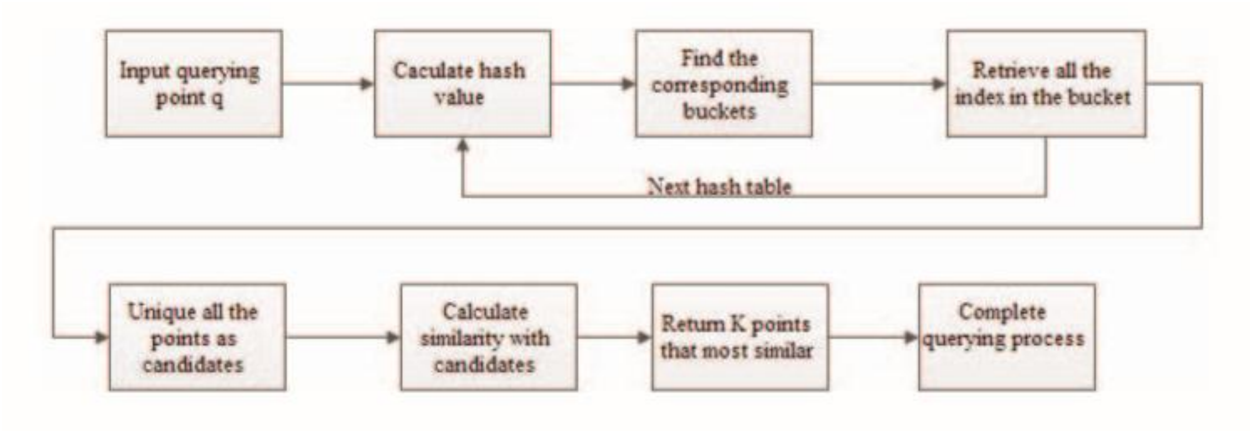
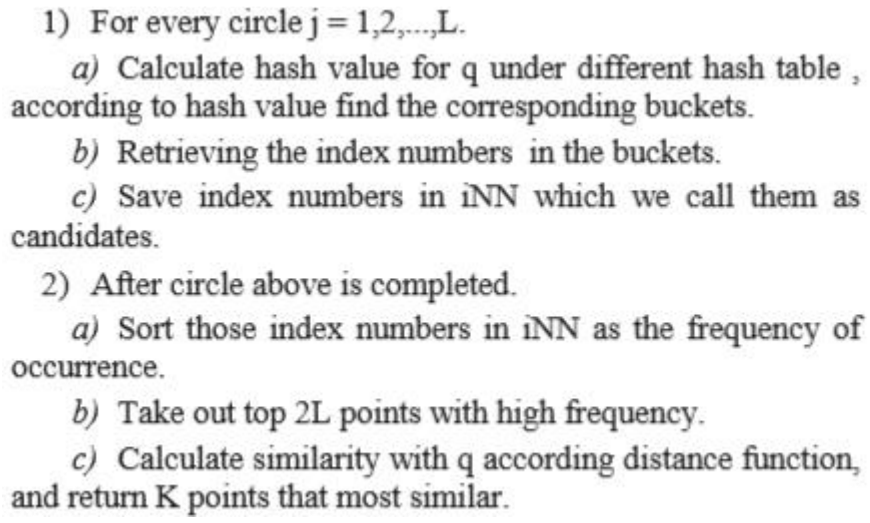

Input querying point q
Caculate hash value
Find the corresponding buckets
Retrieve all the index in the bucket
Next hash table
Unique all the points as candidates
Calculate similarity with candidates
Return K points that most similar
Complete querying process

Fig 2: Querying model of Locality-Sensitive hashing.

After LSH has been created which uses earlier method, we can execute the query search. The step of the proposed technique to improved LSH algorithm implements query search when input a querying points q as shown below:

Fig 3: Algorithm to improve LSH performance.



1) For every circle j = 1,2,...,L.

a) Calculate hash value for q under different hash table , according to hash value find the corresponding buckets.

b) Retrieving the index numbers in the buckets.

c) Save index numbers in iNN which we call them as candidates.

2) After circle above is completed.

a) Sort those index numbers in iNN as the frequency of occurrence.

b) Take out top 2L points with high frequency.

c) Calculate similarity with q according distance function, and return K points that most similar.

Method 2: Entropy based Locality Sensitive Hashing

To improve LSH, we have seen that the most methods are query-directed. Multiprobe LSH expands the number of query buckets to increase candidates. Query-adaptive LSH chooses buckets of efficient hash functions by calculating hash values of query points. When building the index if we can design a set of new hash functions to optimize the mapped values, the performance will be better. In the theory of LSH based on p-stable distributions, different mapped regions always have different number of points after the projection of h-functions. When it extends to the k-dimensional vector the difference will be greater. There may be more points near the origin, resulting in uneven distribution. The original points are mapped to k-dimensional vectors by k h-functions, which will be concentrated near the zero vector. So, the distribution of k-dimensional vectors is uneven by far and the number of points in each bucket is different extremely. When querying, if the mapped vector of the query point is near the zero vector, many candidates will be compared, leading to much time consumed on candidate comparison. If the mapped vector is far from the zero vector, fewer or no candidates will lead to a lower accuracy. So this paper proposes LSH method based on Entropy to overcome this drawback. They used an appropriate mapping method to make k dimensional mapped vectors uniform and the distribution entropy maximum, the number of candidates for each query will be nearly the same. It could be a good solution to these problems, reducing search time and increasing search accuracy. Figure 3 below gives a mapped result of part of the audio dataset.

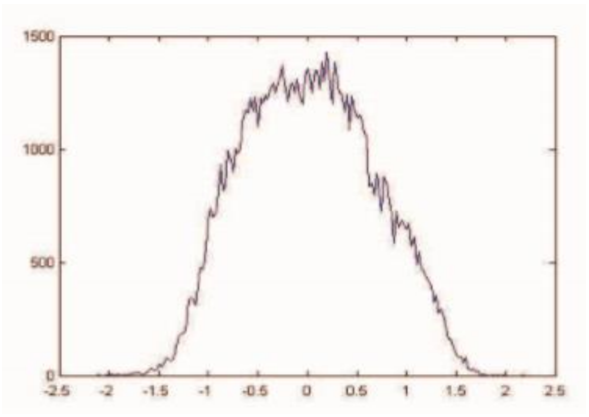


Fig 3

This paper presents a set of new hash functions and explains it in Euclidean space. For a high-dimensional space Sd and a data set of N points, a is a random vector with the distribution Nd (0,E) and v is a point chosen randomly from Sd, the initial mapped value can be calculated by the following formula: **H'a(v)=a·v**. The mapped set is {h'1, h'2,…,h'N}. Then they introduced another parameter r, which represents the number of quantization. The mapped value will be quantified to (0, 1, …., r-1).

The process is as follows:

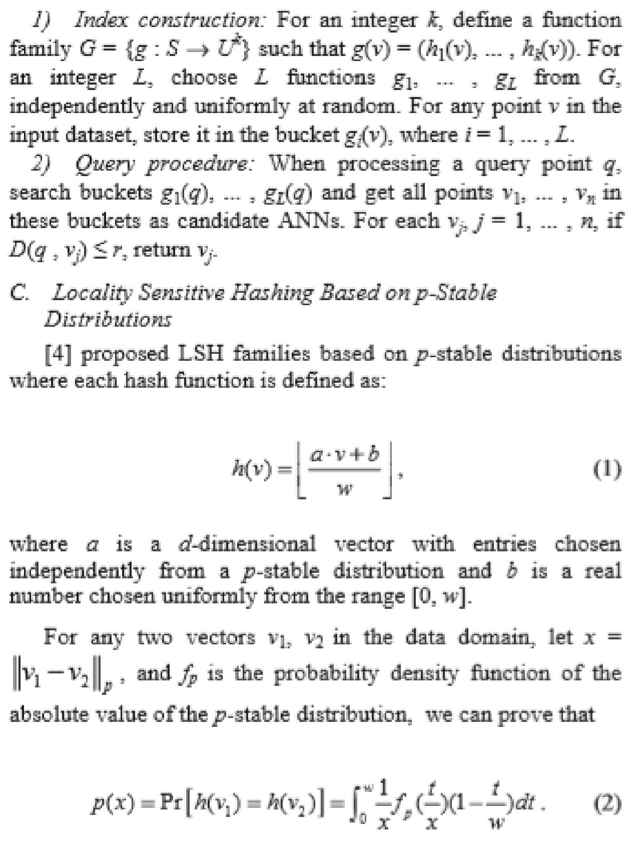
- First, sort initial mapped values and record 1/r, 2/r,…, (r-1)/r cut-off points as q1,q2,…,qr-1. The number of points will be the same in each mapped region segmented by cutoff points.
- Then make use of the following formula to obtain the quantization value:

$$
Ha(v)=\begin{cases}
0 & if \quad H'a(v)\leq q_1 \\
1 & if \quad H'a(v)>q_1 \quad and \quad H'a(v)\leq q_2 \\
\vdots & \\
r\text{-}2 & if \quad H'a(v)>q_{r-2} \quad and \quad H'a(v)\leq q_{r-1} \\
r\text{-}1 & if \quad H'a(v)>q_{r-1}
\end{cases}
$$

- Then randomly select k functions from the family H to form g function. The mapped k-dimensional vector is the final hash value, regarded as a bucket. Points are mapped to r regions evenly by h-functions. After the projection of a g-function composed of k h-functions, the distribution of points is almost uniform.
- This paper introduces distribution entropy to evaluate hash functions. The formula is as follows: **E(g)=-∑p(N(i))*log(p(N(i)))**
- Where, N(i) is the number in the bucket i, and p(N(i)) is the collision probability of the bucket i (p(N(i))= N(i)/N). The entropy of mapped values can be calculated for a g-function by the formula. For a constant number of buckets, uniform collision can result in larger entropy. So, the uniformization of collision probability is to maximize the distribution entropy. In the original LSH, we should create L hash tables. We also choose k*L h-functions, L g-functions, to build L hash tables.
- To increase the accuracy or for fine tuning you can optimize the parameters K, L and r. The optimization of k, L and r is defined as follows:
  Obj: Min(Tg +Tc) St:
  - **1) Tg=kL*tg**
  - **2) Tc=(L*N/rk)*tc**
  - **3) L=˚(log¿)/log(1-prk)˛**
- Where, tg is the time of computing an h-function and tc is the time of computing distance to a candidate. In the definition tg and tc are relevant to the dimension of the data and the computer. r can change from 2 to 6, k is set from 2 to 60, L is determined by k, pr and ¿. We can adjust r and k to minimize (Tg+Tc) and obtain the optimal r, k and L.

Method 3: Frequency based Locality Sensitive Hashing

Frequency Based Locality Sensitive Hashing scheme is like generic LSH (you can see the generic LSH from the paper for more details) scheme except that it uses a single h(v) as hash function and sets a frequency threshold m to select those points which collide with query point more than m times as candidate ANNs.



1) *Index construction:* For an integer k, define a function family $G = \{g : S \to U^k\}$ such that $g(v) = (h_1(v), \ldots, h_k(v))$. For an integer L, choose L functions $g_1, \ldots, g_L$ from G, independently and uniformly at random. For any point v in the input dataset, store it in the bucket $g_i(v)$, where $i = 1, \ldots, L$.

2) *Query procedure:* When processing a query point q, search buckets $g_1(q), \ldots, g_L(q)$ and get all points $v_1, \ldots, v_n$ in these buckets as candidate ANNs. For each v_j, $j = 1, \ldots, n$, if $D(q, v_j) \le r$, return v_j.

C. *Locality Sensitive Hashing Based on p-Stable Distributions*

[4] proposed LSH families based on p-stable distributions where each hash function is defined as:

$$h(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor, \qquad (1)$$

where a is a d-dimensional vector with entries chosen independently from a p-stable distribution and b is a real number chosen uniformly from the range $[0, w]$.

For any two vectors v_1, v_2 in the data domain, let $x = \left\| v_1 - v_2 \right\|_p$, and f_p is the probability density function of the absolute value of the p-stable distribution, we can prove that

$$p(x) = \Pr\left[h(v_1) = h(v_2)\right] = \int_0^w \frac{1}{x} f_p\left(\frac{t}{x}\right)\left(1 - \frac{t}{w}\right) dt. \qquad (2)$$

Generic LSH works as given: For a fixed parameter w, the probability of collision p(x) decreases monotonically with x, so h(v) belongs to LSH family. Exact Euclidean LSH (E2LSH) is a popular implementation of LSH based on p-stable distributions using generic LSH scheme which can be used in Euclidean space, and they just used it in their paper to compare with their method presented in this paper.

Frequency based LSH works as follows:

1) Index construction: For an integer L (L > 1), choose L functions h1, ... , hL from H (based on p-stable distributions), independently and uniformly at random. For any point v in the input dataset, store it in the bucket hi(v), where i = 1, ... , L.

2) Query procedure: For a query point q, search buckets h1(q), ... , hL(q) and get all points v1, ... , vn which appear no less than m times in these buckets as candidate ANNs. For each vj, j = 1, ... , n, if D(q , vj) ” r, return vj.

For their algorithm to work its obvious to see that,

$$t(x) = f(p(x)) = \sum_{i=m}^{L} C_L^i \, p(x)^i (1 - p(x))^{L-i} \,,$$

$$f(u) = \sum_{i=m}^{L} C_L^i u^i (1-u)^{L-i} \,.$$

$$f'(u) = L C_{L-1}^{m-1} u^{m-1} (1-u)^{L-m} > 0 \,.$$

In first equation from the picture above, where x denotes the distance between query point q and any point v in the data domain, t(x) denotes the probability of that v is the candidate ANN of q, and p(x) denotes the probability of that q collides with v in a hash table. Let u = p(x), in Equation 2 from the picture. Then from the third equation from the picture they said that It is easy to see f(u) is a monotonically increasing function.

Method 4: My technique to implement LSH using Jaccard similarity.

Similarity hashing arises from a contrasting idea to traditional hashing. The intuition for similarity hashing is to generate hashes or signatures which preserve the similarity and relationships between two items. Rather than a random and uniform distribution, the aim of similarity hashing is to cluster like items together. Two documents which contain very similar content should result in very similar signatures when passed through a similarity hashing system. Similar content leads to similar hashes. Locality sensitive hashing (LSH) is a formal name for such a system, and a broad academic topic addressing related concerns.

Let’s walk through an example of one kind of similarity hashing. This example should help explore the concept. It will also begin to show that LSH is not a singular hash function but an assembly of many techniques involving the generation and comparison of document signatures.

First, imagine two sentences:

Sentence A: "The dog is brown"

Sentence B: "The dog is happy"

Then, imagine representing those two sentences in a matrix like so:

words:	the	dog	is	brown	happy	sunshine
A:	1	1	1	1	0	0
B:	1	1	1	0	1	0

For each sentence and each column, a 1 indicates a word was present and a 0 indicates the word was not present. Note that the word "sunshine" is added to the matrix, and since neither sentence contain that word, both have a 0 in that column. This is added to the example to help illustrate how those are dropped later as a part of the similarity measurement.

How similar are the two sentences?

How can the similarity be scored?

This can be formulated as a set problem, and the two ordered sets can be compared:

Sentence A = [1, 1, 1, 1, 0, 0]

Sentence B = [1, 1, 1, 0, 1, 0]

Each sentence has a bit in the first 3 columns. And 5 columns have a bit in at least one of the sets. So we'll give these two sentences a similarity score of 3/5 or 0.6. The formal name for this is Jaccard Similarity.

Walking through these steps, we have hashed two sentences and then compared their signatures to generate a similarity score.

Let's move closer to a real world scenario, where it's not just two sentences, but a collection of many documents. Envision a matrix of all the terms and documents in the collection.

words:	ace	cat	bag	brown	happy	water	...
doc 1:	1	1	1	1	0	0	...
doc 2:	1	0	0	1	0	0	...
doc 3:	1	0	1	0	0	0	...
doc 4:	0	0	0	0	0	0	...
doc 5:	1	1	1	1	0	0	...
doc n...							

Such a matrix would quickly get very large and it would also tend to be very sparse, meaning it would contain lots of zeros. When dealing with very large scale collections, holding such large matrices in memory and performing comparison calculations becomes a difficult problem.

So in this method we implement as follows:

- Here we used LSH to find near duplicates for documents. This is done by splitting the Minimum Hash key matrix into bands and then hashing each band. Any documents where a band is hashed to the same bucket are considered near duplicates.
- Creates a new LSH object.
- Minimum Hash matrix where rows are documents, columns are the hash functions.
- Array of document names are stored in collection.
- Number of bands is used to split minimum Hash matrix.
- Then it Computes a list of near duplicate documents. Near duplicate documents are documents where at least one of the bands hash to the same bucket.
- We use a Container object to store the band number and hash value. In LSH, the MinHash matrix is split into B bands: 1,2,...,B and each band is R-rows. The R-rows in a band are hashed to get a hash value for that band. This object stores the band Number and the hash value for that band.

- Then Return the hash code of container for mapping those hash keys and bands.
- Then compare equality against other Pair containers. Return True if band and hash values are equal, otherwise false.
- Generates minimum hash for a collection of documents.
- The permutations are represented by randomized hash functions: **ax + b % p**.
- p is a prime such that p >= n where n is the number of terms in the collection.
- a and b are chosen uniformly at random from {1,2,...,p-1}.
- Minimum Hash matrix generated will by M * N. M = number documents and N = number of permutations.
- Each element in Minimum Hash matrix will be the MinHash value of the document.
- MinHash matrix has documents as rows and permutations as columns.
- There is **minimal preprocessing.**
- Calculates the exact jaccard similarity between two documents.
- Compare: first document, second document……. N documents
- Return Jaccard similarity of these documents.
- Computes the MinHash signature for all documents in the collection.
- Computes the approximate jaccard simularity by using the MinHash signatures.
- Gives the total number of unique terms in the collection of documents after basic **preprocessing.**
- User must specify <folder> with collection of documents, <number of permutations> for use with MinHash matrix, and the <error parameter>.
- This will print out the number of document pairs where the approximate jaccard similarity and exact jaccard similarity differ by more than the <error parameter>.
- Calculates the approximate jaccard similarity and exact jaccard similarity for all pairs of documents and then counts the number of times the difference between the exact and approximate jaccard similarities differs by more than the user specified error parameter.
- Prints time it takes to calculate exact jaccard similarity and time it takes to calculate approximate jaccard similarity.
- Calculates the number of false positives that were hashed together into the same bucket in LSH.
- User specifies <folder> with collection of documents, <number of permutations> for MinHash Matrix, <number of bands> for LSH, <similarity threshold> and <file> to find near duplicates for.
- Afterwards this will print out the number of false positives; documents that were hashed together in the same bucket for LSH.

Conclusion:

In paper 1 they discussed mainly comparing and improving the method with LSH based on hamming distance and LSH based on Euclidean distance (E2LSH). Their experiments were able to show that the improved algorithm has a higher accuracy rate and recall rate than LSH which based on hamming distance. In second paper, to get better performance than that of first paper by using multi-probe or query-adaptive strategy for their proposed new hash functions. The paper also proposed a method to automatically optimize parameters, but the estimation of parameter pr may be inaccurate sometimes, leading to parameter tuning failure. In the third paper which is based on frequency based LSH (FBLSH) which needs fewer hash tables and comparisons than E2LSH with the same accuracy. Besides, FBLSH needs only about 4nL bytes memory while E2LSH needs about 12nL bytes, where n is the size of dataset. That's why FBLSH can reduce the space cost effectively. And also FBLSH consumes much less query time than E2LSH. For large datasets, the frequency counting time becomes longer and FBLSH has similar query time with E2LSH. So far we can conclude that the third paper's proposed approach is more accurate and faster than other two methods.

Benefits over other methods:

- Jaccard similarity has greater accuracy and generates less erroneous results for larger datasets.
- It has less response time than first two methods and no significant difference with the third method.
- It is relative easy to implement.
- It has all benefits of Frequency based LSH.

References:

Wu, T., & Miao, Z. (2016). An improved feature image matching algorithm based on Locality-Sensitive Hashing. 2016 IEEE 13th International Conference on Signal Processing (ICSP). doi:10.1109/icsp.2016.7877927

Wang, Q., Guo, Z., Liu, G., & Guo, J. (2012). Entropy based locality sensitive hashing. 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). doi:10.1109/icassp.2012.6288065

Ling, K., & Wu, G. (2011). Frequency Based Locality Sensitive Hashing. 2011 International Conference on Multimedia Technology. doi:10.1109/icmt.2011.6002015a

https://github.com/gamboviol/lsh

https://en.wikipedia.org/wiki/Locality-sensitive_hashing