Exploring the Efficiency of Covering Locality-Sensitive Hashing

Kasper Kronborg Isager
IT University of Copenhagen, Denmark
kasi@itu.dk

Radoslaw Niemczyk
IT University of Copenhagen, Denmark
radn@itu.dk

October 8, 2016

Abstract

One of the latest developments in the area of locality-sensitive hashing (LSH) is the so-called *covering LSH* scheme. This scheme solves one of the primary drawbacks of classic locality-sensitive hashing, namely *false negatives*, i.e. items that never collide with a query item despite them being similar. Covering LSH is therefore of interest in settings that require exact guarantees of producing a nearest neighbour, making classic LSH unsuitable as it only provides probabilistic guarantees. Given that the covering LSH scheme is still relatively new, it still lacks, to the best of our knowledge, a proper general-purpose implementation and associated evaluation thereof.

By implementing both the covering and the classic LSH schemes and benchmarking them on a real-world dataset designed for the evaluation of approximate nearest neighbour search algorithms, we show that within the same time and memory bounds, covering LSH is able to not only match but in fact outperform the filtering efficiency of classic LSH. This does however come at a negligible cost in insertion time and with an additional implementation specific memory overhead.

1 Introduction

An increasingly popular approach for tackling similarity search in high-dimensional datasets is the so-called *locality-sensitive hashing* (*LSH*) technique, first presented by Piotr Indyk and Rajeev Motwani [1]. LSH has therefore already found itself useful in a wide range of practical applications:

- Nearest neighbour search
- Near-duplicate detection
- Hierarchical clustering
- Genome-wide association study
- Image and audio similarity identification
- Human fingerprint recognition

The basic idea of LSH is to hash items to *buckets* in a way that provides a higher probability of similar items being hashed to the same bucket than dissimilar items. Items that then hash to the same bucket despite them being dissimilar are *false positives*. On the other hand, similar items that never hash to the same bucket are *false negatives* [2, p. 88]. While false positives have no effect on the precision of queries, false negatives may cause the algorithm to never consider items that are in fact the most similar to a query item. The latter becomes a problem in settings that require exact rather than probabilistic guarantees of returning a nearest neighbour, as is the case in for example human fingerprint recognition.

A recent paper by Rasmus Pagh [3] proposes an LSH scheme that completely does away with false negatives at a cost in efficiency. The purpose of our paper is to compare this LSH scheme, named *covering LSH*, with classic LSH on a number of different metrics such as query throughput and filtering efficiency.

Organisation This paper is organised as follows: In section 2 we provide the background for classic LSH and outline the covering LSH scheme and the guarantees that it provides. In section 3 we describe our implementation of the two LSH schemes. In section 4 an experimental evaluation based on a real dataset is made after which we compare the two LSH schemes. Finally, in section 5 we give our parting thoughts on the covering LSH scheme and the cost of the exact guarantees that it provides.

2 Background

As touched upon in section 1, locality-sensitive hashing tackles the problem of similarity search in high-dimensional datasets by relaxing the requirement of finding exact nearest neighbours. Instead, LSH is used as an *approximate* nearest neighbour algorithm, which inherently implies only probabilistic guarantees of finding a nearest neighbour. In general, LSH act as a randomized filter that attempts to reduce an input set to a subset of candidates for a given query item.

The LSH schemes considered in this paper both operate in so-called *Hamming space*:

Definition 1 *A Hamming space $H(d, a)$ is the set of all words of length d over an alphabet of size a. A distance measure over a Hamming space is the number of letters in which two words differ.*

The alphabet that we consider for LSH is $\{0, 1\}$ and words are therefore bit vectors of d dimensions. The definition of the approximate nearest neighbour problem for Hamming space is given as follows in [4]:

Definition 2 *Given a set of vectors of dimensionality d $Z \subset \{0,1\}^d$, $|Z| = n$, the Hamming distance function D, a radius $r > 0$, an approximation factor $c > 1$, and a false negative rate $\delta > 0$, construct a data structure such that, given any query $q \in \{0,1\}^d$, if there exists $x \in Z$ and $D(x, q) \leq r$, it reports some $y \in Z$ where $D(y, q) \leq cr$ with probability $1 - \delta$.*

The data structure mentioned in definition 2 makes use of a family of *locality-sensitive hash functions* in order to hash items to buckets. The definition of this family is given as follows in [1]:

Definition 3 *Given $r > 0$, $c > 1$, and probabilities p_1 and p_2 where $p_1 < p_2$, a family H is said to be (r, cr, p_1, p_2)-sensitive for (Z, D) if for any $x, y \in Z$ we have*

- *if $D(x,y) \leq r$ then $Pr_H[h(x) = h(y)] \geq p_1$,*
- *if $D(x,y) > cr$ then $Pr_H[h(x) = h(y)] \leq p_2$.*

Here, p_1 is the lower bound on the probability of close vectors colliding and p_2 is the upper bound on the probability of distant vectors colliding [2, p. 100]. We therefore want p_1 to be close to 1 and p_2 to be close to 0.

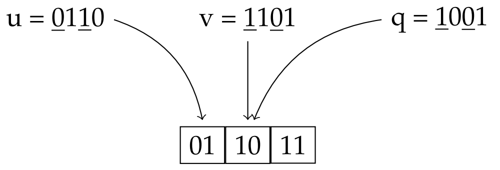
2.1 Classic LSH

The classic LSH family for Hamming space uses a random bit sampling approach for picking a number of components from an input vector. This sample is then used as the key in a map structure, which will be referred to as a *partition*, for locating a bucket that the vector should be stored in. The bits to sample for a given partition are chosen independently and uniformly at random and stored with the partition. When looking for the nearest neighbour of a query vector, this sampling is repeated and the candidate vectors are those found in the bucket that the sample maps to.

Example 1 *Given input vector $v = 1101$, we randomly chose to sample component 1 and 3, giving us the key $v' = 10$. We then proceed to update our map structure with an entry for this key: $10 \rightarrow \{1101\}$*

Given another input vector $u = 0110$, we again sample component 1 and 3, giving us the key $u' = 01$. We then add another entry to our map: $10 \rightarrow \{1101\}, 01 \rightarrow \{0110\}$.

Given a query vector $q = 1001$, we once again sample component 1 and 3, giving us the key $q' = 10$. We then look up this key in our map and receive the following set of candidates: $\{1101\}$.



u = $\underline{0}1\underline{1}0$
v = $\underline{1}1\underline{0}1$
q = $\underline{1}0\underline{0}1$
01
10
11

Figure 1: A graphical depiction of example 1

As can be seen in example 1, v and u are not particularly similar as they only share a single component; by the sampling 1 and 3 they therefore do not map to the same bucket. However, v and q are almost identical as they share all but one component; the sampling 1 and 3 therefore maps them to the same bucket, albeit by chance. We have effectively reduced the set of potential candidates to half of the items in the original input set.

By adjusting the number of components sampled from vectors we can change the probability of collisions happening in the data structure. That is, by increasing the sample size we decrease the chance of vectors colliding, and vice versa, as more components would then have to match in order for a collision to happen.

If we want to keep the same sample size, but still increase the chance of vectors colliding, then we need to use more than one partition. Every partition will then independently chose the bits to sample and the set of candidate vectors will be the union of the vectors found in buckets in each partition.

Collision probabilities By tweaking the sample size and the number of partitions to use, we can control the two probabilities p_1 and p_2 described in definition 3 [2, p. 101]. By choosing a sample size k, the resulting probabilities will be p_1^k and p_2^k. On the other hand, by choosing a number of partitions l, the resulting probabilities will be $1 - (1 - p_1)^l$ and $1 - (1 - p_2)^l$. By combining these, we get the probabilities $1 - (1 - p_1^k)^l$ and $1 - (1 - p_2^k)^l$, giving rise to what [2, p. 89] calls the *S-curve*:

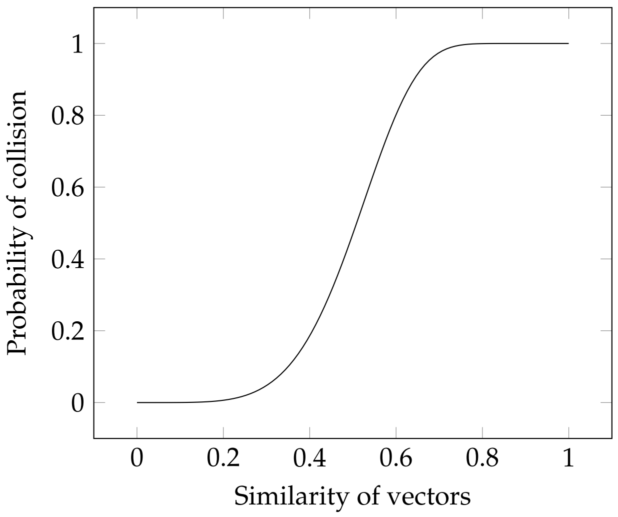


Figure 2: The S-curve for $k = 5, l = 20$

2.2 Covering LSH

Unlike the independently random bit sampling approach of classic LSH, covering LSH relies on a smarter sampling of correlated bits such that it can cover a given radius r. The difference between classic and covering LSH therefore lies in the family of hash functions used. The definition of a covering LSH family for Hamming space is given as follows in [4]:

Definition 4 *An LSH family H is r-covering if for every two binary vectors $x, y \in \{0,1\}^d$ with $D(x,y) \leq r$, there exists $h \in H$ such that $h(x) = h(y)$.*

A covering LSH family makes use of a random mapping $m\colon [d] \rightarrow \{0,1\}^{r+1}$ which constructs d vectors, each consisting of $r + 1$ bits. This mapping is then used for constructing $2^{r+1} - 1$ hash functions, or samplings, each associated with a different partition. In addition, each hash function is associated with a vector representation of $v \in \{1, \ldots, 2^{r+1} - 1\}$. The bits to sample for each function in the family is determined by computing for each bit $b \in \{1, \ldots, d\}$, $m(b) \cdot v \bmod 2$, i.e. the dot product modulo 2 of $m(b)$ and v. If the result is 1, then the bth bit should be sampled.

Example 2 *A 2-covering LSH family for $d = 4$ uses a random mapping $m\colon [4] \rightarrow \{0,1\}^3$, e.g. $m(1) = 011, m(2) = 100, m(3) = 101, m(4) = 001$. This mapping is then used for constructing 7 hash functions, the first of which is $h(1)$:*

$$b(1) = m(1) \cdot v \bmod 2 = 011 \cdot 001 \bmod 2 = 1$$
$$b(2) = m(2) \cdot v \bmod 2 = 100 \cdot 001 \bmod 2 = 0$$
$$b(3) = m(3) \cdot v \bmod 2 = 101 \cdot 001 \bmod 2 = 1$$
$$b(4) = m(4) \cdot v \bmod 2 = 001 \cdot 001 \bmod 2 = 1$$

$h(1)$ therefore samples bit 1, 3, and 4. This computation is repeated for the remaining hash functions, producing the following samplings:

- *$h(2)$ samples bit 1*
- *$h(3)$ samples bit 3 and 4*
- *$h(4)$ samples bit 2 and 3*
- *$h(5)$ samples bit 1, 2, and 4*
- *$h(6)$ samples bit 1, 2, and 3*
- *$h(7)$ samples bit 2 and 4*

	1	2	3
2	$h(5), h(6)$		
3	$h(1), h(6)$	$h(4), h(6)$	
4	$h(1), h(5)$	$h(5), h(7)$	$h(1), h(3)$

Table 1: Depiction of how all possible combinations of 2 bits are covered by functions from the 2-covering family

As can be seen from example 2, which has been adapted from [4, example 2.1], and table 1, a 2-covering LSH family for 4-dimensional vectors is able to "cover" all combinations of bit pairs. This in effect guarantees that no false negatives will be produced within the given radius of 2.

Constraints The covering LSH family works under the assumption that $cr = \log n$. This constraint has in [3] been generalized to arbitrary values of c, r, and n but we will not further touch upon these generalizations in this paper.

3 Implementation

A library implementing both the classic and covering LSH schemes, as described in section 2, has been developed in order to facilitate an experimental evaluation of the two schemes. The library has been released as open-source software and is available at `https://github.com/kasperisager/hemingway`. C++ was chosen as the implementation language as it balances performance with higher-level constructs that help ensure a safe and succinct implementation. The library is fairly simple in that it only exposes two classes: One for representing bit vectors and one for representing the actual LSH data structure.

3.1 Vector

The vector class provides an efficient bit vector representation that allows for arbitrarily large dimensionality by storing bit components in a variable number of 32-bit integer chunks. This representation effectively supports all the operations needed by the LSH schemes:

- Random access in $\mathcal{O}(1)$
- Distance computing in $\mathcal{O}(c)$
- Dot product computing in $\mathcal{O}(c)$
- Equality checking in $\mathcal{O}(c)$
- Hash computing in $\mathcal{O}(c)$
- Bitwise AND in $\mathcal{O}(c)$

Here, c denotes the number of component chunks in a vector. The time complexity of each operation has been listed in order to justify the comparison between the two LSH schemes; as can be seen, no vector operation poses a bottleneck for neither scheme.

Equality and hash The equality and hash operations are used for interoperation with the hash-based data structures of the C++ standard library. As such, they are not related to the concept of hashing in LSH.

Bitwise AND The bitwise AND operation is used for performing the bit sampling described in section 2. For example, to sample bit 1 and 3 of a vector $v = 1101$, we construct a separate vector *mask* $u = 1010$, i.e. a vector with bit 1 and 3 set. The key is then be the bitwise AND of v and u, $1101 \wedge 1010 = 1000$. As can be seen, this is analogous to the key $v' = 10$ from example 1 when bit 2 and 4 are removed.

Population counts It is assumed in [3] that population counts, i.e. the number of 1s in a set of bits, can be counted in $\mathcal{O}(1)$. The `__builtin_popcount`[1] intrinsic supports this and is used in the library, hence complexity $\mathcal{O}(c)$ of the distance, equality, dot product, and bitwise AND operations.

3.2 Table

The table class provides a representation of the LSH data structure itself and is constructed by supplying a configuration for either the classic or covering LSH scheme. In the case of classic LSH, the number of bits to sample from vectors and the number of partitions to use are specified. In the case of covering LSH, the radius is specified.

When configuring the table, the notion of the parameters r, c, and δ from definition 2 is by design left out. The onus of picking parameters appropriate for a given input set and use case is therefore on the client of the library. As a consequence, the data structure also doesn't require specifying the size n of the input set at construction time.

Operations Once configured, vectors can be inserted into and erased from the table using an API similar to that of most data structures in the C++ standard library. Query operations can then be performed against the table once filled with data. A query operation will always return the closest vector found, regardless of the approximation factor c from 2. This is again by design, leaving open the option of the client interpreting the result however they see fit.

Internals Internally, the table consists of several partitions, each represented as a mapping from vector keys to buckets. When adding a vector to the table, this vector is assigned a unique integer identifier and stored in the table. The identifier of the vector is then stored in an associated bucket in each of the different partitions in order to avoid copying the vector itself to every partition. The key used for associating a vector with a bucket is constructed differently depending on the configured LSH scheme, but this

[1] `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`

is in essence the only functional difference between the two schemes.

Hashing Both LSH schemes make use of the Mersenne Twister pseudorandom number generator when constructing hash functions, ensuring a uniform distribution as required by [1] and [3]. The random number generator is in classic LSH used for setting k random bits in vector masks, whereas it is in covering LSH used for setting random bits in vectors for the mapping m.

4 Evaluation

The following section will provide an evaluation of the two schemes based on an automated benchmark implemented as part of the C++ library described in section 3. The benchmark measures the following metrics of each LSH scheme:

- **Insertion throughput:** How many items can be inserted into the data structure per second on average?
- **Query throughput:** How many queries can be performed against the data structure per second on average?
- **Bucket distribution:** How many buckets are created per partition on average?
- **False negative rate:** How many false negatives occur per query on average?

The benchmark performs a 1-NN search in each of three configured tables: A brute force table, a classic table, and a covering table. The purpose of the brute force table is to establish the ground truth by locating the exact nearest neighbour for each of the query items. This ground truth is then used for determining if a false negative has been encountered for each of the query results in the classic and covering tables.

Note: The construction time of the data structure itself under the different schemes is not considered by the benchmark, as we consider this a pre-processing step. However, the construction time required for covering LSH is significantly higher than that required by classic LSH due to the increased complexity of constructing covering families of hash functions. This issue is tackled in [4] which proposes the so-called *fast covering LSH* scheme for efficiently constructing covering families.

Dataset The dataset used for the evaluation is a preprocessed version of the `ANN_SIFT1M` set distributed as part of the Multi Index Hashing (MIH) library located at `https://github.com/norouzi/mih`.

This dataset is created specifically for the evaluation of approximate nearest neighbour search algorithms and is a set of 1 million image feature vectors of 128 dimensions, plus an additional 10,000 query vectors. The version distributed alongside the MIH library is however binarised to vectors of 64 dimensions using LSH, which better suits the purpose of our evaluation.

Parameters Our choice of parameters follows that of the experiments carried out in [4]. That is, for a given radius r, we set the number of partitions to use in the classic table to $l = 2^{r+1} - 1$ and the number of bits to sample to $k = \frac{\log(1-\delta^{\frac{1}{l}})}{\log(1-\frac{r}{d})}$. The benchmark is run for both $\delta = 0.01$ and $\delta = 0.001$. For the covering table we just need the radius r.

4.1 Results

The results were obtained by running the benchmark on a 2.8 GHz Intel Core i5 processor with a 3 MB L3 cache and 8 GB of DDR3 memory. Due to the limited amount of memory available, we have only been able to provide results for $r \in \{2, \dots, 5\}$.

4.1.1 Bucket distribution

We start by considering the average number of buckets created per partition as this turned out to be the defining difference between the two LSH schemes. As can be seen in figure 3, the average number of buckets per partition is in the case of classic LSH dependent on the radius, and as such the number of bits sampled.

This, however, is not the case in covering LSH where the number of buckets per partition remains constant for every radius. This ties into the way covering LSH correlates the bits to sample from vectors, causing the number of buckets per partition to depend on the input set rather than the radius.

The increased number of buckets does however cause an increase in the total memory usage of the data structure, as each bucket in our implementation has an overhead of 24 bytes on a 64-bit operating system.

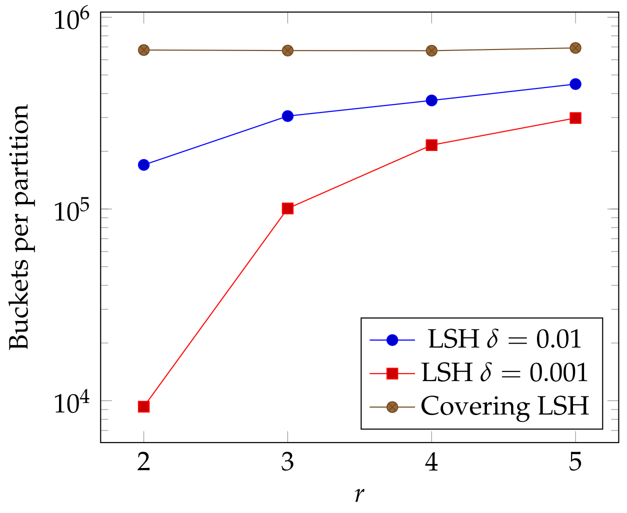


Figure 3: Comparison of bucket distribution

4.1.2 Insertion throughput

As can be seen in figure 4, the insertion throughput of the two LSH schemes follows the same trend as the bucket distribution, albeit in reverse. As such, covering LSH performs worse than classic LSH due to the added overhead of having to construct more buckets. While the difference in bucket distribution is rather significant between the two schemes, the difference in insertion time is negligible in comparison.

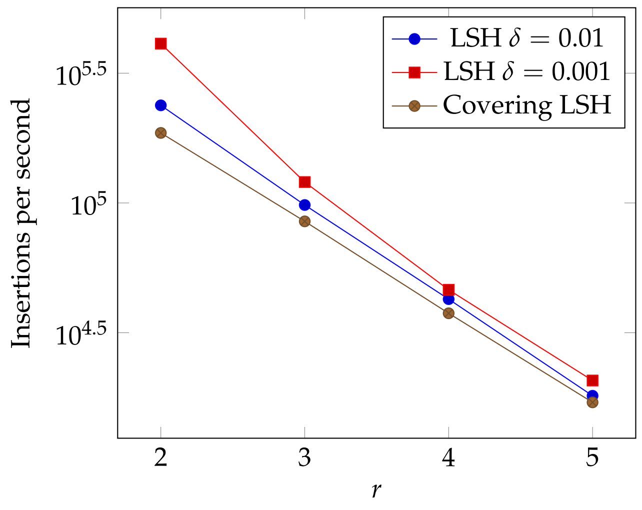


Figure 4: Comparison of insertion throughput

4.1.3 Query throughput

Our initial expectations of the query throughput of the covering LSH scheme was that it would follow the same trend as the insertion throughput, i.e. be slightly less performant than the query throughput of classic LSH. As seen in figure 5, this is however far from the case. Due to the bucket distribution of covering LSH, we can assume that it is able to produce candidate sets that on average are smaller than those produced by classic LSH. The improved filtering efficiency in turn has a significant effect on the query throughput, making covering LSH perform much better than classic LSH on the evaluated dataset.

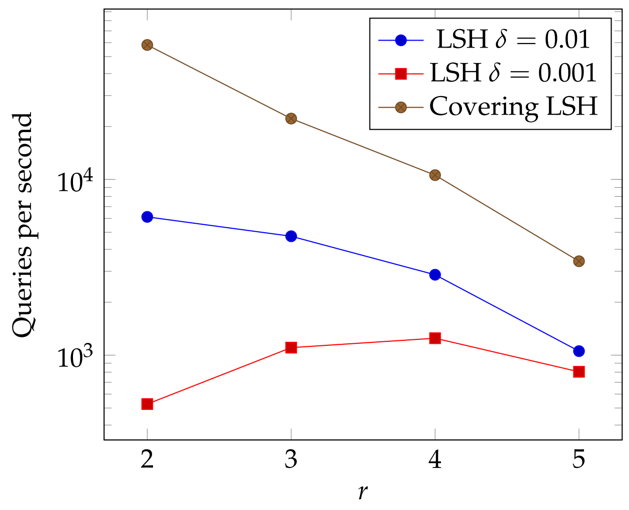


Figure 5: Comparison of query throughput

4.1.4 False negative rates

The last metric considered is the false negative rates of the two LSH schemes. As can be seen in figure 6, the number of false negatives per query is exactly as one would expect; classic LSH produces false negatives proportional to δ and covering LSH produces no false negatives at all, as is also shown by the experiments conducted in [4].

While classic LSH with $\delta = 0.001$ does provide low probabilities of false negatives, it does so at a major decrease in query throughput as witnessed by figure 5. The query throughput of classic LSH with delta $\delta = 0.01$ comes closer to that of covering LSH, but this of course at an increased rate of false negatives.

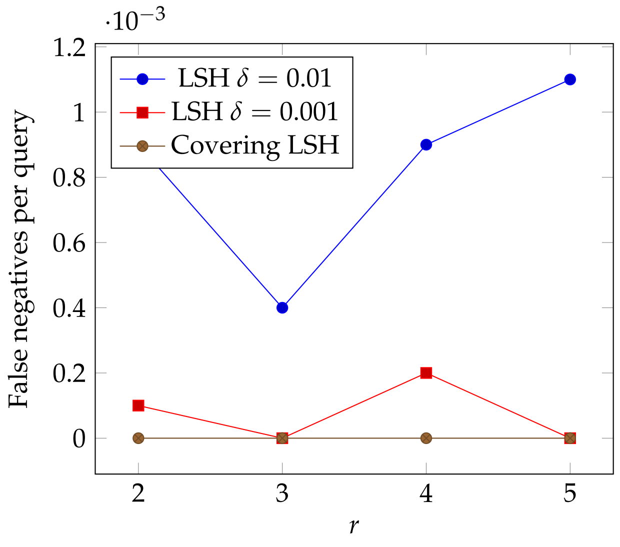


Figure 6: Comparison of false negative rates

5 Conclusion

We have in this paper shown how the proposed covering LSH scheme performs compared to the classic LSH scheme when both are implemented as part of a general-purpose C++ library. Based on our evaluation, we can conclude that the query throughput of covering LSH outperforms that of classic LSH given the same time and memory bounds. This as a result of an improved distribution of items in buckets across partitions, which in turn reduces the average size of candidate sets for query items. The improved filtering efficiency does however come at a cost in the form of a decrease in insertion time and an additional memory overhead, both caused by the increased number of buckets.

In addition, our results verify the claim made in [3] that the efficiency of the covering LSH scheme matches that of the classic LSH scheme of [1] in the case that $cr = \log n$. As stated earlier, the efficiency of our implementation in fact surpasses that of the classic LSH scheme while maintaining the exact guarantees of producing a nearest neighbour.

References

[1] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In J. S. Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 604–613. ACM, 1998.

[2] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[3] R. Pagh. Locality-sensitive hashing without false negatives. *CoRR*, abs/1507.03225, 2015.

[4] N. Pham and R. Pagh. Scalability and total recall with fast coveringlsh. *CoRR*, abs/1602.02620, 2016.