Improving Locality Sensitive Hashing by Efficiently Finding Projected Nearest Neighbors

Omid Jafari, Parth Nagarkar, and Jonathan Montaño

New Mexico State University, Las Cruces, USA
{ojafari,nagarkar,jmon}@nmsu.edu

Abstract. Similarity search in high-dimensional spaces is an important task for many multimedia applications. Due to the notorious *curse of dimensionality*, approximate nearest neighbor techniques are preferred over exact searching techniques since they can return *good enough* results at a much better speed. *Locality Sensitive Hashing* (LSH) is a very popular random hashing technique for finding approximate nearest neighbors. Existing state-of-the-art Locality Sensitive Hashing techniques that focus on improving performance of the overall process, mainly focus on minimizing the total number of IOs while sacrificing the overall processing time. The main time-consuming process in LSH techniques is the process of finding neighboring points in projected spaces. We present a novel index structure called radius-optimized Locality Sensitive Hashing (*roLSH*). With the help of sampling techniques and Neural Networks, we present two techniques to find neighboring points in projected spaces efficiently, without sacrificing the accuracy of the results. Our extensive experimental analysis on real datasets shows the performance benefit of *roLSH* over existing state-of-the-art LSH techniques.

Keywords: Approximate nearest neighbor search · High-dimensional spaces · Locality Sensitive Hashing · Neural Networks

1 Introduction

Finding nearest neighbors is an important problem in many domains such as information retrieval, computer vision, machine learning, multimedia retrieval, etc. For low-dimensions (<10), popular tree-based index structures, such as KD-tree, Quad-tree, etc. are effective, but for higher number of dimensions, these index structures suffer from the well-known problem, *curse of dimensionality* (where the performance of these index structures is often out-performed even by linear scans) [3]. One solution to this problem is to search for approximate results instead of exact results. In many applications where strictly correct results are not necessary, approximate results can produce *good enough* results while achieving much better running times. The goal of the c-approximate version of the Nearest Neighbor problem (ANN) is to find nearest neighbors for a given query point that are within $c * R$ distance (where $c > 1$ is the approximation ratio and R is the search radius).

1.1 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [8] is a very popular technique for solving the Approximate Nearest Neighbor problem in high-dimensional spaces. LSH uses *random* projections to map high-dimensional points to lower dimensional representations. The intuition behind LSH is that nearby points in high-dimensional spaces will map to same (or nearby) hash buckets in the projected lower dimensional space with a high probability (and vice-versa). Since the original LSH index structure was proposed for Hamming distance, LSH families have been proposed for other popular distances such as the Euclidean distance [6]. The main benefits of LSH are three-fold: 1) LSH provides theoretical guarantees on the accuracy of the results, 2) LSH can answer ANN queries in sub-linear time with respect to the dataset size, and 3) LSH can be easily implemented as external memory-based index structures, thus making them more scalable [13]. While the original LSH design suffered from large index sizes [16], recent works [4,7,9] have either improved theoretical bounds or introduced techniques such as *Collision Counting* (Sect. 3) to reduce the number of required hash functions. Due to the popularity of LSH in diverse applications [18,22], several research works have been proposed to improve the search efficiency and/or accuracy of LSH techniques [4,7,9,10,13,14,16,23].

1.2 Motivation of Our Work: Improving the Efficiency of Existing State-of-the-Art LSH Techniques

One of the important benefits of LSH is their ease of implementation as external storage based algorithms. State-of-the-art external memory-based algorithms (namely C2LSH [7], QALSH [9], and I-LSH [13]) use a bucket-expansion strategy to find points from neighboring buckets. C2LSH and QALSH use a bucket exponential expansion strategy, whereas I-LSH uses an incremental expansion strategy. While I-LSH is the state-of-the-art algorithm that minimizes disk I/Os, it achieves this optimization at the expense of a costly overall processing time as shown in Sect. 6.[1] Additionally, random I/Os (disk seeks) are known to be bottleneck in query processing [11] and much more expensive than sequential I/Os [12]. I-LSH reduces overall I/Os by mainly reducing sequential I/Os. In this paper, our goal is to design an LSH external memory technique, *roLSH*, that can reduce overall IOs, *mainly random I/Os*, (by finding neighboring points efficiently) which improves the overall query processing time.

[1] There is no existing work that compares the overall performance of C2LSH, QALSH, and I-LSH. We present a detailed performance analysis between these works as a technical report (https://arxiv.org/abs/2006.11285).

1.3 Contributions of This Paper

In this paper, we propose a novel approach, called radius-optimized Locality Sensitive Hashing (*roLSH*) for efficiently finding top-k approximate nearest neighbors in high-dimensional spaces. Our main contributions are as follows:

- We present a sampling-based technique, *roLSH-samp*, that reduces the overall random disk I/Os which improves the query processing time while satisfying the theoretical guarantees of LSH. We provide the theoretical analysis for the correctness of *roLSH-samp*.
- We further improve the efficiency by proposing a Neural Network-based technique, *roLSH-NN*, for an improved prediction of projected radiuses (and thus further reduction in random disk I/Os), and hence further improving the performance without affecting the query accuracy. To the best of our knowledge, we are the first work to improve LSH parameters by using Neural Networks.
- Lastly, we experimentally evaluate both techniques of *roLSH* on real high-dimensional datasets and show that *roLSH* can outperform the state-of-the-art solutions in terms of performance while providing similar query accuracy.

2 Related Work

Locality Sensitive Hashing is a popular technique for solving the Approximate Nearest Neighbor (ANN) problem in high-dimensional spaces. It was first introduced in [8] for the Hamming distance and later extended to the Euclidean distance (E2LSH) [6]. These structures suffered from large index sizes due to the need to have large number of hash functions in multiple hash tables [7]. Additionally, a *magic radius* need to be inputted to find the neighboring projected points, and in order to find the desired number of results, this *magic radius* was arbitrarily chosen to be very high. Multi-Probe LSH [16] presented a technique to probe neighboring buckets if enough number of results were not found. C2LSH [7] introduced a *Collision Counting* approach that reduced the need to have multiple hash tables, and hence reduced the overall index size. SK-LSH [14] introduced a linear ordering on the disk pages with the help of Z-order curve in order to reduce the overall I/Os. The drawback of SK-LSH was that it was created on the original LSH design, and hence also suffered from the *magic radius* problem. QALSH [9] introduced query-aware hash functions and further reduced the number of hash functions necessary to achieve theoretical guarantees. The work closest to our proposed idea is I-LSH [13], which introduces an incremental strategy for finding nearest neighbors in the projected space, as explained in the next Sect. 3.1. Recently, PM-LSH [23] developed a novel tunable confidence interval while using a PM-tree to solve c-ANN queries.

3 Background and Key Concepts

In this section, we describe the key concepts behind LSH. We mainly use the notations and formulations described in the seminal paper on Euclidean LSH families [6] and C2LSH [7].

Hash Functions: A hash function family H is (R, cR, p_1, p_2)-sensitive if it satisfies the following conditions for any two points x and y in a d-dimensional dataset $D \subset \mathbb{R}^d$: if $||x-y|| \leq R$, then $Pr[h(x) = h(y)] \geq p_1$, and if $||x-y|| > cR$, then $Pr[h(x) = h(y)] \leq p_2$.

Here, p_1 and p_2 are probabilities and c is an approximation ratio. In order for LSH to work, $c > 1$ and $p_1 > p_2$. The above definition states that the two points x and y are hashed to the same bucket with a very high probability $\geq p_1$ if they are close to each other (i.e. the distance between the two points is less than or equal to R), and if they are not close to each other (i.e. the distance between the two points is greater than cR), then they will be hashed to the same bucket with a low probability $\leq p_2$. In the original LSH scheme for Euclidean distance, each hash function is defined as $h_{\boldsymbol{a},b}(x) = \left\lfloor \frac{\boldsymbol{a}.x+b}{w} \right\rfloor$, where $\boldsymbol{a}$ is a d-dimensional random vector with entries chosen independently from the standard normal distribution $N(0,1)$ and b is a real number chosen uniformly from $[0,w)$, such that w is the width of the hash bucket [6]. This leads to the following collision probability function [6], which states that if $||x-y|| = r$, then the probability that x and y map to the same hash bucket for a given hash function $h_{\boldsymbol{a},b}(x)$ is: $P(r) = \int_0^w \frac{1}{r}\frac{2}{\sqrt{2\pi}}e^{\frac{-t^2}{2r^2}}(1-\frac{t}{w})dt$. Here, the collision probability $P(r)$ is decreasing on r for a given w. For a t, which is the largest absolute value of a coordinate of point in D, and for every b uniformly drawn from the interval $[0, c^{\lceil \log_c td \rceil} w^2]$ and $R = c^n$ for some $n \leq \lceil \log_c td \rceil$ we have that $h^R(x) = \left\lfloor \frac{h_{a,b}(x)}{R} \right\rfloor$ is (R, cR, p_1, p_2)-sensitive, where $p_1 = p(1)$ and $p_2 = p(c)$ [7].

Collision Counting: In [7], authors theoretically show that two close points x and y collide in at least l hash layers (out of m hash layers) with a probability $1-\delta$. Further, only those points that collide at least l times with the query point, where l is the collision count threshold, are chosen as candidates. We refer the reader to [7] for further details. Since C2LSH creates only one hash function per hash layer, the number of hash functions are equal to the number of hash layers.

3.1 Existing Techniques for Finding Neighboring Projected Points

C2LSH [7] also introduced the concept of *Virtual Rehashing* that finds neighboring points that collide in neighboring hash buckets. The naive solution to finding neighboring points is to use a large projected radius such that enough neighboring points are found to return top-k results. The projected radius is entirely dependent on the data distribution, and as we show in Fig. 2, these projected radiuses can vary significantly. Hence, using an arbitrarily large radius results in wasted I/Os and unnecessary processing. Instead, *Virtual Rehashing* starts with a very small radius (R = 1), and then exponentially increases the radius in the following sequence: $R = 1, c, c^2, c^3 ....$ If at *level-R*, enough candidates are not found, the radius is increased until enough query results are found. C2LSH [7] and QALSH [9] follow this exponential expansion strategy. I-LSH [13] introduces an incremental strategy where, instead of expanding the search radius exponentially, they find the nearest point to the query in each projection. C2LSH and

QALSH store points in hash buckets which are stored in disk pages. I-LSH stores each data point separately and hence instead of reading disk pages that store a group of data points, only reads a point (which effectively is the same as reading a disk page of 4 bytes). While they save on disk I/O operations by this method, this is a very costly operation (as shown in Sect. 6) since this process has to be done thousands of times for larger radiuses.

4 Problem Specification

The approximate version of the nearest neighbor problem, also called *c-δ -approximate Nearest Neighbor search*, aims to return points that are within $c{*}R$ distance from the query point with probability at least $1 - \delta$, where $c > 1$ is a user-defined approximation ratio, R is the distance of the query point from its nearest neighbor, and δ is a user-defined error probability.

In this paper, our goal is to return c-δ-ANNs for a given query q while reducing the overall processing time and satisfying the theoretical guarantees. In Sect. 5, we present the processing cost breakdown of the LSH process based on which we design our proposed index structure, radius-optimized Locality Sensitive Hashing (*roLSH*).

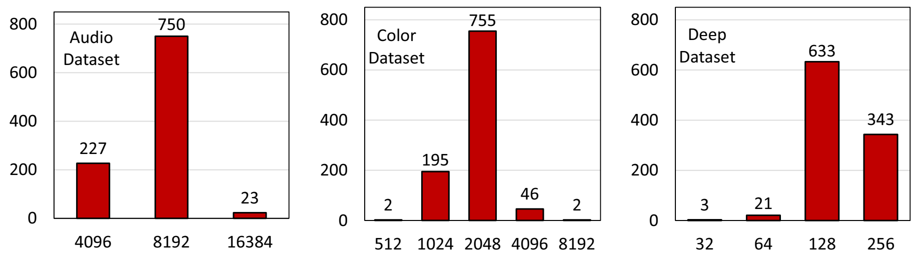

Fig. 1. Frequency (Y-axis) of Final Radius Values (X-axis) for finding Top-100 Points for 1000 Point Queries on Different Datasets using C2LSH

5 roLSH

In this section, we present the design of *roLSH*, which consists of two strategies for efficiently finding neighboring points in the hash functions. We introduce and describe these two strategies in this section: a sampling-based strategy, called *roLSH-samp*, and a Neural Network-based strategy, called *roLSH-NN*.

5.1 Sampling-Based Improved Virtual Rehashing Strategy

In Sect. 2, we explained the original Virtual Rehashing strategy (denoted as *oVR* strategy) as proposed in C2LSH [7]. The initial radius is set to 1, and if sufficient results are not found, then the radius is increased in an exponential sequence: $R = 1, c, c^2, c^3 ...$ until sufficient number of results are found. The main drawback of this approach is when the values of R become larger (i.e. when the difference between two consecutive radius values is large - e.g., 4096 and 8192). In such situations, it happens frequently that very few (or no) nearest neighbor points are found at radius value 4096 but all (and lot more) are found at radius 8192. Thus, for example, if the actual radius of the kth-nearest point was near 5000, then index files corresponding to radius 5000–8192 will be read unnecessarily from the disk, leading to expensive wasted IO operations. Instead, we propose a sampling-based improved Virtual Rehashing strategy (denoted as *roLSH-samp*) based on the following observation:

Observation 1. *For high-dimensional datasets, the required radius values for a k value are similar to each other for different query points for a given dataset.*

This observation was also noted by a very recent paper [23] where the authors show that the homogeneity of the distance distributions of data points in different high-dimensional datasets is very high. Figure 1 shows our observation on popular real high-dimensionsal datasets with varying cardinalities and dimensionalities (Audio [1], Color [5], Deep [2]). For 1000 randomly chosen query points, we report the final radius values (using the Virtual Rehashing technique from C2LSH [7]) for top-100 points. By leveraging the above observation, we design an improved and effective *Virtual Rehashing* technique: we execute a sample set of randomly chosen queries for a given k and count the number of occurrences of the final radius value. We choose our initial radius value that is before the radius with the maximum count of sampled queries. E.g. in the Audio dataset (Fig. 1), the radius with the maximum count is 8192. For these queries, it means that the optimal radius would be between 4096 and 8192. Hence we choose our initial radius value to be 4096. Thus, instead of starting at the initial radius of 1, we find an improved initial starting radius (denoted as $i2R$) based on sampling queries. Note that, since this is done during the indexing phase, it has no overhead during query execution. Once the initial starting radius ($i2R$) is found, we leverage the same exponential sequence strategy as C2LSH (using x as the expansion step counter), such that:

$$R = \begin{cases} i2R + 2^x & 0 \leq x \leq \log_2 i2R \\ 2^x & x > \log_2 i2R \end{cases}$$

Thus, for 1000 random queries on the Audio dataset, using the oVR technique, the average final radius is 7450 for $c = 2$ and $k = 100$. On the other hand, using our improved strategy, the average final radius is 6083, which leads to significant savings in the IO.

Note that, one disadvantage of this approach is that there potentially can be queries that finish with a radius value much lower than the chosen initial radius. E.g., in the Color dataset (Fig. 1), our strategy will choose $i2R = 1024$. As you can see, there were 2 out of 1000 queries whose final radius value to find top-100 points was 512. In this case, the improved Virtual Rehashing strategy (denoted as iVR strategy) will do wasted work by starting (and ending) at 1024.

Lemma 1. *For those queries whose required radius in oVR is at least $(2 \times i2R)$, iVR strategy will generate less IOs than the oVR strategy.*

Proof. Set $R = i2R$. By construction of the sequence of radii in oVR, it is enough to assume that the required radius is $2R$, that is, the actual radius r of the kth-nearest point satisfies $R < r \leq 2R$. In the oVR, the sequence of radii needed to find the kth-nearest point has $\log_2 R + 2$ elements, that is, $1, 2, 4, \ldots, 2R$. On the other hand, for the same query q, iVR analyzes at most $\log_2 R + 1$ radii, that is, $R+1, R+2, \ldots, 2R$. This finishes the proof.

While I-LSH [13] still generates less disk I/Os than *roLSH-samp*, *roLSH-samp* is significantly faster than I-LSH (due to less overall processing time) and also generates less disk seeks than I-LSH for bigger datasets (Sect. 6).

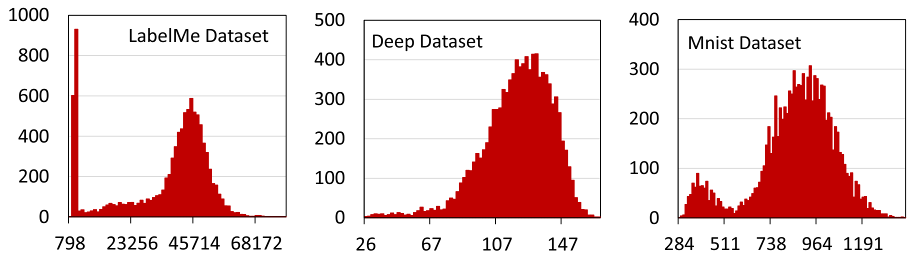

Fig. 2. Frequency (Y Axis) of radiuses (X Axis) for 10,000 Top-100 Queries

5.2 Drawbacks of *roLSH-samp*

The main benefit of *roLSH-samp* is that it is effective in reducing the disk I/Os, especially when the radiuses are large (e.g., the Audio dataset in Fig. 1). There is a minor overhead of utilizing the sampling-based method during the indexing phase. Additionally, we found that we also get good sampling representatives even with a small sampling size (e.g., 100). There are two main drawbacks of *roLSH-samp*: 1) *roLSH-samp* works best when Observation 1 holds true. We found out that Observation 1 holds true for many datasets, but not all. For example, as seen in Fig. 2, the radiuses for top-100 queries on the LabelMe dataset are quite different leading to inefficient performance of *roLSH-samp* (as shown in Sect. 6), 2) It is not easy to do sampling for different k values since the radius changes for different k values. It is not trivial to build a single model and extend it to multiple k values to find the radius for a particular k value. Instead a model needs to be built for *each* k value.

Table 1. Performance comparison of learning techniques

	MLP	Linear Reg.	RANSAC	Decision tree	Gradient boosting
MSE	0.0265	0.3543	0.3542	0.7057	0.2117
R2	0.9687	0.5826	0.5827	0.1698	0.7504

5.3 Neural Network-Based Prediction of Projected Radiuses

To remedy these two drawbacks, we present a Neural Network-based strategy, *roLSH-NN*, that can better predict starting radiuses based on the query location (in each hash function) for any given k value. The main intuition behind *roLSH-NN* is that nearby points in the original space will have similar projected radiuses to find the desired number (k) of nearest neighbors. Hence, our goal is to predict the projected radiuses given the hash locations of a query for a given k.

Formally, let $h_i(q)$ denote the bucket location of q in the ith hash projection. Thus, $H(q) = h_1(q), ..., h_m(q)$ denotes a vector of size m (since there are m hash projections) that contains m bucket locations for a given query point q. Let $R_{act}(q, k)$ denote the smallest radius in the projected space that satisfies the desired number of results (k). Let Q_{tr} be the set of training queries, where for each query $q \in Q_{tr}$, we also find the ground truth (i.e. $R_{act}(q, k)$), which is entered as a target value into the neural network during training. This step is done in the indexing step, and hence does not affect the query processing time. We train a Neural Network with Q_{tr} queries such that for each query q, we input $H(q)$ and k and the Neural Network outputs the predicted radius, $R_{pred}(q, k)$. We explain the different characteristics of our Neural Network in Sect. 6.

Justification for Choosing Neural Networks: Since the problem of predicting radiuses given the hash function is a regression problem, we tried several machine learning techniques. Table 1 shows that Neural Networks (denoted by *MLP* since we use a Multilayer Perceptron Neural Network) have the best MSE and R2 for a sample dataset (Deep) for $Q_{tr} = 10,000$ among different machine learning techniques (using 10-fold cross validation). Hence, we choose Neural Networks over other techniques in the design of *roLSH-NN*.

Underestimation of Radius: When the radius is underestimated (i.e. $R_{pred}(q, k) < R_{act}(q, k)$), the desired number of results are not found and hence we have to enlarge the radius in all projections. One strategy is to follow the same expansion pattern of *roLSH-samp* presented in Sect. 5.1, where the predicted radius is set as $i2R$. We call this strategy *roLSH-NN-iVR*. The drawback of this strategy is that it can lead to excessive (and expensive) disk seeks if the predicted radius is much lower than the actual radius. Since we observe that $R_{pred}(q, k)$ is close to $R_{act}(q, k)$, we also adopt another strategy where we increase the predicted radius, $R_{pred}(q, k)$, linearly by R_{inc} such that $R_{inc} = R_{pred}(q, k) \times \lambda$. This strategy is referred to as *roLSH-NN-λ* in the rest of this paper.

Overestimation of Radius: While overestimation of the projected radius by the Neural Network leads to wasted disk I/Os during query processing, we experimentally show in Sect. 6 that these wasted disk I/Os are still less than the exponential strategy of C2LSH/QALSH and the improvement in the query processing time (as compared with I-LSH) offsets the disk I/Os significantly.

Extension to any k: In order to train the Neural Network to work for any number of desired results (k), we need to include k as an input feature in the training set. In order to simplify the training procedure, we only consider few values of k in the training set Q_{tr}. In Sect. 6, we experimentally show that as more diverse k values are included in the training set, the MSE decreases. Also, we explain the training setup and the different k values in our training set.

6 Experimental Evaluation

In this section, we evaluate the effectiveness of our proposed index structure, *roLSH*, on three real diverse high-dimensional datasets. All experiments were run on the nodes of the Bigdat cluster[2] with the following specifications: two Intel Xeon E5-2695, 256 GB RAM, and CentOS 6.5 operating system. We implement our work on top of C2LSH [7] since we found it to be the fastest external memory-based LSH algorithm (while achieving high accuracy for high-dimensional datasets). Note that, our method is orthogonal to the LSH algorithm and can be used in any state-of-the-art LSH algorithms. All codes were written in C++11 and compiled with gcc v4.7.2 with the -O3 optimization flag. We compare our three strategies, *roLSH-samp*, *roLSH-NN-iVR*, and *roLSH-NN-λ* with the state-of-the-art LSH algorithms C2LSH [7] and I-LSH [13].[3,4]

Table 2. Comparison of (a) Index Construction Time (in sec) and (b) roLSH Index Time Breakdown (in sec) on Different Datasets

Index Time	LabelMe	Deep	Mnist
roLSH-samp	83.5	93.5	1480.8
roLSH-NN	88.6	98.4	1488
C2LSH	80.6	69	1430.2
I-LSH	20.8	25.7	1359.9

Index Time Breakdown	LabelMe	Deep	Mnist
Base Index Time	80.6	69	1430.2
Sampling	2.9	24.5	50.6
NN Training	8	29.4	57.8

6.1 Datasets

We use the following three popular real datasets to evaluate the proposed method:

[2] Supported by NSF Award #1337884.

[3] PM-LSH code is not yet released; hence, we do not compare with it.

[4] The source code for *roLSH* is available at: https://3m.nmsu.edu/rolsh.

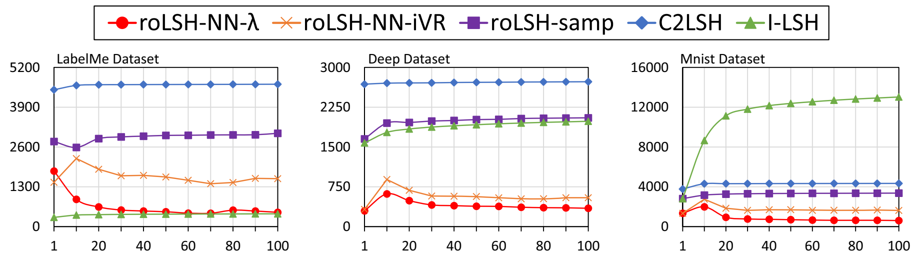

Fig. 3. Number of Disk Seeks (Y axis) for different k (X Axis) on 3 datasets

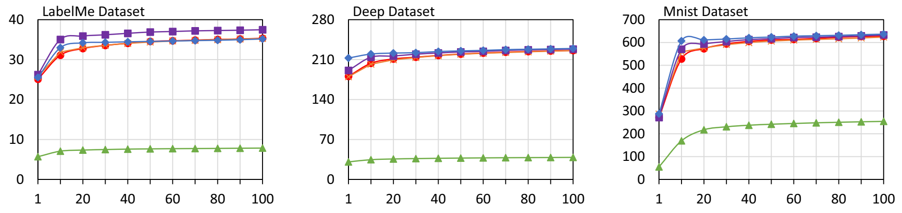

Fig. 4. Amount of Data Read (in MB) (Y axis) for k (X Axis) on 3 datasets

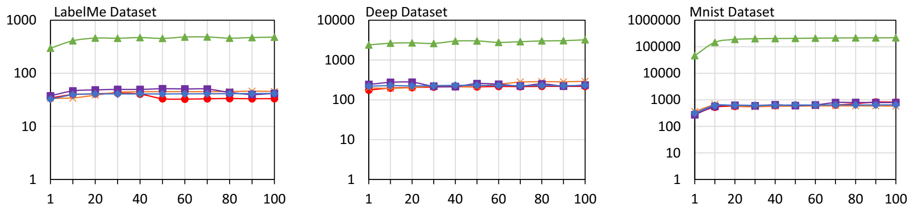

Fig. 5. Algorithm Time (in ms, *log scale*) (Y axis) for k (X Axis) on 3 datasets

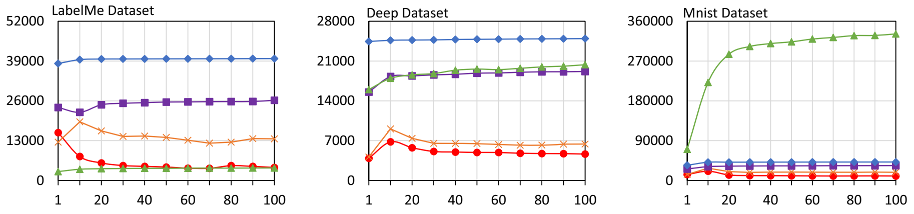

Fig. 6. Query Processing Time (in ms) (Y axis) for k (X Axis) on 3 datasets

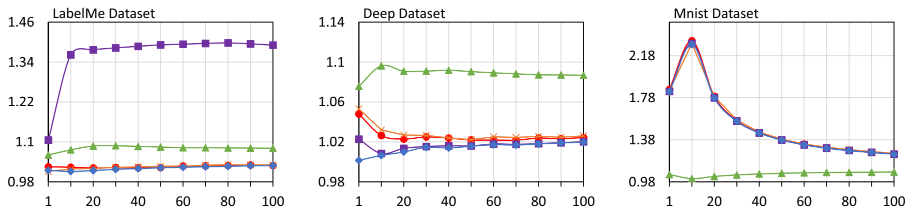

Fig. 7. Accuracy Ratio (Y axis) for different k (X Axis) on 3 datasets

- **LabelMe** [19] consists of $181,093$ 512-dimensional points which were generated by running the GIST feature extraction algorithm on annotated images.
- **Deep** consists of $1,000,000$ 96-dimensional points that were randomly chosen from the Deep1B dataset introduced in [2].
- **Mnist** [15] This dataset contains $8,100,000$ 784-dimensional points that represent images of the digits 0 to 9 which are grayscale and of size 28×28.

6.2 Evaluation Criteria and Parameters

The goal of *roLSH* is to improve the performance efficiency without sacrificing the accuracy of existing LSH techniques. The performance and accuracy of the technique used in this paper are evaluated using the following metrics:

- **Query Processing Time** (QPT): We break down the Query Processing Time into the Index I/O cost, the Algorithm time ($AlgTime$), and the negligible false positive removal cost (denoted by $FPRemTime$, which consists of the cost of reading the data point candidates and computing their exact Euclidean distance for removing false positives). Following [13], we further break down Index I/O cost into the number of disk seeks (i.e. random I/O reads, $noDiskSeeks$) and the amount of data (i.e. index files, $dataRead$) read in MB. We observed that the index I/O times were not consistent (i.e. running the same query multiple times, which needed the same index I/Os, would return drastically different results, mainly because of disk cache and instruction cache issues). Therefore, following [20], for a Seagate 1TB HDD with 7200 RPM, we assume a random seek to cost 8.5 ms on average, and the average time to read data to be 0.156 MB/ms. Thus, we have $QPT = noDiskSeeks * 8.5 + dataRead * 0.156 + AlgTime + FPRemTime$.
- **Accuracy:** We follow the accuracy ratio definition followed by many previous works [7,9,14]: $\frac{1}{k}\sum_{i=1}^{k}\frac{||o_i - q||}{||o_i^* - q||}$. Here, o_i is the ith point returned by the technique and o_i^* is the true ith nearest point from q (ground truth). Ratio of 1 means the returned results have the same distance from the query as the ground truth. The closer the ratio is to 1, the higher is the accuracy.

For the state-of-art methods, we used the same parameters suggested in their papers ($w = 2.719$ for QALSH and $w = 2.184$ for C2LSH). Also, as *roLSH* is built on top of C2LSH, it uses the same parameters as C2LSH. We set the allowed error probability, δ, to be 0.1. The Multilayer Perceptron (MLP) Neural Network is implemented using the Scikit-learn Python package [17]. In this paper, we use the default parameters and options (i.e. 100 hidden layers, *ReLU* activation function, and the *Adam* optimization algorithm). We leave the hyper-parameter tuning analysis to future work. We choose 10,000 training queries randomly from the dataset. 50 different queries were randomly chosen from the dataset for the evaluation. We report an average of the results on these 50 queries.

6.3 Effect of Different Parameters on Performance of *roLSH-NN-λ*

In this section, we present the performance of *roLSH-NN-*λ under different parameters for the Deep dataset.

Effect of Training Size: We consider three different training sizes (5K, 10K, 50K). In our experiments, the MSE reduces (by 18.4% between 5K and 50K training size) as the training size increases and since the MSE decreases (i.e. the predicted radius is close to the actual radius), the overall Query Processing Time (QPT) also decreases (by 33% between 5K and 50K training size). In the following experiments, we choose 10K as the default training size. Due to space limitations, we do not present the results in Sect. 6.3 in detail.

Effect of Number of Different k in Training: We analyze the performance of *roLSH-NN-*λ for different values of k that are present in the training data while keeping the total training size and λ constant. We chose {1, 50, 100}, {1, 25, 50, 75, 100}, and {1, 10, 25, 50, 75, 90, 100} as three different settings. The MSE reduces as more diverse k are included: by 33% between the first two settings, but only by 14% between the last two settings since the neural networks are capable of adequately predicting the radiuses for different k even for the second setting (which is our default in the following experiments).

Effect of Different Radius Increment (λ): We experiment using λ values of 5%, 10%, and 20%. As λ increases, the number of disk seeks decrease (by 32%) since a higher λ eventually results in a larger radius and in turn makes the algorithm stop sooner without processing all projections, but the algorithm time and the amount of I/O increases (by 4% and 1% respectively) since more hash buckets are processed. We choose 10% as our default in further experiments.

6.4 Discussion of the Results

Table 2 (a) shows the time taken to finish the index construction. The reported times show that the sampling and training overhead for *roLSH-samp* and *roLSH-NN* are only 3.4% and 3.9% for the largest dataset (Mnist). Table 2 (b) shows the break-down of index construction time for *roLSH-samp* and *roLSH-NN* (i.e. overhead of the techniques used in each method) for all datasets. The index size of our techniques are similar to C2LSH since we use C2LSH as our underlying LSH implementation. The index size overhead of *roLSH-samp* is 0.1 MB for all datasets, and overhead of *roLSH-NN* is 0.4 MB for LabelMe and 0.5 MB for Deep and Mnist datasets.

Number of Disk Seeks: Figure 3 shows the number of disk seeks (random I/Os) required by these different techniques. It is very interesting to note that while I-LSH performs the best (*roLSH-NN-*λ is a close second) for LabelMe, their performance degrades as the dataset size increases. I-LSH produces significant more disk seeks as the dataset size increases. We believe this is mainly due to the fact that more points need to be accessed incrementally to find the candidates. *roLSH-NN-*λ significantly performs the best for Deep and Mnist datasets because it can accurately predict the radius for different k. Every time *roLSH-NN-*λ underestimates the radius (Sect. 5.3), it has to increment the radius by λ resulting in a disk seek in each projection. Also, as expected, *roLSH-NN-iVR* produces more disk seeks due to radius underestimation (Sect. 5.3).

Amount of Data Read: Figure 4 shows the total amount of data (index files) read. Since I-LSH incrementally increases the search to the nearest point in the projected space (instead of an empirically chosen number, such as λ), it results in the least amount of data read for all datasets. These savings in the I/O are offset due to the expensive search for the nearest point as shown in Fig. 5. Especially for lower k, *roLSH-NN-iVR* and *roLSH-NN-*λ read less data than C2LSH, but as k increases the overall data read is similar for both techniques. It is interesting to note that *roLSH-samp* reads significantly more data for LabelMe dataset. This is due to choosing of a bad starting radius due to the unique distribution of the LabelMe radiuses (Fig. 2 (a)). Moreover, *roLSH-NN-iVR* and *roLSH-NN-*λ read similar amount of data since their starting radius is the same.

Algorithm Time: Figure 5 shows the time needed by the algorithms to find the candidates (excluding the time taken to read the index files). Note the log scale of this figure because the algorithm time for I-LSH was orders of magnitude more than the other techniques. This is because I-LSH expands the radius incrementally in each projection which creates a significant overhead. Figure 5 shows that the overhead of our methods is negligible when compared with C2LSH.

Query Processing Time: Figure 6 shows the overall time required to solve a given k-NN query. I-LSH works well for smaller datasets (LabelMe) but is significantly slower as the dataset size increases (due to high overhead in incrementally finding the next neighbor in each projection). *roLSH-samp* is always faster than C2LSH because of the savings in disk seeks. *roLSH-NN-iVR* and *roLSH-NN-*λ are always much faster than *roLSH-samp* and C2LSH because of their ability to accurately predict radiuses, resulting in significantly less disk seeks and lesser (or similar in some cases) data read than C2LSH. *roLSH-NN-*λ has better performance compared to *roLSH-NN-iVR*, mainly because of having lesser disk seeks as discussed before. This figure shows the performance benefit of *roLSH-NN-*λ over its competitors for different datasets, and confirms that the design of *roLSH-NN-*λ leads to improvement in overall efficiency.

Accuracy: Figure 7 shows the accuracy of all techniques. *roLSH-samp* gives the worst accuracy for LabelMe dataset. We found that this is due to the fact that LabelMe dataset has queries with very different large radiuses. *roLSH-samp* is unable to work well for datasets that have differing radiuses because if the starting radius is chosen wrong, then *roLSH-samp* can significantly overestimate the radius for larger radiuses leading to lower accuracy. *roLSH-NN-*λ always returns similar accuracy to that of C2LSH. I-LSH returns a better accuracy for Mnist dataset due to their usage of *query-aware* hash functions, but the performance is significantly slower as shown in Fig. 6.

7 Conclusion

Locality Sensitive Hashing is a popular technique for efficiently solving Approximate Nearest Neighbor queries in high-dimensional spaces. State-of-the-art LSH techniques improve the overall disk I/Os at the expense of algorithm time. In this paper, we present a unique index structure called radius-optimized Locality Sensitive Hashing (*roLSH*). The goal of *roLSH* is to improve the efficiency of LSH techniques by improving the random disk seeks without any significant overhead in algorithm time. We propose two novel strategies, *roLSH-samp* and *roLSH-NN* that are based on sampling and Neural Networks respectively. Experimental results on real datasets show the benefit of *roLSH* in improving overall performance over existing state-of-the-art techniques, C2LSH and I-LSH.

References

1. Audio dataset. http://www.cs.princeton.edu/cass/audio.tar.gz
2. Babenko, A., Lempitsky, V.: Efficient indexing of billion-scale datasets of deep descriptors. In: CVPR (2016)
3. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces In: CSUR (2001)
4. Christiani, T.: Fast locality-sensitive hashing frameworks for approximate near neighbor search. In: SISAP (2019)
5. Color dataset. http://kdd.ics.uci.edu/databases/CorelFeatures
6. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SOCG 2004 (2004)
7. Gan, J., Feng, J., Fang, Q., Ng, W.: Locality-sensitive hashing scheme based on dynamic collision counting. In: SIGMOD (2012)
8. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB (1999)
9. Huang, Q., Feng, J., Zhang, Y., Fang, Q., Ng, W.: Query-aware locality-sensitive hashing for approximate nearest neighbor search. VLDB **9**, 1–12 (2015)
10. Jafari, O., Ossorgin, J., Nagarkar, P.: qwLSH: cache-conscious indexing for processing similarity search query workloads in high-dimensional spaces. In: ICMR (2019)
11. Kim, A., Xu, L., Siddiqui, T., Huang, S., Madden, S., Parameswaran, A.: Optimally leveraging density and locality for exploratory browsing and sampling. In: HILDA (2018)
12. Leis, V., et al.: Query optimization through the looking glass, and what we found running the join order benchmark. VLDB **27**, 643–668 (2018)

13. Liu, W., Wang, H., Zhang, Y., Wang, W., Qin, L.: I-LSH: I/O efficient c-approximate nearest neighbor search in high-dimensional space. In: ICDE (2019)
14. Liu, Y., Cui, J., Huang, Z., Li, H., Shen, H.T.: SK-LSH: an efficient index structure for approximate nearest neighbor search. VLDB **7**, 745–756 (2014)
15. Loosli, G., Canu, S., Bottou, L.: Training invariant support vector machines using selective sampling. Large Scale Kernel Mach. (2007)
16. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: VLDB (2007)
17. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. JMLR **12**, 2825–2830 (2011)
18. Rong, K., et al.: Locality-sensitive hashing for earthquake detection: a case study of scaling data-driven science. In: VLDB (2018)
19. Russell, B.C., Torralba, A., Murphy, K.P., Freeman, W.T.: LabelME: a database and web-based tool for image annotation. IJCV **77**, 157–173 (2008)
20. Seagate ST2000DM001 Manual. https://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf
21. Sift dataset. http://corpus-texmex.irisa.fr
22. Yang, Z., Ooi, W.T., Sun, Q.: Hierarchical, non-uniform locality sensitive hashing and its application to video identification. In: ICME (2004)
23. Zheng, B., Zhao, X., Weng, L., Hung, N.Q.V., Liu, H., Jensen, C.S.: PM-LSH: a fast and accurate LSH framework for high-dimensional approximate NN search. VLDB **13**, 643–655 (2020)