INVITED PAPER

Optimal Parameters for Locality-Sensitive Hashing

An algorithm is described that optimizes parameters for nearest-neighbor retrieval in web-scale search, at minimum computational cost.

By Malcolm Slaney, *Fellow IEEE*, Yury Lifshits, and Junfeng He

ABSTRACT | Locality-sensitive hashing (LSH) is the basis of many algorithms that use a probabilistic approach to find nearest neighbors. We describe an algorithm for optimizing the parameters and use of LSH. Prior work ignores these issues or suggests a search for the best parameters. We start with two histograms: one that characterizes the distributions of distances to a point's nearest neighbors and the second that characterizes the distance between a query and any point in the data set. Given a desired performance level (the chance of finding the true nearest neighbor) and a simple computational cost model, we return the LSH parameters that allow an LSH index to meet the performance goal and have the minimum computational cost. We can also use this analysis to connect LSH to deterministic nearest-neighbor algorithms such as *k-d* trees and thus start to unify the two approaches.

KEYWORDS | Database index; information retrieval; locality-sensitive hashing; multimedia databases; nearest-neighbor search

I. INTRODUCTION

It seems like a simple problem: Given a set of data and a query, find the point in the data set that is nearest to the query. One calculates the distance between the query and each point in the database, and then returns the identity of the closest point. This is an important problem in many different domains, for example, query by example and multimedia fingerprinting [10]. We can find the nearest neighbors by checking the distance to every point in the data set, but the cost of this approach is prohibitive in all but the smallest data sets, and completely impractical in an interactive setting. We wish to find a solution, an index, whose cost is very, very sublinear in the number of points in the database.

The simplicity of the problem belies the computational complexity when we are talking about high-dimensional, web-scale collections. Most importantly, the curse of dimensionality [19] makes it hard to reason about the problem. Secondarily, the difficulty of the solution is complicated by the fact that there are two thoroughly researched and distinct approaches to the problem: probabilistic and deterministic. The stark difference in approaches makes them hard to compare. Later in this paper we will show that the two methods have much in common and suggest a common framework for further analysis.

The primary contribution of this paper is an algorithm to calculate the optimum parameters for nearest-neighbor search using a simple probabilistic approach known as locality-sensitive hashing (LSH), which we describe in Section II. The probabilistic nature of the data and the algorithm means that we can derive an estimate of the cost of a lookup. Cost is measured by the time it takes to perform a lookup, including cache and disk access. Our optimization procedure depends on the data's distribution, something we can compute from a sample of the data, and the application's performance requirements. Our approach is simple enough that we can use it to say a few things about the performance of deterministic indexing, and thus start to unify the approaches.

In conventional computer hashing, a set of objects (i.e., text strings) are inserted into a table using a function that converts the object into pseudorandom ID called a hash.

Manuscript received July 7, 2011; revised February 8, 2012; accepted March 18, 2012. Date of publication July 17, 2012; date of current version August 16, 2012.
M. Slaney was with Yahoo! Research, Sunnyvale, CA 94089 USA. He is now with Microsoft in Mountain View, CA (e-mail: malcolm@ieee.org).
Y. Lifshits was with Yahoo! Research, Sunnyvale, CA 94089 USA (e-mail: yury@yury.name).
J. He was with Yahoo! Research, Sunnyvale, CA 94089 USA. He is now with Columbia University, New York, NY 10027 USA (e-mail: hejunf@gmail.com).

Digital Object Identifier: 10.1109/JPROC.2012.2193849

 0018-9219/$31.00 ©2012 IEEE

Given the object, we compute an integer-valued hash, and then go to that location in an array to find the matches. This converts a search problem, which implies looking at a large number (N) of different possible objects and could take $O(N)$ or $O(\log(N))$ time, into a memory lookup with cost that could be $O(1)$. This is a striking example of the power of a conventional hash.

In the problems we care about, such as finding nearest neighbors in a multimedia data set, we do *not* have exact matches. Due to the vagaries of the real world, not to mention floating point errors, we are looking for matches that are close to the query. A conventional hash does not work because small differences between objects are magnified so the symbols "foo" and "fu" get sent to widely differing buckets. Some audio fingerprinting work, for example, chooses features that are robust so they can use exact indices to perform the lookup [25]. But in this work we consider queries that are more naturally expressed as a nearest-neighbor problem.

The curse of dimensionality makes designing an index for high-dimensional data a difficult proposition. Fig. 1 shows the normalized distance between any one point (a query) and all the other points in a uniform-random k-dimensional distribution. As the dimensionality increases the distribution becomes narrower, thus implying that all points are equal distance from all other points. This suggests that nearest-neighbor calculations are an ill-posed problem [3] since all points could be nearest neighbors. But in reality we are not concerned with *all* points, just those on the left tail of this distance distribution. With billions of points, we still have a significant number of neighbors even six standard deviations from the mean. Those are the points we want to find. Fortunately for this paper the world is *not* full of photographs with pixels or features from a uniform random distribution, making the nearest-neighbor question better behaved.

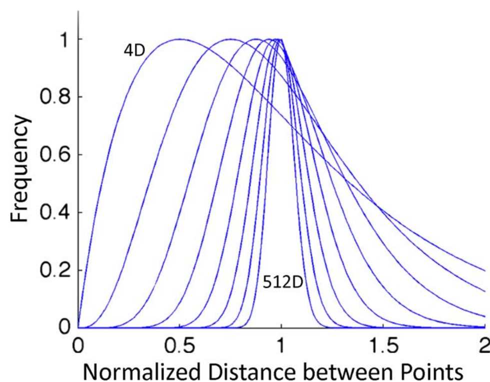

Fig. 1. ***Distance distributions versus data dimensionality. These curves show the likelihood of different (normalized) distances for D-dimensional $N(0, 1)$ data. The distribution becomes sharper as the data become higher dimensional (from 4 to 512 dimensions). To facilitate comparison, we normalize the distance data from each dimensionality to have a mean of one and normalize the peak level of the distributions.***

The focus of this work is on a common approach to finding nearest-neighbors known as LSH. However, there are several parameters to an LSH index and the current literature does not give a definitive statement about how to find the best parameter values. One can search for the best parameters by brute force trials, but these experiments are difficult to perform with large data sets. Dong's work [7] derives some of the expressions needed to optimize LSH, but this paper gives a more complete story. Our initial intuition that LSH would benefit from a large number of bins in each projection turned out to not be true (in all cases we have examined).

Given a query point, one can use LSH algorithms to find both its nearest and near neighbors. In this work, we concentrate on the theoretical and empirical properties of only one aspect of this problem: How do we choose the parameters of an LSH algorithm so that we minimize the cost of the search given a specification for the probability of success? If we get the parameters wrong we could do more work, even more than required for a linear search, or not even find the nearest neighbor. The specification can be as loose or tight as desired, and when we do not succeed we will either return no candidates, or a point that is not the absolute closest. We require an average-case guarantee—on average we will miss the absolute nearest neighbor with a (user-specified) probability of less than δ. Related problems, such as near neighbor, are also solved using the approaches described in this paper (see Section III-G).

A probabilistic approach has just one global value of error probability for all likely queries (as opposed to having separate errors for every query), and therefore it is easy to use as an optimization target. Our optimization algorithm only needs to know the size of data set n, an acceptable error probability δ, and the probability distribution functions (pdfs) d_{nn}, d_{any} for the distances between the query point and 1) its nearest neighbor or 2) any random member of the data set. Given this, we return the optimal parameters for an LSH implementation: the quantization width w, the number of projections k, and the number of repetitions or tables L. We also return the cost of searches over this database. Here, by optimal parameters we mean those corresponding to the absolute minimum search cost for LSH based on p-stable distributions for any data set with the given distance characteristics. We describe these parameters in detail in Section II.

Our main result suggests that the distance profile is a sufficient statistic to optimize LSH. Parameters recommended by our optimization remain valid for any data set with the same distance profile. Moreover, we can use an "upper bound" distribution as input to our optimization in order to cover a larger class of data sets (see Section III-A). Thus, we can be sure that dynamic changes of the data set (provided that they stay under the same distance distribution) will not disturb the behavior of LSH. This

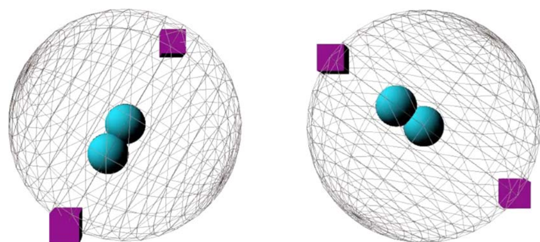

Fig. 2. *Projecting a 3-D data set onto the 2-D plane. Points that are close together (the spheres) stay close together, while points that are far apart (the cubes) are usually far apart after a projection.*

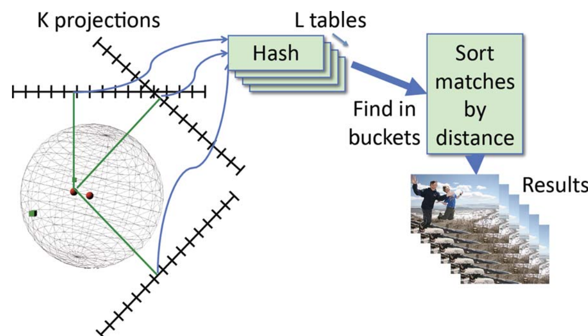

Fig. 3. *Indexing with LSH. Data points are projected k times and quantized. We store points into a bucket (or bin) labeled by a k-dimensional integer vector. This index is hashed with a conventional hash to convert it into a 1-D index. L tables are created in parallel to increase the probability of finding the nearest neighbor.*

stability property is important for practical implementations. Recommended LSH parameters are independent of the representational dimension. Thus, if one takes a 2-D data set and puts it in 100-dimensional space, our optimization algorithm will recommend the same parameters and provide the same search cost. This property was not previously observed for LSH.[1]

This paper describes an approach to find the optimal parameters for a nearest-neighbor search implemented with LSH. We describe LSH and introduce our parameters in Section II. Section III describes the parameter optimization algorithm and various simplifications. Section IV describes experiments we used to validate our results. We talk about related algorithms and deterministic variations such as *k-d* trees in Section V.

II. INTRODUCTION TO LOCALITY-SENSITIVE HASHING

In this work, we consider LSH based on *p*-stable distributions as introduced by Datar *et al.* [6]. Following the tutorial by Slaney and Casey [24], we analyze LSH as an algorithm for exact nearest-neighbor search. Our technique applies to the approximate near-neighbor algorithm as originally proposed by Datar *et al.* [6] and as we describe in Section III-A. We only consider distances computed using the Euclidean norm ($L2$). Below we present a short summary of the algorithm.

A. Basic LSH

LSH is based on a simple idea: After a linear projection and then assignment of points to a bucket via quantization, points that are nearby are more likely to fall in the same bucket than points that are farther away. Fig. 2 illustrates this idea for points in a 3-D space that are projected onto the (paper's) plane.

We use the following notation: D is the input data set; its points are denoted by $\bar{p}$ with various subscripts; $\bar{q}$ is the query point. LSH has three parameters: the quantization width w, the number of projections k, and the number of repetitions L. A Gaussian random vector $\bar{v}$ is a vector with every coordinate independently taken from the normal distribution $N(0,1)$. We pick a random shift value b is taken uniformly from the interval $[0, w)$.

Using LSH consists of two steps: indexing the data and searching for neighbors of a query point. The overall indexing algorithm is illustrated in Fig. 3.

Step 1: Indexing.

- Randomization: Select k random vectors $\bar{v}$ with the same dimensionality as the data, where each coordinate is a Gaussian random variable $N(0,1)$, and a scalar bias term b from a uniform random distribution between 0 and w.
- One-line projection: Take a point $\bar{p} \in D$, compute its dot product with the Gaussian random vector $\bar{v}$, and quantize the result with step w: $\lfloor (\bar{p} \cdot \bar{v} + b)/w \rfloor$. The bias term b does not affect our performance, but simplifies the analysis to follow because it ensures that the quantization noise is uncorrelated with the original data.
- Multiline projection: Obtain an array of k integers by doing k one-line projections. All points that project to the same k values are members of the same (k-dimensional) bin. At this stage, a conventional hash is used to reduce the k-dimensional bin identification vector to a location in memory. With a suitable design, this hash produces few collisions and does not affect our analysis.
- Repeat by hashing the data set to k-dimensional bins into a total of L times. Thus, we place every point in the dataset into L tables.

Step 2: Search.

1) Compute the L (k-dimensional) bins for the query point using the same random vectors and shift values as in the indexing stage.

[1]This also suggests that doing dimensionality reduction, using an algorithm such as principal components analysis (PCA), will harm the retrieval task only so far as the dimensionality-reduction step adds noise to the data set.

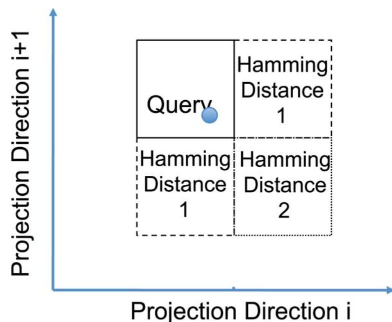

Fig. 4. *Hamming distance in quantized projection space. Quantized projections convert the original feature space into rectangular buckets in a k-dimensional space. Multiprobe looks in nearby buckets to find likely matches. Here we show the adjacent buckets in just two of the k dimensions.*

2) Retrieve *all* points that belong to all identified bins (we call them *candidates*), measure their distance to the query point, and return the one that is closest to query point.

B. Multiprobe

Multiprobe is a popular enhancement to LSH [18]. It is based on the observation that a query's nearest neighbors might not be in the query's bucket, and perhaps we should look in nearby buckets as well. There are many ways to find nearby buckets. For the analysis in this paper, we consider a simple approach based on the Hamming distance. This idea is shown in Fig. 4, where a query is shown after quantization in two of the k projection directions. For a Hamming distance of radius r, we look at all buckets that differ in at most r of the k projection coordinates. We only consider the side of the 1-D query bucket closest to the query.[2] Thus, there are C_k^r buckets at a Hamming distance of r from the query bucket.

We should note that choosing multiprobe bins by using the Hamming distance is suboptimal compared to measuring the exact distance between the query and the nearest corner of any other neighboring bin [18]. We simplify our analysis by using the Hamming distance and this provides guidance for the most common and useful case when $r = 1$.

Looking for data within a small Hamming radius, after projecting and binary quantization, is common in the computer-vision literature. Here we note that many researchers are doing a form of multiprobe, as we define it in this paper [13]. Salakhutdinov [23] uses a deep belief network to create a bit vector (with $k = 20$, 30, and 128 b) to represent an image. He then measures distance with a Hamming metric (for example, $r = 4$ or 5). Weiss [26] describes how to create optimal projections and splits using the graph Laplacian. Their graph encodes image similarity and they split the graph to minimize the cuts. He [11] and Jeǵou [15] look for projections that contain maximal information. All of these approaches use a Hamming metric to measure simularity and are analogous to LSH with just one table ($L = 1$). Similarly, a product quantizer [14] uses the product of k vector quantizers to define a bucket. Each quantizer only sees a subset of the overall vector dimensions and thus each quantizer is analogous to a trivial, axis-aligned projection.

C. Cost Model

We optimize retrieval performance, given the desired error requirement δ, using a simple computational model. For the search phase, we use unit costs U_{hash} and U_{check}. Here, U_{hash} is an upper bound (for any practical value of k) of the cost for computing the location of the bin corresponding to the query point, while U_{check} is a cost of computing a distance between a candidate point and a query point. We suppress the dependence on k here for two reasons: 1) in practice, arithmetic operations are cheap compared to memory lookups; and 2) it allows us to greatly simplify the choice of optimal parameters of LSH. Essentially, we just ignore the benefit that smaller k gives to query hashing, and optimize the total sum of all other costs.

Thus, the total costs of LSH are[3]

$$\text{search time: } LU_{\text{hash}} + |\text{candidates}|U_{\text{check}} \tag{1}$$

and the memory cost for the index is

$$\text{index space: } Ln. \tag{2}$$

D. Experimental Data

We use two different data sets to demonstrate the optimization calculations that follow and verify their performance. For all tests, the design specification (δ) required that the true nearest neighbor be retrieved with a probability of $1 - \delta = 0.5$.

Our real-world data are derived from a data set of more than 850 000 images of European cities. We analyzed the first 100 000 images with a convolutional deep-belief network (DBN) [12] that produced a 1024-dimensional feature vector for each image.[4] This feature vector performed well in an image-classification experiment, so it is a good data set for nearest-neighbor queries. We see similar performance with features based on audio modulation.

[2]This is a one-sided version of multiprobe. One could look on both sides of the query bucket and all that changes in our derivation is the Δ' function shown in Fig. 7. But with large bucket sizes, this is not necessary.

[3]Pauleve' uses the same cost function [22, eq. (3)].

[4]This R7 EC1M data set is available through Yahoo's WebScope program at http://webscope.sandbox.yahoo.com/catalog.php?datatype=r.

We also tested our LSH parameters using a synthetic data set over which we have control of the data's dimensionality. This is important because, as we will see, our simple model is good except when the number of projections is larger than the intrinsic dimensionality of the data.

We synthesized random data for our experiments by taking low-dimensional random data and then blowing it up to 1000 dimensions. Thus, our data had the distance properties of the low-dimensional subspace, but the same computational costs associated with longer vectors. To do this, we generated 100 000 Gaussian random vectors $N(0,1)$ with dimensionalities (d) of 10, 20, 30, and 40. We then multiplied these data by a $d \times 1000$ matrix to expand the original d-dimensional data into 1000 dimensions.

We used our Python LSH implementation in all these tests.[5] To estimate the U_{hash} and U_{check} parameters needed for the LSH optimization, we measured the total central processing unit (CPU) time needed to retrieve the nearest neighbors for a number of queries. We tested different parameter values, and thus had the CPU time as a function of k and L. Each experiment required different numbers of hashing (N_{hash}) and checking (N_{check}) operations and took a number of CPU seconds (T_{cpu}). We used a linear model, $N_{\text{hash}} U_{\text{hash}} + N_{\text{check}} U_{\text{check}} = T_{\text{cpu}}$, and estimate the unit the unit costs with linear regression. On a test machine, a large 2-GHz 64-b CPU, and with this implementation, we estimate U_{hash} is 0.4267 mS and U_{check} is 0.0723 mS. Since these times are in milliseconds, all the cost estimates in this paper are also in milliseconds. For each data set, we found the nearest neighbors for 5000 random queries. We could then estimate the d_{nn} and d_{any} histograms that form the basis of the parameter optimization.

AQ3

III. OPTIMIZATION ALGORITHM

The optimization algorithm $\text{OA}(n, \delta, d_{nn}, d_{\text{any}}, r)$ has five input parameters. Here n is the number of points in a data set and δ is the acceptable probability that we miss the exact nearest neighbor. The function (or an empirical histogram) d_{nn} is the pdf for the distance between any query point and its nearest neighbor. Similarly, d_{any} is the pdf for distance between the query point and any random point in the data set. Finally, r is the allowable Hamming radius for multiprobe.

Given the d_{nn} and d_{any} distance distributions, we compute the probabilities that a query point and the nearest neighbor or any point will hash to the same quantization interval (see Section III-B). These probabilities are a function of the proposed bin width w. From the ratio of these two probability functions we can then find the optimal value for the bin width w, because this width provides the maximum separation between true and false positives in our search. Given the size of the database, we know how many points will hash to each bucket and we select k so that on average each multiline projection returns approximately one candidate. We repeat this procedure L times so that we obtain the true nearest neighbor with the desired probability. The proof in Section III-D describes these steps in more detail, and a simplification that is easier to understand is shown in Section III-F. We start with some implementation notes in Section III-A and describe how to calculate collisions in Section III-B. These probabilities are the key to our derivation.

A. Distance Profiles Versus Task

Our derivations depend only on the two distance histograms. Commonly these histograms will be measured empirically. Starting with the distance profiles leads to two important points. First, everything we do is based on these distance profiles, and the points that are used to create these profiles define our task. We will make this claim more precise in Section III-G. Second, our performance guarantee δ is true for any data with distance histograms no worse than the ones we start with. We explain this latter idea in the remainder of this section. The performance of LSH depends on the contrast between our two distance profiles.[6] Moving the d_{nn} distribution to the right makes it harder for LSH to find the nearest neighbors because the nearest neighbors are farther from the query. Similarly, moving the d_{any} profile to the left makes LSH work harder because there will be more "any" (false positive) neighbors returned by a search. Thus, we can define more pessimistic upper and lower distance profiles. The search for nearest neighbors will be suboptimal, but the probability of returning the true nearest neighbor will be above $1 - \delta$ when averaged over all queries.

We can formalize left and right movements as follows. In order to get stronger accuracy guarantees one can use "lower bound" d_{nn}^{l} and "upper bound" d_{any}^{u} distributions instead of sampled histograms. Here, by lower bound, we mean that for any x, the integral of d_{nn}^{l} on $[0, x]$ is a lower bound for the integral of d_{nn} over the same interval. The upper bound is defined in the opposite way. These two distributions represent pessimistic estimates of the two distributions.

With large databases we can easily subsample the data and calculate d_{any} for two random points in the smaller data set. Unfortunately, the calculation for d_{nn} is not as easy. Calculating d_{nn} requires finding the true nearest neighbor and is most easily accomplished via a brute-force search. Subsampling the data set over which we search for the nearest neighbor leads to a pessimistic estimate of the nearest-neighbor distribution. This is shown in Fig. 5.

[5] This Python LSH implementation and the MATLAB code to calculate the optimum parameters are available via GitHub: https://github.com/yahoo/Optimal-LSH.

[6] Kumar formalized this in a nice way [17].

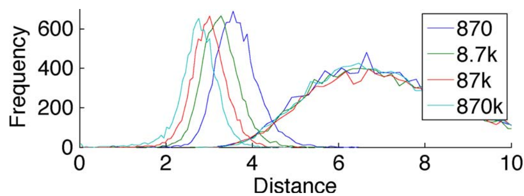

Fig. 5. ***Distance estimates versus sample size for the EC1M data set. The nearest-neighbor distance estimates (left set of curves) shift to the left as larger portions of the data set are used to estimate the nearest-neighbor distribution. The any-neighbor distribution is an unbiased estimate no matter what sample size is used. There are 870 000 images in the full data set.***

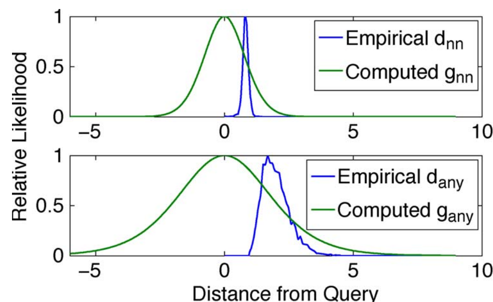

Fig. 6. ***Projection histograms for the EC1M image data set. The narrow, jagged curves are the raw distance histograms, while the smooth curves are the histograms of projection locations.***

Fortunately, the parameter construction process guarantees that we will meet the performance requirements (δ), but we might have suboptimal computational effort.

Note that query sampling allows us to compute highly accurate and tight upper and lower bounds for the actual distributions. Also, using an upper/lower bound ensures that rounding errors in the functional transformations we use in our algorithm are not breaking accuracy guarantees. Finally, bounds on the actual distributions can be used in the case when the data distribution is not uniform. In this paper, we optimize the overall probabilistic behavior, but if the data or query distribution is dense in one portion of the space then one might want to use a specially crafted distance distribution to model the inhomogenous distribution.

B. Collision Probabilities

Before deriving the optimal parameters we first derive expressions for the collision probabilities due to projections. We want to know how likely it is that the true nearest neighbor and any other data point are in the same bucket as the query point. This calculation models the first stages of the LSH algorithm: a dot product with a Gaussian random vector, followed by scaling and quantization. We can then combine k projections into a single table. From these probabilities, we estimate the cost of an LSH lookup, and then optimize the remaining parameters (Section III-C).

Our derivation starts with the histograms of the distance between a query and 1) its nearest neighbor and 2) any neighbor in the database. Fig. 5 shows the distance probabilities for our image data set and these two distributions are well separated. But recall that the any-neighbor distribution represents n^2 points, so there are still a significant number of points, including all the nearest neighbors, in the left tail of this distribution.

It is known [27] that the normal distribution has the following 2-stability property:

$$\text{pdf } \bar{p} \cdot \bar{v} = |p| N(0,1). \tag{3}$$

In other words, the pdf of a dot product between a fixed vector $\bar{p}$ and a Gaussian random vector (zero mean, unit variance) is equal to the pdf of the normal distribution magnified by the norm of input vector $|\bar{p}|$. We now apply this property not to a single vector $|\bar{p}|$, but to the pdf for the difference vectors $\bar{q} - \bar{p}$ and $\bar{q} - \bar{p}_{nn}(\bar{q})$. Recall that the pdfs of norms of the vectors d_{nn}, d_{any} are the input of our optimization algorithm. Thus, we can compute pdfs g_{nn}, g_{any} of dot products between $\bar{q} - \bar{p}$ and random Gaussian vector by doing pdf multiplications between d_{nn}, d_{any} and $N(0,1)$, as shown below [9][7]

$$g_{nn}(y) = \int N(0,1)(x) d_{nn}\left(\frac{y}{x}\right) \frac{1}{|x|}\, dx \tag{4}$$

$$g_{\text{any}}(y) = \int N(0,1)(x) d_{\text{any}}\left(\frac{y}{x}\right) \frac{1}{|x|}\, dx. \tag{5}$$

By way of illustration, Fig. 6 shows the probabilities that the nearest neighbor and any neighbor points project to a given normalized distance. Note that there is much overlap in the distributions. We next normalize the distances by w so that all points with the same integer value will be in the same single-line bucket.

Given the difference value $(\bar{p} \cdot \bar{v} - \bar{q} \cdot \bar{v})$, the probability that p and q are quantized to the *same* value is computed using a triangle function $\Delta_w(x)$, as shown in Fig. 7. This works because the bias random variable b ensures that the quantization boundary is randomly positioned with respect to the two points. When the two points, after projection, have zero distance, then they will always fall in the same bin. If their distance is 1 or more, then they can never fall in the same bin. Thus, the Δ function increases linearly from 0 to 1 on the interval $[-w, 0]$ and decreases symmetrically on $[0, w]$. In order to compute the collision

[7]Datar [6] used pdf multiplication to show that his hash functions are locally sensitive.

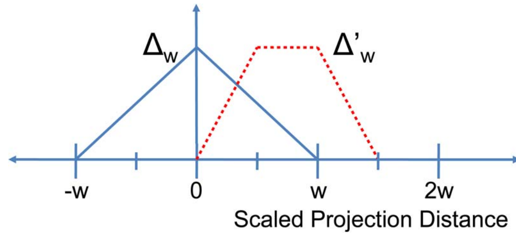

Fig. 7. ***These windows give bucket probabilities as a function of distance. The triangle is $\Delta_w(x)$ and describes the probability that a data point at a distance of x is in the same bucket as the query. The trapezoid is $\Delta'_w(x)$ and describes the same probability for the closest adjacent bucket, given that we know the query is in the right half of its bucket.***

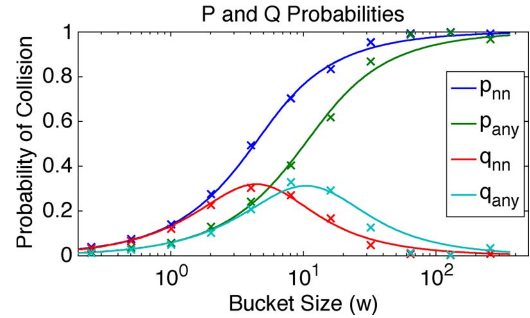

Fig. 8. ***Bucket collision probabilities as a function of bin width for the EC1M data set of photos. All probabilities start at 0, since the collision probabilities are infinitesimally small with small buckets. The two sigmoid curves show the probabilities for the query's bucket. The two unimodal curves show the collision probability for the data in the adjacent bin. The left curve in each set is the probability of finding the nearest neighbor, and the right curve is any other point.***

probabilities for a nearest-neighbor $\bar{p}_{nn}$ and average $\bar{p}_{\text{any}}$ point we should integrate the product of g and Δ

$$p_{nn}(w) = \int g_{nn}(x)\Delta_w(x)\,dx \tag{6}$$

$$p_{\text{any}}(w) = \int g_{\text{any}}(x)\Delta_w(x)\,dx. \tag{7}$$

A similar collision calculation applies for multiprobe. Fig. 7 also shows the Δ' function that corresponds to the simplest multiprobe case, a one-sided search. The query point falls in the right half of its (1-D) bucket and we want to know the probability that a candidate point with the given distance falls in the next bucket to the right.

This probability distribution starts at 0 for zero distance because the query and the candidate are just separated by the boundary. For distances between 0.5 and 1.0, the probability is equal to 1.0 because at those distances the candidate can only be in the adjacent bin. The probability goes to zero as the distance approaches 1.5 because at these distances the candidate point is likely to be two buckets away from the query's bucket. Note that the calculation is symmetric. We show Δ' for the right-hand side adjacent bin, which is used when the query is on the right-hand side of its bucket, but the same equation holds on the left.

Thus, the probability that the query and its nearest neighbor fall into adjacent bins is

$$q_{nn}(w) = \int g_{nn}(x)\Delta'_w(x)\,dx. \tag{8}$$

Note that we use q, instead of p, to represent the probabilities of falling in the *adjacent* bin. Similarly, the probability that the query and any other point fall into adjacent bin is

$$q_{\text{any}}(w) = \int g_{\text{any}}(x)\Delta'_w(x)\,dx. \tag{9}$$

Fig. 8 shows the critical distributions that are the basis of our optimization algorithm. We show both theoretical (solid lines) and experimental (marked points) results for the EC1M image data set. We calculate the distance profiles empirically, as shown in Fig. 5, and from these profiles we use (6) and (7) to compute the expected collision probabilities p_{nn} and p_{any}. Both these probabilities are functions of w. We use a similar process for the q probabilities.

We implemented the LSH algorithm described here using a combination of Python and NumPy. We then built LSH indices with a wide range of quantization intervals (w) and measured the system's performance. The discrete points are empirical measurements of collision probabilities (with $k = 1$ and $L = 1$). The measured distance profiles and probability calculations do a good job of predicting the empirical probabilities shown in Fig. 8.

C. Combining Projections

Our model assumes that each projection is independent of the others. Thus, we can combine the probabilities of different projections and the probes of adjacent bins. The probability that a query and its nearest neighbor fall into the *same* k-dimensional bucket is equal to

$$p_{nn}(w,k) = p_{nn}^k(w). \tag{10}$$

Then, the probability that two points have a Hamming distance $\leq r$ in one table is

$$p_{nn}(w,k,r) = p_{nn}^k(w) + k p_{nn}^{k-1}(w) q_{nn}(w) + \cdots + C_k^r p_{nn}^{k-r}(w) q_{nn}^r(w). \tag{11}$$

In practice, we only consider small r, like $r = 0, 1, 2$. Since $r \ll k$, we note

$$p_{nn}^k(w) \ll k p_{nn}^{k-1}(w) q_{nn}(w) \ll C_k^r p_{nn}^{k-r}(w) q_{nn}^r(w) \quad (12)$$

and hence we only need the last term of (11)

$$p_{nn}(w,k,r) \approx C_k^r p_{nn}^{k-r}(w) q_{nn}^r(w). \quad (13)$$

Similarly, the probability that any two points have a Hamming distance $\leq r$ in one table is

$$p_{\text{any}}(w,k,r) = p_{\text{any}}^k(w) + k p_{\text{any}}^{k-1}(w) q_{\text{any}}(w) + \cdots + C_k^r p_{\text{any}}^{k-r}(w) q_{\text{any}}^r(w) \quad (14)$$

and we approximate p_{any} by

$$p_{\text{any}}(w,k,r) \approx C_k^r p_{\text{any}}^{k-r}(w) q_{\text{any}}^r(w). \quad (15)$$

From the original distance data, we have calculated the collision probabilities. We can now derive the optimal LSH parameters.

D. Theoretical Analysis

Theorem 1: Let d_{nn}, d_{any} be the pdf for $|\bar{q} - \bar{p}|$ distances for some data set distribution $\mathcal{D}$ and a query distribution $\mathcal{Q}$. Then, the following statements are true.

1) LSH with parameters OA.parameters $(n, \delta, d_{nn}, d_{\text{any}}, r)$ works in expected time OA.search_cost$(n, \delta, d_{nn}, d_{\text{any}}, r)$ time.
2) It returns the exact nearest neighbor with probability at least $1 - \delta$.
3) Parameter selection is optimal for any data set/query distributions with the given distance profile (d_{nn}, d_{any}).

Proof: We work backwards to derive the optimal parameters. We predict the optimal number of tables L by hypothesizing values of w and k. Given an estimate of L, we find the k that minimizes the search time cost. We then plug our estimate of k and L to find the minimum search cost and thus find the best w.

1) Optimizing L: Given the collision probabilities as functions of w, we examine the steps of the optimization algorithm in reverse order. Assume that parameters k and w are already selected. Then, we have to select a minimal L that puts the error probability below δ. Given w and k, the probability that we will do all these projections and still miss the nearest neighbor is equal to

$$(1 - p_{nn}(w,k,r))^L \leq \delta$$
$$\log(1 - p_{nn}(w,k,r))L \leq \log(\delta). \quad (16)$$

Let us use the approximation $\log(1 - z) \approx -z$ here for $(1 - p_{nn}(w,k,r))$; this is fair since in real settings $p_{nn}(w,k,r)$ is of order of $1/n$ [see (46)]. Thus, once we find k and w, we should use (13) to find

$$L_{\text{opt}} = \frac{-\log\delta}{p_{nn}(w,k,r)} \approx \frac{-\log\delta}{C_k^r p_{nn}^{k-r}(w) q_{nn}^r(w)}. \quad (17)$$

This is optimum, given the values of w, k, and r. Using fewer tables will reduce the computational effort but will not meet the error guarantee. Using more tables will further (unnecessarily) reduce the likelihood of missing the true nearest neighbor, below δ, and create more work.

2) Search Cost: The time to search an LSH database for one query's nearest neighbors is[8]

$$T_s = U_{\text{hash}} L + U_{\text{check}} L n p_{\text{any}}(w,k,r). \quad (18)$$

With this computational model, the search time is

$$T_s = U_{\text{hash}} \frac{-\log\delta}{C_k^r p_{nn}^{k-r}(w) q_{nn}^r(w)} + U_{\text{check}} n(-\log\delta) \frac{C_k^r p_{\text{any}}^{k-r}(w) q_{\text{any}}^r(w)}{C_k^r p_{nn}^{k-r}(w) q_{nn}^r(w)}. \quad (19)$$

We call this our exact cost model because the only approximations are the independence assumption in (10), which is hard not to do, and (12), which is true in all cases we have seen. One could optimize LSH by performing a brute-force search for the parameters w and k that minimize this expression. These calculations are relatively inexpensive because we have abstracted the data into the two distance histograms.

We can simplify this cost function to derive explicit expressions for the LSH parameters. We note $r \ll k$ and

[8] A more accurate estimate of the search time is: $T_s = U_{\text{hash}} L + U_{\text{bin}} C_k^r L + U_{\text{check}} L n p_{\text{any}}(w,k,r)$ where U_{bin} is the cost to locate one bin. In practice, we only use small r, like $r = 0, 1, 2$. Moreover, U_{bin} is usually much smaller than U_{hash}. In practice, U_{hash} consists of tens of inner products for a high-dimensional feature vector, while U_{bin} is just a memory location operation. So we can assume $U_{\text{bin}} C_k^r \ll U_{\text{hash}}$. And hence, search time can still be approximated as $U_{\text{hash}} L + U_{\text{check}} L n p_{\text{any}}(w,k,r)$.

$C_k^r \approx k^r/r!$. Furthermore, we define the following two expressions (to allow us to isolate terms with k):

$$C_h = U_{\text{hash}}(-\log\delta)r!\left[\frac{p_{nn}(w)}{q_{nn}(w)}\right]^r \tag{20}$$

and

$$C_c = U_{\text{check}}n(-\log\delta)\left[\frac{p_{nn}(w)}{q_{nn}(w)}\frac{q_{\text{any}}(w)}{p_{\text{any}}(w)}\right]^r. \tag{21}$$

Now the search time can be rewritten as

$$T_s = \frac{C_h}{k^r p_{nn}^k(w)} + C_c\left(\frac{p_{\text{any}}(w)}{p_{nn}(w)}\right)^k. \tag{22}$$

3) Optimizing k: The above expression is a sum of two strictly monotonic functions of k: the first term increases with k since $p_{nn}(w)$ is less than one, while the second is an exponential with a base less than one since p_{any} is always smaller than p_{nn}. It is difficult to find the exact optimum for this expression. Instead, we find the value for k such that the costs for k and $k+1$ are equal—the minimum will be in between. That is, we need to solve the equation

$$\frac{C_h}{k^r p_{nn}^k(w)} + C_c\left(\frac{p_{\text{any}}(w)}{p_{nn}(w)}\right)^k = \frac{C_h}{(k+1)^r p_{nn}^{k+1}(w)} + C_c\left(\frac{p_{\text{any}}(w)}{p_{nn}(w)}\right)^{k+1}. \tag{23}$$

Note that $((k+1)^r/k^r) = [(1+1/k)^k]^{(r/k)} \le e^{r/k}$ and since $r \ll k$, then $e^{r/k}$ is very close to 1. So

$$\begin{aligned}\frac{C_h}{k^r p_{nn}^k(w)} + C_c\left(\frac{p_{\text{any}}(w)}{p_{nn}(w)}\right)^k &= \frac{C_h}{(k+1)^r p_{nn}^{k+1}(w)} + C_c\left(\frac{p_{\text{any}}(w)}{p_{nn}(w)}\right)^{k+1} \\ &\approx \frac{C_h}{k^r p_{nn}^{k+1}(w)} + C_c\left(\frac{p_{\text{any}}(w)}{p_{nn}(w)}\right)^{k+1}\end{aligned} \tag{24}$$

or by multiplying both sides by $k^r p_{nn}^{k+1}(w)$

$$C_h p_{nn}(w) + C_c k^r p_{\text{any}}^k(w) p_{nn}(w) = C_h + C_c p_{\text{any}}^{k+1}(w) k^r. \tag{25}$$

With simplification, we have

$$C_c p_{\text{any}}^k(w) k^r\left[p_{nn}(w) - p_{\text{any}}(w)\right] = C_h(1 - p_{nn}(w)) \tag{26}$$

or in other words

$$p_{\text{any}}^k(w) = \frac{C_h(1-p_{nn}(w))}{C_c k^r\left(p_{nn}(w) - p_{\text{any}}(w)\right)}. \tag{27}$$

Putting the definition of C_c and C_h into the above, we have

$$\begin{aligned}p_{\text{any}}^k(w) &= \frac{C_h(1-p_{nn}(w))}{C_c k^r\left(p_{nn}(w) - p_{\text{any}}(w)\right)} \\ &= \frac{\frac{U_{\text{hash}}}{U_{\text{check}}}(1-p_{nn}(w))r!}{k^r\left(p_{nn}(w) - p_{\text{any}}(w)\right)n\left[\frac{q_{\text{any}}(w)}{p_{\text{any}}(w)}\right]^r}.\end{aligned} \tag{28}$$

Thus, the optimum k is defined by the fixed point equation

$$k = \frac{\log n + r\log k + \alpha(w)}{-\log p_{\text{any}}(w)} \tag{29}$$

where

$$\begin{aligned}\alpha(w) = \log\frac{\left(p_{nn}(w) - p_{\text{any}}(w)\right)}{1 - p_{nn}(w)} + r\log\frac{q_{\text{any}}(w)}{p_{\text{any}}(w)} \\ + \log\frac{U_{\text{check}}}{U_{\text{hash}}} - \log(r!).\end{aligned} \tag{30}$$

We wish to find a value for k that makes both sides of this equation equal. First, define $k_0 = (\log n + \alpha(w))/-\log p_{\text{any}}(w)$ so we can express k as a linear function of r

$$k \approx k_0 + r - \log p_{\text{any}}(w)\log(k_0) = k_0 + r'\log(k_0). \tag{31}$$

Putting this approximation into both sides of (29) gives us $k_0 + r'\log(k_0 + r'\log(k_0)) = k_0 + r'\log(k_0) + r'\log(1 + r'(\log(k_0)/k_0))$. The difference between the left- and right-hand sides is $-r'\log(1 + r'(\log(k_0)/k_0))$, which is small and close to 0. So $k = k_0 + r'\log(k_0)$ is a good approximate solution for (29).

The optimum value for k is

$$k_{\text{opt}} \approx \frac{\log n + \alpha(w)}{-\log p_{\text{any}}(w)} + r' \log\left(\frac{\log n + \alpha(w)}{-\log p_{\text{any}}(w)}\right). \quad (32)$$

4) Optimizing w: To simplify the notation in the following derivation we omit the dependence of the projection densities on w and write p_{nn} and p_{any} instead of $p_{nn}(w)$ and $p_{\text{any}}(w)$.

Now that we know how to select the optimal k and L for a given w, we rewrite the whole cost function as a function of w and select w_{opt} as the argmin of the resulting expression. Putting (34) into (22), the search time is

$$T_s = \left[\frac{C_h}{k^r} + C_c (p_{\text{any}})^k\right] p_{nn}^{-k}. \quad (33)$$

We do not have an explicit formula for p_{nn}^k, but from (28), we know the value of p_{any}^k in terms of algorithm parameters (U_{check}, etc.) and our distributional measurements (p_{nn}, etc.). Thus, we can convert p_{nn}^k to p_{any}^k by noting that $x = y^{\log x / \log y}$, therefore

$$p_{nn}^{-k} = \left[p_{\text{any}}^{-k}\right]^{\frac{\log p_{nn}}{\log p_{\text{any}}}} \quad (34)$$

$$T_s = \left[\frac{C_h}{k^r} + C_c (p_{\text{any}})^k\right] \left[p_{\text{any}}^{-k}\right]^{\frac{\log p_{nn}}{\log p_{\text{any}}}}. \quad (35)$$

Using expressions for p_{any}^k (28) and C_h and C_c [(20) and (21)], we find the following expression for T_s:

$$T_s = \left\{(-\log \delta) U_{\text{hash}} r! \left[\frac{p_{nn}}{q_{nn}}\right]^r \frac{1}{k^r} \frac{(1 - p_{\text{any}})}{(p_{nn} - p_{\text{any}})}\right\} \times \left\{\frac{k^r (p_{nn} - p_{\text{any}}) n U_{\text{check}} \left[\frac{q_{\text{any}}}{p_{\text{any}}}\right]^r r!}{(1 - p_{nn}) U_{\text{hash}}}\right\}^{\frac{\log p_{nn}}{\log p_{\text{any}}}}. \quad (36)$$

To minimize the LSH search time, we should find w to minimize (35)

$$w_{\text{opt}} = \arg\min[T_s]. \quad (37)$$

To find the optimum value, we evaluate (37) using the equation for k (32) over a range of values. We choose the w that gives the minimum CPU time. This is a relatively trivial computation compared to the amount of effort required to find the d_{nn} distribution that is the key to our optimization.

Given a description of the data (p_{nn} and p_{any}) and choices for w and k, we know how likely we are to find the true nearest neighbor. We have to check enough tables to make sure we meet the performance guarantee. Thus, we use w_{opt} (37) and k_{opt} (32) to compute L_{opt} by

$$L_{\text{opt}} = \left\lceil \frac{-\log \delta}{C_{k_{\text{opt}}}^r p_{nn}^{k_{\text{opt}} - r}(w_{\text{opt}}) q_{nn}^r (w_{\text{opt}})} \right\rceil. \quad (38)$$

L should be rounded up, otherwise we will not meet our performance requirements (δ).

5) Search Points: Using (15) and (28), the average number of points retrieved in one hash table is

$$\begin{aligned} n p_{\text{any}}(w, k, r) &\approx n C_k^r p_{\text{any}}^{k-r}(w) q_{\text{any}}^r(w) \\ &= \left[\frac{q_{\text{any}}(w)}{p_{\text{any}}(w)}\right]^r n \frac{k^r}{r!} p_{\text{any}}^k(w) \\ &= \frac{U_{\text{hash}}}{U_{\text{check}}} \frac{(1 - p_{nn}(w))}{(p_{nn}(w) - p_{\text{any}}(w))}. \end{aligned} \quad (39)$$

Note that this result is independent of k and r. In other words, for all Hamming distances r, the optimum multiprobe LSH chooses k such that the above equation is satisfied.

Furthermore, as the cost of checking points (U_{check}) goes up, the number of points per returned bucket goes down. Similarly, a larger p_{nn} decreases $(1 - p_{nn}(w))/(p_{nn}(w) - p_{\text{any}}(w))$ and hence decreases the number of returned points. We further note that the ratio of probabilities in (39) is often not far from 1, so the number of points in a bucket is determined by $U_{\text{hash}}/U_{\text{check}}$. As the relative cost of U_{check} goes up, the optimum number of points per bucket goes down, so (hopefully) we spend less time checking candidates.

Our model includes two parameters that describe the cost of major operations in the LSH process. Since both costs are measures of time, it is only their relative value ($U_{\text{check}}/U_{\text{hash}}$) that matters to the parameter calculation. Fig. 9 gives one a feel for their effect. The value for w is not affected by these costs. The optimum number of projections (k) increases as the cost of checking candidates increases, to better weed out false positives. The number of tables (L) must then also increase to make sure we meet the performance goal (δ).

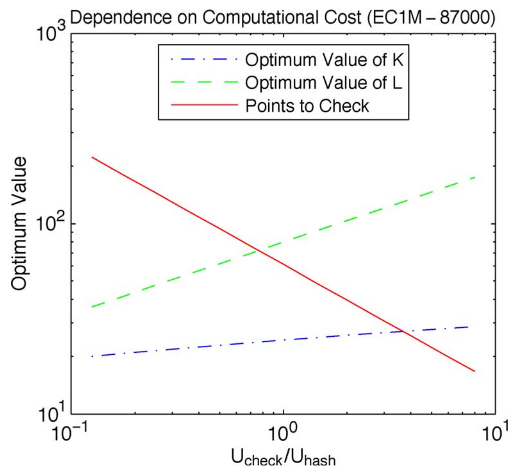

Fig. 9. ***Changes in LSH parameters as costs change. These three curves show how the optimal parameters changes as the ratio of costs due to checking candidates increases over the cost of computing the hash.***

The costs of a search and the space needed for the optimum index are given by

$$\text{search cost}_{\text{opt}} = L_{\text{opt}}\left(U_{\text{hash}} + np_{\text{any}}^{k_{\text{opt}}}(w_{\text{opt}})U_{\text{check}}\right) \tag{40}$$

$$\text{index space}_{\text{opt}} = L_{\text{opt}}n. \tag{41}$$

6) Summary: Wrapping up our proof, the functional transformations turn the distance-profile information into collision probabilities as a function of quantization threshold w. We presented formulas for computing optimal L for any k and w (17) and for computing the optimal k for any given w (32). Then, we rewrite the search cost as a function of the single argument w (37). We walk these steps in the reverse order to implement this algorithm. We select w_{opt} as the argmin of the search cost, then we get $k_{\text{opt}} = k_{\text{opt}}(w_{\text{opt}})$ and $L_{\text{opt}} = L_{\text{opt}}(k_{\text{opt}}, w_{\text{opt}})$. Our error probability is less than δ by the construction of L and our choice of parameters is optimal due to the argmin choice of w and conditional optimality of other parameters. We do not have an expression for the optimal r. But since the best r is small, it is easy to pick an r and evaluate the cost.

Fig. 10 shows typical results for real and synthetic data. While the cost and performance vary over greatly as we vary the bucket width w, the optimization algorithm guarantees that LSH algorithm will meet the performance guarantee (δ). We explain the discrepancies between the predictions and the experimental data in Section IV-A2.

E. Without Multiprobe

For completeness, we show here the formulas for the optimum parameters without multiprobe ($r = 0$). The expression for p_{any}^k (27) becomes

$$p_{\text{any}}^k = \frac{U_{\text{hash}}(1 - p_{nn})}{U_{\text{check}}n(p_{nn} - p_{\text{any}})}. \tag{42}$$

We can then rewrite this to find the optimum for k without multiprobe

$$k_{\text{opt}'} = \frac{\log n + \log\left(p_{nn}(w) - p_{\text{any}}(w)\right)}{-\log p_{\text{any}}(w)} - \frac{\log(1 - p_{nn}(w)) + \log U_{\text{check}} - \log U_{\text{hash}}}{-\log p_{\text{any}}(w)}. \tag{43}$$

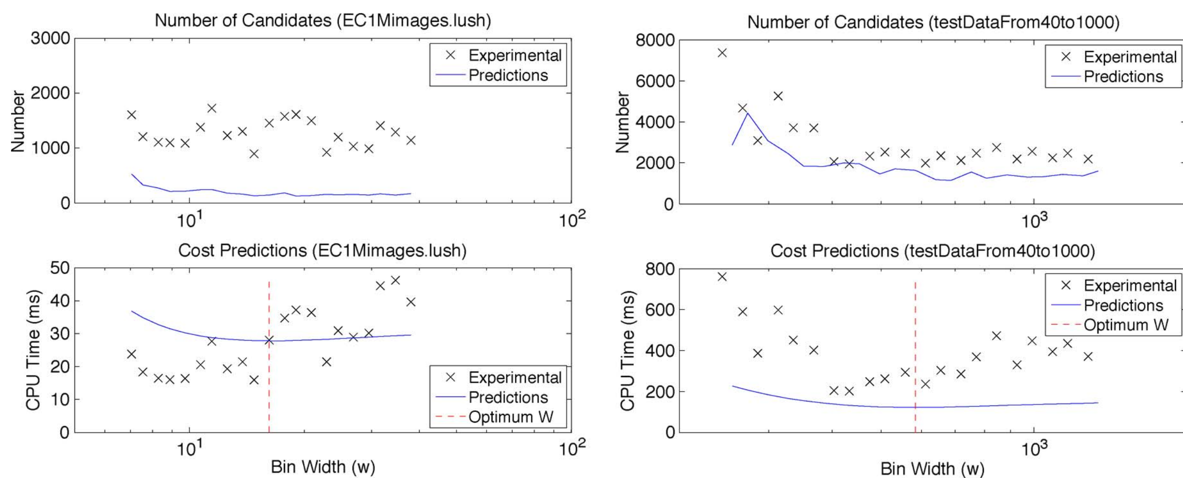

Fig. 10. ***Number of candidates to check (top) and search time (bottom) for LSH indices built for the EC1M (left) and the synthetic 40-dimensional data sets (right). For each potential bucket width*** (w)***, we find the optimum values of k and L, run an experiment with those parameters, and then measure the performance of the LSH index. The solid lines are the predictions based on the "exact" optimization parameters, while the markers show the measured performance.***

The equation for the computational time T_s (36) becomes

$$T_{s'} = (-\log\delta)U_{\text{hash}}\frac{(1-p_{\text{any}})}{(p_{nn}-p_{\text{any}})} \times \left\{\frac{(p_{nn}-p_{\text{any}})nU_{\text{check}}}{(1-p_{nn})U_{\text{hash}}}\right\}^{\frac{\log p_{nn}}{\log p_{\text{any}}}}. \quad (44)$$

We find the $w_{\text{opt}'}$ that minimizes this equation. The equation for L_{opt} (38) without multiprobe becomes

$$L_{\text{opt}'} = \left\lceil\frac{-\log\delta}{p_{nn}^k(w)}\right\rceil. \quad (45)$$

F. Simplified Parameter Selection

It is instructive to further simplify our estimates of the optimal parameter choices. We approximate the optimal k by keeping only the first argument in the first numerator of (43) to find

$$k_{\text{simp}} = \frac{\log n}{-\log p_{\text{any}}(w_{\text{simp}})}. \quad (46)$$

We perform this simplification since 1) we anticipate the $\log n$ term to be the largest factor in the numerator; and 2) this change allows great simplifications in further steps of the algorithm. The probability $p_{\text{any}}(w_{\text{simp}})$ is often close to 0.5 so in effect we partition the space with $\log n$ binary splits. Thus, each bucket contains on average one point.

We just make a change of notation in (45) to write this expression for the simplified L

$$L_{\text{simp}} = \frac{-\log\delta}{p_{nn}^{k_{\text{simp}}}(w_{\text{simp}})}. \quad (47)$$

Now we want to find a simplified expression for w. It is easiest to start with (22) and set $r=0$. We drop the dependence on δ because it does not affect the argmin, and note that the simplified formula for k (46) implies the equality $p_{\text{any}}^{-k} = n$. Therefore

$$\begin{aligned} w_{\text{simp}} &= \arg\min\frac{U_{\text{hash}} + U_{\text{check}}np_{\text{any}}^k}{p_{nn}^k} \\ &= \arg\min\frac{U_{\text{hash}} + U_{\text{check}}}{p_{nn}^k} \\ &= \arg\min(U_{\text{hash}} + U_{\text{check}})\left(p_{\text{any}}^{-k}\right)^{\log p_{nn}/\log p_{\text{any}}} \\ &= \arg\min(U_{\text{hash}} + U_{\text{check}})n^{\log p_{nn}/\log p_{\text{any}}}. \end{aligned} \quad (48)$$

Thus

$$w_{\text{simp}} = \arg\min(\log p_{nn}/\log p_{\text{any}}) \quad (49)$$

is the optimal choice for w in this case. This is an interesting choice because we are essentially choosing a w that maximizes the contrast, in some sense, between the two collision curves. Gionis [8] found a similar expression for a lower bound on the probability of retrieving the nearest neighbor (p_1^k), and we can maximize performance making his lower bound as large as possible.

The simplified costs of a search (SC) and the space needed for the index (IS) are given by

$$\begin{aligned} \text{SC}_{\text{simp}} &= L_{\text{simp}}\left(U_{\text{hash}} + np_{\text{any}}^{k_{\text{simp}}}(w_{\text{simp}})U_{\text{check}}\right) \\ &= (U_{\text{hash}} + U_{\text{check}})n^{\log p_{nn}(w_{\text{simp}})/\log p_{\text{any}}(w_{\text{simp}})} \end{aligned} \quad (50)$$

$$\text{IS}_{\text{simp}} = L_{\text{simp}}n. \quad (51)$$

There are several things we can say about these simplified parameter choices. First, we choose a w that minimized the difference between the logs of p_{nn} and p_{any}. This makes sense since the quantization step means we make a decision, thus throwing away information, and we want to pick a w that gives us the best performance on the basic task. Furthermore, we choose k_{simp} so that we have on average one point in each bucket after doing the k projections. Finally, L_{simp} is chosen so that we find the true nearest neighbor with the desired δ error rate.

G. Other Forms of LSH

The derivation we present here is valid for many different interpretations of d_{nn} and d_{any}. Here we describe three different variations of the nearest-neighbor problem: an arbitrary neighbor, the first n neighbors, and near neighbors. Instead of the nearest neighbor, we can calculate the distance, for example, to the 42nd closest neighbor. The parameters we derive allow LSH to find the desired neighboring point, in this case the 42nd nearest neighbor, with a probability at least $1-\delta$. LSH implemented with these parameters is likely to also return the 1st through 41st points, but our estimates of the computational costs are accurate because these points are part of the d_{any} distribution used to calculate the search cost. Similarly, we can measure and use the distance to 1000 of a query's nearest neighbors. Using the parameters we derive here, we can guarantee that LSH will return each of the first 1000 nearest neighbors with probability at least $1-\delta$. Finally, the near-neighbor problem is often specified as finding a neighbor within a distance $1+\epsilon$ of the true nearest neighbor. Our derivation still applies if d_{nn} is based on all the points with distance to the query less

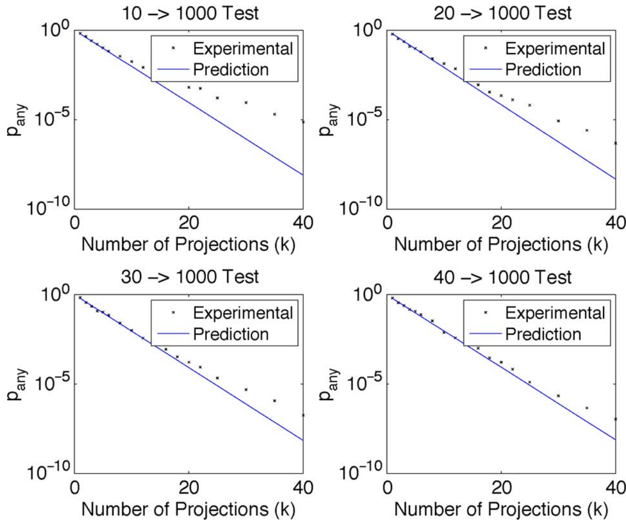

Fig. 11. ***Probability of finding any neighbor. These four panels show the probability of finding the true nearest neighbor as a function of k in four different synthetic data sets. Note the relative magnitude of the discrepancy between theory and experiments for large k.***

than $d_{\text{true}}(1+\epsilon)$ where d_{true} is the distance to the (empirically measured) nearest neighbor.

More precisely, the distance distribution $d_{nn}(x)$ is a conditional probability

$$d_{nn}(x) = P(\text{distance}(q,p) = x | p \text{ belongs to } q\text{'s nearest-neighbor subset}). \quad (52)$$

By this means, we can define a query's nearest-neighbor subset in many different ways, and our theory will always hold. In all cases, the performance guarantee becomes: we guarantee to return any point $\bar{p}$ in $\bar{q}$'s "nearest-neighbor" subset with a probability of at least $1-\delta$. Finally, to be clear, we note that d_{nn}, g_{nn}, and p_{nn} are all conditional probabilities, depending on the definition of the nearest-neighbor subset.

IV. EXPERIMENTS

We describe two types of experiments in this section: validation and extensions. We first show that our theory matches reality by testing an LSH implementation and measuring its performance. This validates our optimization approach in several different ways. Most importantly, we show that we can predict collisions and the number of points returned. We also show how multiprobe affects performance. We use synthetic data for many of these experiments because it is easier to talk about the characteristics of the data. We see the same behavior with real audio or image data (as shown in the illustrations of the previous section). Second we use the theory describing LSH's behavior to discuss how LSH works on very large databases and to connect LSH and other approaches. We performed all experiments using LSH parameters calculated using the "exact" parameters from our optimization algorithm (not brute force or the simplified approximations).

A. Experimental Validation

1) Combining Projections: Fig. 11 shows the number of retrieved results as a function of the number of projections (k) for one table $(L=1)$. We base the theory in Section III-C on the hypothesis that each projection is independent and thus the theoretical probability that any one point (nearest or any neighbor) remains after k projections is p^k. However, we see that the theory underestimates the number of retrieved points, especially for large values of k. (The discrepancy is smaller for the nearest-neighbor counts.)

The four panels of Fig. 11 suggest that the deviation is related to the dimensionality of the data. In retrospect this makes sense. The first $k=d$ projections are all linearly independent of each other. The $(d+1)$th projection in a d-dimensional space has to be a linear combination of the first d projections. Independence (in a linear algebra sense) is not the same as independence (in a statistical sense). Our projections are correlated.

In the remaining simulations of this section, we also show a "compensated" prediction based on the experimental performance after the k projections. To gauge the discrepancy when $k>d$, we do a simulation with the desired k and measure the true p_{any} performance. This then forms a new estimate of $p_{\text{any}}(w,k,r)$ that we plug into the rest of the

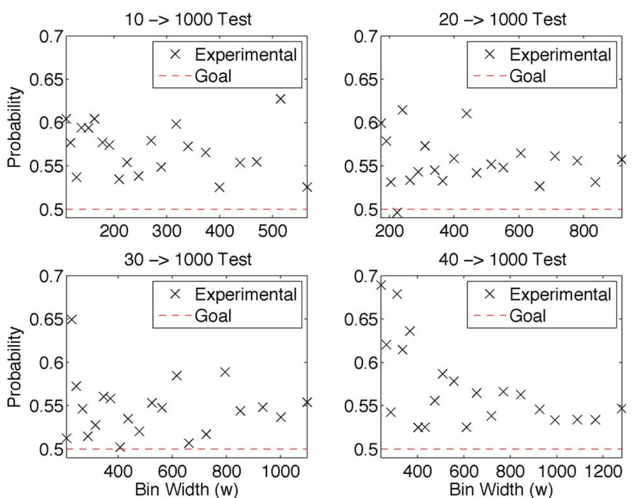

Fig. 12. ***Quality of the LSH results. These four panels show the probability of finding the true nearest neighbor in four different synthetic data sets. In each case, the design specification $1-\delta$ requires that the correct answer be found more than 50% of the time.***

theory. This shows that most of the discrepancies that follow are due to the (under) performance of the k projections.

2) Overall Results: Fig. 12 summarizes the quality of the points retrieved by LSH for the four test dimensionalities. These four panels show the probability of getting the true nearest neighbor for a large number of random queries. For each value of w around the theoretical optimum we predict the best parameters (lowest CPU time) that meet the δ performance guarantee. We are thus showing optimum performance for all values of w (although only one of these bucket sizes will have the absolute minimal CPU time). In all but one trial the measured retrieval accuracy rate is greater than the $0.50 = 1 - \delta$ target.

Fig. 13 shows the number of points returned in all L buckets for all four dimensionalities. This result is important because it determines the number of points which must be checked. If we return too many points, then the retrieval time is slow because they all require checking. While there are discrepancies, the "compensated" curve is a pretty good fit for the experimental results, especially for the larger dimensionalities.

Fig. 14 shows the overall computational effort. These tests were all performed on the same machine using our implementation of LSH in Python. While the long vectors mean that most of the work is done by the NumPy numerical library, there is still overhead in Python for which we are not accounting. In particular, the estimates are better for the longer times, perhaps because this minimizes the impact of the fixed overhead in a query.

Fig. 15 demonstrates the sensitivity of an LSH index to its parameters. For each w and k, we select the L that meets the performance guarantee and then plot the time it would take such an index to answer one query. This figure shows a long, narrow valley where the parameters are optimum. This might explain the difficulty practitioners have in finding the optimal parameters to make practical use of LSH. Note that the curve in Fig. 10 shows the minimum computation time as a function of just w—we choose optimum values for k and L. Thus, the 1-D curves of Fig. 14 follow the valley shown in Fig. 15.

Finally, we return to the real-world experimental results shown in Fig. 10. We believe that underestimating the independence of the k projections, as shown in Fig. 10, explains the discrepancy in the number of EC1M

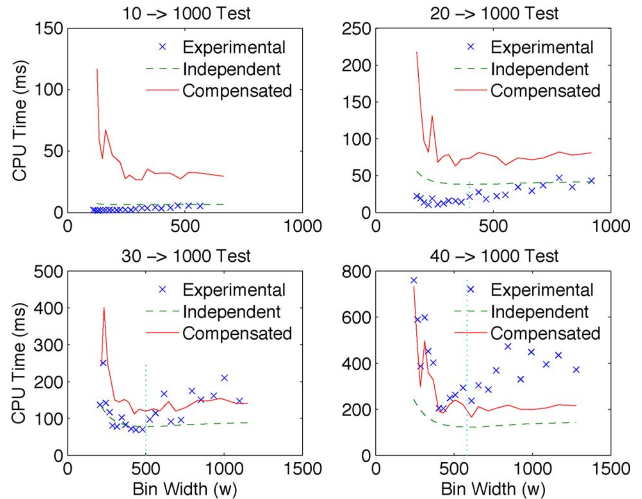

Fig. 14. ***CPU time per query. These four panels show the amount of CPU time per query in an optimal implementation of LSH, as a function of the bin width.***

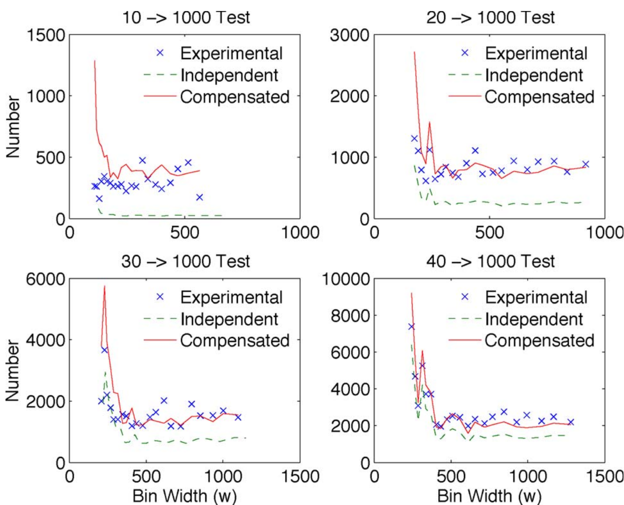

Fig. 13. ***Number of LSH candidates. These four panels show the number of candidates returned in an optimal implementation of LSH, as a function of the bin width.***

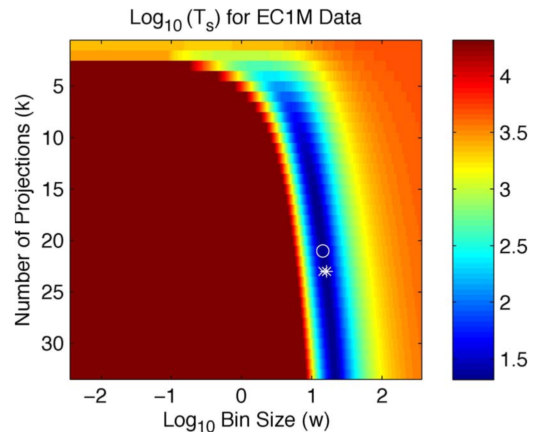

Fig. 15. ***CPU time as a function of k and w. At each point, we use the L that guarantees we meet the performance guarantee, and thus costs dramatically increase when the wrong parameters are used. The minima found using the brute-force (*) and "exact" (x) calculations are on top of each other. The simple estimate is close and is still in the valley. All times are in milliseconds, and for display purposes, we limited the maximum value to 1000 times the minimum value.***

candidates. This suggests that the intrinsic dimensionality of this image feature data is low. Nevertheless, the total CPU time is predicted with good accuracy, probably because the query time is dominated by the Python index. Our higher dimensional synthetic data are a better fit to the model.

3) Multiprobe: Multiprobe offers its own advantages and disadvantages. In the analysis in this paper, we limit the Hamming distance between the query bucket and the buckets we check. Better approaches might look at the true distance to each bucket and thus might choose a bucket with a Hamming distance of 2 before all the buckets with a distance of 1 are exhausted [18]. Still most of the benefit comes from the nearest bucket in each direction and this simplification allows us to understand the advantages of multiprobe.

Fig. 16 shows the effect that multiprobe has on the index parameters. In this plot, we show the probability of retrieving the nearest neighbor p_{nn} as a function of the number of tables (L), with and without multiprobe. Multiprobe, even with $r = 1$, reduces the number of tables that are needed, in this case cutting it in half, while still exceeding the performance guarantee. As expected, the theoretical curves underestimate the collision probabilities because the projections are not independent. We compensate for this by adjusting the theory with a compensation factor equal to $p(k, w)/p(w)^k$ where $p(k, w)$ is the measured collision rate (either p_{nn} or p_{any}) using the optimum w and k.

B. Experimental Extensions

We can use the distribution results from Casey [4] to estimate d_{nn} and d_{any} over many orders of data set size, and thus we can make predictions, based on the optimization in this work, of how LSH behaves on larger data sets. We assume independent and identically distributed Gaussian coefficients in each vector. The any-neighbor distribution comes from the difference of two Gaussian random variables and thus the distance distribution is described by a χ^2 distribution. On the other hand, Casey shows that the nearest-neighbor distribution, as a function of distance x, has the form $cx^{(c-1)}e^{-(w^c)}$. In this expression, c and w are quantities defined in Casey's paper and are based on the dimensionality and the size of the data set. This approximation is valid for large values of N. We simulate data sets with dimensionalities between 4 and 128, and with sizes from 500 000 to 1 billion points.

1) Optimum Bucket Size: LSH originally seemed appealing, especially compared to *k-d* trees, because, depending on w, LSH splits the points from a single projection into many, many different buckets. Just like a normal (string) hash, there could be an explosion of buckets that turn a lookup into a $O(1)$ process. Yet a range of experiments, both with real data and synthetic data, suggest that small buckets are not the optimum solution.

Fig. 17 shows the predicted $p_{\text{any}}(w)$ over a large range of data set sizes. We use p_{any} instead of w to characterize our performance because this probability is a rough measure of the spread of the data into different buckets. If $p_{\text{any}}(w) = 0.5$, then there is a 50% chance that any two points share the same bucket. (The complete story is more complicated since there are potentially an infinite number of possible bins and Gaussian data are unbounded.)

As shown in Fig. 17, the optimum $p_{\text{any}}(w_{\text{optimum}})$ varies over a small range between 0.53 and 0.60, suggesting that two buckets per projection is best. We do all this work and

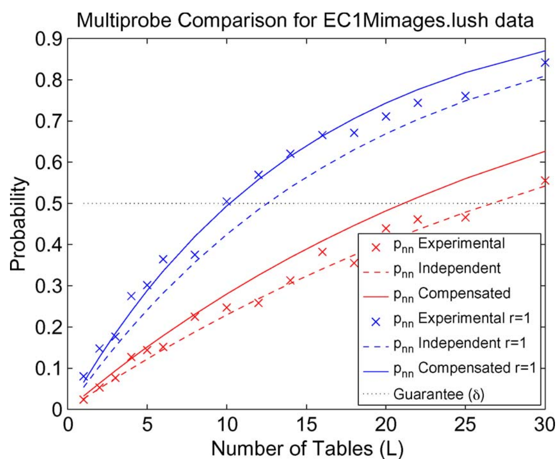

Fig. 16. ***Multiprobe probability comparison test. The lower set of curves shows the probability of retrieving the nearest neighbor without multiprobe, while the upper set shows a better retrieval rate, for the same L, when we explore a Hamming radius of 1.***

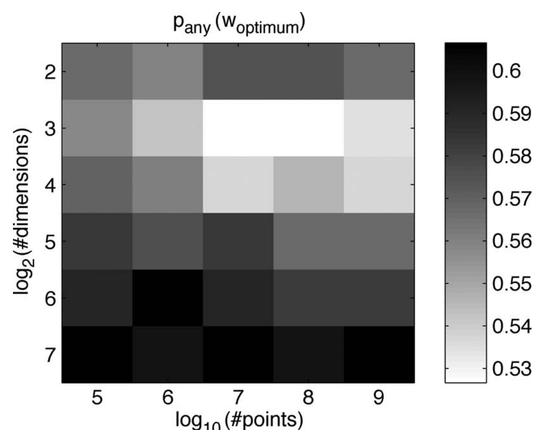

Fig. 17. ***Optimum p_{any} over a wide range of data dimensionality and number of data points. Even over all these orders of magnitude, the optimum value of p_{any} never dips below 0.5.***

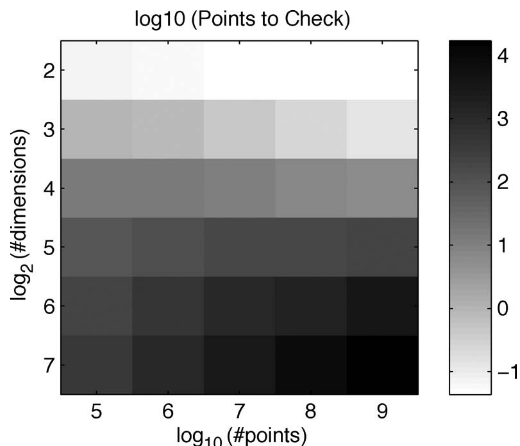

Fig. 18. *Number of candidates that need to be checked to find the nearest neighbor as a function of true data dimensionality and the number of data points.*

we end up with binary splits from each projection.[9] This result is also consistent with that shown, for example, in Fig. 10, where the search cost only goes up slightly as w gets larger. This is the first demonstration that we know of that shows the power of binary LSH. Over this wide range of parameter values we see no case where a small w, and thus more bins per projection, is optimal.

Fig. 18 shows the optimal cost, in points to check, as we vary the data set size and dimensionality. Even with 1 billion (10^9) points the number of points we must check is predicted to be less than 50 000 points (the real number is probably higher because of the interdependence of the projections).

2) Multiprobe: Fig. 19 summarizes the computation and memory differences with multiprobe. Using the synthetic data sets described above, we see that the optimum w declines by a little bit (perhaps 10%). More interestingly, there is little benefit in computational cost. In fact, the computational cost goes up by as much as a factor of 2.5. On the other hand, the amount of memory needed goes down by a factor of 2.5. This is good because in our initial experiments with LSH, and thus the motivation for our study here, we found that the number of distance checks, and thus perhaps cache and virtual memory misses, dominated the computational effort. Reducing memory usage is an important benefit of multiprobe LSH.

The optimum p_{nn} affects the possible performance of multiprobe. The optimum p_{any} without multiprobe is often close to 0.5. The nearest-neighbor probability p_{nn} is always greater than p_{any}. Thus, with p_{nn} for any *one* projection well above 0.5, the chance of finding the nearest neighbor in the adjacent bin is small (since $q_{nn} < 1 - p_{nn} < 0.5$).

[9] It is interesting to note that some of the first work on LSH [8] assumed the data were represented as bit vectors, but later work generalized LSH to multiple buckets using quantization.

V. RELATED AND FUTURE WORK

There are many methods for finding nearest neighbors. In this section, we talk about some alternatives, how the current analysis relates to these other approaches, and suggest avenues for new research.

A. Binary LSH

It is interesting to examine the search cost as a function of the quantization interval w. Fig. 10 is a typical result. The cost declines to a minimum and then only goes up a bit (perhaps a factor of 2) as w gets very large. We see the same behavior in many of our tests. This suggests that performing binary splits might be a reasonable idea. Instead of using a random bias, we might split the data in half, so that $p_{\text{any}} = 0.5$. The easiest way to do this is to perform a projection and then split the data at the median, as suggested by He *et al.* [11]. We have not investigated whether the suboptimal performance, as predicted by the derivation here, for large w is due to the fact that the random bias b is unlikely to split the data exactly in half.

The popularity of binary LSH (see Section II-B) and its power led us to wonder about the connection with *k-d* trees, a popular deterministic approach for finding nearest neighbors.

B. *k-d* Trees

The most prominent alternative to probabilistic algorithms such as LSH are space-partioning schemes such as *k-d* trees. The simplest form of *k-d* trees simply splits the data at each node using axis-aligned projections and a simple threshold [2]. We do this recursively, until the leaf nodes are as small as desired. To answer a query, we examine each node, making a decision to go left or right. At the leaf node, we test a small number of candidate points for the smallest distance. Then, we go back up the tree, revisiting each decision we made, and test whether the distance to the decision boundary is closer than closest point found so far. If the distance to the decision boundary is closer than the current best distance, then we evaluate the other branch of the tree. A more sophisticated tree approach chooses the decision or projection direction to fit the data [5]. A data-adaptive approach such as a random projection (RP) tree finds good splits because it adapts to the data. This is useful if the data are inhomogeneous or have local structure, as is usually the case. But in the end all of these approaches split the data into buckets of arbitrary shape.

We can better understand the similarities and differences of these different approaches by examining the decision boundaries for *k-d* trees, RP Trees, and LSH, as shown in Fig. 20. A simplified analysis based on fully random data minimizes the benefits of the local decisions

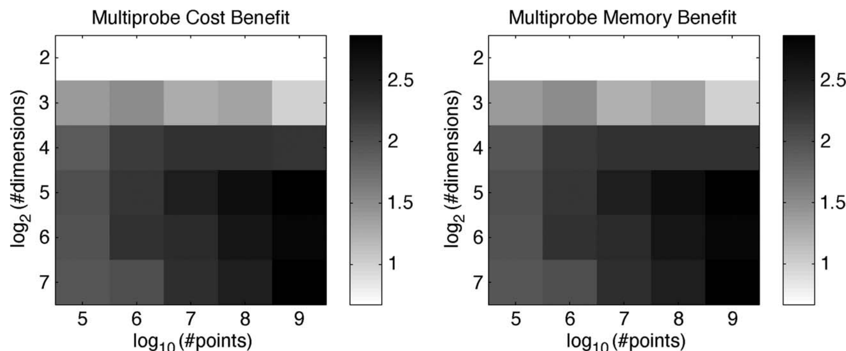

Fig. 19. *Effect of LSH multiprobe on number of candidates to check (left) and memory (right). In both cases, the ratio of the original cost over the multiprobe cost is calculated. For high values of the data dimensionality* (d) *and number of points* (n)*, multiprobe LSH with a Hamming radius of 1* $(r=1)$ *requires more candidate checking but reduces the memory usage, all the while maintaining the same performance guarantee* (δ).

in *k-d* trees. *k-d* trees especially shine when there is local structure in the data, and the decisions made at one node are quite different from that of another node. But still, it is useful to consider this random (worst) case, especially since the analysis is easier.

With uniform data, a *k-d* tree with a depth of k levels is analogous to LSH with k (global) projections, a binary decision where $p_{\text{any}}(w_{\text{optimum}}) = 0.5$, and $L = 1$. Without backtracking, a *k-d* tree is unlikely to retrieve the nearest neighbor.

But with backtracking, a *k-d* tree can retrieve the exact nearest neighbor (with $\delta = 0$). At each level of the tree, the best branch is checked, and then the best or smallest distance found so far is compared to the distance between the query point and the decision boundary. If the best distance so far is large enough, then we check the alternate node. This can be painful, however, because if one checks both sides of each node, then one checks the entire tree. The computational cost of this kind of algorithm is shown in Fig. 21. Even with a relatively low dimensionality such as 20 we are looking at all the data points. This is due to the curse of dimensionality and the structure of the tree.

It is hard to make use of the neighborhood information in a tree because its structure is inhomogenous. Due to the local decisions at each node, a bucket nearby the tree might not be adjacent in feature space to the query bucket. Thus, backtracking is an exhaustive approach (compared to the regular approach in multiprobe). One could be smarter about checking alternative nodes, in a manner analogous to multiprobe, and thus could check fewer buckets. But the overall algorithm looks a lot like binary LSH with $L = 1$. As shown above, for homogeneous data, our LSH analysis suggests that this is suboptimal.

A more powerful alternative is a forest of *k-d* trees [21], [20]. As implied by the name, multiple (random) trees are built to answer the query. Backtracking is not performed; instead the inherent randomness in the way the data are grouped by different trees provides a degree of robustness not found in a single tree. Thus, when the lookup is performed with L *k-d* trees it has much of the same flavor as LSH with L k-dimensional indices. It would be interesting to analyze a forest of *k-d* trees using the approach described here.

Finally, binary LSH is interesting, especially since we have found that under the optimum LSH parameters each dot product is being used to split the data into two buckets. By going with binary LSH, we give up on having an infinite number of (potential) buckets but the analysis is simpler.

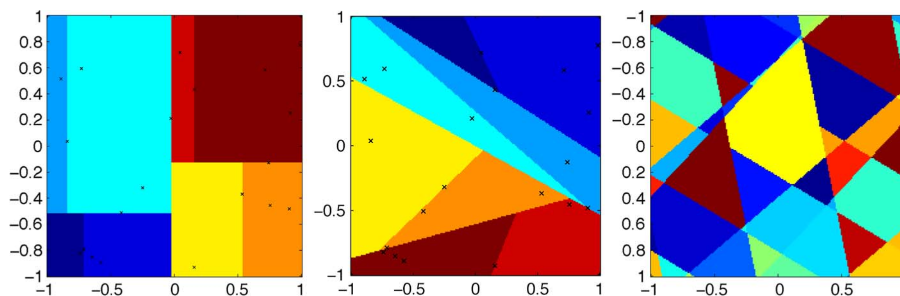

Fig. 20. *Data buckets for two kinds of* k-d *trees and LSH. These three images show typical bucket shapes for random 2-D data using axis-aligned four-level-deep* k-d *trees, RP trees, and LSH with four projections.*

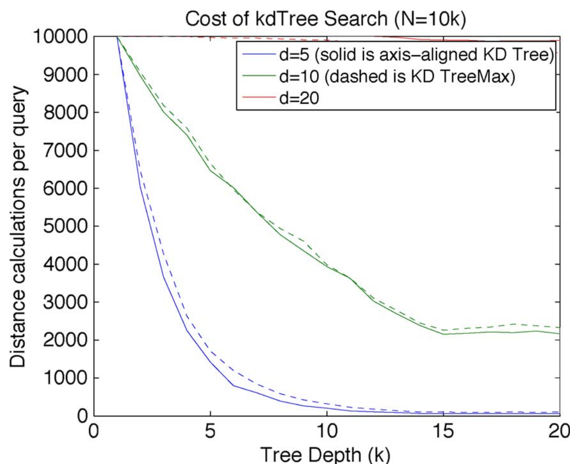

Fig. 21. ***Computational query costs for k-d trees. Three sets of data (5-, 10-, and 20-dimensional uniform random) were used to measure k-d tree performance. The solid lines show the number of candidates checked for normal (axis-aligned) k-d trees and the dashed lines show the RP variant. (This is not a good example of the power of RP trees because there is no local structure to exploit—the data are full dimensional.) Even for a relatively low-dimensional data set with 20 dimensions, we are still forced to evaluate the distance to each point. (The $d = 20$ curve nearly equals 10 000 for all values of k.)***

One possible explanation for the superior behavior is that adding a new quantization interval (by making w smaller) must "cost" more than adding a new direction (increasing k by 1).

Note that Paulevé *et al.* [22] talks about the performance of a number of different "more advanced" quantization techniques. They even show Fig. 3 for random projections with a very small quantization interval. Interestingly, our results indicate that larger quantization interval is better. It is not clear whether this is true because random projections are suboptimal and the best one can do is a binary split, or whether simpler is better is true in general with high-dimensional spaces.

C. Similarity Modifications

One can also improve LSH performance by optimizing the projections [1], [15], [16], [26]. Random projections are easy to analyze, but ignore the structure that is often found in data. One can take several different approaches to choose better projections. While we do not analyze these approaches in this paper, we hope that the parameter estimates we derive here are a useful starting point for understanding nonrandom projections.

In LSH, the projections are chosen randomly (according to some distributions). However, when the budget of bits is small, e.g., only tens or hundreds of bits for each sample, random projections may not be effective enough. Thus, several recent papers, for instance, one on spectral hashing [11], learn more effective projections by splitting the data into equal-sized bins. He *et al.*'s goal is to optimize computational effort and the ability to find the true nearest neighbor. This is different from our approach in two ways. First, in our work, we use a secondary (exact) hash to map the k-dimensional bucket ID into a linear bucket ID. Thus, the effect of larger k is extra vector multiplications, which are cheap because the coefficients often reside in a nearby memory cache. Second, we optimize total CPU time under an absolute performance guarantee (δ).

Researchers at Google start with a desire to maximize the difference between audio fingerprints from the same and different songs [1]. Starting with a high-dimensional feature vector based on a spectrogram (image), they then learn projections that give image segments from the same song and the same label (either 0 or 1). No single projection (learner) does this classification perfectly, but in concert they do very well. This is the same principle that makes LSH work—each projection only tells us a little about another point's similarity. Then, each bit vector identifies an LSH bucket containing other candidate neighbors.

VI. CONCLUSION

In this paper, we have shown how to derive optimum parameters for LSH. We start with estimates of the nearest-neighbor and any-neighbor distributions and the desired performance level (retrieve the absolute nearest neighbor with probability $1 - \delta$). Based on a simplified version of our analysis, we see that we should choose the quantization interval w to maximize the contrast between the nearest-neighbor and any-neighbor collision probabilities. Then, the number of dot products that are combined to make one k-dimensional projection causes on average each k-dimensional bucket to contain just one point. Finally, we use L k-dimensional projections to get the probability of finding the nearest neighbor above $1 - \delta$. We believe this the first derivation of the optimal parameters for LSH.

We extend our analysis to multiprobe, where we increase the chances of retrieval success by also looking in nearby buckets. This changes the optimal bucket size, and thus the other parameters. With some extra computational effort one can significantly reduce the memory requirements.

We confirmed our approach by numerical and experimental simulations on a wide range of theoretical and real databases. Most interestingly, and to our surprise, choosing w so that one gets a binary split at each projection turns out to be near-optimum. This led us to think harder about the connection between probabilistic approaches such as LSH and deterministic approaches such as k-d trees. In the simplest case, the differences are not that great (a local decision versus a global projection).

Many papers have been written claiming results that are superior to those obtained using LSH. As we discovered, and thus the motivation for this work, it is hard to find the

best parameter values using exhaustive search, especially in the large databases that we find interesting. In this paper, we do not claim that LSH is the best solution. Instead, we hope that this paper, the analysis, and the optimization algorithm we describe will be a good starting point for anybody who wants to find nearest neighbors, using random or deterministic algorithms. ■

Acknowledgment

We are grateful for the assistance and comments we received from K. Li (Princeton) and K. Grauman (Univ. of Texas, Austin) and their students. A number of anonymous reviewers have also contributed good questions and ideas.

REFERENCES

[1] S. Baluja and M. Covell, "Learning to hash: Forgiving hash functions and applications," *Data Mining Knowl. Disc.*, vol. 17, no. 3, pp. 402–430, Dec. 2008.

[2] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is 'nearest neighbor' meaningful?" in *Proc. Int. Conf. Database Theory*, 1998, pp. 217–235.

[4] M. A. Casey, C. Rhodes, and M. Slaney, "Analysis of minimum distances in high-dimensional musical spaces," *IEEE Trans. Audio Speech Lang. Process.*, vol. 16, no. 5, pp. 1015–1028, Jul. 2008.

[5] S. Dasgupta and Y. Freund, "Random projection trees for vector quantization," *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3229–3242, Jul. 2009.

[6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. 20th Annu. Symp. Comput. Geometry*, 2004, pp. 253–262.

[7] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li, "Modeling LSH for performance tuning," in *Proc. 17th ACM Conf. Inf. Knowl. Manage.*, 2008, pp. 669–678.

[8] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 518–529.

[9] A. G. Glen, L. M. Leemis, and J. H. Drew, "Computing the distribution of the product of two continuous random variables," *Comput. Stat. Data Anal.*, vol. 44, no. 3, pp. 451–464, Jan. 2004.

[10] K. Grauman, "Efficiently searching for similar images," *Commun. ACM*, vol. 53, no. 6, pp. 84–94, 2010.

[11] J. He, R. Radhakrishnan, S.-F. Chang, and C. Bauer, "Compact hashing with joint optimization of search accuracy and time," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2011, pp. 753–760.

[12] E. Hörster, M. Slaney, M. A. Ranzato, and K. Weinberger, "Unsupervised image ranking," in *Proc. 1st ACM Workshop Large-Scale Multimedia Retrieval Mining*, 2009, pp. 81–88.

[13] H. Jégou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *Proc. 10th Eur. Conf. Comput. Vis. I*, Marseille, France, 2008, pp. 304–317.

[14] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 33, no. 1, pp. 117–128, Jan. 2011.

[15] H. Jégou, T. Furon, and J.-J. Fuchs, "Anti-sparse coding for approximate nearest neighbor search," INRIA Technical Report, RR-7771, Oct. 2012. [Online]. Available: http://hal.inria.fr/inria-00633193

[16] B. Kulis, P. Jain, and K. Grauman, "Fast similarity search for learned metrics," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 12, pp. 2143–2157, Dec. 2009.

[17] J. He, S. Kumar, and S. F. Chang, "On the difficulty of nearest neighbor search," *Int. Conf. Mach. Learn.*, Edinburgh, Scotland, 2012. [Online]. Available: http://www.sanjivk.com/RelativeContrast_ICML12.pdf

[18] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "MultiProbe LSH: Efficient indexing for high-dimensional similarity search," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 950–961.

[19] R. B. Barimont and M. B. Shapiro, "Nearest neighbour searches and the curse of dimensionality," *J. Inst. Math. Appl.*, vol. 24, pp. 59–70, 1979.

[20] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," *Science*, vol. 340, no. 3, pp. 331–340, 2009.

[21] F. Moosmann, E. Nowak, and F. Jurie, "Randomized clustering forests for image classification," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 9, pp. 1632–1646, Sep. 2008.

[22] L. Paulevé, H. Jégou, and L. Amsaleg, "Locality sensitive hashing: A comparison of hash function types and querying mechanisms," *Pattern Recognit. Lett.*, vol. 31, no. 11, pp. 1348–1358, Aug. 2010.

[23] R. Salakhutdinov and G. Hinton, "Semantic hashing," *Int. J. Approx. Reason.*, vol. 50, no. 7, pp. 969–978, 2009.

[24] M. Slaney and M. Casey, "Locality-sensitive hashing for finding nearest neighbors," *IEEE Signal Process. Mag.*, vol. 25, no. 2, pp. 128–131, Mar. 2008.

[25] A. Wang, "An industrial-strength audio search algorithm," in *Proc. ISMIR Int. Symp. Music Inf. Retrieval*, Baltimore, MD, Oct. 2003, pp. 7–13.

[26] Y. Weiss, A. B. Torralba, and R. Fergus, "Spectral hashing," in *Proc. Neural Inf. Process. Syst.*, 2008, pp. 1753–1760.

[27] V. M. Zolotarev, "One-dimensional stable distributions," *Translations of Mathematical Monographs*. Providence, RI: AMS, 1986.

ABOUT THE AUTHORS

Malcolm Slaney (Fellow, IEEE) was a Principal Scientist at Yahoo! Research, and now holds the same position in Microsoft's Conversational Systems Laboratory in Mountain View, CA. He is also a (consulting) Professor at Stanford University's Center for Computer Research in Music and Acoustics (CCRMA), Stanford, CA, where he has led the Hearing Seminar for the last 20 years. Before Yahoo!, he has worked at Bell Laboratory, Schlumberger Palo Alto Research, Apple Computer, Interval Research, and IBM's Almaden Research Center. For the last several years he has helped lead the auditory group at the Telluride Neuromorphic Cognition Workshop. Lately, he has been working on multimedia analysis and music- and image-retrieval algorithms in databases with billions of items. He is a coauthor, with A. C. Kak, of the IEEE book *Principles of Computerized Tomographic Imaging*. This book was republished by SIAM in their "Classics in Applied Mathematics" series. He is coeditor, with S. Greenberg, of the book *Computational Models of Auditory Function*.

Prof. Slaney has served as an Associate Editor of the IEEE Transactions on Audio, Speech, and Signal Processing, IEEE Multimedia Magazine, and the Proceedings of the IEEE.

Yury Lifshits received his Ph.D. in computer science from Steklov Institute of Mathematics in 2007. He is a web researcher and entrepreneur. He has worked at Steklov Institute of Mathematics, California Institute of Technology (Caltech), and Yahoo! Labs. His most recent large research project is The Like Log Study, a social analytics system for world leading media.

Junfeng He received the B.S. and M.S. degrees from Tsinghua University, Beijing, China, in 2002 and 2005, respectively. He is currently working toward the Ph.D. degree at the Digital Video and Multimedia (DVMM) Lab, Electrical Engineering Department, Columbia University, New York, NY. His Ph.D. research has focused on large-scale nearest-neighbor search, with applications mainly for image/video search on both desktop and mobile platforms.

He has taken charge of or participated in many projects related to machine learning, data mining, computer vision, multimedia forensic, human computer interaction (HCI), and computational photography. He has published approximately 20 papers in conferences and journals in these areas, and contributed to five U.S. patents.