Malcolm Slaney
and Michael Casey

Locality-Sensitive Hashing for Finding Nearest Neighbors

The Internet has brought us a wealth of data, all now available at our fingertips. We can easily carry in our pockets thousands of songs, hundreds of thousands of images, and hundreds of hours of video. But even with the rapid growth of computer performance, we don't have the processing power to search this amount of data by brute force.

This lecture note describes a technique known as locality-sensitive hashing (LSH) that allows one to quickly find similar entries in large databases. This approach belongs to a novel and interesting class of algorithms that are known as randomized algorithms. A randomized algorithm does not guarantee an exact answer but instead provides a high probability guarantee that it will return the correct answer or one close to it. By investing additional computational effort, the probability can be pushed as high as desired.

RELEVANCE

There are numerous problems that involve finding similar items. These problems are often solved by finding the nearest neighbor to an object in some metric space. This is an easy problem to state, but when the database is large and the objects are complicated, the processing time grows linearly with the number of items and the complexity of the object. For very large databases of high-dimensional items, LSH is a particularly valuable technique for retrieving items that are similar to a query item. In these searches it can drastically reduce the computational time, at the cost of a small probability of failing to find the absolute closest match.

Digital Object Identifier 10.1109/MSP.2007.914237

PREREQUISITES

This lecture note is based on simple geometric reasoning. Some knowledge of probabilities and a comfort with the mathematics of high-dimensional spaces are useful.

PROBLEM STATEMENT

Given a query point, we wish to find the points in a large database that are closest to the query. We wish to guarantee with a high probability equal to $1 - \delta$ that we return the nearest neighbor for any query point.

Conceptually, this problem is easily solved by iterating through each point in the database and calculating the distance to the query object. However, our database may contain billions of objects—each object described by a vector that contains hundreds of dimensions. Therefore, it is important that we find a solution that does not depend on a linear search of the database. Existing methods to accomplish this search include trees and hashes.

TREES

By building a tree of objects, we can start at the top node when given a query, ask if our query object is to the left or to the right of the current node, and then recursively descend the tree. If the tree is properly constructed, this solves the query problem in $O(\log N)$ time, where N is the number of objects. In a one-dimensional space, this is a binary search. In a multidimensional space, this idea becomes the k-d tree algorithm [1]. The problem with multidimensional algorithms such as k-d trees is that they break down when the dimensionality of the search space is greater than a few dimensions—we end up testing nearly all the nodes in the data set and the computational complexity grows to $O(N)$.

HASHES

By building a hash table, i.e., a data structure that allows us to quickly map between a symbol (i.e., a string) and a value, when given a query we can calculate an arbitrary, pseudorandom function of the symbol that maps the symbol into an integer that indexes a table. Thus a symbol with dozens of characters, and perhaps hundreds of bits of data, is mapped to a relatively small index into the table. A collision occurs when two points hash to the same value and there are special provisions to allow more than one symbol per hash value.

A well-designed hash table allows a symbol lookup in $O(1)$ time with $O(N)$ memory, where N is the number of entries in the table. Also a well-designed hash function separates two symbols that are close together into different buckets. This makes a hash table a good means of finding *exact* matches. To find *approximate* (*near*) matches efficiently we use a locality-sensitive hash.

SOLUTION

LSH is based on the simple idea that, if two points are close together, then after a "projection" operation these two points will remain close together.

This idea can be easily understood using the examples shown in Figure 1. Two points that are close together on the sphere are also close together when the sphere is projected onto the two-dimensional page. This is true no matter how we rotate the sphere. Two other points on the sphere that are far apart will, for some orientations, be close together on the page, but it is more likely that the points will remain far apart. We will describe a different

1053-5888/08/$20.00©2008IEEE

type of projection operator, but thinking about rendering a multidimensional sphere onto a two-dimensional page is a good metaphor.

To further expand this basic idea, we start with a random projection operation that maps a data point from a high-dimensional point to a low-dimensional subspace. First, we note which points are close to our query points. Second, we create projections from a number of different directions and keep track of the nearby points. We keep a list of these found points and note the points that appear close to each other in more than one projection.

Part of the art of solving this problem is defining a projection, defining the notion of "nearby" (similarity test) so that we keep track of a manageable number of points, finding a good hash implementation, and analyzing the hash performance.

RANDOM PROJECTIONS: THE DOT PRODUCT

At the core of LSH is the scalar projection (or dot product), given by $h(\vec{v}) = \vec{v} \cdot \vec{x}$, where $\vec{v}$ is a query point in a high-dimensional space, and $\vec{x}$ is a vector with components that are selected at random from a Gaussian distribution, for example $\mathcal{N}(0, 1)$. This scalar projection is then quantized into a set of hash bins, with the intention that nearby items in the original space will fall into the same bin. The resulting full hash function is given by

$$h^{x,b}(\vec{v}) = \left\lfloor \frac{\vec{x} \cdot \vec{v} + b}{w} \right\rfloor \quad (1)$$

where $\lfloor \cdot \rfloor$ is the floor operation, w is the width of each quantization bin, and b is a random variable uniformly distributed between 0 and w that makes the quantization error easier to analyze, with no loss in performance.

For the projection operator to serve our purposes, it must project nearby points to positions that are close together. This requires that:

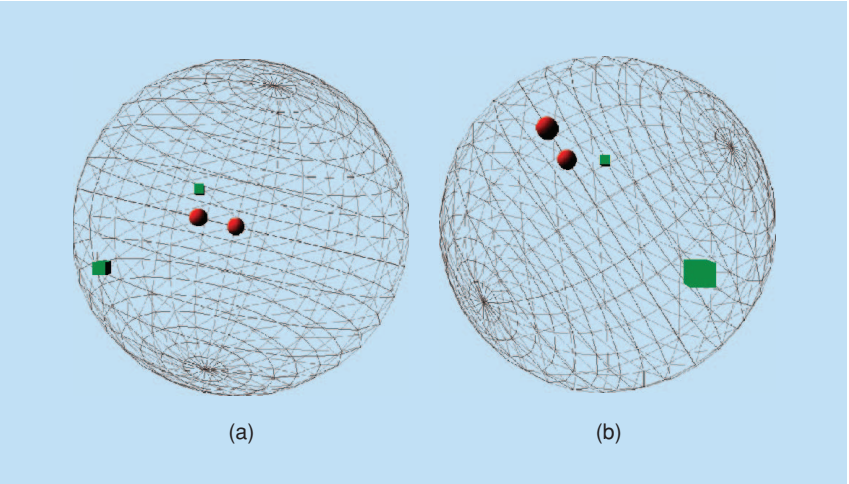
- for any points p and q in $\mathcal{R}^d$ that are close to each other, there is a high probability P_1 that they fall into the same bucket

$$P_H[h(p) = h(q)] \geq P_1 \text{ for } ||p - q|| \leq R_1 \quad (2)$$

- for any points p and q in $\mathcal{R}^d$ that are far apart, there is a low probability $P_2 < P_1$ that they fall into the same bucket

$$P_H[h(p) = h(q)] \leq P_2 \text{ for } ||p - q|| \geq cR_1 = R_2. \quad (3)$$

In (2) and (3), $||\cdot||$ is the L_2 vector norm and $R_2 > R_1$. Note that, due to the linearity of the dot product, the difference between two image points $||h(p) - h(q)||$ has a magnitude whose distribution is proportional to $||p - q||$—therefore, $P_1 > P_2$.



[FIG1] Two examples showing projections of two close (circles) and two distant (squares) points onto the printed page.

RANDOM PROJECTIONS: THE K DOT PRODUCTS

We can further magnify the difference between P_1 and P_2, by performing k dot products in parallel. This increases the ratio of the probabilities (given above) that points at different separations will fall into the same quantization bin, since $(P_1/P_2)^k > (P_1/P_2)$. The resulting projection, is obtained by performing the k independent dot products to transform the query point $\vec{v}$ into k real numbers. As with the scalar (dot product) projection, we quantize the k inner products per (1) with the intention that similar points will fall in the same bucket in all dimensions.

Increasing the quantization bucket width w will increase the number of points that fall into each bucket. To obtain our final nearest neighbor result we will have to perform a linear search through all the points that fall into the same bucket as the query, so varying w effects a trade-off between a larger table with a smaller final linear search, or a more compact table with more points to consider in the final search.

Within each set of k dot products, we achieve success if the query and the nearest neighbor are in the same bin in all k dot products. This occurs with probability P_1^k, which decreases as we include more dot products. To reduce the impact of an "unlucky" quantization in any one projection, we form L independent projections and pool the neighbors from all of these. This is motivated by the fact that a true near neighbor will be unlikely to be unlucky in all the projections. By increasing L we can find the true nearest neighbor with arbitrarily high probability. Our cover-song experiments [2] used 7–14 dot products per hash (k) and more than 150 projections (L).

HASH IMPLEMENTATION

This process of projection and quantization places each data point in a hash bucket described by k integer indices. Since this k-dimensional space is sparse, we can use conventional (exact) hashing methods to efficiently find when points fall into common buckets. For illustration we describe the approach used by E^2 *LSH* [3], but more sophisticated approaches have also been proposed [4], [5].

A naïve search to find reference points in the same bucket as the query point could easily take $O(\log N)$ operations, but we reduce this to $O(1)$ by using a pair of conventional hash functions. First, we use a conventional hash to map the k-dimensional quantized projection into a single linear index by computing

$$T_1 = \left(\sum_i H_i k_i\right) \mod P_1 \qquad (4)$$

where H_i are integer weights and P_1 is the hash table size (a large prime number). The goal of this hash table is to put each distinct point in the k-dimensional space into a separate one of the P_1 table entries, i.e., to avoid "collisions" in which unrelated points hash to the same value. Although a well-constructed hash will distribute entries quite uniformly, as the table gets smaller (perhaps to allow it to fit in memory) the risk of unrelated points colliding naturally increases.

To accommodate this problem, we use a second hash T_2 of the k-dimensional points to check that points retrieved from the hash table do indeed match our query. T_2 has the same form as T_1 but uses different weights and size. We store the values from the second hash (which we call fingerprints) in the bins chosen by the first hash, then on retrieval we can compare the fingerprints of items retrieved from the matching bin to identify the true matches, if they exist. Since these fingerprints are short (for instance, 16 bit values) their comparison is much faster than comparing the full k-dimensional original points (for which the computational expenses and memory requirements grow as k increases). Moreover, the chances of simultaneous collisions under both T_1 and T_2 can quickly be made vanishingly small, even for a relatively small hash table.

PERFORMANCE ANALYSIS

ACCURACY

The accuracy of an LSH is determined by probability that it will find the true nearest neighbor. To analyze this, we introduce the concept of s-stable distributions. A distribution D is s-stable if, for any independent identically distributed (iid) random variables $X_1, \ldots, X_n$ distributed according to D, and any real numbers $v_1, \ldots, v_n$, the random variable $\sum_i v_i X_i$ has a probability distribution that is the same as that of the random variable

$$\left(\sum_i |v_i|^s\right)^{(1/s)} X \qquad (5)$$

where X is drawn from D. For $s = 2$ (the L_2 norm), a Gaussian probability distribution is s-stable.

To analyze the accuracy of our projections, recall that the projections of two close points p and q, separated by the distance $u = ||p - q||$, will always be close. However, because of quantization they might fall on opposite sides of a boundary and thus land in different buckets. The probability that these two points are quantized into the same bucket is given by

$$p(u) = Pr_{a,b}[h_{a,b}(p) = h_{a,b}(q)] = \int_0^w \frac{1}{u} f_s\left(\frac{t}{u}\right)\left(1 - \frac{t}{w}\right) dt \qquad (6)$$

where f_s is the probability density function (pdf) of the hash H as given by (5). For any given bucket width w, this probability falls as the distance u grows.

Using this probability we calculate P_H in (2) and (3) for an L_2 space with R_1 equal to the bin width w [4] as follows:

$$P_2 = 1 - 2F(-w/c) - \frac{2}{\sqrt{2\pi}w/c} \times \left(1 - e^{-(w^2/2c^2)}\right). \qquad (7)$$

Here $F()$is the cumulative pdf of a Gaussian random variable, and c is the ratio of distances from (3). Setting $c = 1$ gives us P_1.

The probability that a single point falls into the same bucket as the query is given by $(P_1)^k$. Consequently, the probability that all L projections fail to produce a collision between the query and the true nearest neighbor is equal to $(1 - P_1^k)^L$. By requiring that the probability of LSH failing to find the true nearest neighbor is no more than δ, a given value of k will require L to be at least

$$L = \frac{[\log \delta]}{\log(1 - P_1^k)}. \qquad (8)$$

The E^2 *LSH* algorithm finds the best value for k by experimentally evaluating the cost of the calculation for samples in the given data set.

SPEED

The amount of time needed to find a nearest neighbor is the time T_g needed to calculate and hash the projections, plus the time T_c needed to search the buckets for collisions. Because there are kL projections, T_g is $O(nkL)$, where n is the dimensionality of the original data space. On the other hand, T_c increases linearly based on the expected number of collisions, i.e., $T_c = O(dLN_c)$, where d is the average number of points in each bucket. N_c, the expected number of collisions for a single projection, is given by

$$N_c = \sum_{q' \in D} p^k(||q - q'||) \qquad (9)$$

where $p()$ from (6) gives the probability that each point contributes to a collision, and D represents all the points in the database. It is easy to see that T_g increases as a function of k, while T_c decreases since $p^k < p$ for $p < 1$ and $k > 1$.

APPLICATIONS

The LSH algorithm has been applied successfully to quickly find nearest neighbors

in very large databases. Instead of finding exact matches as conventional hashes would, LSH takes into account the locality of the points so that nearby points remain nearby. Examples of such applications include finding duplicate pages on the Web, image retrieval, and music retrieval.

FINDING DUPLICATE PAGES ON THE WEB

The Web contains many duplicate pages, partly because content is duplicated across sites and partly because there is more than one URL that points to the same file on a disk. Yet search engines should not return several copies of the same page. A solution to identify Web page duplicates makes use of shingles. Each shingle represents a portion of a Web page and is computed by forming a histogram of the words found within that portion of the page. We can test to see if a portion of the page is duplicated elsewhere on the Web by looking for other shingles with the same histogram. Given that there are billions of pages on the Web and any portion of any page might be a duplicate, there are an enormous number of shingles to test.

AltaVista, the first large-scale Web search engine, used random selections (similarly to LSH) to test the similarity of pages [6]. If the shingles of the new page match shingles from the database, then it is likely that the new page bears a strong resemblance to an existing page. The nearest-neighbor solution is important because Web pages are surrounded by navigational and other information that changes from site to site. An approximate solution to this problem is desired, especially when balanced with the computational savings of a solution like LSH.

RETRIEVING IMAGE AND MUSIC

LSH can be used in image retrieval as an object recognition tool [7]. We compute a detailed metric for many different orientations and configurations of an object we wish to recognize. Then, given a new image we simply check our database to see if a precomputed object's metrics are close to our query. This database contains millions of poses and LSH allows us to quickly check if the query object is known.

In music retrieval typically we use conventional hashes and robust features to find musical matches. The features can be fingerprints, i.e., representations of the audio signal that are robust to common types of abuse that are performed to audio before it reaches our ears [8]. Fingerprints can be computed, for instance, by noting the peaks in the spectrum (because they are robust to noise) and encoding their position in time and space. One then just has to query the database for the same fingerprint.

However, to find similar songs we cannot use fingerprints because these are different when a song is remixed for a new audience or when a different artist performs the same song. Instead, we can use several seconds of the song—a snippet—as a shingle. To determine if two songs are similar, we need to query the database and see if a large enough number of the query shingles are close to one song in the database [2]. Although closeness depends on the feature vector, we know that long shingles provide specificity. This is particularly important because we can eliminate duplicates to improve search results and to link recommendation data between similar songs. As discussed earlier, LSH proves useful to identify nearest neighbors quickly even when the database is very large.

CONCLUSIONS—WHAT WE HAVE LEARNED

In this lecture note, we have described the theory and implementation of a randomized algorithm known as LSH. Unlike conventional computer hashes that are designed to return *exact* matches in $O(1)$ time, an LSH algorithm uses dot products with random vectors to quickly find *nearest* neighbors. LSH provides a probabilistic guarantee that it will return the correct answer. In systems that have other sources of error (perhaps due to mislabeled data) one can reduce the LSH error below the error due to other sources, while significantly improving the computational performance. This makes LSH in particular, and randomized algorithms in general, important in today's world of Internet-sized databases.

ACKNOWLEDGMENTS

We appreciate thoughtful comments we have received from Alex Jaffe, Sara Anderson, and several reviewers.

AUTHORS

Malcolm Slaney (malcolm@ieee.org) is a researcher with Yahoo! Research, Sunnyvale, California, and a consulting professor at Stanford University. He is a coauthor of the book *Principles of Computerized Tomographic Imaging* and coeditor of the book *Computational Models of Hearing*. He is a Senior Member of IEEE.

Michael Casey (m.casey@gold.ac.uk) is a professor of music at Dartmouth College, Hanover, New Hampshire, and visiting research professor of computer science at Goldsmiths College, University of London. His main interests are in the area of music information retrieval. He is a Member of IEEE.

REFERENCES

[1] J. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, 1975.

[2] M. Casey and M. Slaney, "Fast recognition of remixed music audio," in *Proc. IEEE ICASSP*, 2007, pp. IV-1425–1428.

[3] A. Andoni and P. Indyk, "E^{2}LSH 0.1 User Manual," Jun. 2005. [Online]. Available: http://web.mit.edu/andoni/www/LSH

[4] A. Andoni, M. Datar, N. Immorlica, and V. Mirrokni, "Locality-sensitive hashing using stable distributions," in *Nearest Neighbor Methods in Learning and Vision: Theory and Practice*, T. Darrell, P. Indyk, and G. Shakhnarovich, Eds. Cambridge, MA: MIT Press, 2006.

[5] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for near neighbor problem in high dimensions," in *Proc. Symp. Foundations of Computer Science (FOCS'06)*, 2006.

[6] A. Broder, S. Glassman, M. Manasse, and G. Zweig, "Syntactic clustering of the Web," in *Proc. WWW*, Santa Clara, 1997, pp. 1157–1166.

[7] G. Shakhnarovich, P. Viola, and T. Darrell, "Fast pose estimation with parameter-sensitive hashing," in *Nearest Neighbor Methods in Learning and Vision: Theory and Practice*, T. Darrell, P. Indyk, and G. Shakhnarovich, Eds. Cambridge, MA: MIT Press, 2006.

[8] P. Cano, E. Batlle, T. Kalker, and J. Haitsma, "A review of algorithms for audio fingerprinting," in *Proc. Int. Workshop Multimedia Signal Processing*, 2002.

SP