Approximate Computation of Object Distances by Locality-Sensitive Hashing

Selim Mimaroglu and Dan A. Simovici
University of Massachusetts at Boston,
Department of Computer Science,
Boston, Massachusetts 02125, USA,
{smimarog,dsim}@cs.umb.edu

***Abstract*—We propose an approximate computation technique for inter-object distances for binary data sets. Our approach is based on the locality sensitive hashing, scales up with the number of objects and is much faster than the "brute-force" computation of these distances.**

I. Introduction

Locality Sensitive Hashing (LSH), first introduced in [1], can be used for an approximate calculation of distances between the tuples of a table by using randomized hash functions. A close variant of LSH which works best with the Hamming distance is described in [2].

Our data set is a binary table $\mathcal{D}$, having N distinct objects and a set I that consists of n distinct attributes. A set $K \subseteq I$ of k attributes, designated as a *probe* and chosen randomly defines a random hashing function f_K by assigning to a tuple t the numerical binary equivalent of the projection of t on the set K, $t[K]$.

Each hashing function produces a partition of the set of tuples; each block of this partition consists of tuples that collide under that hashing function.

LSH is used for clustering the Web in [3]. In [4] it is used to enhance the agglomerative hierarchical clustering of the single link method [5]. Both of these techniques rely on the same idea provided by LSH: close objects are likely to collide under a high number of randomly chosen hashing functions. Both of these techniques compute the real distances between objects residing in the same blocks.

LSH-Link algorithm [4] has $O(N^2)$ time complexity as the classical single link method and it works as follows. Database $\mathcal{D}$ is hashed into m partitions by using randomly generated functions of LSH on k attributes. In the first phase of the algorithm distances between all pairs of tuples $(\mathbf{u}, \mathbf{v})$ residing in the same blocks are computed, and all the pairs of tuples having distance at most r are merged at once. Note that this pairwise computation takes place on every block of every partition. If after the first phase there is more than one cluster, the algorithm proceeds to the second phase by selecting a new projection size k' such that $k' < k$, and a new distance value r' such that $r' > r$. LSH-Link hashes the whole database $\mathcal{D}$ again by using the new values r', k'. It merges the pair of clusters with respect to r'. If there is more than one cluster, the algorithm proceeds to the next phase by selecting new values r'', k'' and by repeating all the calculations. This continues until there is only one cluster. Authors of [4] point that in a phase several clusters may be merged as opposed to merging only two clusters in classical single link algorithm. Note that if the initial distance r is not chosen carefully the LSH-Link algorithm may take many phases, therefore yielding many redundant hashing and pairwise distance computations. Furthermore, in the worst case this algorithm computes N^2 distances.

The clustering algorithms proposed in [3] and [4] focus on finding the approximate set of near neighbors $\mathsf{ANN}(\mathbf{u})$ of an object $\mathbf{u}$, followed by finding real near neighbors of $\mathbf{u}$ by computing the actual distances $d(\mathbf{u}, \mathbf{v})$ for all $\mathbf{v} \in \mathsf{ANN}(\mathbf{u})$. Note that some of the real neighbors of $\mathbf{u}$ may be missed because LSH does not guarantee to put all the close objects in the same blocks.

We aim for a distinct goal, namely an efficient, approximative computation of the distance matrix of the set of objects using LSH, which allows us to use a variety of standard clustering algorithms.

In the next section relation between randomly generated hash function collisions and distances between objects is explained. Experimental results is followed by conclusions and future work.

II. Collisions and Distances

A *binary data collection* is a sequence $\mathcal{D} = (\mathbf{t}_1, \ldots, \mathbf{t}_N)$ of tuples, where $\mathbf{t}_i \in \{0,1\}^n$.

Let $K = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. The *projection of a tuple* $\boldsymbol{t} \in \{0,1\}^n$ *on* K is the tuple $\mathbf{t}[K] = (t_{i_1}, \ldots, t_{i_k})$. The *$K$-projection of the binary data collection* $\mathcal{D}$ is the binary data collection $\mathcal{D}[K] = (\mathbf{t}_1[K], \ldots, \mathbf{t}_N[K])$.

Each K-projection of $\mathcal{D}$ generates a function $h_K : \{1, \ldots, N\} \longrightarrow \mathbb{N}$, where $h_K(r)$ is the binary equivalent of the sequence $\mathbf{t}_r[K]$. This can be seen in Figure 1, where the function $h_{\{i_1,i_3,i_5\}}$ for the binary data collection shown in Part (a) is given in Part (b) of the figure. $h_{\{i_1,i_3,i_5\}}$ creates a partition with 8 blocks; 2 of these are empty as shown in Figure 2.

The Hamming distance between two tuples $\mathbf{u}, \mathbf{v} \in \{0,1\}^n$ is given by

$$d(\mathbf{u}, \mathbf{v}) = |\{i \in \{1, \ldots, n\} \mid u_i \neq v_i\}|,$$

$\mathcal{D}$

r	i_1	i_2	i_3	i_4	i_5
1	1	0	0	1	1
2	0	1	1	0	0
3	1	0	1	0	0
4	1	1	0	1	0
5	0	1	1	1	1
6	0	0	1	1	1
7	1	0	1	0	1
8	1	1	0	0	1
9	0	1	1	1	0

(a)

r	$h_K(r)$
1	5
2	2
3	6
4	4
5	3
6	3
7	7
8	5
9	2

(b)

Fig. 1. A binary collection and the hashing function h_K for $K = \{i_1, i_3, i_5\}$.

000 {}	*001* {}	*010* {2,9}	*011* {5,6}
100 {4}	*101* {1,8}	*110* {3}	*111* {7}

Fig. 2. All the blocks created by h_K for $K = \{i_1, i_3, i_5\}$. Block descriptors, and corresponding row numbers are shown.

where $\mathbf{u} = (u_1, \ldots, u_n)$ and $\mathbf{v} = (v_1, \ldots, v_n)$.

Suppose that the set of attributes K that defines a probe is chosen at random. There are $\binom{n}{k}$ such choices if $|K| = k$. A collision takes place between two rows $\mathbf{u}$ and $\mathbf{v}$ if the chosen k attributes are among the $n-d$ attributes on which $\mathbf{u}$ and $\mathbf{v}$ are equal, where $d = d(\mathbf{u}, \mathbf{v})$ is the Hamming distance between $\mathbf{u}$ and $\mathbf{v}$. There are $\binom{n-d}{k}$ such choices for the set I. Therefore, for any two rows $\mathbf{u}, \mathbf{v}$ of $\mathcal{D}$ the collision probability for h_K, that is, the probability that $h_K(\mathbf{u}) = h_K(\mathbf{v})$ is

$$p = \frac{\binom{n-d}{k}}{\binom{n}{k}}.$$

If m sets K having k elements are chosen at random, then $C(\mathbf{u}, \mathbf{v})$ the total number of collisions that occur in this experiment is a binomially distributed variable with the distribution $B(m, p)$. Thus, the expected number of collisions is

$$E(C(\mathbf{u}, \mathbf{v})) = m\frac{\binom{n-d}{k}}{\binom{n}{k}}$$

if $k \leq n - d$, which is typically the case. If $k > n - d$ a collision is impossible and $p = 0$. It is clear that the smaller the distance $d(\mathbf{u}, \mathbf{v})$, the larger the number of collisions will be.

Using Stirling's Formula we can write

$$\begin{aligned}
\frac{\binom{n-d}{k}}{\binom{n}{k}} &= \frac{\frac{(n-d)!}{k!(n-d-k)!}}{\frac{n!}{k!(n-k)!}} \\
&= \frac{(n-d)!}{n!}\frac{(n-k)!}{(n-d-k)!} \\
&\approx \frac{(n-d)^{n-d+0.5}(n-k)^{(n-k+0.5)}}{n^{(n+0.5)}(n-d-k)^{n-d-k+0.5}} \\
&= \left(\frac{n^2-nd-nk+dk}{n^2-nd-nk}\right)^{n+0.5} \cdot \\
&\quad \left(\frac{n-d-k}{n-d}\right)^{d} \cdot \left(1-\frac{d}{n-k}\right)^{k}.
\end{aligned}$$

For moderately large values of n the first two factors are close to 1. Thus, the expected value of the number of collisions is

$$E(C(\mathbf{u}, \mathbf{v})) \approx m \cdot \left(1 - \frac{d}{n-k}\right)^k$$

Let $c(\mathbf{u}, \mathbf{v}) = E(C(\mathbf{u}, \mathbf{v}))/m$ be the *relative number of collisions*. Then, we estimate the distance between $\mathbf{u}$ and $\mathbf{v}$ as

$$d(\mathbf{u}, \mathbf{v}) \approx (n-k)(1 - c(\mathbf{u}, \mathbf{v})^{\frac{1}{k}}) \tag{1}$$

Assume that m probes with k attributes are applied, where $1 \leq m$ and $1 \leq k \leq n$. Since we deal with binary data, each attribute may take a value of either 0 or 1. Therefore, the partition that corresponds to a k-probe may contain up to 2^k blocks.

Let $n_1, \ldots, n_{2^k}$ be the sizes of the blocks that correspond to a k-probe. For each block of the partition we need to update the number of collisions of pairs. Therefore, for a block of size n_i we need to perform $\binom{n_i}{2}$ updates of the pair counters. For example, for a block having three elements $\{a, b, c\}$, the collision counts of the pairs: $(a, b), (a, c), (b, c)$ are increased by one. The total time required for this partition is no more than

$$\begin{aligned}
\sum_{i=1}^{2^k}\binom{n_i}{2} &= \frac{1}{2}\left(\sum_{i=1}^{2^k} {n_i}^2 - n\right) \\
&\leq \frac{n^2}{2^k} - n,
\end{aligned}$$

because the largest value of the expression $\sum_{i=1}^{2^k} n_i^2$ under the constraint $\sum_{i=1}^{2^k} n_i = n$ is obtained when $n_1 = \cdots = n_{2^k} = \frac{n}{2^k}$. The process has to be repeated for each of m probes and this requires a time proportional to $m\left(\frac{n^2}{2^k} - n\right)$.

III. Experiments

We discuss the experimental setup and some implementation details. We start with data in binary format where each row is a bit vector. This data representation is very memory-efficient and operations on bit vectors are fast.

A clustering, which corresponds to a probe, is represented by a Java class that is parametrized by the projection size. Components of a clustering include its clusters and members of these clusters. In a clustering Java's Random class is used to select projected attributes randomly.

To produce a clustering, k attributes are randomly selected and the objects are projected on the selected set of attributes. A cluster C consists of those objects that have the same projection $\mathbf{p} \in \{0,1\}^k$ on the set of attributes that constitutes the probe. The value of bit vector $\mathbf{p}$ is the descriptor of the cluster. The cluster itself is represented by a bit vector $\mathbf{b}_C \in \{0,1\}^N$, where $(\mathbf{b}_C)_i = 1$ if and only if the object $\mathbf{u}_i$ belongs to the cluster C.

Note that clusters (blocks) do not overlap, and the identifiers of the objects are placed into appropriate clusters according to the cluster descriptors.

The number of clusterings m is determined by the user and is passed as an argument to the implementation. Both the number of clusterings (which equals the number of probes m) and the width k of the probes are set to positive integers by the user.

All the clusterings are populated in one scan of the database as follows. Each clustering may have at most 2^k non-empty clusters. First, empty clusterings are initialized, then each object in the database $\mathcal{D}$ is passed to all the clusterings. Each clustering projects the object on its own randomly selected attributes and then places the object in the appropriate cluster according to the cluster descriptors.

Assume a clustering projects on first, fifth, and tenth attributes, then the object $\mathbf{1}001\mathbf{0}1101\mathbf{0}$, is placed in cluster 4 of this clustering. Similarly, if another clustering projects on fourth, sixth, and seventh attributes, then the same object $100\mathbf{1}0\mathbf{11}010$, is placed in cluster 7 of this clustering. This computation takes place for each data object. In one scan of database $\mathcal{D}$, m clusterings, each having 2^k clusters can be generated efficiently.

We use an $N \times N$ matrix referred to as the *simultaneous occurrence matrix* to keep track of the number of collisions of each of the object pairs. After obtaining the clusterings, each cluster in every clustering is scanned once and the occurrence matrix component that corresponds to each pair $(\mathbf{u}, \mathbf{v})$ of objects in $\mathcal{D}$ that co-occur in the same cluster is incremented by 1. Note that there are at most $m2^k$ clusters to scan.

For example, assume that we have a 5-object dataset $\mathcal{D}$ with $n = 4$ attributes, the width of the probes is $k = 1$, and we have $m = 3$ clusterings, whose bit vectors are

$$\begin{pmatrix}0\\1\\1\\1\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\1\\0\end{pmatrix}, \text{ and } \begin{pmatrix}1\\0\\1\\0\\0\end{pmatrix}.$$

The simultaneous occurrence matrix is shown below. For example, objects 2 and 4 occur in the same clusters 3 times.

	1	**2**	**3**	**4**	**5**
1	3	0	2	0	2
2	0	3	1	3	1
3	2	1	3	1	1
4	0	3	1	3	1
5	2	1	1	1	3

Using the simultaneous occurrence matrix, the approximate distance matrix can be created using the formula (1). For the current example the distance matrix is shown below.

	1	**2**	**3**	**4**	**5**
1	0	3	1	3	1
2	3	0	2	0	2
3	1	2	0	2	2
4	3	0	2	0	2
5	1	2	2	2	0

We conducted a series of experiments on synthetically generated binary data using a Pentium 3.0GHz computer having 4GB of main memory running on Linux. Our algorithm is implemented in Java. In Figure 3 the total time spent to create our distance matrices, and Hamming distance matrix are shown for varying size databases. Recall that k is the projection size and m is number of clusterings.

In principle our algorithm requires quadratic time. However, in practice, it scales (almost) linearly. Each cluster is represented by a bit vector, therefore the elements of clusters are ordered. When incrementing the pairs in clusters we obey to the order imposed by the bit vectors. Thus, our implementation possess *cache locality* which yields high performance. On data sets having up to 20,000 objects the parabolic growth is almost invisible. We verified high cache hit rate by using debugging and profiling tools on Linux (e.g. *valgrind*).

For calculating the Hamming distance we use bit vectors and XOR operation on the bit vectors, which is the fastest way to compute the Hamming distance. Only the upper half of the Hamming distance matrix is computed because the matrix is symmetric. It can be seen from Figure 3 that our approach is 17 times faster on some databases (20,000 object database).

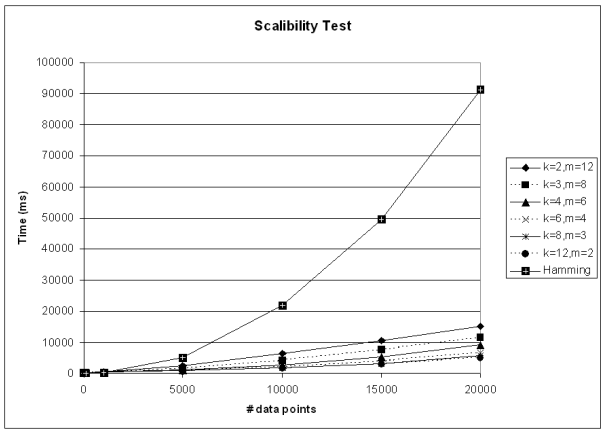

Fig. 3. Scalability test and time comparison.

To evaluate the quality of our approximation we used

the cophenetic correlation coefficient [6] to calculate the correlation between two distance matrices. This coefficient takes a value between 0 and 1, where a higher value implies better correlation. We calculated the cophenetic correlation coefficient between our approximate distance matrix D , and the Hamming distance matrix H. The averages of the matrices D and H are denoted by $\bar{d}$ and $\bar{h}$, respectively. The coefficient is given by

$$c = \frac{\sum (D_{ij} - \bar{d})(H_{ij} - \bar{h})}{\sqrt{\sum (D_{ij} - \bar{d})^2 \sum (H_{ij} - \bar{h})^2}}.$$

In Figure 4 we show the cophenetic correlation coefficient for varying k and m. Note that best experimental results are achieved when $k = 2$. Higher values of m produces better correlations and the coefficient approaches 1 for reasonable values of m. Figure 5 shows that for 200 probes the cophenetic correlation coefficient is 0.963. 200 probes may seem extreme, but each probe scans only 2 attributes out of the total 20 attributes. Therefore each probe scans 10 percent of the database, and 200 probes correspond to a total of 20 full scans of the database. On the other hand to compute a distance matrix of 1000 points, around 500 full scans of the database are required.

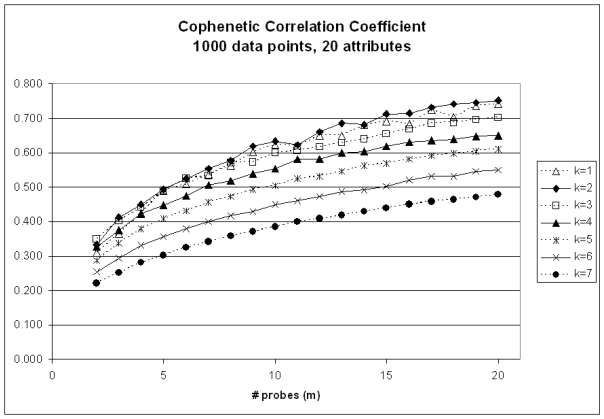

Fig. 4. Cophenetic correlation coefficient of a database having 1000 data points, and 20 attributes.

The accuracy for data sets having considerably more attributes does not degrade as shown in Figure 6. We observe that when $k = 2$, setting m approximately to the number of attributes produces very good results.

Our algorithm is highly parallelizable. We have used Java threads on an Apple - Mac Pro having 2 Intel Xeon quad-core 64 bit processors. This server has 8 cores, 16GB main memory and it can host 8 processes simultaneously. Each clustering is implemented as a Java thread, and these threads are converted to operating system native threads by the compiler. Note that there is some overhead for creating operating system native threads. We relied on the operating system (Mac OS X Leopard) to distribute the work evenly. On a database having 15,000 data points, we computed approximative distance matrices for $k = 4$ and varying number of probes m. In Figure 7 we report

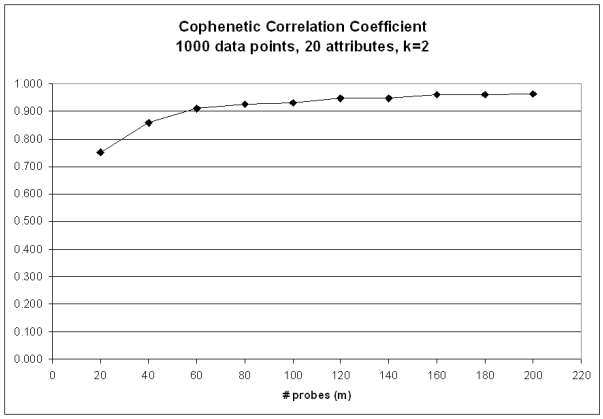

Fig. 5. Cophenetic correlation coefficient of a database having 1000 data points, and 20 attributes, $k = 2$ and higher values of m.

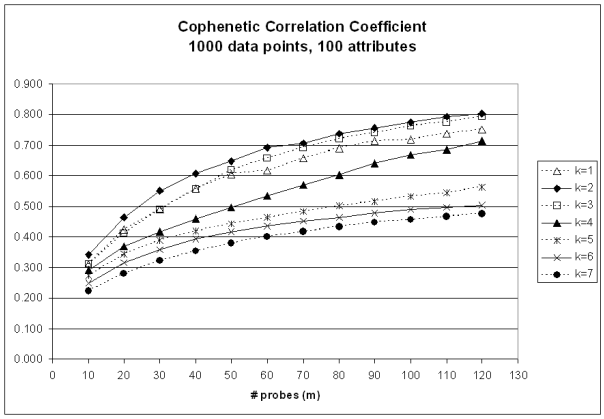

Fig. 6. Cophenetic correlation coefficient of a database having 1000 data points, and 100 attributes.

the total time spent for these computations. The results are as expected: total time stays almost stable for increasing values of m.

# probes (m)	Total Time (ms)
2	2171
3	2045
4	2132
5	2269
6	2281
7	2442
8	2484

Fig. 7. Parallel computation results on a database having 15,000 data points.

IV. Conclusions and Future Work

Computing the distance matrix of a database is a fundamental problem in clustering. We presented a novel approach which is an efficient and approximative computation of the distance matrix. Our technique relies on randomized hash

functions known as Locality Sensitive Hashing (LSH). Experimental results demonstrate that our method is fast and accurate. Our technique is suitable for parallel computation, which takes advantage of modern multiprocessor architectures. We have implemented our approach for both single processor and multiprocessor systems.

As mentioned earlier, each randomized hash function creates a clustering. Combining all the clusterings into a single superior clustering in an efficient, and intelligent way will be our future work.

REFERENCES

[1] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613, 1998.

[2] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 518–529, 1999.

[3] T. Haveliwala, A. Gionis, and P. Indyk, "Scalable techniques for clustering the web," *Proc. of the WebDB Workshop*.

[4] H. Koga, T. Ishibashi, and T. Watanabe, "Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing," *Knowledge and Information Systems*, vol. 12, no. 1, pp. 25–53, 2007.

[5] R. Sibson, "SLINK: An optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1973.

[6] R. Sokal and F. Rohlf, "The comparison of dendrograms by objective methods," *Taxon*, vol. 11, no. 1, pp. 30–40, 1962.