Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search

Anshumali Shrivastava ANSHU@CS.CORNELL.EDU
Dept. of Computer Science, Computing and Information Science (CIS), Cornell University, Ithaca, NY 14853, USA

Ping Li PINGLI@STAT.RUTGERS.EDU
Dept. of Statistics & Biostatistics, Dept. of Computer Science, Rutgers University, Piscataway, NJ 08854, USA

Abstract

The query complexity of *locality sensitive hashing (LSH)* based similarity search is dominated by the number of hash evaluations, and this number grows with the data size (Indyk & Motwani, 1998). In industrial applications such as search where the data are often high-dimensional and binary (e.g., text n-grams), *minwise hashing* is widely adopted, which requires applying a large number of permutations on the data. This is costly in computation and energy-consumption.

In this paper, we propose a hashing technique which generates all the necessary hash evaluations needed for similarity search, using one single permutation. The heart of the proposed hash function is a "rotation" scheme which densifies the sparse sketches of *one permutation hashing* (Li et al., 2012) in an unbiased fashion thereby maintaining the LSH property. This makes the obtained sketches suitable for hash table construction. This idea of rotation presented in this paper could be of independent interest for densifying other types of sparse sketches.

Using our proposed hashing method, the query time of a (K, L)-parameterized LSH is reduced from the typical $O(dKL)$ complexity to merely $O(KL + dL)$, where d is the number of nonzeros of the data vector, K is the number of hashes in each hash table, and L is the number of hash tables. Our experimental evaluation on real data confirms that the proposed scheme significantly reduces the query processing time over minwise hashing without loss in retrieval accuracies.

1. Introduction

Near neighbor search is a fundamental problem with widespread applications in search, databases, learning, computer vision, etc. The task is to return a small number of data points from a dataset, which are most similar to a given input query. Efficient (sub-linear time) algorithms for near neighbor search has been an active research topic since the early days of computer science, for example, the *K-D tree* (Friedman et al., 1975). In this paper, we develop a new *hashing* technique for fast near neighbor search when the data are sparse, ultra-high-dimensional, and binary.

1.1. Massive, Sparse, High-Dimensional (Binary) Data

The use of ultra-high-dimensional data has become popular in machine learning practice. Tong (2008) discussed datasets with 10^{11} items and 10^9 features. Weinberger et al. (2009) experimented with a binary dataset of potentially 16 trillion unique features. Agarwal et al. (2011); Chandra et al. also described linear learning with massive binary data. Chapelle et al. (1999) reported good performance by using binary data for histogram-based image classification.

One reason why binary high-dimensional data are common in practice is due to the ubiquitous use of n-gram (e.g., n-contiguous words) features in text and vision applications. With $n \geq 3$, the size of the dictionary becomes very large and most of the grams appear at most once. More than 10 years ago, industry already used n-grams (Broder et al., 1997; Fetterly et al., 2003) with (e.g.,) $n \geq 5$.

1.2. Fast Approximate Near Neighbor Search and LSH

The framework of *locality sensitive hashing (LSH)* (Indyk & Motwani, 1998) is popular for sub-linear time near neighbor search. The idea of LSH is to bucket (group) the data points probabilistically in a hash table so that similar data points are more likely to reside in the same bucket.The performance of LSH depends on the data types and the implementations of specific underlying hashing algorithms. For binary data, minwise hashing is a popular choice.

1.3. Minwise hashing and b-Bit Minwise Hashing

For binary sparse data, it is routine to store only the locations of the nonzero entries. In other words, we can view binary vector in $\mathbb{R}^D$ equivalently as sets in $\Omega = \{0, 1, 2, ..., D-1\}$. Consider two sets $S_1, S_2 \subseteq \Omega$. A popular measure of similarity is the *resemblance* $R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$.

Consider a random permutation $\pi : \Omega \longrightarrow \Omega$. We apply π on S_1, S_2 and store the two minimum values under π:

Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32. Copyright 2014 by the author(s).

$\min \pi(S_1)$ and $\min \pi(S_2)$, as the hashed values. By an elementary probability argument,

$$\mathbf{Pr}\left(\min \pi(S_1) = \min \pi(S_2)\right) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = R \qquad (1)$$

This method is known as *minwise hashing* (Broder, 1997; Broder et al., 1998; 1997). See Figure 1 for an illustration.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
$\pi(S_1)$ 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1 0
$\pi(S_2)$ 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 0

Figure 1. **(b-bit) minwise hashing**. Consider two binary vectors (sets), $S_1, S_2 \subseteq \Omega = \{0, 1, ..., 23\}$. The two permuted sets are: $\pi(S_1) = \{5, 7, 14, 15, 16, 18, 21, 22\}$, $\pi(S_2) = \{5, 6, 12, 14, 16, 17\}$. Thus, minwise stores $\min \pi(S_1) = 5$ and $\min \pi(S_2) = 5$. With b-bit minwise hashing, we store their lowest b bits. For example, if $b = 1$, we store 1 and 1, respectively.

Li & König (2010); Li et al. (2013) proposed b-bit minwise hashing by only storing the lowest b-bits (as opposed to, e.g., 64 bits) of the hashed values; also see Figure 1. This substantially reduces not only the storage but also the dimensionality. Li et al. (2011) applied this technique in large-scale learning. Shrivastava & Li (2012) directly used these bits to build hash tables for near neighbor search.

1.4. Classical LSH with Minwise Hashing

The crucial formula of the collision probability (1) suggests that minwise hashing belongs to the LSH family.

Let us consider the task of similarity search over a giant collection of sets (binary vectors) $\mathcal{C} = \{S_1, S_2, ..., S_N\}$, where N could run into billions. Given a query S_q, we are interested in finding $S \in \mathcal{C}$ with high value of $\frac{|S_q \cap S|}{|S_q \cup S|}$. The idea of linear scan is infeasible due to the size of $\mathcal{C}$. The idea behind LSH with minwise hashing is to concatenate K different minwise hashes for each $S_i \in \mathcal{C}$, and then store S_i in the bucket indexed by

$$B(S_i) = [h_{\pi_1}(S_i); h_{\pi_2}(S_i); ...; h_{\pi_K}(S_i)], \qquad (2)$$

where $\pi_i : i \in \{1, 2, ..., K\}$ are K different random permutations and $h_{\pi_i}(S) = \min(\pi_i(S))$. Given a query S_q, we retrieve all the elements from bucket $B(S_q)$ for potential similar candidates. The bucket $B(S_q)$ contains elements $S_i \in \mathcal{C}$ whose K different hash values collide with that of the query. By the LSH property of the hash function, i.e., (1), these elements have higher probability of being similar to the query S_q compared to a random point. This probability value can be tuned by choosing appropriate value for parameter K. The entire process is repeated independently L times to ensure good recall. Here, the value of L is again optimized depending on the desired similarity levels.

The query time cost is dominated by $O(dKL)$ hash evaluations, where d is the number of nonzeros of the query vector. Our work will reduce the cost to $O(KL + dL)$. Previously, there were two major attempts to reduce query time (i.e., hashing cost), which generated multiple hash values from one single permutation: (i) *Conditional Random Sampling (CRS)* and (ii) *One permutation hashing (OPH)*.

1.5. Conditional Random Sampling (CRS)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
$\pi(S_1)$ 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1 0
$\pi(S_2)$ 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 0

Figure 2. **Conditional Random Sampling (CRS)**. Suppose the data are already permuted under π. We store the smallest $k = 3$ nonzeros as "sketches": $\{5, 7, 14\}$ and $\{5, 6, 12\}$, respectively. Because $\min\{14, 12\} = 12$, we only look at up to the 12-th column (i.e., 13 columns in total). Because we have already applied a random permutation, any chunk of the data can be viewed as a random sample, in particular, the chunk from the 0-th column to the 12-th column. This looks we equivalently obtain a random sample of size 13, from which we can estimate any similarities, not just resemblance. Also, it clear that the method is naturally applicable to non-binary data. A careful analysis showed that it is (slightly) better NOT to use the last column, i.e., we only use up to the 11-th column (a sample of size 12 instead of 13).

Minwise hashing requires applying many permutations on the data. Li & Church (2007); Li et al. (2006) developed the *Conditional Random Sampling (CRS)* by directly taking the k smallest nonzeros (called *sketches*) from only one permutation. The method is unique in that it constructs an equivalent random sample pairwise (or group-wise) from these nonzeros. Note that the original minwise hashing paper (Broder, 1997) actually also took the smallest k nonzeros from one permutation. Li & Church (2007) showed that CRS is significantly more accurate. CRS naturally extends to multi-way similarity (Li et al., 2010; Shrivastava & Li, 2013) and the method is not restricted to binary data.

Although CRS requires only one permutation, the drawback of that method is that the samples (sketches) are not aligned. Thus, strictly speaking, CRS can not be used for linear learning or near neighbor search by building hash tables; otherwise the performance would not be ideal.

1.6. One Permutation Hashing

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
$\pi(S_1)$ 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1 0
$\pi(S_2)$ 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 0

Figure 3. **One Permutation Hashing**. Instead of only storing the smallest nonzero in each permutation and repeating the permutation k times, we can use just one permutation, break the space evenly into k bins, and store the smallest nonzero in each bin. It is not directly applicable to near neighbor search due to empty bins.

Li et al. (2012) developed *one permutation hashing*. As illustrated in Figure 3, the method breaks the space equally

into k bins after one permutation and stores the smallest nonzero of each bin. In this way, the samples are naturally aligned. Note that if $k = D$, where $D = |\Omega|$, then we get back the (permuted) original data matrix. When there are no empty bins, one permutation hashing estimates the resemblance without bias. However, if empty bins occur, we need a strategy to deal with them. The strategy adopted by Li et al. (2012) is to simply treat empty bins as "zeros" (i.e., *zero-coding*), which works well for linear learning. Note that, in Li et al. (2012), "zero" means zero in the "expanded metric space" (so that indicator function can be explicitly written as an inner product).

1.7. Limitation of One Permutation Hashing

One permutation hashing can not be directly used for near neighbor search by building hash tables because empty bins do not offer indexing capability. In other words, because of these empty bins, it is not possible to determine which bin value to use for bucketing.Using the LSH framework, each hash table may need (e.g.,) $10 \sim 20$ hash values and we may need (e.g.,) 100 or more hash tables. If we generate all the necessary (e.g., 2000) hash values from merely one permutation, the chance of having empty bins might be high in sparse data. For example, suppose on average each data vector has 500 nonzeros. If we use 2000 bins, then roughly $(1 - 1/2000)^{500} \approx 78\%$ of the bins would be empty.

Suppose we adopt the zero-coding strategy, by coding every empty bin with any fixed special number. If empty bins dominate, then two sparse vectors will become artificially "similar". On the other hand, suppose we use the "random-coding" strategy, by coding an empty bin randomly (and independently) as a number in $\{0, 1, 2, D-1\}$. Again, if empty bins dominate, then two sparse vectors which are similar in terms of the original resemblance may artificially become not so similar. We will see later that these schemes lead to significant deviation from the expected behavior. We need a better strategy to handle empty bins.

1.8. Our Proposal: One Permutation with Rotation

Our proposed hashing method is simple and effective. After one permutation hashing, we collect the hashed values for each set. If a bin is empty, we "borrow" the hashed value from the first non-empty bin on the right (circular). Due to the circular operation, the scheme for filling empty bins appears like a "rotation". We can show mathematically that we obtain a valid LSH scheme. Because we generate all necessary hash values from merely one permutation, the query time of a (K, L)-parameterized LSH scheme is reduced from $O(dKL)$ to $O(KL + dL)$, where d is the number of nonzeros of the query vector. We will show empirically that its performance in near neighbor search is virtually identical to that of the original minwise hashing.

An interesting consequence is that our work supplies an unbiased estimate of resemblance regardless of the number of empty bins, unlike the original paper (Li et al., 2012).

2. Our Proposed Hashing Method

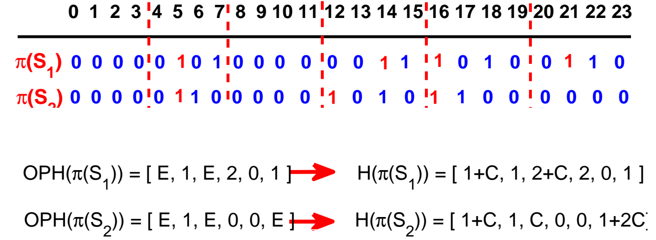

Figure 4. **One Permutation Hashing+Rotation**, with $D = 24$ (space size) and $k = 6$ bins (i.e., $D/k = 4$). For one permutation hashing (OPH), the hashed value is defined in (4). In particular, for set S_1, $OPH(\pi(S_1)) = [E, 1, E, 2, 0, 1]$ where E stands for "empty bins". The 0-th and 2-th bins are empty. In the 1-th bin, the minimum is 5, which becomes 1 after the modular operation: $5 \mod 4 = 1$. Our proposed hash method, defined in (5), needs to fill the two empty bins for S_1. For the 0-th bin, the first non-empty bin on the right is the 1-th bin (i.e., $t = 1$) with hashed value 1. Thus, we fill the 0-th empty bin with "$1 + C$". For the 2-th bin, the first non-empty bin on the right is the 3-th bin (i.e., $t = 1$ again) with hashed value 0 and hence we fill the 2-th bin with "C". The interesting case for S_2 is the 5-th bin, which is empty. The first non-empty bin on the right (circular) is the 1-th bin (as $5 + 2 \mod 6 = 1$). Thus, the 5-th bin becomes "1+2C".

Our proposed hashing method is easy to understand with an example as in Figure 4. Consider $S \subseteq \Omega = \{0, 1, ..., D-1\}$ and a random permutation $\pi : \Omega \longrightarrow \Omega$. As in Figure 3 (also in Figure 4), we divide the space evenly into k bins. For the j-th bin, where $0 \le j \le k-1$, we define the set

$$M_j(\pi(S)) = \left\{ \pi(S) \cap \left[\frac{Dj}{k}, \frac{D(j+1)}{k} \right) \right\} \tag{3}$$

If $M_j(\pi(S))$ is empty, we denote the hashed value under this one permutation hashing by $\underset{j}{OPH}(\pi(S)) = E$, where E stands for "empty". If the set is not empty, we just take the minimum of the set (mod by D/k). In this paper, without loss of generality, we always assume D is divisible by k (otherwise we just increase D by padding zeros). That is, if $M_j(\pi(S)) \neq \phi$, we have

$$\underset{j}{OPH}(\pi(S)) = \min M_j(\pi(S)) \bmod \frac{D}{k} \tag{4}$$

We are now ready to define our hashing scheme, which is basically one permutation hashing plus a "rotation" scheme for filling empty bins. Formally, we define

$$\mathcal{H}_j(\pi(S)) = \begin{cases} \underset{j}{OPH}(\pi(S)) & \text{if } \underset{j}{OPH}(\pi(S)) \neq E \\ \underset{(j+t) \bmod k}{OPH}(\pi(S)) + tC & \text{otherwise} \end{cases} \tag{5}$$

$$t = \min z, \quad s.t. \underset{(j+z) \bmod k}{OPH}(\pi(S)) \neq E \tag{6}$$

Here $C \geq \frac{D}{k}+1$ is a constant, to avoid undesired collisions. Basically, we fill the empty bins with the hashed value of the first non-empty bin on the "right" (circular).

Theorem 1

$$\mathbf{Pr}\left(\mathcal{H}_j(\pi(S_1)) = \mathcal{H}_j(\pi(S_2))\right) = R \tag{7}$$

Theorem 1 proves that our proposed method provides a valid hash function which satisfies the LSH property from one permutation. While the main application of our method is for approximate neighbor search, which we will elaborate in Section 4, another promising use of our work is for estimating resemblance using only linear estimator (i.e., an estimator which is an inner product); see Section 3.

3. Resemblance Estimation

Theorem 1 naturally suggests a linear, unbiased estimator of the resemblance R:

$$\hat{R} = \frac{1}{k}\sum_{j=0}^{k-1} 1\left\{\mathcal{H}_j(\pi(S_1)) = \mathcal{H}_j(\pi(S_2))\right\} \tag{8}$$

Linear estimator is important because it implies that the hashed data form an inner product space which allows us to take advantage of the modern linear learning algorithms (Joachims, 2006; Shalev-Shwartz et al., 2007; Bottou; Fan et al., 2008) such as linear SVM.

The original one permutation hashing paper (Li et al., 2012) proved the following unbiased estimator of R

$$\hat{R}_{OPH,ub} = \frac{N_{mat}}{k - N_{emp}} \tag{9}$$

$$N_{emp} = \sum_{j=0}^{k-1} 1\left\{\underset{j}{OPH}(\pi(S_1)) = E \text{ and } \underset{j}{OPH}(\pi(S_2)) = E\right\}$$

$$N_{mat} = \sum_{j=0}^{k-1} 1\left\{\underset{j}{OPH}(\pi(S_1)) = \underset{j}{OPH}(\pi(S_2)) \neq E\right\}$$

which unfortunately is not a linear estimator because the number of "jointly empty" bins N_{emp} would not be known until we see both hashed sets. To address this issue, Li et al. (2012) provided a modified estimator

$$\hat{R}_{OPH} = \frac{N_{mat}}{\sqrt{k - N_{emp}^{(1)}}\sqrt{k - N_{emp}^{(2)}}} \tag{10}$$

$$N_{emp}^{(1)} = \sum_{j=0}^{k-1} 1\left\{\underset{j}{OPH}(\pi(S_1)) = E\right\},$$

$$N_{emp}^{(2)} = \sum_{j=0}^{k-1} 1\left\{\underset{j}{OPH}(\pi(S_2)) = E\right\}$$

This estimator, although is not unbiased (for estimating R), works well in the context of linear learning. In fact, the normalization by $\sqrt{k - N_{emp}^{(1)}}\sqrt{k - N_{emp}^{(2)}}$ is smoothly integrated in the SVM training procedure which usually requires the input vectors to have unit l_2 norms.

It is easy to see that as k increases (to D), $\hat{R}_{OPH}$ estimates the original (normalized) inner product, not resemblance. From the perspective of applications (in linear learning), this is not necessarily a bad choice, of course.

Here, we provide an experimental study to compare the two estimators, $\hat{R}$ in (8) and $\hat{R}_{OPH}$ in (10). The dataset, extracted from a chunk of Web crawl (with 2^{16} documents), is described in Table 1, which consists of 12 pairs of sets (i.e., total 24 words). Each set consists of the document IDs which contain the word at least once.

Table 1. 12 pairs of words used in Experiment 1. For example, "A" and "THE" correspond to the two sets of document IDs which contained word "A" and word "THE" respectively.

Word 1	Word 2	$\lvert S_1 \rvert$	$\lvert S_2 \rvert$	R
HONG	KONG	940	948	0.925
RIGHTS	RESERVED	12234	11272	0.877
A	THE	39063	42754	0.644
UNITED	STATES	4079	3981	0.591
SAN	FRANCISCO	3194	1651	0.456
CREDIT	CARD	2999	2697	0.285
TOP	BUSINESS	9151	8284	0.163
SEARCH	ENGINE	14029	2708	0.152
TIME	JOB	12386	3263	0.128
LOW	PAY	2936	2828	0.112
SCHOOL	DISTRICT	4555	1471	0.087
REVIEW	PAPER	3197	1944	0.078

Figure 5 and Figure 6 summarizes the estimation accuracies in terms of the bias and mean square error (MSE). For $\hat{R}$, bias = $E\left(\hat{R} - R\right)$, and MSE = $E\left(\hat{R} - R\right)^2$. The definitions for $\hat{R}_{OPH}$ is analogous. The results confirm that our proposed hash method leads to an unbiased estimator regardless of the data sparsity or number of bins k. The estimator in the original one permutation hashing paper can be severely biased when k is too large (i.e., when there are many empty bins). The MSEs suggest that the variance of $\hat{R}$ essentially follows $\frac{R(1-R)}{k}$, which is the variance of the original minwise hashing estimator (Li & König, 2010), unless k is too large. But even when the MSEs deviate from $\frac{R(1-R)}{k}$, they are not large, unlike $\hat{R}_{OPH}$.

This experimental study confirms that our proposed hash method works well for resemblance estimation (and hence it is useful for training resemblance kernel SVM using a linear algorithm). The more exciting application of our work would be approximate near neighbor search.

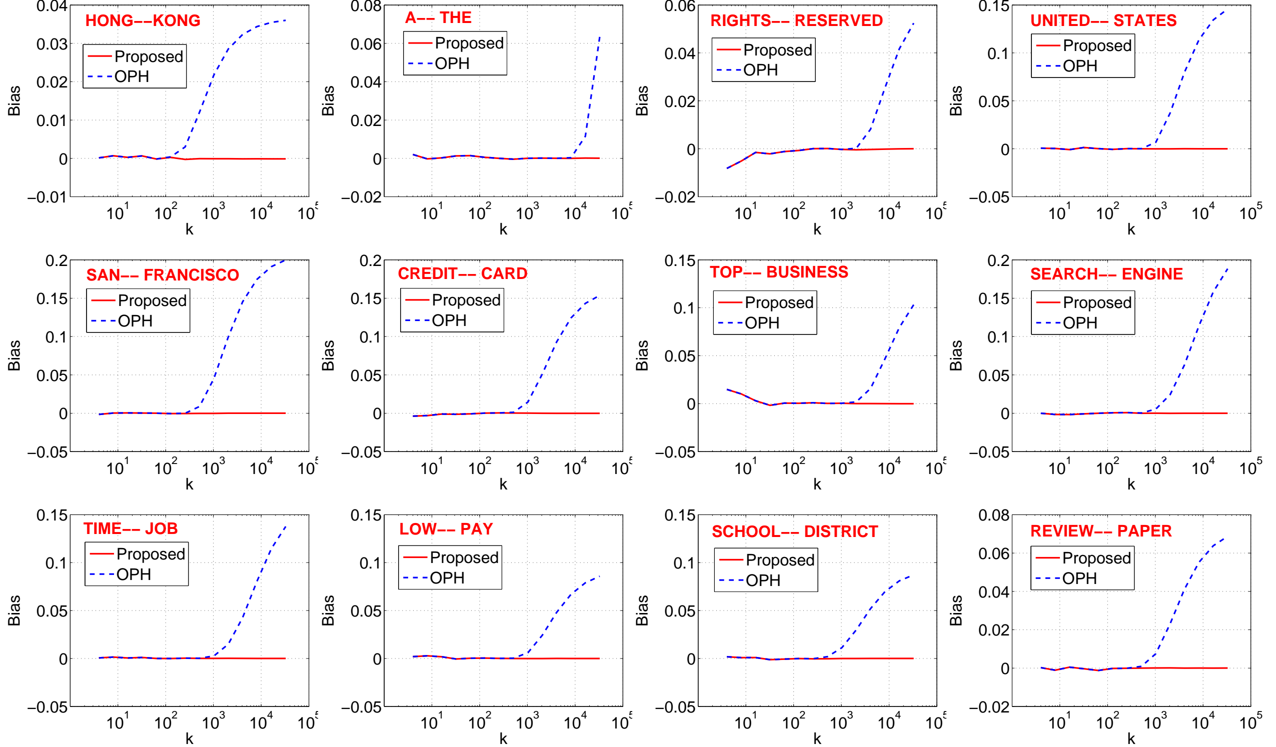

Figure 5. **Bias in resemblance estimation**. The plots are the biases of the proposed estimator $\hat{R}$ defined in (8) and the previous estimator $\hat{R}_{OPH}$ defined in (10) from the original one permutation hashing paper (Li et al., 2012). See Table 1 for a detailed description of the data. It is clear that the proposed estimator $\hat{R}$ is strictly unbiased regardless of k, the number of bins which ranges from 2^2 to 2^{15}.

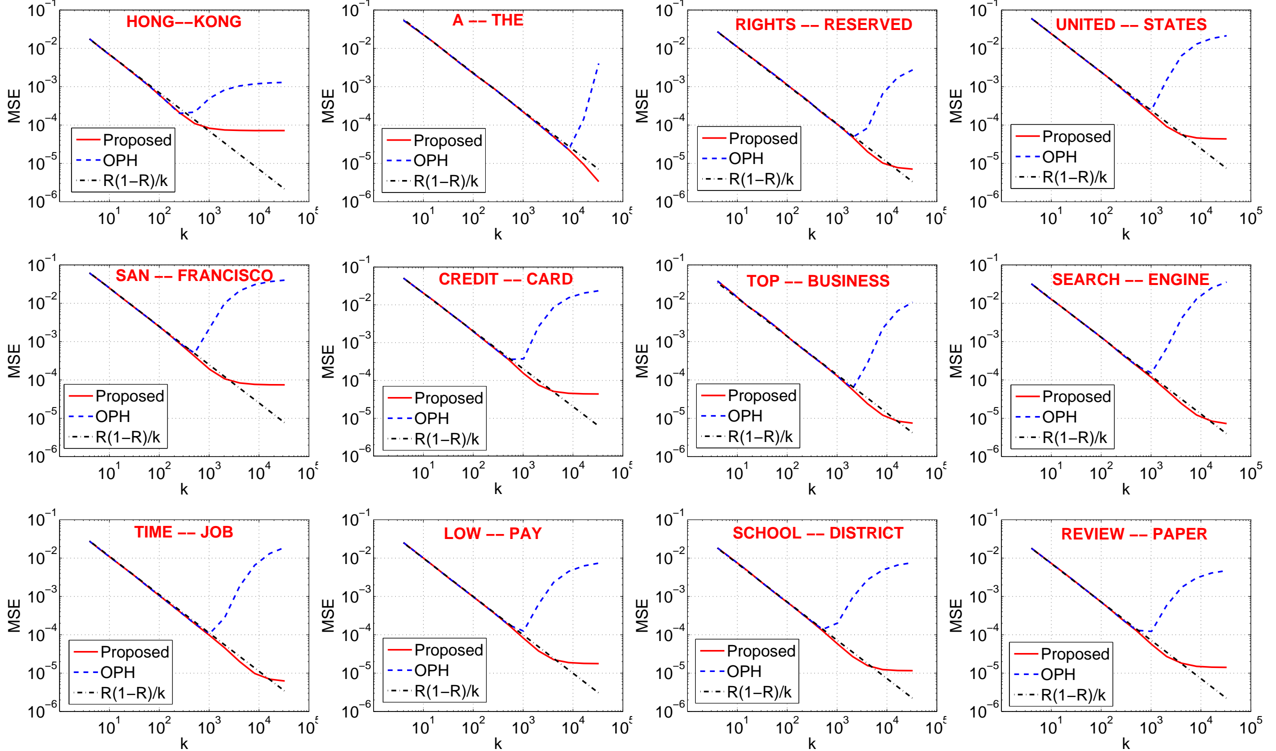

Figure 6. **MSE in resemblance estimation**. See the caption of Figure 5 for more descriptions. The MSEs of $\hat{R}$ essentially follow $R(1-R)/k$ which is the variance of the original minwise hashing, unless k is too large (i.e., when there are too many empty bins). The previous estimator $\hat{R}_{OPH}$ estimates R poorly when k is large, as it is not designed for estimating R.

4. Near Neighbor Search

We implement the classical LSH algorithm described in Section 1.4 using the standard procedures (Andoni & Indyk, 2004) with the following choices of hash functions:

- **Traditional Minwise Hashing** We use the standard minwise hashing scheme $h_i(S) = \min(\pi_i(S))$ which uses $K \times L$ independent permutations.

- **The Proposed Scheme** We use our proposed hashing scheme $\mathcal{H}$ given by (5) which uses only one permutation to generate all the $K \times L$ hash evaluations.

- **Empty Equal Scheme (EE)** We use a heuristic way of dealing with empty bins by treating the event of simultaneous empty bins as a match (or hash collision). This can be achieved by assigning a fixed special symbol to all the empty bins. We call this **Empty Equal (EE)** scheme. This can be formally defined as:

$$h_j^{EE}(S) = \begin{cases} \underset{j}{OPH}(\pi(S)), & \text{if } \underset{j}{OPH}(\pi(S)) \neq E \\ \text{A fixed special symbol}, & \text{otherwise} \end{cases} \quad (11)$$

- **Empty Not Equal Scheme (ENE)** Alternatively, one can also consider the strategy of treating simultaneous empty bins as a mismatch of hash values referred to as **Empty Not Equal (ENE)**. ENE can be reasonably achieved by assigning a new random number to each empty bin independently. The random number will ensure, with high probability, that two empty bins do not match. This leads to the following hash function

$$h_j^{ENE}(S) = \begin{cases} \underset{j}{OPH}(\pi(S)), & \text{if } \underset{j}{OPH}(\pi(S)) \neq E \\ rand(\text{new seed}), & \text{otherwise} \end{cases} \quad (12)$$

Our aim is to compare the performance of minwise hashing with the proposed hash function $\mathcal{H}$. In particular, we would like to evaluate the deviation in performance of $\mathcal{H}$ with respect to the performance of minwise hashing. Since $\mathcal{H}$ has the same collision probability as that of minwise hashing, we expect them to have similar performance. In addition, we would like to study the performance of simple strategies (EE and ENE) on real data.

4.1. Datasets

To evaluate the proposed bucketing scheme, we chose the following three publicly available datasets.

- **MNIST** Standard dataset of handwritten digit samples. The feature dimension is 784 with an average number of around 150 non-zeros. We use the standard partition of MNIST, which consists of 10000 data points in one set and 60000 in the other.

- **NEWS20** Collection of newsgroup documents. The feature dimension is 1,355,191 with 500 non-zeros on an average. We randomly split the dataset in two halves having around 10000 points in each partition.

- **WEBSPAM** Collection of emails documents. The feature dimension is 16,609,143 with 4000 non-zeros on an average. We randomly selected 70000 data points and generated two partitions of 35000 each.

These datasets cover a wide variety in terms of size, sparsity and task. In the above datasets, one partition, the bigger one in case of MNIST, was used for creating hash buckets and the other partition was used as the query set. All datasets were binarized by setting non-zero values to 1.

We perform a rigorous evaluation of these hash functions, by comparing their performances, over a wide range of choices for parameters K and L. In particular, we want to understand if there is a different effect of varying the parameters K and L on the performance of $\mathcal{H}$ as compared to minwise hashing. Given parameters K and L, we need $K \times L$ number of hash evaluations per data point. For minwise hashing, we need $K \times L$ independent permutations while for the other three schemes we bin the data into $k = K \times L$ bins using only one permutation.

For both WEBSPAM and NEWS20, we implemented all the combinations for $K = \{6, 8, 10, 12\}$ and $L = \{4, 8, 16, 32, 64, 128\}$. For MNIST, with only 784 features, we used $K = \{6, 8, 10, 12\}$ and $L = \{4, 8, 16, 32\}$.

We use two metrics for evaluating retrieval: (i) the fraction of the total number of points retrieved by the bucketing scheme per query, (ii) the recall at a given threshold T_0, defined as the ratio of retrieved elements having similarity, with the query, greater than T_0 to the total number of elements having similarity, with the query, greater than T_0. It is important to balance both of them, for instance in linear scan we retrieve everything, and so we always achieve a perfect recall. For a given choice of K and L, we report both of these metrics independently. For reporting the recall, we choose two values of threshold $T_0 = \{0.5,\ 0.8\}$. Since the implementation involves randomizations, we repeat the experiments for each combination of K and L 10 times, and report the average over these 10 runs.

Figure 7 presents the plots of the fraction of points retrieved per query corresponding to $K = 10$ for all the three datasets with different choices of L. Due to space constraint, here we only show the recall plots for various values of L and T_0 corresponding to $K = 10$ in Figure 8. The plots corresponding to $K = \{6, 8, 12\}$ are very similar.

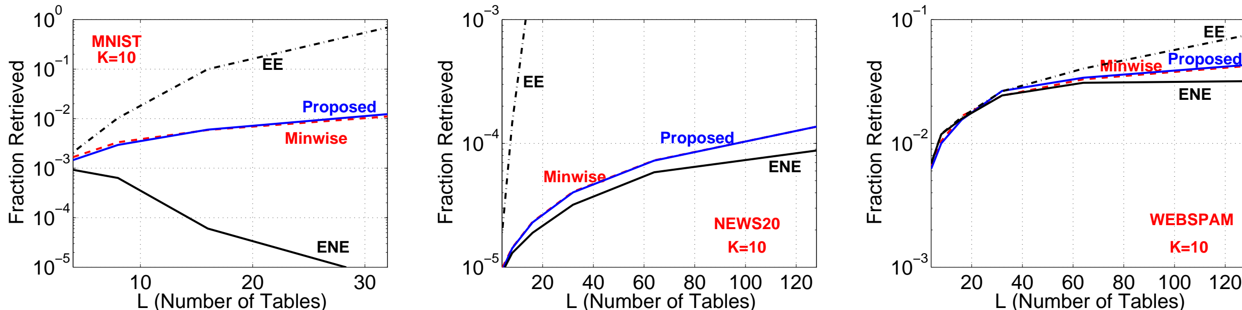

Figure 7. Fraction of points retrieved by the bucketing scheme per query corresponding to $K = 10$ shown over different choices of L. Results are averaged over 10 independent runs. Plots with $K = \{6, 8, 12\}$ are very similar in nature.

One can see that the performance of the proposed hash function $\mathcal{H}$ is indistinguishable from minwise hashing, irrespective of the sparsity of data and the choices of K, L and T_0. Thus, we conclude that minwise hashing can be replaced by $\mathcal{H}$ without loss in the performance and with a huge gain in query processing time (see Section 5).

Except for the WEBSPAM dataset, the EE scheme retrieves almost all the points, and so its not surprising that it achieves a perfect recall even at $T_0 = 0.5$. EE scheme treats the event of simultaneous empty bins as a match. The probability of this event is high in general for sparse data, especially for large k, and therefore even non-similar points have significant chance of being retrieved. On the other hand, ENE shows the opposite behavior. Simultaneous empty bins are highly likely even for very similar sparse vectors, but ENE treats this event as a rejection, and therefore we can see that as $k = K \times L$ increases, the recall values starts decreasing even for the case of $T_0 = 0.8$. WEBSPAM has significantly more nonzeros, so the occurrence of empty bins is rare for small k. Even in WEBSPAM, we observe an undesirable deviation in the performance of EE and ENE with $K \geq 10$.

5. Reduction in Computation Cost

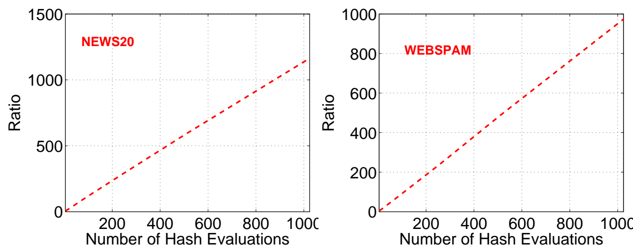

Figure 9. Ratio of time taken by minwise hashing to the time taken by our proposed scheme with respect to the number of hash evaluations, on the NEWS20 and WEBSPAM datasets.

5.1. Query Processing Cost Reduction

Let d denote the average number of nonzeros in the dataset. For running a (K, L)-parameterized LSH algorithm, we need to generate $K \times L$ hash evaluations of a given query vector (Indyk & Motwani, 1998). With minwise hashing, this requires storing and processing $K \times L$ different permutations. The total computation cost for simply processing a query is thus $O(dKL)$.

On the other hand, generating $K \times L$ hash evaluations using the proposed hash function $\mathcal{H}$ requires only processing a single permutation. It involves evaluating $K \times L$ minimums with one permutation hashing and one pass over the $K \times L$ bins to reassign empty values via "rotation" (see Figure 4). Overall, the total query processing cost is $O(d+KL)$. This is a massive saving over minwise hashing.

To verify the above claim, we compute the ratio of the time taken by minwise hashing to the time taken by our proposed scheme $\mathcal{H}$ for NEWS20 and WEBSPAM dataset. We randomly choose 1000 points from each dataset. For every vector, we compute 1024 hash evaluations using minwise hashing and the proposed $\mathcal{H}$. The plot of the ratio with the number of hash evaluations is shown in Figure 9. As expected, with the increase in the number of hash evaluations minwise hashing is linearly more costly than our proposal.

Figure 9 does not include the time required to generate permutations. Minwise hashing requires storing $K \times L$ random permutation in memory while our proposed hash function $\mathcal{H}$ only needs to store 1 permutation. Each fully random permutation needs a space of size D. Thus with minwise hashing storing $K \times L$ permutations take $O(KLD)$ space which can be huge, given that D in billions is common in practice and can even run into trillions. Approximating these $K \times L$ permutations using D cheap universal hash functions (Carter & Wegman, 1977; Nisan, 1990; Mitzenmacher & Vadhan, 2008) reduces the memory cost but comes with extra computational burden of evaluating $K \times L$ universal hash function for each query. These are not our main concerns as we only need one permutation.

5.2. Reduction in Total Query Time

The total query complexity of the LSH algorithm for retrieving approximate near neighbor using minwise hashing

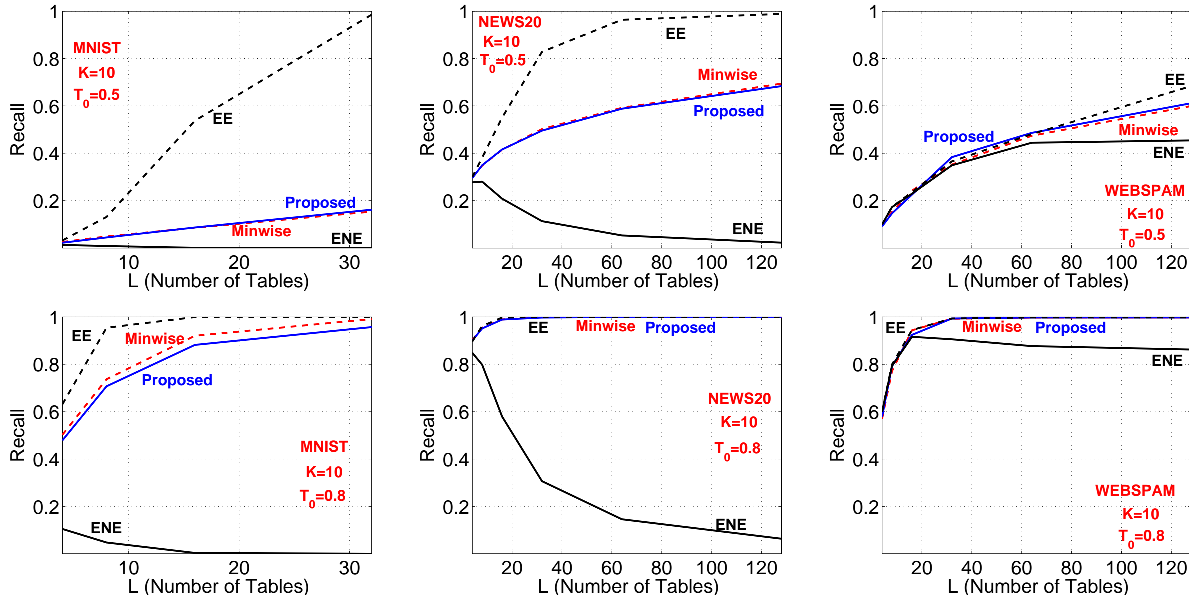

Figure 8. Recall values of points having similarity with the query greater than $T_0 = \{0.5, 0.8\}$ corresponding to $K = 10$ shown over different choices of L. Results are averaged over 10 independent runs. Plots with $K = \{6, 8, 12\}$ are very similar in nature.

is the sum of the query processing cost and the cost incurred in evaluating retrieved candidates. The total running cost of LSH algorithm is dominated by the query processing cost (Indyk & Motwani, 1998; Dasgupta et al., 2011). In theory, the number of data points retrieved using the appropriate choice of LSH parameters (K, L), in the worst case, is $O(L)$ (Indyk & Motwani, 1998). The total cost of evaluating retrieved candidates in brute force fashion is $O(dL)$. Thus, the total query time of LSH based retrieval with minwise hashing is $O(dKL + dL) = O(dKL)$. Since, both scheme behaves very similarly for any given (K, L), the total query time with the proposed scheme is $O(KL+d+dL)$ which comes down to $O(KL + dL)$. This analysis ignores the effect of correlation in the LSH algorithm but we can see from the plots that the effect is negligible.

The need for evaluating all the retrieved data points based on exact similarity leads to $O(dL)$ cost. In practice, an efficient estimate of similarity can be obtained by using the same hash evaluations used for indexing (Shrivastava & Li, 2012). This decreases the re-ranking cost further.

5.3. Pre-processing Cost Reduction

The LSH algorithm needs a costly pre-processing step which is the bucket assignment for all N points in the collection. The bucket assignment takes in total $O(dNKL)$ with minwise hashing. The proposed hashing scheme $\mathcal{H}$ reduces this cost to $O(NKL + Nd)$.

6. Conclusion

With the explosion of data in modern applications, the brute force strategy of scanning all data points for searching for near neighbors is prohibitively expensive, especially in user-facing applications like search. LSH is popular for achieving sub-linear near neighbor search. Minwise hashing, which belongs to the LSH family, is one of the basic primitives in many “big data” processing algorithms. In addition to near-neighbor search, these techniques are frequently used in popular operations like near-duplicate detection (Broder, 1997; Manku et al., 2007; Henzinger, 2006), all-pair similarity (Bayardo et al., 2007), similarity join/record-linkage (Koudas & Srivastava, 2005), temporal correlation (Chien & Immorlica, 2005), etc.

Current large-scale data processing systems deploying indexed tables based on minwise hashing suffer from the problem of costly processing. In this paper, we propose a new hashing scheme based on a novel “rotation” technique to densify one permutation hashes (Li et al., 2012). The obtained hash function possess the properties of minwise hashing, and at the same time it is very fast to compute.

Our experimental evaluations on real data suggest that the proposed hash function offers massive savings in computation cost compared to minwise hashing. These savings come without any trade-offs in the performance measures.

Acknowledgement

The work is supported by NSF-III-1360971, NSF-Bigdata-1419210, ONR-N00014-13-1-0764, and AFOSR-FA9550-13-1-0137. We thank the constructive review comments from SIGKDD 2013, NIPS 2013, and ICML 2014, for improving the quality of the paper. The sample matlab code for our proposed hash function is available upon request.

References

Agarwal, Alekh, Chapelle, Olivier, Dudik, Miroslav, and Langford, John. A reliable effective terascale linear learning system. Technical report, arXiv:1110.4198, 2011.

Andoni, Alexandr and Indyk, Piotr. E2lsh: Exact euclidean locality sensitive hashing. Technical report, 2004.

Bayardo, Roberto J., Ma, Yiming, and Srikant, Ramakrishnan. Scaling up all pairs similarity search. In *WWW*, pp. 131–140, 2007.

Bottou, Leon. http://leon.bottou.org/projects/sgd.

Broder, Andrei Z. On the resemblance and containment of documents. In *the Compression and Complexity of Sequences*, pp. 21–29, Positano, Italy, 1997.

Broder, Andrei Z., Glassman, Steven C., Manasse, Mark S., and Zweig, Geoffrey. Syntactic clustering of the web. In *WWW*, pp. 1157 – 1166, Santa Clara, CA, 1997.

Broder, Andrei Z., Charikar, Moses, Frieze, Alan M., and Mitzenmacher, Michael. Min-wise independent permutations. In *STOC*, pp. 327–336, Dallas, TX, 1998.

Carter, J. Lawrence and Wegman, Mark N. Universal classes of hash functions. In *STOC*, pp. 106–112, 1977.

Chandra, Tushar, Ie, Eugene, Goldman, Kenneth, Llinares, Tomas Lloret, McFadden, Jim, Pereira, Fernando, Redstone, Joshua, Shaked, Tal, and Singer, Yoram. Sibyl: a system for large scale machine learning.

Chapelle, Olivier, Haffner, Patrick, and Vapnik, Vladimir N. Support vector machines for histogram-based image classification. *IEEE Trans. Neural Networks*, 10(5):1055–1064, 1999.

Chien, Steve and Immorlica, Nicole. Semantic similarity between search engine queries using temporal correlation. In *WWW*, pp. 2–11, 2005.

Dasgupta, Anirban, Kumar, Ravi, and Sarlós, Tamás. Fast locality-sensitive hashing. In *KDD*, pp. 1073–1081, 2011.

Fan, Rong-En, Chang, Kai-Wei, Hsieh, Cho-Jui, Wang, Xiang-Rui, and Lin, Chih-Jen. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

Fetterly, Dennis, Manasse, Mark, Najork, Marc, and Wiener, Janet L. A large-scale study of the evolution of web pages. In *WWW*, pp. 669–678, Budapest, Hungary, 2003.

Friedman, Jerome H., Baskett, F., and Shustek, L. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, 24:1000–1006, 1975.

Henzinger, Monika Rauch. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pp. 284–291, 2006.

Indyk, Piotr and Motwani, Rajeev. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pp. 604–613, Dallas, TX, 1998.

Joachims, Thorsten. Training linear svms in linear time. In *KDD*, pp. 217–226, Pittsburgh, PA, 2006.

Koudas, Nick and Srivastava, Divesh. Approximate joins: Concepts and techniques. In *VLDB*, pp. 1363, 2005.

Li, Ping and Church, Kenneth W. A sketch algorithm for estimating two-way and multi-way associations. *Computational Linguistics (Preliminary results appeared in HLT/EMNLP 2005)*, 33(3):305–354, 2007.

Li, Ping and König, Arnd Christian. b-bit minwise hashing. In *Proceedings of the 19th International Conference on World Wide Web*, pp. 671–680, Raleigh, NC, 2010.

Li, Ping, Church, Kenneth W., and Hastie, Trevor J. Conditional random sampling: A sketch-based sampling technique for sparse data. In *NIPS*, pp. 873–880, Vancouver, BC, Canada, 2006.

Li, Ping, König, Arnd Christian, and Gui, Wenhao. b-bit minwise hashing for estimating three-way similarities. In *Advances in Neural Information Processing Systems*, Vancouver, BC, 2010.

Li, Ping, Shrivastava, Anshumali, Moore, Joshua, and König, Arnd Christian. Hashing algorithms for large-scale learning. In *NIPS*, Granada, Spain, 2011.

Li, Ping, Owen, Art B, and Zhang, Cun-Hui. One permutation hashing. In *NIPS*, Lake Tahoe, NV, 2012.

Li, Ping, Shrivastava, Anshumali, and König, Arnd Christian. b-bit minwise hashing in practice. In *Internetware*, Changsha, China, 2013.

Manku, Gurmeet Singh, Jain, Arvind, and Sarma, Anish Das. Detecting Near-Duplicates for Web-Crawling. In *WWW*, Banff, Alberta, Canada, 2007.

Mitzenmacher, Michael and Vadhan, Salil. Why simple hash functions work: exploiting the entropy in a data stream. In *SODA*, 2008.

Nisan, Noam. Pseudorandom generators for space-bounded computations. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC, pp. 204–212, 1990.

Shalev-Shwartz, Shai, Singer, Yoram, and Srebro, Nathan. Pegasos: Primal estimated sub-gradient solver for svm. In *ICML*, pp. 807–814, Corvalis, Oregon, 2007.

Shrivastava, Anshumali and Li, Ping. Fast near neighbor search in high-dimensional binary data. In *ECML*, 2012.

Shrivastava, Anshumali and Li, Ping. Beyond pairwise: Provably fast algorithms for approximate k-way similarity search. In *NIPS*, Lake Tahoe, NV, 2013.

Tong, Simon. Lessons learned developing a practical large scale machine learning system. http://googleresearch.blogspot.com/2010/04/lessons-learned-developing-practical.html, 2008.

Weinberger, Kilian, Dasgupta, Anirban, Langford, John, Smola, Alex, and Attenberg, Josh. Feature hashing for large scale multitask learning. In *ICML*, pp. 1113–1120, 2009.