Fast Near Neighbor Search in High-Dimensional Binary Data

Anshumali Shrivastava and Ping Li

Cornell University, Ithaca NY 14853, USA

Abstract. Numerous applications in search, databases, machine learning, and computer vision, can benefit from efficient algorithms for near neighbor search. This paper proposes a simple framework for *fast near neighbor search in high-dimensional binary data*, which are common in practice (e.g., text). We develop a very simple and effective strategy for sub-linear time near neighbor search, by creating hash tables directly using the bits generated by b-bit minwise hashing. The advantages of our method are demonstrated through thorough comparisons with two strong baselines: *spectral hashing* and *sign (1-bit) random projections*.

1 Introduction

As a fundamental problem, the task of *near neighbor search* is to identify a set of data points which are "most similar" to a query data point. Efficient algorithms for near neighbor search have numerous applications in the context of search, databases, machine learning, recommending systems, computer vision, etc.

Consider a data matrix $\mathbf{X} \in \mathbb{R}^{n \times D}$, i.e., n samples in D dimensions. In modern applications, both n and D can be large, e.g., billions or even larger [1]. Intuitively, near neighbor search may be accomplished by two simple strategies. The first strategy is to pre-compute and store all pairwise similarities at $O(n^2)$ space, which is only feasible for small number of samples (e.g., $n < 10^5$).

The second simple strategy is to scan all n data points and compute similarities on the fly, which however also encounters difficulties: (i) The data matrix $\mathbf{X}$ itself may be too large for the memory. (ii) Computing similarities on the fly can be too time-consuming when the dimensionality D is high. (iii) The cost of scanning all n data points is prohibitive and may not meet the demand in user-facing applications (e.g., search). (iv) Parallelizing linear scans will not be energy-efficient if a significant portion of the computations is not needed.

Our proposed (simple) solution is built on the recent work of b-bit minwise hashing [2, 3] and is specifically designed for binary high-dimensional data.

1.1 Binary, Ultra-High Dimensional Data

For example, consider a Web-scale term-doc matrix $\mathbf{X} \in \mathbb{R}^{n \times D}$ with each row representing one Web page. Then roughly $n = O(10^{10})$. Assuming 10^5 common English words, then the dimensionality $D = O(10^5)$ using the uni-gram model

and $D = O(10^{10})$ using the bi-gram model. Certain industry applications used 5-grams [4–6] (i.e., $D = O(10^{25})$ is conceptually possible). Usually, when using 3- to 5-grams, most of the grams only occur at most once in each document. It is thus common to utilize only binary data when using n-grams.

1.2 *b*-bit Minwise Hashing

Minwise hashing [5] is a standard technique for efficiently computing set similarities in the context of search. The method mainly focuses on binary (0/1) data, which can be viewed as sets. Consider two sets $S_1, S_2 \subseteq \Omega = \{0, 1, 2, ..., D-1\}$, the method applies a random permutation $\pi : \Omega \to \Omega$ on S_1 and S_2 and utilizes

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

to estimate R, the resemblance between S_1 and S_2. A prior common practice was to store each hashed value, e.g., $\min(\pi(S_1))$, using 64 bits [6], which can lead to prohibitive storage and computational costs in certain industrial applications [7]. b-bit minwise hashing [2] is a simple solution by storing only the lowest b bits of each hashed value. For convenience, we define

$$z_j = \min(\pi(S_j)), \qquad z_j^{(b)} = \text{the lowest } b \text{ bits of } z_j.$$

Assuming D is large, [2] derived a new collision probability:

$$\begin{aligned}
&P_b(R) = \mathbf{Pr}\left(z_1^{(b)} = z_2^{(b)}\right) = C_{1,b} + (1 - C_{2,b})\, R \qquad (2)\\
&r_1 = \frac{f_1}{D}, \quad r_2 = \frac{f_2}{D}, \quad f_1 = |S_1|, \quad f_2 = |S_2|\\
&C_{1,b} = A_{1,b}\frac{r_2}{r_1 + r_2} + A_{2,b}\frac{r_1}{r_1 + r_2}, \qquad C_{2,b} = A_{1,b}\frac{r_1}{r_1 + r_2} + A_{2,b}\frac{r_2}{r_1 + r_2},\\
&A_{1,b} = \frac{r_1\left[1 - r_1\right]^{2^b - 1}}{1 - \left[1 - r_1\right]^{2^b}}, \qquad A_{2,b} = \frac{r_2\left[1 - r_2\right]^{2^b - 1}}{1 - \left[1 - r_2\right]^{2^b}}.
\end{aligned}$$

This result suggests an unbiased estimator of R from k permutations π_1, ..., π_k:

$$\hat{R}_b = \frac{\hat{P}_b - C_{1,b}}{1 - C_{2,b}}, \qquad \hat{P}_b = \frac{1}{k}\sum_{j=1}^{k} 1\{z_{1,\pi_j}^{(b)} = z_{2,\pi_j}^{(b)}\} \tag{3}$$

whose variance would be

$$\mathrm{Var}\left(\hat{R}_b\right) = \frac{1}{k}\frac{\left[C_{1,b} + (1 - C_{2,b})R\right]\left[1 - C_{1,b} - (1 - C_{2,b})R\right]}{\left[1 - C_{2,b}\right]^2} \tag{4}$$

The advantage of b-bit minwise hashing can be demonstrated through the "variance-space" trade-off: $\mathrm{Var}\left(\hat{R}_b\right) \times b$. Basically, when the data are highly similar, a small b (e.g., 1 or 2) may be good enough. However, when the data are not very similar, b can not be too small.

1.3 Our Proposal for Sub-Linear Time Near Neighbor Search

Our proposed method is simple, by directly using the bits generated from b-bit minwise hashing to build hash tables, which allow us to search near neighbors in sub-linear time (i.e., no need to scan all data points).

Specifically, we hash the data points using k random permutations and store each hash value using b bits (e.g., $b \leq 4$). For each data point, we concatenate the resultant $B = b \times k$ bits as a *signature*. The size of the space is $2^B = 2^{b \times k}$, which is not too large for small b and k (e.g., $bk = 16$). This way, we create a table of 2^B buckets, numbered from 0 to $2^B - 1$; and each bucket stores the pointers of the data points whose signatures match the bucket number. In the testing phrase, we apply the same k permutations to a query data point to generate a bk-bit signature and only search data points in the corresponding bucket.

Of course, using only one hash table will likely miss many true near neighbors. As a remedy, we generate (using independent random permutations) L hash tables; and the query result is the union of the data points retrieved in L tables.

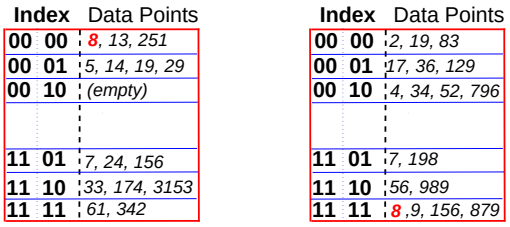

Fig. 1. An example of hash tables, with $b = 2$, $k = 2$, and $L = 2$.

In the example in Figure 1, we choose $b = 2$ bits and $k = 2$ permutations, i.e., one hash table has 2^4 buckets. Given n data points, we apply $k = 2$ permutations and store $b = 2$ bits of each hashed value to generate n (4-bit) signatures. Consider data point 8. After $k = 2$ permutations, the lowest b-bits of the hashed values are 00 and 00. Therefore, its signature is 0000 in binary and hence we place a pointer to data point 8 in bucket number 0 (as in the left panel of Figure 1).

In this example, we choose to build $L = 2$ tables. Thus we apply another $k = 2$ permutations and place the n data points to the second table (as in the right panel of Figure 1) according to their signatures. This time, the signature of data point 8 becomes 1111 in binary and hence we place it in the last bucket.

Suppose in the testing phrase, the two (4-bit) signatures of a new data point are 0000 and 1111, respectively. We then only search the near neighbors in the set $\{8, 9, 13, 156, 251, 879\}$, which is much smaller than the set of n data points.

2 Other Methods for Efficient Near Neighbor Search

Developing efficient algorithms for finding near neighbors has been an active research topic since the early days of modern computing. For example, K-D trees [12] and variants often work reasonably well in very low-dimensional data.

Our technique can be viewed as an instance of *Locality Sensitive Hashing (LSH)* [13–15], which represents a very general family of algorithms for near neighbor search. The performance of any LSH scheme depends on the underlying algorithm. Our idea of directly using the bits generated from b-bit miniwise hashing to build hash tables is novel and requires own analysis.

The effectiveness of our proposed algorithm can be demonstrated through thorough comparisons with strong baselines. In this paper, we focus on *spectral hashing (SH)* [8] and the LSH based on *sign random projections* [9–11].

2.1 Centered and Noncentered Spectral Hashing (SH-C, SH-NC)

Spectral hashing (SH) [8] is a representative example of "learning-based hashing" algorithms, which typically require a (very) expensive training step. It appears that more recent learning-based hashing algorithms, e.g., [16, 17] have not shown a definite advantage over SH. Moreover, other learning-based search algorithms are often much more complex than SH. Thus, to ensure our comparison study is fair and repeatable, we focus on SH.

Given a data matrix $\mathbf{X} \in \mathbb{R}^{n \times D}$, SH first computes the top eigenvectors of the sample covariance matrix and maps the data according to the top eigenvectors. The mapped data are then thresholded to be binary (0/1), which are the hash code bits for near neighbor search. Clearly, for massive high-dimensional data, SH is prohibitively memory-intensive and time-consuming. Also, storing these eigenvectors (for testing new data) requires excessive disk space when D is large.

We made two modifications to the original SH implementation [8]. Here, we quote from their Matlab code [8] to illustrate the major computational cost:

```
[pc, l] = eigs(cov(X), npca);   X = X * pc;
```

Our first modification is to replace the eigen-decomposition by SVD, which avoids materializing the covariance matrix (of size $D \times D$). That is, we first remove the mean from $\mathbf{X}$ (called "centering") and then apply Matlab "svds" (instead of "eigs") on the centered $\mathbf{X}$. This modification can substantially reduce the memory consumption without altering the results.

The centering step (or directly using "eigs"), however, can be disastrous because after centering the data are no longer sparse. For example, with centering, training merely 4000 data points in about 16 million dimensions (i.e., the *Webspam* dataset) took 2 days in a workstation with 96GB memory, to obtain 192-bit hash codes. Storing those 192 eigenvectors consumed 24GB disk space after compression (using the "-v7.3" save option in Matlab).

In order to make reliable comparisons with SH, we implemented both centered version (SH-C) and noncentered version (SH-NC). Since we focus on binary

data in this study, it is not clear if centering is at all necessary. In fact, our experiments will show that SH-NC often perform similarly as SH-C.

Even with the above two modifications, SH-NC is still very expensive. For example, it took over one day for training 35,000 data points of the *Webspam* dataset to produce 256-bit hash code. The prohibitive cost for storing the eigenvectors remains the same as SH-C (about 32GB for 256 bits).

Once the hash code has been generated, searching for near neighbors amounts to finding data points whose hash codes are closest (in *hamming distance*) to the hash code of the query point [8]. Strictly speaking, there is no proof that one can build hash tables using the bits of SH in the sense of LSH. Therefore, to ensure that our comparisons are fair and repeatable, we only experimentally compare the code quality of SH with b-bit minwise hashing, in Section 3.

2.2 Sign Random Projections (SRP)

The method of random projections utilizes a random matrix $P \in \mathbb{R}^{D\times k}$ whose entries are i.i.d. normal, i.e., $P_{ij} \sim N(0,1)$. Consider two sets S_1, S_2. One first generates two projected vectors $v_1, v_2 \in \mathbb{R}^k$: $v_{1j} = \sum_{i\in S_1} P_{ij}, \quad v_{2j} = \sum_{i\in S_2} P_{ij}$, and then estimates the size of intersection $a = |S_1 \cap S_1|$ by $\frac{1}{k}\sum_{j=1}^{k} v_{1j}v_{2j}$.

It turns out that this method is not accurate as shown in [3]. Interestingly, using only the signs of the projected data can be much more accurate (in terms of variance per bit). Basically, the method of *sign random projections* estimates the similarity using the following collision probability:

$$\mathbf{Pr}\left(\text{sign}(v_{1,j}) = \text{sign}(v_{2,j})\right) = 1 - \frac{\theta}{\pi}, \qquad j = 1, 2, ..., k, \tag{5}$$

where $\theta = \cos^{-1}\left(\frac{a}{\sqrt{f_1 f_2}}\right)$ is the angle. This formula was presented in [9] and popularized by [10]. The variance was analyzed and compared in [11].

We will first compare SRP with b-bit minwise hashing in terms of hash code quality. We will then build hash tables to compare the performance in sub-linear time near neighbor search. See Appendix A for the variance-space comparisons.

3 Comparing Hash Code Quality

We tested three algorithms (b-bit, SH, SRP) on two binary datasets: *Webspam* and *EM30k*. The *Webspam* dataset was used in [3], which also demonstrated that using the binary-quantized version did not result in loss of classification accuracy. For our experiments, we sampled $n = 70,000$ examples from the original dataset. The dimensionality is $D = 16,609,143$.

The *EM30k* dataset was used in [18] to demonstrate the effectiveness of image feature expansions. We sampled $n = 30,000$ examples from the original dataset. The dimensionality is $D = 34,950,038$.

3.1 The Evaluation Procedure

We evaluate the algorithms in terms of the **precision-recall** curves. Basically, for each data point, we sort all other data points in the dataset in descending (estimated) similarities using the hash code of length B. We walk down the list (up to 1000 data points) to retrieve the "top T" data points, which are most similar (in terms of the original similarities) to that query point. We choose $T = 5$, $T = 10$, $T = 20$, and $T = 50$. The precision and recall are defined as:

$$\text{Precision} = \frac{\#\text{ True Positive}}{\#\text{Retrieved}}, \qquad \text{Recall} = \frac{\#\text{ True Positive}}{T} \tag{6}$$

We vary # retrieved data points from 1 to 1000 spaced at 1, to obtain continuous precision-recall curves. The final results are averaged over all the test data points.

3.2 Experimental Results on Webspam (4000)

We first experimented with 4000 data points from the *Webspam* dataset, which is small enough so that we could train the centered version of spectral hashing (i.e., SH-C). On a workstation with 96 GB memory, SH-C took about 2 days and used about 90GB memory at the peak, to produce 192-bit hash code.

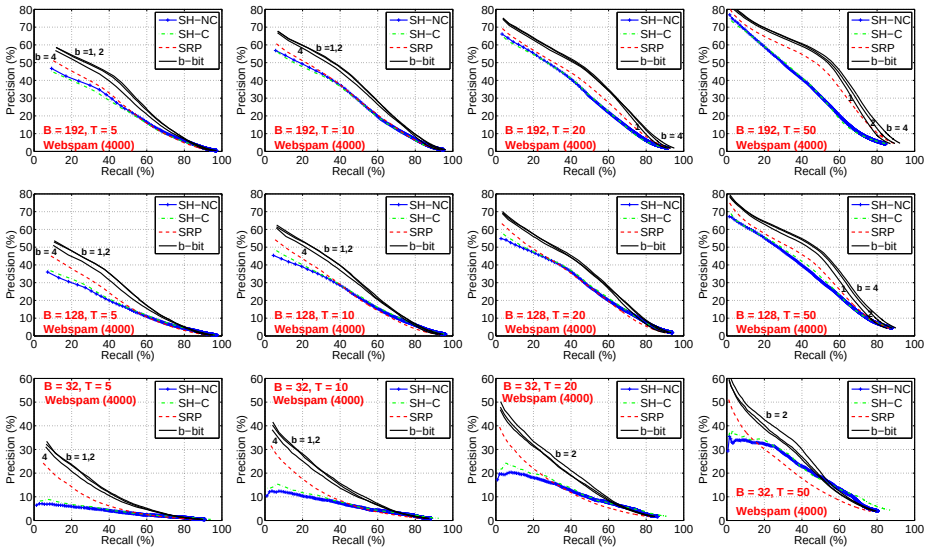

Fig. 2. Precision-recall curves (the higher the better) for all four methods (SRP, b-bit, SH-C, and SH-NC) on a small subset (4000 data points) of the *Webspam* dataset. The task is to retrieve the top T near neighbors (for $T = 5$, 10, 20, 50). B is the bit length.

Figure 2 presents the results of b-bit hashing, SH-C, SH-NC, and SRP in terms of the precision-recall curves (the higher the better), for $B = 192$, 128, and 32 bits. Basically, for b-bit hashing, we choose $b = 1, 2, 4$ and k so that $b \times k = B$. For example, if $B = 192$ and $b = 2$, then $k = 96$. As analyzed in [2], for a pair of data points which are very similar, then using smaller b will outperform using larger b in terms of the variance-space tradeoff. Thus, it is not surprising if $b = 1$ or 2 shows better performance than $b = 4$ for this dataset.

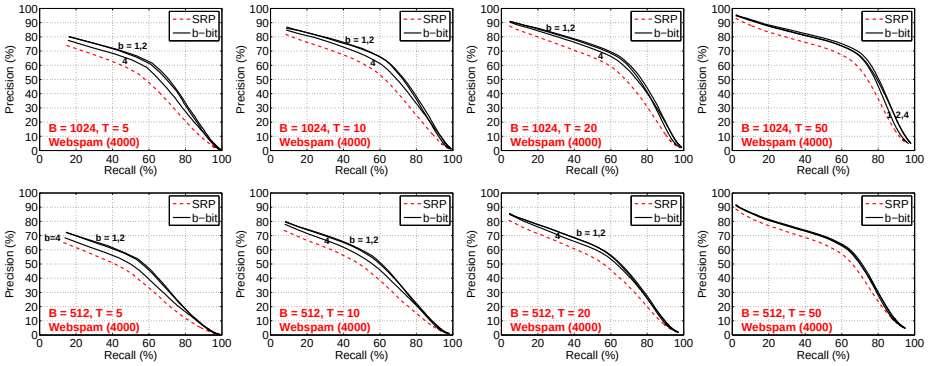

Fig. 3. Precision-recall curves for SRP and b-bit hashing on 4000 data points of the *Webspam* dataset using 1024-bit and 512-bit codes, for which we could not run SH.

Figure 3 compares b-bit hashing with SRP with much longer hash code (1024 bits and 512 bits). These two figures demonstrate that:

- SH-C and SH-NC perform very similarly in this case, while SH-NC is substantially less expensive (several hours as opposed to 2 days).
- SRP is better than SH and is noticeably worse than b-bit hashing for all b.

We need to clarify how we obtained the gold-standard list for each method. For b-bit hashing, we used the original resemblances. For SRP, we used the original cosines. For SH, following [8] we used the original Euclidian distances.

3.3 Experimental Results on Webspam (35000)

Based on 35000 (which are more reliable than 4000) data points of the *Webspam* dataset, Figure 4 again illustrates that SRP is better than SH-NC and is worse than b-bit hashing. Note that we can not train SH-C on 35000 data points. We limited the SH bit length to 256 because 256 eigenvectors already occupied 32GB disk space after compression. On the other hand, we can use much longer code lengths for the two inexpensive methods, SRP and b-bit hashing. As shown in Figure 5, for 512 bits and 1024 bits, b-bit hashing still outperformed SRP.

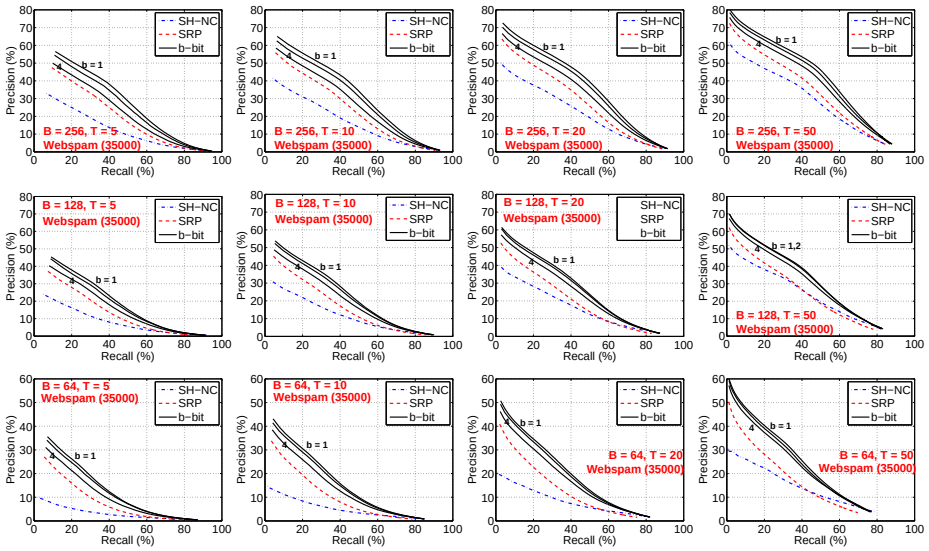

Fig. 4. Precision-recall curves for three methods (SRP, b-bit, and SH-NC) on 35000 data points of the *Webspam* dataset. Again, b-bit outperformed SH and SRP.

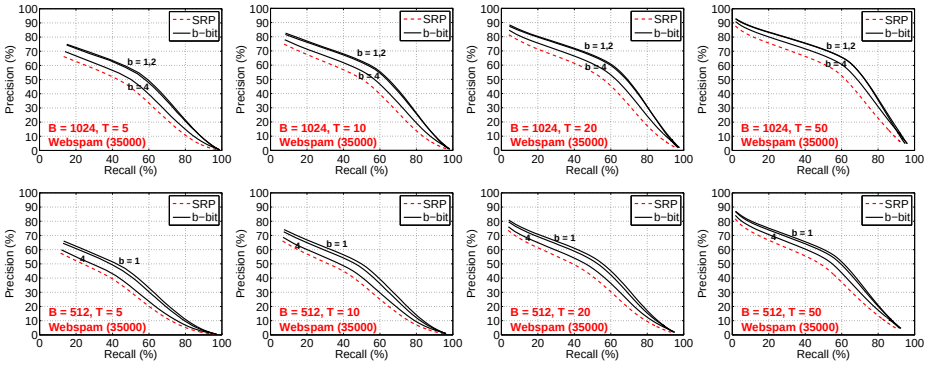

Fig. 5. Precision-recall curves for SRP and b-bit minwise hashing on 35000 data points of the *Webspam* dataset, for longer code lengths (1024 bits and 512 bits).

3.4 EM30k (15000)

For this dataset, as the dimensionality is so high, it is difficult to train SH at a meaningful scale. Therefore, we only compare SRP with b-bit hashing in Figure 6, which clearly demonstrates the advantage of b-bit minwise hashing.

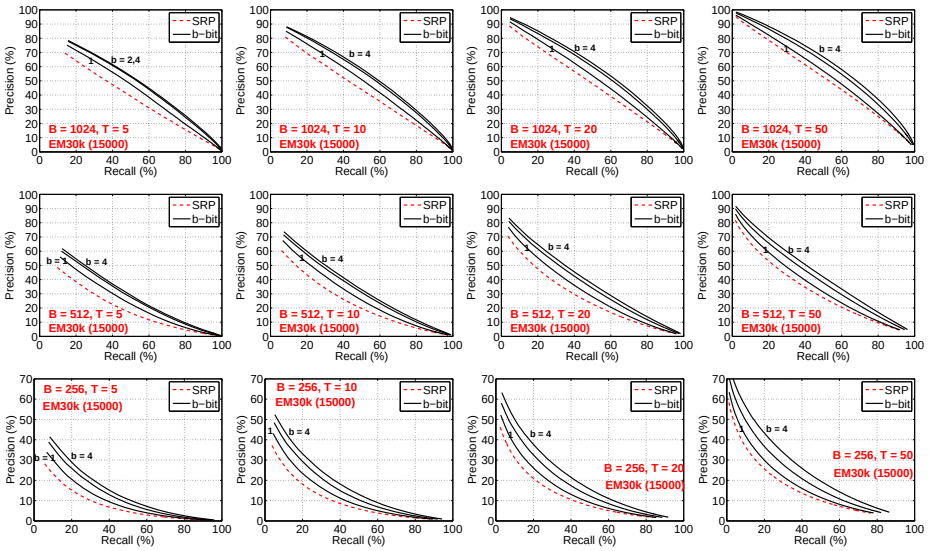

Fig. 6. Precision-recall curves for SRP and b-bit minwise hashing on 15000 data points of the *EM30k* dataset.

4 Sub-Linear Time Near Neighbor Search

We have presented our simple strategy in Section 1.3 and Figure 1. Basically, we apply k permutations to generate one hash table. For each permutation, we store each hashed data using only b bits and concatenate k b-bit strings to form a *signature*. The data point (in fact, only its pointer) is placed in a table of 2^B buckets ($B = b \times k$). We generate L such hash tables using independent permutations. In the testing phrase, given a query data point, we apply the same random permutations to generate signatures and only search for data points (called the *candidate set*) in the corresponding buckets.

In the next step, there are many possible ways of selecting near neighbors from the candidate set. For example, suppose we know the exact nearest neighbor has a resemblance R_0 and our goal is to retrieve T points whose resemblances to the query point are $\geq cR_0$ ($c < 1$). Then we just need to keep scanning the data points in the candidate set until we encounter T such data points, assuming that we are able to compute the exact similarities. In reality, however, we often do not know the desired threshold R_0, nor do we have a clear choice of c. Also, we usually can not afford to compute the exact similarities.

To make our comparisons easy and fair, we simply re-rank all the retrieved data points and compute the precision-recall curves by walking down the list of data points (up to 1000) sorted by descending order of similarities. For simplicity, to re-rank the data points in the candidate set, we use the estimated similarities from $k \times L$ permutations and b bits per hashed value.

4.1 Theoretical Analysis

Collision Probability. Eq. (2) presents the basic collision probability $P_b(R)$. After the hash tables have been constructed with parameters b, k, L, we can easily write down the overall collision probability (a commonly used measure):

$$P_{b,k,L}(R) = 1 - \left(1 - P_b^k(R)\right)^L \tag{7}$$

which is the probability at which a data point with similarity R will match the signature of the query data point at least in one of the L hash tables.

For simplicity, in this section we will always assume that the data are sparse, i.e., $r_1 \to 0$, $r_2 \to 0$ in (2) which leads to convenient simplification of (2):

$$P_b(R) = \frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right) R. \tag{8}$$

Required Number of Tables L. Suppose we require $P_{b,k,L}(R) > 1 - \delta$, then the number of hash tables (denoted by L) should be

$$L \geq \frac{\log 1/\delta}{\log\left(\frac{1}{1-P_b^k(R)}\right)} \tag{9}$$

which can be satisfied by a combination of b and k. The optimal choice depends on the threshold level R, which is often unfortunately unknown in practice.

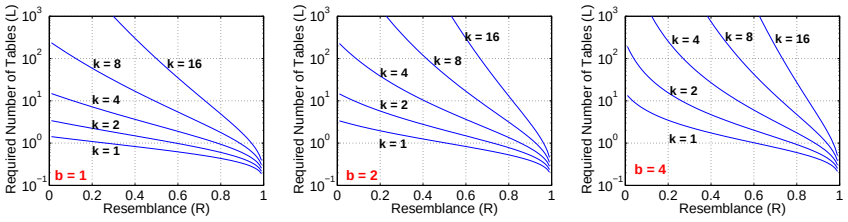

Fig. 7. Required number of tables (L) as in (9), without the $\log 1/\delta$ term. The numbers in the plots should multiply by $\log 1/\delta$, which is about 3 when $\delta = 0.05$.

Number of Retrieved Points before Re-ranking. The expected number of total retrieved points (before re-ranking) is an integral, which involves the data distribution. For simplicity, by assuming a uniform distribution, the fraction of the data points retrieved before the re-ranking step would be (See Appendix B):

$$\int_0^1 P_{b,k,L}(tR)dt = 1 - \sum_{i=0}^{L} \binom{L}{i} (-1)^i \frac{1}{2^{bki}} \frac{1}{(2^b-1)R} \frac{\left((2^b-1)R+1\right)^{ki+1} - 1}{ki+1} \tag{10}$$

Figure 8 plots (10) to illustrate that the value is small for a range of parameters.

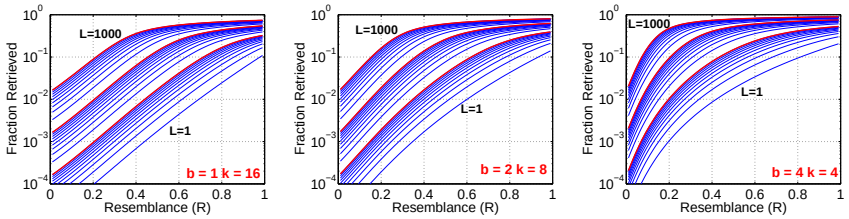

Fig. 8. Numerical values for (10), the fraction of retrieved points.

Threshold Analysis. To better view the threshold, one commonly used strategy is to examine the point R_0 where the 2nd derivative is zero (i.e., the inflection point of $P_{b,k,L}(R)$): $\left.\frac{\partial^2 P_{b,k,L}}{\partial R^2}\right|_{R_0} = 0$, which turns out to be:

$$R_0 = \frac{\left(\frac{k-1}{Lk-1}\right)^{1/k} - \frac{1}{2^b}}{1 - \frac{1}{2^b}} \tag{11}$$

Figure 9 plots (11). For example, suppose we fix $L = 100$ and $B = b \times k = 16$. If we use $b = 4$, then $R_0 \approx 0.52$. If we use $b = 2$, then $R_0 \approx 0.4$. In other words, a larger b is preferred if we expect that the near neighbors have low similarities.

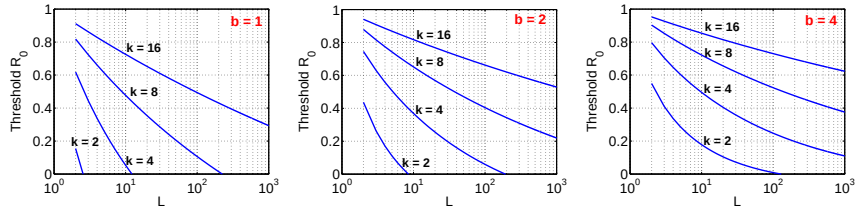

Fig. 9. The threshold R_0 computed by (11), i.e., inflection point of $P_{b,k,L}(R)$.

4.2 Experimental Results on the Webspam Dataset

We use 35,000 data points to build hash tables and another 35,000 data points for testing. We build hash tables from both b-bit minwise hashing and sign random projections, to conduct shoulder-by-shoulder comparisons.

Figure 10 plots the fractions of retrieved data points before re-ranking. b-bit hashing with $b = 1$ or 2 retrieves similar numbers of data points. This means, if we also see that the b-bit hashing (with $b = 1$ or 2) has better precision-recall curves than SRP, we know that b-bit hashing is definitely better.

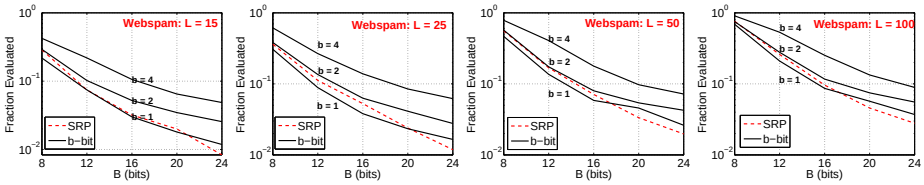

Fig. 10. Fractions of retrieved data points (before re-ranking) on the *Webspam* dataset.

Figures 11 and 12 plot the precision-recall curves for $L = 100$ and 50 tables, respectively, demonstrating the advantage of b-bit minwise hashing over SRP.

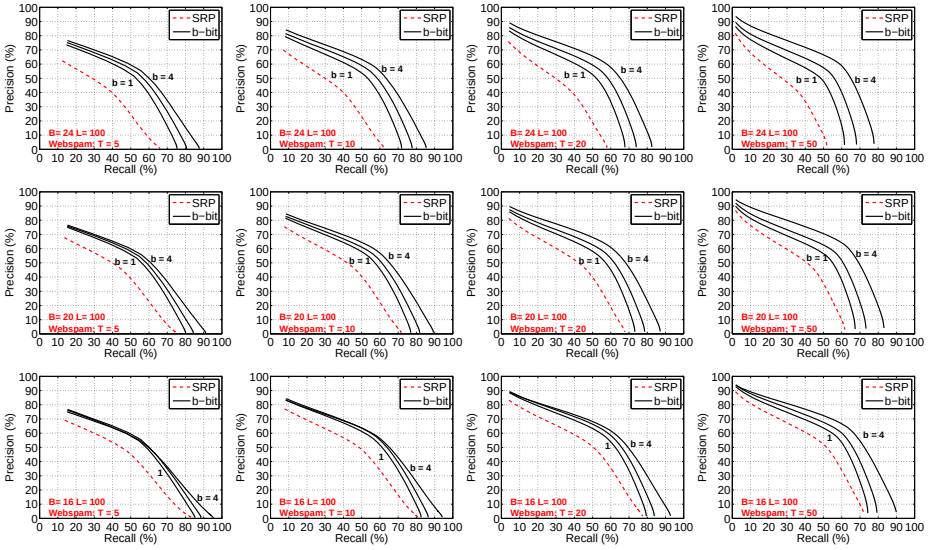

Fig. 11. Precision-recall curves for SRP and b-bit minwise hashing on the *Webspam* dataset using $L = 100$ tables, for top $T = 5$, 10, 20, and $T = 50$ near neighbors.

4.3 Experimental Results on EM30k Dataset

For this dataset, we choose 5000 data points (out of 30000) as the query points and use the rest 25000 points for building hash tables.

Figure 13 plots the # retrieved data points before the re-ranking step. We can see that b-bit hashing with $b = 2$ retrieves similar numbers of data points. Again, this means, if we also see that the b-bit hashing (with $b = 1$ or 2) has better precision-recall curves than SRP, then b-bit hashing is certainly better.

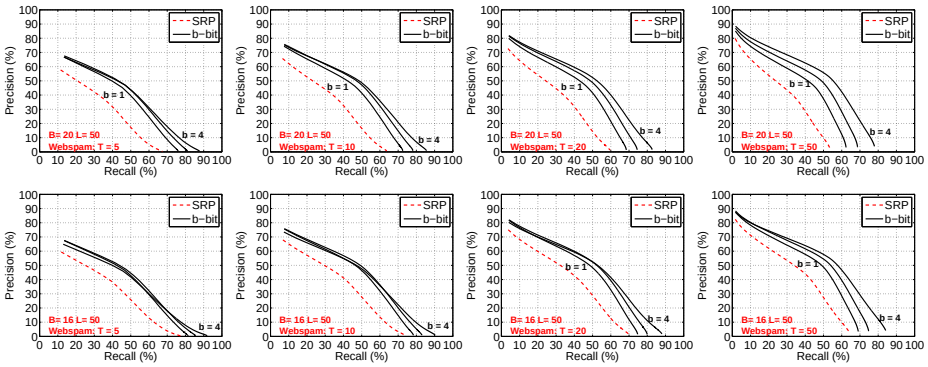

Fig. 12. Precision-Recall curves for SRP and b-bit minwise hashing on the *Webspam* dataset, using $L = 50$ tables.

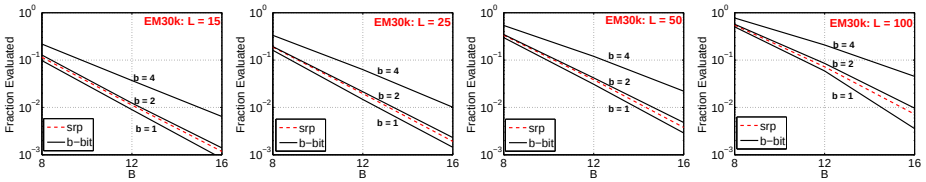

Fig. 13. Fractions of retrieved data points (before re-ranking) on the *EM30k* dataset.

Figure 14 presents the precision-recall curves for $L = 100$ tables, again demonstrating the advantage of b-bit hashing over SRP.

5 Conclusion

This paper reports the first study of directly using the bits generated by b-bit minwise hashing to construct hash tables, for achieving sub-linear time near neighbor search in high-dimensional binary data. Our proposed scheme is extremely simple and exhibits superb performance compared to two strong baselines: spectral hashing (SH) and sign random projections (SRP).

Acknowledgement

This work is supported by NSF (DMS-0808864, SES-1131848), ONR (YIP-N000140910911), and DARPA (FA-8650-11-1-7149).

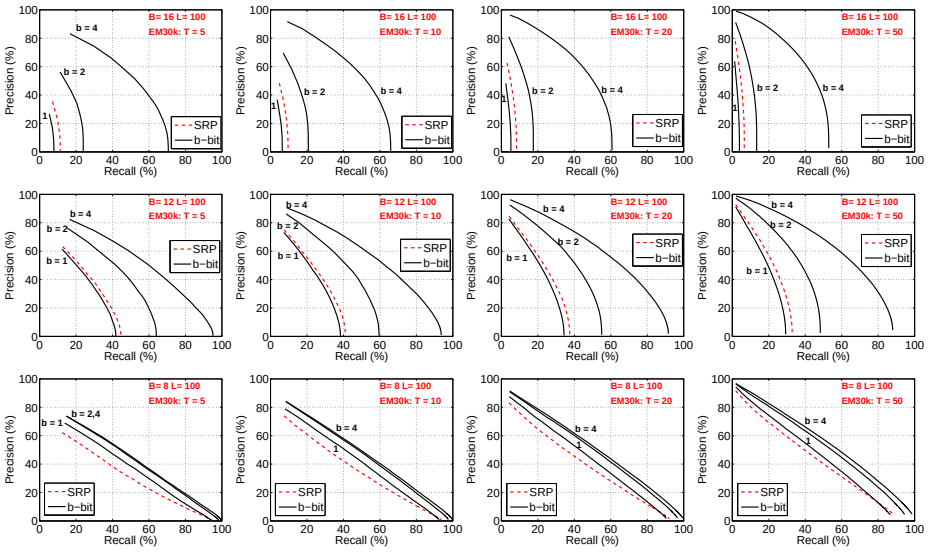

Fig. 14. Precision-Recall curves for SRP and b-bit minwise hashing on the *EM30k* dataset, using $L = 100$ tables.

References

1. Tong, S.: Lessons learned developing a practical large scale machine learning system. http://googleresearch.blogspot.com/2010/04/lessons-learned-developing-practical.html (2008)
2. Li, P., König, A.C.: b-bit minwise hashing. In: WWW, Raleigh, NC (2010) 671–680
3. Li, P., Shrivastava, A., Moore, J., König, A.C.: Hashing algorithms for large-scale learning. In: NIPS, Vancouver, BC (2011)
4. Broder, A.Z.: On the resemblance and containment of documents. In: the Compression and Complexity of Sequences, Positano, Italy (1997) 21–29
5. Broder, A.Z., Glassman, S.C., Manasse, M.S., Zweig, G.: Syntactic clustering of the web. In: WWW, Santa Clara, CA (1997) 1157 – 1166
6. Fetterly, D., Manasse, M., Najork, M., Wiener, J.L.: A large-scale study of the evolution of web pages. In: WWW, Budapest, Hungary (2003) 669–678
7. Manku, G.S., Jain, A., Sarma, A.D.: Detecting Near-Duplicates for Web-Crawling. In: WWW, Banff, Alberta, Canada (2007)
8. Weiss, Y., Torralba, A., Fergus, R.: Spectral hashing. In: NIPS. (2008)
9. Goemans, M.X., Williamson, D.P.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of ACM **42**(6) (1995) 1115–1145
10. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: STOC, Montreal, Quebec, Canada (2002) 380–388
11. Li, P., Hastie, T.J., Church, K.W.: Improving random projections using marginal information. In: COLT, Pittsburgh, PA (2006) 635–649

12. Friedman, J.H., Baskett, F., Shustek, L.: An algorithm for finding nearest neighbors. IEEE Transactions on Computers **24** (1975) 1000–1006
13. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: STOC, Dallas, TX (1998) 604–613
14. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: Commun. ACM. Volume 51. (2008) 117–122
15. Rajaraman, A., Ullman, J.: Mining of Massive Datasets. (http://i.stanford.edu/ ullman/mmds.html)
16. Salakhutdinov, R., Hinton, G.E.: Semantic hashing. Int. J. Approx. Reasoning **50**(7) (2009) 969–978
17. Li, Z., Ning, H., Cao, L., Zhang, T., Gong, Y., Huang, T.S.: Learning to search efficiently in high dimensions. In: NIPS. (2011)
18. Li, P.: Image classification with hashing on locally and gloablly expanded features. Technical report

A Variance-Space Comparisons (*b*-Bit Hashing v.s. SRP)

From the collision probability (5) of sign random projections (SRP), we can estimate the angle $\theta = \cos^{-1}\left(\frac{a}{\sqrt{f_1 f_2}}\right)$, with variance

$$\mathrm{Var}\left(\hat{\theta}\right) = \frac{\pi^2}{k}\left(1 - \frac{\theta}{\pi}\right)\left(\frac{\theta}{\pi}\right) = \frac{\theta(\pi-\theta)}{k}.$$

We can then estimate the intersection $a = |S_1 \cap S_2|$ by

$$\hat{a}_S = \cos\hat{\theta}\sqrt{f_1 f_2}, \qquad Var\left(\hat{a}_S\right) = \frac{\theta(\pi-\theta)}{k} f_1 f_2 \sin^2(\theta)$$

and the resemblance by $\hat{R}_S = \frac{\hat{a}_S}{f_1+f_2-\hat{a}_S}$

$$Var\left(\hat{R}_S\right) = \frac{\theta(\pi-\theta)}{k} f_1 f_2 \sin^2(\theta)\left(\frac{f_1+f_2}{(f_1+f_2-a)^2}\right)^2 + O\left(\frac{1}{k^2}\right).$$

We already know the variance of the b-bit minwise hashing estimator (4), denoted by $Var\left(\hat{R}_b\right)$. To compare it with $Var\left(\hat{R}_S\right)$, we define

$$W_b = \frac{Var\left(\hat{R}_S\right)}{Var\left(\hat{R}_b\right) \times b} = \frac{\theta(\pi-\theta) f_1 f_2 \sin^2(\theta)\left(\frac{f_1+f_2}{(f_1+f_2-a)^2}\right)^2}{\frac{[C_{1,b}+(1-C_{2,b})R][1-C_{1,b}-(1-C_{2,b})R]}{[1-C_{2,b}]^2}} \tag{12}$$

where $C_{1,b}$, $C_{2,b}$ (functions of r_1, r_2, b) are defined in (2). $W_b > 1$ means b-bit minwise hashing is more accurate than SRP at the same storage; see Figure 15.

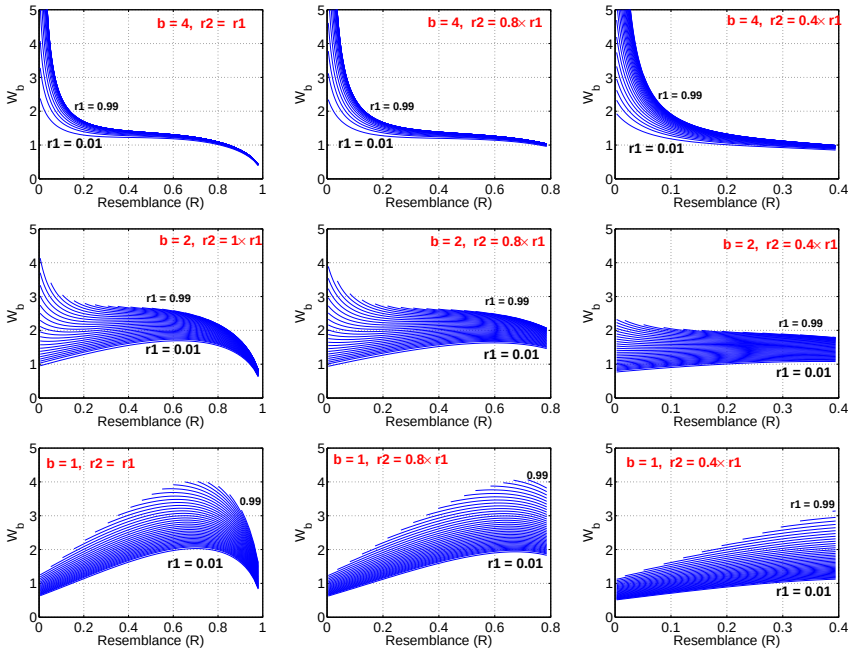

Fig. 15. W_b, $b = 4, 2, 1$, as defined in (12). r_1 and r_2 are defined in (2). Because $W_b > 1$ in most cases (sometimes significantly so), we know that b-bit minwise hashing is more accurate than 1-bit random projections at the same storage cost.

B The Derivation of (10)

$$\int_0^1 P_{b,k,L}(tR)dt = \int_0^1 1 - \left(1 - P_b^k(tR)\right)^L dt = 1 - \int_0^1 \left(1 - P_b^k(tR)\right)^L dt$$

$$= 1 - \int_0^1 \sum_{i=0}^{L} \binom{L}{i} (-1)^i P_b^{ki}(tR)dt = 1 - \sum_{i=0}^{L} \binom{L}{i} (-1)^i \int_0^1 P_b^{ki}(tR)dt$$

$$= 1 - \sum_{i=0}^{L} \binom{L}{i} (-1)^i \frac{1}{2^{bki}} \int_0^1 \left(1 + (2^b - 1)tR\right)^{ki} dt$$

$$= 1 - \sum_{i=0}^{L} \binom{L}{i} (-1)^i \frac{1}{2^{bki}} \sum_{j=0}^{ki} (2^b - 1)^j R^j \binom{ki}{j} \int_0^1 t^j dt$$

$$= 1 - \sum_{i=0}^{L} \binom{L}{i} (-1)^i \frac{1}{2^{bki}} \sum_{j=0}^{ki} \binom{ki}{j} (2^b - 1)^j R^j \frac{1}{j+1}$$

$$= 1 - \sum_{i=0}^{L} \binom{L}{i} (-1)^i \frac{1}{2^{bki}} \frac{1}{(2^b - 1)R} \frac{\left((2^b - 1)R + 1\right)^{ki+1} - 1}{ki + 1}$$