b-Bit Minwise Hashing in Practice

Ping Li
Department of Statistics & Biostatistics
Deptartment of Computer Science
Rutgers University, Piscataway, NJ 08854
pingli@stat.rutgers.edu

Anshumali Shrivastava
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
anshu@cs.cornell.edu

Arnd Christian König
Microsoft Research
Microsoft Corporation
Redmond, WA 98052
chrisko@microsoft.com

ABSTRACT

Minwise hashing is a standard technique in the context of search for approximating set similarities. The recent work [26, 32] demonstrated a potential use of b-bit minwise hashing [23, 24] for efficient search and learning on massive, high-dimensional, binary data (which are typical for many applications in Web search and text mining). In this paper, we focus on a number of critical issues which must be addressed before one can apply b-bit minwise hashing to the volumes of data often used industrial applications.

Minwise hashing requires an expensive preprocessing step that computes k (e.g., 500) minimal values after applying the corresponding permutations for each data vector. We developed a parallelization scheme using GPUs and observed that the preprocessing time can be reduced by a factor of $20 \sim 80$ and becomes substantially smaller than the data loading time. Reducing the preprocessing time is highly beneficial in practice, e.g., for duplicate Web page detection (where minwise hashing is a major step in the crawling pipeline) or for increasing the testing speed of online classifiers.

Another critical issue is that for very large data sets it becomes impossible to store a (fully) random permutation matrix, due to its space requirements. Our paper is the first study to demonstrate that b-bit minwise hashing implemented using simple hash functions, e.g., the 2-universal (2U) and 4-universal (4U) hash families, can produce very similar learning results as using fully random permutations. Experiments on datasets of up to 200GB are presented.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

Minwise hashing [3–5] is a standard technique for efficiently computing set similarities in the context of search, with further applications in the context of content matching for online advertising [28], detection of redundancy in enterprise file systems [13], syntactic similarity algorithms for enterprise information management [8], Web spam [35], etc. The recent development of b-bit minwise hashing [23, 24] provided a substantial improvement in the estimation accuracy and speed by proposing a new estimator that stores only the lowest b bits of each hashed value.

More recently, [25, 26] proposed the use of b-bit minwise hashing in the context of learning algorithms such as SVM or logistic regression on large binary data (which is typical in Web classification tasks). b-bit minwise hashing can enable scalable learning where otherwise massive (and expensive) parallel architectures would have been required, at negligible reduction in learning quality. Furthermore, [32] proposed to directly use the bits from b-bit minwise hashing to build hash tables to facilitate sublinear time near neighbor search in binary data. [32] demonstrated its superior performance compared to other competing algorithms.

In this study, we address two critical issues in order to apply b-bit minwise hashing to truly large-scale industrial applications. The **first issue** is that the current practice of minwise hashing requires a very expensive preprocessing step. The **second issue** has to do with random permutation generation, which must be approximated when the data dimensionality is extremely high. We will also explain why the other related and active line of research work on **one-permutation hashing** [21, 22, 25] has not provided an adequate solution to these two issues.

To understand these issues, we begin with a review of the method.

1.1 A Review of b-Bit Minwise Hashing

Minwise hashing mainly focuses on binary (0/1) data, which can be viewed as sets. Consider sets $S_1, S_2 \subseteq \Omega = \{0, 1, 2, ..., D-1\}$, minwise hashing applies a random permutation $\pi : \Omega \to \Omega$ on S_1 and S_2 and uses the following collision probability

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

to estimate R, which is the resemblance between S_1 and S_2. With k permutations: π_1, ..., π_k, one can estimate R without bias:

$$\hat{R}_M = \frac{1}{k}\sum_{j=1}^{k} 1\{z_{1,j} = z_{2,j}\} \qquad (2)$$

$$z_{1,j} = \min(\pi_j(S_1)), \;\; z_{2,j} = \min(\pi_j(S_2)).$$

A common practice is to store each hashed value, e.g., $\min(\pi(S_1))$, using 64 bits [12]. The storage (and computational) cost is prohibitive in industrial applications [27]. The recent work of b-bit minwise hashing [23,24] provides a simple solution by storing only the lowest b bits of each hashed value.

Internetware'13, October 23 - 24, 2013, Changsha, China.

For convenience, we define

$z_1^{(b)}$ = the lowest b bits of z_1, $\qquad$ $z_2^{(b)}$ = the lowest b-bits of z_2.

THEOREM 1. *[23] Assume D is large.*

$$P_b = \mathbf{Pr}\left(z_1^{(b)} = z_2^{(b)}\right) = C_{1,b} + (1 - C_{2,b})\, R, \qquad (3)$$

where $C_{1,b}$ and $C_{2,b}$ are functions of $(D, |S_1|, |S_2|, |S_1 \cap S_2|)$. □

Based on Theorem 1, we can estimate P_b (and R) from k independent permutations $\pi_1, \pi_2, ..., \pi_k$:

$$\hat{R}_b = \frac{\hat{P}_b - C_{1,b}}{1 - C_{2,b}}, \qquad \hat{P}_b = \frac{1}{k}\sum_{j=1}^{k} 1\left\{z_{1,\pi_j}^{(b)} = z_{2,\pi_j}^{(b)}\right\}, \quad (4)$$

The estimator $\hat{P}_b$ is an inner product between two vectors in $2^b \times k$ dimensions with exactly k 1's, because

$$1\left\{z_1^{(b)} = z_2^{(b)}\right\} = \sum_{t=0}^{2^b-1} 1\{z_1^{(b)} = t\} \times 1\{z_2^{(b)} = t\} \qquad (5)$$

This provides a practical strategy for using b-bit minwise hashing for large-scale learning. That is, each original data vector is transformed into a new data point consisting of k b-bit integers, which is expanded into a $2^b \times k$-length binary vector at the run-time.

These days, many machine learning applications, especially in the context of search, are faced with large and inherently high-dimensional datasets. For example, [34] discusses training datasets with (on average) $n = 10^{11}$ items and $D = 10^9$ distinct features. [36] experimented with a dataset of potentially $D = 16$ trillion (1.6×10^{13}) unique features. Effective algorithms for data/feature reduction are highly beneficial for these industry applications.

1.2 Linear Learning Algorithms

Clearly, b-bit minwise hashing can approximate both linear and nonlinear kernels (if they are functions of the inner products). We focus on linear learning because many high-dimensional datasets used in the context of search are naturally suitable for linear algorithms. Realistically, for industrial applications, "*almost all the big impact algorithms operate in pseudo-linear or better time*" [20].

Linear algorithms such as linear SVM and logistic regression have become very powerful and extremely popular. Representative software packages include SVM$^{\text{perf}}$ [18], Pegasos [30], Bottou's SGD SVM [2], and LIBLINEAR [11].

Given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, $\mathbf{x}_i \in \mathbb{R}^D$, $y_i \in \{-1, 1\}$, the L_2-regularized linear SVM solves the following optimization problem:

$$\min_{\mathbf{w}} \ \frac{1}{2}\mathbf{w^T w} + C\sum_{i=1}^{n} \max\left\{1 - y_i \mathbf{w^T x_i},\ 0\right\}, \qquad (6)$$

and the L_2-regularized logistic regression solves a similar problem:

$$\min_{\mathbf{w}} \ \frac{1}{2}\mathbf{w^T w} + C\sum_{i=1}^{n} \log\left(1 + e^{-y_i \mathbf{w^T x_i}}\right). \qquad (7)$$

Here $C > 0$ is an important penalty parameter.

1.3 Issue 1: Expensive Preprocessing

Minwise hashing requires an expensive preprocessing step in order to compute k (e.g., $k = 500$) minimal values (after permutation) for each data vector. Note that in prior studies on duplicate detection [3], k was usually not too large (i.e., 200), mainly because duplicate detection concerns highly similar pairs. With b-bit minwise hashing, we have to use larger values of k according to the analysis in [23] even in the context of duplicate detection (but still obtain significant storage reduction). Also note that classification tasks are quite different from duplicate detection. For example, we use $k \leq 500$ in our experiments on the *rcv1* dataset; but it looks there might be further (small) improvements by using $k > 500$.

In the context of Web page duplicate detection, the cost of the preprocessing overhead is significant simply because of the huge data volume. While parallelizing this task is conceptually simple (as the signatures of different pages can be computed independently) it still comes at the cost of using additional hardware and electricity. Any significant improvements in the speed of computing minhashes thus likely will be directly reflected in the cost of the required infrastructure.

For machine learning research and applications, this expensive preprocessing step can be a significant issue in scenarios where (either due to changing data distributions or features) models are frequently re-trained. Additionally, in user-facing applications, the testing time performance can be severely affected by the preprocessing step if the (new) incoming data have not been previously processed. In the context of approximate near neighbor search by building many hashing tables [17,32], often substantially more permutations are required, compared to linear learning.

This paper studies how to speed up the execution of the signature computation through the use of graphical processing units (GPUs). GPUs offer, compared to current CPUs, higher instruction parallelism and very low latency access to the internal GPU memory, but comparatively slow latencies when accessing the main memory [19]. As a result, many data processing algorithms (especially such with random memory access patterns) do not benefit significantly when implemented using a GPU. However, the characteristics of the minwise hashing algorithm make it very well suited for execution using a GPU. The algorithm accesses each set S_i in its entirety, which allows for the use of main memory pre-fetching to reduce access latencies. Moreover, since we compute k different hash minima for each item in a set, the algorithm can make good use of the high degree of parallelism in GPUs. This is especially true for b-bit minwise hashing, which, compared to the original algorithm, typically increases the number of hash functions (and minima) to be computed by a factor of 3 or more. Also, note that any improvements to the speed of (b-bit) minwise hashing are directly applicable to large-scale instances of the other applications of minwise hashing mentioned previously.

1.4 Issue 2: Massive Permutation Matrix

When the data dimension (D) is not too large, e.g., millions, the implementation of b-bit minwise hashing for learning is straightforward. Basically, we can assume a "fully random permutation matrix" of size $D \times k$, which defines k permutation mappings. This is actually how researchers use (e.g., Matlab) simulations to verify the theoretical results assuming perfectly random permutations.

Unfortunately, when the dimension is on the order of billions (let alone 2^{64}), it becomes impractical (or too expensive) to store such a permutation matrix. Thus, we have to resort to simple hash functions such as various forms of 2-universal (2U) hashing (e.g., [9]). Now the question is how reliable those hash functions are in the context of learning with b-bit minwise hashing.

There were prior studies on the impact of limited randomness on the estimation accuracy of (64-bit) minwise hashing, e.g., [16,

29]. However, no prior studies reported how the learning accuracies were affected by the use of simple hash functions for b-bit minwise hashing. This study provides the empirical support that, as long as the data are reasonably sparse (as virtually always the case in the context of search), using 2U/4U hash functions results in negligible reduction of learning accuracies (unless $b = 1$ and k is very small).

One limitation of GPUs is that they have fairly limited memory [1]. Thus, it becomes even more beneficial if we can reliably replace a massive permutation matrix with simple hash functions.

1.5 Discussion on One-Permutation Hashing

The very original paper on minwse hashing [3] actually only used one permutation and the consecutive smallest k entries of the permuted set. By taking entries consecutively, the samples are no longer "aligned" in that one can use a simple unified inner product of the samples of the sets to rigorously estimate the original similarity. This serious drawback seriously limited the use of the original one-permutation version of minwise hashing either for neighbor search or learning. This was also the reason why the authors quickly moved to the k-permutation version [4].

Using also only one permutation and the k consecutive elements of permuted set, [21] substantially improved the original estimation scheme of [3]. Later, [22] developed **Conditional Random Sampling (CRS)** by extending the idea of [21] to real-valued (not just binary) data. Obviously, CRS still suffered the similar drawback of the original one-permutation scheme [3].

Recently, a significant progress on **one-permutation hashing** has been accomplished by [25], which divided the space into equal-sized k bins after one permutation and then took the minimum in each bin. This way, the samples are naturally "aligned" with one caveat that there might be many empty bins. The issue of "empty bins" may not be too serious for linear learning because [25] could heuristically treat empty bins as zeros. For near neighbor search which uses hash values for indexing, this issue is very urgent because empty bins do not carry indexing information.

In summary, while one-permutation hashing is promising, at the moment it could not provide an adequate solution to the issue of expensive preprocessing, due to the presence of empty bins. Thus, our work on using GPUs for minwise hashing is still valuable. Of course, even with one permutation, we often still have to resort to simple hashing functions instead of perfect random permutations; and the work in this paper will provide guidelines for practice.

2. SIMPLE HASH FUNCTIONS

As previously discussed, in large-scale industry practice, it is often infeasible to assume perfect random permutations. For example, when $D = 2^{30}$ (about 1 billion) and $k = 500$, a matrix of $D \times k$ integers (4-byte each) would require > 2000GB of storage.

To overcome the difficulty in achieving perfect permutations, the common practice is to use the so-called *universal hashing* [6]. One standard 2-universal (2U) hash function is, for $j = 1$ to k,

$$h_j^{(2U)}(t) = \{a_{1,j} + a_{2,j}\ t \mod p\} \mod D, \tag{8}$$

where $p > D$ is a prime number and $a_{1,j}$, $a_{2,j}$ are chosen uniformly from $\{0, 1, ..., p-1\}$. To increase randomness, one can also use the following 4-universal (4U) hash function:

$$h_j^{(4U)}(t) = \left\{\sum_{i=1}^{4} a_{i,j} t^{i-1} \mod p\right\} \mod D, \tag{9}$$

where the $a_{i,j}$ ($i = 1, 2, 3, 4$) are chosen uniformly from $\{0, 1, ..., p-1\}$. The storage cost for retaining the $a_{i,j}$'s is minimal, compared to storing a permutation matrix. In theory, the 4U hash function is (in the worst-case) more random than the 2U hash function.

Now, to compute the minwise hashes for a given feature vector (e.g., a parsed document represented as a list of 1-grams, 2-grams, and 3-grams, where each n-gram can be viewed as a binary feature), we iterate over all non-zero features; any non-zero location t in the original feature vector is mapped to its new location $h_j(t)$; we then iterate over all mapped locations to find their minimum, which will be the jth hashed value for that feature vector.

3. GPU FOR FAST PREPROCESSING

In this section we will describe and analyze the use of graphics processors (GPUs) for fast computation of minwise hashes. We will first sketch the relevant properties of GPUs in general and then describe in how far minwise hashing computation is suited for execution on this architecture. Subsequently, we will describe our implementation and analyze the resulting performance improvements over a CPU-based implementation.

3.1 Introduction

The use of GPUs as general-purpose coprocessors is relatively recent and primarily due to their high computational power at comparatively low cost. In comparison with commodity CPUs, GPUs offer significantly increased computation speed and memory bandwidth. However, since GPUs have been designed for graphics processing, the programming model (which includes massively parallel Single-Instruction-Multiple-Data (SIMD) processing and limited bus speeds for data transfers to/from main memory) is not suitable for arbitrary data processing applications [15]. GPUs consist of a number of SIMD multiprocessors. At each clock cycle, all processors in a multiprocessor execute identical instructions, but on different parts of the data. Thus, GPUs can leverage spatial locality in data access and group accesses to consecutive memory addresses into a single access; this is referred to as *coalesced access*.

3.2 Our Approach

In light of the properties of GPU processing, our GPU algorithm to compute b-bit minwise hashes proceeds in **3 distinct phases**: First, we read in chunks of 10K sets from disk into main memory and write these to the GPU memory. Then, we compute the hash values and the corresponding minima by applying all k hash functions to the data currently in the GPU and retaining, for each hash function and set, the corresponding minima. Finally, we write out the resulting minima back to main memory and repeat the process.

This batch-style computation has a number of advantages. Because we transfer larger blocks of data, the main memory latency is reduced through the use of main memory pre-fetching. Moreover, because the computation within the GPU itself scans through consecutive blocks of data in the GPU-internal memory (as opposed to random memory access patterns), performing the same computation (with a different hash function) for each set entry k times, we can take full advantage of coalesced access and the massive parallelism inherent in the GPU architecture.

Because GPUs are known to have fairly limited memory capacity, it becomes even more impractical to store a fully random permutation matrix; and hence it is crucial to utilize simple hash functions. We implemented both 2U and 4U hash functions introduced in Section 2. However, because the modulo operations in the definitions of the 2U/4U hash functions are expensive especially for GPUs [1], we have used the following tricks to avoid them and make our approach (more) suitable for GPU-based execution.

3.3 Avoid Modulo Operations in 2U Hashing

To avoid the modulo operations in 2U hashing, we adopt a common trick [10]. Here, for simplicity, we assume $D = 2^s < 2^{32}$ (note that $D = 2^{30}$ corresponds to about a billion features). It is known that the following hash function is essentially 2U [10]:

$$h_j^{(s)}(t) = \left\{a_{1,j} + a_{2,j}\, t \mod 2^{32}\right\} \mod 2^s, \qquad (10)$$

where $a_{1,j}$ is chosen uniformly from $\{0, 1, ..., 2^{32} - 1\}$ and $a_{2,j}$ uniformly from $\{1, 3, ..., 2^{32} - 1\}$ (i.e., $a_{2,j}$ is odd). This scheme is much faster because we can effectively leverage the integer overflow mechanism and the efficient bit-shift operation. In this paper, we always implement 2U hash using $h_j^{(s)}$.

3.4 Avoid Modulo Operations in 4U Hashing

It is slightly tricky to avoid the modulo operations in evaluating 4U hash functions. Assuming $D < p = 2^{31} - 1$ (a prime number), we provide the $C\#$ code to compute $v \; mod \; p$ with $p = 2^{31} - 1$:

```
private static ulong BitMod(ulong v)
{
    ulong p = 2147483647; // p = 2^31-1
    v = (v >> 31) + (v & p);
    if (v >= 2 * p)
       v = (v >> 31) + (v & p);
    if (v >= p)
       return v - p;
    else
       return v;
}
```

To better understand the code, consider

$$v \; mod \; p = x, \qquad \text{and} \qquad v \; mod \; 2^{31} = y$$
$$\Longrightarrow v = p \times Z + x = 2^{31} \times S + y$$
$$\Longrightarrow x = 2^{31}(S - Z) + Z + y$$

for two integers S and Z. S and y can be efficiently evaluated using bit operations: $S = v >> 31$ and $y = v \;\&\; p$.

A recent paper [33] implemented a similar trick for $p = 2^{61} - 1$, which was simpler than ours because with $p = 2^{61} - 1$ there is no need to check the condition "if (v >= 2 * p)". We find the case of $p = 2^{31} - 1$ useful in machine learning practice because it suffices for datasets with less than a billion features. Note that a large value of p potentially increases the dimensionality of the hashed data.

3.5 Experiments: Datasets

Table 1 summarizes the two datasets used in this evaluation: *webspam* and *rcv1*. The *webspam* dataset was used in the recent paper [26]. Since the *webspam* dataset (24 GB in LibSVM format) may be too small compared to datasets used in industrial practice, in this paper we also present an empirical study on the *expanded rcv1* dataset [2], which we generated by using the original features + all pairwise combinations (products) of features + 1/30 of 3-way combinations (products) of features. Note that, for *rcv1*, we did not include the original test set in [2], which has only 20242 examples. To ensure reliable test results, we randomly split our expanded *rcv1* dataset into two halves, for training and testing.

Table 1: Data information

Dataset	n	D	# Avg Nonzeros	Train / Test
Webspam (24 GB)	350000	16609143	3728	80% / 20%
Rcv1 (200 GB)	781265	1010017424	12062	50% / 50%

3.6 Experiments: Platform

The GPU platform we use in our experiments is the NVIDIA Tesla C2050, which has 15 Simultaneous Multiprocessors (SMs), each with 2 groups of 16 scalar processors (hence 2 sets of 16-element wide SIMD units). The peak (single precision) GFlops of this GPU are 1030, with a peak memory bandwidth of 144 GB/s. In comparison, the numbers for a Intel Xeon processor X5670 (Westmere) processor are 278 GFlops and 60 GB/s.

3.7 Experiments: CPU Results

We use the setting of $k = 500$ for these experiments. Table 2 shows the overhead of the CPU-based implementation, broken down into the time required to load the data into memory and the time for the minwise hashing computation. For 2U, we always use the 2U hash function (10). For 4U (Mod), we use the 4U hash function (9) which requires the modulo operation. For 4U (Bit), we use the implementation in Section 3.4, which converted the modulo operation into bit operations. Note that for *rcv1* dataset, we only report the experimental results for 2U hashing.

Table 2: The data loading and preprocessing (for $k = 500$ permutations) times (in seconds). Note that we measured the data loading times of LIBLINEAR which used a plain text data format. The data loading times could be reduced by a factor of 5 or so when the data were converted into binary. In other words, the (relative) preprocessing costs of minwise hashing would be even much more expensive if we optimized the data loading procedure of LIBLINEAR. This further explains why reducing the cost by using GPUs could be so beneficial.

Dataset	Loading	Permu	2U	4U (Mod)	4U (Bit)
Webspam	9.7×10^2	6.1×10^3	4.1×10^3	4.4×10^4	1.4×10^4
Rcv1	1.0×10^4	–	3.0×10^4	–	–

Table 2 shows that the preprocessing using CPUs (even for 2U) can be very expensive, substantially more than data-loading. 4U hashing with modulo operations can take an order of magnitude more time than 2U hashing. As expected, the cost for 4U hashing can be substantially reduced if modulo operations are avoided.

Note that for *webspam* dataset (with only 16 million features), using permutations is actually faster than the algebra required for 4U hash functions. The main constraint here is the storage space. The permutations are generated once and then stored in main memory. This makes them impractical for use with larger feature sets such as the *rcv1* data (with about a billion features)

3.8 Experiments: GPU results

The total overhead for the GPU-based processing for batch size = 10K is summarized in Table 3, demonstrating the substantial time reduction compared to the CPU-based processing in Table 2. For example, the cost of 2U processing on the *webspam* dataset is reduced from 4100 seconds to 51 seconds, a 80-fold reduction. We also observe improvements of similar magnitude for 4U processing (both modulo and bit versions) on *webspam*. For the *rcv1* dataset, the time reduction of the GPU-based implementation is about 20-fold, compared to the CPU-based implementation.

Figures 1 to 3 provide the breakdowns of the overhead for the GPU-based implementations, using 2U hashing, 4U hashing with modulo operations, and 4U hashing without modulo operations, respectively. As shown in the figures, we separate the overhead into three components: (i) time spent transferring the data from main

Table 3: The data loading and preprocessing (for $k = 500$ permutations) times (in seconds) for using GPUs.

Dataset	Loading	GPU 2U	GPU 4U (Mod)	GPU 4U (Bit)
Webspam	9.7×10^2	51	5.2×10^2	1.2×10^2
Rcv1	1.0×10^4	1.4×10^3	1.5×10^4	3.2×10^3

memory to the GPU ("**CPU** $\rightarrow$ **GPU**"), (ii) the actual computation ("**GPU Kernel**") and (iii) transferring the k minima back to main memory ("**GPU** $\rightarrow$ **CPU**").

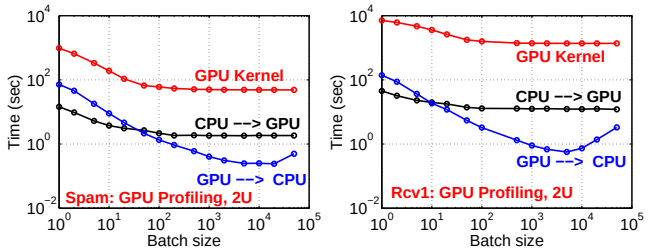

Figure 1: 2U. Overhead of the three phases of the GPU-based implementation using 2U hash functions, for both *webspam* (left panel) and *rcv1* (right panel) datasets.

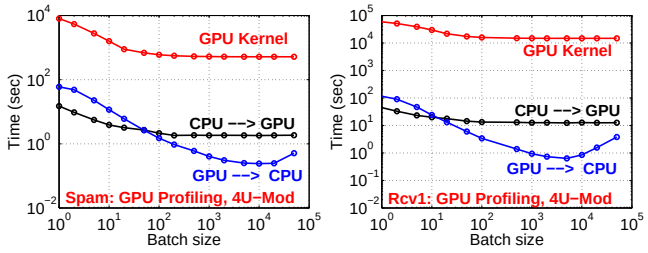

Figure 2: 4U-Mod. Overhead of the three phases of the GPU-based implementation using 4U hash functions with modulo operations, for both datasets.

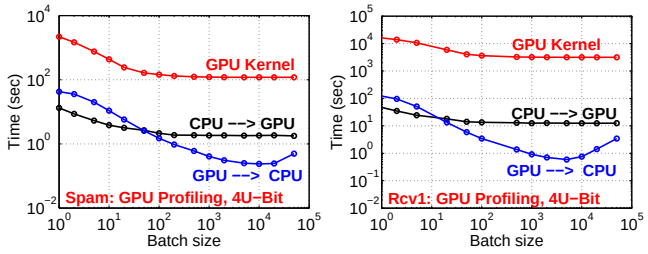

Figure 3: 4U-Bit. Overhead of the three phases of the GPU-based implementation using 4U hash functions without modulo operations (Section 3.4), for both datasets.

3.9 Optimizing GPU Utilization

In order to make best use of the massive parallelism of the GPU we need to ensure that we schedule the computation in such a way that as many multiprocessors as possible are kept busy; in particular, it is not sufficient to compute only k independent hashes in parallel. Instead, we process batches of B data vectors simultaneously, computing k parallel hashes for each entry. One side-effect of this is a noticeable reduction in the host-device memory transfer time as we are now transferring larger, consecutive blocks of memory, thereby reducing the aggregate memory latency.

Moreover, to ensure that all multiprocessors are busy we also need a sufficient number blocks per grid, and we need sufficient threads per multiprocessor to mask the latency. Here, we launch the kernel with B blocks with k threads each. Note that we report results only for $k = 500$ but we observe the same magnitude of improvements with e.g., $k = 200$. We keep the number of threads equal to the number of hashes, which provides very intuitive coding: the blocks IDs correspond to the data vector IDs and thread IDs to hashes. To find the optimum choice of the block number B, we varied B and recorded the resulting performance. As expected, it can be clearly seen from Figures 1 to 3 that when B increases the multiprocessor-utilization increases as well, until no more multiprocessors are idle; we do not observe any performance gain after that.

4. VALIDATION OF THE USE OF 2U/4U HASH FUNCTIONS FOR LEARNING

For large-scale industrial applications, because storing a fully random permutation matrix is not practical, we have to resort to simple hash functions such as 2U or 4U hash families. However, before we can recommend them to practitioners, we must first validate on a smaller dataset that using such hash functions will not hurt the learning performance. To the best of our knowledge, this section is the first empirical study of the impact of hashing functions on machine learning with b-bit minwise hashing.

In addition, Appendix A provides another set of experiments for estimating resemblances using b-bit minwise hashing with simple hash functions. Those experiments demonstrate that, as long as the data are not too dense, using 2U hash will produce very similar estimates as using fully random permutations. That set of experiments may help understand the experimental results in this section. Note that both datasets listed in Table 1 are extremely sparse.

The *webspam* dataset is small enough (24GB and 16 million features) that we can conduct experiments using a permutation matrix. We chose LIBLINEAR as the underlying learning procedure. All experiments were conducted on workstations with Xeon(R) CPU (W5590@3.33GHz) and 48GB RAM, on a Windows 7 System.

4.1 Experimental Results

We experimented with both 2U and 4U hash schemes for training linear SVM and logistic regression. We tried out 30 values for the regularization parameter C in the interval $[10^{-3}, 100]$. We experimented with 11 k values from $k = 10$ to $k = 500$, and for 7 b values: $b = 1, 2, 4, 6, 8, 10, 16$. Each experiment was repeated 50 times. The total number of training experiments turns out to be

$$2 \times 2 \times 30 \times 11 \times 7 \times 50 = 462000.$$

To maximize the repeatability, whenever page space allows, we always would like to present the detailed experimental results for all the parameters instead of, for example, only reporting the best results or the cross-validated results. In this subsection, we only present the results for *webspam* dataset using linear SVM because the results for logistic regression lead to the same conclusion.

Figure 4 presents the SVM test accuracies (averaged over 50 runs). For the test cases that correspond to the most likely parameter settings used in practice (e.g., $k \geq 200$ and $b \geq 4$), we can see

that the results from the three hashing schemes (full permutations, 2U, and 4U) are essentially identical. Overall, it appears that 4U is slightly better than 2U when $b = 1$ or k is very small.

This set of experiments can provide us with a strong experimental evidence that the simple and highly efficient 2U hashing scheme may be sufficient in practice, when used in the context of large-scale machine learning using b-bit minwise hashing.

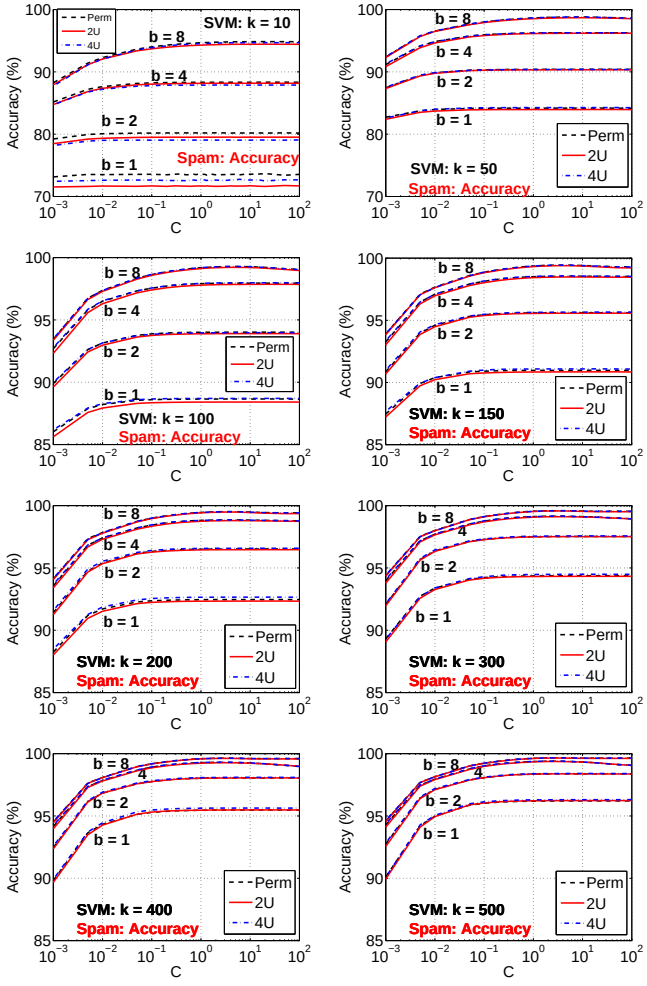

Figure 4: Linear SVM test accuracies on *webspam*, using three hashing schemes: permutations, 2U, and 4U. Both 2U and 4U perform well as the curves essentially overlap except when $b = 1$ or small k. It appears that 4U is only slightly better than 2U.

5. LEARNING ON RCV1 DATA (200GB)

Compared to *webspam*, the size of the expanded *rcv1* dataset may be more close to the training data sizes used in industrial applications. We report the experiments on linear SVM and logistic regression, as well as the comparisons with the VW hash algorithm.

5.1 Experiments on Linear SVM

Figure 5 and Figure 6 respectively provide the test accuracies and train times, for training linear SVM. We can not report the baseline because the original dataset exceeds the memory capacity. Using merely $k = 30$ and $b = 12$, our method can achieve $> 90\%$ test accuracies. With $k \geq 300$, we can achieve $> 95\%$ test accuracies.

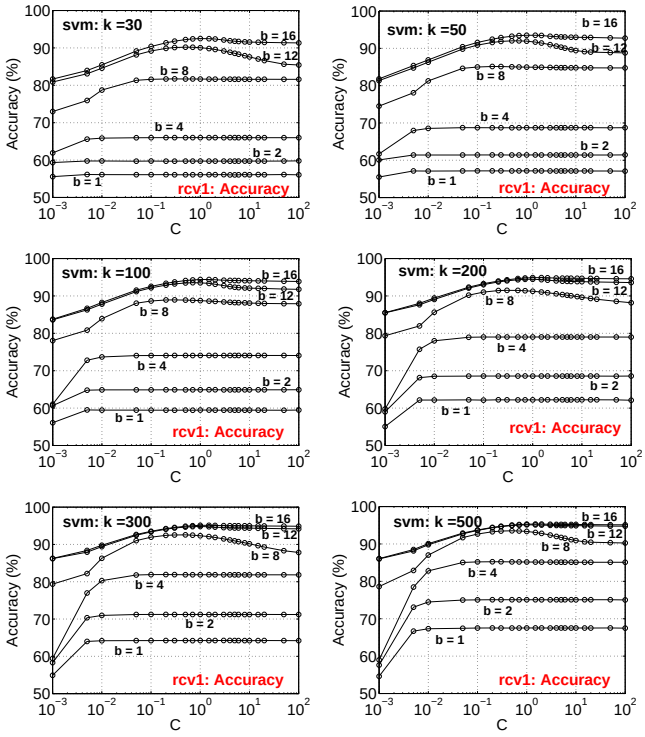

Figure 5: Linear SVM test accuracy on rcv1.

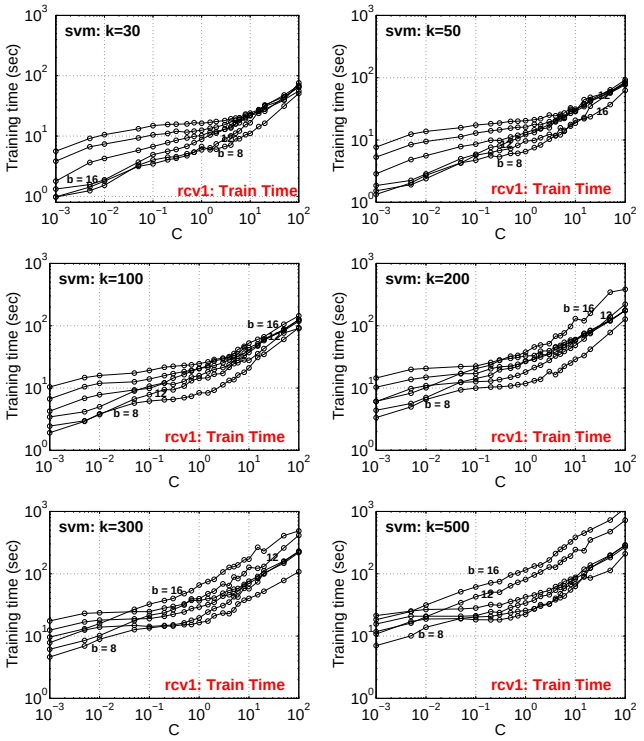

Figure 6: Linear SVM training time on rcv1.

5.2 Experiments on Logistic Regression

Figure 7 and Figure 8 respectively present the test accuracies and training times for training logistic regression. Again, using

merely $k = 30$ and $b = 12$, our method can achieve $> 90\%$ test accuracies. With $k \geq 300$, we can achieve $> 95\%$ test accuracies.

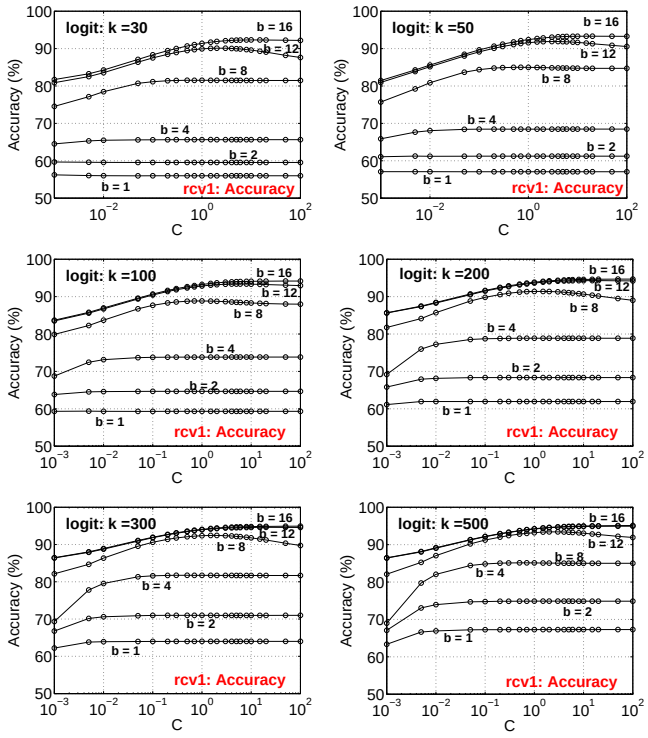

Figure 7: Logistic regression test accuracy on rcv1.

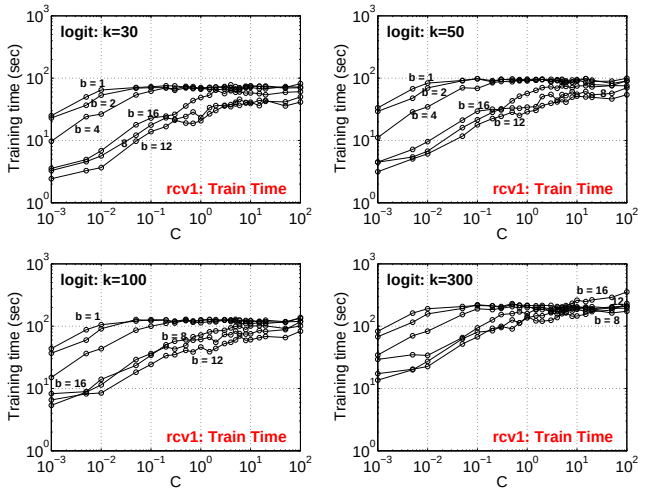

Figure 8: Logistic regression training time on rcv1.

To help understand the significance of these results, next we provide a comparison study with the VW hashing algorithm [31, 36].

5.3 Comparisons with VW Algorithm

The Vowpal Wabbit (VW) algorithm [31, 36] is an influential hashing method for data/dimension reduction. Since [26] only compared b-bit minwise hashing with VW on a small dataset, it is more informative to conduct a comparison of the two algorithms on this much larger dataset (200GB). We experimented with VW using $k = 2^5$ to 2^{14} hash bins (sample size). Note that $2^{14} = 16384$. It is difficult to train LIBLINEAR with $k = 2^{15}$ because the training size of the hashed data by VW is close to 48 GB when $k = 2^{15}$.

Figure 9 and Figure 10 plot the test accuracies for SVM and logistic regression, respectively. In each figure, every panel has the same set of solid curves for VW but a different set of dashed curves for different values of b in b-bit minwise hashing. Since the range of k is very large, here we choose to present the test accuracies against k. Representative C values (0.01, 0.1, 1, 10) are selected for the presentations.

From Figures 9 and 10, we can see clearly that b-bit minwise hashing is substantially more accurate than VW at the same storage size. In other words, in order to achieve the same accuracy, VW will require substantially more storage than b-bit minwise hashing. In fact, from the figures, it looks almost like 1-bit minwise hashing, at the same k, can achieve similar test accuracies as VW.

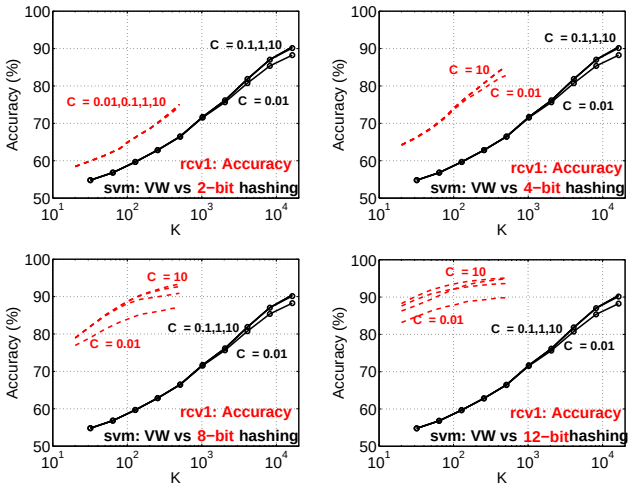

Figure 9: SVM test accuracy on rcv1 for comparing VW (solid) with b-bit minwise hashing (dashed). Each panel plots the same results for VW and results for b-bit minwise hashing for a different b. We select $C = 0.01, 0.1, 1, 10$.

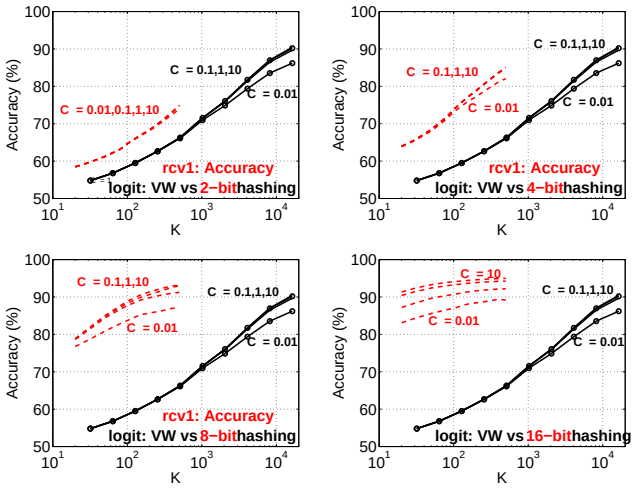

Figure 10: Logistic Regression test accuracy on rcv1 for comparing VW with b-bit minwise hashing.

Figure 11 presents the training times for comparing VW with 8-bit minwise hashing. In this case, we can see that even at the same k, 8-bit hashing may have some computational advantages compared to VW. Of course, as it is clear that VW will require a much larger k in order to achieve the same accuracies as 8-bit minwise hashing, we know that the advantage of b-bit minwise hashing in terms of training time reduction is also enormous.

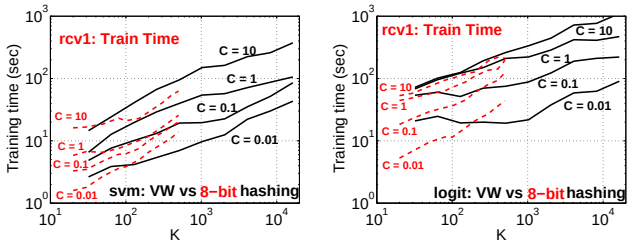

Figure 11: Training time for SVM (left) and logistic regression (right) on rcv1 for comparing VW with 8-bit minwise hashing.

Note that our comparison focuses on the VW hashing algorithm, not the VW online learning platform. The prior work [7] experimented with the VW online learning platform on *webspam* and reported an accuracy of 98.42% (compared to $> 99.5\%$ in our experiments with b-bit hashing) after 597 seconds of training.

6. CONCLUSION

Minwise Hashing is a standard technique for similarity computation which has also recently been shown [25, 26, 32] to be a valuable data reduction technique in machine learning and near neighbor search, where it can reduce both the computational overhead as well as the required infrastructure and energy consumption by orders of magnitude, at often negligible reduction in accuracy.

However, the use of b-bit minwise hashing on truly large learning datasets requires study of two related challenges. Datasets with very large numbers of features make it impossible to use precomputed permutation matrices for the permutation step, due to prohibitive storage requirements. Also, for very large data, the initial pre-processing phase during which minhash signatures are computed, consumes significant resources.

In the context of duplicate detection (which normally concerns only highly similar pairs of documents) using minwise hashing with 64 bits per hashed value, the prior studies (e.g., [3]) demonstrated that it would be sufficient to use about $k \approx 200$ permutations. However, b-bit minwise hashing (for small values of b) does require more permutations than the original minwise hashing, as explained in [23], for example, by increasing k by a factor of 3 when using $b = 1$ and the resemblance threshold is $R = 0.5$. In the context of machine learning and b-bit minwise hashing, we have also found that in some datasets k has to be fairly large, e.g., $k = 500$ or even more. This is because machine learning algorithms use all similarities, not just highly similar pairs.

In this study, we were able to show that, when using 2- or 4-universal hash functions, we see accuracy similar to the one achieved by using fully random permutations. This validates the use of b-bit minwise hashing for data with extremely large numbers of features.

We were able to formulate an implementation of the minwise hashing algorithm that effectively leverages the properties of current GPU architectures, in particular their massive parallelism and SIMD instruction processing, while minimizing the impact of their constraints (most notably slow modulo operations and the limited bandwidth available for main memory data transfer). We observed that the new GPU-based implementation resulted in speed-ups of between 20-80$\times$ for the minhash computation.

Therefore, we can conclude that b-bit minwise hashing can be naturally integrated with search and learning algorithms, to solve extremely large-scale learning problems in industrial applications.

Open problem: While our use of GPUs results in a significant improvement of the processing speed for the standard (k-permutation) minwise hashing, parallelization does not provide an energy-efficient solution. There is a line of work on **one permutation hashing** [21, 22, 25], which at the moment still suffers from the severe issue of "empty bins" especially when it is used for approximate near neighbor search. How to solve this open problem (i.e., the empty bin issue) for one permutation hashing will be an interesting and important research problem with an urgent need from practice.

Acknowledgement

This work is partally supported by NSF-BigData-1250914, ONR-N000141310261, NSF-SES-1131848, and AFOSR-FA9550-13-1-0137.

APPENDIX

A. RESEMBLANCE ESTIMATION USING SIMPLE HASH FUNCTIONS

This section studies the effect of using 2U/4U hashing function in place of (fully) random permutation matrices on the accuracy of resemblance estimation via b-bit minwise hashing. This will provide us a better understanding why the learning results using b-bit minwise hashing are not noticeably affected by replacing the fully random permutation matrix with 2U/4U hash functions. As we shall see, as long as the original data are not too dense, using 2U/4U hash functions will not result in loss of estimation accuracy. As we observed that results from 2U and 4U are essentially indistinguishable, we only report the 2U experiments.

The task we study here is the estimation of word associations. The dataset, extracted from commercial Web crawls, consists of 10 pairs of sets (20 English words). Each set consists of the document IDs which contain the word at least once. See Table 4.

Table 4: Data information of the 10 pairs of English words. For example, "KONG" and "HONG" correspond to the two sets of document IDs which contained word "KONG" and word "HONG" respectively.

Word 1	Word 2	f_1	f_2	R
KONG	HONG	948	940	0.925
RIGHTS	RESERVED	12234	11272	0.877
OF	AND	37339	36289	0.771
GAMBIA	KIRIBATI	206	186	0.712
UNITED	STATES	4079	3981	0.591
SAN	FRANCISCO	3194	1651	0.476
CREDIT	CARD	2999	2697	0.285
TIME	JOB	37339	36289	0.128
LOW	PAY	2936	2828	0.112
A	TEST	39063	2278	0.052

We implemented both 2U hash (10) and 4U hash schemes, for $D = 2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24}, 2^{26}, 2^{28}, 2^{30}, 2^{32}$. Note that $D \geq 2^{16}$ is necessary for this dataset. After sufficient number of repetitions,

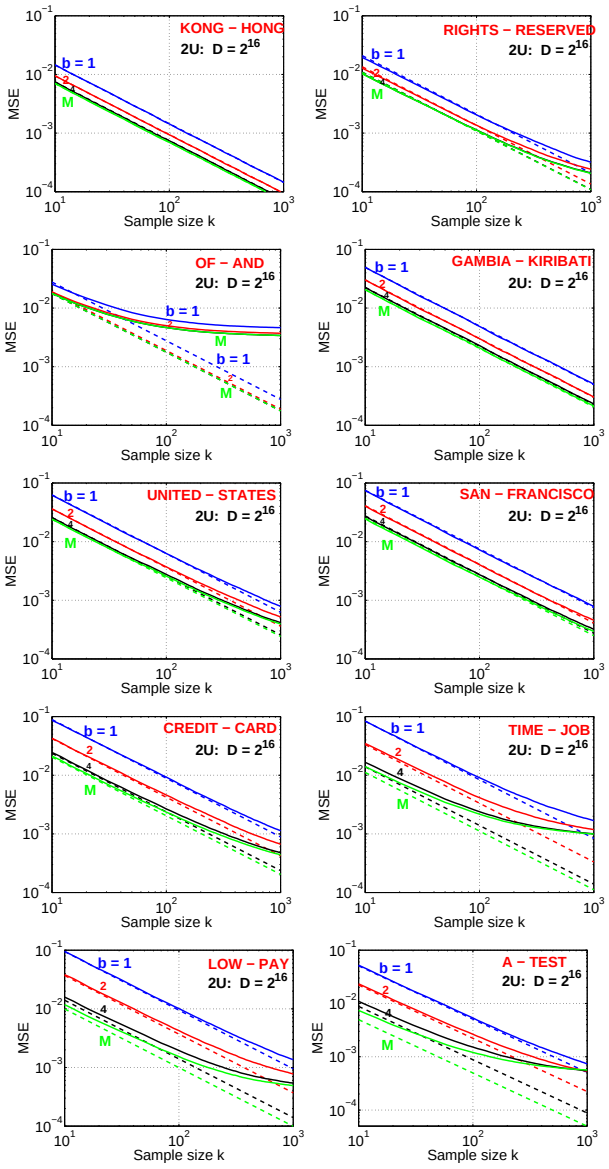

Figure 12: Mean square errors (MSEs) of the resemblance estimates using (4) and 2U hashing with $D = 2^{16}$, on the 10 English word vector pairs in Table 4. We present $b = 1, 2, 4$ and the original minwise hashing (i.e., "M"). The dashed curves are the theoretical variances (Eq. (11) in [23]). Ideally the solid and dashed curves should overlap (e.g., KONG-HONG). Due to limited randomness, when the data are fairy dense (e.g., OF-AND), the empirical estimates deviate from theoretical predictions.

we computed the simulated mean square error (MSE = Var + Bias^2) for each case, to compare with the theoretical variance (Eq. (11) in [23]), which was derived by assuming perfect random permutations. Ideally, the empirical MSEs and the theoretical variances should overlap. Indeed, we observe this is always the case when $D \geq 2^{20}$. This is the reason why we only plot the results for $D \leq 2^{20}$ in Figures 12 to 14. In fact, as shown in Figure 12, when the data are not dense (e.g., KONG-HONG, GABMIA-KIRIBATI, SAN-FRANCISCO), using 2U can achieve very similar results as using perfect random permutations, even at the smallest $D = 2^{16}$.

Practical Implication: In practice, we expect the data vectors to be very sparse for a large number of applications, especially the many search-related tasks where features correspond to the presence/absence of text n-grams. For these tasks, the large number of distinct words (e.g., [14] reports 38M distinct 1-grams in an early Wikipedia corpus) and the much smaller number of terms in individual documents combine to cause this property. Therefore, we expect that 2U/4U hash functions will perform well when used for b-bit minwise hashing, as verified in the main body of the paper.

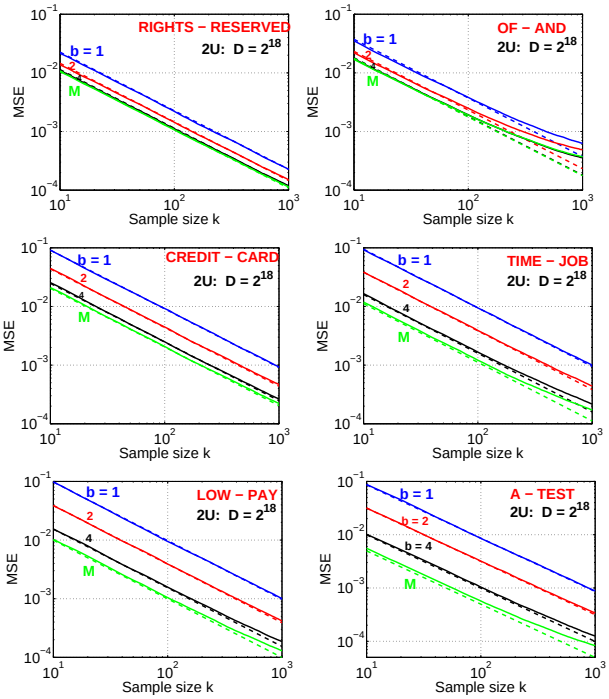

Figure 13: Mean square errors (MSEs) of the resemblance estimates using (4) and 2U hashing with $D = 2^{18}$, on 6 English word vector pairs which do not perform too well with $D = 2^{16}$ in Figure 12. We can see that the results become much better.

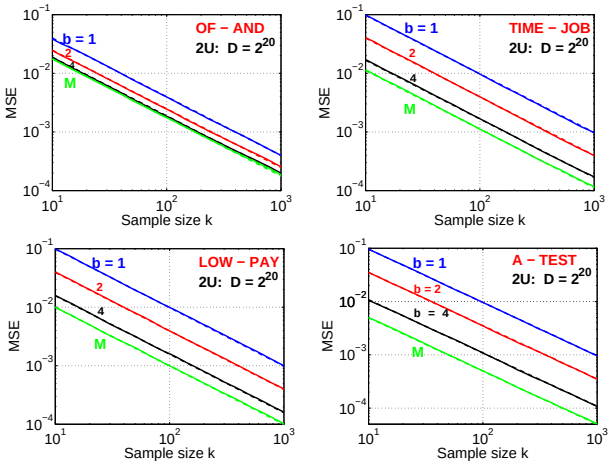

Figure 14: Mean square errors (MSEs) of the resemblance estimates using (4) and 2U hashing with $D = 2^{20}$, on 3 English word vector pairs which do not perform too well with $D = 2^{18}$ in Figure 13. We can see now all the dashed curves (theoretical) match the solid curves (empirical) now.

B. REFERENCES

[1] *http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf*.

[2] Leon Bottou. http://leon.bottou.org/projects/sgd.

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

[4] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, pages 327–336, Dallas, TX, 1998.

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

[6] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *STOC*, pages 106–112, 1977.

[7] Kai-Wei Chang and Dan Roth. Selective block minimization for faster convergence of limited memory large-scale linear models. In *KDD*, pages 699–707, 2011.

[8] Ludmila Cherkasova, Kave Eshghi, Charles B. Morrey III, Joseph Tucek, and Alistair C. Veitch. Applying syntactic similarity algorithms for enterprise information management. In *KDD*, pages 1087–1096, Paris, France, 2009.

[9] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, pages 569–580, 1996.

[10] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

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

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

[13] George Forman, Kave Eshghi, and Jaap Suermondt. Efficient detection of large-scale redundancy in enterprise file systems. *SIGOPS Oper. Syst. Rev.*, 43(1):84–91, 2009.

[14] Venkatesh Ganti, Arnd Christian König, and Xiao Li. Precomputing search features for fast and accurate query classification. In *WSDM*, pages 61–70, 2010.

[15] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34:21:1–21:39, December 2009.

[16] Piotr Indyk. A Small Approximately Min-wise Independent Family of Hash Functions. *J. Algorithms*, 38, 2001.

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

[18] Thorsten Joachims. Training linear svms in linear time. In *KDD*, pages 217–226, Pittsburgh, PA, 2006.

[19] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[20] John Langford. http://cacm.acm.org/blogs/blog-cacm/108385-research-directions-for-machine-learning-and-algorithms/fulltext.

[21] Ping Li and Kenneth W. Church. Using sketches to estimate associations. In *HLT/EMNLP*, pages 708–715, Vancouver, BC, Canada, 2005 (The full paper appeared in Commputational Linguistics in 2007).

[22] Ping Li, Kenneth W. Church, and Trevor J. Hastie. Conditional random sampling: A sketch-based sampling technique for sparse data. In *NIPS*, pages 873–880, Vancouver, BC, Canada, 2006 (Newer results appeared in NIPS 2008.

[23] Ping Li and Arnd Christian König. b-bit minwise hashing. In *WWW*, pages 671–680, Raleigh, NC, 2010.

[24] Ping Li, Arnd Christian König, and Wenhao Gui. b-bit minwise hashing for estimating three-way similarities. In *NIPS*, Vancouver, BC, 2010.

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

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

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

[28] Sandeep Pandey, Andrei Broder, Flavio Chierichetti, Vanja Josifovski, Ravi Kumar, and Sergei Vassilvitskii. Nearest-neighbor caching for content-match applications. In *WWW*, pages 441–450, Madrid, Spain, 2009.

[29] Mihai Pătraşcu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *ICALP*, pages 715–726, 2010.

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

[31] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and S.V.N. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10:2615–2637, 2009.

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

[33] Mikkel Thorup and Yin Zhang. Tabulation based 5-universal hashing and linear probing. In *ALENEX*, pages 62–76, 2010.

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

[35] Tanguy Urvoy, Emmanuel Chauveau, Pascal Filoche, and Thomas Lavergne. Tracking web spam with html style similarities. *ACM Trans. Web*, 2(1):1–28, 2008.

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