Efficient Data Reduction and Summarization

Ping Li

Department of Statistical Science

Faculty of Computing and Information Science

Cornell University

December 8, 2011

Hashing Algorithms for Large-Scale Learning

Ping Li

Department of Statistical Science

Faculty of Computing and Information Science

Cornell University

References

1. Ping Li, Anshumali Shrivastava, and Christian König, Training Logistic Regression and SVM on **200GB** Data Using b-Bit Minwise Hashing and Comparisons with Vowpal Wabbit (VW), Technical Reports, 2011
2. Ping Li, Anshumali Shrivastava, Joshua Moore, and Christian König, Hashing Algorithms for Large-Scale Learning, NIPS 2011
3. Ping Li and Christian König, Theory and Applications of b-Bit Minwise Hashing, Research Highlight Article in Communications of ACM, August 2011
4. Ping Li, Christian König, and Wenhao Gui, b-Bit Minwise Hashing for Estimating Three-Way Similarities, NIPS 2010
5. Ping Li, Christian König, b-Bit Minwise Hashing, WWW 2010

How Large Are the Datasets?

Conceptually, consider the data as a matrix of size $n \times D$.

In modern applications, # examples $n = 10^6$ is common and $n = 10^9$ is not rare, for example, images, documents, spams, social and search click-through data. In a sense, click-through data can have unlimited # examples.

Very high-dimensional (image, text, biological) data are common, e.g., $D = 10^6$ (million), $D = 10^9$ (billion), $D = 10^{12}$ (trillion), $D = 2^{64}$ or even higher. In a sense, D can be made arbitrarily higher by taking into account pairwise, 3-way or higher interactions.

Massive, High-dimensional, Sparse, and Binary Data

Binary sparse data are very common in the real-world:

- For many applications (such as text), binary sparse data are very natural.
- Many datasets can be quantized/thresholded to be binary without hurting the prediction accuracy.
- In some cases, even when the "original" data are not too sparse, they often become sparse when considering pariwise and higher-order interactions.

An Example of Binary (0/1) Sparse Massive Data

A set $S \subseteq \Omega = \{0, 1, ..., D-1\}$ can be viewed as 0/1 vector in D dimensions.

Shingling: Each document (Web page) can be viewed as a set of w-shingles.

For example, after parsing, a sentence "today is a nice day" becomes

- $w = 1$: {"today", "is", "a", "nice", "day"}
- $w = 2$: {"today is", "is a", "a nice", "nice day"}
- $w = 3$: {"today is a", "is a nice", "a nice day"}

Previous studies used $w \geq 5$, as single-word (unit-gram) model is not sufficient.

Shingling generates extremely high dimensional vectors, e.g., $D = (10^5)^w$.

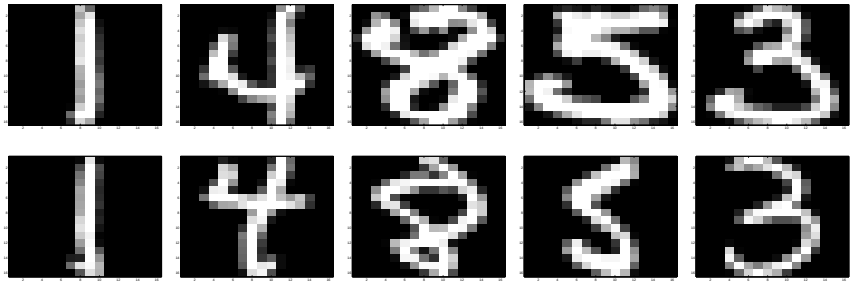
$(10^5)^5 = 10^{25} = 2^{83}$, although in current practice, it seems $D = 2^{64}$ suffices.

Challenges of Learning with Massive High-dimensional Data

- The data often can not fit in memory (even when the data are sparse).
- Data loading takes too long. For example, online algorithms require less memory but often need to iterate over the data for several (or many) passes to achieve sufficient accuracies. Data loading in general dominates the cost.
- Training can be very expensive, even for simple linear models such as logistic regression and linear SVM.
- Testing may be too slow to meet the demand. This is especially important for applications in search engines or interactive data visual analytics.
- The model itself can be too large to store, for example, we can not really store a vector of weights for fitting logistic regression on data of 2^{64} dimensions.

Another Example of Binary Massive Sparse Data

The zipcode recognition task can be formulated as a 10-class classification problem. The well-known MNIST8m dataset contains 8 million 28×28 small images. We try to use linear SVM (perhaps the fastest classification algorithm).

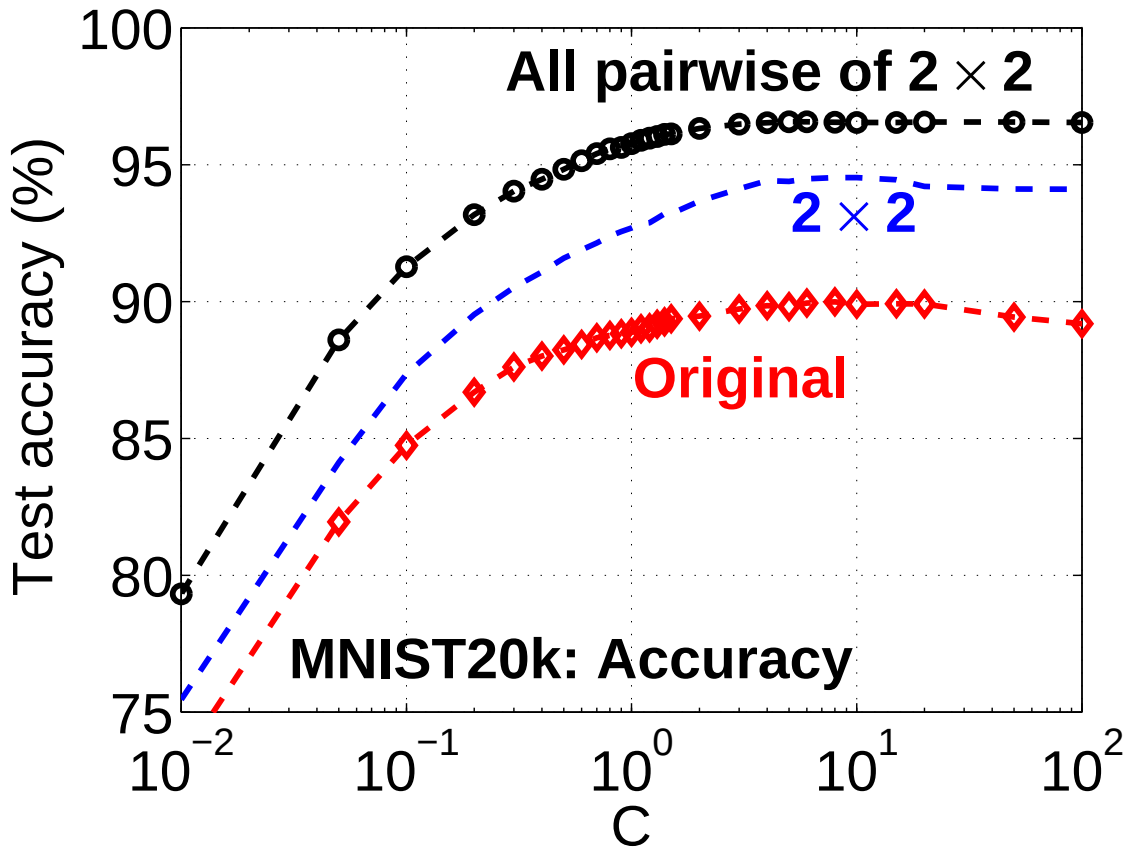


Features for linear SVM

1. Original pixel intensities. $784 (= 28 \times 28)$ dimensions.
2. All overlapping 2×2 (or higher) patches (absent/present). Binary in 12448 dimensions. $(28-1)^2 \times 2^{2\times 2} + 784 = 12448$.
3. All pairwise on top of 2×2 patches. Binary in 77,482,576 dimensions. $12448 \times (12448-1)/2 + 12448 = 77482576$.

Linear SVM experiments on the first 20,000 examples (MNIST20k)

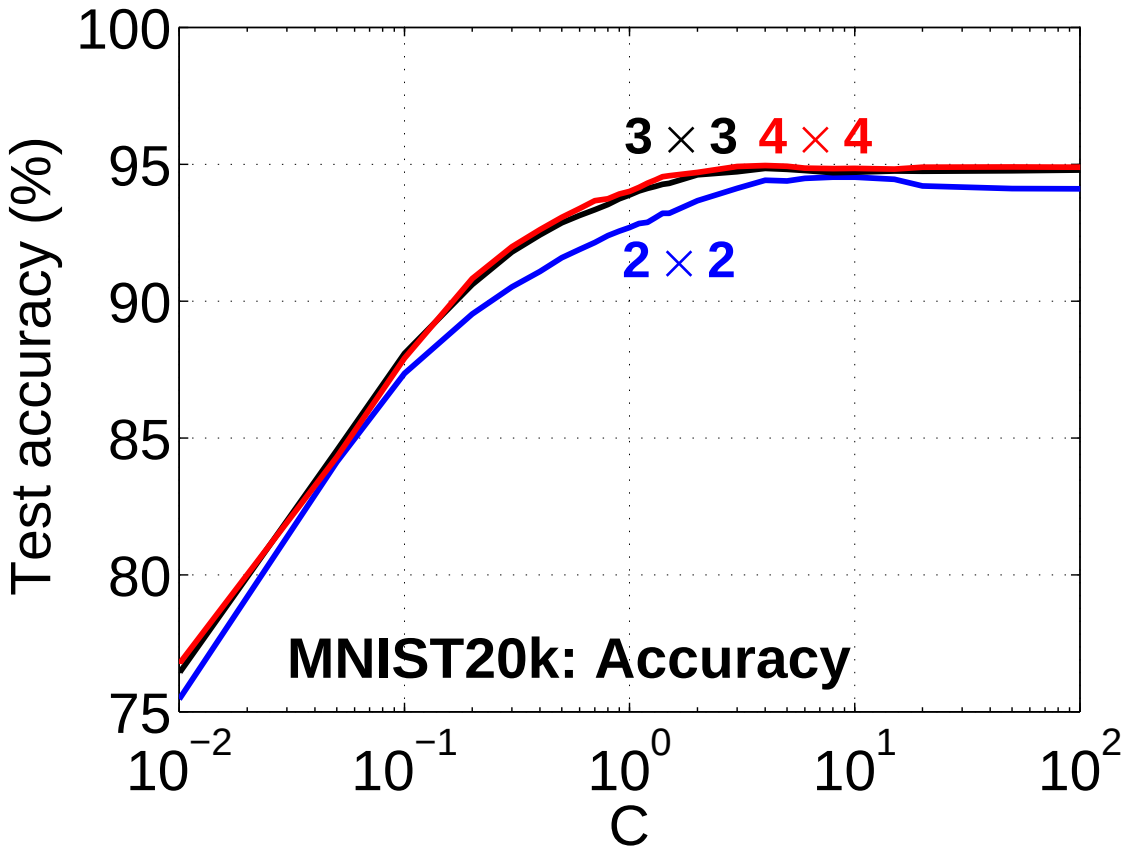
The first 10,000 for training and the remaining 10,000 for testing.



C is the usual regularization parameter in linear SVM (the only parameter).

3×3: all overlapping 3×2 patches + all 2×2 patches

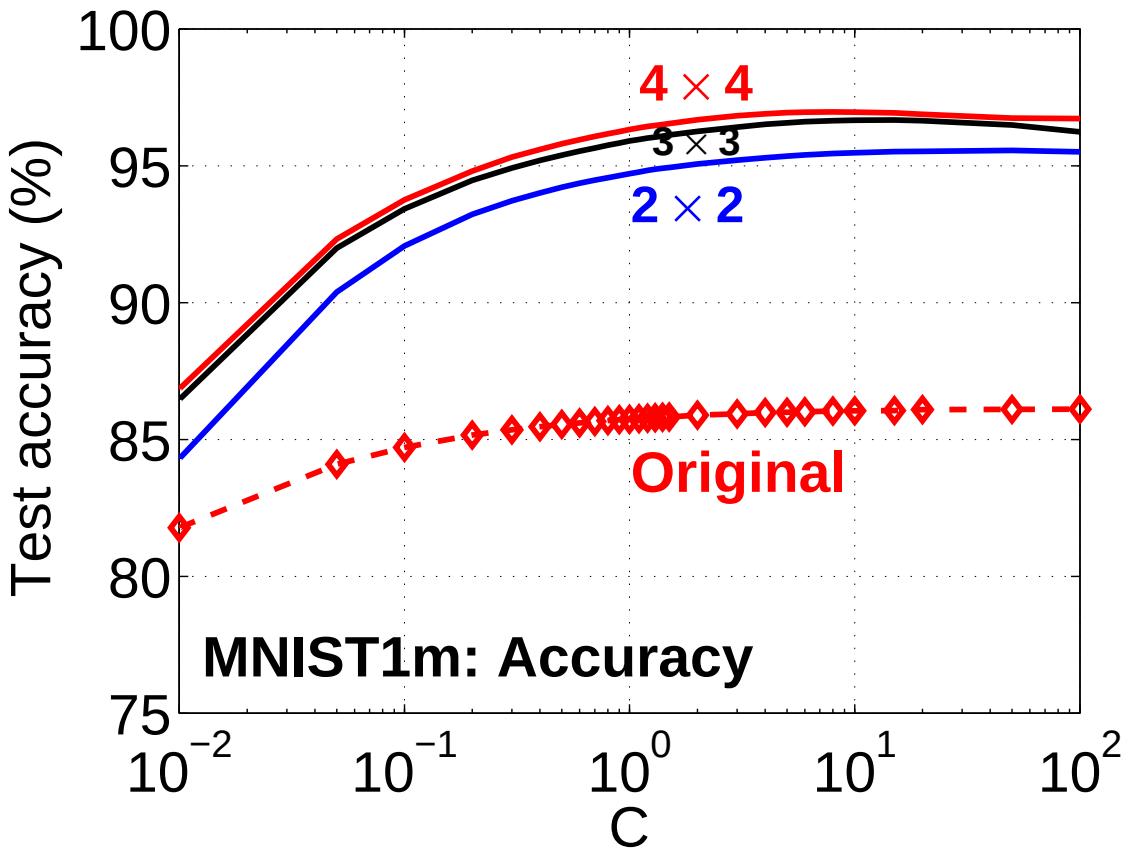
4×4: all overlapping 4×4 patches + ...



Increasing features does not help much, possibly because # examples is small.

Linear SVM experiments on the first 1,000,000 examples (MNIST1m)

The first 500,000 for training and the remaining 500,000 for testing.

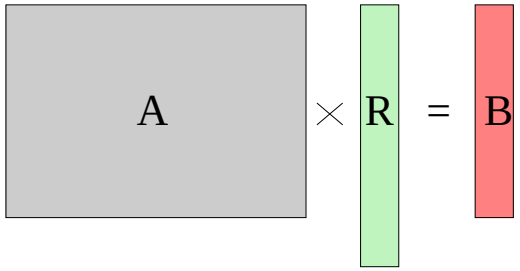


All pairwise on top of 2×2 patches would require > 1TB storage on MNIST1m.

Remind these are merely tiny images to start with.

A Popular Solution Based on Normal Random Projections

Random Projections: Replace original data matrix $\mathbf{A}$ by $\mathbf{B} = \mathbf{A} \times \mathbf{R}$



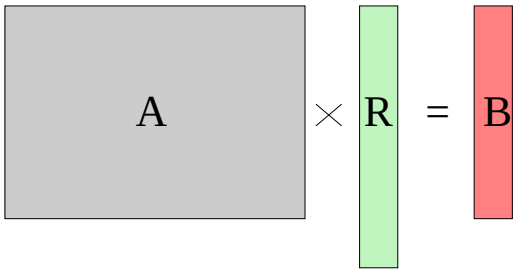
$\mathbf{R} \in \mathbb{R}^{D \times k}$: a random matrix, with i.i.d. entries sampled from $N(0, 1)$.

$\mathbf{B} \in \mathbb{R}^{n \times k}$: projected matrix, also random.

$\mathbf{B}$ approximately preserves the Euclidean distance and inner products between any two rows of $\mathbf{A}$. In particular, $E\left(\mathbf{B}\mathbf{B}^{\mathsf{T}}\right) = \mathbf{A}\mathbf{A}^{\mathsf{T}}$.

Using Random Projections for Linear Learning

The task is straightforward.



Since the new (projected) data $\mathbf{B}$ preserve the inner product of the original data $\mathbf{A}$, we can simply feed $\mathbf{B}$ into (e.g.,) SVM or logistic regression solvers.

These days, linear algorithms are extremely fast (if the data can fit in memory).

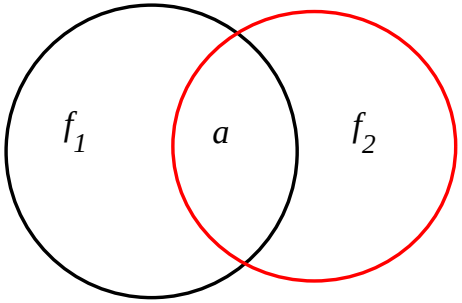
——— However ——

Random projections may not be good for inner products because the variance can be much much larger than b-bit minwise hashing.

Notation for b-Bit Minwise Hashing

A binary (0/1) vector can be equivalently viewed as a set (locations of nonzeros).

Consider two sets $S_1, S_2 \subseteq \Omega = \{0, 1, 2, ..., D-1\}$ (e.g., $D = 2^{64}$)



$$f_1 = |S_1|, \qquad f_2 = |S_2|, \qquad a = |S_1 \cap S_2|.$$

The **resemblance** R is a popular measure of set similarity

$$R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{a}{f_1 + f_2 - a}$$

Minwise Hashing

Suppose a random permutation π is performed on Ω, i.e.,

$$\pi: \Omega \longrightarrow \Omega, \qquad \text{where} \quad \Omega = \{0, 1, ..., D-1\}.$$

An elementary probability argument shows that

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

An Example

$D = 5$. $S_1 = \{0, 3, 4\}$, $S_2 = \{1, 2, 3\}$, $R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{1}{5}$.

One realization of the permutation π can be

$$0 \Longrightarrow 3$$
$$1 \Longrightarrow 2$$
$$2 \Longrightarrow 0$$
$$3 \Longrightarrow 4$$
$$4 \Longrightarrow 1$$

$$\pi(S_1) = \{3, 4, 1\} = \{1, 3, 4\}, \qquad \pi(S_2) = \{2, 0, 4\} = \{0, 2, 4\}$$

In this realization, $\min(\pi(S_1)) \neq \min(\pi(S_2))$.

Minwise Hashing Estimator

After k independent permutations, π_1, π_2, ..., π_k, one can estimate R without bias, as a binomial:

$$\hat{R}_M = \frac{1}{k}\sum_{j=1}^{k} 1\{\min(\pi_j(S_1)) = \min(\pi_j(S_2))\},$$

$$\text{Var}\left(\hat{R}_M\right) = \frac{1}{k}R(1-R).$$

Problems with Minwise Hashing

- **Similarity search: high storage and computational cost**

  In industry, each hashed value, e.g., $\min(\pi(S_1))$, is usually stored using 64 bits. Often $k = 200$ samples (hashed values) are needed. The total storage can be prohibitive: $n \times 64 \times 200 = 16$ TB, if $n = 10^{10}$ (web pages).

  $\Longrightarrow$ computational problems, transmission problems, etc.

- **Statistical learning: high dimensionality and model size**

  We recently realize that minwise hashing can be used in statistical learning. However, 64-bit hashing leads to impractically high-dimensional model.

b-bit minwise hashing naturally solves both problems!

The Basic Probability Result for b-Bit Minwise Hashing

Consider two sets, S_1 and S_2,

$$S_1, S_2 \subseteq \Omega = \{0, 1, 2, ..., D-1\},$$
$$f_1 = |S_1|,\ f_2 = |S_2|,\ a = |S_1 \cap S_2|$$

Define the minimum values under $\pi : \Omega \rightarrow \Omega$ to be z_1 and z_2:

$$z_1 = \min\left(\pi\left(S_1\right)\right), \qquad z_2 = \min\left(\pi\left(S_2\right)\right).$$

and their b-bit versions

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

Example: if $z_1 = 7 (= 111$ in binary), then $z_1^{(1)} = 1$, $\ z_1^{(2)} = 3$.

Theorem

Assume D is large (which is virtually always true)

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

Recall, (assuming infinite precision, or as many digits as needed), we have

$$\mathbf{Pr}\left(z_1 = z_2\right) = R$$

Theorem

Assume D is large (which is virtually always true)

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

$$r_1 = \frac{f_1}{D}, \qquad r_2 = \frac{f_2}{D}, \qquad f_1 = |S_1|, \qquad f_2 = |S_2|,$$

$$C_{1,b} = A_{1,b}\frac{r_2}{r_1 + r_2} + A_{2,b}\frac{r_1}{r_1 + r_2},$$
$$C_{2,b} = A_{1,b}\frac{r_1}{r_1 + r_2} + A_{2,b}\frac{r_2}{r_1 + r_2},$$

$$A_{1,b} = \frac{r_1\left[1 - r_1\right]^{2^b - 1}}{1 - \left[1 - r_1\right]^{2^b}}, \qquad A_{2,b} = \frac{r_2\left[1 - r_2\right]^{2^b - 1}}{1 - \left[1 - r_2\right]^{2^b}}.$$

The Variance-Space Trade-off

Smaller $b \Longrightarrow$ smaller space for each sample. However,

smaller $b \Longrightarrow$ larger variance.

$B(b; R, r_1, r_2)$ characterizes the var-space trade-off:

$$B(b; R, r_1, r_2) = b \times \text{Var}\left(\hat{R}_b\right)$$

Lower $B(b)$ is better.

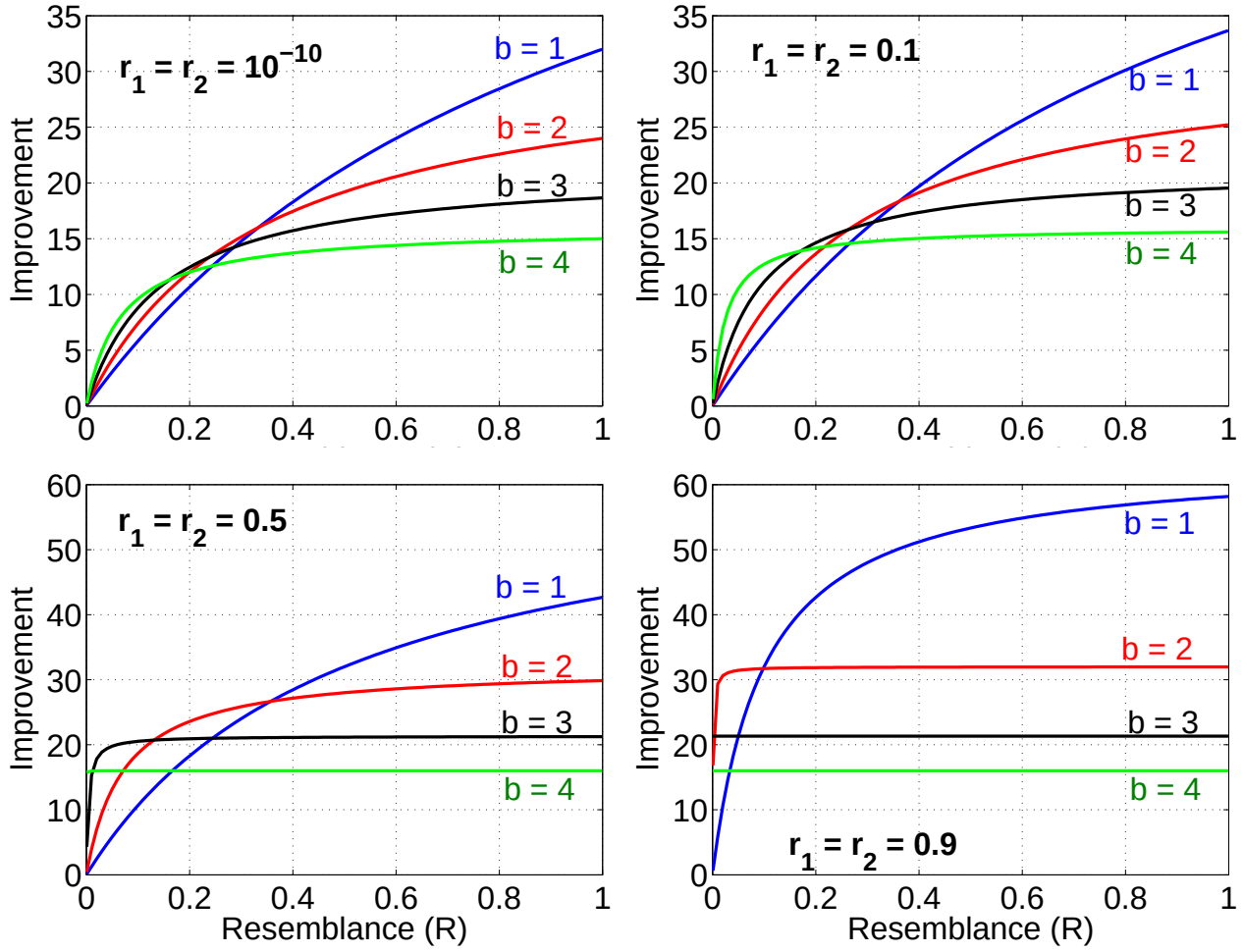
The ratio,

$$\frac{B(b_1; R, r_1, r_2)}{B(b_2; R, r_1, r_2)}$$

measures the improvement of using $b = b_2$ (e.g., $b_2 = 1$) over using $b = b_1$ (e.g., $b_1 = 64$).

$\frac{B(64)}{B(b)} = 20$ means, to achieve the same accuracy (variance), using $b = 64$ bits per hashed value will require 20 times more space (in bits) than using $b = 1$.

$\frac{B(64)}{B(b)}$, relative improvement of using $b = 1, 2, 3, 4$ bits, compared to 64 bits.



Relative Improvements in the Least Favorable Situation

In the least favorable condition, the improvement would be

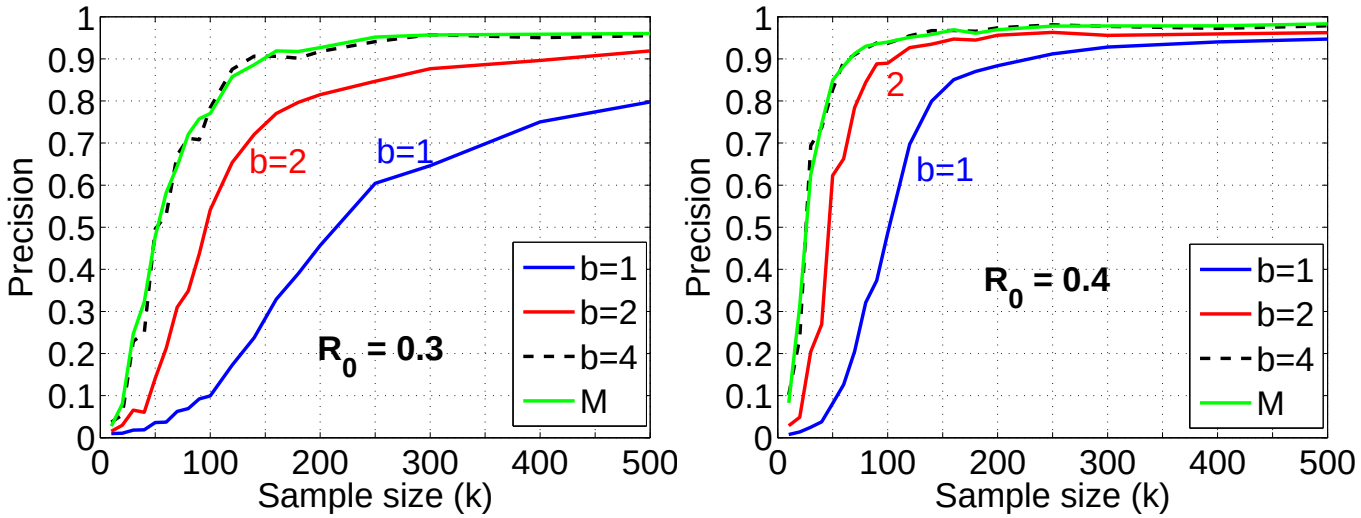
$$\frac{B(64)}{B(1)} = 64\frac{R}{R+1}$$

If $R = 0.5$ (the commonly used similarity threshold), then the improvement will be $\frac{64}{3} = 21.3$-fold, which is large enough for industry.

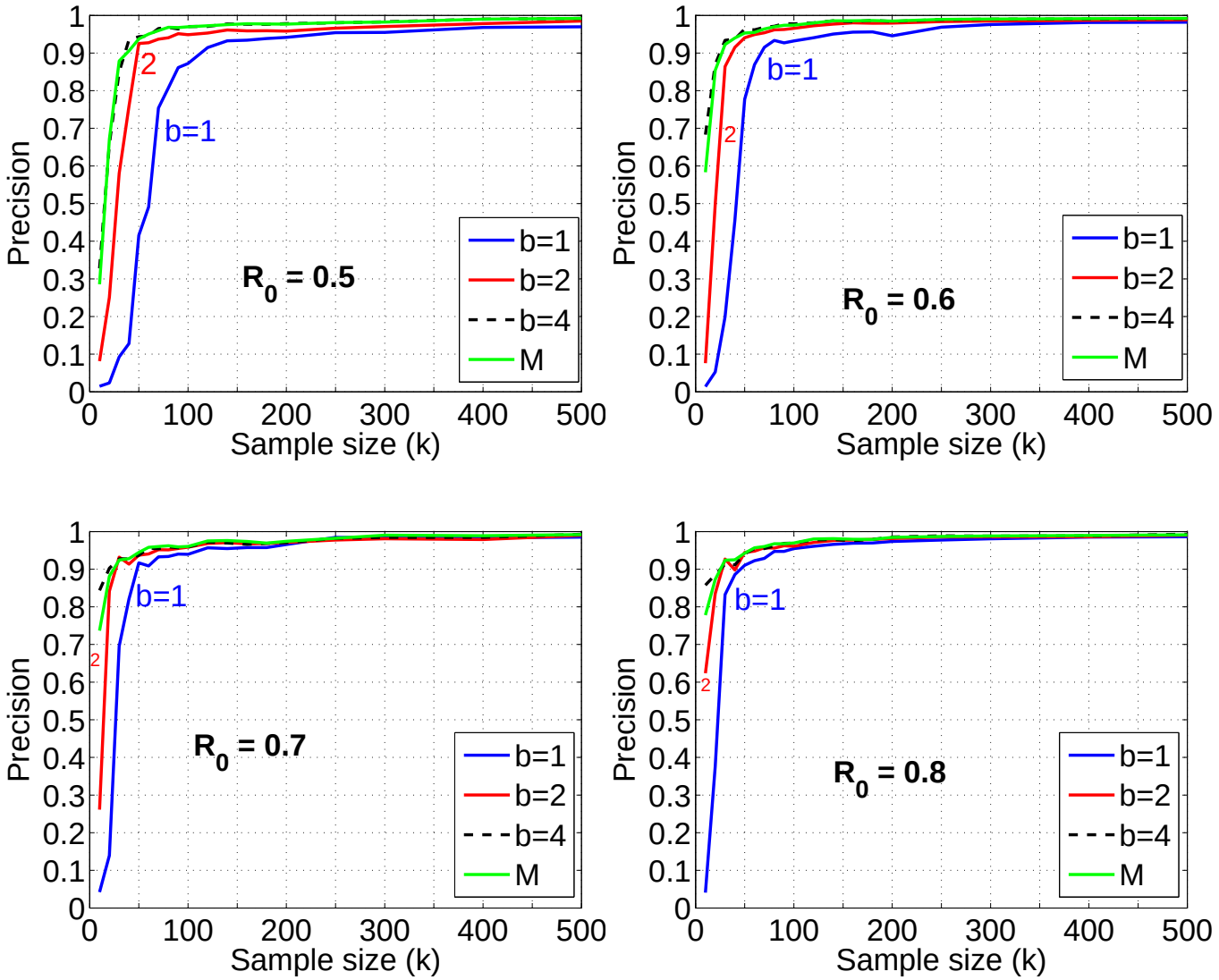
It turns out in terms of dimension reduction, the improvement is like $\frac{2^{64}}{2^8}$-fold.

Experiments: Duplicate detection on Microsoft news articles

The dataset was crawled as part of the BLEWS project at Microsoft. We computed pairwise resemblances for all documents and retrieved documents pairs with resemblance R larger than a threshold $\mathbf{R_0}$.



Using $b = 1$ or 2 is almost as good as $b = 64$ especially for higher threshold $\mathbf{R_0}$.



b-Bit Minwise Hashing for Large-Scale linear Learning

Linear learning algorithms require the estimators to be inner products.

If we look carefully, the estimator of minwise hashing is indeed an inner product of two (extremely sparse) vectors in $D \times k$ dimensions (infeasible when $D = 2^{64}$):

$$\hat{R}_M = \frac{1}{k} \sum_{j=1}^{k} 1\{\min(\pi_j(S_1)) = \min(\pi_j(S_2))$$

because, as $z_1 = \min(\pi(S_1))$, $z_2 = \min(\pi(S_2)) \in \Omega = \{0, 1, ..., D-1\}$.

$$1\{z_1 = z_2\} = \sum_{i=0}^{D-1} 1\{z_1 = i\} \times \{z_2 = i\}$$

For example, $D = 5$, $z_1 = 2$, $z_2 = 3$,

$1\{z_1 = z_2\} =$ inner product between $\{0, 0, 1, 0, 0\}$, and $\{0, 1, 0, 0, 0\}$.

Integrating b-Bit Minwise Hashing for (Linear) Learning

1. We apply k independent random permutations on each (binary) feature vector $\mathbf{x}_i$ and store the lowest b bits of each hashed value. This way, we obtain a new dataset which can be stored using merely nbk bits.

2. At run-time, we expand each new data point into a $2^b \times k$-length vector, i.e. we concatenate the k vectors (each of length 2^b). The new feature vector has exactly k 1's.

Consider an example of $k = 3$ and $b = 2$. For set (vector) S_1:

Original hashed values $(k = 3)$: $12013 \quad 25964 \quad 20191$

Original binary representations :

$010111011101101 \qquad 110010101101100 \qquad 100111011011111$

Lowest $b = 2$ binary digits : $01 \;\; 00 \;\; 11$

Corresponding decimal values : $1 \;\; 0 \;\; 3$

Expanded $2^b = 4$ binary digits : $0010 \qquad 0001 \qquad 1000$

New feature vector fed to a solver : $\{0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0\}$

Then we apply the same permutations and procedures on sets S_2, S_3, ...

Datasets and Solver for Linear Learning

This talk only presents two text datasets. Experiments on image data (1TB or larger) will be presented in the future.

Dataset	# Examples (n)	# Dims (D)	Avg # Nonzeros	Train / Test
Webspam (24 GB)	350,000	16,609,143	3728	80% / 20%
Rcv1 (200 GB)	781,265	1,010,017,424	12062	50% / 50%

We chose LIBLINEAR as the basic solver for linear learning. Note that our method is purely statistical/problilsitic and is independent of the underlying optimization procedures.

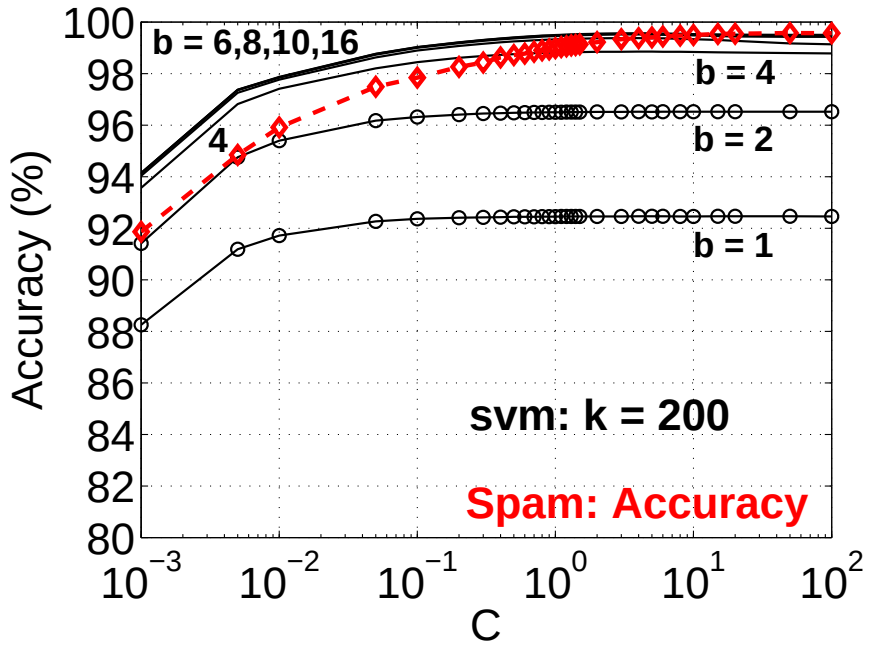
All experiments were conducted on workstations with Xeon(R) CPU (W5590@3.33GHz) and 48GB RAM, under Windows 7 System.

Experimental Results on Linear SVM and Webspam

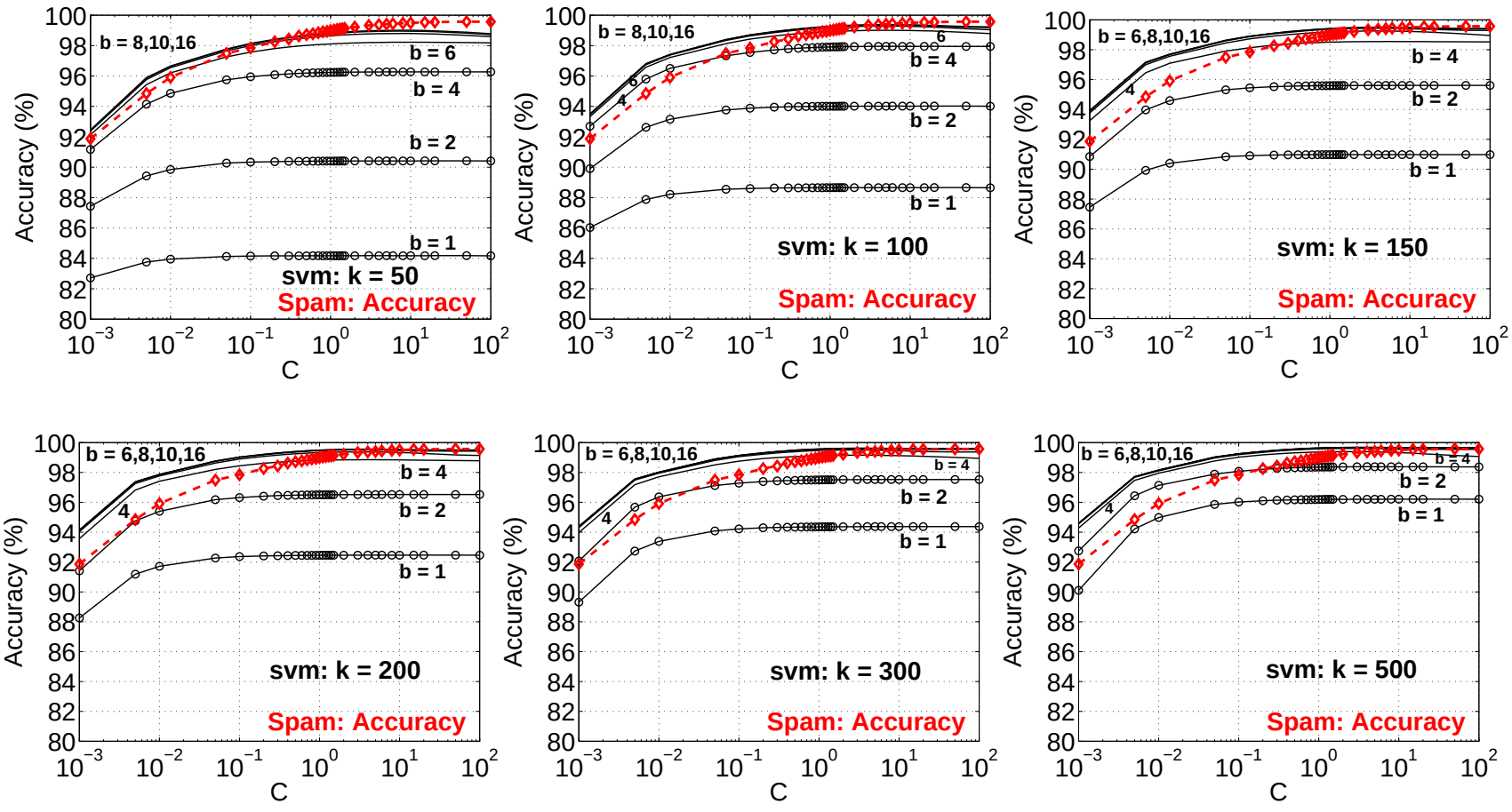
- We conducted our extensive experiments for a wide range of regularization C values (from 10^{-3} to 10^2) with fine spacings in $[0.1,\ 10]$.

- We experimented with $k = 30$ to $k = 500$, and $b = 1$, 2, 4, 8, and 16.

Testing Accuracy

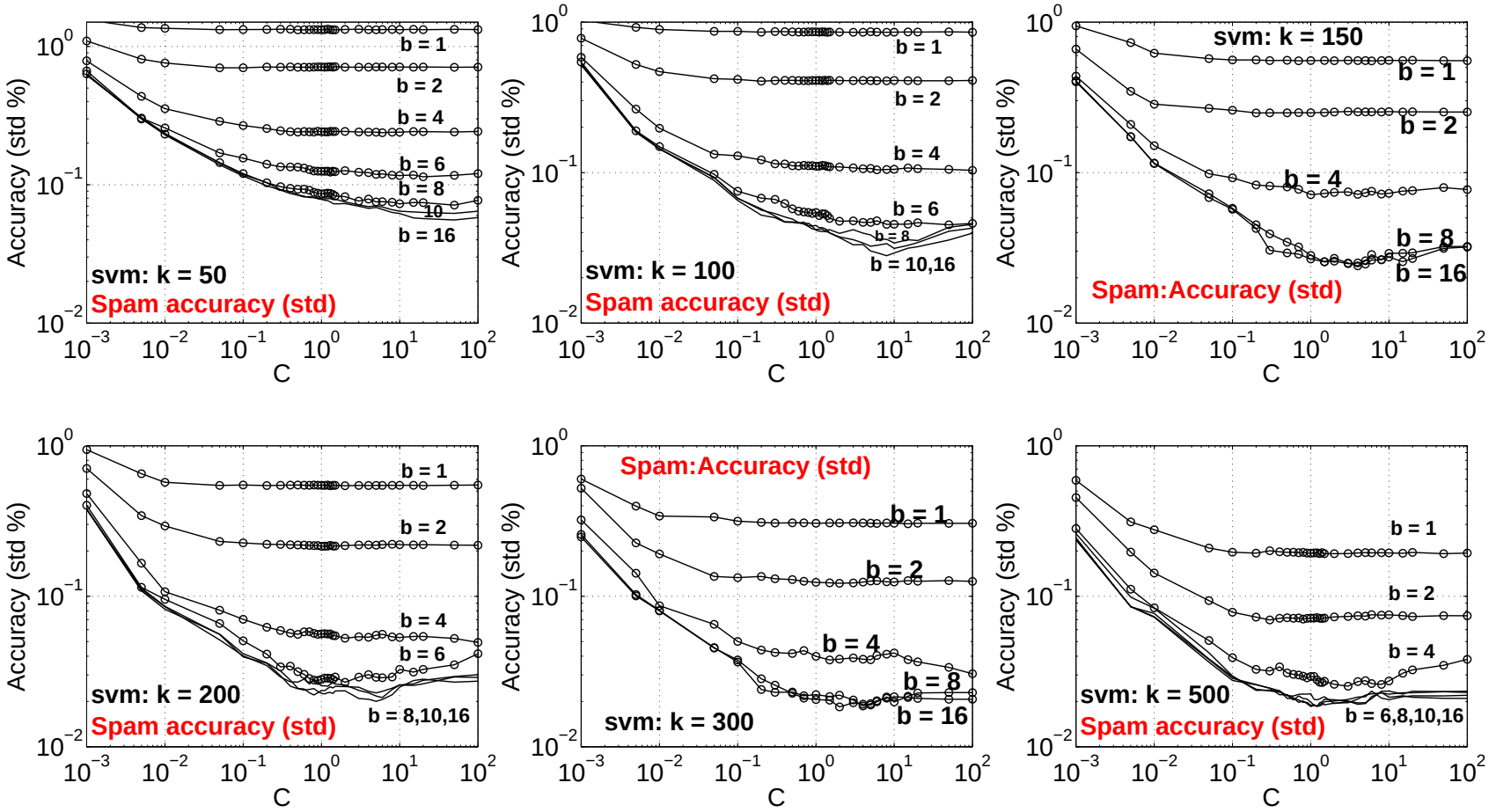


- Solid: b-bit hashing. Dashed (red) the original data
- Using $b \geq 8$ and $k \geq 200$ achieves about the same test accuracies as using the original data.
- The results are averaged over 50 runs.

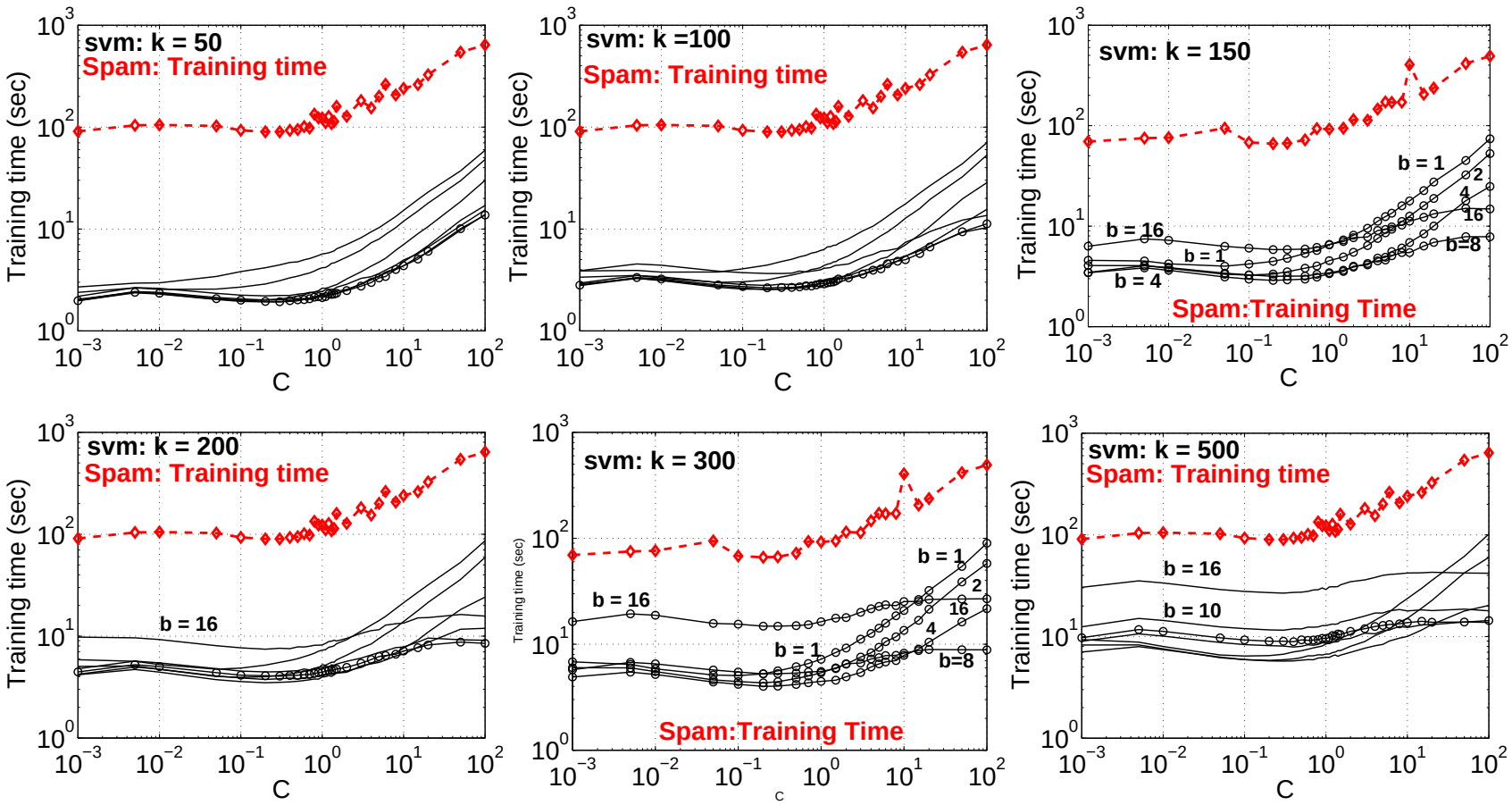

b = 8,10,16
b = 6
b = 4
b = 2
b = 1
svm: k = 50
Spam: Accuracy
svm: k = 100
svm: k = 150
b = 6,8,10,16
svm: k = 200
svm: k = 300
svm: k = 500
Accuracy (%)
C

Stability of Testing Accuracy (Standard Deviation)

Our method produces very stable results, especially $b \geq 4$

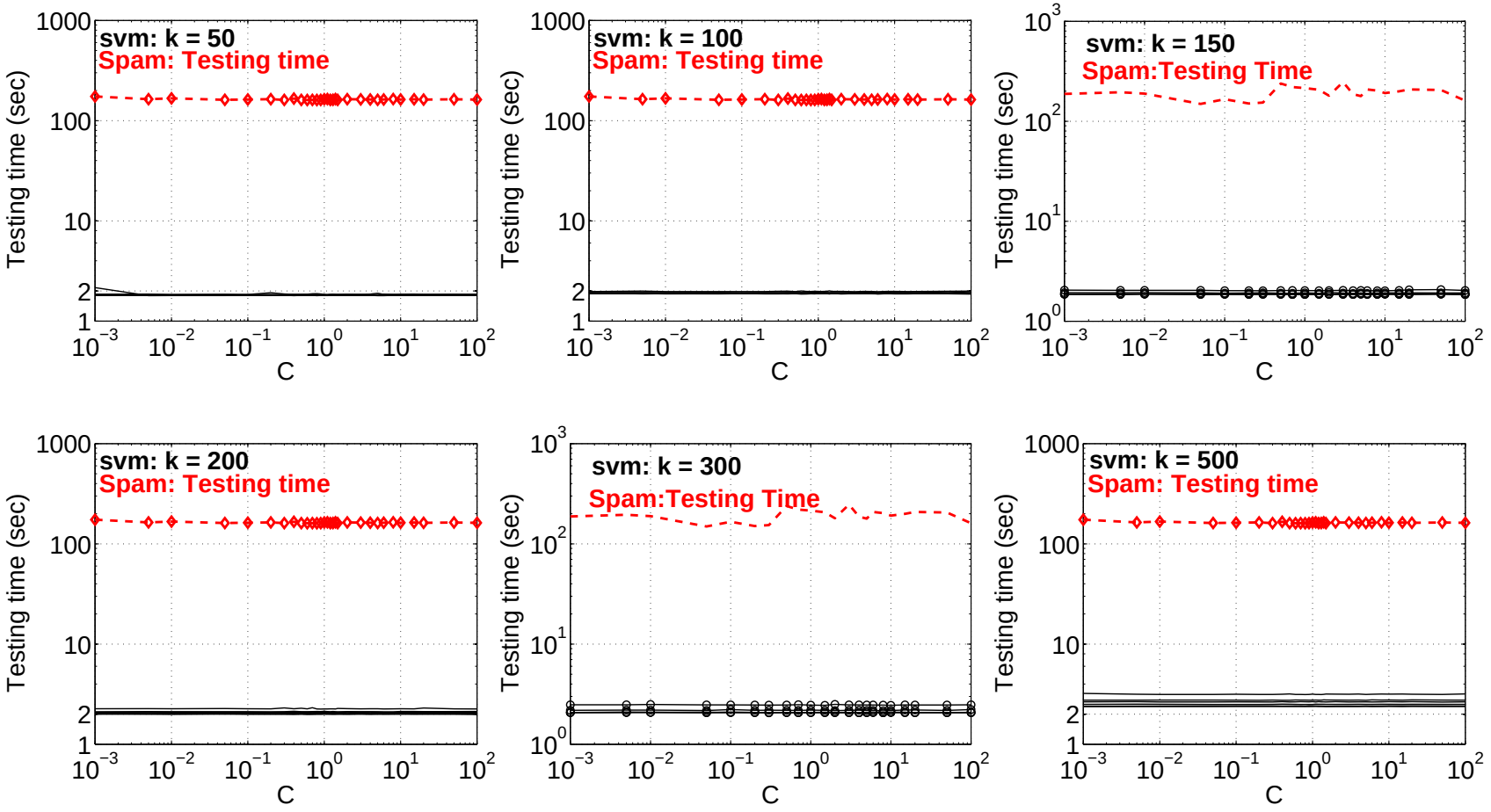


Training Time



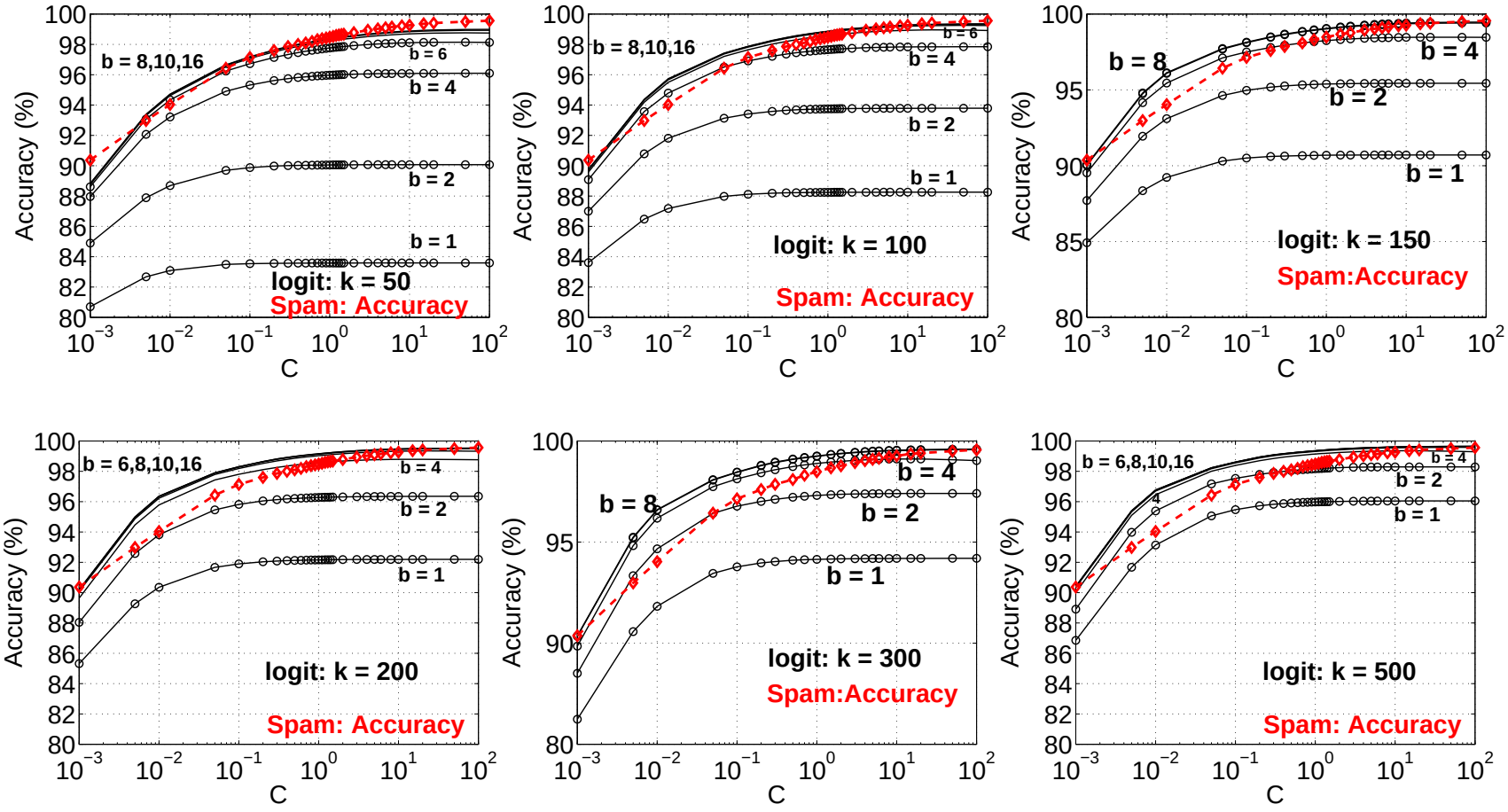
- They did not include data loading time (which is small for b-bit hashing)
- The original training time is about 100 seconds.
- b-bit minwise hashing needs about $3 \sim 7$ seconds (3 seconds when $b = 8$).

Testing Time

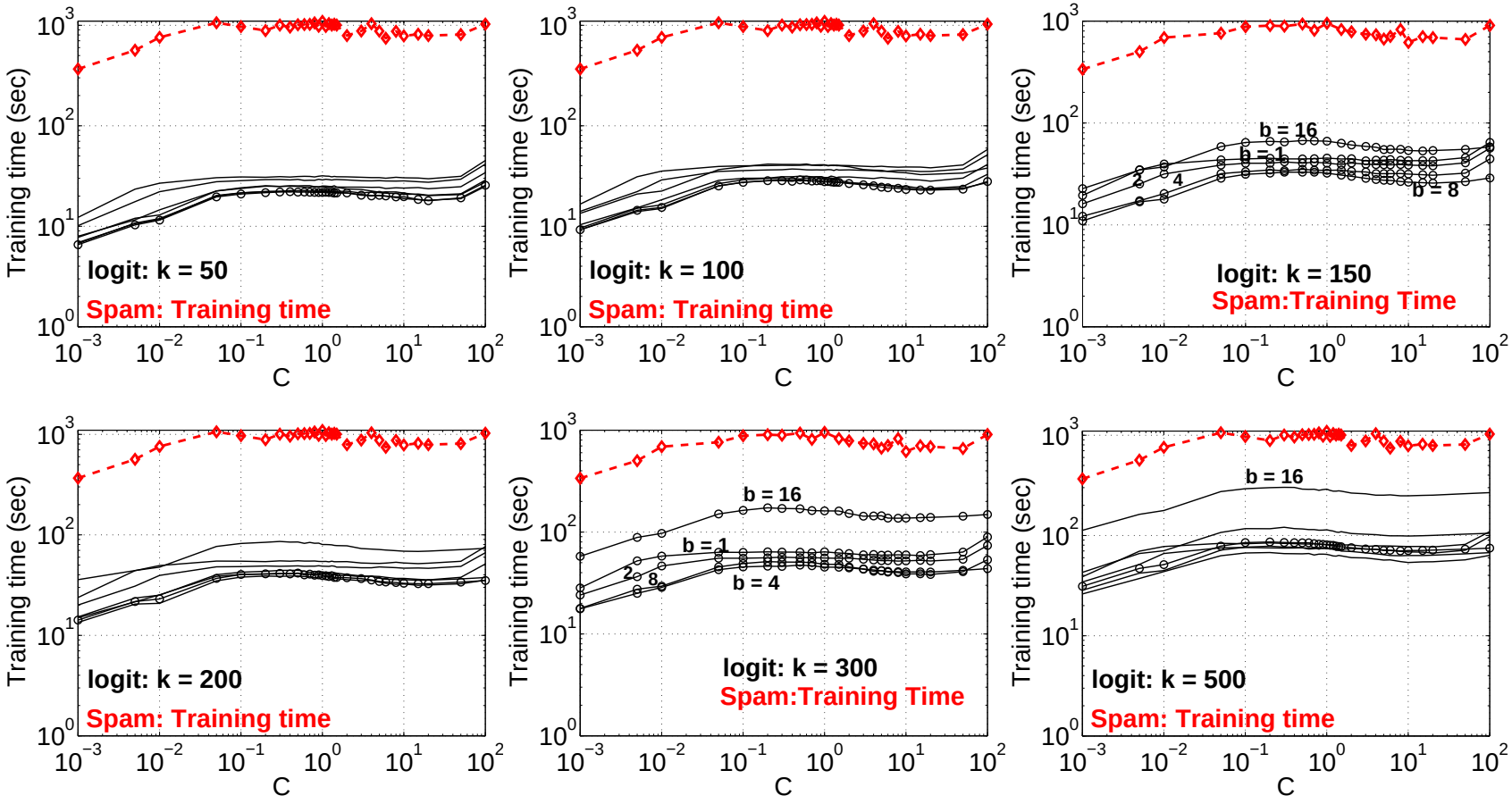


Experimental Results on L_2-Regularized Logistic Regression

Testing Accuracy



Training Time



Comparisons with Other Algorithms

We conducted extensive experiments with the VW algorithm, which has the same variance as random projections. Vowpal Wabbit (VW) refers to the algorithm in (Weinberger et. al., ICML 2009).

Consider two sets S_1, S_2, $f_1 = |S_1|$, $f_2 = |S_2|$, $a = |S_1 \cap S_2|$, $R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{a}{f_1 + f_2 - a}$. Then, from their variances

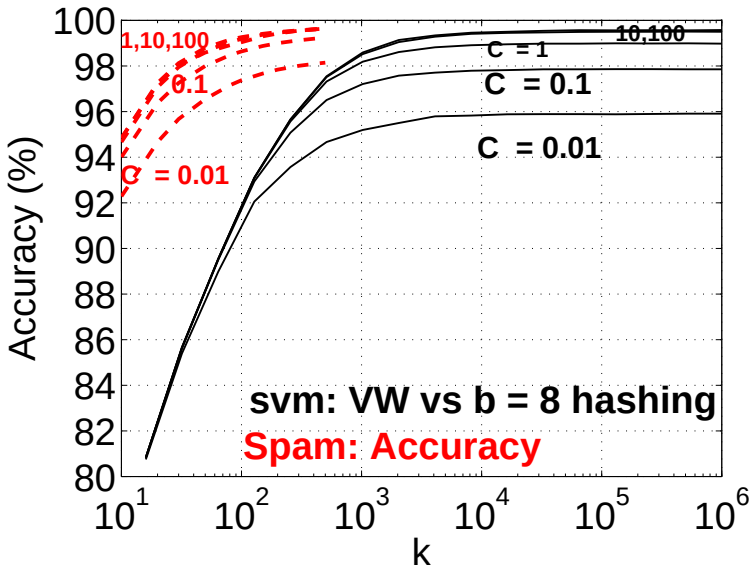
$$Var\left(\hat{a}_{VW}\right) \approx \frac{1}{k}\left(f_1 f_2 + a^2\right)$$

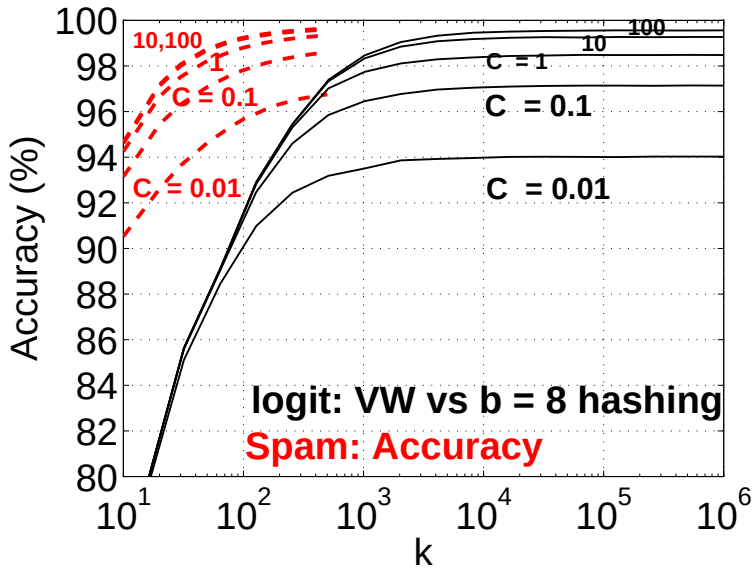
$$Var\left(\hat{R}_{MINWISE}\right) = \frac{1}{k} R\left(1 - R\right)$$

we can immediately see the significant advantages of minwise hashing, especially when $a \approx 0$ (which is common in practice).

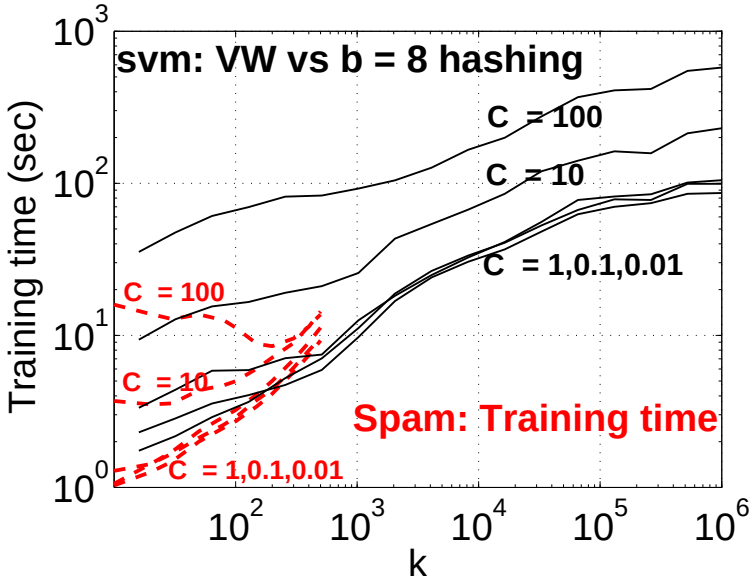
Comparing b-Bit Minwise Hashing with VW

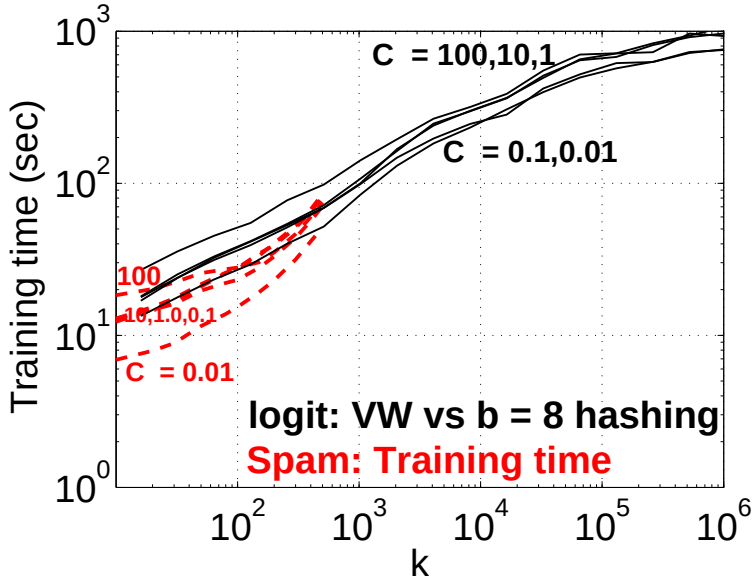
8-bit minwise hashing (dashed, red) with $k \geq 200$ (and $C \geq 1$) achieves about the same test accuracy as VW with $k = 10^4 \sim 10^6$.





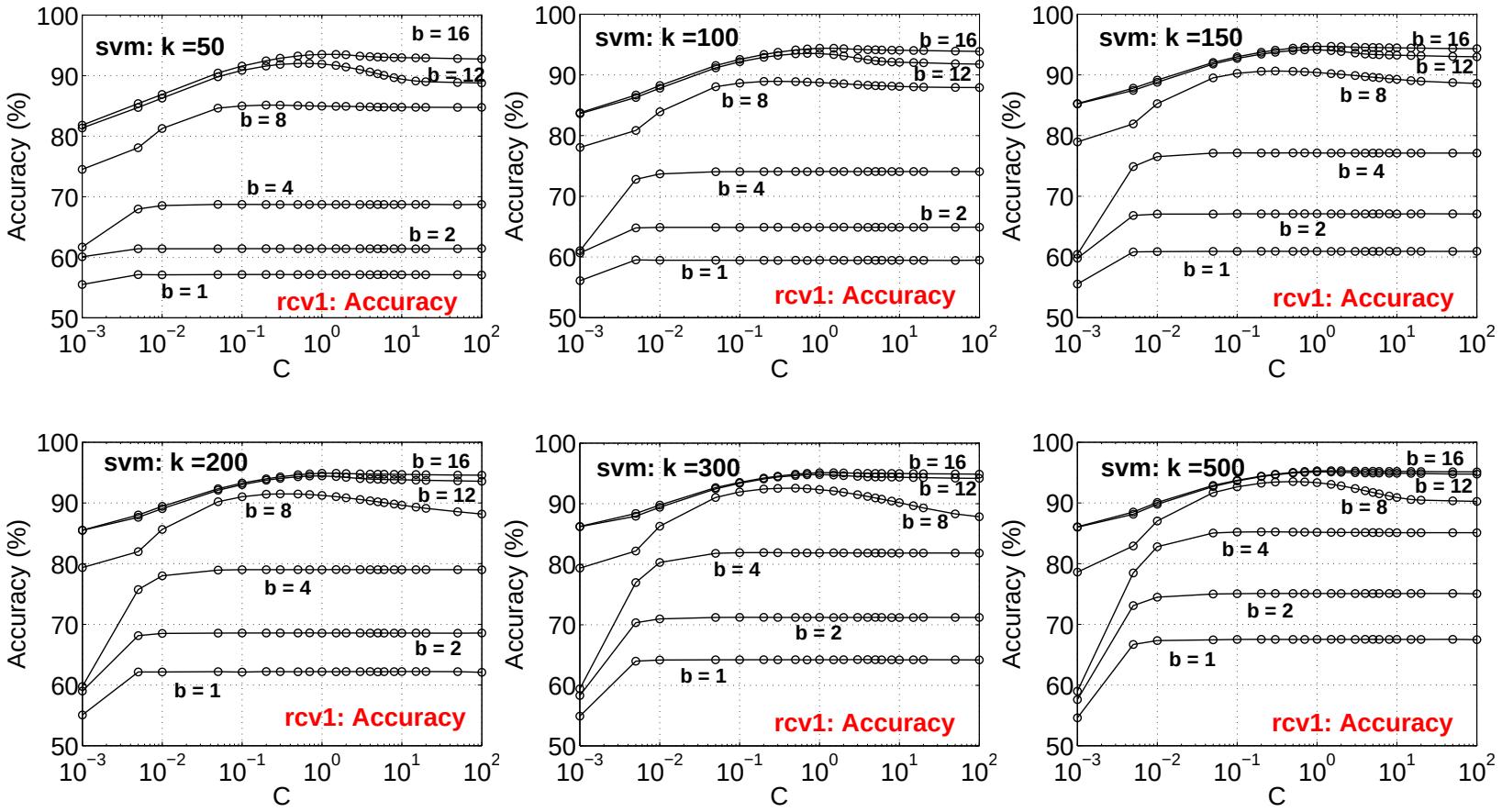
8-bit hashing is substantially faster than VW (to achieve the same accuracy).



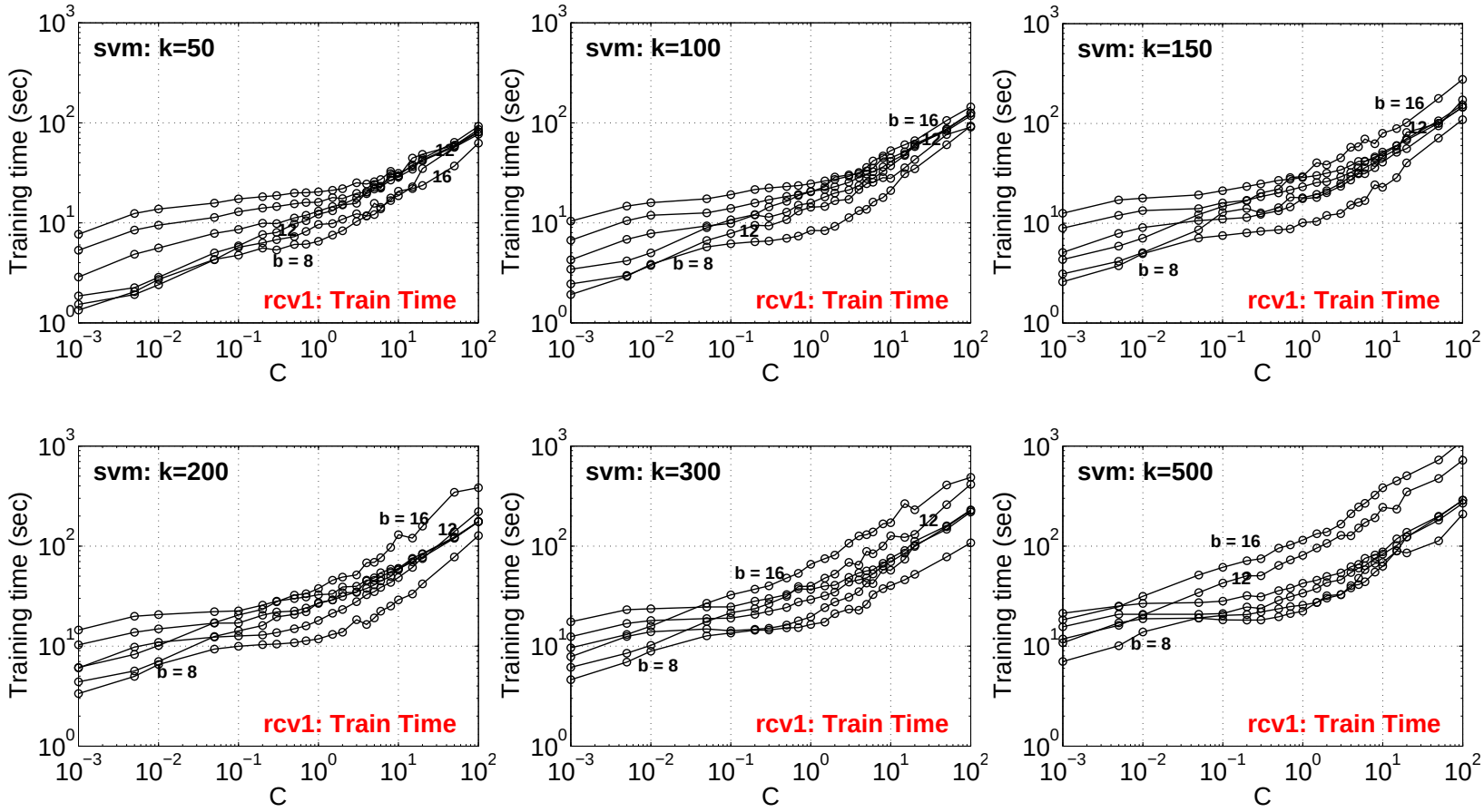


Experimental Results on Rcv1 (200GB)

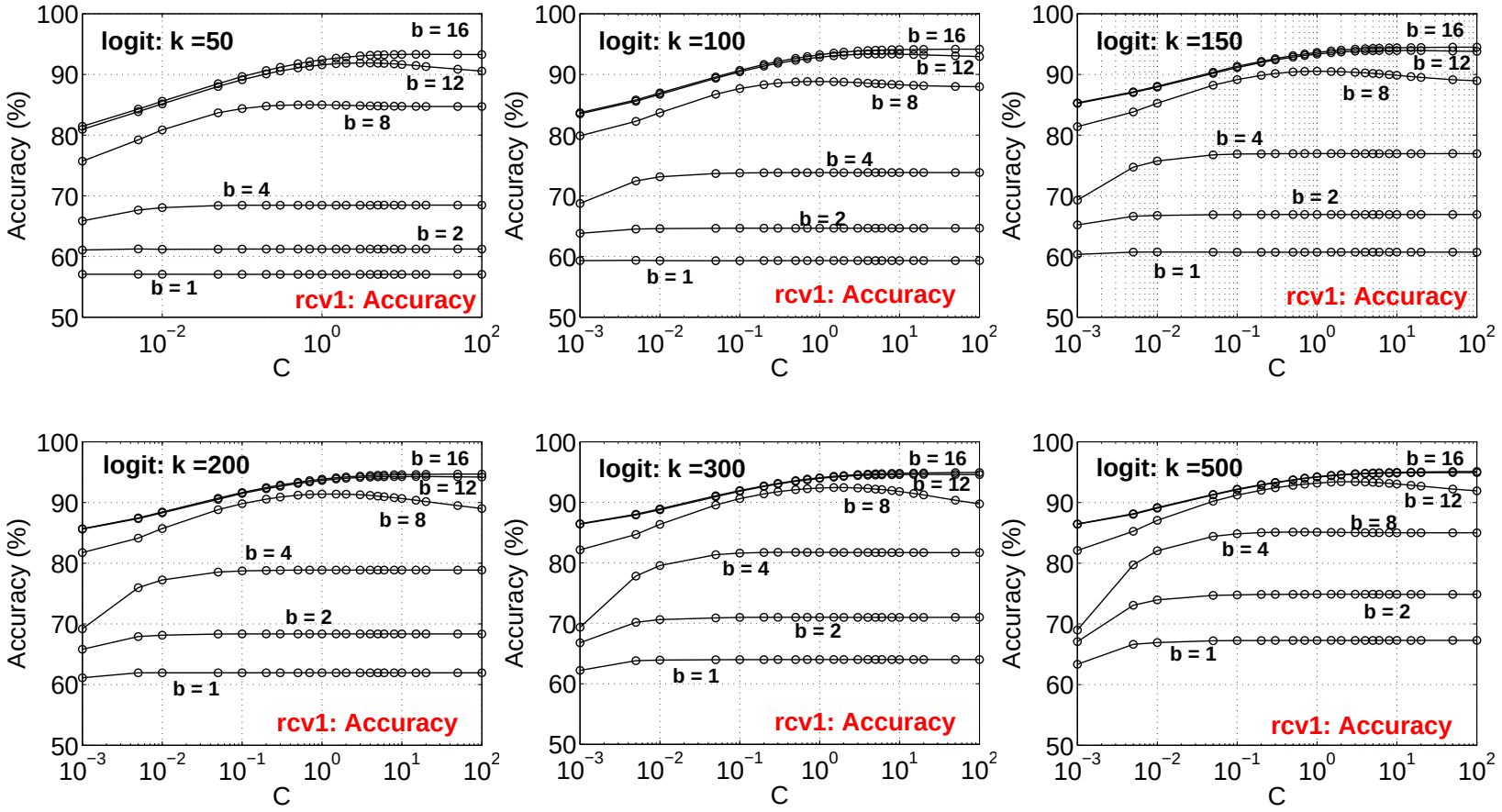
Test accuracy using linear SVM (Can not train the original data)



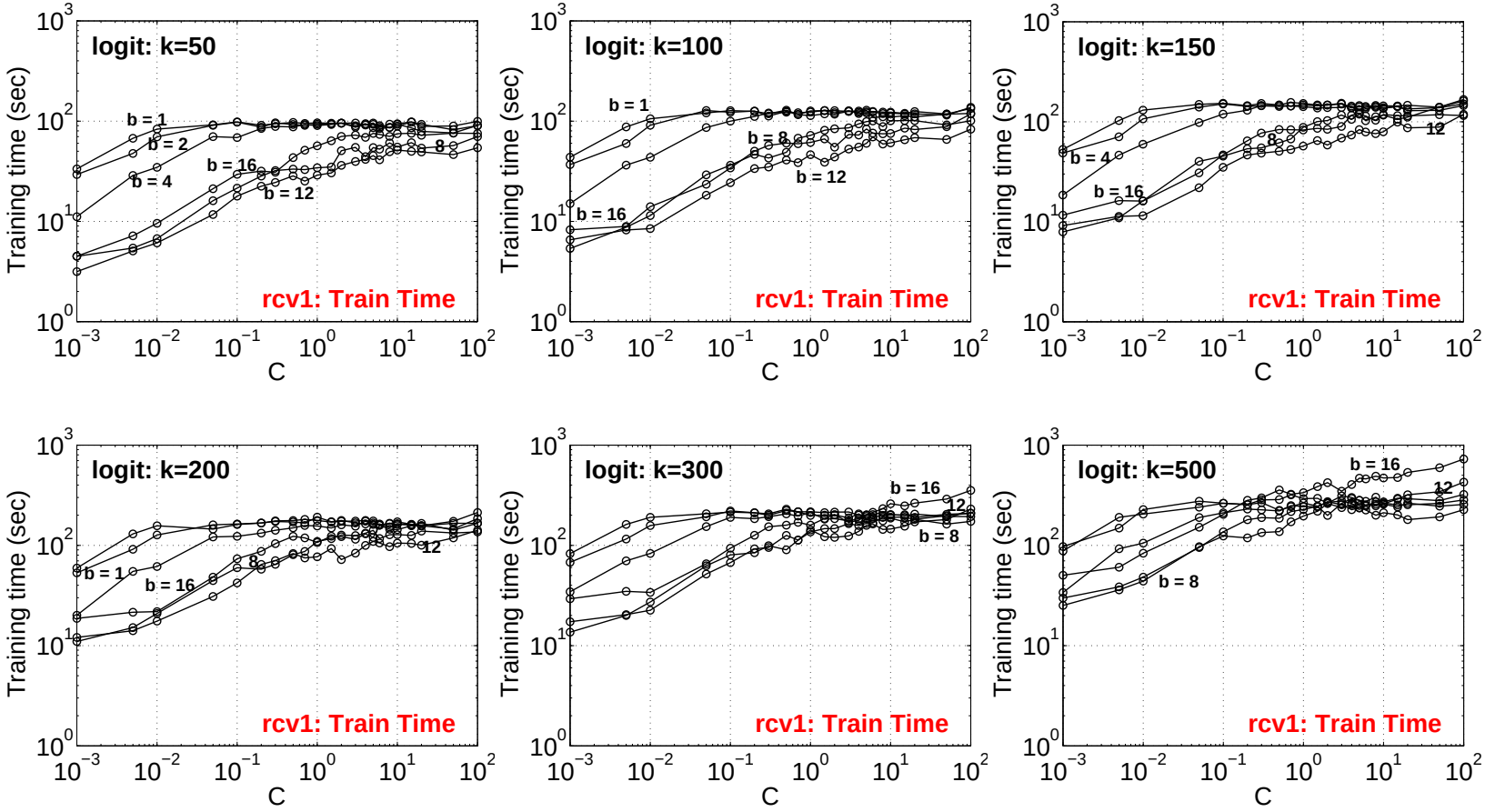
Training time using linear SVM



Test accuracy using logistic regression

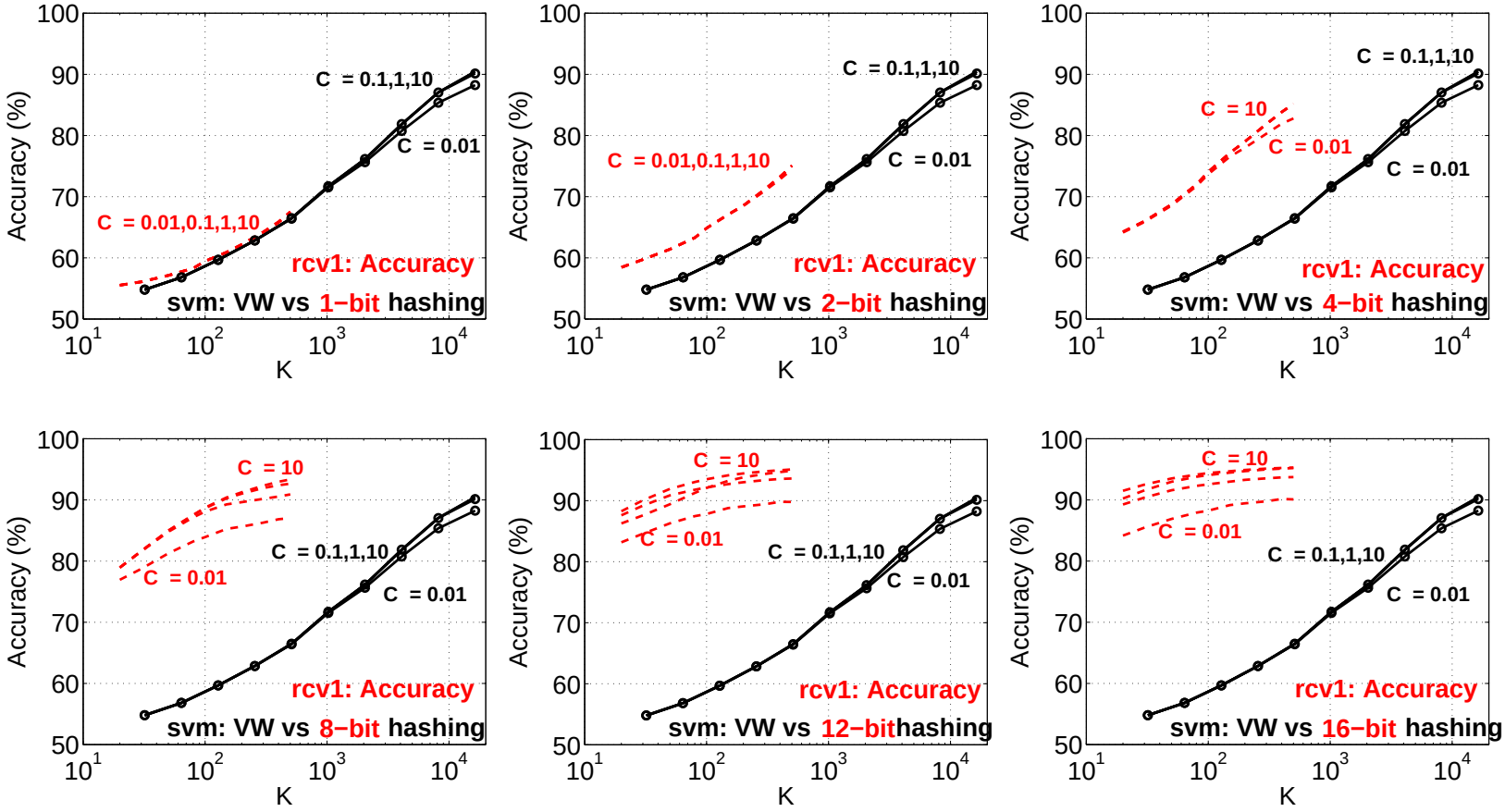


Training time using logistic regression

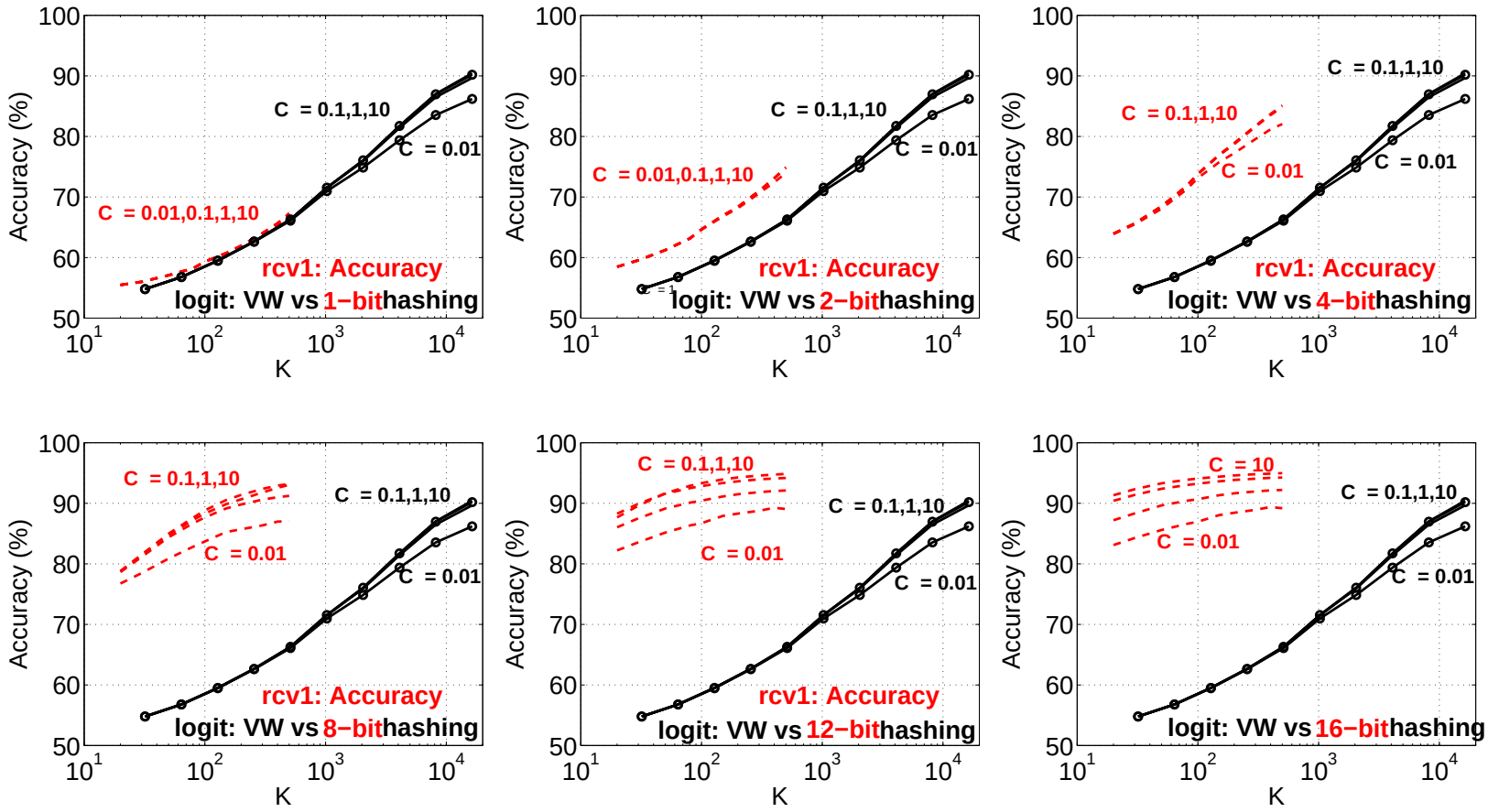


Comparisons with VW on Rcv1

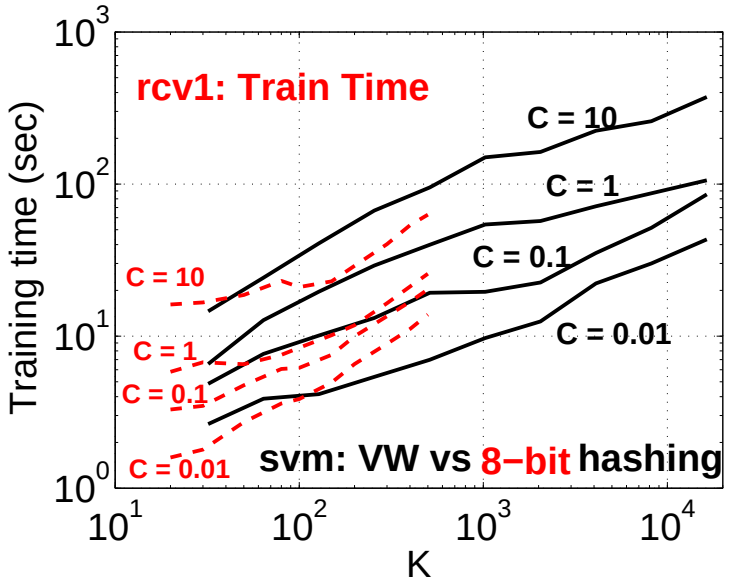
Test accuracy using linear SVM

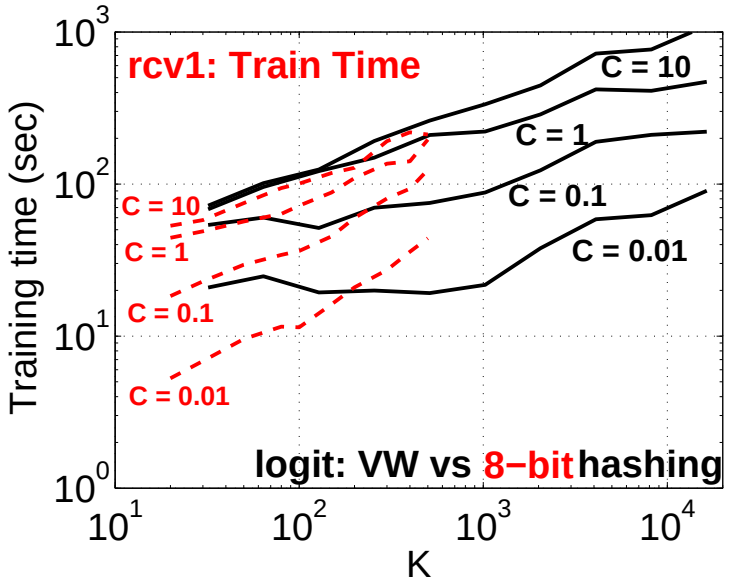


Test accuracy using logistic regression



Training time





Conclusion

- Minwise Hashing is the standard technique in the context of search.
- b-bit minwise hashing is a substantial improvement by using only b bits per hashed value instead of 64 bits (e.g., a 20-fold reduction in space).
- Our work is the first proposal to use minwise hashing in linear learning. However, the resultant data dimension may be too large to be practical.
- b-bit minwise hashing provides a very simple solution, by dramatically reducing the dimensionality from (e.g.,) $2^{64} \times k$ to $2^b \times k$.
- Thus, b-bit minwise hashing can be easily integrated with linear learning, to solve extremely large problems, especially when data do not fit in memory.
- b-bit minwise hashing has substantial advantages (in many aspects) over other methods such as random projections and VW.

Questions?