Probabilistic Hashing Algorithms

for Large-Scale Search and Learning

Cun-Hui Zhang

Department of Statistics and Biostatistics

Rutgers University

Pisctaway, NJ 08854, USA

Outline

1. **Hashing Algorithms for Large-Scale Learning**

2. **Hashing algorithms for indexing and efficient near neighbor search**

Hashing algorithms for machine learning

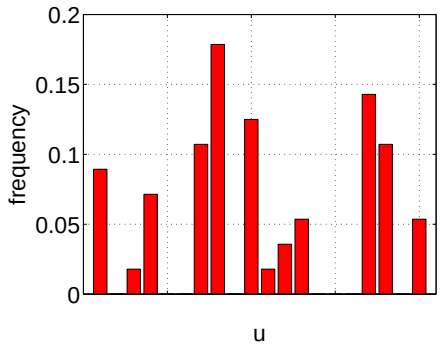
Hashing + logistic regression and hashing + DNN can often be nice combinations:

- Hashing for dealing with ultra-high-dimensional data
- Hashing for building compact (e.g., single machine) learning models
- Hashing for building more complex (and more accurate) learning models
- Hashing as a feature engineering tool
- Hashing for building nonlinear learning models at linear cost

One Major Source of High-Dimensional Data: Histogram

Histogram-based features are very popular in practice, for example, natural language processing (NLP) and computer vision.

It can be viewed as high-dimensional vector: $u_i \geq 0, \;\; i = 1, 2, ..., D$



The size of the space D can often be extremely large. For example, D can be the total number of words, or combinations of words (or characters, or visual words).

In search industry, $D = 2^{64}$ is often used, for convenience.

An Example of Text Data Representation by n-grams

Each document (Web page) can be viewed as a set of n-grams.

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

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

It is common to use $n \geq 5$.

Using n-grams generates extremely high dimensional vectors, e.g., $D = (10^5)^n$.

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

As a highly successful practice, n-gram representations have many variants, e.g., word n-grams, character n-grams, skip n-grams, etc.

Webspam: A Small Example of n-gram Data

Task: Classifying 350K documents into spam or non-spam (binary classification).

	Dim. (D)	Training Time	Accuracy
1-gram + linear SVM	254	20 sec	93.30%
3-gram + linear SVM	16,609,143	200 sec	99.6%
3-gram + kernel SVM	16,609,143	**About a Week**	99.6%

(Character) 1-gram: Frequencies of occurrences of single characters.

(Character) 3-gram: Frequencies of occurrences of 3-contiguous characters.

————-

Ref: P. Li, et. al. Hashing Algorithms for Large-scale Learning, NIPS 2011

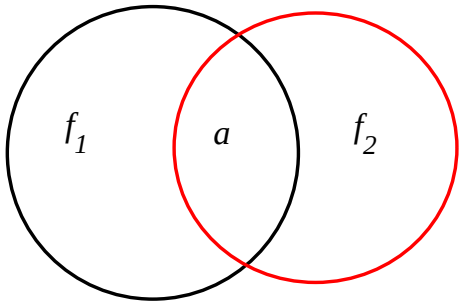
Challenges with Using High-Dimensional Features

- **High dimensionality** This may lead to large & costly (in both training and testing) statistical models and create large search space.
- **High storage cost** If the (expanded) features are fully materialized, they might be way too large to store/transmit, even for very sparse data.
- **Binary vs. non-binary** While in NLP and search it is popular to use very high-dimensional and binary representations, the current mainstream practice in (e.g.,) computer vision is to use non-binary features. In general, binary representations require a much large space (dimensionality).

Minwise Hashing for Binary (0/1) Data

A binary (0/1) vector $\Longleftrightarrow$ 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

$$R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{a}{f_1 + f_2 - a}. \qquad \text{(compared to "cosine" } \frac{a}{\sqrt{f_1 f_2}})$$

Minwise Hashing: Standard Algorithm in the Context of Search

The standard practice in the search industry: (e.g., Broder et al 1997)

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 example, $\min(\pi(S_1)) \neq \min(\pi(S_2))$.

Minwise Hashing in 0/1 Data Matrix

Original Data Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_1:	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0
S_2:	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0
S_3:	0	0	0	1	0	0	1	1	0	0	0	0	0	0	1	0

Permuted Data Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(S_1)$:	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	0
$\pi(S_2)$:	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
$\pi(S_3)$:	1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0

$$\min(\pi(S_1)) = 2, \quad \min(\pi(S_2)) = 0, \quad \min(\pi(S_3)) = 0$$

An Example with $k = 3$ Permutations

Input: sets S_1, S_2, ...,

Hashed values for S_1 : 113 264 1091

Hashed values for S_2 : 2049 103 1091

Hashed values for S_3 : ...

....

Minwise Hashing Estimator

After k permutations, π_1, π_2, ..., π_k, one can estimate R without bias:

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

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

Issues with Minwise Hashing and Solutions

1. **Expensive storage (and computation)**: In the standard practice, each hashed value was stored using 64 bits.

   Solution: b-bit minwise hashing by using only the lowest b bits.

2. **How to do linear kernel learning**:

   Solution: One can show that b-bit minwise hashing results in positive definite (PD) linear kernel matrix. The data dimensionality is reduced from 2^{64} to 2^b.

3. **Expensive and energy-consuming (pre)processing for k permutations**:

   Solution: One permutation hashing, which is even more accurate.

Integrating b-Bit Minwise Hashing for (Linear) Learning

Very simple:

1. Apply k independent random permutations on each (binary) feature vector $\mathbf{x}_i$ and store the lowest b bits of each hashed value. The storage costs nbk bits.

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

An Example with $k = 3$ Permutations

Input: sets S_1, S_2, ...,

Hashed values for S_1 :	113	264	1091
Hashed values for S_2 :	2049	103	1091
Hashed values for S_3 : ...			
....			

Observation: the estimator can be written as an inner product

$$\begin{aligned}\hat{R}_M =& \frac{1}{k}\sum_{j=1}^{k} 1\{\min(\pi_j(S_1)) = \min(\pi_j(S_2))\} \\ =& \frac{1}{k}\sum_{j=1}^{k}\sum_{i=1}^{D} 1\{\min(\pi_j(S_1)) = i\} \times 1\{\min(\pi_j(S_2) = i\}\}\end{aligned}$$

If D is too large, we take the lowest b bits and the space is only of size 2^b.

Ref: P. Li, et. al. Hashing Algorithms for Large-scale Learning, NIPS 2011

Suppose $D = 4$. Then we can expand each number in $\{0,\ 1,\ 2,\ 3\}$ as a vector of length 4.

$$0 \Longrightarrow [1\ \ 0\ \ 0\ \ 0], \qquad 1 \Longrightarrow [0\ \ 1\ \ 0\ \ 0], \qquad 2 \Longrightarrow [0\ \ 0\ \ 1\ \ 0], \qquad 3 \Longrightarrow [0\ \ 0\ \ 0\ \ 1]$$

Then the indicator function can be evaluated using inner product between two vectors:

$$1\{2 = 3\} =< [0\ \ 0\ \ 1\ \ 0], [0\ \ 0\ \ 0\ \ 1] >= 0$$

$$1\{1 = 1\} =< [0\ \ 1\ \ 0\ \ 0], [0\ \ 1\ \ 0\ \ 0] >= 1$$

An Example with $k = 3$ Permutations and $b = 2$ Bits

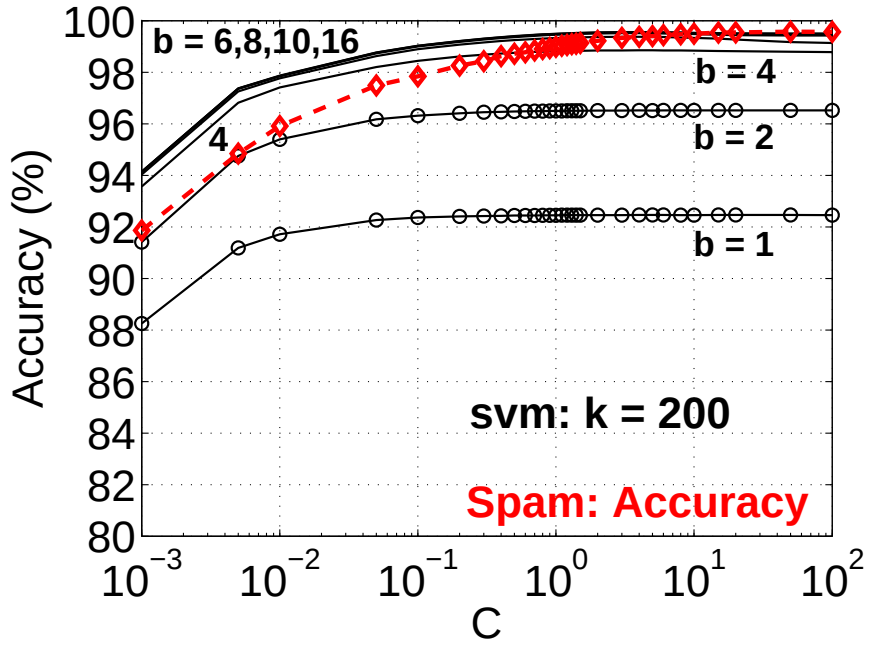
For set (vector) S_1: (Original high-dimensional binary feature vector)

Hashed values :	113	264	1091
Binary :	1110001	100001000	10001000011
Lowest $b = 2$ bits :	01	00	11
Decimal values :	1	0	3
Expansions (2^b) :	0100	1000	0001

New binary feature vector : $[0, 1, 0, 0, \quad 1, 0, 0, 0, \quad 0, 0, 0, 1] \times \frac{1}{\sqrt{3}}$

Same procedures on sets S_2, S_3, ...

Experiments on Webspam (3-gram) Data: Testing Accuracy



- C is the l_2 regularization parameter of linear SVM.
- Dashed: using the original data (**24GB** disk space).
- Solid: b-bit hashing. Using $b = 8$ and $k = 200$ achieves about the same test accuracies as using the original data. Space: **70MB** (350000×200)

What Is Happening?

	Dim. (D)	Training Time	Accuracy
1-gram + linear SVM	254	20 sec	93.30%
3-gram + linear SVM	16,609,143	200 sec	99.6%
3-gram + kernel SVM	16,609,143	**About a Week**	99.6%

1. **By engineering:**

   Webspam, unigram, 254 dim $\Longrightarrow$3-gram, 16M dim, 4000 nonzeros per doc

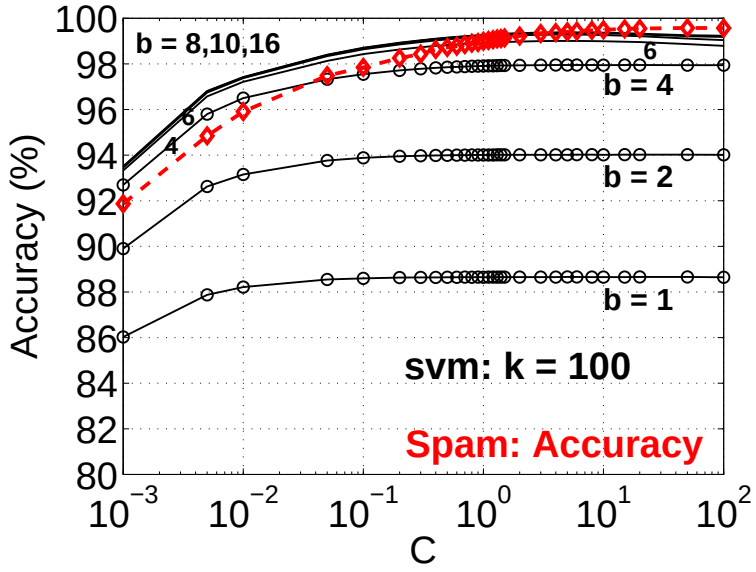
   Accuracy: $93.3\% \Longrightarrow 99.6\%$

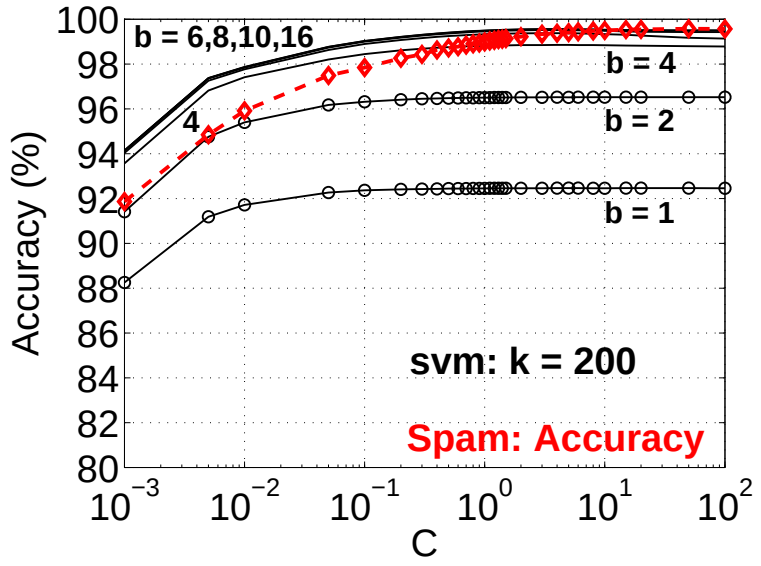
2. **By probability/statistics:**

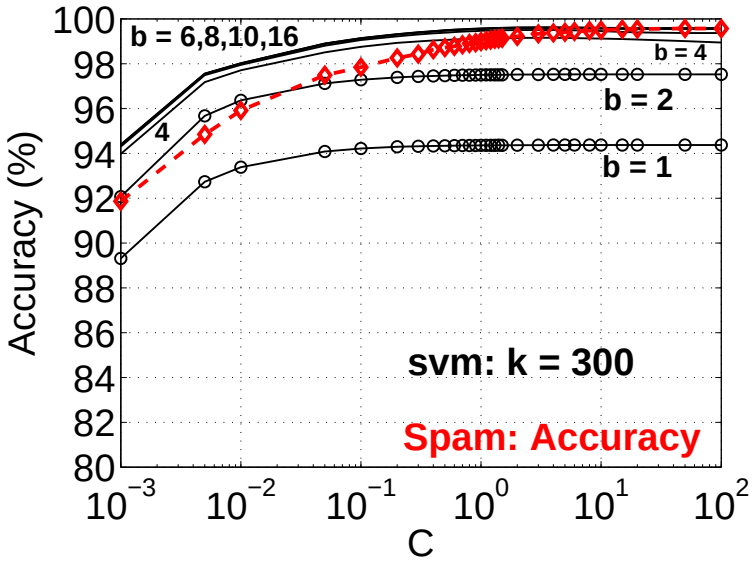
   16M dim, 4000 nonzeros $\Longrightarrow k = 200$ nonzeros, $2^8 \times k = 51200$ dim

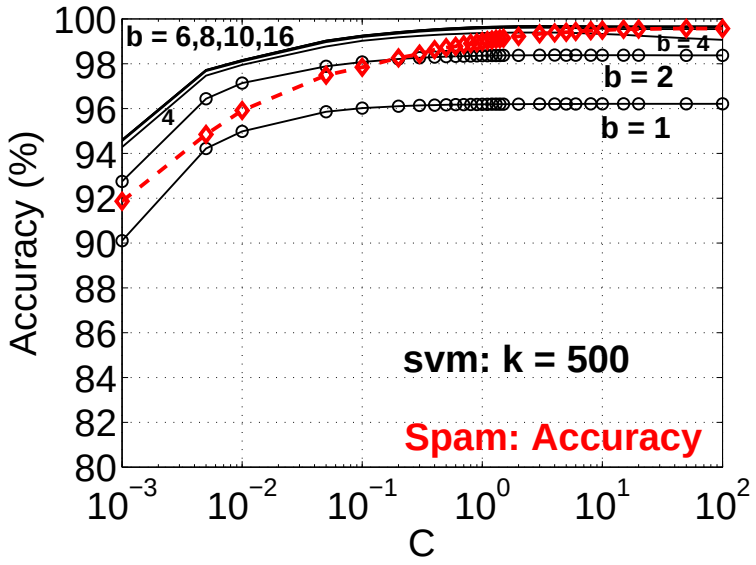
   Accuracy: $99.6\% \Longrightarrow 99.6\%$

Hashing can be viewed as part of feature engineering

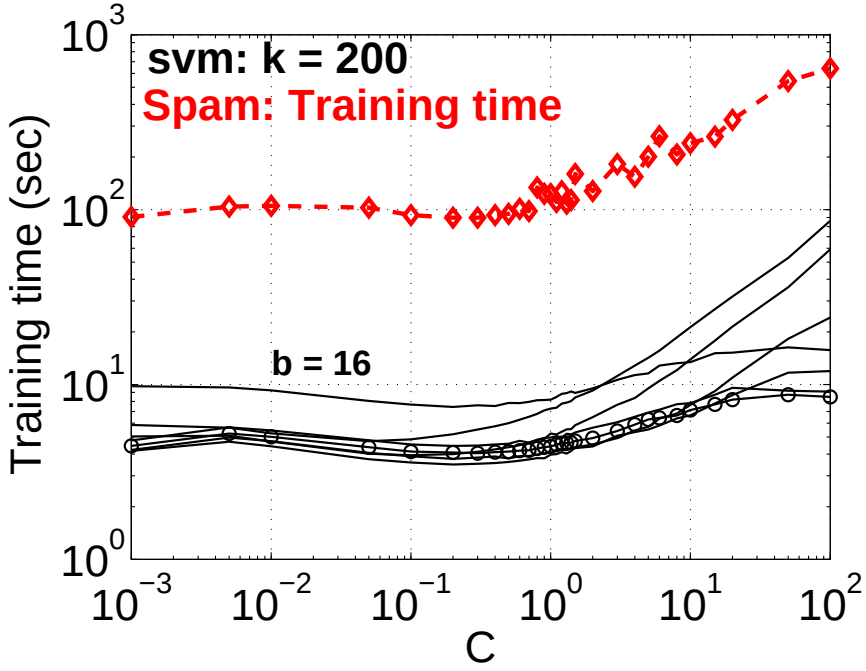

b = 8,10,16
6
4
b = 4
b = 2
b = 1
svm: k = 100
Spam: Accuracy
Accuracy (%)
C


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


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

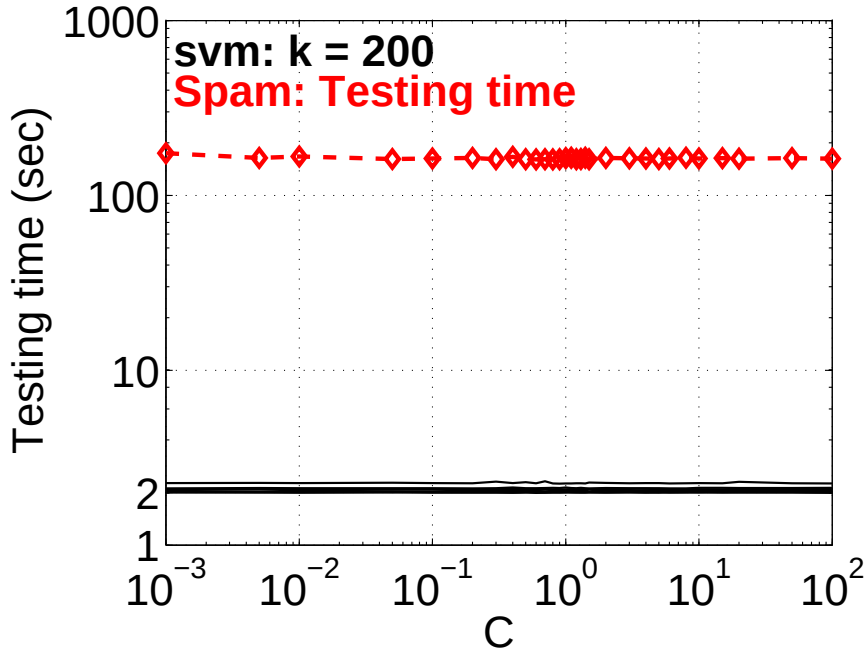

b = 6,8,10,16
b = 4
b = 2
4
b = 1
svm: k = 500
Spam: Accuracy
Accuracy (%)
C

Training Time



- They did not include data loading time (which is small for b-bit hashing)
- The original training time is about 200 seconds.
- b-bit minwise hashing needs about $3 \sim 7$ seconds (3 seconds when $b = 8$).

Testing Time

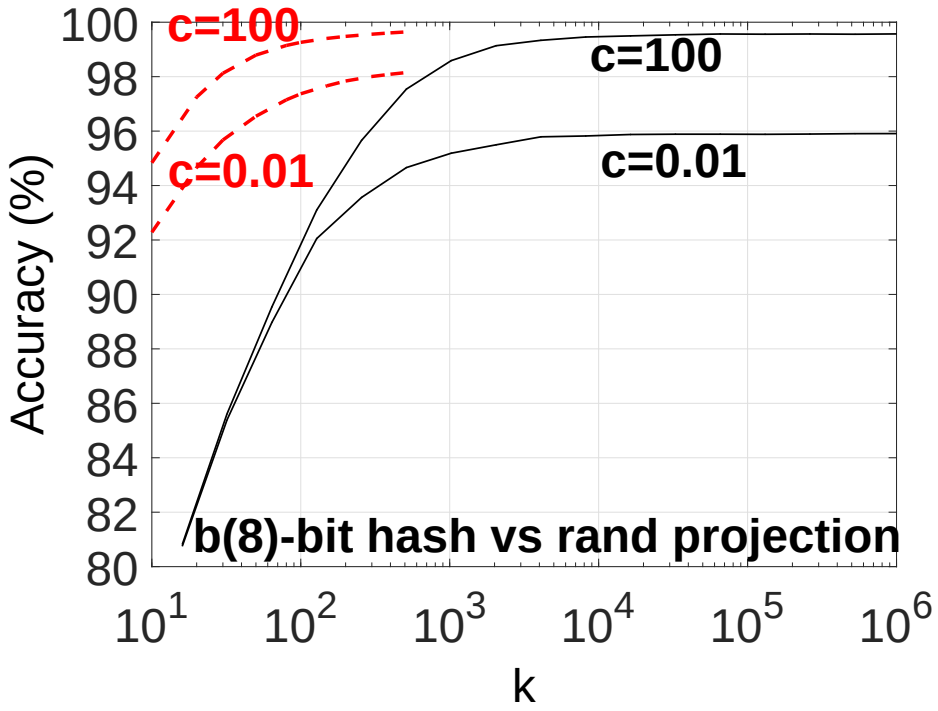


However, here we assume the test data have already been processed.

Comparison with Very Sparse Random Projections

8-bit minwise hashing (**dashed, red**): $k = 200$

Sparse random projections and variants: $k = 10^4 \sim 10^6$.



The Problem of Expensive Preprocessing

500 permutations (more are needed for near neighbor search) on the entire data can be very expensive. A serious issue when the new testing data have not been processed.

Two solutions:

1. **Parallel solution by GPUs**: Achieved up to 100-fold improvement in speed.

   Ref: Li, Shrivastava, König, GPU-Based Minwise Hashing, WWW'12 (poster)

2. **One Permutation Hashing**: (Recommended for learning and search)

   Ref: Li, Owen, Zhang, One Permutation Hashing, NIPS 2012

   Ref: Shrivastava and Li, Densified One Permutation Hashing ... , ICML 2014

Intuition: Minwise Hashing Ought to Be Wasteful

Original Data Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_1:	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0
S_2:	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0
S_3:	0	0	0	1	0	0	1	1	0	0	0	0	0	0	1	0

Permuted Data Matrix

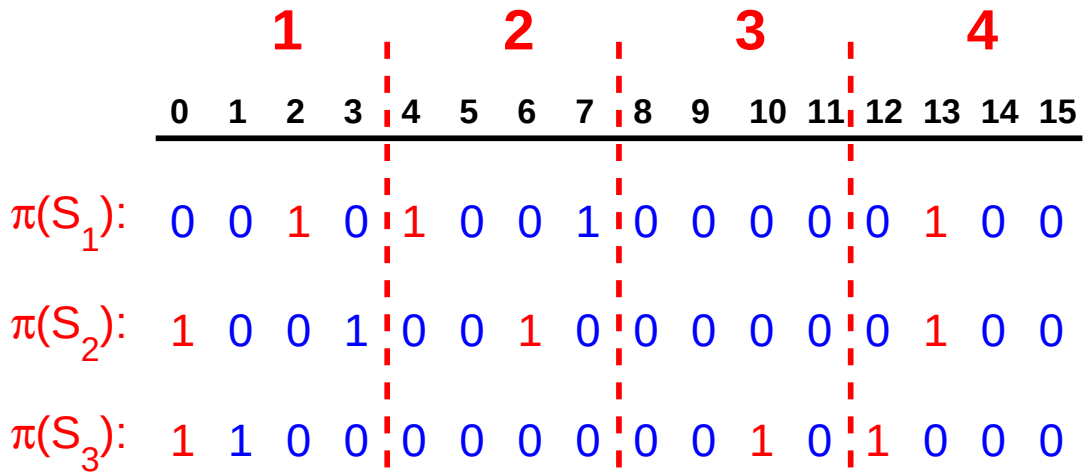
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(S_1)$:	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	0
$\pi(S_2)$:	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0
$\pi(S_3)$:	1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0

Only store the minimums and repeat the process k (e.g., 500) times.

One Permutation Hashing

$S_1, S_2, S_3 \subseteq \Omega = \{0, 1, ..., 15\}$ (i.e., $D = 16$). The figure presents the permuted sets as three binary (0/1) vectors:

$\pi(S_1) = \{2, 4, 7, 13\}, \quad \pi(S_2) = \{0, 6, 13\}, \quad \pi(S_3) = \{0, 1, 10, 12\}$



One permutation hashing: divide the space Ω evenly into $k = 4$ bins and select the smallest nonzero in each bin.

Ref: P. Li, A. Owen, C.-H. Zhang, One Permutation Hashing, NIPS 2012

Two major tasks:

- Theoretical analysis and estimators

- Practical strategies to deal with empty bins, should they occur. There are different strategies for different applications (learning or search).

Two Definitions

Recall: the space is divided evenly into k bins

 # Jointly empty bins: $$N_{emp} = \sum_{j=1}^{k} I_{emp,j}$$

 # Matched bins: $$N_{mat} = \sum_{j=1}^{k} I_{mat,j}$$

where $I_{emp,j}$ and $I_{mat,j}$ are defined for the j-th bin, as

$$I_{emp,j} = \begin{cases} 1 & \text{if both } \pi(S_1) \text{ and } \pi(S_2) \text{ are empty in the } j\text{-th bin} \\ 0 & \text{otherwise} \end{cases}$$

$$I_{mat,j} = \begin{cases} 1 & \text{if both } \pi(S_1) \text{ and } \pi(S_1) \text{ are not empty and the smallest element} \\ & \text{of } \pi(S_1) \text{ matches the smallest element of } \pi(S_2), \text{ in the } j\text{-th bin} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{emp} = \sum_{j=1}^{k} I_{emp,j}, \qquad N_{mat} = \sum_{j=1}^{k} I_{mat,j}$$

Results for the Number of Jointly Empty Bins N_{emp}

Notation:

$$f_1 = |S_1|, \quad f_2 = |S_2|, \quad a = |S_1 \cap S_2|, \quad f = |S_1 \cup S_2| = f_1 + f_2 - a.$$

Expectation:

$$\frac{E\left(N_{emp}\right)}{k} = \prod_{j=0}^{f-1} \frac{D\left(1-\frac{1}{k}\right)-j}{D-j} \leq \left(1-\frac{1}{k}\right)^f$$

Variance:

$$\frac{Var\left(N_{emp}\right)}{k^2} = \frac{1}{k}\left(\frac{E(N_{emp})}{k}\right)\left(1-\frac{E(N_{emp})}{k}\right)$$
$$-\left(1-\frac{1}{k}\right)\left(\left(\prod_{j=0}^{f-1}\frac{D\left(1-\frac{1}{k}\right)-j}{D-j}\right)^2 - \prod_{j=0}^{f-1}\frac{D\left(1-\frac{2}{k}\right)-j}{D-j}\right)$$

———-

Perhaps too complicated!

Impact of Empty Bins

Approximation in sparse data, i.e., $f = |S_1 \cup S_2| = f_1 + f_2 - a \ll D$.

Expectation: $$\frac{E\left(N_{emp}\right)}{k} = \left(1 - \frac{1}{k}\right)^f \left(1 - O\left(\frac{f^2}{kD}\right)\right)$$

Implication: $\frac{E(N_{emp})}{k} \approx \left(1 - \frac{1}{k}\right)^f \approx e^{-f/k}$.

$f/k = 5 \Longrightarrow \frac{E(N_{emp})}{k} \approx 0.0067$ (negligible).

$f/k = 1 \Longrightarrow \frac{E(N_{emp})}{k} \approx 0.3679$ (noticeable).

We must find a solution to deal with empty bins.

Probability distribution function of N_{emp}:

$$\mathbf{Pr}\left(N_{emp}=j\right)=\sum_{s=0}^{k-j}(-1)^{s}\frac{k!}{j!s!(k-j-s)!}\prod_{t=0}^{f-1}\frac{D\left(1-\frac{j+s}{k}\right)-t}{D-t}$$

in case some readers are curious.

An Unbiased Estimator

Estimator : $\hat{R}_{mat} = \frac{N_{mat}}{k - N_{emp}}$

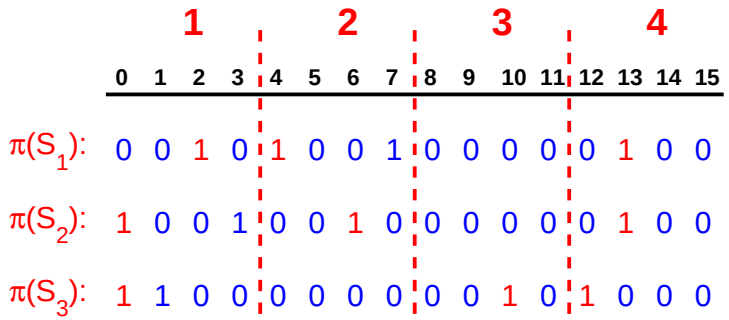
Expectation : $E\left(\hat{R}_{mat}\right) = R$ (slightly surprising)

Variance: $Var\left(\hat{R}_{mat}\right)$ (Recall $f = |S_1 \cup S_2|$)

$$= \frac{R(1-R)}{k}\left(E\left(\frac{1}{1 - N_{emp}/k}\right)\left(1 + \frac{1}{f-1}\right) - \frac{1}{f-1}\right)$$

$$< \frac{R(1-R)}{k}$$ (Variance of original minwise hashing)

Summary of Advantages of One Permutation Hashing



- Computationally much more efficient and energy-efficient.
- Testing unprocessed data is much faster, crucial for user-facing applications.
- Implementation is easier, from the perspective of random number generation, e.g., storing a permutation vector of length $D = 10^9$ (4GB) is not a big deal.

One Permutation Hashing for Learning

Almost the same as k-permutation hashing, but we must deal with empty bins *.

Zero coding: Encode empty bins as (e.g.,) 00000000 in the expanded space.

This simple strategy often works well for learning (but not for near neighbor search).

Zero Coding Example

One permutation hashing with $k = 4$ and $b = 2$ (* indicates empty bin)

For set (vector) S_1:

Original hashed values $(k = 4)$: $12013 \quad 25964 \quad 20191 \quad *$

Original binary representations :

$01011101110101 \quad 11001010110100 \quad 10011101101111 \quad *$

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

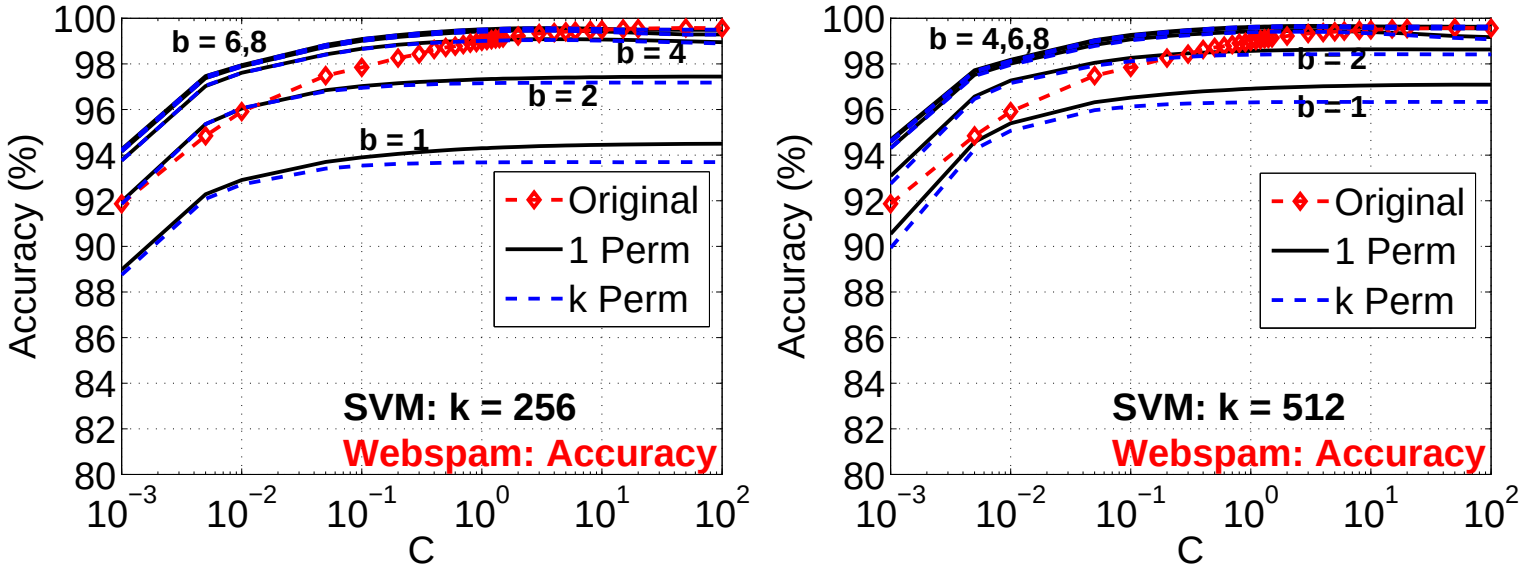
Corresponding decimal values : $1 \quad 0 \quad 3 \quad *$

Expanded $2^b = 4$ binary digits : $0100 \quad 1000 \quad 0001 \quad 0000$

New feature vector : $[0, 1, 0, 0, \ 1, 0, 0, 0, \ 0, 0, 0, 1, \ 0, 0, 0, 0] \times \frac{1}{\sqrt{4-1}}$

Same procedures on sets S_2, S_3, ...

Experimental Results on Webspam Data



One permutation hashing (zero coding) is even slightly more accurate than k-permutation hashing (at merely $1/k$ of the original cost).

Limitation of One Permutation Hashing

One permutation hashing can not be directly used for near neighbor search by building hash tables because **empty bins** do not offer indexing capability.

In other words, because of these empty bins, it is not possible to determine which bin value to use for bucketing.

General Hashing Table Scheme for Fast Near Search

Index		Data Points
00	00	***8***, *13, 251*
00	01	*5, 14, 19, 29*
00	10	*(empty)*
11	01	*7, 24, 156*
11	10	*33, 174, 3153*
11	11	*61, 342*

Index		Data Points
00	00	*2, 19, 83*
00	01	*17, 36, 129*
00	10	*4, 34, 52, 796*
11	01	*7, 198*
11	10	*56, 989*
11	11	***8*** *,9, 156, 879*

When a new data vector arrives, it is hashed to binary code and falls into one bucket in each table. The candidates for its near neighbors are the union of data points from the buckets.

If part of the code is missing (due to empty bins), then we must “guess” the missing part.

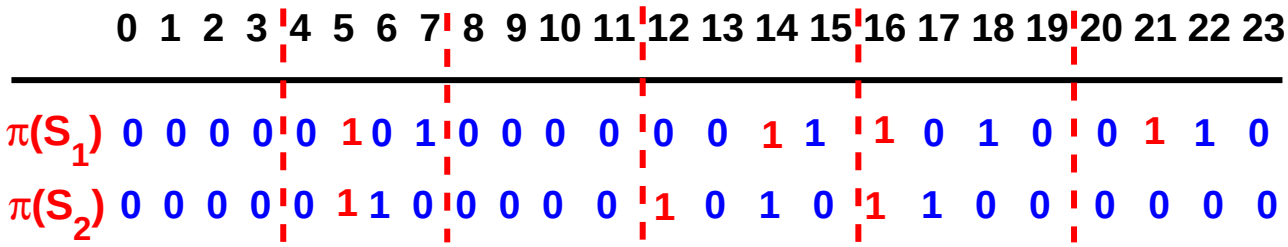
Neither Zero-coding nor Random-Coding Would Work

Zero-coding, or "empty-equal" (EE), scheme: If empty bins dominate, then two sparse vectors will become artificially "similar".

Random-coding, or "empty-not-equal" (ENE), scheme: By coding an empty bin randomly, again if empty bins dominate, then two sparse vectors which are similar in terms of the original resemblance may artificially become not so similar.

One Proposal: Borrowing from Neighboring Non-Empty Bins

The original one permutation hashing (**OPH**) is densified to become **H**:



by borrowing hashed values from neighboring non-empty bins.

OPH($\pi(S_1)$) = [E, 1, E, 2, 0, 1] → H($\pi(S_1)$) = [1+C, 1, 2+C, 2, 0, 1]

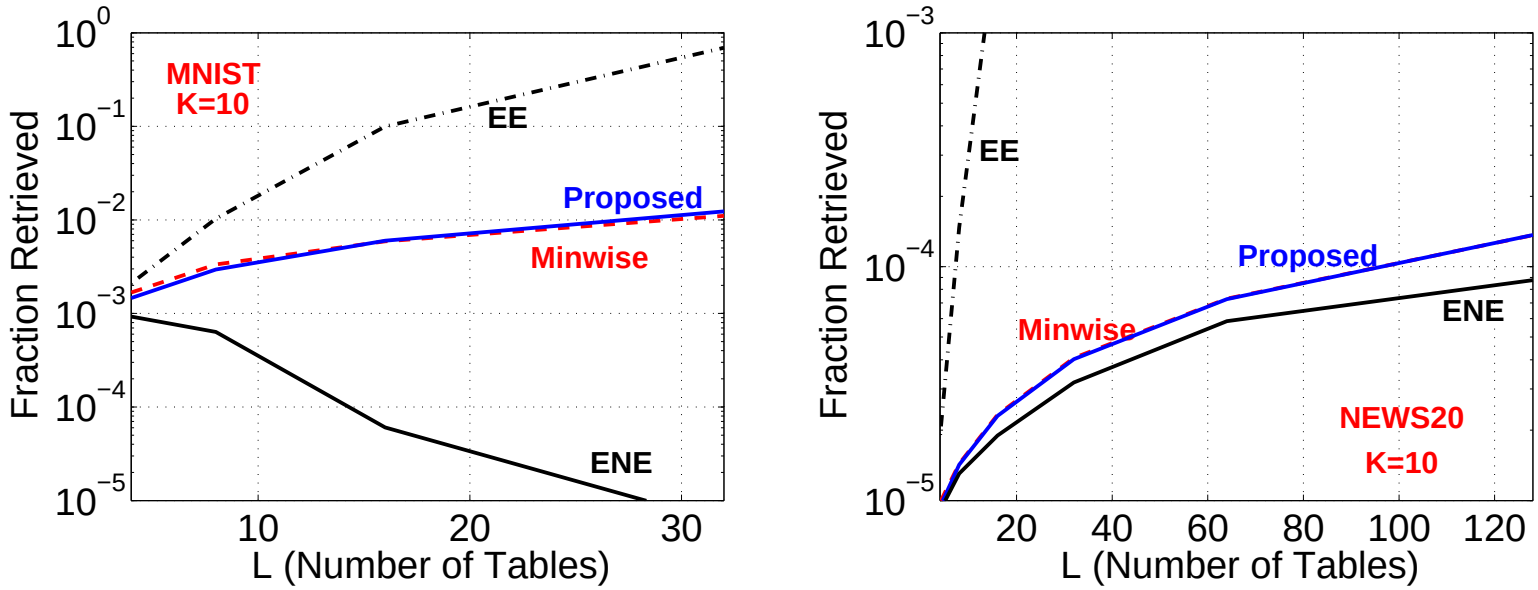
OPH($\pi(S_2)$) = [E, 1, E, 0, 0, E] → H($\pi(S_2)$) = [1+C, 1, C, 0, 0, 1+2C]

$C \geq D/k + 1$ is a constant for avoiding undesired collision.

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

Ref: Shrivastava and Li, Densifying One Permutation Hashing ..., ICML'14

Experiments for Near Neighbor Search



1-permutation with densification ("Proposed") matches the original minwise hashing.

Similarity for Non-Binary Data

For two n-dim vectors u and v, we can define their correlation (cosine) similarity:

Linear (Cosine) $$\rho = \rho(u, v) = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \sqrt{\sum_{i=1}^{n} v_i^2}}$$

and the radial basis function (RBF) kernel (where $\gamma > 0$ is a crucial tuning parameter):

RBF $$RBF(u, v; \gamma) = e^{-\gamma(1-\rho)}$$

For small datasets, the RBF kernel can be very useful as it often achieves good accuracies.

For example, we can directly use the all pairwise kernel matrix if it is small enough.

The Generalized Min-Max (GMM) Kernel

Data Transformation: Consider the original data vector u_i, $i = 1$ to n. We define the following transformation, depending on whether an entry u_i is positive or negative:

$$\begin{cases} \tilde{u}_{2i-1} = u_i, \quad \tilde{u}_{2i} = 0 & \text{if } u_i > 0 \\ \tilde{u}_{2i-1} = 0, \quad \tilde{u}_{2i} = -u_i & \text{if } u_i \leq 0 \end{cases}$$

For example, $n = 2$ and $u = [-5 \;\; 3]$, the transformed data vector becomes $\tilde{u} = [0 \;\; 5 \;\; 3 \;\; 0]$.

Generalized Min-Max (GMM) Kernel:

$$GMM(u, v) = \frac{\sum_{i=1}^{2n} \min(\tilde{u}_i, \; \tilde{v}_i)}{\sum_{i=1}^{2n} \max(\tilde{u}_i, \; \tilde{v}_i)}$$

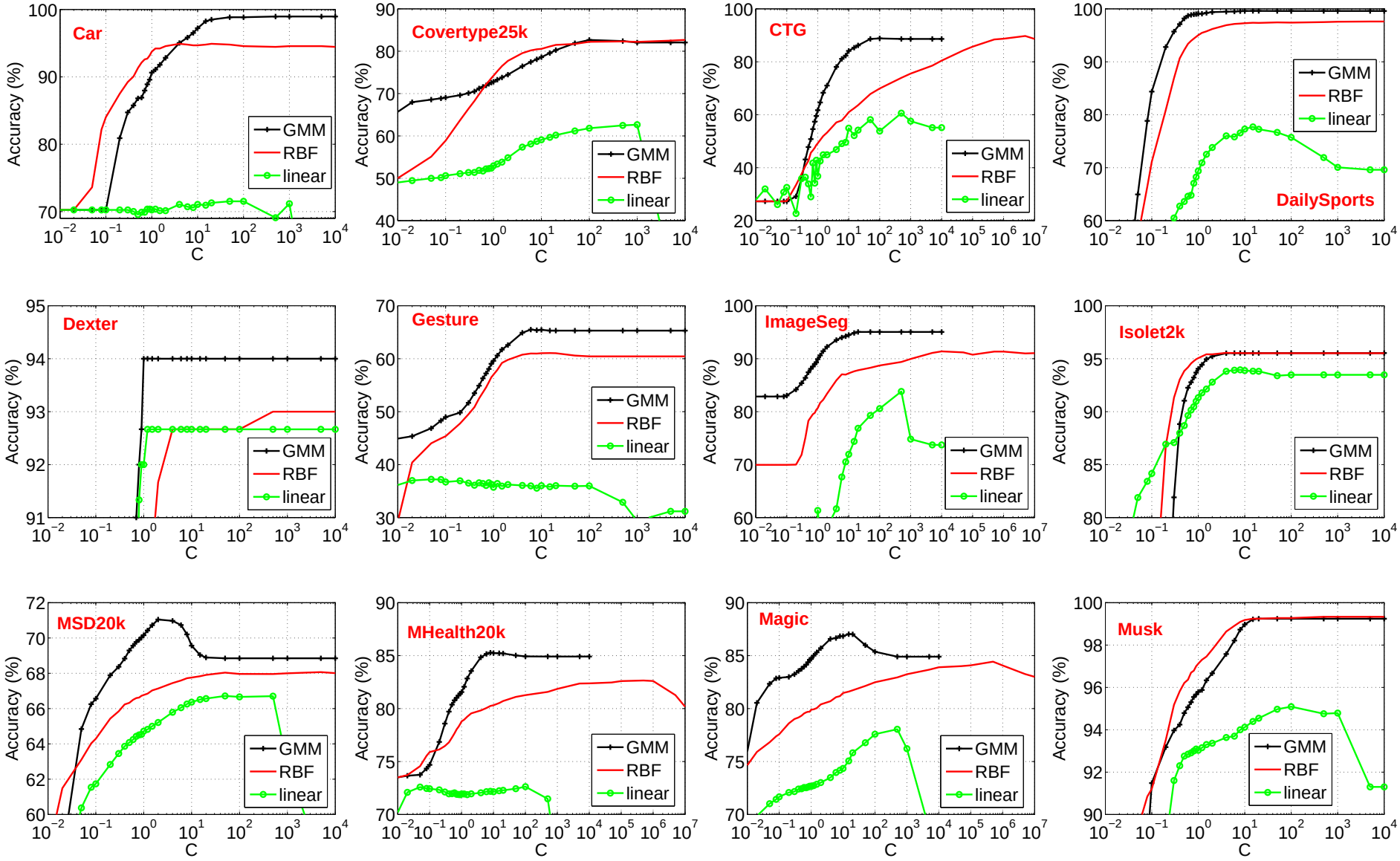
GMM is an effective measure of data similarity, as shown through an extensive experimental study on kernel SVM classification.

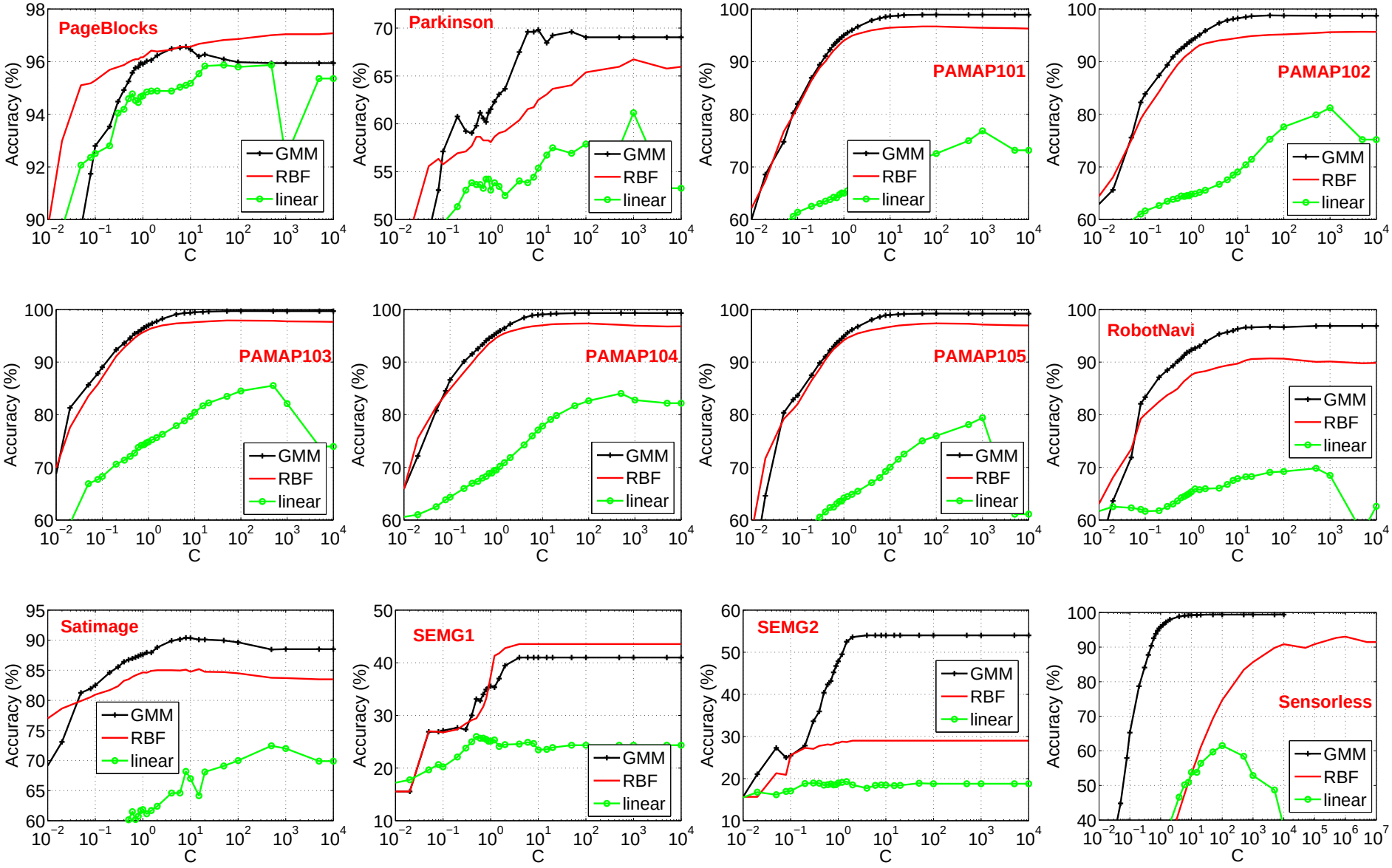
An Experimental Study on Kernel SVMs and Public Datasets

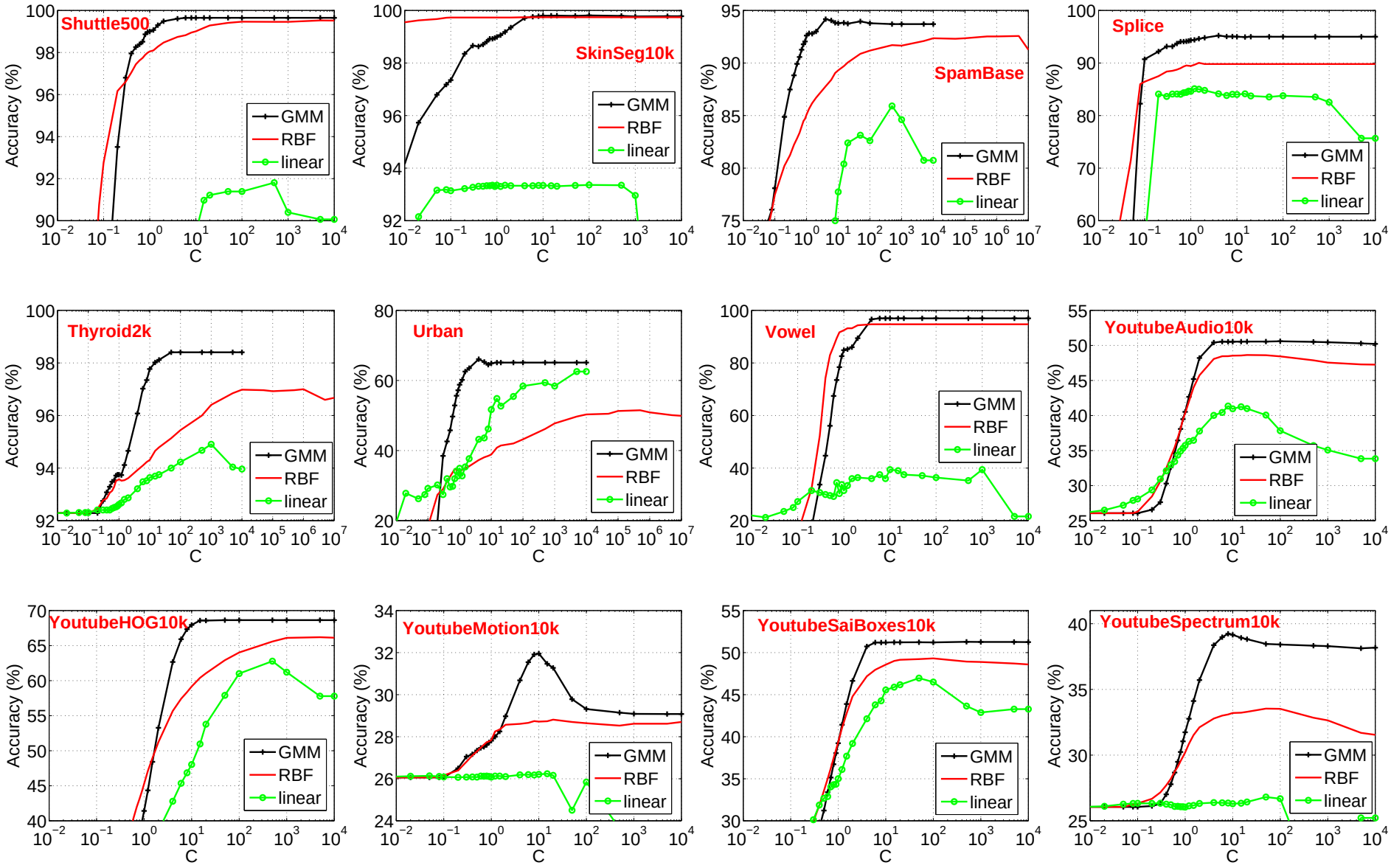
We report test classification accuracies for the linear SVM, kernel SVM with RBF, and kernel SVM with GMM, at the best l_2-regularization C values.

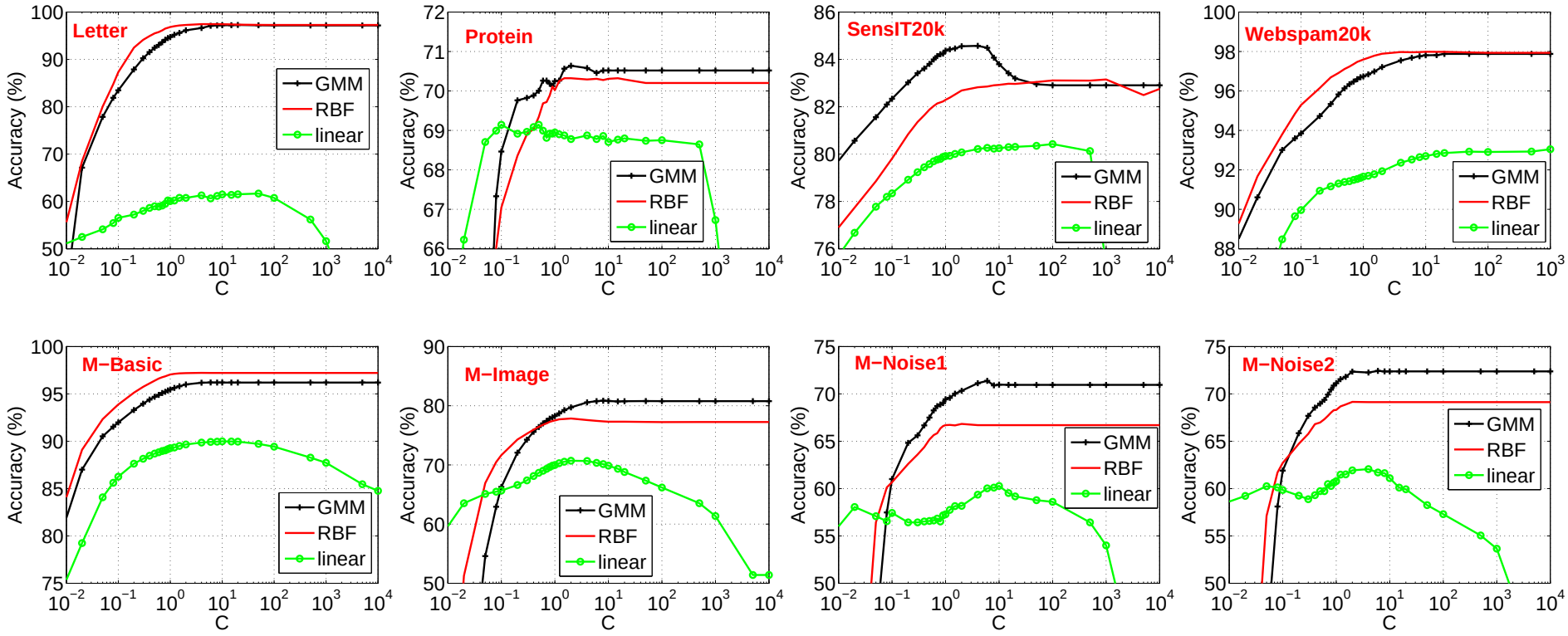
For the RBF kernel, we exhaustively experimented with 58 different values of $\gamma \in \{0.001,$ 0.01, 0.1:0.1:2, 2.5, 3:1:20 25:5:50, 60:10:100, 120, 150, 200, 300, 500, 1000$\}$.

Test classification accuracies using kernel SVMs




PageBlocks
Parkinson
PAMAP101
PAMAP102
PAMAP103
PAMAP104
PAMAP105
RobotNavi
Satimage
SEMG1
SEMG2
Sensorless
GMM
RBF
linear
Accuracy (%)
C


Shuttle500
SkinSeg10k
SpamBase
Splice
Thyroid2k
Urban
Vowel
YoutubeAudio10k
YoutubeHOG10k
YoutubeMotion10k
YoutubeSaiBoxes10k
YoutubeSpectrum10k
GMM
RBF
linear
Accuracy (%)
C



Typically, both GMM and RBF kernels substantially improve linear SVM. C is the l_2-regularization parameter of SVM. For the RBF kernel, we report the result at the best γ value for every C value.

For most datasets, the tuning-free GMM kernel outperforms the best-tuned RBF kernel.

Theory of the GMM Kernel

Given n-dimensional data vectors: x and y, we first define $x_i = x_{i+} - x_{i-}$, where

$$x_{i+} = \begin{cases} x_i & \text{if } x_i \geq 0 \\ 0 & \text{otherwise} \end{cases}, \quad x_{i-} = \begin{cases} -x_i & \text{if } x_i < 0, \\ 0 & \text{otherwise} \end{cases}$$

Then we compute GMM as follows:

$$GMM(x,y) = \frac{\sum_{i=1}^{n} \left[\min(x_{i+}, y_{i+}) + \min(x_{i-}, y_{i-})\right]}{\sum_{i=1}^{n} \left[\max(x_{i+}, y_{i+}) + \max(x_{i-}, y_{i-})\right]} \triangleq g_n(x,y)$$

Two fundamental questions, as $n \to \infty$ (i.e., as dimensionality is large),

1. What is the limit of $g_n(x,y)$?
2. How fast does it converge to the limit?

Why do we care?

The Cosine Similarity

In web search and machine learning, arguably the most commonly adopted measure of similarity might be the "cosine":

$$Cos(x,y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i^2}} \stackrel{\triangle}{=} c_n(x,y)$$

The popularity of cosine can be explained by the fact that, if the entries of the data vectors are independent normally distributed, then $c_n(x,y)$ converges to the data population correlation. More precisely, if each coordinate (x_i, y_i) is an iid copy of (X,Y), where

$$(X,Y)^{\mathsf{T}} \sim N\left(0, \Sigma = \begin{pmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{pmatrix}\right)$$

then as $n \to \infty$, $c_n(x,y)$ converges in distribution to a normal

$$n^{1/2}\left(c_n - \rho\right) \stackrel{D}{\Longrightarrow} N\left(0, (1-\rho^2)^2\right)$$

Normality Assumption is Crucial for Cosine

Theorem: Consider n iid samples $(x_i, y_i) \sim t_{\Sigma,\nu}$, i.e., a bivariate t-distribution with covariance matrix Σ and ν degrees of freedom, where $\nu > 4$ and $\Sigma = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$, $-1 \leq \rho \leq 1$. As $n \to \infty$, $c_n(x, y) = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2 \sum_{i=1}^n y_i^2}}$ converges in distribution to a normal:

$$n^{1/2}(c_n - \rho) \overset{D}{\Longrightarrow} N\left(0, \frac{\nu - 2}{\nu - 4}(1 - \rho^2)^2\right)$$

In other words, it is only safe to use cosine if the data are very close to normal (large ν). If c_n does not converge, then it is not a meaningful measure. If c_n converges but the rate is slow and the variance is large (or does not exist), then it is not reliable to use c_n.

GMM Under t-Distribution

Consider n iid samples $(x_i, y_i) \sim t_{\Sigma,\nu}$, where $\nu > 2$ and $\Sigma = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$, $-1 \leq \rho \leq 1$.

As $n \to \infty$,

$$g_n \longrightarrow \frac{1 - \sqrt{(1-\rho)/2}}{1 + \sqrt{(1-\rho)/2}}$$

If the degree of freedom $\nu \to \infty$, then $t_{\Sigma,\nu}$ converges to normal with covariance Σ.

GMM Under t-Distribution

Theorem: Consider n iid samples $(x_i, y_i) \sim t_{\Sigma,\nu}$, where $\nu > 2$ and $\Sigma = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$, $-1 \leq \rho \leq 1$. As $n \to \infty$, then the GMM kernel $g_n(x, y)$ converges in distribution to a normal:

$$n^{1/2}\left(g_n - \frac{1-\sqrt{(1-\rho)/2}}{1+\sqrt{(1-\rho)/2}}\right) \overset{D}{\Longrightarrow} N\left(0, \frac{V}{H^4}\frac{8}{\pi}\frac{1}{\nu-2}\frac{\Gamma^2(\nu/2)}{\Gamma^2(\nu/2-1/2)}\right)$$

where $\Gamma(.)$ is gamma function, $H = \frac{2}{\pi}(1+\sin\alpha)$, $\alpha = \sin^{-1}\left(\sqrt{1/2-\rho/2}\right)$, and $V = \frac{4}{\pi^3}\sin^2\alpha \times (3\pi - 8\cos\alpha + 2\sin 2\alpha + \pi\cos 2\alpha - 8\alpha\sin\alpha - 4\alpha\cos 2\alpha)$.

---

g_n converges as long as $\nu \geq 1$. The variance is bounded if $\nu > 2$.

Ref: P. Li and C.-H. Zhang Theory of the GMM Kernel, WWW 2017

Estimator of ρ from GMM

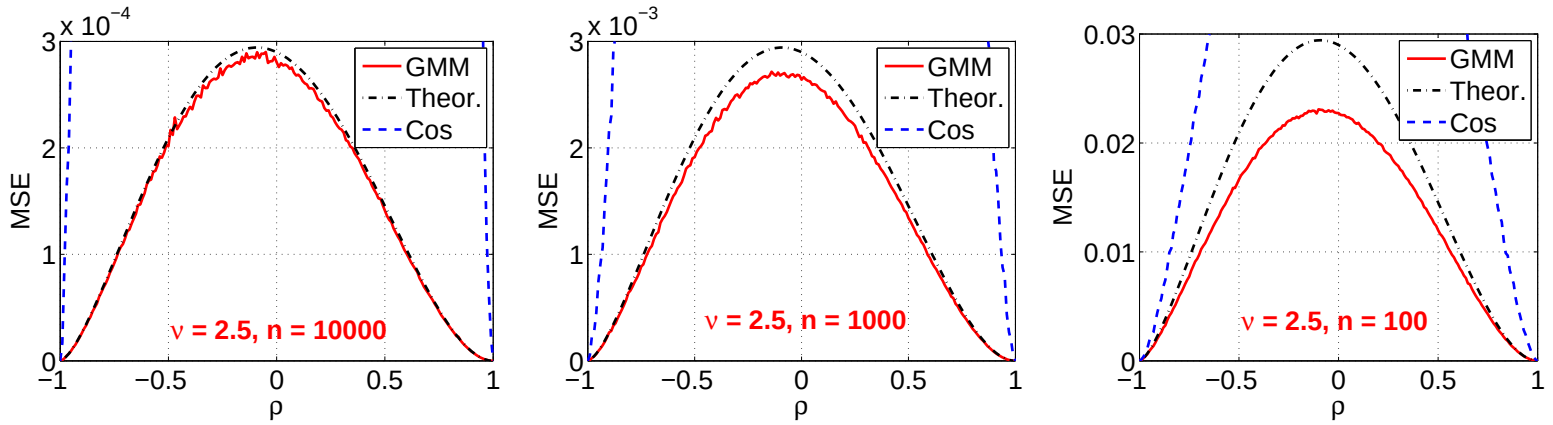
$g_n(x,y) \to \frac{1-\sqrt{(1-\rho)/2}}{1+\sqrt{(1-\rho)/2}}$, which provides a robust and convenient way to estimate the similarity between data vectors:

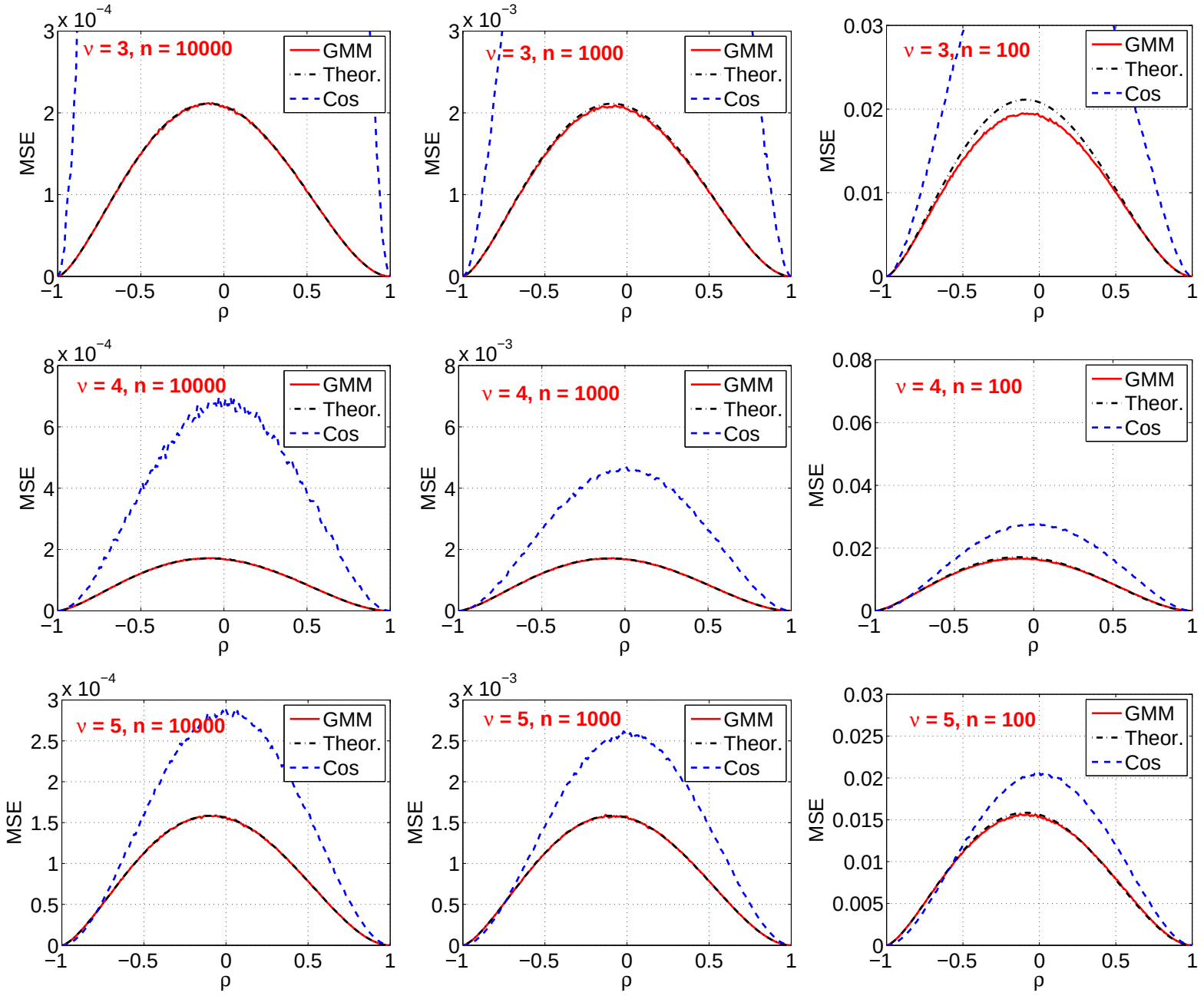
$$\hat{\rho}_g = 1 - 2\left(\frac{1-g_n}{1+g_n}\right)^2$$

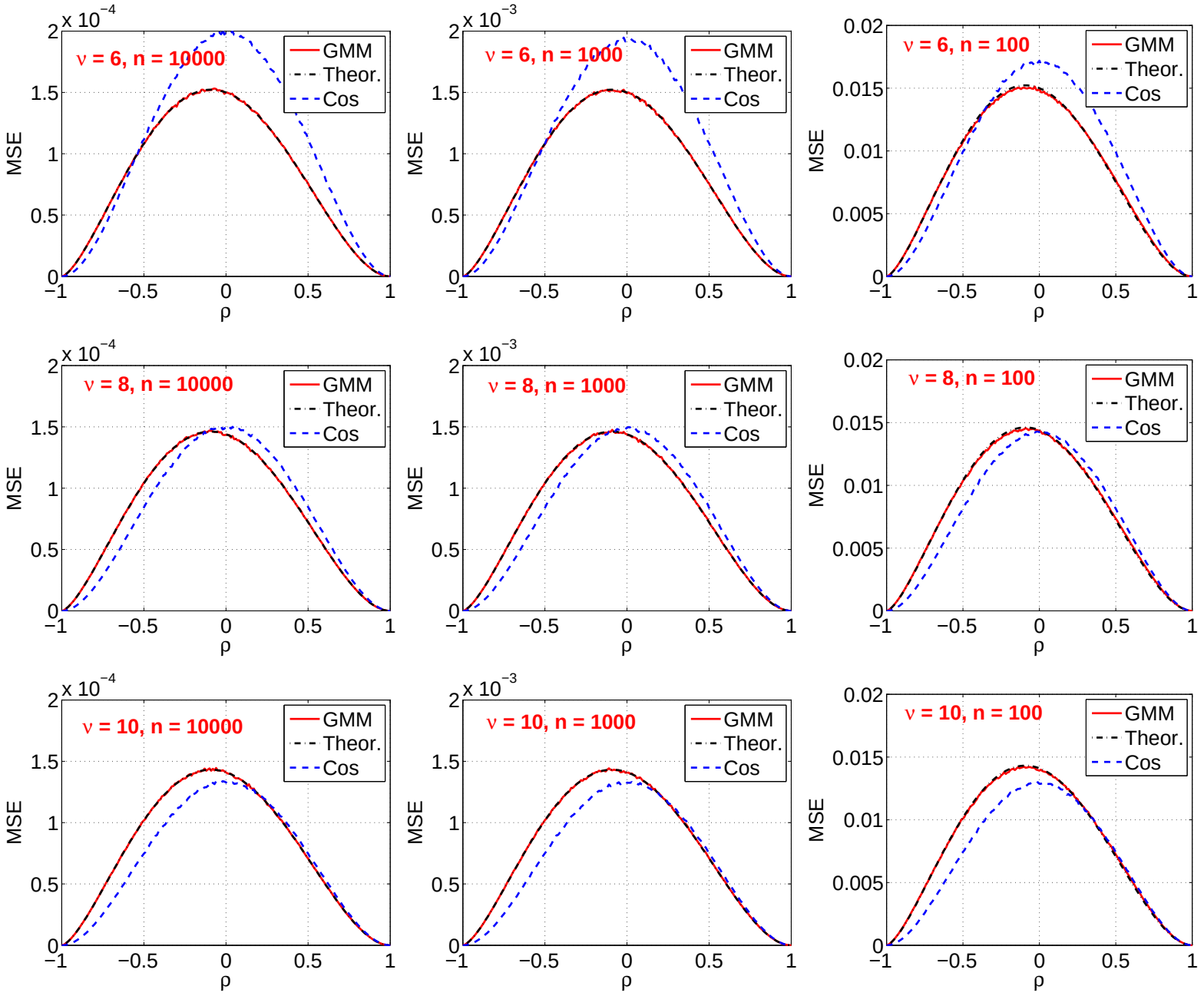
The variance of $\hat{\rho}_g$ can be precisely evaluated from results in previous pages.

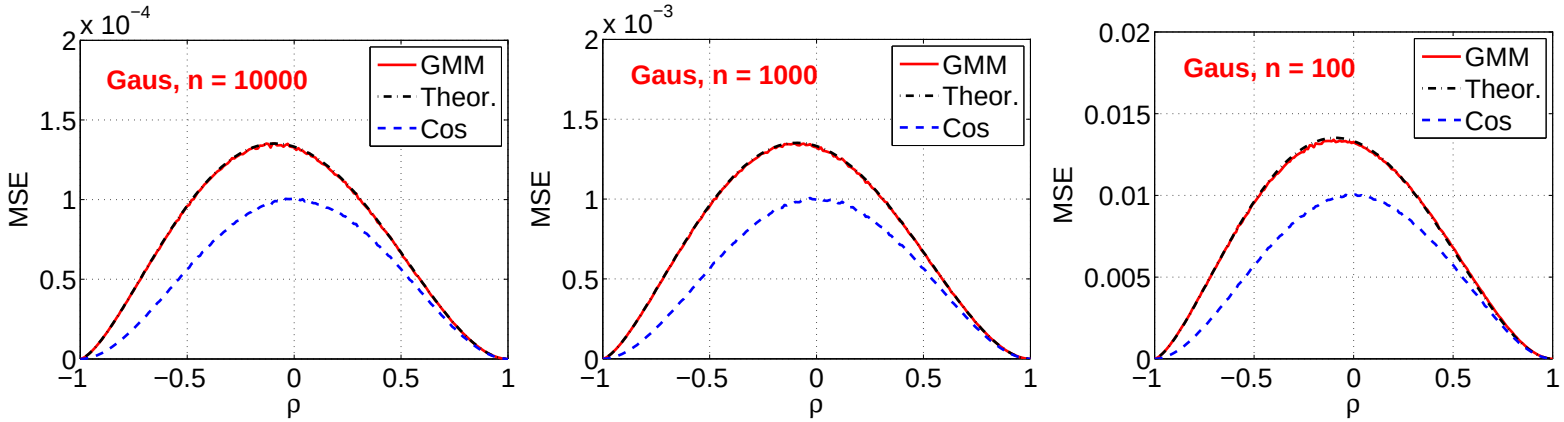
———-

We visualize variances for comparing the two estimators: $\hat{\rho}_g$ and $\hat{\rho}_c$ (using cosine).




ν = 3, n = 10000
ν = 3, n = 1000
ν = 3, n = 100
ν = 4, n = 10000
ν = 4, n = 1000
ν = 4, n = 100
ν = 5, n = 10000
ν = 5, n = 1000
ν = 5, n = 100
GMM
Theor.
Cos
MSE
ρ
x 10^-4
x 10^-3


ν = 6, n = 10000
ν = 6, n = 1000
ν = 6, n = 100
ν = 8, n = 10000
ν = 8, n = 1000
ν = 8, n = 100
ν = 10, n = 10000
ν = 10, n = 1000
ν = 10, n = 100
GMM
Theor.
Cos
MSE
ρ
x 10^−4
x 10^−3



$\hat{\rho}_g$ can be substantially more accurate than $\hat{\rho}_c$. The theoretical asymptotic variance formula, despite the complexity of its expression, is accurate when ν is not too close to 2.

Roughly speaking, when $\nu < 8$, it is preferable to use $\hat{\rho}_g$, the estimator based on GMM. Even when data are perfectly Gaussian, using $\hat{\rho}_g$ does not result in too much loss of accuracy.

GCWS for Hashing the GMM Kernel

The "generalized consistent weighted sampling" (GCWS) provides an effective way to linearize the nonlinear GMM kernel.

Input: Data vector x_i ($i = 1$ to n)

Generate vector $\tilde{x}$ in $2n$-dim by transforming original data into nonnegative data

For i from 1 to $2n$

$r_i \sim Gamma(2,1), c_i \sim Gamma(2,1), \beta_i \sim Uniform(0,1)$

$t_i \leftarrow \lfloor \frac{\log \tilde{x}_i}{r_i} + \beta_i \rfloor, \quad a_i \leftarrow \log(c_i) - r_i(t_i + 1 - \beta_i)$

End For

Output: $i^* \leftarrow arg\min_i \ a_i, \qquad t^* \leftarrow t_{i^*}$

CWS originates from a series of prior studies by Manasse, Ioffe, among others.

Given two data vectors x and y, one can transform them into nonnegative vectors $\tilde{x}$ and $\tilde{y}$ and generate random tuples:

$$\left(i^*_{\tilde{x},j}, t^*_{\tilde{x},j}\right) \text{ and } \left(i^*_{\tilde{y},j}, t^*_{\tilde{y},j}\right), \ j = 1, 2, ..., k$$

The basic probability results of CWS says

$$\mathbf{Pr}\left\{\left(i^*_{\tilde{x},j}, t^*_{\tilde{x},j}\right) = \left(i^*_{\tilde{y},j}, t^*_{\tilde{y},j}\right)\right\} = GMM(x, y)$$

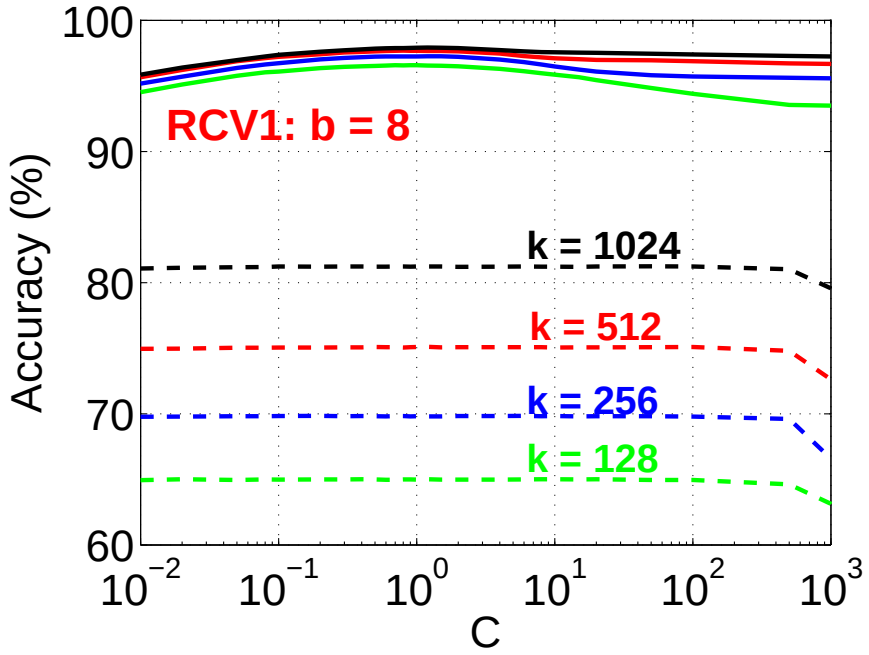
Just like minwise hashing, this provides a strategy to use GCWS and linear algorithms for approximating computing the GMM kernel SVMs.

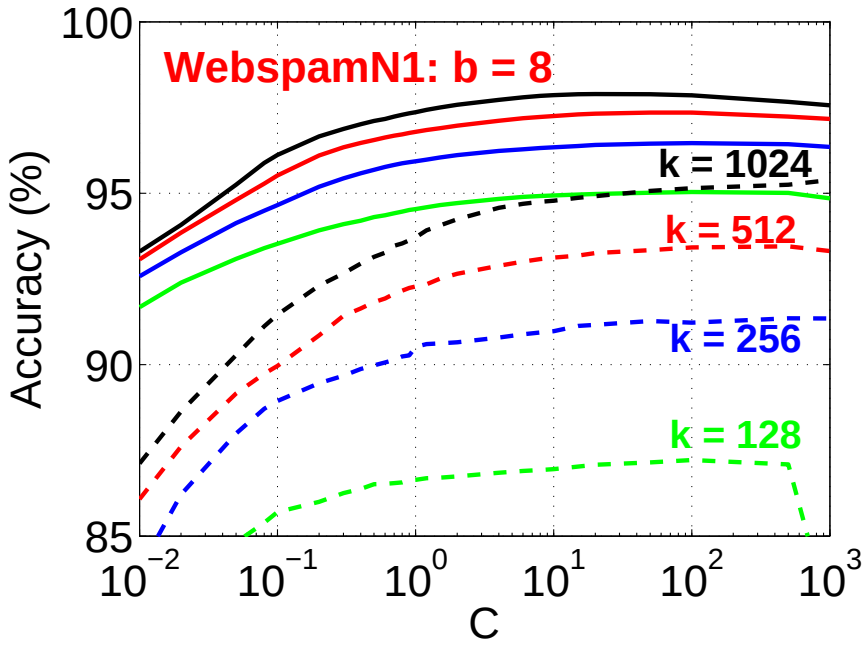
Hashing + Linear SVM Results on Two Classification Tasks

Solid curves: classification accuracies using GCWS + linear SVM

Dashed curves: classification accuracies using RFF + linear SVM

RFF = Random Fourier Features





GCWS hashing is substantially more accurate than NRFF hashing at the same sample size k.

A lot of details are skipped.

Conclusion

1. Minwse hashing is a well-known effective hashing strategy for high-dim binary (0/1) data
2. One permutation hashing (and variants) solved the computational bottleneck of minhash
3. One can use one permutation hashing for both search and machine learning
4. For non-binary data, the GMM kernel is an effective similarity measure and GCWS hashing is a promising algorithm for large-scale computations with the GMM kernel.