Electronic Preprint for Journal of Information Processing Vol.26

Regular Paper

Compressed Vector Set: A Fast and Space-Efficient Data Mining Framework

Masafumi Oyamada[1,2,a)] Jianquan Liu[1] Shinji Ito[1] Kazuyo Narita[1]
Takuya Araki[1] Hiroyuki Kitagawa[2]

Received: September 10, 2017, Accepted: December 9, 2017

Abstract: In this paper, we present CVS (Compressed Vector Set), a fast and space-efficient data mining framework that efficiently handles both sparse and dense datasets. CVS holds a set of vectors in a compressed format and conducts primitive vector operations, such as ℓ_p-norm and dot product, without decompression. By combining these primitive operations, CVS accelerates prominent data mining or machine learning algorithms including k-nearest neighbor algorithm, stochastic gradient descent algorithm on logistic regression, and kernel methods. In contrast to the commonly used sparse matrix/vector representation, which is not effective for dense datasets, CVS efficiently handles sparse datasets and dense datasets in a unified manner. Our experimental results demonstrate that CVS can process both dense datasets and sparse datasets faster than conventional sparse vector representation with smaller memory usage.

Keywords: data compression, run-length encoding, nearest neighbor search, machine learning

1. Introduction

Edge devices, such as sensors and mobile devices, are prevalent in our life. To conduct data mining tasks on edge devices in a real-time manner, it is necessary to conduct the tasks in the edge devices themselves, instead of delegating the tasks to the powerful remote computers. However, since edge devices have relatively small memory footprints and low computational power, the applicability of edge devices to the data mining tasks is limited. Motivated by this problem, several studies have tried to reduce the memory usage and the computational time for specific tasks or specific data types [9], [10], [12].

For sparse matrix data that is mostly filled with zero values, *sparse matrix representation* [9], [10], which only holds non-zero value and its position, can concisely represent the matrix and reduce the cost of some mathematical operations. However, actual data is not always sparse but can be dense. Since storing dense data with a sparse matrix format can increase the size, sparse matrix representation cannot be used in a versatile way.

In this paper, we present *CVS* (*Compressed Vector Set*), a general framework for fast and space-efficient data mining, which successfully supports both sparse and dense datasets. CVS compresses vector sets by *run-length encoding* and conducts fundamental mathematical operations on them without decompression. By combining fundamental mathematical operations, CVS runs advanced data mining algorithms such as k-nearest neighbor search, stochastic gradient descent on logistic regression, and kernel methods. To reduce the size of compressed vectors as much as possible, CVS (1) reorders the dimensions of vectors if the mathematical operations are dimension-order insensitive, and (2) discretizes vectors if the result of data mining algorithms is less affected by the precision of values in vectors.

We summarize our contributions:

(1) We present algorithms to conduct advanced data mining tasks on vectors compressed by run-length encoding (Section 3 and Section 4).
(2) We observe that reordering dimensions of the vectors further reduce the compression size without changing computational results. Based on the observation, we tackle the dimension-order reordering problem that finds best reordering pattern, which is NP-hard. We show a polynomial time algorithm for finding a dimension-reordering pattern that empirically yields good compression rate (Section 5).
(3) We demonstrate the combination of data discretization and dimension-reordering can drastically improve the performance without much affecting the accuracy of data mining tasks (Section 6).
(4) We demonstrate the effectiveness of CVS compared to the conventional sparse vector representation in data mining and machine learning tasks on real datasets (Section 7).

The rest of this paper is organized as follows. Section 3 describes how CVS compresses and conduct mathematical operations on a set of vectors. Section 3 shows how CVS conducts

[1] NEC Corporation, Kawasaki, Kanagawa 211–8666, Japan
[2] University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[a)] m-oyamada@cq.jp.nec.com

This paper is based on our previous conference paper [19]. We drastically rewrote the content to improve the generality, theoretical soundness, and presentation. We also made several extensions: (1) advanced data mining algorithms on CVS including k-nearest neighbor algorithm, stochastic gradient descent on logistic regression, and kernel methods, (2) a lossy-compression technique with data discretization, (3) additional experiments including comparison against sparse vector representation and performance demonstration of data discretization, and (4) literature reviews of related technologies including deep neural network compression and bitmap-index reordering.

© 2018 Information Processing Society of Japan

concrete data mining and machine learning tasks on compressed vectors without decompression. Section 5 describes how CVS improves the compression rate with dimension-reordering. Section 5 describes how CVS improves the compression rate with data discretization. Section 7 evaluates CVS by experiments with various datasets. Section 8 describes related work. Section 9 concludes the paper.

2. Preliminaries

In this section, we review concepts that relate to Compressed Vector Set (CVS): sparse vector representation and run-length encoding. List of main symbols are summarized in **Table 1**.

2.1 Sparse Vector Representation

Sparse matrix representations [9], [10] represent a vector that is mostly filled with zero values in a space-efficient way. One of the most commonly-used representations is to represent a vector with non-zero values and its positions. For example, a sparse vector

$$\boldsymbol{x} = (0, 0, 42, 99, 0, 0, 0)$$

is represented by two vectors

$$\text{values} = (42, 99)$$
$$\text{positions} = (2, 3)$$

in this format.

While the sparse vector representation reduces the space-efficiency, they also reduce the computational complexity of several mathematical operations. For example, the dot product of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, which are comprised of B_x and B_y non-zero elements, can be carried out in $O(B_x + B_y)$ time.

One downside of the sparse vector representation is its inefficiency for dense data. For example, a dense vector

$$\boldsymbol{x} = (7, 7, 42, 99, 7, 7, 7)$$

is represented by

$$\text{values} = (7, 7, 42, 99, 7, 7, 7)$$
$$\text{positions} = (0, 1, 2, 3, 4, 5, 6)$$

in the sparse vector representation, which doubled the size compared to the original vector. In contrast, our proposed CVS can represent a dense data efficiently when the number of distinct elements is not so large. We compare the performance of sparse vector representation and CVS empirically in Section 7.

2.2 Run-length Encoding

Run-length encoding (*RLE*) [2] is a data compression technique that represents a sequence of n consecutive same values $x, \ldots, x$ with a block $\langle n, x\rangle$. For example, RLE represents $(1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1)$ as $(\langle 5, 1\rangle, \langle 9, 2\rangle, \langle 3, 1\rangle)$, reducing the number of elements to represent the data from 17 to 6. In this paper, we denote the RLE-compressed form of the vector $\boldsymbol{x}$ as

$$\text{RLE}(\boldsymbol{x}) = (\langle n_1, x_1\rangle, \ldots, \langle n_B, x_B\rangle),$$

Table 1 List of symbols.

Symbol	Description
$\boldsymbol{x}$	A vector ($\boldsymbol{x} \in \mathbb{R}^D$)
$\mathcal{X}$	A vector set ($\mathcal{X} = \{ \boldsymbol{x}_1, \ldots, \boldsymbol{x}_J \}$)
J	Number of vectors in a vector set
X	Vector set as a matrix ($X \in \mathbb{R}^{D \times J}$)
D	Dimension of a vector
RLE($\boldsymbol{x}$)	A compressed form of a vector $\boldsymbol{x}$
$\langle n, x\rangle$	A block in a compressed vector (n-consecutive x)
B	Number of blocks in a compressed vector

where B is the total number of blocks and n_b is the number of values in the b-th block. We also use $|\text{RLE}(\boldsymbol{x})|$ to represent the total number of blocks of vector $\boldsymbol{x}$.

When the input vector $\boldsymbol{x}$ does not contain many consecutive same values, RLE cannot efficiently encode the data and sometimes increase the size; RLE represents $(1, 2, 3, 4, 1, 1, 1, 1)$ as $(\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 4, 1\rangle)$, which increased the data size. *PackBits* [2] is a simple but effective method to alleviate this problem, which has two encoding rules: (1) ordinary RLE encoding and (2) raw data encoding. (1) When PackBits finds a sequence of n consecutive same values $x, \ldots, x$, it does RLE encoding: replaces the sequence with a block $\langle n, x\rangle$. (2) Otherwise, PackBits is facing a sequence of n consecutive different values $x_1, \ldots, x_n$, and PackBits does raw data encoding: replaces the sequence by $\langle -n, x_1, \ldots, x_n\rangle$. For example, PackBits encodes a data sequence $(1, 2, 3, 4, 1, 1, 1, 1)$ to $(\langle -4, 1, 2, 3, 4\rangle, \langle 4, 1\rangle)$ successfully reducing the data size, whereas RLE encodes the sequence to $(\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 4, 1\rangle)$ increasing the data size.

3. Compressed Vector Set

In this section, we describe the idea of Compressed Vector Set (CVS), which compresses a vector set with run-length encoding to save the storage space, and conducts mathematical operations on the set of vectors without decompression reducing the computational time.

3.1 Vector Compression

Given a set of vectors $\mathcal{X} = \{ \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4, \boldsymbol{x}_5 \}$, where

$$\boldsymbol{x}_1 = (1, 1, 2, 2, 2),$$
$$\boldsymbol{x}_2 = (2, 2, 1, 2, 2),$$
$$\boldsymbol{x}_3 = (2, 3, 3, 3, 3),$$
$$\boldsymbol{x}_4 = (2, 3, 3, 2, 2),$$
$$\boldsymbol{x}_5 = (2, 3, 1, 2, 2),$$

CVS compresses each vector in $\mathcal{X}$ using RLE/PackBits:

$$\text{RLE}(\boldsymbol{x}_1) = (\langle 2, 1\rangle, \langle 3, 2\rangle),$$
$$\text{RLE}(\boldsymbol{x}_2) = (\langle 2, 2\rangle, \langle 1, 1\rangle, \langle 2, 2\rangle),$$
$$\text{RLE}(\boldsymbol{x}_3) = (\langle 1, 2\rangle, \langle 4, 3\rangle),$$
$$\text{RLE}(\boldsymbol{x}_4) = (\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 2, 2\rangle),$$
$$\text{RLE}(\boldsymbol{x}_5) = (\langle -3, 2, 3, 1\rangle, \langle 2, 2\rangle).$$

3.2 Operation without Decompression

CVS conducts mathematical operations on compressed vectors

© 2018 Information Processing Society of Japan

directly without decompressing the vectors, reducing both the computational time and the memory usage. The key insight under the technique is that an RLE-encoded vector contains preliminary knowledge that *value x appears n times consecutively*. In the rest of this section, we demonstrate how CVS utilizes this knowledge to perform actual mathematical operations without decompression.

3.2.1 ℓ_p-norm

First, we demonstrate how CVS computes ℓ_p-norm of a compressed vector. ℓ_p-norm is an essential statistics, which appears in many complex operations including a cosine similarity and mathematical optimization algorithms [7].

Consider ℓ_2-norm (Euculidian-norm) of a vector

$$\boldsymbol{x} = (1, 2, 2, 2, 2).$$

Naïvely, ℓ_2-norm of $\boldsymbol{x}$ is computed as

$$\|\boldsymbol{x}\|_2 = \sqrt{1^2 + 2^2 + 2^2 + 2^2 + 2^2}, \tag{1}$$

which requires five multiplications and four additions inside the square root.

In CVS, we have

$$\mathrm{RLE}(\boldsymbol{x}) = (\langle 1, 1\rangle, \langle 4, 2\rangle),$$

and ℓ_2-norm of $\boldsymbol{x}$ is computed as

$$\|\boldsymbol{x}\|_2 = \sqrt{1^2 + 4 \times 2^2}, \tag{2}$$

which only needs three multiplications and one addition inside the square root, of which the computational cost is less than Eq. (1), the naïve one.

CVS can compute general ℓ_p-norm without decompression. Suppose we have a vector $\boldsymbol{x}$ that can be expressed by B blocks in RLE:

$$\boldsymbol{x} = (x_1, \ldots, x_D),$$
$$\mathrm{RLE}(\boldsymbol{x}) = (\langle n_1, x_1\rangle, \ldots, \langle n_B, x_B\rangle).$$

Naïvely, ℓ_p-norm of $\boldsymbol{x}$ is computed as

$$\|\boldsymbol{x}\|_p = \left(\sum_{d=1}^{D} x_d^p\right)^{1/p} \tag{3}$$

whereas CVS computes ℓ_p-norm by

$$\|\mathrm{RLE}(\boldsymbol{x})\|_p = \left(\sum_{b=1}^{B} n_b x_b^p\right)^{1/p}. \tag{4}$$

From the Eqs. (3) and (4), it is obvious that CVS reduces the computational complexity of ℓ_p-norm from $O(D)$ to $O(B)$ where D is the number of dimensions of the input vector and B is the number of blocks in the compressed vector.

3.2.2 Dot Product

Consider the dot product of two vectors $\boldsymbol{x}_1 \in \mathbb{R}^{11}$ and $\boldsymbol{x}_2 \in \mathbb{R}^{11}$, whose compressed forms are

$$\mathrm{RLE}(\boldsymbol{x}_1) = (\langle 5, 4\rangle, \langle 6, 8\rangle),$$
$$\mathrm{RLE}(\boldsymbol{x}_2) = (\langle 3, 1\rangle, \langle 5, 3\rangle, \langle 3, 2\rangle).$$

Algorithm 1: Dot product of $\boldsymbol{x}$ and $\boldsymbol{y}$ without decompression.

Input: $\mathrm{RLE}(\boldsymbol{x}) = \langle n_1, x_1\rangle, \ldots, \langle n_{B_x}, x_{B_x}\rangle$ and
$\mathrm{RLE}(\boldsymbol{y}) = \langle n'_1, y_1\rangle, \ldots, \langle n'_{B_y}, y_{B_y}\rangle$, where $B_x \le B_y$.
Output: $\boldsymbol{x}^\top \boldsymbol{y}$

```
p ← 0
y_remains ← n'_1
j_next ← 0
for i ← 2 to B_x do
    x_remains ← n_i
    y_sum ← 0
    for j ← j_next to B_y do
        y_sum ← y_sum + min(x_remains, y_remains) × RLE(y)[j]
        if x_remains < y_remains then
            y_remains ← y_remains − x_remains
            break
        else
            x_remains ← x_remains − y_remains
            y_remains ← n'_{j_next+1}
            j_next ← j_next + 1
    p ← p + RLE(x)[i] × y_sum
return p
```

Naïvely, the dot product requires eleven multiplications and ten additions. In contrast, CVS computes the dot product as

$$\underbrace{4}_{n_1} \times \underbrace{(1 \times 3 + 3 \times 2)}_{\text{1st } y_{\text{sum}}} + \underbrace{8}_{n_2} \times \underbrace{(3 \times 3 + 2 \times 3)}_{\text{2nd } y_{\text{sum}}}, \tag{5}$$

by using the Algorithm 1. In Eq. (5), six multiplications and three additions are required, which is less than the naïve one. The computational complexity of the Algorithm 1 is $O(B_x + B_y)$, where B_x and B_y are the number of blocks to represent the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. The Algorithm 1 is analogous to the algorithm for merging two sorted arrays or sort-merge join algorithm in relational database systems. In similar way, CVS can compute addition $\boldsymbol{x} + \boldsymbol{y}$, subtraction $\boldsymbol{x} - \boldsymbol{y}$, and division $\boldsymbol{x}/\boldsymbol{y}$ of two compressed vectors in $O(B_x + B_y)$ time.

4. Data Mining Algorithms on CVS

So far, we have described two basic computations on CVS: ℓ_p-norm on a vector and dot product of two vectors. In practice, many complex data mining algorithms can be conducted without decompression using the concept of those computations. In this section, we demonstrate several data mining algorithms on CVS.

4.1 k-Nearest Neighbors Algorithm

k-Nearest Neighbors Algorithm (*k-NN*) is an important algorithm used in many applications including document search [18], density estimation [8], and instance-based classifiers. Given a query vector $\boldsymbol{q}$ and a set of vectors $\mathcal{X}$, k-NN algorithm finds k vectors that are closest to query vector $\boldsymbol{q}$ from $\mathcal{X}$.

Since k-NN computes all the distances between the query vector $\boldsymbol{q}$ and the set of vectors $\mathcal{X}$, its runtime is slow on large vector sets. While spatial index structures such as R-Tree and and SR-Tree [14] can reduce the number of distance computations by pruning unpromising vectors, still high-dimensional Euclidean distance computations are required, which can be heavy. CVS can co-exists with these methods and support reducing the com-

© 2018 Information Processing Society of Japan

putational time and memory usage of the distance computation.

Recall the Euclidean distance of vector $\boldsymbol{x}$ and vector $\boldsymbol{y}$ is defined as

$$\|\boldsymbol{x}-\boldsymbol{y}\|_2 = \sqrt{\|\boldsymbol{x}\|_2^2+\|\boldsymbol{y}\|_2^2-2\boldsymbol{x}^\top\boldsymbol{y}}. \tag{6}$$

Since Eq. (6) mainly consists of ℓ_2-norm and dot product, CVS efficiently computes the Euclidean distance of $\boldsymbol{x}$ and $\boldsymbol{y}$ without decompression.

4.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) algorithm is a mathematical optimization algorithm to find a local minimum of a function. In machine learning, many machine learning models including linear regression, logistic regression, and deep neural networks can be trained by SGD algorithm.

Let us consider the logistic regression with SGD algorithm on CVS. SGD randomly picks a training data $(\boldsymbol{x}_i, y_i)$ and update the regression coefficients $\boldsymbol{w}_{t+1}$ in the following equation

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \gamma\boldsymbol{x}_i\left\{y_i - \frac{1}{1+\exp(-\boldsymbol{w}_t^\top\boldsymbol{x}_i)}\right\}, \tag{7}$$

where $\boldsymbol{w}_t$ is the current regression coefficients (prediction model) and γ is the learning rate. Iteratively repeating the update Eq. (7), SGD converges to the global optima and completes the training of the model. Since Eq. (7) mainly consists of dot products and additions of vectors, CVS can be naturally applied.

Furthermore, prediction using the trained model $\boldsymbol{w}$ can also be made by CVS efficiently, since the prediction simply consists of a dot product. Concretely, logistic regression predicts the objective value $\hat{y}$ for input feature vector $\boldsymbol{x}$ by

$$p(\hat{y}|\boldsymbol{x},\boldsymbol{w}) = \frac{1}{1+\exp(-\boldsymbol{w}^\top\boldsymbol{x})}. \tag{8}$$

4.3 Kernel Method

Kernel methods, such as support vector machine and spectral clustering, are widely used in machine learning tasks because it can capture non-linearity in data well. One downside of kernel method is its high computational cost; Kernel method computes data-to-data similarity $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ through a kernel function $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ for all $(i, j) \in [J] \times [J]$, i.e., kernel methods compute the similarities of all vector pairs. This similarity computation requires $O(D^2J^2)$ time, which is quite large.

CVS can alleviate the computational cost of data-to-data similarity computation. Consider *radial basis function kernel* (*RBF-Kernel*), which is defined as

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2). \tag{9}$$

CVS reduces the computational complexity of Eq. (9) from $O(D)$ to $O(B_i + B_j)$ where B_i and B_j are number of blocks to represent original vectors in RLE, which is often much smaller than D. Thus, the total computational complexity of kernel method is reduced from $O(D^2J^2)$ to $O((B_i + B_j)^2J^2)$.

5. Dimension Reordering

Directly compressing the provided set of vectors does not always yield a good compression result. In this section, we show that CVS can improve the compression rate by reordering the dimensions of vectors without affecting the computational result.

5.1 Motivation of Dimension Reordering

Suppose we have a set of 6-dimensional vectors

$$\boldsymbol{x}_1 = (2, 1, 2, 1, 2, 1),$$
$$\boldsymbol{x}_2 = (1, 2, 1, 1, 2, 2),$$
$$\boldsymbol{x}_3 = (2, 1, 1, 2, 1, 1).$$

By compressing each vector with RLE/Packbits, we obtain the compressed form of the vectors as

$$\mathrm{RLE}(\boldsymbol{x}_1) = (\langle -6, 2, 1, 2, 1, 2, 1\rangle),$$
$$\mathrm{RLE}(\boldsymbol{x}_2) = (\langle -2, 1, 2\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle),$$
$$\mathrm{RLE}(\boldsymbol{x}_3) = (\langle 1, 2\rangle, \langle 2, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle).$$

However, if we reorder the dimensions by considering a permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 1 & 4 & 6 & 3 & 2 \end{pmatrix}$$

on the dimensions of the vectors, we obtain vectors

$$\boldsymbol{x}'_1 = (1, 1, 2, 2, 2, 1),$$
$$\boldsymbol{x}'_2 = (2, 2, 2, 1, 1, 1),$$
$$\boldsymbol{x}'_3 = (1, 1, 1, 1, 2, 2),$$

which yield the better result compared to the original vectors as

$$\mathrm{RLE}(\boldsymbol{x}'_1) = (\langle 2, 1\rangle, \langle 3, 2\rangle, \langle 1, 1\rangle),$$
$$\mathrm{RLE}(\boldsymbol{x}'_2) = (\langle 2, 3\rangle, \langle 1, 3\rangle),$$
$$\mathrm{RLE}(\boldsymbol{x}'_3) = (\langle 4, 1\rangle, \langle 2, 2\rangle).$$

Many data mining tasks consist of mathematical operations that are not affected by order of vector dimensions. We refer to such a mathematical operation as *dimension-order insensitive*. If data mining tasks on CVS are known to be dimension-order insensitive, CVS reorders the dimensions of vectors to improve the compression rate. All the operations we demonstrated in Section 3.2 are dimension-order insensitive.

5.2 Problem Definition

We hereafter treat a set of vectors $\mathcal{X}$ to be compressed by CVS as matrix X whose column-vector represents a vector and row-vector represents a dimension. For instance, the vector set in the previous section is treated as

$$X = \begin{bmatrix} \boldsymbol{x}_1^\top & \boldsymbol{x}_2^\top & \boldsymbol{x}_3^\top \end{bmatrix} = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 1 \\ 1 & 1 & 2 \\ 2 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix}.$$

Given a set of D-dimensional J vectors as matrix $X \in \mathbb{R}^{D\times J}$, CVS solves the following problem to find the best dimension-order:

© 2018 Information Processing Society of Japan

Problem 1 (Dimension-reordering)

$$\sigma^* = \arg\min_{\sigma \in \mathcal{P}(D)} \sum_{j}^{J} |RLE(\sigma(\boldsymbol{x}_j))|$$

where $\mathcal{P}(D)$ is all permutations on D-dimensions, $|\cdot|$ is the number of blocks to represent the vector in the compression, $\boldsymbol{x}_j$ is the j-th column vector of matrix X, and $\sigma(\boldsymbol{x}_j)$ is the vector obtained by reordering the dimensions of $\boldsymbol{x}_j$ using the permutation σ.

Unfortunately, Problem 1 is NP-hard. The problem is equivalent to the *Bitmap-index* reordering problem in column-oriented database systems, which is proved to be an NP-hard problem [16]. Also, the problem is found to be reduced from the Traveling Salesman Problem (TSP) under the Hamming distance, allowing us to investigate the use of heuristics for TSP [16]. In the rest of this section, we introduce two heuristic approaches for dimension reordering: greedy method and Scored-lex-sort method, both of which run in polynomial time.

5.3 Greedy Method

First, we introduce a greedy approach to find a good dimension order for Problem 1. The greedy method selects a row (a dimension), and then greedily selects the row that has minimum hamming distance with the previously selected row from the remaining rows as shown in Algorithm 2. In Algorithm 2, $a_{i*} \leftarrow b_{j*}$ denotes replacing the i-th row of the matrix A with the j-th row of the matrix B and $O_{D,J}$ denotes a zero matrix whose size is $D \times J$.

The greedy method runs in polynomial time. The number of row comparison regarding Hamming distance in the greedy method is $(D-1) + (D-2) + ... + 1 = D(D-1)/2$, and each row comparison needs J element comparison. Thus, the computational complexity of the greedy method is $O(D^2 J)$. Although $O(D^2 J)$ is polynomial, it is not appropriate for processing vectors with a large number of dimensions, D.

Algorithm 2: Greedy method

Input: $X \in \mathbb{R}^{D \times J}$: Input matrix.
Output: Reordered matrix
Data: P: Row pool.
$\mathbb{X}_{\text{cand}}$: Candidates set.
X': Reordered matrix.

```
foreach i ∈ {1, ..., D} do
    X' ← O_{D,J}
    /* Select i-th row in matrix X as the beginning
       row, and add other rows to the pool P.      */
    P ← {1, ..., D} \ {i}
    x'_{1*} ← x_{i*}
    for k ← 2 to D do
        /* Pick up j-th row that has minimum hamming
           distance with the beginning row from pool P,
           and add it to reordered matrix X'.       */
        j ← arg min_{j∈P} HammingDistance(X, k − 1, j)
        P ← P \ {j}
        x'_{k*} ← x_{j*}
    X_cand ← X_cand ∪ {X'}
return arg min_{X'∈X_cand} Size(X')
```

5.4 Scored-lex-sort Method

Second, we introduce *Scored-lex-sort method*, an efficient approach to find an approximate solution for the Problem 1 based on lexicographical sort. Scored-lex-sort method runs in $O(J\,D \log D)$, which is better than greedy method's $O(D^2\,J)$. To introduce scored-lex-sort method, we first describe ordinary lexicographical sort of a matrix.

Lexicographical sort of a matrix reorders rows of a matrix in lexicographical order, where the order of two rows are defined by comparing whose elements from left-to-right. For example, lexicographical sort of the rows of matrix

$$X = \begin{pmatrix} 3 & 4 & 7 & 3 & 0 & 4 \\ 2 & 4 & 2 & 2 & 0 & 0 \\ 3 & 4 & 2 & 5 & 8 & 4 \\ 1 & 3 & 7 & 3 & 8 & 4 \\ 5 & 7 & 7 & 5 & 8 & 4 \end{pmatrix} \tag{10}$$

produces a sorted matrix

$$X_{\text{lex}} = \begin{pmatrix} 1 & 3 & 7 & 3 & 8 & 4 \\ 2 & 4 & 2 & 2 & 0 & 0 \\ 3 & 4 & 2 & 5 & 8 & 4 \\ 3 & 4 & 7 & 3 & 0 & 4 \\ 5 & 7 & 7 & 5 & 8 & 4 \end{pmatrix}. \tag{11}$$

Since lexicographical sorting of a matrix preferentially sorts left-side columns, it sometimes does not much improve the compression rate of the matrix. For instance, X_{lex}'s compression rate is worse than the original matrix X's one. To alleviate this problem, we introduce *Scored-lex-sort method*. Scored-lex-sort method first computes scores of all columns of the matrix, and then preferentially sorts columns that have high score. Effectiveness of Scored-lex-sort method highly depends on the definition of the score. We use 1/Cardinality(**x**) as the score of a column **x**, where the function Cardinality(**x**) counts the number of distinct elements in column **x**. This score definition is based on the insight that a column with low-cardinality contains a lot of same elements, and preferentially sorting such columns improves the compression effect of RLE. For example, matrix X in the previous example is sorted into

$$X_{\text{lex}^*} = \begin{pmatrix} 2 & 4 & 2 & 2 & 0 & 0 \\ 3 & 4 & 2 & 5 & 8 & 4 \\ 3 & 4 & 7 & 3 & 0 & 4 \\ 1 & 3 & 7 & 3 & 8 & 4 \\ 5 & 7 & 7 & 5 & 8 & 4 \end{pmatrix}, \tag{12}$$

which has better compression rate than the original matrix X and the matrix X_{lex} that is sorted by the normal lexicographical order.

Algorithm 3 shows the algorithm of row comparison function in Scored-lex-sort method. In the algorithm, x_{ij} indicates the (i, j) element in the matrix X. For the sorting part, we can use arbitrary sorting algorithms such as the merge-sort algorithm to reorder a huge matrix in the external sorting manner.

Scored-lex-sort method runs in polynomial time, and its computational complexity is smaller than the one of the greedy method. As mentioned before, Scored-lex-sort method can use

© 2018 Information Processing Society of Japan

Algorithm 3: Row-compare function in Scored-lex-sort method

Input: $X \in \mathbb{R}^{D \times J}$: Input matrix.
$p \in \{ 1, ..., D \}$: Row number of the first input row.
$q \in \{ 1, ..., D \}$: Row number of the second input row.
Output: A numerical number indicating whether p-th row is bigger/smaller than or equal to q-th row
Data: Q, Priority queue

foreach $k \in \{ 1, ..., J \}$ **do**
  Get the score of k-th column, and add the column number k to the priority queue Q using the score as the priority.
while *Q is not empty* **do**
  $k \leftarrow$ Dequeue(Q)
  if $x_{pk} > x_{qk}$ **then**
    return *1, which indicates p-th row > q-th row.*
  else if $x_{pk} < x_{qk}$ **then**
    return *-1, which indicates p-th row < q-th row.*
return *0, which indicates p-th row = q-th row.*

Algorithm 4: Vector discretization

Input: $X \in \mathbb{R}^{D \times J}$: Collection of D-dimensional J vectors.
K: Discretization level.

foreach $d \in \{ 1, ..., D \}$ **do**
  foreach $j \in \{ 1, ..., J \}$ **do**
    Standardize d-th dimension (μ_d is the mean and σ_d is the standard deviation of d-th dimension).
    $X_{dj} = \frac{X_{dj} - \mu_d}{\sigma_d}$
Groups values into k clusters using k-means algorithm, obtaining centroid c_k and cluster assignment $z_{dj} \in \{ 1, ..., K \}$ for each value.
$\{ c_k \}_{k=1}^{K}, \{ z_{dj} \}_{d=1,j=1}^{D,J} \leftarrow \mathrm{kMeans}(\{ X_{dj} \}_{d=1,j=1}^{D,J}, K)$.
foreach $d \in \{ 1, ..., D \}$ **do**
  foreach $j \in \{ 1, ..., J \}$ **do**
    Replace the value of j-th vector by the centroid of the cluster the vector belongs to.
    $X_{dj} = c_{z_{dj}}$
return X

arbitrary general sorting algorithms. General sorting algorithms are known to sort D records in $O(D \log D)$ [5]. In a comparison operation of two rows, Scored-lex-sort method needs to compare J elements as described in Algorithm 3. Thus, the computational complexity of Scored-lex-sort method is $O(J\,D \log D)$, which is sufficiently applicable to large matrices.

6. Lossy Compression by Discretization

We have targeted discrete-valued vectors. However, real-world data is not only discrete-valued but also real-valued. Unfortunately, RLE is ineffective for real-valued data, because real-valued data rarely have same consecutive values. To deal with the problem, CVS employ discretization technique to convert real-valued vectors into discrete-valued vectors.

CVS uses a clustering-based discretization approach to improve the compression rate of the vectors as shown in Algorithm 4. In this approach, we first standardize each vector dimensions to have zero mean and unit variance, gather all feature values, and make k clusters from all feature values. Then, values in each cluster are replaced by the cluster centroid. In doing so, we can successfully convert real-valued data into k-discretized data.

One downside of discretization is that it loses the precision of vectors. Through empirical studies, we found that the data mining results, such as predictive performance in k-NN classifiers, are not so affected by the precision of vector values. For example, in binary image classification task, k-NN classifier on CVS with 3-level discretization only degrades the F1-Score 0.005 point while gaining 6× prediction performance improvement (**Fig. 1**). We will elaborate on this discussion in the experiments section.

7. Experiments

7.1 Experimental setup

System: All of our experiments were run on a machine that has 16GB RAM and dual-core 3.6GHz CPU running Linux 3.8.0. Our proposed system, CVS, is implemented in C++ and compiled by clang++ 3.8.

Datasets: We used two types of vector sets in our experiments: sparse vector sets (bag of words) [3] and dense high-dimensional vector sets [11]. **Table 2** shows the information of the datasets we used.

Methods: We measured the performance of conventional sparse matrix representation and CVS in several different configurations: **(Sparse)** refers to the conventional sparse matrix representation that only holds non-zero values by $(position, value)$ format, **(RLE)** refers to RLE/Packbits on vectors, **(RLE-Sort)** refers to RLE/Packbits on vectors with dimension-reordering, **(RLEn)** refers to RLE/Packbits with n-level data discretization, and **(RLEn-Sort)** refers to RLE/Packbits with n-level data discretization and dimension-reordering. In the rest of this section, we use these notations to explain the configuration on each result.

7.2 Space-efficiency

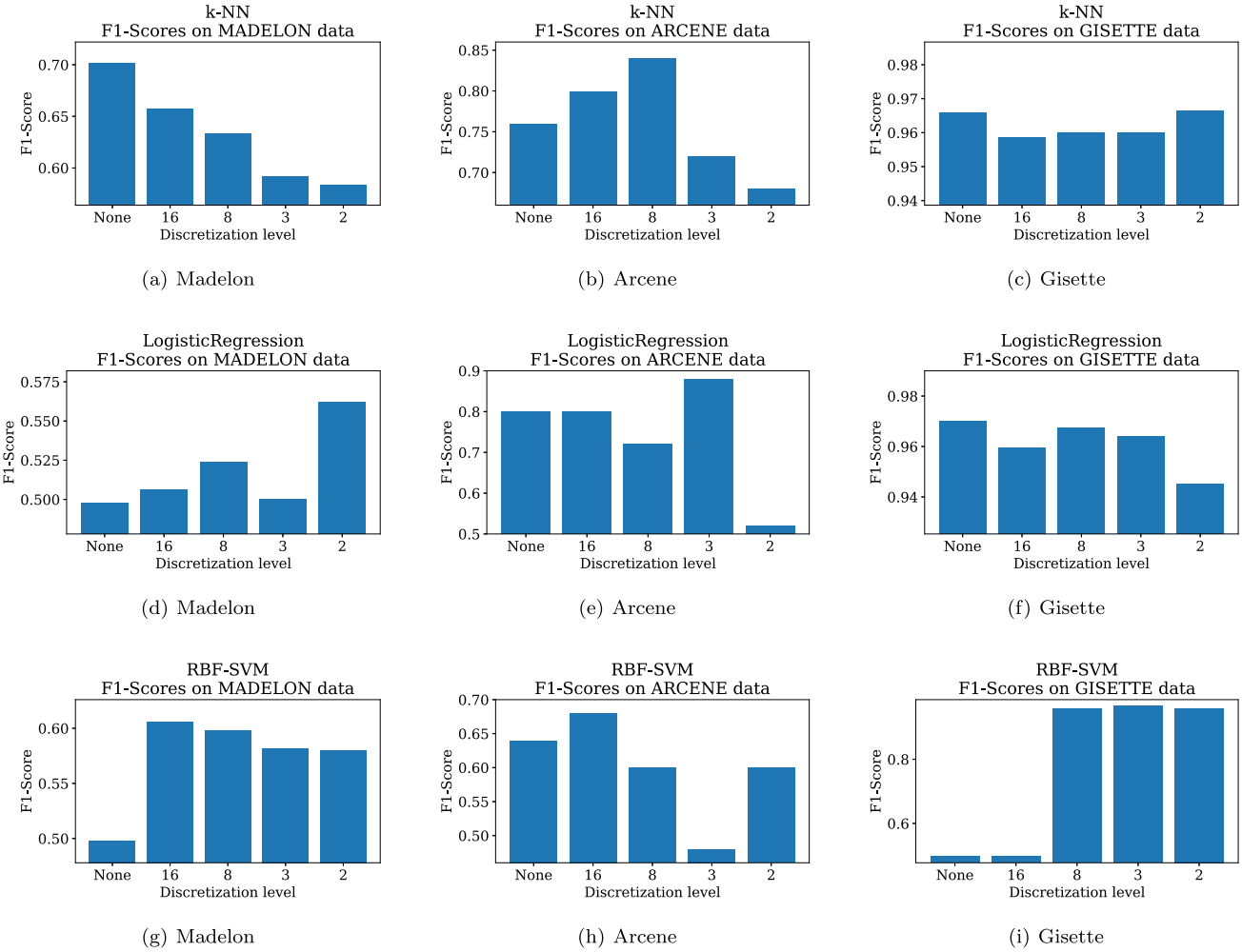
First, we demonstrate the space-efficiency of CVS on real-world vector sets with different configurations including vanilla RLE, dimension-reordering by Scored-lex-sort method, and data discretization. **Table 3** shows the compression rates of CVS with different configurations, where the compression rate is defined as `compression_rate = compressed_size / original_size`. Data sizes shown in Table 3 correspond to both the secondary storage usages of the compressed datasets and memory usages of data mining algorithms introduced in Section 4 (k-NN classifier, logistic regression with SGD [*1] and kernel method) on the datasets.

From Table 3, we have the following observations:

- Vanilla RLE is effective for sparse matrices because sparse matrices are mostly filled by zero and RLE effectively represents such zero sequences.
- Discretization is especially effective for dense matrices. Without discretization, RLE on dense matrices can increase the storage usage (e.g., RLE in Madelon).
- Only doing discretization is not enough. Combining discretization and dimension-ordering can drastically improve

[*1] Because SGD is an online algorithm, its memory usage can be reduced to the size of a vector instead of the whole dataset. However, because vectors are randomly scanned, loading whole dataset into the memory is required for efficient computation.

© 2018 Information Processing Society of Japan



(a) Madelon (b) Arcene (c) Gisette

(d) Madelon (e) Arcene (f) Gisette

(g) Madelon (h) Arcene (i) Gisette

Fig. 1 F1-scores (harmonic mean of precision and recall) of each classifier (k-NN classifier, logistic regression, and RBF-SVM) on each dataset with discretization. X-axis refers to the discretization level (# of distinct values) and Y-axis refers to the F1-score. We can observe that the discretization does not much degrade the predictive performance and sometimes does improve F1-scores.

Table 2 Datasets used in experiments.

	Sparse (Bag of Words) [3]			Dense [11]		
	NIPS	KOS	Enron	Madelon	Arcene	Gisette
# of vectors	1,500	3,430	39,861	2,000	900	6,000
# of dimensions	12,419	6,906	28,102	500	10,000	5,000
Density (Non-zero elements ratio)	0.040	0.014	0.003	0.999	0.540	0.129
# of distinct values	120	33	134	660	899	692

Table 3 Data compression effect on each dataset by conventional sparse matrix representation (position + value) and our proposed framework CVS. Each number refers to the main/secondary storage usage in megabytes and (%) refers to the compression rate. Compression formats with asterisk (*) refer to lossy compression. While conventional sparse matrix representation successfully compresses the sparse datasets, it fails to compresses the dense datasets. In contrast, our proposed CVS successfully compresses dense and sparse datasets.

	Sparse (Bag of Words)			Dense		
	NIPS	KOS	Enron	Madelon	Arcene	Gisette
Original	71.06	90.36	4273.12	3.81	3.81	114.44
Sparse Matrix	8.54 (12.01%)	4.04 (4.47%)	42.46 (0.99%)	11.44 (300.26%)	6.19 (162.46%)	44.53 (38.91%)
RLE	10.0 (14.07%)	5.07 (5.62%)	53.69 (1.25%)	3.86 (101.40%)	3.84 (100.90%)	49.92 (43.62%)
RLE-Sort	9.12 (12.84%)	4.75 (5.26%)	51.71 (1.21%)	3.88 (101.79%)	3.20 (83.97%)	39.03 (34.10%)
RLE8*	9.94 (13.99%)	5.07 (5.62%)	53.67 (1.25%)	3.55 (93.10%)	2.91 (78.94%)	41.18 (36.31%)
RLE8-Sort*	8.88 (12.50%)	4.76 (5.27%)	51.66 (1.20%)	3.51 (92.13%)	2.65 (71.74%)	38.70 (34.13%)
RLE3*	9.84 (13.85%)	4.52 (5.00%)	53.36 (1.24%)	2.43 (63.73%)	1.53 (51.33%)	40.40 (35.63%)
RLE3-Sort*	8.54 (12.02%)	4.52 (5.00%)	50.34 (1.17%)	2.39 (62.76%)	1.02 (34.16%)	38.79 (34.21%)
RLE2*	9.82 (13.82%)	5.04 (5.58%)	53.32 (1.24%)	2.10 (62.26%)	0.83 (42.19%)	38.13 (33.66%)
RLE2-Sort*	8.41 (11.84%)	4.49 (4.97%)	49.79 (1.16%)	2.05 (60.75%)	0.48 (24.85%)	35.82 (31.63%)

© 2018 Information Processing Society of Japan

Table 4 Runtime speedups of k-NN classifiers with Euclidean distance on compressed data. Compression formats with asterisk (*) refer to lossy compression. While conventional sparse matrix represents in sparse data, our proposed method successfully improves the performance in both sparse and dense data. Further, sorting vector dimension and discretization drastically improve the performance.

	Sparse (Bag of Words)			Dense		
	NIPS	KOS	Enron	Madelon	Arcene	Gisette
Sparse	4.68	16.33	132.95	0.18	0.50	1.30
RLE	8.69	23.36	146.97	1.42	1.22	1.97
RLE-Sort	8.99	23.52	202.55	1.31	1.27	3.03
RLE8*	8.67	44.47	169.47	1.33	1.27	3.06
RLE8-Sort*	18.83	46.56	192.66	2.20	1.62	3.15
RLE3*	15.07	45.80	170.65	3.80	2.41	3.38
RLE3-Sort*	20.36	46.71	203.52	3.87	6.13	9.33
RLE2*	18.32	22.76	151.62	4.13	4.93	3.62
RLE2-Sort*	20.82	48.02	218.20	4.26	7.82	8.77

Table 5 Runtime speedups of logistic regression with SGD optimization on compressed data.

	Sparse (Bag of Words)			Dense		
	NIPS	KOS	Enron	Madelon	Arcene	Gisette
Sparse	12.73	41.55	174.21	0.65	1.30	4.81
RLE	16.72	46.24	182.26	1.81	2.38	6.61
RLE-Sort	19.08	51.26	199.80	1.85	2.78	7.92
RLE8*	22.65	44.28	165.18	2.64	2.68	7.48
RLE8-Sort*	24.98	50.28	187.48	2.69	3.08	8.19
RLE3*	23.04	45.61	164.27	4.28	4.10	7.86
RLE3-Sort*	26.24	49.57	132.69	4.34	5.36	7.99
RLE2-Sort*	25.51	52.24	189.67	5.19	6.31	9.47

Table 6 Runtime speedups of a kernel machine on compressed data.

	Sparse (Bag of Words)			Dense		
	NIPS	KOS	Enron	Madelon	Arcene	Gisette
Sparse	5.15	20.97	182.89	0.16	0.33	1.60
RLE	8.40	24.12	198.21	0.74	0.90	2.51
RLE-Sort	8.94	26.50	205.23	0.75	1.01	3.00
RLE8*	8.55	24.17	193.54	1.02	1.31	2.93
RLE8-Sort*	9.13	26.61	198.31	1.07	1.48	3.23
RLE3*	8.33	22.92	176.66	1.65	2.19	3.15
RLE3-Sort*	11.46	27.30	258.16	1.99	3.31	3.38
RLE2*	8.77	22.95	188.04	1.96	2.71	3.45
RLE2-Sort*	11.70	35.47	306.45	2.81	4.42	3.78

the compression rate (e.g., RLE8 in Gisette).

7.3 Runtime Speedup

Second, we compare the runtime-speedup of our method CVS and competitive method sparse vector representation on real-world vector sets. In this experiment, we measured the computational time of three classifiers we introduced in Section 4: k-NN classifier, logistic regression by SGD optimization, and a nonlinear kernel machine (support vector machine with RBF-kernel).

7.3.1 Setup

For k-NN, we measured the time of the prediction for test vectors, a computationally intensive process that looks through all the training vectors. Test vector is randomly picked up from the vector set, and top-k nearest vectors are selected by its Euclidean distance. We set $k = 5$, repeat the procedure 100 times measuring the elapsed time, and aggregate the results by taking the average. We used efficient Euclidean distance computation for both sparse vector representation and CVS, which run in $O(B_x + B_y)$ time.

In logistic regression with SGD optimization and kernel method (support vector machine with RBF-kernel), we measured the time to train the model. Since the model vector $\boldsymbol{w}$ trained is rarely sparse in both methods, we used ordinary dense vector representation for $\boldsymbol{w}$, and only used the compression method (sparse vector representation and CVS) for the input vectors $\{\boldsymbol{x}_j\}_j^J$.

7.3.2 Results

Table 4, **Table 5**, and **Table 6** show the performance comparison of classifiers. From Tables 4 to Table 6, we have the following observations:

- In sparse data, both sparse vector representation and our CVS improve the performance by orders of magnitude.
- While sparse vector representation is ineffective for dense data, CVS successfully improved the performance on dense data.
- Dimension reordering gives drastic performance improvements on both sparse and dense vector sets. For example, in Enron, a sparse dataset, the performance improvement increased from 146.97× to 202.55× by dimension reordering in k-NN classifier. In Gisette, a dense dataset, the performance improvement also increased from 1.97× to 3.03×, almost doubled, in k-NN classifier. We can also observe simi-

© 2018 Information Processing Society of Japan

lar results in logistic regression (Table 5) and kernel method (Table 6).

- Data discretization is effective for datasets that have large number of distinct values (Madelon, Arcene, Gisette). For datasets that have relatively smaller number of distinct values (NIPS, KOS, Enron), data discretization does not much improve the performance.

7.4 Discretization and Accuracy

Third, we discuss the effect of lossy-compression (discretization) on accuracy. We measured the predictive performance differences that come from data discretization for three classifiers we introduced in Section 4: k-NN classifier, logistic regression by SGD optimization, and a nonlinear kernel machine (support vector machine with RBF-kernel). We used Algorithm 4 for data discretization and varied discretization level $K \in \{2, 3, 8, 16\}$. For the distance measure in k-means algorithm, we used Euclidean distance.

Figure 1 shows the F1-score (harmonic mean of precision and recall) of each classifier on each dataset. X-axis refers to the discretization level (# of distinct values) and Y-axis refers to the F1-score. From Fig.1 (that shows accuracies) and Table 4 (that shows speedups), we observe that the discretization does not degrade the accuracy of classifiers much, while it drastically improves the compression rate and computational performance. Furthermore, discretization sometimes can improve the predictive performance classifiers. One may think this phenomena strange but data discretization is a well adopted feature-engineering technique to improve the predictive performance by reducing the effect of observation noise and outliers, which is referred to as *data binning* in machine learning area.

8. Related Work

CVS relates to vector and matrix computation frameworks that utilize data sparsity. For example, Eigen [10], a vector computation library, can represent sparse matrices and sparse vectors by concise data structures. In contrast to the sparse matrix representation, CVS targets to not only the sparse vector sets but also the dense vector sets.

Run-length encoding is widely used in data intensive systems. For example, MADlib [12], a data analytics system built on top of a relational database system, uses run-length encoding to handle sparse data efficiently. Also, some columnar database systems employ run-length encoding its column data [1], [17]. However, to the best of our knowledge, none of these works addressed the performance improvement of run-length encoding on data mining and machine learning tasks and the effect of dimension-reordering and data discretization.

In machine learning research area, compressing the specific machine learning models has been actively studied recently. **(Binary features)** Tabei et al. studied partial least squares regression (PLS) on compressed data encoded by grammar-based codes. To assist accessing the elements in the compressed matrix, they proposed a tree-based special data structure. Their approach targets to use binary-features (so-called fingerprints data), which differs from our approach that targets to arbitrary real values. **(Relational data)** Rendle proposed a method to accelerate machine-learning algorithms by utilizing block structures in a matrix [22]. In his work, input matrices are assumed to have special block structures that come from denormalization of relational tables, whereas CVS does not impose any assumptions on the input data. **(Deep neural networks)** In deep neural network research community, to reduce the size of huge deep learning models, DNN model compression has been recently actively studied. Approaches include low-rank approximation of parameter tensors [6], [13], binary representation [21] or ternary representation [23] of parameters, and pruning unimportant nodes from parameters [15].

Brodie *et al.* tackled the row-reordering problem we have defined in Section 5.2, and proposed a method that is similar to our greedy method, whose computational complexity is $O(D^2 J)$ [4]. In their situation, the greedy method was enough, because their aim was to compress the state-transition tables of a regular expression, and commonly such tables are not so large. In database systems area, the problem of reordering bitmap indices has been studied to get better compression result, which is equivalent to our dimension-reordering problem with binary values [16], [20].

9. Conclusion

In this paper, we proposed CVS (Compressed Vector Set), a general framework for concisely storing vector sets and conducting mathematical operations on the vector sets efficiently. CVS holds a set of vectors in a compressed format and conducts mathematical operations, such as ℓ_p-norm and dot product, without decompression. We demonstrate that CVS accelerates several data mining algorithms including k-nearest neighbor algorithm, stochastic gradient descent algorithm on logistic regression, and kernel methods. Our experimental results demonstrated that CVS can process both dense datasets and sparse datasets faster than conventional sparse vector representation with smaller memory usage.

References

[1] Abadi, D., Madden, S. and Hachem, N.: Column-Stores vs. Row-Stores: How different are they really?, *SIGMOD* (2008) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1376712⟩.

[2] Apple Inc.: Apple Technical Note TN1023 (1996).

[3] Bache, K. and Lichman, M.: UCI Machine Learning Repository (2013).

[4] Brodie, B.C., Taylor, D.E. and Cytron, R.K.: A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching, *ISCA*, pp.191–202, IEEE Computer Society (2006).

[5] Cormen, T.H., Stein, C., Rivest, R.L. and Leiserson, C.E.: *Introduction to Algorithms*, McGraw-Hill Higher Education, 2nd edition (2001).

[6] Denton, E., Zaremba, W., Bruna, J., LeCun, Y. and Fergus, R.: Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation, *NIPS* (2014) (online), available from ⟨http://arxiv.org/abs/1404.0736⟩.

[7] Fu, W.J.: Penalized Regressions: The Bridge versus the Lasso, *Journal of Computational and Graphical Statistics*, Vol.7, No.3, pp.397–416 (online), DOI: 10.2307/1390712 (1998).

[8] Fukunaga, K. and Hostetler, L.: The estimation of the gradient of a density function, with applications in pattern recognition, *IEEE Trans. Information Theory*, Vol.21, No.1, pp.32–40 (online), DOI: 10.1109/TIT.1975.1055330 (1975).

[9] Golub, G.H. and Van Loan, C.F.: *Matrix Computations* (*3rd Ed.*), Johns Hopkins University Press, Baltimore, MD, USA (1996).

[10] Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010), available from ⟨http://eigen.tuxfamily.org⟩.

© 2018 Information Processing Society of Japan

[11] Guyon, I., Gunn, S.R., Ben-Hur, A. and Dror, G.: Result Analysis of the NIPS 2003 Feature Selection Challenge, *NIPS* (2004).

[12] Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K. and Kumar, A.: The MADlib analytics library or MAD skills, the SQL, *VLDB*, Vol.5, No.12, pp.1700–1711 (2012) (online), available from ⟨http://dl.acm.org/citation.cfm?id=2367502.2367510⟩.

[13] Jaderberg, M., Vedaldi, A. and Zisserman, A.: Speeding up Convolutional Neural Networks with Low Rank Expansions, *CoRR*, Vol.abs/1405.3 (2014) (online), available from ⟨http://arxiv.org/abs/1405.3866⟩.

[14] Katayama, N. and Satoh, S.: The SR-tree: An index structure for high-dimensional nearest neighbor queries, *SIGMOD*, pp.369–380, ACM (1997).

[15] LeCun, Y., Denker, J.S. and Solla, S.A.: Optimal Brain Damage, *NIPS*, Touretzky, D.S. (Ed.), pp.598–605, Morgan Kaufmann (1989).

[16] Lemire, D., Kaser, O. and Gutarra, E.: Reordering Rows for Better Compression: Beyond the Lexicographic Order, *TODS*, Vol.37, No.3, pp.20:1–20:29 (online), DOI: 10.1145/2338626.2338633 (2012).

[17] Li, Y.: BitWeaving: Fast Scans for Main Memory Data Processing, *SIGMOD* (2013).

[18] Manning, C.D., Raghavan, P. and Schütze, H.: *Introduction to Information Retrieval*, Cambridge University Press, New York, NY, USA (2008).

[19] Oyamada, M., Liu, J., Narita, K. and Araki, T.: MOARLE: Matrix Operation Accelerator Based on Run-Length Encoding, *APWeb*, Chen, L., Jia, Y., Sellis, T. and Liu, G. (Eds.), pp.425–436, Springer International Publishing (2014) (online), available from ⟨http://dx.doi.org/10.1007/978-3-319-11116-2_37⟩.

[20] Pourabbas, E., Shoshani, A. and Wu, K.: Minimizing Index Size by Reordering Rows and Columns, *SSDBM'12*, Berlin, Heidelberg, Springer-Verlag, pp.467–484 (2012).

[21] Rastegari, M., Ordonez, V., Redmon, J. and Farhadi, A.: XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks, *ECCV*, Leibe, B., Matas, J., Sebe, N. and Welling, M. (Eds.), pp.525–542, Cham, Springer International Publishing (2016) (online), available from ⟨http://dx.doi.org/10.1007/978-3-319-46493-0_32⟩.

[22] Rendle, S.: Scaling Factorization Machines to Relational Data, *PVLDB*, Vol.6, No.5, pp.337–348 (2013).

[23] Zhu, C., Han, S., Mao, H. and Dally, W.J.: Trained Ternary Quantization, *ICLR*, pp.1–10 (2016) (online), available from ⟨http://arxiv.org/abs/1612.01064⟩.

Masafumi Oyamada received his B.E. and M.E. degrees from University of Tsukuba, Japan, in 2011 and 2013, respectively. He is currently a Senior Researcher and an Assistant Manager at Data Science Research Laboratories, NEC Corporation. He is also a Ph.D. student at University of Tsukuba. His research interests include database systems, machine learning, and information retrieval. He is a member of the Database Society of Japan (DBSJ).

Jianquan Liu received his B.E. degree from Shantou University, China, M.E. and Ph.D. degrees from University of Tsukuba, Japan, in 2005, 2009, and 2012, respectively. He was a Development Engineer in Tencent Inc. from 2005 to 2006, and was a visiting research assistant at the Chinese University of Hong Kong in 2010. He joined NEC Corporation in 2012, and is currently a Senior Researcher and an Assistant Manager at the System Platform Research Laboratories. He is also an Adjunct Assistant Professor at Hosei University, Japan. His research interests include multimedia databases, data mining, information retrieval, cloud computing, and social network analysis. Currently, he is/was serving as the PC Co-chair of IEEE conferences (ICSC2018, ISM2017, ICSC2017, ICRC2017, and BigMM2016), and the Workshop Co-chair of ICSC2016. He is a member of ACM and the Database Society of Japan (DBSJ).

Shinji Ito is a researcher at NEC Corporation. His research interests include numerical analysis, mathematical optimization, and machine learning. He received his M.S. degree in Mathematical Informatics and B.S degree in Mathematical Engineering and Information Physics from The University of Tokyo in 2015 and 2013, respectively.

Kazuyo Narita is a researcher at NEC Corporation. Her research interests include distributed systems, query processing, machine learning and feature engineering. She received her M.Eng. degree in Science and Engineering from University of Tsukuba in 2006.

Takuya Araki received his B.E., M.E., and Ph.D. degrees from The University of Tokyo, Japan in 1994, 1996, and 1999, respectively. He was a Visiting Researcher at Argonne National Laboratory from 2003 to 2004. He is currently a Principal Researcher of System Platform Research Laboratories, NEC Corporation. His research interests include parallel and distributed computing, big data analytics, and multimedia information retrieval. He is a member of IPSJ.

© 2018 Information Processing Society of Japan

Hiroyuki Kitagawa received his B.Sc. degree in physics and M.Sc. and Dr.Sc. degrees in Computer Science, all from The University of Tokyo. He is currently a Full Professor at Center for Computational Sciences and Center for Artificial Intelligence Research, University of Tsukuba. His research interests include data integration, databases, data mining, and information retrieval. He served as President of the Database Society of Japan from 2014 to 2016. He is an IEICE Fellow, an IPSJ Fellow, a member of ACM and IEEE, and an Associate Member of the Science Council of Japan.

(Editor in Charge: *Yasunori Ishihara*)

© 2018 Information Processing Society of Japan