PAPER

# Accelerating a Lloyd-Type k-Means Clustering Algorithm with Summable Lower Bounds in a Lower-Dimensional Space

**Kazuo AOYAMA**[†a)], **Kazumi SAITO**[††], ***and*** **Tetsuo IKEDA**[†††], ***Members***

**SUMMARY** This paper presents an efficient acceleration algorithm for Lloyd-type *k*-means clustering, which is suitable to a large-scale and high-dimensional data set with potentially numerous classes. The algorithm employs a novel projection-based filter (*PRJ*) to avoid unnecessary distance calculations, resulting in high-speed performance keeping the same results as a standard Lloyd's algorithm. The *PRJ* exploits a summable lower bound on a squared distance defined in a lower-dimensional space to which data points are projected. The summable lower bound can make the bound tighter dynamically by incremental addition of components in the lower-dimensional space within each iteration although the existing lower bounds used in other acceleration algorithms work only once as a fixed filter. Experimental results on large-scale and high-dimensional real image data sets demonstrate that the proposed algorithm works at high speed and with low memory consumption when large *k* values are given, compared with the state-of-the-art algorithms.

***key words:*** *algorithm, clustering, k-means, lower bound, singular value decomposition, principal component analysis, dimensionality reduction, performance*

## 1. Introduction

A *k*-means clustering problem is a problem of partitioning a given object set into *k* subsets (clusters) with a given positive integer of *k* so that an objective function is minimized. Since a data object is typically represented as a point in a Euclidean space of a feature space, the objective function is defined by the sum of squared Euclidean distances between all pairs of a point and a specified centroid that is a mean of points of the cluster to which the point is assigned. In general, exactly solving the problem for practically used data object sets requires a lot of computational costs. This leads to research topics of identifying a complexity class of the problem [1], [2] and developing efficient exact algorithms [3], [4], approximate algorithms with some constraints [5]–[7], and heuristic algorithms [8]–[10].

In these algorithms, a well-known iterative heuristic algorithm [8], [9], which is called Lloyd's algorithm, has been widely used because of its simplicity and empirical success. Lloyd's algorithm starts with given *k* centroids as seeds for *k* clusters, and repeats two steps of an assignment and a mean-update step until its convergence. In order to analyze the algorithm and improve the performance, various topics closely related to each other have been studied, such as the bounds on the number of iterations [11], [12], clustering accuracy to the optimal solution with seeding methods [13]–[16], and acceleration or speed-up of the algorithm for practical use [14], [17].

Acceleration algorithms keep the same results as Lloyd's algorithm if they start with the same initial centroids. Lloyd's algorithm incurs a high computational cost of distance calculations between a point and all *k* centroids at the assignment step. The key of the acceleration is to reduce the number of the distance calculations by avoiding unnecessary ones, i.e., the calculations of the distances from a point to centroids obviously farther than the centroid of the cluster to which the point is currently assigned. The acceleration algorithms identify such centroids by exploiting the lower bound on a distance, which is calculated with a low computational cost [17]–[25].

A typical algorithm compares the upper bound on a distance between a point and the centroid of a cluster to which the point is assigned with the lower bounds on distances the point and other $(k-1)$ centroids. Then it avoids calculating the exact distance between the point and the centroid of which the lower bound to the point is larger than the upper bound. Hereinafter the foregoing upper and lower bounds on the distances are referred to as the upper bound and the lower bounds unless confusion arises. The bounds are generally calculated by utilizing the triangle inequality in a property in a metric space.

There are several usages of the triangle inequality. Elkan's algorithm [17] and its variants [18]–[22] use a point, a current centroid, and its corresponding centroid at the last iteration as three points consisting of a triangle in the triangle inequality. Hattori's algorithm [23] employs a set of pivots instead of the last-iteration centroids. The Annular algorithm [21] and Bottesch's algorithm [24] adopt an origin as a single pivot*. Newling's algorithm utilizes centroids as pivots and identifies the centroids for exact distance calculations [25].

From the perspective of the trade-off between the speed and the memory consumption, however, these algorithms are *not* always *sufficient* as shown in Sect. 5. A more accelerated Lloyd-type algorithm with lower memory consumption is expected for a *large-scale* and *high-dimensional* data

Manuscript received December 4, 2017.
Manuscript publicized August 2, 2018.

[†]The author is with NTT Communication Science Laboratories, Kyoto-fu, 619–0237 Japan.
[††]The author is with Kanagawa University, Hiratsuka-shi, 259–1293 Japan.
[†††]The author is with University of Shizuoka, Shizuoka-shi, 422–8526 Japan.
a) E-mail: aoyama.kazuo@lab.ntt.co.jp

DOI: 10.1587/transinf.2017EDP7392

*In the geometric viewpoint, the Cauchy-Schwarz inequality in Bottesch's algorithm is equivalent to the triangle inequality.

set with a given *large k* value.

Our contributions are threefold.

1. We propose a novel scheme for efficiently accelerating Lloyd's algorithm in Sect. 4.1. A key of the acceleration is a newly introduced *summable lower bound* on a squared Euclidean distance. The summable lower bound is based on the following properties of a squared distance. Since a squared distance is the sum of its squared elements, a partial sum of the squared elements is the lower bound on the squared distance. The more squared elements are added, the tighter the lower bound becomes. We generate a lower-dimensional space rather than using an original space so that the summable lower bound in the generated space becomes tighter by the addition of fewer elements. In order to identify farther centroids from a point than the centroid to which the point is currently assigned, we exploit the two spaces, the generated and the original space, simultaneously.
2. We present a practical algorithm based on the scheme in Sect. 4.2. The algorithm has two distinct filters, a projection based filter (*PRJ*) based on the summable lower bound and an invariant centroid-pair based filter (*ICP*) similar to those in [23], [24], [26]. The *PRJ* utilizes singular value decomposition (SVD) of a given data point set to obtain orthonormal bases that generate a lower-dimensional space. Complementary effects of the two filters are shown in Sect. 5.3.
3. We experimentally demonstrate that our proposed algorithm achieves superior performance under the balance of the speed and the memory consumption, compared with the state-of-the-art algorithms [19], [20], when they are applied to large-scale and high-dimensional real image data sets with large $k$ values in Sect. 5.4.

The remainder of this paper consists of the following five sections. Section 2 describes some definitions and the notation. Section 3 briefly reviews related work from the two distinct viewpoints of accelerated Lloyd-type algorithms and the use of low-dimensional spaces for $k$-means clustering. Section 4 details the novel framework and the practical algorithm based on the scheme. Section 5 shows our experimental settings and demonstrates the results. The final section provides our conclusion and future work.

## 2. Definition and Notation

This section provides some definitions and the notation. Given a set of points $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n\}$, $|\mathcal{X}| = n$, and a positive integer of $k$, $k$-means clustering problem is to find a set of $k$ clusters, $C^* = \{C_1^*, C_2^*, \cdots, C_k^*\}$, defined by

$$C^* = \underset{C=\{C_1,\cdots,C_k\}}{\arg\min}\ J(C;\mathcal{X},k)\,, \tag{1}$$

$$J(C;\mathcal{X},k) = \sum_{C_j\in C}\sum_{\boldsymbol{x}_i\in C_j} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|^2\,, \tag{2}$$

$$\bigcup_{j=1}^{k} C_j = C \quad \text{(Covering)}\,, \tag{3}$$

$$C_j \cap C_{j'} = \emptyset \quad \text{for} \quad j \neq j' \quad \text{(Disjoint set)}\,, \tag{4}$$

where $\|*\|$ denotes the $L_2$ norm of any vector, $C$ the set of $k$ clusters, $\boldsymbol{\mu}_j \in \mathbb{R}^D$ the centroid (or mean) column vector of cluster $C_j$, $\boldsymbol{x}_i \in \mathbb{R}^D$ the point column vector, and $D$ the dimensionality of the original space.

The problem is also regarded as a problem of finding an optimal assignment of each point to a cluster. Then it is formulated based on an assignment matrix (or a cluster indicator matrix) $\boldsymbol{A} \in \mathbb{R}^{n\times k}$ with constraints as follows.

$$\boldsymbol{A}^* = \underset{\boldsymbol{A}}{\arg\min}\, \|\boldsymbol{X} - \boldsymbol{A}\boldsymbol{A}^T\boldsymbol{X}\|_F^2\,, \tag{5}$$

$$\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{I}_k\,, \tag{6}$$

where $\|*\|_F$ denotes the Frobenius norm of any matrix, $\boldsymbol{X} \in \mathbb{R}^{n\times D}$ the data point matrix, $\boldsymbol{A}^T$ the transpose of $\boldsymbol{A}$, and $\boldsymbol{I}_k \in \mathbb{R}^{k\times k}$ the identity matrix. Let $a_{ij}$ be the element at the $i$-th row and the $j$-th column of $\boldsymbol{A}$. Let $\boldsymbol{A}(i,:) = [a_{i,1}, \cdots, a_{i,k}]$ and $\boldsymbol{A}(j,:) = [a_{1,j}, \cdots, a_{n,j}]^T$ be the $i$-th row and the $j$-th column of $\boldsymbol{A}$, respectively. Then $a_{ij} = 1/\sqrt{|C_j|}$ if $\boldsymbol{x}_i$ is assigned to $C_j$ and $a_{ij} = 0$ otherwise. $\boldsymbol{A}(i,:)$ has only one non-zero element (Disjoint). $\boldsymbol{A}(:,j)$ has $|C_j| > 0$ non-zero elements and $\sum_{j=1}^{k}|C_j| = n$ (Covering). Let $\boldsymbol{M} \in \mathbb{R}^{k\times D}$ be the mean matrix of which the $j$-th row is $\boldsymbol{\mu}_j$. Then the $j$-th row of $\boldsymbol{A}^T\boldsymbol{X}$ is $\sqrt{|C_j|}\boldsymbol{\mu}_j$ and the $i$-th row of $\boldsymbol{A}\boldsymbol{A}^T\boldsymbol{X}$ is the mean vector of the cluster to which $\boldsymbol{x}_i$ is assigned.

For convenience, let $sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ and $d(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ denote $\|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|^2$ and $\|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|$, respectively. Let $sqd_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ and $d_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ denote the lower bounds on $sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ and $d(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$, respectively. Let us introduce a superscript $t$ indicating an iteration count of iterative algorithms. For instance, $C_j^{[t]}$ denotes the $j$-th cluster just after the assignment step at the $t$-th iteration and $\boldsymbol{\mu}_j^{[t]}$ its mean vector calculated at the mean-update step at the $t$-th iteration. To distinguish centroids in the ascending order of a distance from $\boldsymbol{x}_i$, let us introduce a function $ctr_h(\boldsymbol{x}_i)$, $h = 1, 2, \cdots, k$, that returns the $h$-th closest centroid to $\boldsymbol{x}_i$. Then let $\boldsymbol{\mu}_{h(\boldsymbol{x}_i)}$ denote the centroid returned by $ctr_h(\boldsymbol{x}_i)$. As a modified function of $ctr_h(\boldsymbol{x}_i)$, let $ctr_a(\boldsymbol{x}_i)$ be a function that returns the centroid to which $\boldsymbol{x}_i$ is assigned. For example, the functions are used as $ctr_1(\boldsymbol{x}_i) = ctr_a(\boldsymbol{x}_i)$ and $\boldsymbol{\mu}_j = \boldsymbol{\mu}_{a(\boldsymbol{x}_i)} = ctr_a(\boldsymbol{x}_i)$. Furthermore, let $ctr_{a^{[t]}}(\boldsymbol{x}_i)$ and $ctr_{j^{[t]}}(\boldsymbol{x}_i)$ denote $ctr_a(\boldsymbol{x}_i)$ and $ctr_j(\boldsymbol{x}_i)$ at the $t$-th iteration, respectively.

For easy reference, we summarize the frequently used symbols in Table 1.

## 3. Related Work

This section reviews two distinct topics: accelerated Lloyd-type algorithms that is a class to which the proposed algorithm belongs, and the use of low-dimensional spaces for clustering. The former focuses on definition and usage of the lower bound on a distance or a squared distance. The

**Table 1** Notation.

| Symbol | Description and Definitions |
|---|---|
| $\mathcal{X}$ | **Set** of given points, $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n\}$ |
| $C^{[t]}$ | **Set** of clusters at the $t$-th iteration $C^{[t]} = \{C_1^{[t]}, \cdots, C_j^{[t]}, \cdots, C_k^{[t]}\}$ |
| $\mathcal{M}^{[t]}$ | **Set** of centroids at the $t$-th iteration $\mathcal{M}^{[t]} = \{\boldsymbol{\mu}_1^{[t]}, \cdots, \boldsymbol{\mu}_j^{[t]}, \cdots, \boldsymbol{\mu}_k^{[t]}\}$ $\boldsymbol{\mu}_j^{[t]}$ is the centroid of $C_j^{[t]}$ |
| $\mathcal{Z}_m$ | **Set** of projection vectors $\mathcal{Z}_m = \{\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_m\}$ |
| $n$ | **Number** of points, $n = \lvert\mathcal{X}\rvert$ |
| $k$ | **Number** of centroids, $k = \lvert\mathcal{M}^{[t]}\rvert$ |
| $D$ | **Dimensionality** of the original space |
| $m$ | **Number** of projection vectors, $m = \lvert\mathcal{Z}_m\rvert$ and Dimensionality of the projected space |
| $\boldsymbol{x}_i \in \mathbb{R}^D$ | Column **vector** of the $i$-th point |
| $\boldsymbol{\mu}_j \in \mathbb{R}^D$ | Column **vector** of the $j$-th centroid |
| $\boldsymbol{X} \in \mathbb{R}^{n \times D}$ | **Matrix** consisting of point row vectors $\boldsymbol{X} = [\,\boldsymbol{x}_1, \cdots, \boldsymbol{x}_i, \cdots, \boldsymbol{x}_n\,]^T$ |
| $\boldsymbol{M} \in \mathbb{R}^{k \times D}$ | **Matrix** consisting of centroid row vectors $\boldsymbol{M} = [\,\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_j, \cdots, \boldsymbol{\mu}_k\,]^T$ |
| $\boldsymbol{A} \in \mathbb{R}^{n \times k}$ | Assignment **matrix** or Indicator **matrix** $a_{ij} = 1/\sqrt{\lvert C_j\rvert}$ if $\boldsymbol{x}_i$ is assigned to $C_j$ |
| $\boldsymbol{Z} \in \mathbb{R}^{m \times D}$ | **Matrix** consisting of projection row vectors $\boldsymbol{Z} = [\,z_1, \cdots, z_h, \cdots, z_m\,]^T$ |
| $ctr_h(\boldsymbol{x}_i)$ | **Function** of $\boldsymbol{x}_i$ that returns the $h$-th closest centroid to $\boldsymbol{x}_i$ |
| $d(\boldsymbol{\alpha}, \boldsymbol{\beta})$ | **Function** of points $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ that returns a distance between $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ |
| $d_{LB}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ | **Function** of points $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ that returns the lower bound on $d(\boldsymbol{\alpha}, \boldsymbol{\beta})$ |
| $sqd(\boldsymbol{\alpha}, \boldsymbol{\beta})$ | **Function** of points $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ that returns a squared distance between $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ |
| $sqd_{LB}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ | **Function** of points $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ that returns the lower bound on $sqd(\boldsymbol{\alpha}, \boldsymbol{\beta})$ |

latter reveals major differences between the existing algorithms and our proposed algorithm by addressing applied constraints and usage of low-dimensional spaces.

### 3.1 Accelerated Lloyd-Type Algorithms

Algorithms for solving the $k$-means clustering problem are classified into three classes of exact, approximate, and heuristic algorithms. The exact algorithms solve Eq. (2) or Eq. (5) [3], [4]. Most naively (impractically), the problem is solved by computing the objective function with respect to all the assignments of $k^n$ without any constraints. The approximate algorithms guarantee the accuracy of the solution in various manners [5]–[7]. The Lloyd-type algorithms belong to the heuristics that guarantee nothing in terms of their solution but are empirically known to find a good solution.

We begin by Lloyd's algorithm [8] followed by six acceleration algorithms in the order of Elkan's algorithm [17], Hamerly's algorithm [18] and its improved versions [21], [22], both Drake's algorithm [19] and Ding's algorithm [20], Hattori's algorithm [23], and Bottesch's algorithm [24]. Here the details of the algorithms are omitted for emphasizing their main filters.

**Algorithm 1** Lloyd-type algorithm

1: **Input:** $\mathcal{X}$, $\mathcal{M}^{[t-1]}$, $(k)$
2: **Output:** $C^{[t]} = \{C_1^{[t]}, C_2^{[t]}, \cdots, C_k^{[t]}\}$, $\mathcal{M}^{[t]}$
3: $C_j^{[t]} \leftarrow \emptyset$, $j = 1, 2, \cdots, k$
4: **for all** $\boldsymbol{x}_i \in \mathcal{X}$ **do**
5: $\quad d_{min} \leftarrow d(\boldsymbol{x}_i, ctr_{a^{[t-1]}}(\boldsymbol{x}_i))$
6: $\quad a \leftarrow a^{[t-1]}$
7: $\quad$ (I)
8: $\quad$ **for all** $\boldsymbol{\mu}_j^{[t-1]} \in \mathcal{M}^{[t-1]}$ (II) **do**
9: $\quad\quad$ (III)
10: $\quad\quad$ **if** $d(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]}) < d_{min}$ **then**
11: $\quad\quad\quad d_{min} \leftarrow d(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]})$ and $a \leftarrow j^{[t-1]}$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad C_a^{[t]} \leftarrow C_a^{[t]} \cup \{\boldsymbol{x}_i\}$
15: **end for**
16: $\boldsymbol{\mu}_j^{[t]} \leftarrow \sum_{\boldsymbol{x}_i \in C_j^{[t]}} \boldsymbol{x}_i \,/\, \lvert C_j^{[t]}\rvert$, $j = 1, 2, \cdots, k$
17: (IV)
18: **return** $C^{[t]} = \{C_1^{[t]}, C_2^{[t]}, \cdots, C_k^{[t]}\}, \mathcal{M}^{[t]}$

**[ Lloyd ]** Lloyd's algorithm is an iterative algorithm consisting of two steps: the assignment and the mean-update step. After $k$ centorids of seeds for $k$ clusters are given in any way as an initial state, the assignment step starts, where $n \times k$ distances between $n$ points $\boldsymbol{x}_i$ and the $k$ centroids are calculated and each $\boldsymbol{x}_i$ is assigned to the cluster of which centroid is the closest centroid $\boldsymbol{\mu}_{a(\boldsymbol{x}_i)}$ to $\boldsymbol{x}_i$. After all points are assigned to the identified clusters, the mean-update step calculates and updates means of the clusters. The algorithm iterates the two steps until the convergence or a given termination condition is satisfied. Algorithm 1 shows the overview of the Lloyd-type algorithms at the $t$-th iteration. The assignment step and the mean-update step correspond to lines 4–15 and lines 16–17, respectively.

**[ Elkan ]** This algorithm is a classical acceleration algorithm based on the triangle inequality. It keeps track of the lower bounds on distances between each point and $k$ centroids. A main filter is set at line 9 (III) in Algorithm 1 as follows.

**if** $d(\boldsymbol{x}_i, ctr_a(\boldsymbol{x}_i)) < d_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]}) - \delta(\boldsymbol{\mu}_j^{[t-1]})$
**then** continue ,

where $\delta(\boldsymbol{\mu}_j^{[t-1]}) = d(\boldsymbol{\mu}_j^{[t-1]}, \boldsymbol{\mu}_j^{[t-2]})$, that is, the mean's displacement between the iterations. The distance lower bound, which is represented by the right-hand side in the condition, is updated at line 17 (IV) after substituting $[t]$ for $[t-1]$. This algorithm requires the memory capacity of $O(n \cdot k)$ to store the distance lower bounds between each point and all centroids for the main filter. The required memory capacity makes it difficult to apply this algorithm to a large-scale data set.

**[ Hamerly ]** To suppress the memory consumption of Elkan's algorithm, Hamerly's algorithm stores only one dis-

tance lower bound of the second closest centroid to a point, resulting in the required memory capacity of $O(n)$. A main filter is set at line 7 (I) in Algorithm 1 as

**if** $d(\boldsymbol{x}_i, ctr_a(\boldsymbol{x}_i)) < d_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_2^{[t-1]}) - \delta_{max}(*)$
**then** continue ,

where $\delta_{max}(*) = \max_{\mathcal{M}^{[t-1]} \setminus \{\boldsymbol{\mu}_a^{[t-1]}\}} d(\boldsymbol{\mu}_j^{[t-1]}, \boldsymbol{\mu}_j^{[t-2]})$, i.e., the maximum displacement of all the means. The lower bounds are updated at line 17 (IV). This algorithm employs an all-or-nothing strategy by the filter at line 7 (I) with the looser lower bound than that of Elkan's algorithm. To compensate the weakness, a geometric method using the direction of a mean movement for making a lower bound tighter [22] and a pruning method using centroid norms for limiting the centroids at line 8 (II) [21] have been developed.

**[Drake$^+$ and Ding$^+$]** The two algorithms relax the all-or-nothing strategy of Hamerly's algorithm based on one lower bound for each point at the expense of more memory capacity. Main filters of Drake's and Ding's algorithm are regarded as a serial-type and a parallel-type filter, respectively. Drake's algorithm utilizes $b$ distance lower bounds ($1 < b < k$) from a point to its $b$ closest centroids with the memory capacity of $O(n \cdot b)$. The first $(b-1)$ lower bounds are determined in the same way as Elkan's algorithm and the last one is done like Hamerly's algorithm. A filter based on these lower bounds is set at line 7 (I).

In contrast, Ding's algorithm partitions $k$ centroids into $g$ groups ($1 < g < k$), and stores one distance lower bound from a point to its closest centroid in each group, resulting in the memory capacity of $O(n \cdot g)$. This scheme confines a negative impact of a large $\delta_{max}(*)$ in one group and prevents the lower bounds in the other groups from being loose. A group filter is set at line 7 (I). In the limits of ($b$ or $g \to 1$) and ($b$ or $g \to k$), the corresponding algorithms nearly approach Hamerly's and Elkan's algorithm, respectively.

**[Hattori$^+$]** Instead of keeping track of the lower bounds like the foregoing algorithms, this algorithm employs $p$ pivots ($1 < p \ll k$) to obtain the distance lower bound between a point and a centroid by the triangle inequality as

$$d_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]}; \boldsymbol{v}_h) = | \, d(\boldsymbol{v}_h, \boldsymbol{x}_i) - d(\boldsymbol{v}_h, \boldsymbol{\mu}_j^{[t-1]}) \, | \, ,$$

where $\boldsymbol{v}_h$, $h = 1, 2, \cdots, p$, denotes the pivot. For the lower bound calculation, this stores the distances from each of $p$ pivots to both the data points and centroids with the memory capacity of $O(n \cdot p)$, where $n \gg k$. A filter inserted at line 9 (III) is built as

**if** $\exists \boldsymbol{v}_h \in \{\boldsymbol{v}_1, \cdots, \boldsymbol{v}_h, \cdots, \boldsymbol{v}_p\}$;
$d(\boldsymbol{x}_i, ctr_a(\boldsymbol{x}_i)) < d_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]}; \boldsymbol{v}_h)$
**then** continue .

This filter judges via the $p$ pivots whether a distance calculation between a point and a centroid is necessary or not.

**[Bottesch$^+$]** In this algorithm, the lower bound on a squared Euclidean distance between $\boldsymbol{x}_i$ and $\boldsymbol{\mu}_j$ is defined by applying the Cauchy-Schwarz inequality to their blocked vectors. The same lower bound is also obtained by setting a single pivot in Hattori's algorithm at the origin of the space. A squared distance between $\boldsymbol{x}_i$ and $\boldsymbol{\mu}_j$ and the corresponding lower bound are expressed as follows.

$$sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j) = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{\mu}_j\|^2 - 2\, \boldsymbol{x}_i^T \boldsymbol{\mu}_j \, .$$

$$sqd_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j) = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{\mu}_j\|^2 - 2\, \|\boldsymbol{x}_i\| \, \|\boldsymbol{\mu}_j\| \, .$$

$sqd_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j) \le sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ is valid by the Cauchy-Schwarz inequality of $\boldsymbol{x}_i^T \boldsymbol{\mu}_j \le \|\boldsymbol{x}_i\| \, \|\boldsymbol{\mu}_j\|$ and the pivot-based triangle inequality of $sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j) \ge | \, d(\boldsymbol{o}, \boldsymbol{x}_i) - d(\boldsymbol{o}, \boldsymbol{\mu}_j) \, |^2$, where $\boldsymbol{o}$ denotes the origin. When $\boldsymbol{x}_i$ and $\boldsymbol{\mu}_j$ are expressed by block vectors each of which is divided into $r$ blocks ($1 < r \le D$) with the assumption of $D \bmod r = 0$, $(\boldsymbol{x}_i^T \boldsymbol{\mu}_j)$ is expressed as follows.

$$\sum_{h=1}^{D} x_{ih} \cdot \mu_{jh} = \sum_{s=0}^{r-1} \left( \sum_{h=s\cdot(\frac{D}{r})+1}^{(s+1)(\frac{D}{r})} x_{ih} \cdot \mu_{jh} \right) \tag{7}$$

$$\le \sum_{s=0}^{r-1} \left( \sum_{h=s\cdot(\frac{D}{r})+1}^{(s+1)(\frac{D}{r})} x_{ih}^2 \right)^{1/2} \left( \sum_{h=s\cdot(\frac{D}{r})+1}^{(s+1)(\frac{D}{r})} \mu_{ih}^2 \right)^{1/2} \tag{8}$$

$$\le \left( \textstyle\sum_{h=1}^{D} x_{ih}^2 \right)^{1/2} \left( \textstyle\sum_{h=1}^{D} \mu_{ih}^2 \right)^{1/2} = \|\boldsymbol{x}_i\| \cdot \|\boldsymbol{\mu}_j\| \tag{9}$$

This algorithm utilizes two filters based on Eq. (8) with parameter $r$ and Eq. (9). The filters based on the lower bounds are set at line 9 (III) as

**if** $sqd(\boldsymbol{x}_i, ctr_a(\boldsymbol{x}_i)) < sqd_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]})$
**then** continue .

To store the $r$ norms in Eq. (8), the filter requires the memory capacity of $O(n \cdot r)$ when $n \gg k$.

**[Complementary filter]** Our proposed algorithm adopts as a complementary filter an invariant centroid-pair based filter (*ICP*), which is used in the existing algorithms in [23], [24], [26]. *ICP* omits a distance calculation between an invariant centroid of $\boldsymbol{\mu}_j^{[t-1]} = \boldsymbol{\mu}_j^{[t-2]}$ and a point $\boldsymbol{x}_i$ of which assigned cluster's centroid is also invariant, i.e., $ctr_{a^{[t-1]}}(\boldsymbol{x}_i) = ctr_{a^{[t-2]}}(\boldsymbol{x}_i)$. *ICP* is known to be effective at the last stage in the convergence process [23]. Since a main filter *PRJ* in the proposed algorithm is effective at the early stage, *ICP* plays a role of the complementary filter. The proposed algorithm with both the filters is shown in Sect. 4.2.

### 3.2 Use of Low-Dimensional Spaces

The existing algorithms perform clustering of data points *themselves* in a low-dimensional space to which the data points are projected[†]. The algorithms work effectively only if the given number of clusters $k$ is smaller than or equal to the smaller one of the two values: the dimensionality $D$ in the original space and the number of data points $n$, i.e., $k \le \min(D, n)$. They construct the low-dimensional space by using one of feature selection such as column sampling, random projection based on fast Johnson-Lindenstrauss transform [27], [28], and (approximate) singular value decomposition (SVD) of $\boldsymbol{X} \in \mathbb{R}^{n \times D}$.

Since our proposed algorithm exploits SVD or principal component analysis (PCA), we focus on the SVD approach here. Using SVD, $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, where $\boldsymbol{\Sigma} \in \mathbb{R}^{r \times r}$

[†]For the projection, given data points $\boldsymbol{x}_i \in \mathbb{R}^D$ are centered as $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i - (\sum_{i=1}^{n} \boldsymbol{x}_i)/n$. Even in the projection case, we use the identical symbols for simplicity.

the positive diagonal singular matrix, $r = \mathrm{rank}(\boldsymbol{X})$, and $\boldsymbol{U} \in \mathbb{R}^{n\times r}$ and $\boldsymbol{V} \in \mathbb{R}^{D\times r}$ denote the left and the right singular matrix of $\boldsymbol{X}$, respectively. Kumar's algorithm in [29] executes Lloyd's algorithm in the low-dimensional space spanned by the column vectors in $\boldsymbol{V}$ corresponding to $k$ largest singular values of $\boldsymbol{X}$, where $k$ is the same as the number of clusters. Then it obtains $k$ means in the low-dimensional space and uses them for initial seeds of Lloyd's algorithm in the original space.

It is shown in [6], [7] that an objective function based on the point assignment in the low-dimensional space approximates that in the original space under some constraints. For instance, Boutsidis' algorithm [6] obtains assignment matrix $\tilde{\boldsymbol{A}} \in \mathbb{R}^{n\times k}$, which satisfies the specified approximation condition, by clustering of data points $\tilde{\boldsymbol{X}}$ projected to a low-dimensional space spanned by the row vectors of projection matrix $\boldsymbol{Z} = \boldsymbol{V}^T \in \mathbb{R}^{k\times D}$, i.e., $\tilde{\boldsymbol{X}} = \boldsymbol{X}\boldsymbol{Z}^T$. Then the following relationship between the objective functions holds with a bounded probability.

$$\|\boldsymbol{X} - \tilde{\boldsymbol{A}}\tilde{\boldsymbol{A}}^T\boldsymbol{X}\|_F^2 \le \alpha \|\boldsymbol{X} - \boldsymbol{A}^*\boldsymbol{A}^{*T}\boldsymbol{X}\|_F^2 , \tag{10}$$

where $\boldsymbol{A}^*$ is defined by Eq. (5) and $\alpha > 2$ is a constant.

In terms of dimensionality $m$ of a lower-dimensional space, the foregoing two algorithms in [6], [29] require $m = k \le \min(n, D)$ while the other in [7] does $m = \lceil k/\epsilon \rceil$ where $0 \le \epsilon < 1$. In contrast, it often happens that $k > D$ ($D < n$) in our problem setting where a large-scale and high-dimensional point set with potentially numerous classes, i.e., a large $k$ value is given. Thus our condition is out of the range set by the existing algorithms.

Besides, our proposed scheme differs from the foregoing algorithms in the usage of a lower-dimensional space. The algorithms execute a clustering algorithm such as Lloyd's algorithm in the lower-dimensional space while our scheme utilizes a lower-dimensional space for calculating the lower bound on a squared distance in the original space, resulting in the acceleration of Lloyd's algorithm in the original space.

## 4. Proposed Acceleration Algorithm

We first explain a novel scheme for efficiently accelerating the Lloyd-type algorithms. Our key ideas are threefold. The first is to introduce a summable lower bound to the Lloyd-type algorithm, which is the lower bound on a squared distance obtained by exploiting a property of a squared (Euclidean) distance effectively. The second is to generate a space where the lower bound becomes tighter than an original space. The last is to utilize the generated and the original space simultaneously for the acceleration. Next, we proposed a practical algorithm that employs a lower-dimensional space generated by the SVD of a data point matrix and makes the lower bound tighter incrementally by adding a squared elements in a squared distance.

### 4.1 Scheme of Projection Based Filter

A squared distance between points has a useful property to define the lower bound on the squared distance. The partial sum of squared elements consisting of the squared distance is the lower bound on the squared distance. A squared distance of $\boldsymbol{x}_i$ and $\boldsymbol{\mu}_j$ is expressed as

$$sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j) = \textstyle\sum_{h\in\Psi^{[D]}} (x_{ih} - \mu_{jh})^2 ,$$

where $\Psi^{[D]} = \{1, 2, \cdots, D\}$ and the superscript $[D]$ denotes $|\Psi^{[D]}|$. The partial sum of $m$ squared elements ($m < D$) in the squared distance is expressed as

$$sqd_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j) = \textstyle\sum_{h\in\Psi^{[m]}} (x_{ih} - \mu_{jh})^2 ,$$

where $\Psi^{[m]} \subset \Psi^{[D]}$. This formulation makes the lower bound summable. The more squared elements are added, the tighter the lower bound becomes. Suppose $\Psi^{[m+1]} = \{q\} \cup \Psi^{[m]}$, where $q \in \Psi^{[D]} \setminus \Psi^{[m]}$. Then,

$$\textstyle\sum_{h\in\Psi^{[m]}} (x_{ih} - \mu_{jh})^2 \le \sum_{h\in\Psi^{[m+1]}} (x_{ih} - \mu_{jh})^2 .$$

If this addition is repeated from $m = 1$ to $m = D$, then the lower bound asymptotically approaches the squared distance as

$$\lim_{m\to D} \textstyle\sum_{h\in\Psi^{[m]}} (x_{ih} - \mu_{jh})^2 = sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j) .$$

To improve this asymptotic rate, it is considered to simply add the squared elements in their descending order for the *one* vector. To obtain the same benefit, we employs an appropriately generated space spanned by orthonormal bases, and in particular limits the number of the orthonormal bases $m$ to a small value, i.e, a lower-dimensional space, for efficiency. To obtain the orthonormal bases for the lower-dimensional space, we compute the SVD of data point matrix and select $m$ column vectors in a right singular matrix, which correspond to the $m$ largest singular values.

Our projection based filter *PRJ* avoids unnecessary distance calculations by comparing a squared distance between a point and its currently assigned centroid in an original space with the summable lower bounds on squared distances between the point and the other centroids in the lower-dimensional space. *PRJ* does not calculate $sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ if $sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_{a(\boldsymbol{x}_i)})$ is smaller than or equal to the lower bound $sqd_{LB}(\boldsymbol{x}_i, \boldsymbol{\mu}_j)$ ($j \ne a(\boldsymbol{x}_i)$) is smaller than or equal to calculated in the lower-dimensional space, otherwise it updates the lower bound by the addition of the remaining squared elements and judges whether the condition is fulfilled or not again. Thus our scheme effectively exploits the property of the squared distance.

### 4.2 Acceleration Algorithm

The proposed algorithm contains two filters of *PRJ* as a main filter and an invariant centroid-pair based filter (*ICP*) as a complementary filter, which work effectively at the

early and the last stage in the convergence process, respectively, as shown later in Fig. 3.

For *PRJ*, our algorithm generates $m$ orthonormal bases as projection vectors $\boldsymbol{z}_h \in \mathbb{R}^D$, $h = 1, 2, \cdots, m \ll D$, by the SVD (or the PCA) of a data point matrix, and stores components $\boldsymbol{z}_h^T \boldsymbol{x}_i$ for all points with the memory capacity of $O(n \cdot m)$ before a clustering iteration starts. Besides, $\boldsymbol{z}_h^T \boldsymbol{\mu}_j$, $j = 1, 2, \cdots, k$, is also stored at each clustering iteration since the centroid may change its position. To optimize a projection vector set $\mathcal{Z}_m = \{\boldsymbol{z}_1, \boldsymbol{z}_2, \cdots, \boldsymbol{z}_m\}$, we adopt an orthogonal iteration, which iteratively computes $m$ dominant eigenvectors associated with the $m$ largest eigenvalues. For the effective addition of the *summable* lower bound, $\boldsymbol{z}_h$ is permutated in descending order of eigenvalues. Let $\boldsymbol{Z} \in \mathbb{R}^{m \times D}$ be the orthogonal projection matrix of which the $h$-th row vector is $\boldsymbol{z}_h^T$. Then the point corresponding to $\boldsymbol{x}_i$ projected to the $m$-dimensional projection space is expressed as $\boldsymbol{y}_i = \boldsymbol{Z}\boldsymbol{x}_i \in \mathbb{R}^m$, and a set of $\boldsymbol{y}_i$ is denoted by $\mathcal{Y}$. Note that the computational cost for determining the orthogonal projection matrix $\boldsymbol{Z}$ is much smaller than that for clustering in our setting since several times of the optimization iterations are sufficient as shown in Sect. 5.3.

Algorithm 2 overviews the assignment step in our proposed algorithm except a mean update step and the preprocessing before the clustering process starts. *PRJ* is shown at lines 11–18 and *ICP* at lines 8–10. The two filters in Algorithm 2 are inserted into line 9 (Ⅲ) in Algorithm 1. *ICP* skips the distance calculation between the point $\boldsymbol{x}_i$ and the centroid $\boldsymbol{\mu}_j^{[t-1]}$ satisfying the following condition. The point $\boldsymbol{x}_i$ is assigned to the cluster of which centroid did not change its position at the last iteration, i.e., $C_a^{[t-1]} = C_a^{[t-2]}$.

**Algorithm 2** Assignment step in proposed algorithm

1: **Input:** $\mathcal{X}$, $\mathcal{M}^{[t-1]}$, $\mathcal{Z}_m$, $(k)$
2: $\mathcal{Y} = \{\boldsymbol{y}_1, \cdots, \boldsymbol{y}_n\}$, where $\boldsymbol{y}_i = \boldsymbol{Z}\boldsymbol{x}_i \in \mathbb{R}^m$
3: **Output:** $C^{[t]} = \{C_1^{[t]}, C_2^{[t]}, \cdots, C_k^{[t]}\}$ and $\mathcal{M}^{[t]}$
4: $C_j^{[t]} \leftarrow \emptyset$ and calculate $\boldsymbol{Z}\boldsymbol{\mu}_j^{[t-1]}$ for all $j$
5: **for all** $\boldsymbol{x}_i \in \mathcal{X}$ **do**
6: $sqd_{min} \leftarrow sqd(\boldsymbol{x}_i, ctr_{a^{[t-1]}}(\boldsymbol{x}_i))$ and $a \leftarrow a^{[t-1]}$
7: **for all** $\boldsymbol{\mu}_j^{[t-1]} \in \mathcal{M}^{[t-1]}$ **do**
8: **if** $C_a^{[t-1]} = C_a^{[t-2]}$ and $C_j^{[t-1]} = C_j^{[t-2]}$ **then**
9: **go to** line 7 to evaluate the next $\boldsymbol{\mu}_j^{[t-1]}$
10: **end if**
11: $h \leftarrow 1$ and $sqd_{LB} \leftarrow 0$
12: **while** $h \leq m$ **do**
13: $sqd_{LB} \leftarrow sqd_{LB} + (y_{ih} - \boldsymbol{z}_h^T \boldsymbol{\mu}_j)^2$
14: **if** $sqd_{min} \leq sqd_{LB}$ **then**
15: **go to** line 7 to evaluate the next $\boldsymbol{\mu}_j^{[t-1]}$
16: **else** $h \leftarrow h + 1$
17: **end if**
18: **end while**
19: **if** $sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]}) < sqd_{min}$ **then**
20: $sqd_{min} \leftarrow sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]})$ and $a \leftarrow j^{[t-1]}$
21: **end if**
22: **end for**
23: $C_a^{[t]} \leftarrow C_a^{[t]} \cup \{\boldsymbol{x}_i\}$
24: **end for**
25: **return** $C^{[t]} = \{C_1^{[t]}, C_2^{[t]}, \cdots, C_k^{[t]}\}$ and $\mathcal{M}^{[t]}$

The centroid $\boldsymbol{\mu}_j^{[t-1]}$ does not change its position either as $C_j^{[t-1]} = C_j^{[t-2]}$. Then $\boldsymbol{x}_i$ is never assigned to $C_j^{[t]}$. *PRJ* skips the distance calculation between $\boldsymbol{x}_i$ and $\boldsymbol{\mu}_j^{[t-1]}$ if the tentative minimum squared distance $sqd_{min}$ in the original space is smaller than or equal to $\sum_{h=1}^{h'}(y_{ih} - \boldsymbol{z}_h^T \boldsymbol{\mu}_j^{[t-1]})^2$, $h' \leq m$, in the lower-dimensional space. If the condition is not fulfilled, *PRJ* makes the summable lower bound tighter incrementally, and judges the condition based on the updated lower bound again. Finally, if the filtering process fails as $sqd_{min} > \sum_{h=1}^{m}(y_{ih} - \boldsymbol{z}_h^T \boldsymbol{\mu}_j^{[t-1]})^2$, $sqd(\boldsymbol{x}_i, \boldsymbol{\mu}_j^{[t-1]})$ is calculated at lines 19–21.

Let us consider the computational cost of *PRJ* based on the number of multiplications related to the squared distance calculations. Then a baseline is the computational cost of Lloyd's algorithm for each iteration of $(n \cdot k \cdot D)$. The *PRJ* cost consists of three parts. First, the cost for generating the projected vectors of all centroids by $\boldsymbol{Z}\boldsymbol{\mu}_j^{[t-1]}$ is $(m \cdot k \cdot D)$. Second, the cost for calculating the lower bounds is $\sum_{h=1}^{m} h \cdot f(h)$, where $f(h)$ be the number of centroids that satisfy the filtering condition at line 14 at the $h$-th addition. Last, the penalty cost for recalculating the squared distances to the *unremoved* centroids, which are not filtered out with the condition at line 14, is

$$(n \cdot k - \theta - \sum_{h=1}^{m} f(h)) \cdot (m + D) \,,$$

where $\theta$ denotes the number of skipped squared distance calculations by *ICP* and the first term expresses the number of pairs of point $\boldsymbol{x}_i$ and centroid $\boldsymbol{\mu}_j^{[t-1]}$, which are evaluated at line 19. Therefore, the computational cost of *PRJ* with $m$ projection vectors is expressed by

$$\begin{aligned} cost(m) &= (m \cdot k \cdot D) + \textstyle\sum_{h=1}^{m} h \cdot f(h) \\ &\quad + \left(n \cdot k - \theta - \textstyle\sum_{h=1}^{m} f(h)\right) \cdot (m + D) \,. \end{aligned} \tag{11}$$

Let $\{1 - cost(m)/(n \cdot k \cdot D)\}$ be the reduction rate on the number of the multiplications in *PRJ* for convenience. The higher the reduction rate is, the more effective *PRJ* is. The effectiveness of *PRJ* is experimentally shown in Sect. 5.

## 5. Experiments

We describe data sets and performance measures for our experiments, and then show experimental results in terms of the properties that the proposed algorithm itself has and performance comparison with the state-of-the-art algorithms. In our experiments, all the algorithms were executed on a shared-memory computer system equipped with two Xeon E5-2697v3 2.6GHz CPUs and a 256GB main memory by 50-thread parallel processing based on OpenMP within the memory capacity.

### 5.1 Data Sets

We employed two different types of large-scale and high-dimensional (not sparse) real image data sets: One was *80*

*million tiny images* (*TinyImages* for short) [30] and the other *Holidays* [31].

The data set of *TinyImages* contains images represented by a global descriptor of a 384-dimensional GIST feature vector [32]. Each feature vector is regarded as a point in a 384-dimensional Euclidean space. We chose 10,219,916 feature vectors at random without duplication from the whole vectors as our experimental data set (10M-size). Furthermore, we made three data sets with the different sizes of 1,459,988 (1.5M-size), 2,919,976 (3M-size), and 5,839,952 (6M-size). In *Holidays* image data set, each image is represented by a set of local descriptors of SIFT keypoints extracted by the scale-invariant feature transform (SIFT) [33]. A SIFT keypoint can be regarded as a 128-dimensional feature vector. We adopted as a data set 20,964,516 feature vectors in 679 images chosen from the whole 1491 images.

Here, let us make particular reference to memory consumption, which is one of performance measures. Our in-house implementation required the physical memory sizes of 31.40GB and 21.47GB for *TinyImages* (10M-size) and *Holidays*, respectively.

### 5.2 Performance Measures

We evaluated performance of the algorithms with three measures. The first is the total elapsed time that each algorithm needs from just after loading a data set *until the convergence*. The elapsed time includes the CPU time required by preprocessing before a clustering process starts, e.g., the CPU time spent by SVD in the proposed algorithm. Besides we also adopted a truncated elapsed time until the algorithm satisfied a given condition expressed as

$$\gamma > |\, J^{[t]} - J^{[t-1]} \,| \,/\, J^{[t-1]} \,, \tag{12}$$

$$J^{[t]} = J(C^{[t]}; \mathcal{X}, k) \,, \tag{13}$$

where $J(C^{[t]}; \mathcal{X}, k)$ denotes the objective function value at the $t$-th iteration and $\gamma$ a given threshold parameter. In our setting, $\gamma = 1 \times 10^{-4}$. The truncated elapsed time is effective as a practical measure because the Lloyd-type algorithm has a tendency to rapidly decrease its objective function value at the early stage in the iterations. Figure 1 shows the objective function properties when Lloyd's algorithm with an initial state determined at random was applied to *TinyImages* (10M-size) at $k = 20{,}000$ and it converged at 509 iterations.

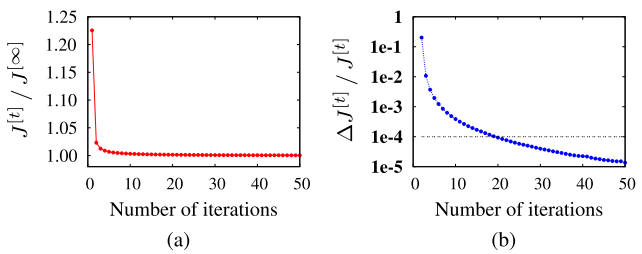


**Fig. 1** Objective function properties along the iterations when Lloyd's algorithm was applied to *TinyImages* (10M-size) at $k = 20{,}000$. (a) $J^{[t]} / J^{[\infty]}$ in linear scale and (b) $\Delta J^{[t]} / J^{[t]}$ in **logarithmic scale** where $\Delta J^{[t]} = |\, J^{[t+1]} - J^{[t]} \,|$.

Figures 1 (a) and (b) show $J^{[t]}/J^{[\infty]}$ and $|\, J^{[t]} - J^{[t-1]} \,| \,/\, J^{[t-1]}$ along the number of iteration $t$, respectively, where $J^{[\infty]}$ denotes the objective function value at the convergence. It was observed that $J^{[t]}/J^{[\infty]}$ quickly approached to 1.0 in early several iterations. In this case, the foregoing condition was satisfied at only the 20-th iteration.

The second is a computational cost of the number of exact distance calculations and the number of multiplications related to the distance calculations. They are simplified as reduction rates. The reduction rate on the number of multiplications is defined in Eq. (11). Regarding the number of distance calculations, it is defined as the rate of the number of unremoved distance calculations to the baseline of $(n \cdot k)$ for each iteration. These measures are more appropriate to evaluating algorithmic aspects than the total elapsed time because they are independent of computer architectures and implementations. The last is the maximum physical memory size occupied through the convergence process. An efficient algorithm should achieve short elapsed time and a large reduction rate with low memory consumption.

### 5.3 Properties of the Proposed Algorithm

We show three properties of the proposed algorithm. The first is the effect of the number of projection vectors $m$ to the performance. The second is each effect of the two filters, *PRJ* and *ICP*. The last is the scalability in terms of the number of clusters $k$ and the data size $n$. Hereinafter, we provide typical results each of which was obtained under an experimental condition, but also observed the similar tendencies under the other conditions.

Figure 2 shows the performance along the number of projection vectors $m$ from 5 to 100 when the algorithm with $k = 20{,}000$ was applied to *TinyImages* (10M-size). Figure 2 (a) shows the average reduction rate and the average elapsed time, which is the total elapsed time divided by the number of iterations until the convergence. The average reduction rate was over 0.93 even when $m = 10$, and approached asymptotically to 1.0 with $m$. In contrast, the average elapsed time had the minimum value at $m = 90$. This is because positive and negative effects in terms of the elapsed time balance: the positive one is to increase the average reduction rate with $m$ and the negative one is to increase the computational cost for calculating the lower bound. Fig-

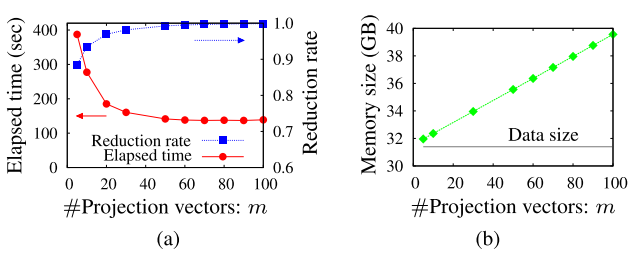


**Fig. 2** Proposed algorithm performance along the number of projection vectors $m$ when applied to *TinyImages* (10M-size) at $k = 20{,}000$. (a) Average elapsed time and average reduction rate, and (b) Maximum physical memory size including data size.

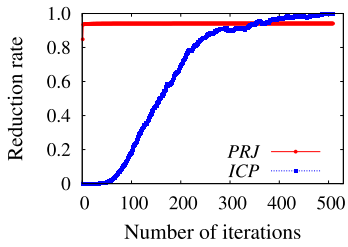


**Fig. 3** Reduction rate on the number of multiplications when the algorithm with either *PRJ* or *ICP* was applied to *TinyImages* (10M-size), given $k$ = 20,000 and $m$ = 30.

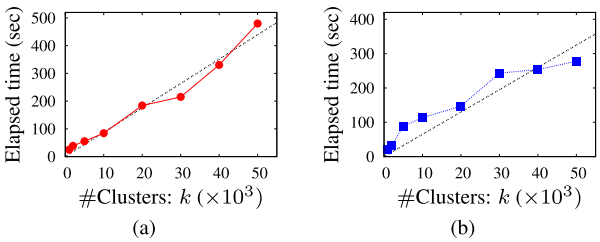


**Fig. 4** Average elapsed time along the number of clusters $k$ when $m$ = 10: (a) *TinyImages* (10M-size) and (b) *Holidays*.

ure 2 (b) shows the maximum physical memory size. The proposed algorithm needs memory capacity proportional to the number of projection vectors $m$, except the data size, which is approximated by $m \cdot (n + D + k) \cdot (\text{sizeof}(double)) \sim m \cdot n \cdot (\text{sizeof}(double))$ for $n \geq D, k$ in our implementation.

It seems that computing the SVD of a large-scale data point matrix incurs a high cost before clustering. In our case, however, the computational cost of the SVD is negligible compared with that of the $k$-means clustering process because of two reasons. One is that our proposed algorithm does not need an exact singular matrix (or an eigenvector set) as far as the orthogonality is guaranteed as described in Sect. 4.1. The other is that the $m$ value in our algorithm is small like $m \ll \min(D, n)$ as shown in Fig. 2 (a). For computing the SVD, the proposed algorithm adopts an orthogonal iteration, which is a simple and fast method (e.g. [34]). In fact, as shown later in Fig. 6, the elapsed time for one iteration in the orthogonal iteration was almost 210 sec while the elapsed time for one iteration for clustering exceeded it before 156 iterations in all the 509 iterations until convergence. The other experiments showed the same tendency. In each experiment, the elapsed time of the orthogonal iteration was much smaller than that of the clustering.

Figure 3 shows the reduction rate on the number of multiplications along the number of iterations until the convergence when the algorithm employing either *PRJ* or *ICP* as its filter was applied to *TinyImages* (10M-size), given $k$ = 20,000 and $m$ = 30. The *PRJ* and *ICP* were effective in a mutually complementary way at the very early and the last stage in the convergence process, respectively.

Figures 4 (a) and (b) show the average elapsed time along the number of clusters $k$ for the *TinyImages* (10M-size) and *Holidays*, given $m$ = 10, respectively. Each average elapsed time was almost linear or sublinear to $k$ although there exist the ranges of $k$ where the slopes were slightly different from the lines. It was observed that the algorithm was scalable to the number of clusters $k$.

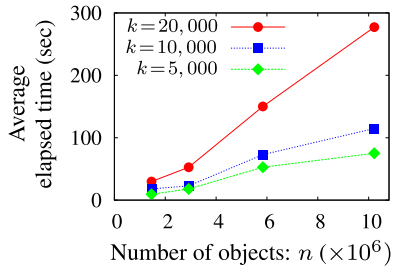


**Fig. 5** Average elapsed time through all iterations along the number of data points $n$ for *TinyImages* when $m$ = 10 and $k$ = 5,000, 1,0000, and 20,000.

Figure 5 shows the scalability of the proposed algorithm, regarding the number of data points $n$. The algorithm was applied to the four *TinyImages* data sets of 1.5M-size, 3M-size, 6M-size, and 10M-size under $k$ = 5,000, 10,000, and 20,000, and $m$ = 10. Each average elapsed time was almost linear with respect to $n$ in the range of nearly $1 \times 10^6$ to $1 \times 10^7$. We think that this scalability comes from the fact that the high fixed reduction rate was kept even in the different size of the data sets. Thus the proposed algorithm is suitable for large-scale and potentially numerous-class data sets, given a large value of $k$.

### 5.4 Comparison with Existing Algorithms

We compared the proposed algorithm (*Proposed*) with Drake's algorithm (*Drake*$^+$) and Ding's algorithm (*Ding*$^+$) for the distinct data sets of *TinyImages* (10M-size) and *Holidays*, varying their parameters under $k$ = 20,000. Drake's and Ding's algorithm (in-house implementation) were selected as the compared algorithms since they indicated higher performance than the other existing Lloyd-type algorithms such as Hamerly's and Bottesch's algorithm in Sect. 3 in our preliminary experiments[†].

Tables 2 and 3 show characteristic performance of the algorithms when they were applied to the *TinyImages* and *Holidays*, respectively. The average elapsed time through all the iterations and the maximum memory size except that occupied by the data point set are listed. Figures 6 and 7 show the elapsed time required by each algorithm under the conditions in Tables 2 and 3, respectively.

When allowed to use almost the same memory size, the three algorithms indicated the performance shown at the rows of $(m, b, g)$ = (30, 20, 20) in Table 2 and in Fig. 6 (a). The proposed algorithm successfully operated in the around 17% average elapsed time of that required by Ding's algo-

[†]All the algorithms were implemented in "*C*" with OpenMP for parallel processing although the original Drake$^+$ algorithm and the Ding$^+$ algorithm are implemented in "*C++*" with the standard template library and besides the Ding$^+$ algorithm is done in *GraphLab*, a framework for parallel processing [35].

**Table 2** Average elapsed time and memory size except that for the data points of *TinyImages* (∼ 31.40 GB).

| Algorithm | Parameter $(m, b, g)$ | Elapsed time (sec) | Memory size (GB) |
|---|---|---|---|
| ***Proposed*** | $m = 30$ | **160.5** | **2.56** |
| *Drake*$^+$ | $b = 20$ | **986.3** | **3.16** |
| | $b = 400$ | 556.1 | 48.7 |
| *Ding*$^+$ | $g = 20$ | **954.2** | **3.30** |
| | $g = 400$ | 208.6 | 48.8 |

**Table 3** Average elapsed time and memory size except that for the data set of *Holidays* (∼ 21.47 GB).

| Algorithm | Parameter $(m, b, g)$ | Elapsed time (sec) | Memory size (GB) |
|---|---|---|---|
| ***Proposed*** | $m = 20$ | **149.1** | **4.52** |
| *Drake*$^+$ | $b = 10$ | **323.7** | **4.64** |
| | $b = 200$ | 112.5 | 51.48 |
| *Ding*$^+$ | $g = 10$ | **238.5** | **4.95** |
| | $g = 200$ | 42.60 | 51.81 |

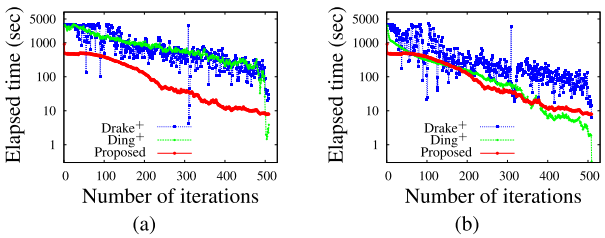


**Fig. 6** Elapsed time (**logarithmic scale**) of each algorithm with $k = 20,000$ for *TinyImages*, which corresponds to that in Table 2. (a) Almost the memory size was employed, and (b) *Drake*$^+$ and *Ding*$^+$ used the much larger memory size.

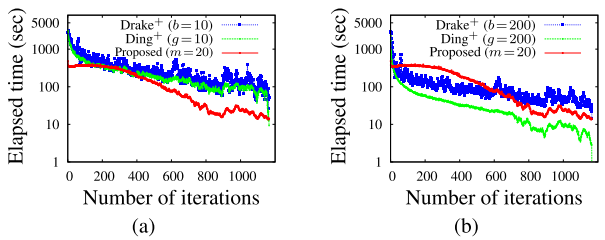


**Fig. 7** Elapsed time (**logarithmic scale**) of each algorithm with $k = 20,000$ for *Holidays*, which corresponds to that in Table 3. (a) Almost the memory size was employed, and (b) *Drake*$^+$ and *Ding*$^+$ used the much larger memory size.

rithm, and kept the much shorter elapsed time through all the iteration. Even when Drake's and Ding's algorithm were allowed to use the large memory size of around 49GB, their performances were below that of the proposed algorithm as shown at the rows of $(m, b, g) = (30, 400, 400)$ in Table 2 and in Fig. 6 (b), although they reduced the average elapsed time to the half and the one-fifth, respectively. Thus the proposed algorithm achieved the better speed performance than the compared algorithms with only their 1/20 memory size for *TinyImages* (10M-size).

**Table 4** Truncated elapsed time under the condition in Eq. (12) of $\gamma = 1 \times 10^{-4}$ for *TinyImages* (10M-size) and *Holidays* when $k = 20,000$. The numbers of iterations at the termination for *TinyImages* and *Holidays* are 20 and 26, respectively. Note that the truncated elapsed time is not averaged unlike the average elapsed time in Tables 2 and 3.

| Algorithm | *TinyImages* | | *Holidays* | |
|---|---|---|---|---|
| | Param. $(m, b, g)$ | Elapsed time (sec) | Param. $(m, b, g)$ | Elapsed time (sec) |
| ***Proposed*** | $m = 30$ | **12,037** | $m = 20$ | **10,009** |
| *Drake*$^+$ | $b = 20$ | **56,787** | $b = 10$ | **51,138** |
| | $b = 400$ | 61,145 | $b = 200$ | 26,816 |
| *Ding*$^+$ | $g = 20$ | **67,001** | $b = 10$ | **40,460** |
| | $g = 400$ | 29,039 | $g = 200$ | 12,924 |

Table 3 shows their performance for the other data set *Holidays* in the two different cases, $(m, b, g) = (20, 10, 10)$ and $(m, b, g) = (20, 200, 200)$, similar to those in Table 2. In the case of using the almost same memory sizes, which is shown at the rows of $(m, b, g) = (20, 10, 10)$ in Table 3 and in Fig. 7 (a), the average elapsed time that the proposed algorithm required was only 47% and 64% of those required by Drake's and Ding's algorithm, respectively. As shown at the rows of $(m, b, g) = (20, 200, 200)$ in Table 3 and in Fig. 7 (b), when the much larger memory size of around 51GB was available, which is 11 times the memory size that the proposed algorithm used, Drake's and Ding's algorithm lowered the average elapsed time to 35% and 18%, respectively. By controlling the parameters $b$ and $g$, those algorithm traded the occupied memory size for speed in our experiments. In contrast, the proposed algorithm achieved the high-speed performance with the much lower memory consumption without a trade-off parameter.

Table 4 shows the truncated elapsed time of each algorithm when the algorithms were applied to *TinyImages* (10M-size) and *Holidays* under the same conditions as those in Tables 2 and 3. The parameter $\gamma$ in Eq. (12) was set at $1 \times 10^{-4}$. Then the numbers of iterations were 20 and 26 for *TinyImages* and *Holidays* until the termination condition was satisfied. Note that the truncated elapsed time in Table 4 is not averaged, i.e., it is just the elapsed time until the termination. The truncated elapsed time of the proposed algorithm includes the elapsed time spent by SVD in preprocessing. For both the data sets, the proposed algorithm was fastest than the others even if they were operated with the large memory capacity. Thus the proposed algorithm works well in practice.

## 6. Conclusion

We experimentally validated that the proposed algorithm worked at high speed and with low memory consumption for the large-scale and high-dimensional real image data sets, given large $k$ values, compared with the state-of-the-art algorithms: Drake's and Ding's algorithm. For the TinyImages data set, in particular, our algorithm achieved almost six times the speed performance of them with the nearly identical memory consumption. Besides, it was observed

that our algorithm had the scalability in terms of data size $n$ and the number of clusters $k$. Thus the proposed algorithm is more suitable for large-scale and high-dimensional data sets with potentially numerous classes.

There remain the two directions as the future work. One is to incorporate the existing algorithms using the distance lower bound based on the triangle inequality or the Cauchy-Schwarz inequality into our framework if this leads to the improvement of the clustering performance. The other is to design a more effective space where the summable lower bound on the squared distance becomes tighter with smaller dimensionality. Then a method for determining the dimensionality should be also developed.

## Acknowledgments

This work was partly supported by JSPS KAKENHI Grant Number JP17K00159.

### References

[1] M. Mahajan, P. Nimbhorkar, and K. Varadarajan, "The planar $k$-means problem is NP-hard," Theoretical Computer Science, vol.442, pp.13–21, 2012.
[2] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, "NP-hardness of Euclidean sum-of-squares clustering," Machine Learning, vol.75, no.2, pp.245–248, 2009.
[3] O. du Merle, P. Hansen, B. Jaumard, and N. Mladenovic, "An interior point algorithm for minimum sum-of-squares clustering," SIAM J. Sci. Comput., vol.21, no.4, pp.1485–1505, 2000.
[4] B. Babaki, T. Guns, and S. Nijssen, "Constrained clustering using column generation," Proc. 11th Int. Conf. Integration of AI and OR Techniques in Constraint Programming (CPAIOR), ed. H. Simonis, Lecture Notes in Computer Science, vol.8451, pp.438–454, Springer, Cham, 2014.
[5] A. Kumar, Y. Sabharwal, and S. Sen, "Linear-time approximation schemes for clustering problems in any dimensions," J. ACM, vol.57, no.2, article 5, 2010.
[6] C. Boutsidis, A. Zouzias, M.W. Mahoney, and P. Drineas, "Randomized dimensionality reduction for $k$-means clustering," IEEE Trans. Inf. Theory, vol.61, no.2, pp.1045–1062, 2015.
[7] M.B. Cohen, S. Elder, C. Musco, C. Musco, and M. Persu, "Dimensionality reduction for k-means clustering and low rank approximation," Proc. 47th Annu. ACM Symp. Theory Comput. (STOC), pp.163–172, 2015.
[8] S.P. Lloyd, "Least squares quantization in PCM," IEEE Trans. Inf. Theory, vol.28, no.2, pp.129–137, 1982.
[9] J.B. MacQueen, "Some methods for clasiffication and analysis of multivariate observations," Proc. 5th Berkeley Symp. Mathematical Statistics and Probability, pp.281–297, 1967.
[10] D. Sculley, "Web-scale k-means clustering," Proc. 19th ACM Int. Conf. World Wide Web (WWW), pp.1177–1178, 2010.
[11] S. Har-Peled and B. Sadri, "How fast is the $k$-means method?," Algorithmica, vol.41, no.3, pp.185–202, 2005.
[12] D. Arthur and S. Vassilvitskii, "How slow is the $k$-means method?," Proc. 22nd ACM Symp. Comput. Geometry (SoCG), pp.144–153, 2006.
[13] R. Ostrovsky, Y. Rabani, L.J. Schulman, and C. Swamy, "The effectiveness of Lloyd-type methods for the k-means problem," Proc. 47th Annu. IEEE Symp. Found. Computer Science (FOCS), pp.165–176, 2006.
[14] D. Arthur and S. Vassilvitskii, "$k$-means++: The advantages of careful seeding," Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA), pp.1027–1035, 2007.
[15] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," Proc. 38th Int. Conf. VLDB Endowment, vol.5, no.7, pp.622–633, 2012.
[16] O. Bachem, M. Lucic, H. Hassani, and A. Krause, "Fast and provably good seedings for k-means," Proc. Advances in Neural Information Processing Systems (NIPS), Track1 Clustering, Dec. 6, 2016.
[17] C. Elkan, "Using the triangle inequality to accelerate k-means," Proc. 20th Int. Conf. Machine Learning (ICML), pp.147–153, 2003.
[18] G. Hamerly, "Making $k$-means even faster," Proc. SIAM Int. Conf. Data Mining (SDM), pp.130–140, 2010.
[19] J. Drake and G. Hamerly, "Accelerated k-means with adaptive distance bounds," Proc. 5th NIPS Workshop on Optimization for Machine Learning, Dec. 8, 2012.
[20] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. Mytkowicz, "Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup," Proc. 32nd Int. Conf. Machine Learning (ICML), pp.579–587, 2015.
[21] G. Hamerly and J. Drake, "Accelerating Lloyd's algorithm for $k$-means clustering," Partitional Clustering Algorithms, ed. M.E. Celebi, ch. 2, pp.41–78, Springer, Cham, 2015.
[22] P. Ryšavý and G. Hamerly, "Geometric methods to accelerate $k$-means algorithms," Proc. SIAM Int. Conf. Data Mining (SDM), pp.324–332, 2016.
[23] T. Hattori, K. Aoyama, K. Saito, T. Ikeda, and E. Kobayashi, "Pivot-based k-means algorithm for numerous-class data sets," Proc. SIAM Int. Conf. Data Mining (SDM), pp.333–341, 2016.
[24] T. Bottesch, T. Bühler, and M. Kächele, "Speeding up k-means by approximating Euclidean distances via block vectors," Proc. 33rd Int. Conf. Machine Learning (ICML), Clustering, June 20, 2016.
[25] J. Newling and F. Fleuret, "Fast k-means with accurate bounds," Proc. 33rd Int. Conf. Machine Learning (ICML), Clustering, June 20, 2016.
[26] T. Kaukoranta, P. Fränti, and O. Nevalainen, "A fast exact GLA based on code vector activity detection," IEEE Trans. Image Process., vol.9, no.8, pp.1337–1342, 2000.
[27] N. Ailon and B. Chazelle, "Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform," Proc. 38th Annu. ACM Symp. Theory Comput. (STOC), pp.557–563, 2006.
[28] N. Ailon and B. Chazelle, "The fast Johnson-Lindenstrauss transform and approximate nearest neighbors," SIAM J. Comput., vol.39, no.1, pp.302–322, 2009.
[29] A. Kumar and R. Kannan, "Clustering with spectral norm and the k-means algorithm," Proc. 51st Annu. IEEE Symp. Found. Computer Science (FOCS), pp.299–308, 2010.
[30] A. Torralba, R. Fergus, and W.T. Freeman, "80 million tiny images: A large dataset for non-parametric object and scene recognition," IEEE Trans. Pattern Anal. Mach. Intell., vol.30, no.11, pp.1958–1970, 2008.
[31] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometry consistency for large scale image search," Proc. 10th European Conf. Comput. Vision (ECCV), Lecture Notes in Computer Science, vol.5302, pp.304–317, pp.304–317, Springer, Berlin, Heidelberg, 2008.
[32] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," Proc. IEEE Int. Conf. Comput. Vision Pattern Recogn. (CVPR), pp.1–8, 2008.
[33] D.G. Lowe, "Object recognition from local scale-invariant features," Proc. 7th IEEE Int. Conf. Comput. Vision (ICCV), pp.1150–1157, 1999.
[34] G.H. Golub and C.F.V. Loan, Matrix Computations, 4 ed., JHU Press, 2012.
[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein, "Graphlab: A new parallel framework for machine learning," Proc. Conf. Uncertainty in Artificial Intelligence (UAI), Miscellaneous, July 11, 2010.

**Kazuo Aoyama** received the B.E. degree in applied physics from Waseda University in 1986 and the M.E. degree from Tokyo Institute of Technology in 1988. In 1988, he joined NTT Laboratories, NTT Corporation. His research interests includes device modeling, LSI design methodology, computer architecture, data structure and algorithms, and machine learning.

**Kazumi Saito** received the B.S. degree in mathematics from Keio University in 1985 and the Ph.D. degree in engineering from University of Tokyo in 1998. In 1985, he joined the NTT Electrical Communication Laboratories. In 2007, he joined the University of Shizuoka. In 2018, he joined the Kanagawa University. He is currently a professor at the Faculty of Science. His current research interests are machine learning and statistical analysis of complex networks.

**Tetsuo Ikeda** received the Master of Computer Science from University of Tokyo in 1981, and the Doctor of Engineering from University of Tokyo in 2001. In 1981, he joined the NTT Electrical Communication Laboratories. In 2002, he joined the Iwate Prefectural University. In 2007, he joined the University of Shizuoka. He is currently a professor at the School of Management and Information. His current research interests are data engineering, information retrieval and GIS.