On Simplifying Large-Scale Spatial Vectors: Fast, Memory-Efficient, and Cost-Predictable k-means

Yushuai Ji[†], Zepeng Liu[†], Sheng Wang[†*], Yuan Sun[§], and Zhiyong Peng[†‡]
[†]School of Computer Science, Wuhan University
[‡]Big Data Institute, Wuhan University
[§]La Trobe Business School, La Trobe University
[yushuai, liuzp_063, swangcs, peng]@whu.edu.cn, yuan.sun@latrobe.edu.au

***Abstract*—The k-means algorithm can simplify large-scale spatial vectors, such as 2D geo-locations and 3D point clouds, to support fast analytics and learning. However, when processing large-scale datasets, existing k-means algorithms have been developed to achieve high performance with significant computational resources, such as memory and CPU usage time. These algorithms, though effective, are not well-suited for resource-constrained devices. In this paper, we propose a fast, memory-efficient, and cost-predictable k-means called Dask-means. We first accelerate k-means by designing a memory-efficient accelerator, which utilizes an optimized nearest neighbor search over a memory-tunable index to assign spatial vectors to clusters in batches. We then design a lightweight cost estimator to predict the memory cost and runtime of the k-means task, allowing it to request appropriate memory from devices or adjust the accelerator's required space to meet memory constraints, and ensure sufficient CPU time for running k-means. Experiments show that when simplifying datasets with scale such as 10^6, Dask-means uses less than 30MB of memory, achieves over 168 times speedup compared to the widely-used Lloyd's algorithm. We also validate Dask-means on mobile devices, where it demonstrates significant speedup and low memory cost compared to other state-of-the-art (SOTA) k-means algorithms. Our cost estimator estimates the memory cost with a difference of less than 3% from the actual ones and predicts runtime with an MSE up to 33.3% lower than SOTA methods.**

I. Introduction

Sensors, such as GPS and lidar, are commonly found in resource-constrained devices like autonomous vehicles (AVs) [60] and drones [25], [53]. They generate a wealth of spatial vectors [28], which are geometric representations of spatial objects, for example, 2D trajectory datasets and 3D point cloud datasets collected from GPS and lidar deployed in AVs [15].

These spatial vectors can be widely applied on resource-constrained devices in visualization and learning tasks such as classification [36] and segmentation [30], [49]. For technologies such as 3D object recognition [40], processing all cloud points is unnecessary since a dense point cloud contains many redundant spatial vectors, and processing all spatial vectors significantly increases storage and processing costs.

The most straightforward way to tackle this limitation is by simplifying a dataset, and the two most widely used simplifying methods are random selection [30] and clustering algorithms (e.g., k-means [39]). However, randomly selecting a subset of spatial vectors from the dataset may not be evenly distributed and, therefore, cannot accurately represent the dataset [51]. This makes k-means a better choice and widely used to simplify point clouds [41], [55], [65], [66] and summarize datasets [9], [14], [34]. For example, as shown in Fig. 1, the spatial vectors selected by our k-means algorithm are more evenly distributed than the randomly selected ones.

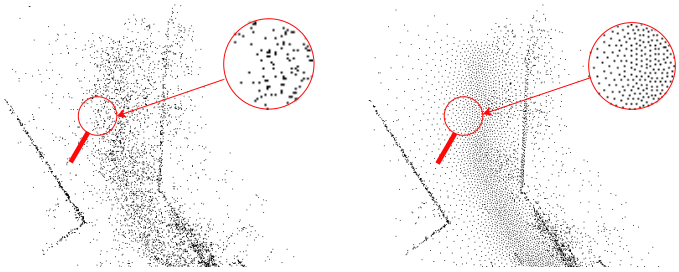


Fig. 1. The simplified point clouds by random sampling (left) and our k-means clustering algorithm (right).

However, as the scales of datasets expand significantly and reach millions, technologies such as object detection in AVs [40] still require the k-means algorithm to be highly efficient. Existing k-means algorithms have been developed to achieve efficiency using significant computational resources, such as memory. These algorithms, though effective, are not well-suited for resource-constrained devices. For example, Google Coral [4] and Raspberry Pi [12] typically have 4GB of memory. This raises a critical research question: *how to design a fast and memory-efficient k-means algorithm to simplify large-scale spatial vectors on resource-constrained devices?*

To answer the question, we need to tackle two key challenges: 1) high space cost for storing bounds and indexes to reduce unnecessary distance computations for accelerating k-means tasks, and 2) degradation of the efficiency of k-means algorithms due to limited memory resources for storing information, such as indexes, and insufficient CPU resources for running it. Various techniques have been proposed to address these challenges, but they still have shortcomings when applied to resource-constrained devices, as summarized below.

Sacrificing Substantial Space for Accelerating. Existing k-means algorithms have high time or space complexity, making them inapplicable to simplify large-scale datasets on resource-constrained devices. Memory-efficient k-means algorithms such as Lloyd's algorithm [39], Index [44], Hamerly [26], and NoBound [64], are computationally slow,

*Sheng Wang is the corresponding author.

especially for clustering tasks that involve both a large number of spatial vectors and clusters. Although there are algorithms that trade off memory for speed (e.g., [21], [46], [52]), they require more memory than is available on these devices.

Inaccruate Memory & Runtime Estimation. For memory cost estimation, existing methods are either designed for programs like Java-like projects [10], which cannot be directly applied to k−means, or are tailored for a few machine learning (ML) models [7], [13], [23], such as neural networks. For runtime estimation, most methods [18], [20], [57] involve training an ML model to predict runtime based on features provided by k−means. However, existing methods often lead to high training overhead because the models require a large number of samples to be generated for model training, which also requires substantial computational time.

In this paper, we propose Dask−means, a fast, memory-efficient, and cost-predictable dataset simplification k-means algorithm for large-scale spatial vectors. To accelerate Lloyd's algorithm without costing substantial memory, we build indexes on spatial vectors and cluster centroids. The index supports optimized k Nearest Neighbor (kNN[1]) search to assign spatial vectors to a cluster efficiently. To predict the memory cost and runtime of Dask−means accurately, we propose a lightweight cost estimator, which can analyze the space cost of the pruning mechanism, and estimate the overall runtime by predicting the iteration number and runtime of each iteration separately. Overall, our main contributions are:

- We design pruning mechanisms that apply a three-pronged optimized kNN search on the centroid index to batch prune spatial vectors to accelerate k-means tasks (see Section IV).
- We predict the memory cost of k-means tasks by building a mapping function between the dataset and the index, and estimate runtime by separately predicting the iteration number and the runtime of each iteration (see Section V).
- Experiments on the tested datasets show that Dask−means accelerates Lloyd's algorithm by up to 168 times. Our cost estimator predicts memory cost with a difference of less than 3% from the actual values and estimates runtime with an MSE 33.3% lower than SOTA models (see Section VI).

II. Background and Preliminaries

A. Notations

We use different text formatting styles to represent mathematical concepts: plain letters for scalars, bold letters for vectors, capitalized letters for objects, and bold capitalized letters for a set containing vectors. For example, x stands for a scalar, $\mathbf{p}$ represents a spatial vector, N denotes an index node, and $\mathbf{D}$ represents a dataset. Without loss of generality, we denote the d-dimensional Euclidean space as $\mathbb{R}^d$, the set of positive real numbers as $\mathbb{R}^+$, and the set of positive integers as $\mathbb{Z}^+$. Moreover, we use $||\cdot||$ as the Euclidean norm. The notation details are presented in Table I.

[1]We use k to differentiate from k in k-means as they represent different concepts.

TABLE I
Summary of Notations.

Notation	Description
$n \in \mathbb{Z}^+$	The dataset size
$\mathbf{p} = (x_1, x_2, ..., x_d) \in \mathbb{R}^d$	The spatial vector
$\mathbf{D} = \{\mathbf{p}_i\}_{i=1}^n \in \mathbb{R}^{n \times d}$	The dataset
$k \in \mathbb{Z}^+$	The number of clusters
$S = \{S_1, S_2, \cdots, S_k\}$	k exclusive subsets
$\mathbf{c}_j \in \mathbb{R}^d$	The mean of the spatial vectors in S_j
N	The spatial vector index node (ball node)
$\mathbf{p}^*$	$\mathbf{p}^*$ is the pivot of a node N
r	The radius of N
N_C	The centroids index node
$m \in \mathbb{R}$	The available memory
$f \in \mathbb{Z}^+$	The leaf node capacity
$t \in \mathbb{R}^+$	The runtime of k−means
$q \in \mathbb{Z}^+$	The maximum number of iterations

B. Definition of k-means

The k-means is a bivariate optimization problem. Given a dataset $\mathbf{D} = \{\mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_n\} \in \mathbb{R}^{n \times d}$, k-means aims to partition $\mathbf{D}$ into k exclusive subsets $S = \{S_1, S_2, \cdots, S_k\}$ to minimize the *Sum of Squared Error*:

$$\underset{S}{\operatorname{argmin}} \sum_{j=1}^{k} \sum_{\mathbf{p} \in S_j} \|\mathbf{p} - \mathbf{c}_j\|^2, \tag{1}$$

where the *centroid* $\mathbf{c}_j = \frac{1}{|S_j|} \sum_{\mathbf{p} \in S_j} \mathbf{p}$ is the mean of spatial vectors in cluster S_j. Lloyd's algorithm [39] is one of the most widely used methods to solve the k-means problem by assigning each spatial vector to its nearest centroid and iteratively refining the centroids. The algorithm requires computing $n \times k$ distances in the assignment phase of each iteration, which is computationally prohibitive for applying it to datasets where both n and k are large.

C. Accelerated Lloyd's Algorithms for k−means

We focus on techniques such as hardware-based algorithms, index-based algorithms, and sequential algorithms to speed up k-means, as they yield the same results as Lloyd's algorithm.

Hardware-based Algorithm. Several researchers [35], [38] design parallel k−means algorithm for GPUs. For instance, Li et al. [38] develop a parallel k−means using a general-purpose parallel programming model. Although the methods are applicable to edge devices, they require significant computational resources, making them unsuitable for resource-constrained devices. Others focus on accelerating k−means on specific processors, such as CPU-FPGA [8], FPGA [59], and heterogeneous many-core supercomputers [67]. Additionally, Bender et al. [11] use two-level memory systems to speed up k-means. However, they lack generality, as many edge devices do not have this type of processor or storage system.

Index-based Algorithm. Instead of assigning spatial vectors one by one, Moore et al. [44] proposed an index that stores spatial vectors in a hierarchical tree structure called *Ball-tree* [47]. The *spatial vector index* can avoid the distance computation between a centroid and a set of spatial vectors.

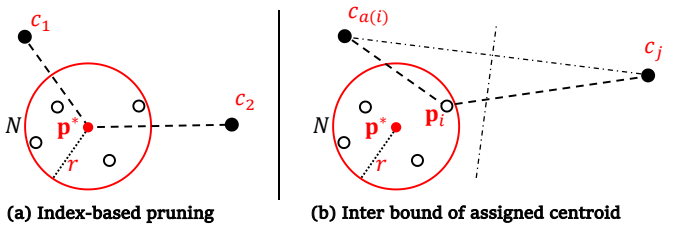


Fig. 2. Pruning using ball node and inter bound.

For example, given two centroids $\mathbf{c}_1$ and $\mathbf{c}_2$, all the spatial vectors in a ball node N are closer to centroid $\mathbf{c}_1$ than $\mathbf{c}_2$ if

$$\|\mathbf{p}^* - \mathbf{c}_1\| + r < \|\mathbf{p}^* - \mathbf{c}_2\| - r, \tag{2}$$

where $\mathbf{p}^*$ is the pivot of a node N that bounds all spatial vectors within a radius r, as shown in Fig. 2(a). The index structure requires extra memory cost, which is proportional to the number of nodes in the Ball-tree. The drawback of the index-based algorithm is that it scans all cetroids one by one, which needs k distance computations, to assign the spatial vectors in an index node to their nearest centroid in batch.

Sequential Algorithms. As shown in Fig. 2(b), to check whether a spatial vector $\mathbf{p}_i$ belongs to a cluster with centroid $\mathbf{c}_j$, Elkan et al. [21] store the lower bound on the distance from $\mathbf{p}_i$ to $\mathbf{c}_j$, which needs $O(nk)$ memory for all pairs of spatial vectors and centroids. Firstly, an *inter bound* is derived as $\|\mathbf{c}_{a(i)} - \mathbf{c}_j\|$, where $a(i)$ denotes the id of the centroid that $\mathbf{p}_i$ was assigned in the previous iteration.[2] If $\|\mathbf{p}_i - \mathbf{c}_{a(i)}\| < \|\mathbf{c}_{a(i)} - \mathbf{c}_j\|/2$, then $\mathbf{c}_j$ can be pruned. As storing all bounds to every centroid uses much space, Hamerly et al. [26] proposed to choose the minimum one as the only centroid inter bound:

$$\mathbf{cb}[a(i)] = \min_{\mathbf{c}_j \in \mathbf{C} \& j \neq a(i)} \|\mathbf{c}_{a(i)} - \mathbf{c}_j\|. \tag{3}$$

Elkan et al. [21] stored the computed distances to accelerate the next iteration, and computes the moving distance of each centroid $\Delta[j] = \|\mathbf{c}_j - \mathbf{c}_j^{'}\|$, also called *drift*, to estimate the lower bound using triangle inequality: $\|\mathbf{p}_i - \mathbf{c}_j\| \geq \|\mathbf{p}_i - \mathbf{c}_j^{'}\| - \|\mathbf{c}_j - \mathbf{c}_j^{'}\|$, where $\mathbf{c}_j^{'}$ denotes the position of centroid j in the previous iteration. Drake [19], Hamerly and Drake [27], Newling and Fleuret [45], Ryšavý and Hamerly [52] proposed even tighter bounds, but they all require substantial memory and become prohibitively costly when k is large. Furthermore, since the bounds must be updated across iterations, this overhead slows down the clustering process, making these algorithms unsuitable for simplifying large-scale point clouds.

Several memory-efficient sequential algorithms were proposed, including [26] described above; [19] that stores $\frac{k}{4}$ minimum lower bounds; Yinyang [17] that divides k centroids into $\frac{k}{10}$ groups and each group has one lower bound; and Dual-tree [50] that extends the single upper and lower bound of [26] to index-based algorithm [44], and use index to group centroids [17] for pruning centroids in batch. In contrast to [26], Dual-tree [50] needs extra memory for a spatial vector index, where each spatial vector and node maintains two bounds and other pruning information.

Moreover, Xia et al [64] accelerate k−`means` with no bound (short as NoBound), but it needs to create a centroid distance matrix ($k \cdot k$) in each iteration for pruning centroids outside a radius range. For an overview of all existing accelerating algorithms, we suggest readers refer to Section 4.2 of our recent evaluation paper [61]. As shown in Table II, we compare existing memory-efficient algorithms from five new perspectives.

[2]To facilitate our illustration, each cluster is simply represented by its centroid if no ambiguity is caused.

TABLE II
COMPARISON WITH MEMORY-EFFICIENT SEQUENTIAL ALGORITHMS.

Algorithm	Prune Centroids In-batch	Update-free Bounds	Assign Points In-batch	Memory-cost Tunable	Run-time Predictable
`Hamerly` [26]	✓	×	×	×	×
`Drake` [19]	✓	×	×	✓	×
`Yinyang` [17]	✓	×	×	✓	×
`Dual-tree` [50]	✓	×	✓	×	×
`NoBound` [64]	×	✓	×	×	×
`Dask-means`	✓	✓	✓	✓	✓

D. Cost Estimator

Memory Cost Estimation. Several technologies [10], [29], [58] have been proposed to predict the memory cost for programs such as Java-like programs [10], and can be applied in k−`means`. For example, Verbauwhede et al. [58] estimate the memory cost of digital signal processing programs by modeling array dependencies and execution sequences as an integer linear programming (ILP) problem, which is then solved using an ILP solver. Albert et al. [10] introduce a parametric technology to infer the memory cost of Java-like programs by analyzing object lifetimes. Heo et al. [29] propose a resource-aware, flow-sensitive analysis towards estimating memory cost using online abstraction coarsening. However, they can only predict the memory cost for k−`means` written in specific programming languages.

TensorFlow [7] and several ML model performance analysis works [13] estimate memory cost by summarizing the parameters, dataset, and outputs. However, they are just a subset of the whole memory cost. Moreover, TensorFlow cannot analyze the memory costs related to indexes and bounds, which can affect the final memory cost. Additionally, Gao et al. [23] propose DNNMem, which calculates the memory cost of the computation graph and the deep learning (DL) model runtime. However, it only works for DL models.

Runtime Estimation. A bunch of models have been proposed for estimating runtime, but they are time-consuming. Models like Bayes DistNet [57], XGBoost [24], and AutoML [43] require an impractical number of training samples before they can positively impact prediction time. Generating tens of thousands of k-means samples for training could take several hours or even days for a resource-constrained device. Eggensperger et al. [20] propose DisNet to predict the runtime accurately by a neural network. However, these ML models

repeatedly perform forward propagation, loss calculation, and backpropagation over multiple epochs until the neural network reaches satisfactory performance, which is time-consuming.

Alternative methods [22], [31], [37], [43] for predicting k-means runtime use linear regression, which is time efficient. For example, Leyton-Brown et al. [37] use ridge regression to predict runtime. Fan et al. [22] predict runtime using linear regression by data censoring. Hutter [31] and Mohr [43] compare various regression models in terms of training time, prediction time, and prediction error. However, their analysis applies to general ML models and shows low accuracy when predicting for the k-means tasks (see Section VI).

Moreover, several models [22], [56] use the posterior information during task execution to adjust the predicted runtime. For example, Fan et al. [22] propose TRIP to reduce underestimation rates of prediction by incorporating elastic net regularization (two penalties) into the linear regression model. Similarly, Tang et al. [56] improve runtime accuracy by multiplying a user-supplied runtime estimate with an adjusting parameter. However, adding penalties or an adjusting parameter requires extensive experimentation to find the optimal values, which is not practical when the dataset or setting parameters in ML models change.

III. Framework of Dask-means

In this section, we introduce `Dask-means`, which can accelerate k-means significantly, especially on resource-constrained devices. As shown in Fig. 3, `Dask-means` consists of two modules described below.

Memory-efficient Accelerator. Recall that the index-based algorithm [44] requires a time-intensive scan of all cluster centroids when assigning spatial vectors to clusters. Sequential algorithms [19], [21], in contrast, require substantial memory to maintain bounds for spatial vectors and nodes across iterations for centroid pruning. To avoid these limitations, as shown in Fig. 3(a), we construct an index on spatial vectors, denoted as spatial vector index. This index leverages nodes to represent a group of spatial vectors, thus avoiding distance computations between a centroid and a batch of spatial vectors. Then, we design an optimized kNN search over the index built on centroids (namely the centroid index) and maintain the proposed inter bound to prune unnecessary computations.

Lightweight Cost Estimator. As shown in Fig. 3(b), we propose a lightweight cost estimator to predict the memory cost and the runtime for k-means. We first propose a memory estimate method to predict memory cost by building a mapping function between the dataset and the index. This method also allows us to adjust the hyperparameters of the proposed accelerator to build a memory-tunable index that accelerates the k-means task. We estimate the runtime by separately predicting the iteration number and each iteration's runtime. Notably, we then extract posterior information from the last iteration of the k-means task to adjust the predicted runtime.

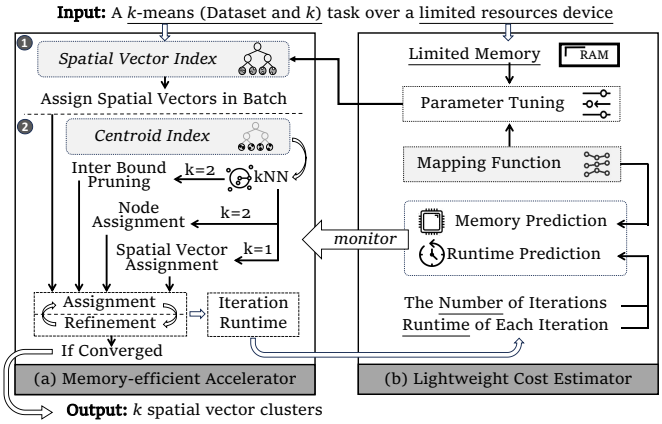


Fig. 3. Framework of `Dask-means`.

IV. Memory-efficient Accelerator

A. Pruning Mechanisms

We design pruning mechanisms that apply a three-pronged optimized kNN over the centroid index to batch prune nodes and spatial vectors, thus accelerating k-means without the need to store bounds for spatial vectors. We first prune distance computations between centroids and a set of spatial vectors by applying kNN to find the nearest centroids of an index node (or a spatial vector), with the kNN bounds inherited from parent nodes. We then use kNN to search for the nearest centroids of the target one to avoid scanning all centroids. Notably, we add two drifts in estimating the inter bound to accelerate kNN.

Indexes on Spatial Vectors and Centroids. We build a Ball-tree index structure for spatial vectors with the root node denoted as R, and another Ball-tree index structure for the clusters' centroids with the root node denoted as R_C. In Fig. 4, we show a toy example where an index node of spatial vectors (the big red circle on the left) covers its two child nodes (the small red dotted circles); the black circle on the right denotes a centroid node which covers six centroids. Note that the Ball-tree for spatial vectors needs to be built only once, while the Ball-tree for centroids must be constructed in each iteration of the algorithm as the centroids move.

The nodes (denoted as N and N_C) in the spatial vector and centroid indexes are slightly different; both of them need to store the pivot vector $\mathbf{p}^*$ (the mean of all covered spatial vectors/centroids in the node) and radius r to bound child nodes (or spatial vectors if it is a leaf node with capacity f). But each node N of the spatial vector index also stores the number of spatial vectors it covers, denoted as $|N|$, e.g., $|N| = 8$ in Fig. 4.

Furthermore, each index node N (or spatial vector $\mathbf{p}_i$) stores an integer $a(N)$ (or $a(i)$) to denote the id of the cluster it was assigned to in the previous iteration. For the current iteration, we can compute the distance between a spatial vector $\mathbf{p}_i$ and the centroid of its previous cluster $\mathbf{c}_{a(i)}$. If the distance is smaller than the inter bound, i.e.,

$$\|\mathbf{p}_i - \mathbf{c}_{a(i)}\| < \frac{\mathbf{cb}[a(i)]}{2}, \tag{4}$$

where $\mathbf{cb}[a(i)]$ is defined in Eq. (3), $\mathbf{p}_i$ still belongs to the cluster $a(i)$ in the current iteration [21].

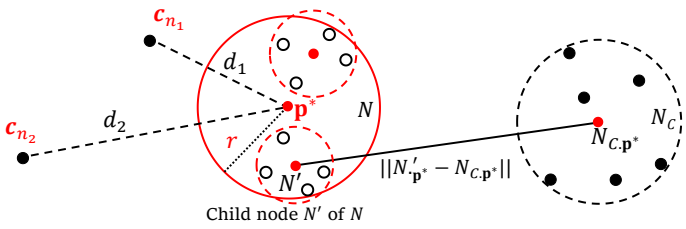


Fig. 4. Pruning with a single indexing tree, where a spatial vector node N contains two child nodes; pruning with centroid index node N_C, where $\mathbf{c}_{n_1}$ and $\mathbf{c}_{n_2}$ represent the two nearest centroids to N's pivot ($\mathbf{p}^*$), with the corresponding distances d_1 and d_2 (where $d_2 > d_1$); $N.\mathbf{p}^*$ refers to the pivot of N' and $N_C.\mathbf{p}$ refers to the pivot of N_C.

Similarly, if a node N was assigned to the cluster $a(N)$ in the previous iteration, we can compute the distance between N and the centroid of cluster $a(N)$, which denotes $\mathbf{c}_{a(N)}$. If the upper bound on the distance between N's points and $\mathbf{c}_{a(N)}$ is smaller than the inter bound, i.e.,

$$\|N.\mathbf{p}^* - \mathbf{c}_{a(N)}\| + N.r < \frac{\mathbf{cb}[a(N)]}{2}. \tag{5}$$

Then all the spatial vectors in the node N can be directly assigned to cluster $a(N)$ in the current iteration; otherwise, we search for the two nearest centroids, $\mathbf{c}_{n_1}$ and $\mathbf{c}_{n_2}$, of N's pivot, and denote the corresponding distances as d_1 and d_2, where $d_2 > d_1$. If the distance gap $d_2 - d_1$ is bigger than $2N.r$, all the spatial vectors in the node N can be assigned to the cluster with centroid $\mathbf{c}_{n_1}$:

$$d_2 - N.r \geq d_1 + N.r \rightarrow a(N) = n_1. \tag{6}$$

If the node N still cannot be assigned, we split N into two (e.g., the two small red dotted circles in Fig. 4) and repeat the above process for each child node. If the node N is a leaf, we search for the nearest centroid of each spatial vector in N and assign the spatial vectors to their nearest centroids.

The bottleneck in the above process is searching for the nearest centroids of an index node (or a spatial vector). A naïve approach would be to compute the distance from the index node (or spatial vector) to each of the centroids, which is computationally expensive. In the following, we use kNN to search for the nearest centroids efficiently.

Using kNN to Search for (Two) Nearest Centroids. To find the two nearest centroids of an index node, checking if all k centroids are not pruned by the inter bound has a worst-case time complexity of $O(k)$. Here, we use the kNN search method based on the index structure of the centroids. This method reduces the time complexity to $O(\log_2 k)$ on average, by pruning a set of centroids in a centroid index node if its lower bound to the query vector $\mathbf{q}$ is larger than the current results held in a priority queue H. Initially, H is filled with arbitrarily large numbers if no result has been found. We can prune certain centroid nodes in advance by deriving a tight upper bound on the distance from the query vector $\mathbf{q}$ to its two nearest centroids, as detailed below.

kNN Bounds Inherited from Parent Nodes. To further prune centroid nodes during the kNN search, we compute an upper bound on the distance from the pivot of a node ($N^{'}.\mathbf{p}^*$) to its two nearest neighbors:

$$\begin{aligned} ub_1(N^{'}.\mathbf{p}^*) &= d_1(N.\mathbf{p}^*) + N.r, \\ ub_2(N^{'}.\mathbf{p}^*) &= d_2(N.\mathbf{p}^*) + N.r, \end{aligned} \tag{7}$$

where N is the parent node of $N^{'}$; $d_1(N.\mathbf{p}^*)$ and $d_2(N.\mathbf{p}^*)$ are the distances from $N.\mathbf{p}^*$ to its two nearest centroids, as shown in Fig. 4. When searching for the two nearest centroids of $N^{'}.\mathbf{p}^*$, we can prune a centroid node N_C, if the lower bound on the distance between the centroids in N_C and $N^{'}.\mathbf{p}^*$ is larger than $ub_2(N^{'}.\mathbf{p}^*)$,

$$\|N^{'}.\mathbf{p}^* - N_C.\mathbf{p}^*\| - N_C.r > ub_2(N^{'}.\mathbf{p}^*). \tag{8}$$

If we search for the nearest centroid of $N^{'}.\mathbf{p}^*$, we can replace $ub_2(N^{'}.\mathbf{p}^*)$ with $ub_1(N^{'}.\mathbf{p}^*)$ in the above inequality for pruning centroid nodes.

Accelerating Inter Bound Computation. To compute a tight inter bound for a centroid $\mathbf{c}_j$, we need to find the minimum distance from $\mathbf{c}_j$ to other centroids. In contrast to [64] which computes the pairwise distances between centroids, we use kNN to search for the nearest centroid of $\mathbf{c}_j$ efficiently. We also derive an upper bound on the distance from $\mathbf{c}_j$ to its nearest centroid to further prune centroid nodes. Let $\mathbf{cb}[j]$ denote the distance from $\mathbf{c}_j$ to its nearest centroid in the previous iteration of the algorithm; $\Delta[j]$ denote the drift of $\mathbf{c}_j$; and $\max(\Delta)$ denote the maximum drift of $\mathbf{c}_j$'s nearest centroid, the upper bound is defined as:

$$ub = \mathbf{cb}[j] + \Delta[j] + \max(\Delta). \tag{9}$$

In summary, we have used kNN to accelerate various components of our approach: 1) inter bound computation with $\mathtt{k} = 2$; 2) node assignment with $\mathtt{k} = 2$; and 3) spatial vector assignment with $\mathtt{k} = 1$. All of them can be accelerated by an update-free upper bound from parent nodes.

B. Algorithm Design

Algorithm 1 shows the process of pruning mechanism over `Dask-means`. After creating the spatial vector index on $\mathbf{D}$ and the centroid index on the initial k centroids, `Dask-means` uses recursion to traverse the spatial vector index and centroid index to conduct the assignment with a bound-armed kNN search. After assigning all the spatial vectors to their nearest centroid, it refines the centroids and checks whether any of the centroids move; if so, it continues. To refine the new centroid efficiently, `Dask-means` maintains a dynamic sum vector $\mathbf{sv}(j)$ for each cluster with a unique id j. It updates $\mathbf{sv}(j)$ when a spatial vector $\mathbf{p}$ moves in ($\mathbf{sv}(j) = \mathbf{sv}(j) + \mathbf{p}$) or out ($\mathbf{sv}(j) = \mathbf{sv}(j) - \mathbf{p}$) (see Lines 25 and 38), where $\mathbf{p}$ can be replaced by $N.\mathbf{p}^* \cdot |N|$ if a node N moves. Finally, a new centroid $\mathbf{c}_j$ can be computed by $\frac{\mathbf{sv}(j)}{|S_j|}$ in Line 12. Our algorithm can be easily implemented with two recursive traversal functions presented below. Notably, We analyze the time complexity of the proposed pruning mechanism in Appendixes VIII-A due to the page limitation.

Algorithm 1: `Accelerator`(k, $\mathbf{D}$, M)

Input: k: the number of clusters, $\mathbf{D}$: dataset, M: available main memory
Output: k centroids: $\mathbf{C} = \{c_1, \ldots, c_k\}$

Create Ball-tree on $\mathbf{D}$ according to M, get the root node R;
Initialize centroids $\mathbf{C}$ according to k and $\mathbf{D}$;
$it \leftarrow 1$;
while *did not converge* **do**
  Create Ball-tree on $\mathbf{C}$ and get root node R_C;
  foreach $\mathbf{c}_j \in \mathbf{C}$ **do**
    Set ub as ∞ if $it = 1$ else set ub using Eq. (9);
    $[Q, H] \leftarrow$ **kNN**$(2, \mathbf{c}_j, R_C, ub)$;
    $\mathbf{cb}[j] \leftarrow H[2]$; `// defined in Eq. (3)`
  $[S, \mathbf{sv}] \leftarrow$**Assign**$(R, R_C, \infty)$;
  foreach $\mathbf{c}_j \in \mathbf{C}$ **do**
    Refine centroid: $\mathbf{c}_j \leftarrow \frac{\mathbf{sv}(j)}{|S_j|}$, and compute $\Delta[j]$;
  $it \leftarrow it + 1$;
return $\mathbf{C}$;

Function Assign (N, R_C, ub) **:**
  Input: N: node (or spatial vector) to be assigned, R_C: root node of centroid index, ub: upper bound
  Output: S: cluster with nodes & spatial vectors, $\mathbf{sv}$: sum vector of cluster S
  ---
  if N *is node* **then**
    if $\|N.\mathbf{p}^* - \mathbf{c}_{a(N)}\| + N.r < \frac{\mathbf{cb}[a(N)]}{2}$ **then**
      Assign node N to cluster $a(N)$;
      return
    $[Q, H] \leftarrow$ **kNN**$(2, N.\mathbf{p}^*, R_C, ub)$;
    $\mathbf{c}_{n_1}, \mathbf{c}_{n_2}, d_1, d_2 \leftarrow Q[1], Q[2], H[1], H[2]$;
    if $(d_2 - d_1) > 2 * N.r$ **then**
      if $a(N) \neq n_1$ **then**
        Update $S_{a(N)}$, $\mathbf{sv}(a(N))$ and S_{n_1}, $\mathbf{sv}(n_1)$;
      Assign node N to cluster n_1;
      return
    else
      foreach *child node or spatial vector* N' *of* N **do**
        Assign$(N', R_C, d_2 + N.r)$;
  else
    if $\|N - \mathbf{c}_{a(N)}\| < \frac{\mathbf{cb}[a(N)]}{2}$ **then**
      Assign spatial vector N to cluster $a(N)$;
      return
    $[Q, H] \leftarrow$ **kNN**$(1, N, R_C, ub)$;
    $\mathbf{c}_{n_1} \leftarrow Q[1]$;
    if $a(N) \neq n_1$ **then**
      Update $S_{a(N)}$, $\mathbf{sv}(a(N))$, S_{n_1}, and $\mathbf{sv}(n_1)$;
    Assign spatial vector N to cluster n_1;
  return $[S, \mathbf{sv}]$;

Function kNN (k, $\mathbf{q}$, N_C, ub) **:**
  Input: k: the number of neighbors, $\mathbf{q}$: query vector, N_C: centroid node, ub: upper bound distance to the nearest centroid
  Output: Q: a priority queue holding kNN, H: a priority queue holding distances of the kNN

  Initialize the distances in the priority queue H to ub;
  if N_C *is a leaf node* **then**
    foreach *spatial vector* $\mathbf{p}_i \in N_C$ **do**
      $d \leftarrow \|\mathbf{p}_i - \mathbf{q}\|$;
      if $d < H[k]$ **then**
        Update Q and H using $\mathbf{p}_i$ and d;
  else
    foreach *child node* N_C' *of* N_C **do**
      $d \leftarrow \|\mathbf{q} - N_C'.\mathbf{p}^*\| - N_C'.r$;
      if $d < H[k]$ **then**
        kNN(k, $\mathbf{q}$, N_C', $H[k]$);
  return $[Q, H]$;

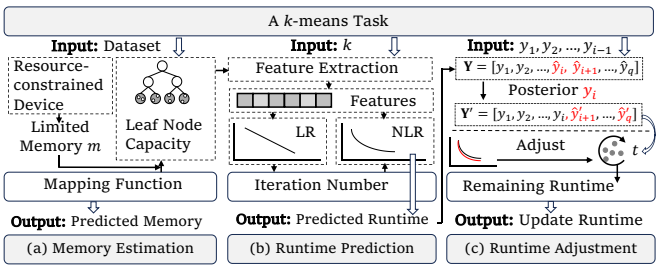


Fig. 5. Overview of our lightweight cost estimator, where y_i denotes the actual runtime for the i-th iteration ($i = 1, 2, \ldots, q$), and $\hat{y}_j$ represents the predicted runtime for the j-th iteration ($j = 1, 2, \ldots, q$).

Recursive Traversal on Spatial Vector Index. The function **Assign** traverses the spatial vector index to assign spatial vectors in batch or one by one. From the root node of the spatial vector index, the function searches for the two nearest centroids using **kNN**. After computing the distance gap $d_2 - d_1$, it checks whether the centroid can be pruned; if not, it sets the bound to its child nodes and performs another **Assign** operation recursively.

Recursive Traversal on Centroid Index. Function **kNN** recursively searches the centroid index to get one or two nearest centroids, using an upper bound ub to prune certain centroid nodes, and ub is initialized as the bound from the parent node and is updated with the latest centroid's distance found in H. A centroid node can be pruned if the lower bound on the distance from the query vector to each of the centroids in the node is greater than ub.

V. Lightweight Cost Estimator

Overview. We design a lightweight cost estimator to accelerate the k-`means` algorithm. Firstly, as shown in Fig. 5(a), we propose a memory estimation method to predict memory costs by building a mapping function between the index and the memory. This method also allows us to create memory-tunable indexes under memory constraints, thereby accelerating the k-means tasks. Secondly, as shown in Fig. 5(b), we predict the runtime of the k-means task by estimating the iteration number using a *linear regressor* (**LR**) and the runtime of each iteration using a *non-linear regressor* (**NLR**). Finally, as shown in Fig. 5(c), we monitor the progress of the k-means task by dynamically updating the remaining runtime. Specifically, we use posterior information from the last iteration of the k-means task to adjust the predicted runtime using a *Gaussian Process* (**GP**) with an asymmetric kernel function.

A. Memory Cost Estimation

The memory required for the k-`means` algorithm includes storing the dataset, maintaining the bounds, and the memory occupied by the indexing structure. Besides storing the dataset, the additional memory required is solely related to the indexing structure due to the fact that `Dask-means` does not maintain any bound. Hence, we estimate the memory cost of the index by establishing a mapping function between the leaf node capacity and the memory cost, denoted as m. For the index (using the balanced Ball-tree structure), each node includes a vector (a center of each partitioned sub-space, 3

dimensions), three floats (radius r, number of spatial vectors covered, cluster ID), and two pointers to child nodes (left and right) or a set of spatial vectors in leaf nodes (up to capacity f). Thus, we estimate the memory cost of a leaf node as $3+3+f$, and an internal node as $3+3+2=8$. Then the overall memory cost (number of floats) of all the nodes is:

$$\begin{aligned}\mathcal{M}(n,f) &= \lceil\frac{2n}{f}\rceil\cdot(6+f)+(\lceil\frac{2n}{f}\rceil-1)\cdot 8\\ &\approx 2n+\frac{28n}{f}-16,\end{aligned} \tag{10}$$

where $\lceil\frac{2n}{f}\rceil$ and $\lceil\frac{2n}{f}\rceil-1$ are the numbers of leaf nodes and internal nodes, respectively. This estimation is based on the assumption that each leaf node has $\frac{f}{2}$ spatial vectors, and the balanced Ball-tree with a height $\lceil\log_2\frac{2n}{f}\rceil$. It is worth noting that, in the real case, most nodes are not fully filled – they are half full on average. Therefore, we double the number of leaf nodes and internal nodes. Similarly, the centroid index also occupies $\mathcal{M}(k,f)$ units of memory.[3]

Moreover, the array used to indicate which cluster each spatial vector is assigned to will occupy n integers. This array stores the cluster IDs for the spatial vectors and helps identify which clusters they belong to. Hence, compared to Lloyd's algorithm, `Dask-means` requires additional memory, which can be described as follows:

$$m=\mathcal{M}(n,f)+\mathcal{M}(k,f)+n\approx(2+\frac{28}{f})(n+k)-32+n. \tag{11}$$

Based on this analysis, we can adjust the node capacity f according to the available memory m when clustering must be performed in resource-constrained devices,

Memory-tunable Index. A common index structure, such as the kd-tree and cover-tree used in [50], needs to store the leaf nodes as two spatial vectors at most, and the memory cost is at least $\mathcal{M}(n,2)$, which is much higher than our index, which utilizes the ball-true structure. Instead, we automatically configure the leaf node capacity f (the leaf node size of two index trees) based on the memory constraint, denoted as $m^{'}$. Specifically, under a given memory constraint $m^{'}$, f can be calculated using Eq. (11) as follows:

$$f\approx\frac{28(n+k)}{m^{'}-3n+32-2k}. \tag{12}$$

Overall, no bound is maintained for each spatial vector, and although we need to maintain an inter bound for each centroid in each iteration, the cost is negligible as $k\ll n$. Hence, the size of our index can be auto-configured according to the available memory via tuning the leaf node capacity f.

B. Runtime Prediction

1) Non-Linear Regressor: Unlike traditional methods [33], [57] that rely on training samples to directly predict k-means task runtime, denoted as t, our approach estimates the total runtime by predicting the iteration number and each iteration's runtime, respectively. Firstly, we estimate the k-means iteration number by using a linear regressor. Specifically, instead of using a positive integer to represent the iteration number, denoted as υ, we use a dummy array, denoted as $\mathbf{u}$, which is composed of 1s in the first υ positions and 0s in the remaining positions. For example, if $\upsilon=2$ and the maximum iteration number, denoted as q, is 5, then $\mathbf{u}=[1,1,0,0,0]$.

Then, we predict each iteration's runtime by designing a polynomial expression in a non-linear regressor. Finally, we calculate the total runtime as shown below:

$$t=\sum_{i=1}^{q}u_i\times\hat{y}_i, \tag{13}$$

where u_i is the value at the i-th position in $\mathbf{u}$, and y_i is the predicted runtime of the i-th iteration.

Iteration Number Estimation. We predict the iteration number υ using a linear regressor, such as multiple linear regression [54], which builds the function from the meta-feature to υ. Notably, extracting a meta-feature to describe (or represent) a dataset, such as n, k, and d, is not informative. Hence, in addition to these features, we also extract novel and more complex features to capture certain properties of data distribution based on our index. Specifically, the index construction actually conducts a more in-depth scan of the spatial vectors and reveals whether the spatial vectors assemble well in the space. Hence, the information can include tree depth, number of leaf nodes, number of internal nodes, and average spatial vectors per leaf node.

Building Non-linear Regressor. We design a non-linear regressor with $\mathbf{u}$ to model how meta-features, including n, k, d, and f, affect the runtime of k-means. We notice that the extracted meta-features are not independent. For example, n and f jointly determine the index structure, which affects the efficiency of kNN and affects the runtime of the assignment process in k-means. Therefore, we need to consider interaction terms (or *interaction feature*), such as nf. The regressor considering interaction feature can be expressed using a polynomial expression as follows:

$$\hat{y}_j=\sum_{i_1,i_2,\ldots,i_\lambda=0}^{\lambda}u_j\beta_{i_1i_2\ldots i_\lambda}x_{j1}^{i_1}x_{j2}^{i_2}\ldots x_{j\lambda}^{i_\lambda}+e, \tag{14}$$

where λ is the number of meta-features, $\beta_{i_1i_2\ldots i_\lambda}$ is the regression coefficient, $(x_{j1},\ x_{j2},\ \cdots,\ x_{j\lambda})$ are the meta-features obtained for the j-th iteration, and e is the residual term. Then the runtime of the k-means task, denoted as $\hat{y}$, can be represented as follows:

$$\hat{y}=\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}'\begin{bmatrix}u_1&0&\cdots&0\\0&u_2&\cdots&0\\\vdots&\vdots&&\vdots\\0&0&\cdots&u_q\end{bmatrix}\begin{bmatrix}x_{11}&x_{12}&\cdots&\sum_{i=1}^{\lambda}x_{1i}^{\lambda}\\x_{21}&x_{22}&\cdots&\sum_{i=1}^{\lambda}x_{2i}^{\lambda}\\\vdots&\vdots&&\vdots\\x_{q1}&x_{q2}&\cdots&\sum_{i=1}^{\lambda}x_{qi}^{\lambda}\end{bmatrix}\begin{bmatrix}\beta_1\\\beta_2\\\vdots\\\beta_{\sum_{i=1}^{\lambda}\binom{\lambda}{i}}\end{bmatrix}+e. \tag{15}$$

For simplicity, we represent Eq. (15) with the following equation:

$$\hat{y}=\mathbf{1}'\mathbf{u}\mathbf{x}\mathbf{b}+e, \tag{16}$$

[3]Here, we assume using a 64bit system on resource-constrained devices.

where $\mathbf{1} \in \mathbb{R}^{1\times q}$, $\mathbf{u} \in \mathbb{R}^{q\times q}$, $\mathbf{x} \in \mathbb{R}^{q\times\sum_{i=1}^{\lambda}\binom{\lambda}{i}}$, and $\mathbf{b} \in \mathbb{R}^{\sum_{i=1}^{\lambda}\binom{\lambda}{i}\times 1}$. Given n_1 samples, we represent $[\mathbf{1}'\mathbf{u}\mathbf{x}_1, \mathbf{1}'\mathbf{u}\mathbf{x}_2, \ldots, \mathbf{1}'\mathbf{u}\mathbf{x}_n]'$ as $\mathbf{X}$. The resulting non-linear model then can be solved using ordinary least squares (OLS) [42]. The solution for $\mathbf{b}$ is as follows:

$$\mathbf{b} = [\sum_{i=1}^{n} \mathbf{1}'\mathbf{u}\mathbf{x}_i\mathbf{1}'\mathbf{u}\mathbf{x}_i]^{-1}[\mathbf{1}'\mathbf{u}\mathbf{x}_1, \mathbf{1}'\mathbf{u}\mathbf{x}_2, \cdots, \mathbf{1}'\mathbf{u}\mathbf{x}_n]\mathbf{y}. \quad (17)$$

Therefore, we feed $\mathbf{u}$ and $\mathbf{x}$ into Eq. (17) to obtain $\mathbf{b}$, and subsequently use the trained regressor to predict t.

2) Runtime Adjustment with GP: We design a GP with an asymmetric kernel function to iteratively adjust the predicted runtime $\hat{\mathbf{Y}}$, hence monitoring the progress of k-means. Specifically, once the actual runtime y_i for the i-th iteration becomes available at the end of that iteration, we can figure out the posterior information by examining the difference between y_i and $\hat{y}_i$ to refine $\hat{\mathbf{Y}}$. This way, with each completed iteration, we can further adjust the estimated runtime.

A commonly used method for adjusting the predicted runtime of iterative algorithms is *Weighted Average* [63]. This method assumes that the predicted time for the next iteration depends solely on previous iterations. However, in practice, information from the current iteration can affect the runtime of all subsequent iterations. For example, if k-means converges within the current iteration, the runtime for all future iterations will be 0, as the k-means task is complete. To address this limitation, GP is a better choice because GP adjusts the prediction of runtime for all iterations based on the degree of correlation between subsequent and current iterations.

Formulation of GP. We build a GP over the predicted runtime, which can be expressed as follows:

$$g(i) \sim \text{GP}(\mu(i), cov(i, i^{'})), \quad (18)$$

where $g(i)$ is the ratio between the predicted runtime $\hat{y}_i$ and the actual runtime y_i for the i-th iteration, $\mu(i)$ represents the mean of $g(i)$, and $cov(i, i^{'})$ represents the *kernel function* (or covariance function) that describes the correlation between the i-th iteration and the $i^{'}$-th iteration. Notably, when the given k-means task has not yet run, we assume perfectly accurate predictions, i.e., $\hat{y}_i = y_i$, which implies $g(i) = 1$. Under this condition, the initial mean function of the GP becomes a constant function equal to 1 for all iterations.

Asymmetric Kernel Function of GP. Unlike a classical GP [32], where posterior information can be bidirectional. For example, A commonly used kernel function is the Radial Basis Function (KBF) kernel [32], which can be shown as follows:

$$cov(i, i^{'}) = \exp\left(-\frac{\|i^{'} - i\|^2}{2\sigma^2}\right), \quad (19)$$

where σ is a hyperparameter for adjusting the correlation between the i-th iteration and the $i^{'}$-th iteration. Here we need to account for the fact that posterior information from the current iteration of k-means affects only subsequent iterations (i.e., completed iterations influence upcoming ones), which means the correlation should only propagate in the direction of increasing i. Therefore, we design the specific expressions for $\text{cov}(i, i^{'})$. To simulate the unidirectional propagation of correlation in an iterative process, we design a new kernel function, which is shown as follows:

$$cov(i, i^{'}) = \begin{cases} 0, & \text{if } i^{'} - i \leq -1; \\ \exp\left(-\frac{h(i^{'}-i)^2}{2\sigma^2}\right), & \text{if } i^{'} - i > -1; \end{cases} \quad (20)$$

where the iteration numbers $i^{'}$ correlated with i are restricted to the interval $(i-1, +\infty)$. This implies that the actual runtime of the i-th iteration only affects the iterations within the range of $(i-1, +\infty)$. Moreover, To ensure that the convergence function is continuously differentiable over its domain, we design $h(\delta)$ as follows:

$$h(\delta) = \begin{cases} \ln(\delta + 1), & \text{if } -1 < \delta \leq 0; \\ \delta, & \text{if } \delta > 0; \end{cases} \quad (21)$$

where, $h(\delta)$ ensures differentiability of $cov(i, i^{'})$ at $(i^{'} - i) = -1$, thus guaranteeing the differentiability of the kernel function in its' domain.

TABLE III
AN OVERVIEW OF THE DATASETS (M FOR MILLION).

Dataset	Dimensionality	Scale	Description
T-drive	2	1M	Trajectory data point
Porto	2	1M	Trajectory data point
Argo-AVL	2	1M	Trajectory data point
Argo-PC	3	1M	Point cloud data
3D-RD	3	0.43M	Point cloud data
Shapenet	3	1M	Point cloud data
Apoll-TD	128	0.5M	Embedded trajectory data
Argo-ETD	256	0.5M	Embedded trajectory data

VI. EXPERIMENTS

We verify the following three questions: **1)** whether `Dask-means` outperforms existing algorithms for (very) large n and k, such as $n = 10^7$ and $k = 10^4$; **2)** whether `Dask-means` uses less memory and performs better compared to other SOTA algorithms; and **3)** whether the proposed cost estimator in `Dask-means` shows superior accuracy in estimating runtime and memory cost.

A. Experimental Settings

Dataset. `Dask-means` is designed for spatial vectors from sensors such as GPS and lidar. For 2D datasets, we select T-drive [68], Porto [3], and Argo-AVL [62], a trajectory dataset from test vehicles in a specific area. For 3D datasets, we select point cloud data including Argo-PC [62], 3D-RD [1], and Shapenet [2]. We also validate our algorithm on high-dimensional datasets. The trajectory datasets are from Argoverse, denoted as Argo-ETD, and ApolloScape, referred to as Apoll-TD, with each trajectory data embedded into fixed-length vectors. The details are provided in Table III.

Implementations. We implement `Dask-means` and comparisons using C++. We test the performance of our algorithm

TABLE IV
THE PERFORMANCE OF DASK-MEANS IN TERMS OF RUNTIME.

Dataset	Settings	Lloyd	NoBound	Dual-tree	Hamerly	Drake	Yinyang	Elkan	NoInB	NokNN	Dask-means
T-drive	$k = 10^2$	128.10	954.26	65.47	34.31	88.21	70.25	21.52	19.55	30.16	**13.13**
	$k = 10^3$	1234.78	385.21	98.87	295.60	541.52	649.61	159.23	385.21	285.01	**28.49**
	$k = 10^4$	24755.76	6225.76	601.22	5853.36	N/A	13954.17	N/A	6225.76	15547.69	**211.36**
Porto	$k = 10^2$	131.11	1119.78	72.51	36.71	72.95	71.22	23.61	23.05	32.61	**15.13**
	$k = 10^3$	1227.00	412.38	102.86	298.38	520.28	642.69	162.86	412.38	314.19	**32.07**
	$k = 10^4$	12295.26	3036.00	300.22	2950.58	N/A	6933.70	N/A	3036.00	8822.68	**237.80**
Argo-AVL	$k = 10^2$	133.59	140.65	61.95	34.15	84.61	68.37	19.58	19.46	31.80	**10.21**
	$k = 10^3$	1261.24	316.66	101.94	296.20	757.24	639.45	160.00	316.66	378.36	**25.93**
	$k = 10^4$	12512.56	3093.48	285.84	2886.71	N/A	6853.31	N/A	3093.48	8858.64	**103.83**
Argo-PC	$k = 10^2$	135.33	112.83	43.82	42.76	77.42	76.00	19.22	11.68	13.69	**8.17**
	$k = 10^3$	1319.38	301.65	63.82	387.85	542.36	711.50	161.41	301.65	238.85	**16.85**
	$k = 10^4$	13247.99	3334.37	270.62	3824.05	N/A	7399.46	N/A	3334.37	9316.83	**78.56**
3D-RD	$k = 10^2$	59.16	34.50	19.37	18.98	19.25	33.29	9.25	21.81	28.03	**6.71**
	$k = 10^3$	573.37	135.14	37.52	165.94	230.50	309.79	75.27	135.14	430.44	**21.99**
	$k = 10^4$	5754.00	1724.97	188.54	1631.71	2546.64	3255.45	853.89	1724.97	5545.52	**47.96**
Shapenet	$k = 10^2$	139.69	143.99	48.89	44.80	49.88	80.66	**24.05**	87.02	260.27	33.54
	$k = 10^3$	1352.89	319.90	58.82	386.38	557.32	741.57	178.24	319.90	1174.97	**77.57**
	$k = 10^4$	13160.80	3368.25	227.03	3794.34	N/A	7674.99	N/A	3368.25	12926.79	**183.74**

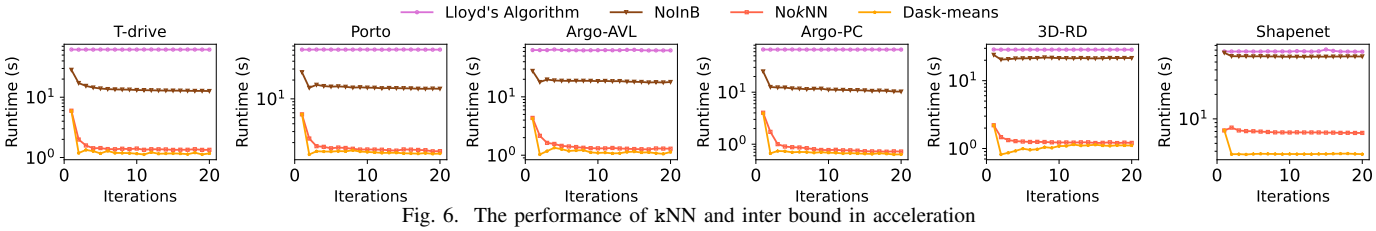


Fig. 6. The performance of kNN and inter bound in acceleration

on both a server and a smartphone: 1) The server, equipped with an i9-14900KF CPU and 128 GB RAM, allows us to simulate the k-means task on resource-limited devices and easily implement our lightweight estimator to predict runtime and memory usage; and 2) We test our algorithm on an OPPO Reno11 5G Android smartphone [5] equipped with a Dimensity 8200 CPU and 12 GB of RAM. Due to page limitations, the detailed information about the smartphone (see Table IX) and images of the k-means algorithms running on it (see Table 15) are presented in the Appendix VIII. This validation demonstrates its superior performance on edge devices compared to other algorithms. Our code is publicly available on GitHub [6].

Comparisons. To answer the first two questions, besides Lloyd's algorithm, we compare Dask-means with the most memory-efficient k-means algorithms including NoBound [64] and Hamerly [26], and three widely-used algorithms, including Dual-tree [50], Drake [19], and Yinyang [17]. Moreover, we compare Dask-means with the k-means algorithm used in scikit-learn [48], known as Elkan [21].

To answer the third question, we use several SOTA cost estimators as competitors (see Section II-D), including XGBoost [24], DisNet [20], and AutoML [43] to predict runtime. We configure the XGBoost with a learning rate of 0.1 and restrict the maximum depth of each tree to 5. Additionally, it specifies that 100 trees are used in the XGBoost model, with a column sampling ratio of 0.3 per tree. Moreover, we set up the DisNet with two hidden layers, the first having 128 neurons and the second with 64 neurons, both of which use ReLU activation. The DisNet model is trained for 1000 epochs with a default learning rate of 1e-4. For AutoML, we set the regularization coefficient as 0.1 and then run the model at a maximum iteration number of 1000 times with the tolerance for convergence set as 0.1. For memory prediction, although there are many estimation methods (see Section II-D), none are designed for k-means tasks in resource-constrained devices.

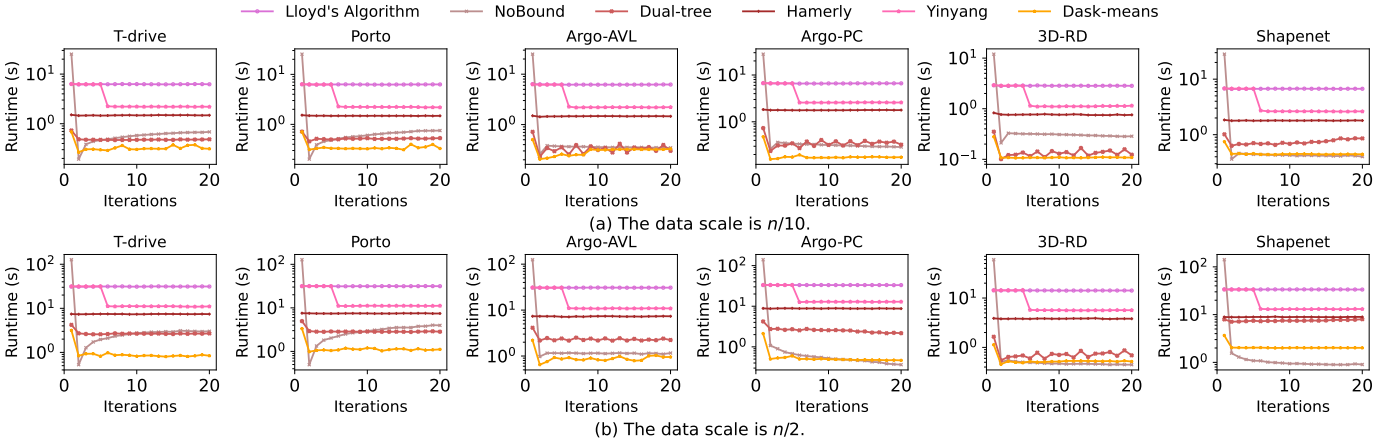
B. Efficiency of Proposed Accelerator

kNN and Inter Bound's Effectiveness in Accelerating. We demonstrate the effectiveness of kNN and the inter bound used in Dask-means. The algorithm only using the inter bound is called NokNN, while the one only using kNN is called NoInB. By default, we set the leaf node capacity to $f = 30$. We also limit the maximum number of iterations to 20 to save time. This is because, as depicted in Fig. 6, each iteration's runtime has already stabilized after the 15-th and 20-th iterations.
Observations. (1) Both NokNN and NoInB can efficiently accelerate Lloyd's algorithm by pruning the number of distance computations. (2) NoInB exhibits higher efficiency compared to NokNN. This indicates that using kNN yields higher pruning power than using inter bound.

Comparisons with SOTAs. We compare Dask-means with other algorithms from two aspects, including the runtime of each iteration and the total runtime of k-means algorithms (due to page limitations, we provide a comparison of the runtime of each iteration in Appendix VIII-B). As shown

TABLE V
VALIDATING PRUNING POWER OF DASK-MEANS IN HIGH-DIMENSIONAL DATASETS.

Dataset	Settings	Lloyd	NoBound	Dual-tree	Hamerly	Drake	Yinyang	Elkan	NoInB	NokNN	Dask-means
Apoll-TD	$k = 10^2$	26.11	217.18	47.25	**23.07**	108.68	49.53	26.07	24.19	42.99	24.04
	$k = 10^3$	258.03	1897.33	158.05	227.64	1416.37	491.13	261.01	40.65	263.91	**40.45**
	$k = 10^4$	2826.21	N/A	1352.58	2327.68	N/A	5119.22	2815.43	193.18	2547.86	**192.69**
Argo-ETD	$k = 10^2$	49.52	408.68	100.50	**45.21**	211.17	94.99	49.58	46.42	83.35	46.23
	$k = 10^3$	495.19	3700.64	323.40	448.26	2731.09	954.21	496.54	78.01	521.65	**77.59**
	$k = 10^4$	5680.86	N/A	2719.24	4780.29	N/A	10468.70	5864.86	384.28	5169.44	**378.82**



(a) The data scale is $n/10$.

(b) The data scale is $n/2$.

Fig. 7. The per-iteration runtime of the k-means algorithm under different data scales.

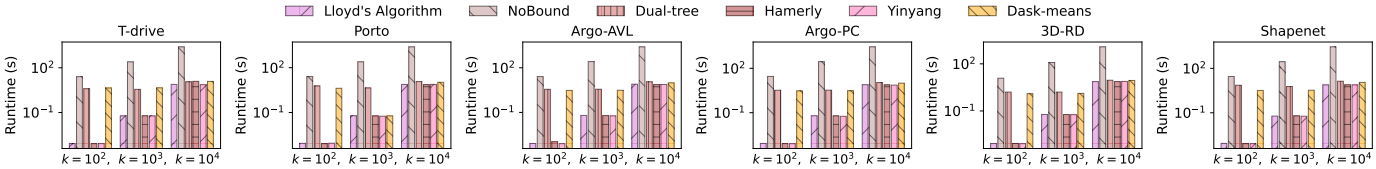


Fig. 8. The initialization runtime of each k-means algorithm.

in Table IV, we evaluate the efficiency of Dask-means by comparing its runtime against SOTA k-means algorithms.

Observations. (1) When k is small (e.g., $k = 10^2$), Dask-means performs better than most SOTAs in pruning power in most cases, but it's not always the best. For example, Elkan outperforms it because Dask-means requires additional time to construct the spatial vector index and centroid index, while kNN on these indexes is inefficient when k is small. (2) Whereas when k is large, Dask-means demonstrates superior runtime performance due to the effective pruning power by the centroid index. For instance, when $k = 10^4$, Dask-means achieves a speedup of over 168 times compared to Lloyd's algorithm on the Argo-PC dataset. (3) When $k = 10^4$, Elkan is unable to execute because it requires to store $n \times k$ lower bounds, which leads to excessive memory cost. Similarly, Drake is unable to execute because it requires storing at least $\frac{k}{8}$ lower bounds for each spatial vector, which is memory-intensive. (4) As shown in Fig. 7, Dask-means demonstrates the best acceleration in almost all data scales. However, its performance diminishes with smaller data scales. This decline is attributed to the fact that, at smaller scales, our proposed kNN search for spatial vectors on the index does not significantly outperform one-by-one searching, while still requiring additional time to build the index.

Efficiency of Initialization. As shown in Fig. 8, we compare the initialization times of various k-means algorithms, such as the time for building the centroid index. This comparison helps clarify that the limited acceleration effects of certain algorithms are caused by the significant time consumed during their initialization. It is worth noting that we exclude Elkan and Drake from our comparisons due to their lack of memory efficiency. At $k = 10^4$, their initialization processes would result in excessive memory overhead on the server.

Observations. (1) The initialization time of NoBound is the longest because it requires computing an $n \times d$ distance matrix, which may contribute to its inefficiency. (2) The initialization time of Dask-means is longer than that of Lloyd's algorithm, Hamerly, and Yinyang due to the additional time needed to build an index over spatial vectors. (3) Different values of k have a significant impact on construction time, but the initialization time of Dask-means is less affected.

Comparison of Space Efficiency. We compare Dask-means with the other SOTAs in memory cost in Fig. 9 (we set $k = 10^3$). Memory cost is the amount of memory required to store the information, such as indexes and bounds in Dask-means.

Observations. (1) Elkan and Drake consume significantly more memory than other algorithms. Specifically, Elkan requires storing $n \times k$ lower bounds to avoid distance computations, while Drake stores $\frac{k}{8}$ to $\frac{k}{4}$ lower bounds for each spatial vector. In contrast, Dask-means uses less than 1% of the memory consumed by these algorithms. Moreover, Yinyang also consumes more memory than Dask-means because it

TABLE VI
AVERAGE PRECISION OF OUR MEMORY ESTIMATION METHOD.

Parameters	Accuracy			
Increasing k	$k = 10$ 0.963	$k = 10^3$ 0.963	$k = 10^4$ 0.963	$k = 5 \times 10^4$ 0.963
Increasing n	$n^{'} = 0.01n$ 0.989	$n^{'} = 0.05n$ 0.983	$n^{'} = 0.25n$ 0.976	$n^{'} = n$ 0.974
Increasing f	$f = 30$ 0.964	$f = 100$ 0.992	$f = 150$ 0.993	$f = 200$ 0.997

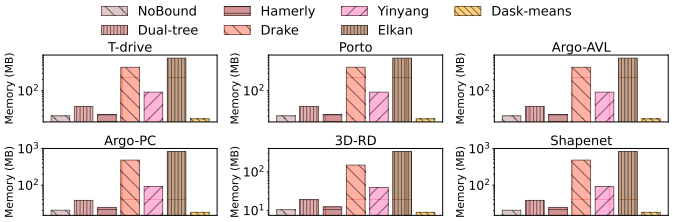


Fig. 9. The performance of `Dask-means` in terms of memory cost.

needs to store the distance from each spatial vector to its assigned cluster. (3) Although `NoBound` uses little memory, its pruning power is much worse than `Dask-means`, as shown in Table IV.

Verification on High-dimensional Datasets. We compare `Dask-means` with selected k-`means` algorithms on high-dimensional datasets, including Apoll-TD and Argo-ETD, focusing on pruning power via runtime. The runtime performance of `Dask-means` is shown in Table V.

Observations. `Dask-means` performs the best in most cases. However, when tackling high-dimensional datasets, almost all k-`means` algorithms perform poorly. This is due to the "curse of dimensionality". It is worth noting that the acceleration performance of `Dask-means` is significantly lower in high-dimensional cases compared to low-dimensional ones. For example, it is only about 15 times faster than Lloyd's algorithm.

Verification on Edge Devices. We validate `Dask-means` on a smartphone and compare its runtime with SOTAs. Due to the maximum response time limits imposed by the Android environment on program execution, the data scale is set to $\frac{1}{20}$ of the original dataset, with $k = 100$.

Observations. (1) As shown in Fig. 10(a), `Dask-means` is generally very fast, although it can be slower than `Drake` in some cases. However, `Drake` needs to store between $\frac{k}{8}$ and $\frac{k}{4}$ lower bounds for each spatial vector, which consumes significantly more memory than `Dask-means`, making it not memory-efficient. (2) As shown in Fig. 10(b), in some cases, `Dask-means` consumes more memory than `Hamerly`, as `Hamerly` only requires storing one upper bound and one lower bound for each spatial vector. However, its pruning power is weaker compared to `Dask-means`.

Summary of Lessons Learned. Through the evaluation of `Dask-means` in runtime and memory cost, we further learn:

- Both `NokNN` and `NoInB` accelerate Lloyd's algorithm, but `NoInB` is much more efficient, likely because our estimated bounds are too loose.

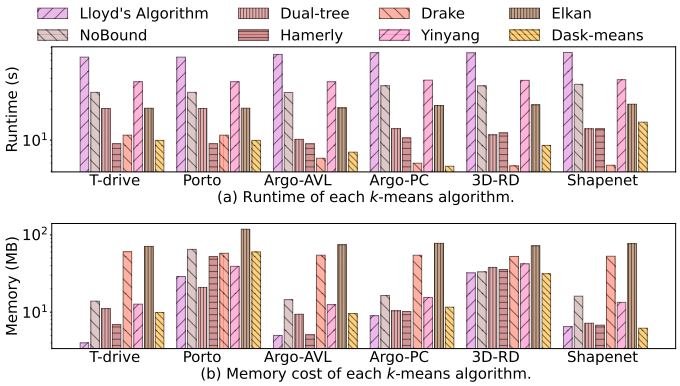


Fig. 10. The performance of `Dask-means` in the smartphone.

- The value of k has only a slight effect on efficiency. This is consistent with the observation that $\log_2 k$ and the dataset scale n have a linear relation with the running time.
- For high-dimensional datasets, `Dask-means` can still accelerate Lloyd's Algorithm; however, its acceleration performance is significantly lower than that for low-dimensional datasets due to the "curse of dimensionality".

C. Evaluation of Our Cost Estimator

We test our cost estimator to demonstrate its superiority in predicting memory cost and runtime. We generate 2000 k-means tasks as a sample set and divide them into three parts: 80% for training, 10% for validation, and 10% for testing. For each k-means task, we randomly select a dataset with a size ranging from 1×10^5 to 1×10^8 and choose k randomly between 1×10^2 and 1×10^4. We then extract the features, run `Dask-means`, and record the runtime. It is important to note that for predicting runtime, we choose $\beta = 4$ and $\sigma = 50$ as default parameters (details on the selection of a suitable β and σ can be found in Appendix VIII-C).

Memory Cost Estimation. We first show that the proposed cost estimator can accurately estimate the memory cost of `Dask-means`. It is worth noting that the estimated memory cost is often less than the actual memory used (see Section V-A). Hence, we measure the accuracy of our memory estimation method using the ratio of the estimated memory to the actual memory consumed.

Observations. As shown in Table VI, when k (i.e., the number of centroids) increases, the prediction accuracy of our proposed cost estimator remains unchanged. This is because the memory used by the centroid index is much smaller than the memory used to construct the spatial vector index. Moreover, as $n^{'}$ (the number of spatial vectors for k-`means`) increases, the prediction accuracy of our proposed cost estimator decreases. This is because an increase in dataset scale leads to more nodes in index structure, and we estimate memory usage by assuming that each index node only includes the spatial vector and pointer it stores, without considering additional information like locks in the "vector" structure. Hence, increasing the number of nodes adds more unestimated information, reducing the accuracy of the memory estimation. Similarly, when f increases, the index has fewer nodes, resulting in higher prediction accuracy.

TABLE VII
THE IMPACT OF THE MEMORY LIMITATION ON DASK-MEANS.

Efficiency	Dataset	T-drive			Porto			Argo-AVL			Argo-PC			3D-RD			Shapenet		
	Available Memory (MB)	15	20	30	15	20	30	15	20	30	15	20	30	15	20	30	15	20	30
Runtime (s)	$k = 10^2$	13.51	14.82	18.14	15.67	16.59	19.42	10.08	11.03	13.14	8.04	9.14	11.14	6.82	6.63	7.28	32.31	28.01	28.06
	$k = 10^3$	28.86	25.98	27.15	32.66	28.71	29.39	25.76	20.80	21.87	17.06	15.74	16.46	20.87	21.74	18.38	76.95	93.12	122.15
	$k = 10^4$	105.32	92.23	83.77	117.40	100.55	89.50	86.72	80.61	67.12	71.70	59.03	49.87	48.41	54.66	68.39	179.13	188.42	199.32
Pruned Vectors (M)	$k = 10^2$	18.62	19.32	19.66	18.37	19.15	19.57	18.12	19.06	19.55	19.35	19.68	19.84	6.82	6.90	7.62	8.92	12.20	15.04
	$k = 10^3$	15.91	17.85	18.92	15.07	17.36	18.63	13.72	16.80	18.49	16.20	18.11	19.19	1.98	3.91	5.79	2.30	5.37	9.37
	$k = 10^4$	7.24	12.18	15.47	5.50	10.61	14.51	5.16	10.73	15.12	5.10	11.41	15.86	0.09	0.88	2.71	0.02	0.71	3.74

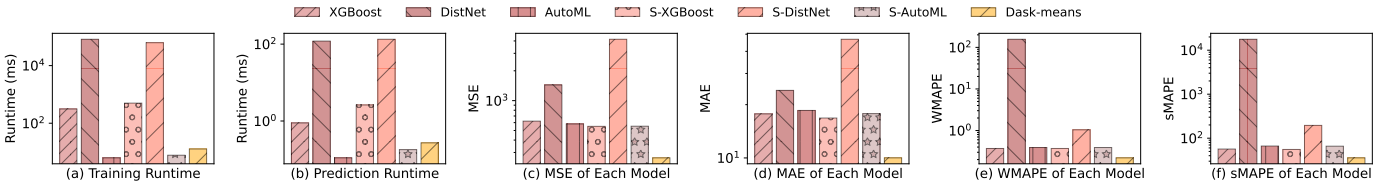


Fig. 11. The performance of our cost estimator in terms of predicting runtime.

Impact of Memory Constraints. As shown in Table VII, under various memory limits, we evaluate the efficiency of the memory-tunable index for accelerating k-means tasks.

Observations. (1) As shown in Table VII, as memory cost increases, the number of pruned spatial vectors also rises. A higher memory cost leads to a smaller f, and kNN search on an index with a smaller f consistently results in a reduced search radius. Consequently, more unnecessary spatial vectors and nodes are pruned, improving the index's pruning capability. (2) We find that as memory increases, the runtime does not necessarily decrease. This is because, while more memory improves the index's pruning power, it also requires additional time to build the index, which offsets the time saved from improved pruning. Additionally, as k increases, the runtime also increases, indicating that k-means converge faster with smaller values of k.

Comparisons with SOTAs in Runtime Prediction. We use four metrics [16] to assess the accuracy in terms of runtime: Mean Squared Error (**MSE**), Mean Absolute Error (**MAE**), Weighted Absolute Mean Percentage Error (**WAMPE**), and Symmetric Mean Absolute Percentage Error (**SMAPE**). Then we compare our cost estimator with SOTA models, observing each model's training time, prediction time, and accuracy in predicting runtime. Moreover, we modify existing models to predict each iteration separately and then sum the predictions to obtain the total runtime. The modified models are labeled with `S-`, such as `S-XGBoost`, `S-DisNet`, and `S-AutoML`.

Observations. (1) Fig. 11(a) shows that our cost estimator has the shortest training time compared to others, similar to `AutoML`. This is because both the proposed cost estimator and `AutoML` require only one pass through the dataset to obtain regression parameters. (2) Fig. 11(b) illustrates that prediction methods like `Dask-means` and `AutoML` have similar prediction times, typically a few milliseconds. Additionally, compared to the overall runtime of `Dask-means`, which requires several seconds to minutes per iteration, this prediction time is negligible. (3) Fig. 11(c), (d), (e), and (f) demonstrate that our cost estimator achieves the highest prediction accuracy, with the smallest MSE, MAE, WMAPE, and sMAPE compared to others. Moreover, it shows that using complex iterative algorithms does not necessarily lead to better performance. For example, regression models often achieve higher accuracy than `XGBoost`. Moreover, models such as `XGBoost` perform worse after modification.

Summary of Lessons Learned. Through the evaluation of our cost estimator, we further learn:

- As the leaf node capacity f increases, the runtime of the k-means task does not necessarily increase. This is because, although pruning with a larger radius r has a lower success probability, the time to build the index also decreases.
- Our cost estimator predicts runtime more accurately than others. However, it's important to note that the runtime of different k-means tasks varies significantly, leading to discrepancies between predicted and actual times that can be several times the actual k-means runtime.
- Our runtime adjustment method dynamically corrects runtime. However, if parameters like σ are not chosen properly, such as $\sigma = 2$, its adjustment capability will significantly decrease and may decrease prediction accuracy.

VII. CONCLUSIONS

To accelerate k-`means` for simplifying large-scale spatial vectors, we leveraged fast kNN search and assigned spatial vectors to the nearest centroid in batches by indexing on both spatial vectors and centroids. Without updating the bounds for the next iteration, novel bounds were designed to further accelerate the kNN search. Moreover, we designed a lightweight cost estimator to predict the k-means memory cost and runtime accurately. Experiments on real-world datasets verified the efficiency of `Dask-means` on resource-constrained devices.

In future work, we will design a distributed k-`means` on resource-constrained devices to leverage the remaining computational power of edge devices to accelerate k-`means`. Additionally, we plan to design a more lightweight and accurate cost estimator and extend it to other iterative algorithms.

REFERENCES

[1] 3d road network (north jutland, denmark). https://networkrepository.com/3D-spatial-network.php.

[2] Shapenet. https://shapenet.org/.

[3] Taxi service trajectory prediction challenge 2015. https://figshare.com/articles/dataset/Porto_taxi_trajectories/12302165.

[4] FlyingFox. https://www.hackster.io/flyingfox/flyingfox-821a16, 2021.

[5] Oppo reno11 5g, 2024. https://www.oppo.com/en/smartphones/series-reno/reno11/.

[6] Repository of Dask-means. https://github.com/notNNORTH/Dask-means-cpp, 2024.

[7] TensorFlow Shape Infer . https://malmaud.github.io/tfdocs/shape_inference/, 2024.

[8] T. S. Abdelrahman. Cooperative software-hardware acceleration of k-means on a tightly coupled CPU-FPGA system. *ACM Trans. Archit. Code Optim.*, 17(3):20:1–20:24, 2020.

[9] M. Ahmed. Data summarization: a survey. *Knowl Inf Syst*, 58:249–273, 2019.

[10] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In *ISMM*, pages 121–130, 2010.

[11] M. A. Bender, J. Berry, S. D. Hammond, B. Moore, B. Moseley, and C. A. Phillips. k-Means Clustering on Two-Level Memory Systems. In *MEMSYS*, pages 197–205, 2015.

[12] I. Brand, J. Roy, A. Ray, J. Oberlin, and S. Oberlix. PiDrone: An Autonomous Educational Drone Using Raspberry Pi and Python. In *IROS*, pages 5697–5703, 2018.

[13] A. Canziani, A. Paszke, and E. Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[14] S. Castelo, F. Chirigati, R. Rampin, A. Santos, A. Bessa, and J. Freire. Auctus: A Dataset Search Engine for Data Augmentation. *PVLDB*, 14(12):2791 – 2794, 2021.

[15] M.-F. Chang, J. Lambert, P. Sangkloy, J. Singh, S. Bak, A. Hartnett, D. Wang, P. Carr, S. Lucey, D. Ramanan, and J. Hays. Argoverse: 3D Tracking and Forecasting with Rich Maps. In *CVPR*, pages 8748–8757, 2019.

[16] D. Chicco, M. J. Warrens, and G. Jurman. The coefficient of determination r-squared is more informative than smape, mae, mape, MSE and RMSE in regression analysis evaluation. *PeerJ Comput. Sci.*, 7:e623, 2021.

[17] 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. In *ICML*, pages 579–587, 2015.

[18] T. Doan and J. Kalita. Predicting run time of classification algorithms using meta-learning. *Int. J. Mach. Learn. Cybern.*, 8(6):1929–1943, 2017.

[19] J. Drake. Faster k-means Clustering. In *MS Thesis*, 2013.

[20] K. Eggensperger, M. Lindauer, and F. Hutter. Neural networks for predicting algorithm runtime distributions. In *IJCAI*, pages 1442–1448, 2018.

[21] C. Elkan. Using the triangle inequality to accelerate k-means. In *ICML*, page 147–153, 2003.

[22] Y. Fan, P. Rich, W. E. Allcock, M. E. Papka, and Z. Lan. Trade-off between prediction accuracy and underestimation rate in job runtime estimates. In *CLUSTER*, pages 530–540, 2017.

[23] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang. Estimating GPU memory consumption of deep learning models. In *FSE*, pages 1342–1352, 2020.

[24] B. R. Gunnarsson, S. vanden Broucke, and J. D. Weerdt. A direct data aware LSTM neural network architecture for complete remaining trace and runtime prediction. *IEEE Trans. Serv. Comput.*, 16(4):2330–2342, 2023.

[25] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun. Deep Learning for 3D Point Clouds: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 6 2020.

[26] G. Hamerly. Making k-means even faster. In *SDM*, pages 130–140, 2010.

[27] G. Hamerly and J. Drake. *Accelerating Lloyd's Algorithm for k-Means Clustering*. 2015.

[28] W. He, Z. Jiang, M. Kriby, Y. Xie, X. Jia, D. Yan, and Y. Zhou. Quantifying and reducing registration uncertainty of spatial vector labels on earth imagery. In *KDD*, pages 554–564, 2022.

[29] K. Heo, H. Oh, and H. Yang. Resource-aware program analysis via online abstraction coarsening. In *ICSE*, pages 94–104, 2019.

[30] Q. Hu, B. Yang, L. Xie, S. Rosa, Y. Guo, Z. Wang, N. Trigoni, and A. Markham. RandLA-Net: Efficient Semantic Segmentation of Large-Scale Point Clouds. In *CVPR*, pages 11105–11114, 2020.

[31] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods and evaluation (extended abstract). In Q. Yang and M. J. Wooldridge, editors, *IJCAI*, pages 4197–4201, 2015.

[32] S. Jayasumana, R. I. Hartley, M. Salzmann, H. Li, and M. T. Harandi. Kernel methods on riemannian manifolds with gaussian RBF kernels. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(12):2464–2477, 2015.

[33] T. Kapus and C. Cadar. A segmented memory model for symbolic execution. In *FSE*, pages 774–784, 2019.

[34] M. Kleindessner, P. Awasthi, and J. Morgenstern. Fair k-Center Clustering for Data Summarization. In *ICML*, 2019.

[35] M. Krulis and M. Kratochvíl. Detailed analysis and optimization of CUDA k-means algorithm. In *ICPP*, pages 69:1–69:11, 2020.

[36] I. Lang, A. Manor, and S. Avidan. SampleNet: Differentiable point cloud sampling. In *CVPR*, pages 7578–7588, 2020.

[37] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *J. ACM*, 56(4):22:1–22:52, 2009.

[38] Y. Li, K. Zhao, X. Chu, and J. Liu. Speeding up k-Means algorithm by GPUs. *Journal of Computer and System Sciences*, 79:216–229, 2013.

[39] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[40] C. Lv, W. Lin, and B. Zhao. Approximate intrinsic voxel structure for point cloud simplification. *IEEE Trans. Image Process.*, 30:7241–7255, 2021.

[41] S. Mariam, A. Chew, and C. Meng. Density Based Clustering for 3D Object Detection in Point Clouds. In *CVPR*, pages 10608–10617, 2020.

[42] D. Maulud and A. M. Abdulazeez. A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, 1(2):140–147, 2020.

[43] F. Mohr, M. Wever, A. Tornede, and E. Hüllermeier. Predicting machine learning pipeline runtimes in the context of automated machine learning. *IEEE Trans. Pattern Anal. Mach. Intell.*, 43(9):3055–3066, 2021.

[44] A. W. Moore. The Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data. In *UAI*, pages 397–405, 2000.

[45] J. Newling and F. Fleuret. Fast k-means with accurate bounds. In *ICML*, pages 936–944, 2016.

[46] J. Newling and F. Fleuret. K-Medoids For K-Means Seeding. In *NIPS*, pages 5201–5209, 2017.

[47] S. M. Omohundro. Five Balltree Construction Algorithms. Technical report, 1989.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[49] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *CVPR*, pages 652–660, 2017.

[50] C. R. R. A Dual-Tree Algorithm for Fast k-means Clustering With Large k. In *SDM*, pages 300–308, 2017.

[51] X. Roynard, J. E. Deschaud, and F. Goulette. Paris-Lille-3D: A large and high-quality ground-truth urban point cloud dataset for automatic segmentation and classification. *International Journal of Robotics Research*, 37(6):545–557, 2018.

[52] P. Ryšavý and G. Hamerly. Geometric methods to accelerate k -means algorithms. In *SDM*, pages 324–332, 2016.

[53] J. Shao, H. Zhang, Y. Mao, and J. Zhang. Branchy-GNN: a Device-Edge Co-Inference Framework for Efficient Point Cloud Processing. Technical report, 2020.

[54] X. Su, X. Yan, and C.-L. Tsai. Linear regression. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(3):275–294, 2012.

[55] X. Sun, H. Ma, Y. Sun, and M. Liu. A Novel Point Cloud Compression Algorithm Based on Clustering. *IEEE Robotics and Automation Letters*, 4(2):2132–2139, 2019.

[56] W. Tang, N. Desai, D. Buettner, and Z. Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p. In *IPDPS*, pages 1–11, 2010.

[57] J. Tuero and M. Buro. Bayes distnet - A robust neural network for algorithm runtime distribution predictions. In *AAAI*, pages 12418–12426, 2021.

[58] I. Verbauwhede, C. J. Scheers, and J. M. Rabaey. Memory estimation for high level synthesis. In *DAC*, pages 143–148, 1994.

[59] C. Wang, L. Gong, F. Jia, and X. Zhou. An FPGA based accelerator for clustering algorithms with custom instructions. *IEEE Trans. Computers*, 70(5):725–732, 2021.

[60] S. Wang, Z. Bao, J. S. Culpepper, and G. Cong. A survey on trajectory data management, analytics, and learning. *ACM Comput. Surv.*, 54(2):39:1–39:36, 2022.

[61] S. Wang, Y. Sun, and Z. Bao. On the Efficiency of K-Means Clustering: Evaluation, Optimization, and Algorithm Selection. *PVLDB*, 14(2):163–176, 2021.

[62] B. Wilson, W. Qi, T. Agarwal, J. Lambert, J. Singh, S. Khandelwal, B. Pan, R. Kumar, A. Hartnett, J. K. Pontes, D. Ramanan, P. Carr, and J. Hays. Argoverse 2: Next generation datasets for self-driving perception and forecasting. *CoRR*, abs/2301.00493, 2023.

[63] M. Wortsman, G. Ilharco, S. Y. Gadre, R. Roelofs, R. G. Lopes, A. S. Morcos, H. Namkoong, A. Farhadi, Y. Carmon, S. Kornblith, and L. Schmidt. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *ICML*, volume 162, pages 23965–23998, 2022.

[64] S. Xia, D. Peng, D. Meng, C. Zhang, G. Wang, E. Giem, W. Wei, and Z. Chen. A Fast Adaptive k-means with No Bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2020.

[65] X. Xu and G. Hee Lee. Weakly Supervised Semantic Point Cloud Segmentation: Towards 10x Fewer Labels. In *CVPR*, pages 13706–13715, 2020.

[66] X. Yin, Y. Sasaki, W. Wang, and K. Shimizu. 3D Object Detection Method Based on YOLO and K-Means for Image and Point Clouds. Technical report, 2020.

[67] T. Yu, W. Zhao, P. Liu, V. Janjic, X. Yan, S. Wang, H. Fu, G. Yang, and J. Thomson. Large-scale automatic k-means clustering for heterogeneous many-core supercomputer. *IEEE Trans. Parallel Distributed Syst.*, 31(5):997–1008, 2020.

[68] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: Driving directions based on taxi trajectories. In *GIS*, pages 99–108, 2010.

VIII. Appendix

A. Complexity Analysis

We analyze the time complexity of the proposed pruning mechanism. We first analyze the construction time and search time on different types of indexes (using Ball-tree structures). A balanced Ball-tree containing n spatial vectors has a height of $\lceil \log_2 \frac{2n}{f} \rceil$ when each leaf node contains $\frac{f}{2}$ spatial vectors. Assume that the dataset consists of d-dimensional spatial vector. Then the construction time of a balanced Ball-tree is $O(dn \log_2 \frac{2n}{f})$ [47] and the kNN search on a balanced Ball-tree costs $O(d(\log_2 \frac{2n}{f} + f))$ time, which is the best case. On the other hand, for a degenerate Ball-tree with height $n - f$, the construction time is $O(dn^2)$ and the complexity of kNN search can be as high as $O(dn)$ in the worst case.

In each iteration of the clustering algorithm, it takes $O(dk \log_2 \frac{2k}{f}) \sim O(dk^2)$ time to create a Ball-tree on C. Then, in lines 5-7, the computation of the inter bound for each centroid costs $O(dk(\log_2 \frac{2k}{f} + f)) \sim O(dk^2)$ time. The **Assign** function, in the worst case, needs to scan the whole Ball-tree on $\mathbf{D}$ and this process costs $O(dn(\log_2 \frac{2k}{f} + f)) \sim O(dnk)$ time. Lastly, it takes $O(k)$ time to refine centroids. Thus, the total time complexity of `Dask-means` is $O(d(n+2k)\log_2 \frac{2k}{f} + d(n+k)f) \sim O(d(n+2k)k)$.

Note that the total runtime is related to the iteration number of k-means. However, the above time complexity for each iteration is just theoretical analysis, and calculating the total runtime is still challenging, as it is not clear when k-means tasks converge. Next, we will design a cost estimator to predict the memory cost and the runtime accurately for k-means tasks.

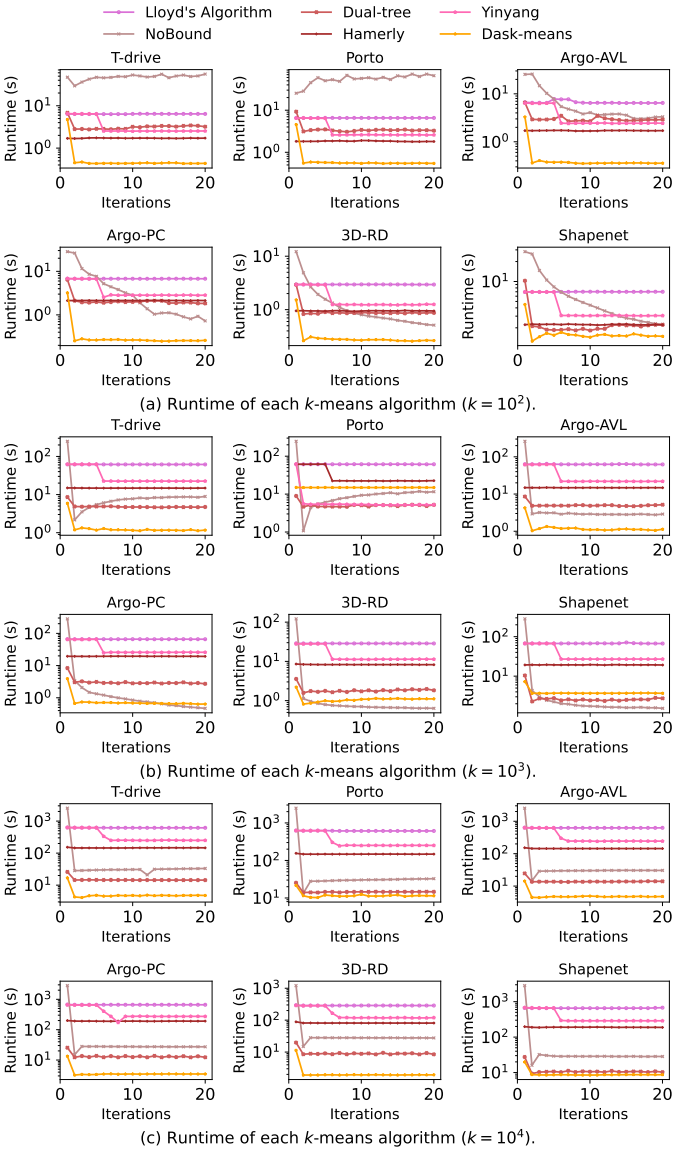
B. Additional Comparisons with SOTAs

As shown in Fig. 12, we evaluate the efficiency of `Dask-means` by comparing its per-iteration runtime with other SOTA k`-means` algorithms.

Observations. (1) `Dask-means` achieves the best per-iteration acceleration in most cases when k takes on different values. However, when k is not large, such as $k = 10^3$, as shown in Porto in Fig. 12(b), `Dask-means` is slower than `NoBound`, `Hamerly`, and `Dual-tree`. This is because `Dask-means` incurs additional time due to constructing two extra indexes, while its pruning power is less effective. Moreover, we observe that `NoBound` does not accelerate Lloyd's algorithm and is even slower when k is small, as shown in T-drive in Fig. 12(b). (2) When k is relatively large, such as $k = 10^4$, the per-iteration runtime stabilizes after the first five rounds. (3) The per-iteration runtime of `Hamerly` remains consistent, indicating that the pruning power from assigning each point upper and lower bounds remains stable.

C. Parameter Selection for Our Cost Model

We test different β (see Section V-B1) values within the range $(1, 6)$ and various σ values (see Section V-B2) within the range $(1, 100)$ to determine the suitable β and σ. Moreover, we verify whether the interaction features can improve the prediction accuracy of the runtime prediction method.



(a) Runtime of each k-means algorithm ($k = 10^2$).

(b) Runtime of each k-means algorithm ($k = 10^3$).

(c) Runtime of each k-means algorithm ($k = 10^4$).

Fig. 12. The runtime performance of k`-means` algorithms in each iteration.

TABLE VIII
The Impact of the Interaction Features and β.

Degree	Basic Feature				Interaction Feature			
	MSE	MAE	WMAPE	sMAPE	MSE	MAE	WMAPE	sMAPE
β = 1	600.48	18.44	0.41	62.15	525.33	17.39	0.39	59.06
β = 2	245.62	12.12	0.27	37.79	229.01	10.52	0.23	37.44
β = 3	324.07	11.26	0.25	31.52	264.76	10.01	0.22	35.72
β = 4	324.68	11.29	0.25	28.78	**227.47**	**9.44**	**0.21**	**25.72**
β = 5	335.36	12.07	0.27	34.04	232.52	10.75	0.24	36.90
β = 6	383.38	13.70	0.30	40.51	1167.00	13.66	0.30	39.20

Observations. (1) As shown in Table VIII, the four evaluation metrics decrease as β increases, reaching their minimum at $\beta = 4$. Beyond this point, the metrics increase as β continues to grow. Hence $\beta = 4$ is a suitable choice. Moreover, adding the interaction features improves the cost estimator's prediction accuracy. (2) As shown in Fig. 13, when $\sigma = 50$, our method reaches its strongest adjustment capability. However, if σ is poorly chosen, the values of the four metrics become large. For example, $\sigma = 2$ assumes a weak correlation between

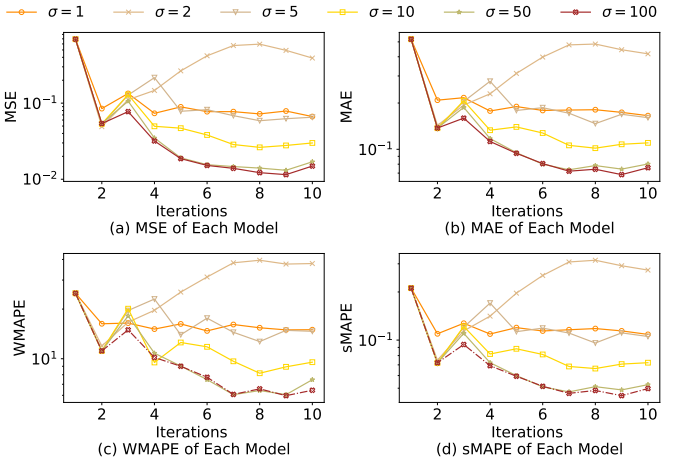


Fig. 13. The impact of α in adjusting the runtime.

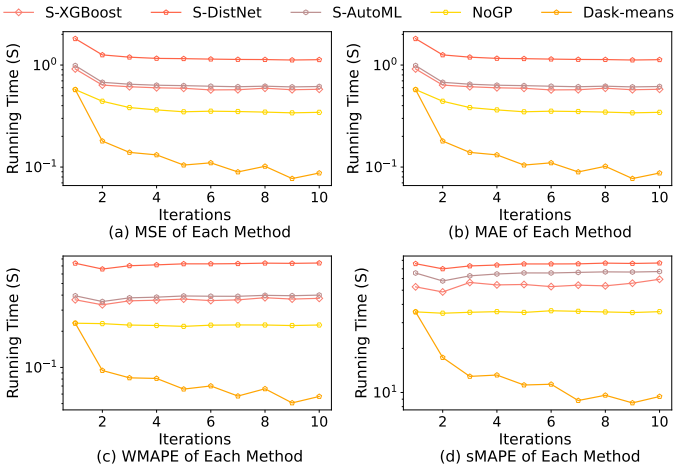


Fig. 14. The performance of our cost estimator in adjusting predicted runtime.

iterations, which is unrealistic. For example, once the final centroids are found and k-means is completed, there are no further iterations (the runtime for the next iteration is 0). Moreover, as runtime progresses, we find that the MSE, MAE, WMAPE, and sMAPE decrease at a roughly constant rate, indicating that adjusting σ has less impact as the k-means tasks approach convergence.

D. Verification for Predicted Runtime Adjustment

We verify that using the proposed cost estimator can adjust the runtime dynamically based on the posterior information they acquired from the current iteration. The calculation of metrics is obtained by comparing the predicted runtime with the actual runtime for each specified iteration. Notably, `Dask-means` without applying GP is referred to as `NoGP`.

Observations. As shown in Fig. 14, compared to other SOTA methods, our cost estimator performs best across four metrics. Moreover, our cost estimator effectively corrects predicted runtime compared to `NoGP`. Furthermore, as k`-means` runs longer (with more iterations), more posterior information is obtained, improving the ability to adjust prediction times.

TABLE IX
INFORMATION OF SMARTPHONE.

Attribute	Specification
Model	OPPO Reno11 5G
Dimensions	74.3 × 162.4 × 7.99 mm
Weight	182g
SoC	MediaTek Dimensity 7050 (MT6877V)
CPU	8-core ARM Cortex-A78/A55 (2.6/2.0 GHz)
GPU	ARM Mali-G68 MC4, 950 MHz, Cores: 4
RAM	12 GB, 2133 MHz
Storage	256 GB
Display	6.7 in, OLED, 1080 x 2412 pixels, 30 bit
Battery	55000mAh/19.45Wh
Fast Charge	SUPERVOOCTM 67W and SUPERVOOCTM 2.0
Biometrics	Fingerprint and Facial Recognition
OS	ColorOS 14 (Android 14)
Camera	9280 × 6920 pixels, 3840 × 2160 pixels, 30 fps
SIM card	Nano-SIM
USB	2.0, USB Type-C
Bluetooth	5.3
Positioning	GPS, A-GPS, GLONASS, BeiDou, Galileo, QZss

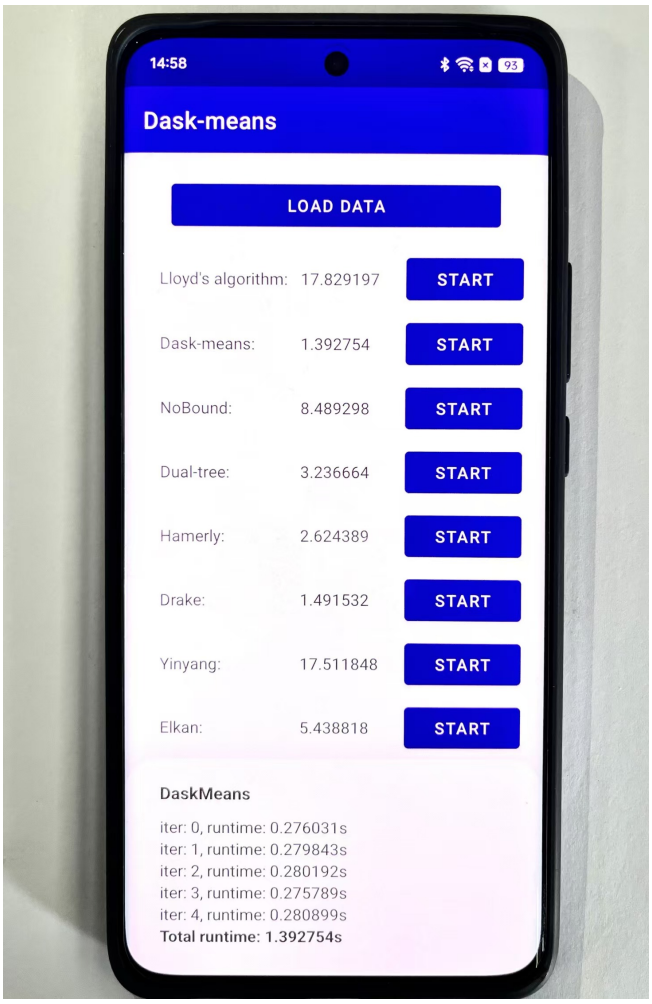


Fig. 15. Running k`-means` algorithms on the smartphone.