On the Efficiency of K-Means Clustering: Evaluation, Optimization, and Algorithm Selection

[Experiments, Analyses & Benchmarks]

Sheng Wang
New York University
swang@nyu.edu

Yuan Sun
RMIT University
yuan.sun@rmit.edu.au

Zhifeng Bao
RMIT University
zhifeng.bao@rmit.edu.au

ABSTRACT

This paper presents a thorough evaluation of the existing methods that accelerate Lloyd's algorithm for fast k-means clustering. To do so, we analyze the pruning mechanisms of existing methods, and summarize their common pipeline into a unified evaluation framework `UniK`. Our `UniK` embraces a class of well-known methods and enables a fine-grained performance breakdown of existing methods. Within `UniK`, we thoroughly evaluate the pros and cons of existing methods using multiple performance metrics on a number of datasets. Furthermore, we derive an optimized algorithm over `UniK`, which effectively hybridizes multiple existing methods for more aggressive pruning. To take this further, we also investigate whether the most efficient method for a given clustering task can be automatically selected by machine learning, to benefit practitioners and researchers.

PVLDB Reference Format:
Sheng Wang, Yuan Sun, Zhifeng Bao. On the Efficiency of K-Means Clustering: Evaluation, Optimization, and Algorithm Selection. *PVLDB*, 14(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

As one of the most widely-used clustering algorithms, k-means aims to partition a set of n points into k ($k < n$) clusters where each point is assigned to the cluster with the nearest centroid [44, 68]. Answering k-means is NP-hard, and the Lloyd's algorithm [49] forms the cornerstone to solve it. Essentially, the Lloyd's algorithm randomly initializes k centroids, then assigns each point to the cluster with the nearest centroid and refines each centroid iteratively. In each iteration, it requires to compute $n \cdot k$ distances in the assignment step and access n data points in the refinement step. Such intensive computations make the Lloyd's algorithm slow, especially in partitioning large dataset.

Accelerating the Lloyd's algorithm for k-means clustering has been investigated for more than 20 years since the first work was published [57]. Most of the existing acceleration methods focus on how to reduce intensive distance computations, which can be broadly divided into two categories: 1) the index-based methods that group and prune points in batch [57, 51, 45, 27], and 2) the sequential-based methods that scan each point one by one and utilize a bound based on triangle inequality to avoid calculating certain distance [58, 38, 41, 37, 36, 35, 42, 53, 60, 26, 47, 34].

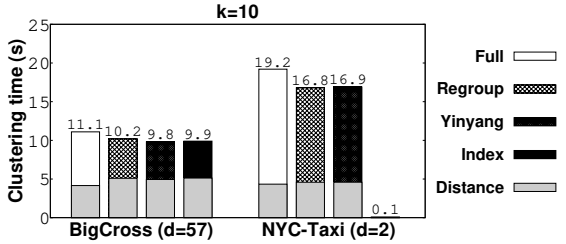


Figure 1: The performances of representative methods, Regroup, Yinyang, Index and Full (using multiple pruning mechanisms) on two clustering datasets, where d is the dimensionality of the dataset. The gray bar ("Distance") denotes the time taken by each method to compute distance.

1.1 Motivations

Conducting a thorough evaluation of existing k-means algorithms. Whilst a large body of methods have been proposed to accelerate the Lloyd's algorithm for k-means clustering, there is still a lack of thorough evaluation on the efficiency of these methods. Moreover, there seems to be some misunderstanding on the performance of certain methods in the literature. For example, the index-based method [45] was interpreted to be slower compared to the sequential-based methods (e.g., Yinyang [35], Regroup [60]) when the dimensionality of dataset is greater than 20 [35], and hence was discarded by the machine learning community in its most recent studies [35, 53, 60]. However, we show in Figure 1 that the index-based method is in fact relatively fast and has the potential to significantly accelerate large-scale clustering when using a proper data structure. This motivates us to conduct a fair and more thorough efficiency evaluation on existing methods.

In fact, most existing studies consider the number of distance computations as the main metric to improve the efficiency of their methods. However, a method that computes fewer number of distances does not simply guarantee to have a shorter computational time. For example in Figure 1, the Full method, which is armed with multiple pruning techniques, has the least number of distance computation, but

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

overall is the slowest on the BigCross dataset. This is because other metrics, such as the number of data accesses and the time taken to compute a bound for pruning also account for part of the computational time, should also be considered. To identify the key metrics, it is essential to analyse the pruning mechanisms of existing methods and extract a unified framework, such that existing methods can well fit to enable a fine-grained performance breakdown of existing methods and in turn a more comprehensive evaluation.

Selecting the best k-means algorithm for a given clustering task. Recently, fast k-means clustering for an arbitrary dataset has attracted much attention [50]. Unfortunately, there is no single algorithm that is expected to be the "fastest" for clustering all datasets, which is also in line with the *"no free lunch" theorem* in optimization [67]. This necessitates an effective selection approach that is able to adaptively select the best algorithm for a given clustering task. However, the existing selection criterias are still based on simple rules, e.g., choosing the index-based method when the dimensionality of dataset is less than 20. Given the complex nature of clustering, such naive selection rules are unlikely to work well in practice. In this regard, we aim to develop a more effective algorithm selection approach in this paper.

Given the large amount of data collected from our evaluations, it is logical to apply machine learning to automatically learn an optimal mapping from a clustering task to the best performing algorithm. Note that the idea of using machine learning for algorithm selection has been explored before, e.g., *meta-learning* [65] and auto-tuning in database management [48, 64]. However, we argue that it is nontrivial to apply this technique to k-means clustering, because problem-specific features have to be carefully designed to describe datasets to be clustered. Furthermore, we will evaluate the effects of using different machine learning models and various feature sets for k-means algorithm selection.

1.2 Our Contributions

In this paper, we aim to design a unified experimental framework to evaluate various existing k-means clustering methods, and design a machine learning (ML) approach to automatically select the best method for a given clustering task. More specifically, we make the following contributions:

- We review the index-based and sequential-based methods and describe their pruning mechanisms in Section 3 and 4.
- Inspired by the common pruning pipeline of existing methods, we design a unified framework `UniK` in Section 5, that enables us to compare existing methods more fairly and comprehensively. Some key insights obtained are: 1) the index-based method can be very fast even for high-dimensional data, when equipped with a proper data structure such as Ball-tree [63]; and 2) no single method always performs the best across all cases. Detailed evaluations are in Section 7.2.
- This leads us into designing an adaptive setting for our `UniK`, which applies the bound-based pruning from sequential methods to assign points in batch without scanning all centroids. In Section 7.2.3, we evaluate our adaptive `UniK` and show that it outperforms the existing k-means algorithms when tested on various real-world datasets.
- To take it further, we adopt machine learning models to automatically select the best method for a given new clustering task in Section 6. This is achieved by learning from our evaluation records which contain the performance of all the methods on various datasets. An evaluation on multiple learning models is also conducted in Section 7.3.

2. PRELIMINARIES

2.1 Definition of k-means

Given a dataset $D = \{x_1, x_2, \cdots, x_n\}$ of n points, and each point has d dimensions, k-means aims to partition D into k mutually exclusive subsets $S = \{S_1, S_2, \cdots, S_k\}$ to minimize the *Sum of Squared Error*,

$$\arg\min_{S} \sum_{j=1}^{k} \sum_{x \in S_j} \|x - c_j\|, \tag{1}$$

where $c_j = \frac{1}{|S_j|} \sum_{x \in S_j} x$, namely the *centroid*, is the mean of points in S_j.

2.2 Lloyd's Algorithm

With the initialized k centroids, the Lloyd's algorithm for k-means [49] conducts the assignment and refinement in an iterative manner until all the centroids do not change.

Initialization. It randomly chooses k points in the space as the initial centroids. Normally, k-means++ [20] is the default initialization method which aims to make k centroids far away from each other.

Assignment. It needs to assign each of the n data points to a cluster with the nearest centroid, and therefore requires $n \cdot k$ number of distance computations.

Refinement. It needs to read every data point in a cluster to update the centroid, hence n data accesses are conducted.

EXAMPLE 1. *Figure 3 shows two centroids c_1, c_2 (red) and five data points (black) bounded by a node N. The assignment step in Lloyd's algorithm computes the distance from every point x_i to every centroid c_j to determine the nearest cluster.*[1]

2.3 Acceleration

Given a dataset and k, there are four types of acceleration for fast k-means with the Lloyd's algorithm:

- **Hardware Acceleration.** Parallelization [70], GPU [69], and cache [23] can accelerate it at physical level.
- **Approximate Acceleration.** It aims to find approximate clustering results within a bounded error w.r.t. the exact result of the Lloyd's algorithm, by developing techniques like sampling [19] and mini-batch [54].
- **Fast Convergence.** It uses efficient initialization techniques such as kmeans++ [20, 22]. As Celebi et al. [28] have done an evaluation on this, it will not be our focus.
- **Exact Lloyd's Algorithm.** It focuses on reducing the number of distance computations in the Lloyd's algorithm, and can be integrated with the above methods to reduce their running time.

Clustering has been evaluated from different perspectives. For example Muller et al. [52] evaluated clustering in subspace projections of high-dimensional data; Hassanzadeh et al. [43] proposed a framework to evaluate clustering for duplicate detection. In contrast, this is the first work that evaluates all accelerating tricks to reduce distance computations in the exact Lloyd's algorithm. Figure 2 summarizes

[1]By default, index i and j refer to data and cluster index respectively in the rest of this paper.

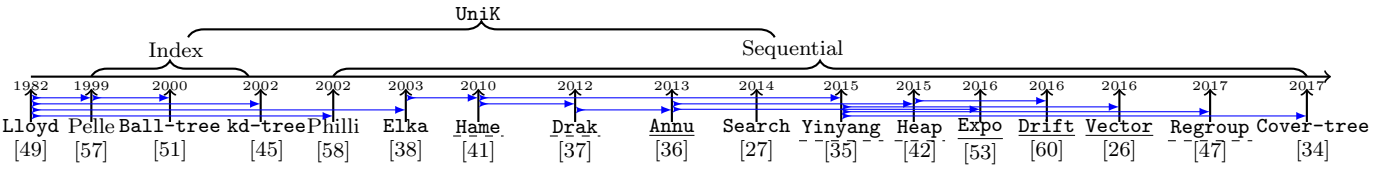
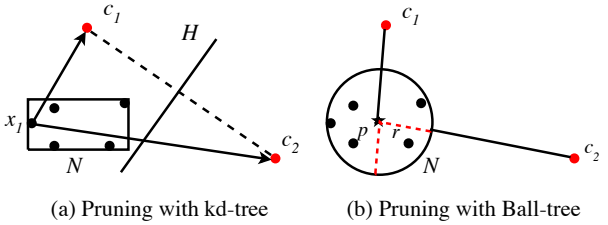


Figure 2: The research timeline of fast Lloyd's algorithms for k-means, where the blue arrows show the successive relationship, and the underlines differentiate two types of sequential methods covered in Section 4.2 (dash lines) and 4.3.



(a) Pruning with kd-tree (b) Pruning with Ball-tree

Figure 3: Example of assignment of node N in kd-tree and Ball-tree, based on a hyperplane H and a ball with radius r.

a timeline of fast Lloyd's algorithms for k-means clustering, which have been published in well-known venues such as ICML, KDD, TPAMI, SDM, UAI, and WSDM.

3. INDEX-BASED ALGORITHMS

By assigning points to the nearest clusters in batch, index-based algorithms have been proposed to support fast k-means, such as kd-tree [45, 57] and Ball-tree [51]. Intuitively, if an index node that covers m data points is assigned to a cluster directly, then roughly $m \cdot k$ number of distance computations and m data accesses can be reduced.

3.1 Typical Indexes

kd-tree. Indexing the dataset using kd-tree [24] can accelerate k-means with batch-pruning for low dimensional data [45, 57], where its intuition is presented in Figure 3(a): node N locates in the hyperplane H of c_1 completely, thus all the points in N are closer to c_1 than c_2 and then N cannot be assigned to c_2. Unfortunately, bisecting hyperplane in a high-dimensional space using kd-tree (Voronoi diagram is a common method) costs much time with a complexity of $\mathcal{O}(k \log k + k^{\lceil \frac{d}{2} \rceil})$ [30].

Ball-tree. Moore et al. [51] proposed to use Ball-tree [63] (a.k.a. metric-tree) to group points using balls, which is a circle in a 2-dimensional space as shown in Figure 3(b). All the points in a ball N are closer to centroid c_1 than c_2 if

$$\|p - c_1\| + r < \|p - c_2\| - r, \tag{2}$$

where p and r are the center and radius of the ball N. The left-hand side of Equation 2 is an upper bound on the distance from the points in N to c_1, while the right-hand side is a lower bound on the distance from the points in N to c_2. However, data points in a high-dimensional space are dispersed, resulting in a much bigger radius r and degenerating its pruning power, i.e., it is unlikely to find a ball N and two centroids c_1 and c_2 such that Equation 2 holds. In this case, using Ball-tree can be even slower than directly assigning each point to a cluster, due to the computational overhead of traversing the Ball-tree.

Other Indexes. Apart from kd-tree and Ball-tree, there are other advanced data structures in metric space that can be used to prune distance computations [29, 61]. This includes Hierarchical k-means tree [40, 66], M-tree [32] and Cover-tree [25, 34], to name a few. We will evaluate all these typical indexes in our experiment (Section 7.2.1).

3.2 Pre-assignment Search

Broder et al. [27] proposed to search around a centroid c_j the data points that lie within a distance threshold, and directly assign these points to c_j, in order to avoid certain distance computations. For example in Figure 3(a), we can search for data points around c_1 within a distance threshold $\frac{\|c_1 - c_2\|}{2}$, and the points found can be assigned to c_1 directly, as they are closer to c_1 than c_2. Here, we can use an index like kd-tree or Ball-tree to conduct a fast *similarity search* [31] to obtain those points within this distance threshold. Similar operation can be conducted on c_2. Then, we can use Lloyd's algorithm to assign the rest of data points to a cluster. For those points that cannot be assigned by this method, we then sequentially scan k centroids to find a cluster to assign them. In this sense, it also belongs to the sequential-based algorithm to be introduced in Section 4. We denote this method as `Search`.

4. SEQUENTIAL-BASED ALGORITHMS

Essentially, sequential-based algorithms try to exploit the triangle inequality to derive various forms of distance bounds, and use them to reduce the number of distance computations in the assignment step of Lloyd's algorithm. However, it is inevitable to access the stored data and update the distance bound of each point. These extra costs actually accounts for a significant portion of the total running time.

4.1 Triangle Inequality

To check whether a point x_i belongs to a cluster of centroid c_j,[2] we can first compare the lower bound on the distance between x_i and c_j, denoted as $lb(i, j)$, against the upper bound on the distance from x_i to its closest centroid $ub(i)$. If $lb(i, j) > ub(i)$, then x_i does not belong to the cluster c_j and we do not need to compute the distance between x_i and c_j. Formally, we denote this inequality as

$$lb(i, j) > ub(i) \rightarrow a(i) \neq j, \tag{3}$$

where $a(i) \neq j$ means that x_i is not assigned to the cluster c_j. Since the pruning is conducted in a pairwise fashion (point, centroid), we call it *local pruning*.

Elkan et al. [38] obtained the lower bound $lb(i, j)$ in the following way: (1) the distance between centroids is used to derive the first lower bound $lb(i, j) = \frac{\|c_{a(i)} - c_j\|}{2}$.[3] We call it as *inter-bound*. (2) The centroid drift from the previous cluster c_j' to the current cluster c_j is used to derive the second lower bound via the triangle inequality: $lb(i, j) =$

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

[3]A similar idea was proposed by Phillips [58] one year earlier.

$\|x_i - c_j^{'}\| - \|c_j^{'} - c_j\| < \|x_i - c_j\|$. We call it as *drift-bound*. (3) Between these two bounds, the bigger one is chosen as the final $lb(i,j)$. We denote the algorithm using both inter-bound and drift-bound as `Elka`.

The *inter-bound* requires to compute the pairwise distance between centroids, and thus costs $\frac{k(k-1)}{2}$ number of distance computations. The *drift-bound* uses $n \cdot k$ units of memory, as it stores a bound for each (point, centroid) pair. The main issues of `Elka` are: 1) it uses much space; 2) the bounds derived might not be very tight. In what follows, we introduce the methods that aim to address each of these issues.

4.2 Methods with a Focus of Less Bounds

4.2.1 Hamerly Algorithm

Instead of storing a lower bound for each centroid, storing a global bound can save much space. Motivated by this, Hamerly [41] proposed to store the minimum lower bound as the global bound. Further, a global lower bound $lb(i)$ for each point x_i is used before scanning every centroid, i.e., $lb(i) = \max\left(\min_{j \neq a^{'}(i)} lb(i,j), \min_{j \neq a^{'}(i)} \frac{\|c_{a^{'}(i)} - c_j\|}{2}\right)$ where $a^{'}(i)$ points to previous cluster of x_i, x_i can stay if: $lb(i) > ub(i) \rightarrow a(i) = a^{'}(i)$. This is known as the *global pruning*, and we denote this algorithm as `Hame`.

4.2.2 Sort-Centers

Beyond a single bound, Drake et al. [37] proposed to store $b < k$ bounds for each point, which can reduce both the storage space and the distance computation by bound. The b points that are selected based on their closeness to the assigned centroid, so it needs to be updated frequently in each iteration. Moreover, the selection of parameter b is also highly dependable on datasets. By default, we use a fixed ratio suggested in [37], i.e., $b = \lceil \frac{k}{4} \rceil$. We denote this algorithm as `Drak`.

4.2.3 Yinyang: Group Pruning

`Hame` needs to conduct $k-1$ local pruning if the global pruning fails, i.e., $lb(i) \leq ub(i)$, Ding et al. [35] proposed to divide k centroids into $t = \lceil \frac{k}{10} \rceil$ groups when k is much bigger than 10, and add a pruning on a group between local pruning and global pruning. This is known as *group pruning*.

In the first iteration, k centroids are divided into several groups based on k-means. Each group will maintain a bound rather than each point. However, the fixed group may lead to a loose bound with the increase of iterations. Kwedlo et al. [47] regrouped the k centroids in each iteration while [35] only did it in the first iteration, and the grouping used a more efficient way than k-means, and the group bounds become tighter. We denote these two algorithms as `Yinyang` and `Regroup`. An index such as Cover-tree [25] can also be used to group k centroids [34].

4.2.4 Storing Bound Gaps in a Heap

Hamerly et al. [42] further proposed a combination of the global lower bound $lb(i)$ and upper bound $ub(i)$ in `Hame`, and used their gap $lu(i) = lb(i) - ub(i)$. It can further reduce the space on storing bounds, but needs a heap for each cluster to hold it dynamically, and the update of this heap incurs extra costs. We denote this algorithm as `Heap`. Then each point will be attached with such a gap $lu(i)$ and inserted into a queue, unless when $lu(i) \geq 0$, it will stay in the current cluster and the distance between all other points and every centroid has to be computed.

4.3 Methods with a Focus of Tighter Bounds

All the above bounds are based on the triangle inequality over a point and multiple centroids. To further tighten the bounds, L2-Norm is used by [26, 36, 42, 60]. We denote the following four algorithms as `Annu`, `Expo`, `Drift`, and `Vector`.

4.3.1 Annular Algorithm: Sorting Centers by Norm

Drake et al. [36, 42] proposed an annular algorithm to filter the centroids directly. Through using the *Norm* of centroids, an off-line sorting can estimate a bound to determine the two closest centroids and tighten the upper and lower bounds. The basic idea is to pre-compute the distance from centroid to the origin (a.k.a. norm $\|c\|$), and further employ the triangle inequality to derive an annular area around the origin which all the candidate centroids are inside:

$$\mathcal{J}(i) = \{j : |\|c_j\| - \|x_i\|| \leq \max\left(ub(i), \|x_i - c_{j^{''}}\|\right)\}, \quad (4)$$

where $c_{j^{''}}$ is the second nearest centroid of x_i.

4.3.2 Exponion Algorithm: Tighter Lower Bound

To further shrink the annular range of candidates [36] around the origin, Newling et al. [53] proposed to use a circle range around the assigned centroid. It filters out the centroids which will not be the nearest, and returns a centroid set $\mathcal{J}^{'}(i)$ which is a subset of $\mathcal{J}(i)$:

$$\mathcal{J}^{'}(i) = \{j : \|c_j - c_{a^{'}(i)}\| \leq 2ub(i) + \|c_{a^{'}(i)} - c_{a^{'}(i)}^{'}\|\}, \quad (5)$$

4.3.3 Tighter Centroid Drift Bound

In the `Elka` method, the centroid drift (e.g., $\|c_j^{'} - c_j\|$ has to be computed every time when the centroid moves to a new one. It is used to update the lower bound of distance from every point to the new cluster $\|x_1 - c_1\|$ in every iteration, while using triangle inequality among x_1, c_1, and $c^{'}$ to update cannot get a tighter bound, and finding a smaller drift is crucial. By replacing this in `Elka`, Rysavy et al. [60] proposed to use the distance between centroid and the origin point (e.g., [0, 0] in two dimension space), and compute a tighter drift δ. Note that the distance from $c_{a^{'}(i)}$ to the origin point, i.e., $\|c_{a^{'}(i)}\|$, can be pre-computed. Here, we only show the drift computation in a 2-dimension case.

$$\delta(i,j) = 2 \cdot \frac{c_{a^{'}(i)}[1] \cdot ra - c_{a^{'}(i)}[2] \cdot \sqrt{\|c_{a^{'}(i)}\|^2 - ra^2}}{\|c_{a^{'}(i)}\|^2} \quad (6)$$

where ra is the radius of the current assigned cluster, i.e., the distance from the centroid to the farthest points in the cluster. It has been proven that $\delta(i,j) < \|c_j^{'} - c_j\|$ [60], then we can update $lb(i,j) = lb(i,j) - \delta(i,j)$. For high-dimensional cases, $c_{a^{'}(i)}[1]$ and $c_{a^{'}(i)}[2]$ can be computed using a more complex conversion in Algorithm 2 of [60], and we will not elaborate it here.

4.3.4 Block Vector

Bottesch et al. [26] calculated bounds based on norms and the Hölder's inequality to have a tighter bound between the centroid and a point. The idea is to divide each data point into multiple blocks of equal size, similar to *dimensionality reduction*. By default, the data point is divided into two blocks, i.e., $x_i^B = \{\frac{\sum_{z=1}^{d/2} x_i[z]}{d/2}, \frac{\sum_{z=d/2}^{d} x_i[z]}{d/2}\}$. Then a tighter

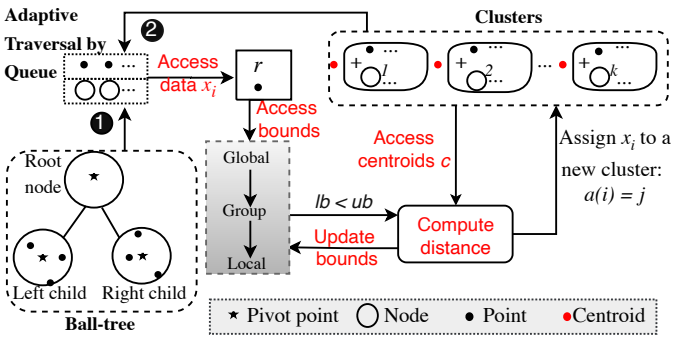


Figure 4: Framework of UniK with a common pruning pipeline

lower bound can be obtained by using the pre-computed norm $\|x_i\|$ and $\|c_j\|$, and inner product of the block vector x_i^B and c_j^B.

$$lb(i,j) = \sqrt{\|x_i\|^2 + \|c_j\|^2 - 2 \cdot \langle x_i^B, c_j^B \rangle} \tag{7}$$

where $\langle x_i^B, c_j^B \rangle$ denotes the inner vector, i.e., $\langle x_i^B, c_j^B \rangle = x_i^B[1] \cdot c_j^B[1] + x_i^B[2] \cdot c_j^B[2]$.

5. EVALUATION FRAMEWORK

After reviewing the sequential-based methods, we conclude a common pruning pipeline in Figure 4. Five core operators (highlighted in red color) execute in the following order: (1) access the data point, (2) then access the global bound to see whether lb is bigger than ub – (3) if yes, the point maintains in the current cluster; (4) otherwise, access group and local bounds to determine whether to read centroids and compute distances to find the nearest one to assign, and (5) then update the bounds.

Next, we will present an evaluation framework (UniK) based on the above pipeline, as shown in Algorithm 1. UniK supports multiple traversal mechanisms to smoothly switch between different index and sequential methods. Note that the above pruning pipeline mainly works for point data, and existing index methods need to scan k centroids without bound-related operations. Hence, before describing UniK, we will introduce an optimized assignment such that the bound-based prunings on both nodes and points can well fit into UniK in a unified way, followed by an optimized refinement without any more data access. Our experiments show that UniK with these optimizations can further accelerate k-means.

5.1 Optimizations in UniK

5.1.1 The Assignment Step

To scan all nodes and points simultaneously, we define an advanced node that shares exactly the same property with the point, and meanwhile is attached with more information that can help prune via bounds.

Advanced Index. Given a tree-structured index T built from the dataset D, there are usually three kinds of nodes: root node N_{rt}, leaf node N_l, and internal node N_i. To enable the pruning capability for k-means, we enrich each node with some extra information as defined below:

DEFINITION 1. *(**Node**) $N = (p, r, sv, \psi, L_N, L_P, num, h)$ covers its child nodes L_N if N is an internal or root node, or a set of child points L_P if N is a leaf node.*

where pivot point p is the mean of all the points, radius r is the distance from p to the furthest point in N, the sum vector sv represents the sum of all the points under N, $\psi = \|N^{'}.p - p\|$ is the distance from p to the pivot of its parent node $N^{'}$, num is the number of points covered in the range of N, and h is N's height (or depth). We also analyze the space cost of this index in our *supplementary material* (see Section A.2). To accommodate the assigned points and nodes, we further enrich the clusters as below:

DEFINITION 2. *(**Cluster**) $S_j = (c_j, sv, L_N, L_P, num)$ covers a list of nodes L_N and points L_P, and num is the number of points in S_j, i.e., $num = |L_P| + \sum_{N \in L_N} N.num$.*

Node Assignment. Given a node N, we will assign N to a cluster S_j if the gap between the distance from pivot p to the nearest centroid $c_{n_1(p)}$ and the distance from p to the second nearest centroid $c_{j''}$ is larger than $2r$, i.e.,

$$\|p - c_{j''}\| - r > \|p - c_{n_1(p)}\| + r \rightarrow a(N) = n_1(p), \tag{8}$$

To achieve the above condition, a straightforward way is to scan k centroids to find the two nearest centroids for p, similar to [51]. To further avoid distance computation and shrink the range, we use the idea of group pruning (Equation 5) based on the pivot point p to prune those centroids that cannot be the nearest two. Before the group pruning, we use a global pruning similar to Equation 8 by combining r:

$$lb(p) - r > ub(i) + r \rightarrow a(N) = a^{'}(N), \tag{9}$$

After the global and group pruning, we use the local bounds by combining the radius r again to avoid the distance computation between the pivot point p and centroids:

$$lb(p,j) - r > ub(p) + r \rightarrow a(N) \neq j, \tag{10}$$

By comparing the bounds for point and node, we combine them into a single pipeline by setting $r = 0$ when the checked object is a point. Node assignment differs from point assignment in the way that, the node needs to be further split and checked if it is not pruned by Equation 8 and 9.

Before computing the distance when the child node or the point is scanned, we estimate the child's pivot bounds to the centroids based on the parent-to-child distance ψ using triangle inequality, which will be further used to prune in a new round without computing the real distance. Specifically, let N_c denote one child of N, then the upper bound and lower bound of N can be passed to N_c by:

$$\begin{aligned} &lb(N_c.p, j) = lb(N.p, j) - N_c.\psi, \\ &lb(N_c.p) = lb(N.p) - N_c.\psi, \quad ub(N_c.p) = ub(N.p) + N_c.\psi. \end{aligned} \tag{11}$$

5.1.2 The Incremental Refinement Step

To update centroids after all the points are assigned, a traditional refinement will read all the points in each center again, sum them up and get the mean vector. Hence, the whole dataset needs to be scanned again in each iteration. Ding et al. [35] optimized this by updating the previous centroids with those points that changed clusters only, so only a subset of points will be re-accessed. However, we can save this cost if maintaining a sum vector in the assignment, which has scanned all the points.

More specifically, we maintain a sum vector sv that will be updated when an object (node or point) moves in or out during the assignment step. Such an update will be rare when the iteration goes deeper, as only some points change their clusters across iterations. Since index node is assigned in batch, then in order to avoid accessing all the points under

it, each node's sum vector sv and the number of points ψ can be computed in advance by an iterative procedure [51] once the index is built. For example, as shown in the right part of Figure 4, when an object is assigned to a new cluster, we will update the sum vector (the symbol of "+") of the two clusters which this object moves in and out accordingly.

5.2 Detailed Framework Design

Algorithm 1 presents a general form of our unified framework under `UniK`, which is composed of a queue data structure and four core functions. The queue $\mathcal{Q}$ is used to hold both the points and the nodes (called as *object* for unity purpose) to be assigned. For example, when a node cannot be assigned by Equation 8, its child nodes are enqueued. The four core functions are elaborated as follows.

`Initialization.` After randomly selecting k points as the centroids c (line 21), we assign the root node N_{rt} to the first cluster S_1 if an index has been built (line 23); otherwise, we temporally store the whole point set D in S_1 (line 26). Then in each cluster, all the nodes and points are pushed into $\mathcal{Q}$ (line 5). An object o polled from the queue $\mathcal{Q}$ is pruned in the following manner: the global pruning is first conducted in line 11, we assign o (denoted as $a(o)$) if the upper bound is smaller than the lower bound minus twice of the radius.

`GroupLocalPruning.` If the global pruning fails, we use the group pruning in line 28 to filter partial centroids. Then we conduct the local pruning (line 32); if it fails, we compute the distance from the centroid to the pivot point, and update the two nearest neighbors' distances (line 35).

`AssignObject.` After the scanning of centroids, we obtain the gap between the distance to the two nearest clusters. Then, we check whether the object is a node, and compute the gap using Equation 8 to see whether we can assign the whole node (line 38). If not, we further split the node and push all its children into the queue (line 41–42). If it can be assigned or the object is a point, we will further check whether the object should stay in the current cluster (line 45); otherwise, we update the cluster (line 46).

`Refinement.` We divide the sum vector sv by the number of points inside to update the centroid in line 50. The refinement can be done without accessing the dataset again, as we have updated the cluster's sum vector in line 46 when the object cannot remain in the current cluster.

5.3 Multiple Traversal Mechanisms

The symbols ⚙ in Algorithm 1 are knobs that indicate whether to apply pruning techniques used in various index and sequential methods which we evaluate in this paper. By turning certain knobs on, we can switch to a specific clustering algorithm. Formally, we formulate it as below:

Definition 3. ***(Knob Configuration)*** *A knob ⚙ = {0,1} controls a setting (selection), e.g., whether to use index or not, or whether to use block vector or not. A knob configuration $\theta \in \Theta$ is a vector of all knobs in Algorithm 1, e.g., $[0, 1, 0, \cdots, 1]$, where Θ denotes the configuration space.*

Most existing algorithms can fit into `UniK` by projecting to a specific knob configuration, and our optimization proposed in Section 5.1 can be activated by a new knob configuration.[4]

[4]By enabling all bound knobs, we will get the *Full* method in Figure 1. We also argue that this formulation will be useful to cater for more new configurations in Θ to be explored by future studies, but it is not the focus of this paper.

Algorithm 1: `UniK-`k`-means` (k, D)

Input: k: #clusters, D: dataset.
Output: k centroids: $c = \{c_1, \ldots, c_k\}$.

```
1  Initialization(t, c, Q, S);
2  while c changed or t < t_max do
3      for every cluster S_i ∈ S do
4          ⚙Update lb(i, j) with the centroid drifts;
5          Q.add(L_N, L_P);
6          while Q.isNotEmpty() do
7              o ← Q.poll(), r ← 0;
8              ⚙Update ub and lb;
9              if ⚙o is node then
10                 r ← o.getRadius();
11             if ⚙lb(i) − r > ub(i) + r then
12                 o stays in current cluster: a(o) ← a'(o);
13             else
14                 ⚙gap = GroupLocalPruning(o);
15                 AssignObject(o, S, Q, gap);
16     Refinement(S);
17     t ← t + 1;
18 return {c_1, ..., c_k};
19 Function Initialization(t, c, Q, S):
20     t ← 0, Q ← ∅;
21     Initialize centroids c = {c_1, ··· , c_k};
22     if ⚙N_rt ≠ null then
23         S_1.add(N_rt, N_rt.v);                    // index [45]
24     else
25         ⚙Search on each centroid;                 // Search [27]
26         S_1.add(D);
27 Function GroupLocalPruning(o):
28     ⚙Get J(o) (J'(o)) by Equation 5 // Expo, Annu
29     min ← +∞, min_2 ← +∞;
30     for every centroid c_j ∈ J(o) do
31         ⚙Pass group pruning;  // Yinyang, Regroup
32         if ⚙ub(i) > lb(i, j) then
33             lb(i, j) ← ‖o − c_j‖;
34             if lb(i, j) < min then
35                 a(i) ← j, min ← lb(i, j), min_2 ← min;
36     return min_2 − min;
37 Function AssignObject(o, S, Q, gap):
38     if ⚙o is node and Equation 8: gap < 2r then
39         S_a'(i).reduce(o.v, o);
40         for every child N_c ∈ o.L_N do
41             Update N_c's bound by Equation 11;
42             S_a'(i).add(N_c.sv, N_c);
43             Q.push(N_c);
44     else
45         if a'(i) ≠ a(i) then
46             S_a'(i).reduce(o.sv, o);
47             S_a(i).add(o.sv, o);
48 Function Refinement(S):
49     for every cluster S_j ∈ S do
50         c_j ← S_j.sv / S_j.num;
```

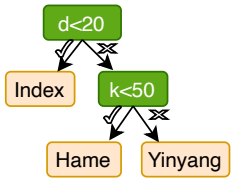


Figure 5: A basic decision tree (BDT) for simple algorithm tuning.

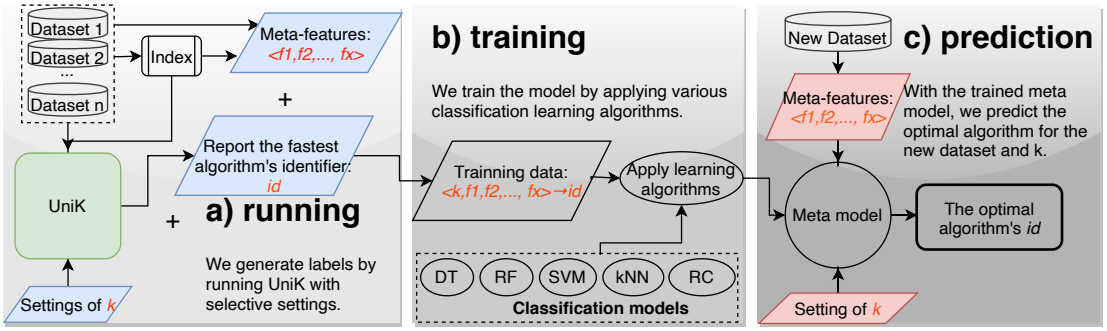


Figure 6: Three modules of `UTune`: a) selective running, b) meta-model training, and c) prediction.

For example in Figure 4 we present two traversal mechanisms (i.e. parts "①" and "②") based on knobs, where we traverse from the root node by using ① in the first iteration, but in the following iterations we traverse from the nodes maintained in the current cluster by using ②. This is because the pruning effect is not always good especially for high dimensional data, and it will still cost extra time in the tree traversal in next iteration if starting from the root node. If most nodes in the index can be pruned, traversal from the root (i.e., ①) in each iteration can gain better performance than scanning nodes from clusters.

To be more adaptive, in the first and second iterations of `UniK`, we can traverse from the root ① and current nodes in the clusters ②, respectively (as shown in Algorithm 1). Then we compare the assignment time of these two iterations. If the time spent on the first iteration ① is bigger than that on the second iteration ②, we give up traversing from the root in subsequent iterations, and scan the current nodes and points maintained in each cluster; otherwise, we empty the point and node list of each cluster, and then push the root node at the end of each iteration to keep benefiting from the index in multiple iterations. We denote these two traversal styles as *index-single* and *index-multiple*.

By default, we set the index as Ball-tree and set the bound configuration same as Yinyang (Section 4.2.3). Our experiments in Section 7.2.3 and 7.3 show the the superiority of such a configuration over both Ball-tree and Yinyang in some datasets. When we disable all the bound configurations, it will be a pure index method [51]. Hence, besides multiple knobs of bound configuration, we also have four configuration knobs on the index traversal: 1) not using index; 2) pure; 3) index-single; 4) index-multiple. Next, we will further discuss how to choose the right knobs to turn on.

6. CONFIGURATION AUTO-TUNING

In this section, we study how to select a fast algorithm for a given clustering task. This is critical to both an evaluation framework (like this paper) and practitioners. Our literature review shows that existing studies [34, 35] rely on simple yet fuzzy decision rules, e.g., not using index method for high-dimensional data or choosing `Yinyang` when k is big. In Figure 5, we illustrate a basic decision tree (BDT) based on these rules. However, our experiments show this BDT does not work very well.

Therefore, we choose to train an ML model based on our evaluation data to automatically select a fast algorithm for a given clustering task. The algorithm selection is equivalent to a problem of tuning "knobs" in our evaluation framework `UniK` (Algorithm 1), in the sense that each of the existing algorithms corresponds to a unique knob configuration.

Table 1: A summary of features $\boldsymbol{F} = \{f1, f2, \cdots, fx\}$.

Type	Feature	Description	Normalize
Basic	n	The scale of dataset	-
	k	Number of clusters	-
	d	Dimensionality of dataset	-
Tree	$h(T)$	Height of index tree T	$\log_2 \frac{n}{f}$
	$\lvert N_i\rvert, \lvert N_l\rvert$	#Internal & leaf nodes	$\frac{n}{f}$
	$\mu(h)$, $\sigma(h)$	Imbalance of tree	$\log_2 \frac{n}{f}$
Leaf	$\mu(r)$, $\sigma(r)$	Radius of leaf nodes	$N_{rt}.r$
	$\mu(\psi)$, $\sigma(\psi)$	Distance to parent node	$N_{rt}.r$
	$\mu(\lvert L_p\rvert)$, $\sigma(\lvert L_p\rvert)$	#Covered points in N_i	f

An Overview. We model the knob configuration (algorithm selection) problem as a typical classification problem, where our goal is to predict the best knob configuration for a given clustering task. To this end, we extract meta-features to describe clustering datasets, and generate class labels (i.e., ground truth) from our evaluation logs that contain the records of which configuration performing the best for a particular dataset. We then feed the meta-features and class labels into an off-the-shelf classification algorithm to learn a mapping from a clustering dataset (with certain features) to the best performing algorithm configuration. We can then use the trained model to predict a good configuration for clustering a given dataset. We name our auto-tuning tool as `UTune`, and illustrate its three modules in Figure 6.

6.1 Generating Training Data

Class Label Generation. Since the knob configuration space is large, it is computationally intractable to try all possible knob configurations and label the dataset with the best-performing configuration. Thus, we only focus on a few knob configurations corresponding to the high-performing existing methods as our selection pool [64]. The pseudocode of our selective running can be found in our *supplementary material* (see Algorithm 2), and the main idea is three-fold: 1) we limit the number of iterations t_{max} as the running time of each is similar after several iterations (see Figure 13); 2) we exclude those algorithms that have low rank during our evaluation, e.g., `Search` [27], and our experiments (see Figure 12) show that only five methods always have high ranks; 3) we test index optimizations if the pure index method outperforms sequential methods. Such a selective running enables us to generate more training data within a given amount of time, and thus further improves the prediction accuracy (see Table 5).

Meta-Feature Extraction. We extract a feature vector $\boldsymbol{F}$ to describe (or represent) a dataset, such as dimensionality, scale, and k shown in Table 1.

In addition to the basic features, we also extract novel and

Table 2: An overview of datasets, the index construction time (second), and #nodes of Ball-tree.

Ref.	Name	n	d	Time	#Nodes	Used by
[1]	BigCross	1.16M	57	10.8	183k	[62]
[2]	Conflong	165k	3	0.26	21.8k	[53]
[11]	Covtype	581k	55	3.87	88.3k	[41, 42, 38]
[3]	Europe	169k	2	0.27	11.2k	[53]
[14]	KeggDirect	53.4k	24	0.17	2.8k	[53, 35]
[13]	KeggUndirect	65.5k	29	0.31	4.5k	[53, 35]
[5]	NYC-Taxi	3.5M	2	8.7	228k	-
[15]	Skin	245k	4	0.33	21.2k	[53]
[12]	Power	2.07M	9	4.3	43.7k	[34]
[10]	RoadNetwork	434k	4	0.55	6.9k	[35]
[16]	US-Census	2.45M	68	204	135k	[53]
[9]	Mnist	60k	784	4.8	7.3k	[53, 41, 35]

more complex features that can capture certain properties of data distribution based on our built index. Recall the pruning mechanisms in Section 5, they work well when data shows good assembling distribution. The index construction actually conducts a more in-depth scanning of the data and reveals whether the data assemble well in the space. Specifically, the information we got includes the tree depth $h(T)$, number of leaf nodes $|N_l|$, number of internal nodes $|N_i|$, and imbalance of tree (i.e., the mean value and standard deviation of all leaf nodes' heights h).

Further, we extract more features of all the leaf nodes, including the radius of nodes r, the distance from child to parent nodes ψ, and the number of points in the leaf nodes $|L_p|$, during index construction. Specifically, we select all the leaf nodes N and extract their $(r, \psi, |L_p|)$, then further compute their mean μ and standard deviation σ, which are all normalized by their maximum values in the tree.

6.2 Meta-Model Training and Prediction

We model the algorithm selection as a multi-label classification problem in terms of using an index or not, and using which one of the five bound configurations. The prediction needs to be conducted in two parts based on our two ground truth files which shows the rank of various index and bound configurations, respectively. Firstly, we predict the optimal bound configuration. Secondly, we predict the index configuration and whether the bound should be combined, as mentioned in Section 5.3. Then we combine the results from these two predictions and generate the final configuration. For a new clustering task, we can extract the features based on our built index (e.g., Ball-tree), and use the learned model to predict a high-ranked knob configuration of algorithms.

7. EXPERIMENTAL EVALUATIONS

Our evaluation seeks to answer the following questions:

- Which index structure is proper for index methods?
- How does the performance of sequential methods vary?
- Can our evaluation framework `UniK` enable further improvement on the performance of existing clustering methods?
- Can `UTune` predict a fast algorithm through learning?

7.1 Experimental Setup

Implementation. We implemented all the algorithms in Java 1.8.0_201, and used Scikit-learn 0.22.2 [4, 39] to train our classification models. All experiments were performed on a server using an Intel Xeon E5 CPU with 256 GB RAM running RHEL v6.3 Linux. Our code is available at [6] for reproducibility, which can also be found from the *supplementary material* of this submission with instructions (see Section A.4).

Measurement. Same as the traditional methods [35, 55], we measure the running time and the percentage of pruned distance computation (i.e., pruning power). In order to run more rounds of experiments, we record the total running time of the first ten iterations (after which the running time usually becomes stable as shown in Figure 13). Moreover, we measure #data access, bound access, bound updates, and footprint. For each measurement above, we report the average value across ten sets of randomly initialized centroids using k-means++ [20].

Datasets. We select a range of real-world datasets (Table 2), most of them are from the UCI repositories [21], and also used in the experiments of the state-of-the-art such as [35, 53]. Moreover, we introduce several recent new datasets, including pick-up locations of NYC taxi trips. The hyperlink for each dataset can be found in the reference.

7.2 Evaluation of Existing Methods in UniK

7.2.1 Index Methods

We implemented five indices: kd-tree, Hierarchical k-means tree (HKT) [40], Ball-tree [56], M-tree [32], and Cover-tree [25] covered in Section 3.1. The latter four that bound points by a radius can be easily extended to support `UniK`.

Index Construction. Figure 7 compares these five indices over BigCross, and shows the construction time and clustering time w.r.t. the dimension d and the data scale n, respectively. Here, we set $n = 10,000$ when varying d, and M-tree is quite slow for a bigger n; that also explains why we ignore its performance.

Observations. (1) With the increase of d and n, the construction time increases and is more sensitive to n, but it is still tolerable for Ball-tree, Cover-tree, and kd-tree. (2) In average, Ball-tree is the fastest in clustering and 2nd fastest in index construction. (3) Even though kd-tree is the fastest in index construction, its leaf-nodes can only cover one point whereas Ball-tree can cover multiple points. Thus, kd-tree has many more nodes than Ball-tree (the ratio is around the capacity $f = 30$ according to our experiments), and Ball-tree is more applicable for large-scale clustering. (4) Columns 5 and 6 of Table 2 also show the index construction time and the number of nodes of Ball-tree. We find that the index of most datasets can be built within ten seconds, which means the delay of index building before clustering is tolerable.

Clustering Efficiency. *Observations*. (1) With the increase of data scale, the cost of clustering rises dramatically for every index, but Ball-tree still beats other indices. (2) When the dimensionality increases, kd-tree's performance degrades the most due to its complex pruning mechanism by using hyperplane; other indices are not impacted much as they use a radius to prune without any extra cost. (3) A bigger k makes the clustering slower, in a linear manner.

Our Choice. Thus, we choose Ball-tree [56] as `UniK`'s default index structure and conduct a further comparison with sequential methods and our proposed `UniK` and `UTune`. The space cost of the index is not high compared with most sequential methods, which will be presented in Figure 10. Moreover, setting a proper capacity f for leaf nodes in M-tree, HKT, and Ball-tree is crucial,[5] and a larger f can save more space as the number of nodes will decrease. To balance the space and efficiency, we set a relatively small capacity (i.e.

[5]Cover-tree and kd-tree do not have a parameter on capacity.

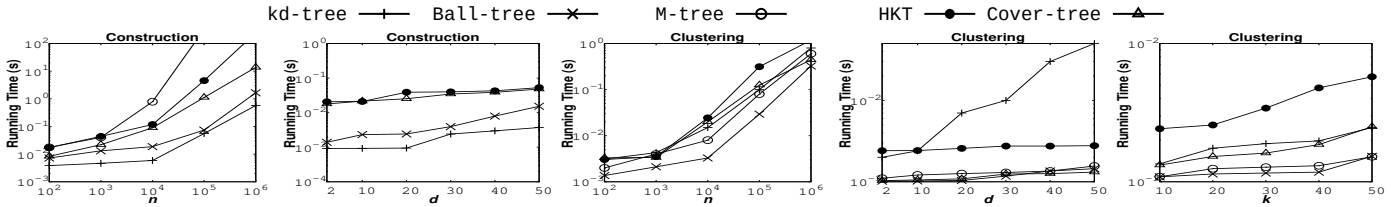


Figure 7: Construction and clustering performance of five index structures.

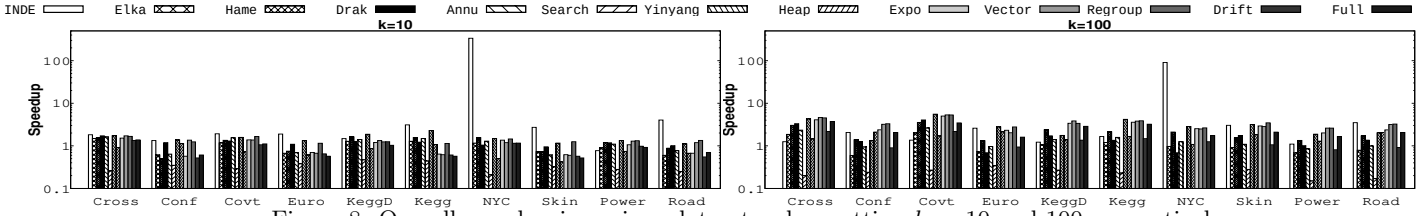


Figure 8: Overall speedup in various datasets when setting k as 10 and 100, respectively.

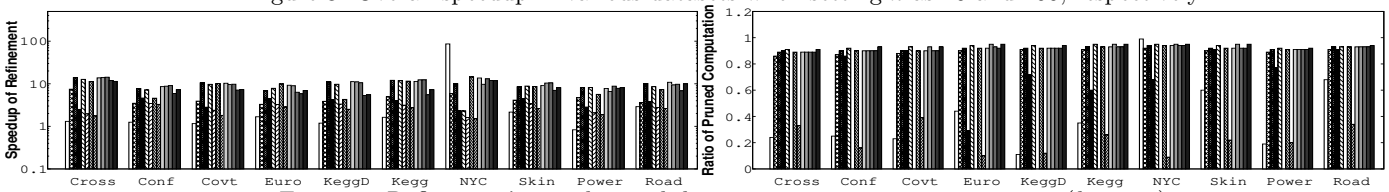


Figure 9: Refinement's speedup and distance computation pruning ratio ($k = 100$).

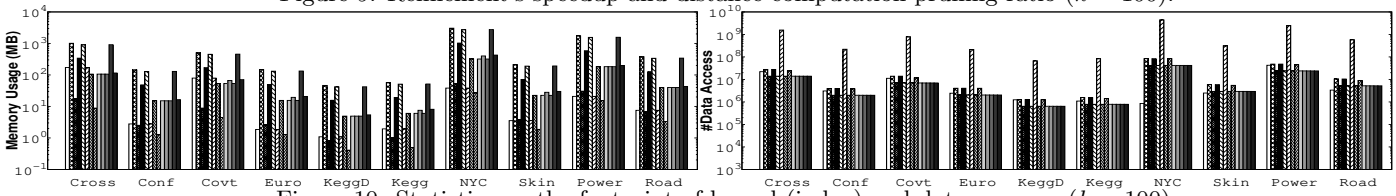


Figure 10: Statistics on the footprint of bound (index) and data accesses ($k = 100$).

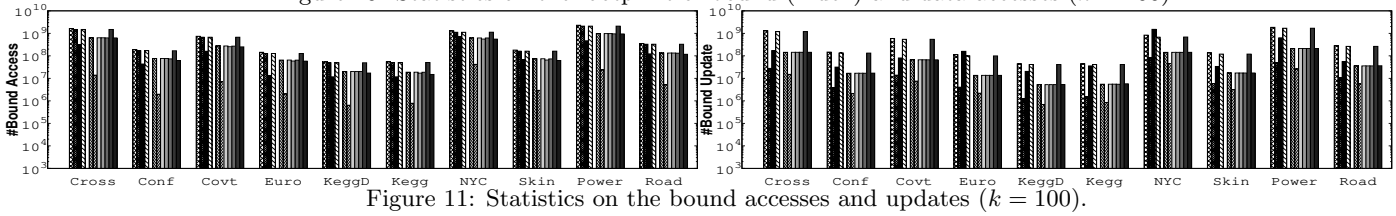


Figure 11: Statistics on the bound accesses and updates ($k = 100$).

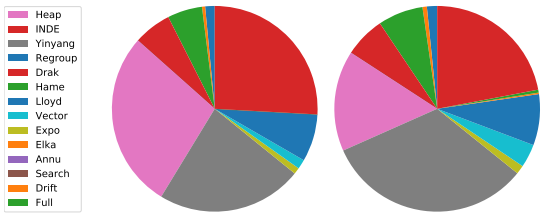


Figure 12: Leaderboard of sequential methods as top 1 & 3.

30), and a sensitivity test of capacity will be conducted later in Figure 14, together with `UniK`. It shows that the clustering efficiency is not affected much by the capacity.

7.2.2 *Sequential Methods*

Learderboard. Figure 12 shows a leaderboard of various sequential methods by setting different parameter settings (k, n, and d) across all the datasets in Table 2. We can observe that five sequential methods have competitive performance: `Hame`, `Drak`, `Heap`, `Yinyang`, and `Regroup`.

Our Choice. Thus, we will use these five sequential algorithms as the selection pool in our auto-tuning model `UTune`.

Speedup and Pruning Ratio. We first investigate the speedup and the pruning ratio in distance computation. Figure 8 shows the overall speedup over the Lloyd's algorithm, compared with the representative index method (`INDE`): Ball-tree. Since assignment occupies most time of the clustering time and it shows a similar trend with the overall speedup, we ignore it here, and it can be found in our *supplementary material* (see Section A.3). Interestingly, the improvement of refinement (Figure 9) using our incremental method significantly improves the efficiency for all algorithms.

On the low-dimensional NYC dataset, we observe that the index method can beat all existing sequential methods in term of running time. This also occurs in several relatively high-dimensional datasets when $k = 10$, such as KeggD and Kegg. Among all the sequential methods, the `Regroup` method and `Yinyang` are two fastest methods on most datasets. We also observe that the speedup is not consistent with the pruning ratio, e.g., the index is almost 150 times (it is 400 when $k = 10$) faster on NYC, while its pruning ratio is only 10% (30% when $k = 10$) more than others. A reverse trend happens in the fully optimized algorithm (`Full`) which has the highest pruning ratio but very low efficiency.

The above observations verify our claims that **index method can be very fast, and a higher pruning ratio cannot simply guarantee better performance**. Next, we measure the data accesses, bound accesses and bound updates, which can cost as much as the distance computation.

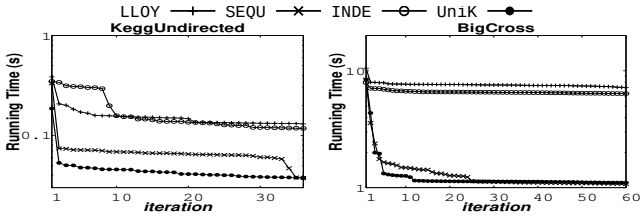


Figure 13: Running time of each iteration over the KeggUndirected ($d = 29$) and BigCross dataset ($d = 57$).

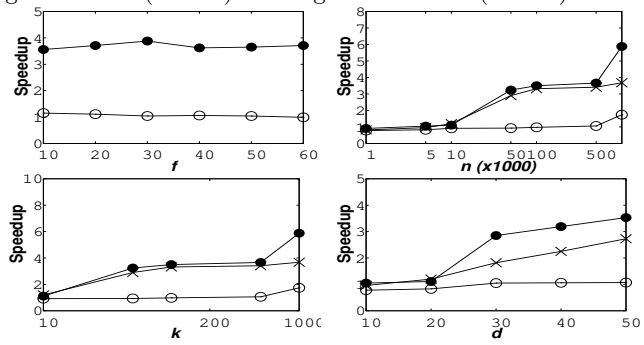


Figure 14: Sensitivity test on capacity f, data scale n, k, and dimension d over the BigCross dataset.

Role of Data Access & Bound Access/Update. Figure 10 presents the memory usage of each algorithm. We find: among all the space-saving sequential methods, `Heap` can save the most, and index method's overhead is also much less than those sequential methods, and it will not increase with the rise of k. Moreover in Figure 11, the index method has much less data access, which can explain why it is much faster; `Yinyang` has much less bound access and update which can explain why it is faster than other methods that have very similar pruning ratio. This also reveals that **data access, bound access and bound updates are three crucial factors that should also be considered in the future algorithm design and evaluation**. A similar performance breakdown of setting $k = 10$ and more analysis can be found in our *supplementary material* (see Section A.3).

Table 3: Bound and data accesses in the first iteration.

Dataset	Criteria	Lloyd	SEQU	INDE	UniK
Cross $k = 100$	Time (s)	96.0	33.1	55.3	16.3
	Pruned	0	84%	45%	91%
	Bound	0	1.5B	0	0.9B
	Point	100M	30M	15.8M	9.8M
	Node	0	0	931k	108k

7.2.3 Our Unified Method UniK

Next, we compare `UniK` with (arguably the best) index method (`INDE`): `Ball-tree` and sequential method (`SEQU`): `Yinyang`, and conduct a series of fine-grained evaluations on specific datasets. Detailed results on each dataset are in Table 6 when comparing with `UTune` later.

Running Time Per Iteration. Figure 13 shows the running time on each iteration. We observe that the time spent per iteration decreases sharply in the first few iterations and then becomes stable. `UniK` is the fastest because of our adaptive traversal mechanism, and `INDE` and `SEQU` dominate each other in two datasets respectively.

Access on Data and Bound. Specifically, we count the number of bound accesses, data accesses, and distance computations for BigCross, as shown in Table 3. The number of bound accesses and data accesses of `SEQU` is much higher than `INDE` and `UniK`, and `UniK` has the minimum number of accesses. For example, `SEQU` needs 1.5 billion number of bound accesses, and `UniK` only needs 0.9 billion, as most points have been pruned by index.

Robustness to Parameters. We test the robustness of `UniK` to various parameters in Figure 14, especially the capacity of the leaf nodes. By increasing the capacity, the performance decreases slightly but does not fluctuate much. Like other two methods, `UniK`'s speedup rises slightly when increasing n, d, and k.

7.2.4 Summary of Lessons Learned

From the above evaluations, we first summarize some key insights that might differ from existing studies:

- Ball-tree is very fast in both construction and clustering. For low-dimensional spatial datasets such as NYC, Ball-tree can beat all sequential methods to a great extent. Moreover, Ball-tree also wins in high-dimensional datasets such as BigCross and Kegg when k equals 10. This is because the data points assemble well, and having a small r helps the pruning in batch. Hence, by choosing the right index structure (Ball-tree), the index method can also perform well in high-dimensional data, which breaks the traditional conclusions towards the index methods [35, 53].
- Among all the sequential methods, there are five methods alternately being the fastest in various clustering tasks. They are: `Hame`, `Drak`, `Heap`, `Yinyang`, and `Regroup`. As a common target on reducing space on bounds, they also reduce the overhead to maintain and update those bounds, which further leads to faster performance.
- Several recently proposed tight bounds do not work well (e.g. [26, 60]), and their space consumption is also high. The main possible reason is that frequent bound access, comparison, and expensive updates are needed, and large amount of bounds have to be stored which further increases the maintenance costs. Recall Section 4.3, we can also see that it is complex to compute and update these bounds.

We also have several new and deeper insights:

- By integrating the index and sequential methods, `UniK` achieves an average of 53% performance improvement in our tested clustering tasks, and even up to 73% when $k \geq 100$ (details in Table 6). Compared with index methods, it avoids intensive distance computations when assigning nodes by bound-based pruning. Compared with sequential methods, it prevents intensive data accesses and distance computations simultaneously using batch pruning.
- We have shown the aggregate rank of each method over all datasets in term of efficiency using pie charts in Figure 12. Further, we rate them in Table 4 based on multiple metrics we used, and more experiments results are not shown due to space limit. Here we do not list other index methods as Ball-tree clearly dominates the rest. We divided the metrics into two groups: beginners and researchers, where the former is for general data mining users, and the latter is for research purposes on accelerating k-means over existing algorithms. With our rates, users can choose a proper algorithm based on their demands.
- Besides the metrics we have evaluated, Table 4 also includes the robustness to various clustering tasks and ease of implementation that system engineers can refer to. For

Table 4: Evaluation Summary of Section 7.2. The darker the circle ∘, the higher degree of its corresponding criteria.

Users	Beginners			Researchers			Domain		
Criteria / Algorithm	Leaderboard	Space-saving	Parameter-free	Fewer data access	Fewer bound access	Fewer distance	Robustness	Ease of implementation	Analysts
Ball-tree [51]	●	◉	◉	●	●	◉	○	◉	🌍
Elka [38]	○	○	●	⊙	⊙	⊙	⊙	◉	
Hame [41]	◉	◉	●	⊙	◉	◉	⊙	◉	
Drak [37]	◉	◉	◉	◉	◉	◉	◉	⊙	
Annu [36]	○	◉	●	◉	⊙	⊙	◉	◉	
Heap [42]	●	●	●	◉	◉	◉	◉	⊙	💻
Yinyang [35]	●	◉	◉	◉	◉	◉	◉	⊙	🖥
Expo [53]	○	◉	●	⊙	⊙	⊙	⊙	◉	
Drift [60]	⊙	⊙	●	⊙	⊙	⊙	⊙	⊙	
Vector [26]	⊙	⊙	◉	⊙	⊙	⊙	⊙	◉	
Regroup [60]	◉	◉	◉	◉	◉	◉	◉	⊙	
UniK	●	◉	◉	◉	◉	◉	●	⊙	🖥

domain-specific analysts, we highly suggest spatial analysts (🌍) to adapt Ball-tree as it can bring significant acceleration for low-dimensional spatial data. Based on users' computing devices, we recommend that analysts with servers (🖥) can choose `Yinyang` or our `UniK`, and `Hame` will be proper for laptops (💻) with limited memory.

7.3 Evaluation of Auto-tuning

Based on the above comprehensive evaluations on the efficiency of index methods and sequential methods, we choose two representative algorithms, one for each class, denoted as `INDE` (Ball-tree) and `SEQU` (`Yinyang`).[6] Next we compare them with `UniK` (with our default index and bound configurations), and our `UTune` with various learning models.

7.3.1 Model Training

Ground Truth Generation. Since available public datasets for clustering are not that many, when running clustering methods to generate the ground truth, we alter $k = \{10, 100, 200, 400, 600, 800, 1000\}$, $d = \{10^3, 10^4, 10^5, 10^6\}$, and $n = \{10, 20, 30, 40, 50\}$ over all the datasets in Table 2. Figure 15 presents the efficiency of the full running and selective running (proposed in Section 6.1 and we used five methods) on the datasets chosen based on our leaderboards (see Figure 12). We can observe that selective running is much faster, as it skips those slow methods and saves much time to run more parameter settings and get more training data, which can further improve the prediction precision, as later exhibited in Table 5.

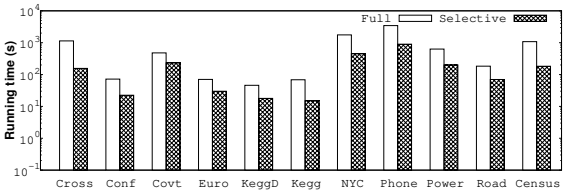


Figure 15: Efficiency comparison on ground truth generation.

Adopted ML Models. With the ground truth obtained, we adopt most classical classification models [46] for training and prediction: decision trees (**DT**), random forests (**RF**), k nearest neighbor (**kNN**), support vector machine (**SVM**), and Ridge linear classifier (**RC**). We divide the ground truth into two parts: 70% for training and 30% for testing.

Prediction Accuracy. We adopt a rank-aware quality metric called *mean reciprocal rank* (**MRR**) [33] to measure the precision. Given a set of testing records R, our loss function is defined as below:

$$MRR(R) = \frac{1}{|R|} \cdot \sum_{l \in R} \frac{1}{rank(p_l)} \tag{12}$$

where $rank(p_l)$ denotes the ranking of prediction p_l in the labeled ground truth.

Table 5 shows the MRR of different models (the BDT in Figure 5, DT, RF, SVM, kNN, RC) in predicting the index configuration (i.e. Index@MRR) and the choice of bound to be used (i.e. Bound@MRR). The MRR result can be further interpreted from two dimensions: (i) For each model, we distinguish the MRR precision over the training data obtained from the full running and the selective running (highlighted as a prefix "S-"), respectively; (ii) For each model, we keep adding three groups of features in the training phase, namely *basic features*, index features (*Tree*), and advanced features on leaf level (*Leaf*), to verify their effectiveness. Details on these features are in Table 1. For completeness purpose, we also report the training and prediction time in our *supplementary material* (see Table 8).

Observations. (1) Within the same limited time, selective running has higher precision and can achieve 92% if using decision tree (with a depth of 10) or SVM, while BDT that relies on fuzzy rules only achieves 43%. This is because selective running manages to generate more training records than full running (e.g., 1600 vs. 436 in this case). (2) With more index and leaf features, we can have a higher precision than using basic features only when using the selective running ground-truth. (3) Among all the classifier models, the decision tree has the highest precision and it is also very fast for both training and prediction.

7.3.2 Verification

Among all the prediction models of high accuracy, we select the decision tree (DT) to support `UTune`. Then, we compare `UTune` with the representatives: `INDE`, `SEQU`, `UniK`, to verify whether our predicted configuration works well. Table 6 presents the running time of Lloyd's algorithm, and the speedup brought by `INDE`, `SEQU`, `UniK`, and `UTune` over Lloyd's. The percentage of the pruned distance computations is shown below the speedup. In our *supplementary material*, we also show the corresponding assignment and refinement time (see Section A.3).

Observations. (1) On average, `UniK` outperforms the index and sequential methods in multiple cases, especially when k is big, and the integration of bound and indexing further improves the pruning ratio. (2) `UniK` cannot always be fast over high-dimensional data such as Power, which is also consistent with the *no free lunch theorem* [67]. The main reason is that the index cannot prune well in the first iteration for the datasets that do not assemble well, while `UniK` can alter to sequential method and avoid being slow in the following iterations. (3) By applying an auto-tuning in predicting the optimal configuration, `UTune` achieves the best performance across all datasets. The performance gap is even larger on low-dimensional datasets such as NYC (up to 389 times speedup), where `UTune` rightly predicts the configuration and improves the pruning percentage over `SEQU`.

[6]These two algorithms also compose the BDT in Figure 5, where `Yinyang` is same as `Hame` by setting $t = 1$ when $k < 50$.

Table 5: Evaluation of knob configuration in terms of MRR prediction accuracy.

Accuracy	BDT	**Basic features**					**+ Tree-features**					**+ Leaf-features**				
		DT	RF	SVM	kNN	RC	DT	RF	SVM	kNN	RC	DT	RF	SVM	kNN	RC
Bound@MRR	0.41	0.70	0.68	0.62	0.63	0.57	0.67	0.70	0.64	0.63	0.59	0.69	0.68	0.63	0.63	0.60
Index@MRR	0.37	0.80	0.82	0.84	0.74	0.68	0.83	0.77	0.83	0.74	0.70	0.74	0.77	0.83	0.74	0.74
S-Bound@MRR	0.42	0.84	0.83	0.81	0.82	0.74	0.86	0.87	0.81	0.82	0.74	0.89	0.87	0.88	0.88	0.80
S-Index@MRR	**0.43**	0.89	0.87	0.86	0.83	0.83	0.91	0.90	0.87	0.83	0.85	**0.92**	0.92	0.92	0.86	0.84

Table 6: Overall speedup over the running time (second) of Lloyd's algorithm (the gray column) on various datasets.

Data	$k = 10$					$k = 100$					$k = 1000$				
	Lloyd	×Speedup				Lloyd	×Speedup				Lloyd	×Speedup			
		SEQU	INDE	UniK	UTune		SEQU	INDE	UniK	UTune		SEQU	INDE	UniK	UTune
Cross	262	1.64 71%	1.76 67%	1.36 78%	1.76 84%	1463	2.83 86%	2.16 59%	**3.24** 90%	4.70 90%	13530	3.31 90%	1.81 46%	4.04 94%	**7.73** 93%
Conf	2.45	1.32 68%	1.30 61%	1.30 74%	1.32 68%	9.00	1.53 90%	1.63 25%	2.22 90%	2.50 90%	50.75	2.83 90%	1.47 6%	2.88 93%	2.93 93%
Covt	0.65	1.89 74%	2.18 72%	1.68 87%	2.18 72%	2.53	5.62 90%	1.43 23%	5.61 94%	5.67 93%	10.39	7.47 92%	1.04 4%	6.66 92%	7.47 92%
Euro	15.3	1.38 75%	1.42 67%	1.39 84%	1.48 35%	111	3.24 92%	2.53 45%	3.79 90%	4.03 95%	381.9	2.65 94%	0.63 11%	3.12 95%	3.13 95%
KeggD	0.45	2.93 83%	3.59 79%	**4.22** 84%	4.3 95%	1.16	2.61 92%	1.21 11%	4.00 71%	5.8 95%	8.50	6.58 93%	1.23 11%	7.01 89%	**7.57** 95%
Kegg	0.49	1.98 78%	2.83 83%	2.40 94%	2.83 83%	2.49	4.69 93%	1.79 31%	**5.87** 95%	6.15 96%	18.64	6.67 93%	0.94 51%	6.52 95%	6.67 93%
NYC	15.3	1.39 84%	389 99%	31.4 99%	**389** 99%	75.6	4.19 94%	153 99%	55.6 99%	153 99%	229.8	1.69 93%	11.05 93%	7.53 95%	13.3 96%
Skin	0.56	1.30 79%	2.54 87%	2.40 88%	2.54 87%	2.92	2.35 92%	2.60 56%	4.09 96%	4.13 96%	21.41	2.70 93%	1.38 27%	3.28 94%	3.54 95%
Power	6.38	1.43 78%	0.77 53%	0.87 82%	1.43 78%	32.9	2.39 91%	1.02 18%	2.53 93%	2.60 91%	223.9	2.17 92%	0.96 2%	2.26 92%	2.5 92%
Road	6.02	1.36 84%	8.64 96%	8.19 98%	8.64 96%	21.2	2.57 93%	3.68 69%	4.60 93%	4.93 97%	132.8	2.40 94%	1.58 27%	2.69 93%	2.87 95%
Census	11.9	1.31 62%	0.82 26%	1.14 67%	1.55 69%	94.7	3.65 84%	1.14 15%	3.51 85%	3.67 84%	791	5.85 91%	1.05 9%	5.87 91%	5.87 91%
Mnist	7.44	1.13 1%	0.91 0%	0.98 1%	1.36 27%	67.3	1.21 17%	0.98 15%	1.22 18%	3.94 77%	709	1.69 37%	1.04 2%	1.54 38%	5.13 83%
Spam	0.12	1.13 79%	1.42 22%	1.15 78%	1.62 80%	0.69	5.80 91%	2.12 29%	12.59 94%	12.59 94%	4.87	4.35 79%	2.87 60%	8.87 90%	8.87 90%
Shuttle	0.20	3.65 71%	0.72 0%	0.57 71%	3.65 71%	1.15	5.62 91%	3.67 9%	5.47 91%	6.53 90%	4.61	4.85 88%	1.94 9%	5.17 92%	5.17 92%
MSD	8.93	1.17 41%	0.72 0%	0.92 56%	1.17 31%	21.2	2.04 58%	1.21 8%	2.17 71%	2.17 71%	592	2.33 69%	1.17 9%	2.57 79%	2.57 79%

To investigate whether our adopted ML models can generalize well to datasets that have not been seen during training, we further test `UTune` on three new datasets: Spam [7], Shuttle [8], and MSD [17]. The results are consistent with our earlier finding, i.e., `UTune` beats other methods in most times, and is always in the leaderboard. This confirms a good generalization capability of our ML model.

7.3.3 Summary of Lessons Learned

Through an evaluation on multiple learning models for auto-configuration, we further learn:

- Automatic algorithm selection for fast k-means clustering is feasible by using a meta-learning model. Through `UTune`, we can predict an algorithm configuration that leads to better performance than state-of-the-art methods which we have reported in last section, also including our `UniK`. Moreover, the learning cost is also low if using the offline evaluation logs.
- There are several ways to improve the precision of models. Firstly, our selective running based on evaluation can save much time to generate more logs for model training. Secondly, building indexes can provide more features that help improve the prediction accuracy.
- By using very basic machine learning models without fine-tuning, our algorithm selection approach already achieves 92% prediction accuracy. Note that finding the best choice of learning models is orthogonal to this work, though.

8. CONCLUSIONS

We evaluated existing accelerating algorithms for fast k-means, in a unified framework which enables a fine-grained performance breakdown of existing methods. To auto-configure the pipeline for the optimal performance, we extracted features from our built index and selectively evaluated various methods, then trained a meta-model which predicts the optimal configuration effectively. Our experiments on real datasets showed that our unified framework with auto-tuning can accelerate k-means for various datasets effectively. In the future, we will work on learning with a rank-aware loss function like MRR, i.e., designing specific machine learning models to further improve precision. We will also extract more features from the dataset and index, and discover new promising configurations and add them into `UniK`. More discussions on these future opportunities can be found in our *supplementary material* (see Section A.5).

9. REFERENCES

[1] BigCross Dataset. https://s3.amazonaws.com/h2o-training/clustering/BigCross.data.gz.

[2] ConfLongDemo-JSI — Machine Learning Data. http://networkrepository.com/ConfLongDemo-JSI.php.

[3] Europediff dataset. http://cs.joensuu.fi/sipu/datasets/europediff.txt.

[4] K-Means clustering in Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[5] NYC Taxi Limousine Commission - Trip Record Data). http://www.nyc.gov/html/tlc/html/technology/data.shtml.

[6] Repository of UniK. https://github.com/tgbnhy/fast-kmeans.

[7] Spambase Data Set. https://archive.ics.uci.edu/ml/datasets/spambase.

[8] Statlog (Shuttle) Data Set. https://archive.ics.uci.edu/ml/datasets/Statlog+%28Shuttle%29.

[9] THE MNIST DATABASE. http://yann.lecun.com/exdb/mnist/.

[10] UCI Machine Learning Repository: 3D Road Network (North Jutland, Denmark) Data Set. https://archive.ics.uci.edu/ml/datasets/3D+Road+Network+(North+Jutland,+Denmark).

[11] UCI Machine Learning Repository: Covertype Data Set. https://archive.ics.uci.edu/ml/datasets/covertype.

[12] UCI Machine Learning Repository: Individual household electric power consumption Data Set. https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption.

[13] UCI Machine Learning Repository: KEGG Metabolic Reaction Network (Undirected) Data Set. https://archive.ics.uci.edu/ml/datasets/KEGG+Metabolic+Reaction+Network+(Undirected).

[14] UCI Machine Learning Repository: KEGG Metabolic Relation Network (Directed) Data Set. https://archive.ics.uci.edu/ml/datasets/KEGG+Metabolic+Relation+Network+(Directed).

[15] UCI Machine Learning Repository: Skin Segmentation Data Set. https://archive.ics.uci.edu/ml/datasets/skin+segmentation.

[16] UCI Machine Learning Repository: US Census Data (1990) Data Set. https://archive.ics.uci.edu/ml/machine-learning-databases/census1990-mld/.

[17] YearPredictionMSD Data Set. https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD.

[18] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB Journal*, 24(4):557–581, 2015.

[19] A. Aggarwal, A. Deshpande, and R. Kannan. Adaptive Sampling for k-Means Clustering. In *APPROX and RANDOM*, pages 15–28, 2009.

[20] D. Arthur and S. Vassilvitskii. K-Means++: The advantages of careful seeding. In *SODA*, pages 1027–1025, 2007.

[21] K. Bache and M. Lichman. UCI Machine Learning Repository, 2013.

[22] B. Bahmani, R. Kumar, and S. Vassilvitskii. Scalable K-Means++. *PVLDB*, 5(7):622–633, 2012.

[23] 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.

[24] J. L. Bentley. Multidimensional Binary SearchTrees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.

[25] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[26] T. Bottesch, K. Markus, M. Kaechele, and U. N. I. Ulm. Speeding up k -means by approximating Euclidean distances via block vectors. In *ICML*, pages 2578–2586, 2016.

[27] A. Broder, L. Garcia-Pueyo, V. Josifovski, S. Vassilvitskii, and S. Venkatesan. Scalable K-Means by ranked retrieval. In *WSDM*, pages 233–242, 2014.

[28] M. E. Celebi, H. A. Kingravi, and P. A. Vela. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Systems With Applications*, 40:200–210, 2013.

[29] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[30] B. Chazelle. An Optimal Convex Hull Algorithm in Any Fixed Dimension. Technical report, 1993.

[31] L. Chen, Y. Gao, B. Zheng, C. S. Jensen, H. Yang, and K. Yang. Pivot-based metric indexing. *PVLDB*, 10(10):1058–1069, 2017.

[32] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[33] N. Craswell. Mean Reciprocal Rank. *Encyclopedia of database systems*, 2009.

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

[35] 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.

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

[37] J. Drake and G. Hamerly. Accelerated k-means with adaptive distance bounds. In *OPT*, pages 2–5, 2012.

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

[39] P. et Al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] K. Fukunaga and P. M. Narendra. A Branch and Bound Algorithm for Computing k-Nearest Neighbors. *IEEE Trans. Computers*, 24(7):750–753, 1975.

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

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

[43] O. Hassanzadeh, F. Chiang, H. C. Lee, and R. J. Miller. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.

[44] A. K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[45] T. Kanungo, S. Member, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, A. Y. Wu, and S. Member. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[46] S. B. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. *Informatica*, 31:249–268, 2007.

[47] W. Kwedlo. Two Modifications of Yinyang K-means Algorithm. In *ICAISC*, pages 94–103, 2017.

[48] G. Li, X. Zhou, S. Li, and B. Gao. QTune: A Query-Aware database tuning system with deep reinforcement learning. In *PVLDB*, volume 12, pages 2118–2130, 2019.

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

[50] A. Lulli, M. Dell'Amico, P. Michiardi, and L. Ricci. Ng-Dbscan: Scalable Density-Based Clustering for Arbitrary Data. *PVLDB*, 10(3):157–168, 2016.

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

[52] E. Müller, S. Günnemann, I. Assent, and T. Seidl. Evaluating clustering in subspace projections of high dimensional data. *PVLDB*, 2(1):1270–1281, 2009.

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

[54] J. Newling and F. Fleuret. Nested Mini-Batch K-Means. In *NIPS*, pages 1352–1360, 2016.

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

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

[57] D. Pelleg and A. Moore. Accelerating exact k -means algorithms with geometric reasoning. In *KDD*, pages 277–281, New York, New York, USA, 1999.

[58] S. J. Phillips. Acceleration of k-means and related clustering algorithms. In *ALENEX*, pages 166–177, 2002.

[59] A. Rivolli, L. P. F. Garcia, C. Soares, J. Vanschoren, and A. C. P. L. F. de Carvalho. Towards Reproducible Empirical Research in Meta-learning. Technical report, 2018.

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

[61] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[62] M. Shindler, A. Wong, and A. Meyerson. Fast and Accurate k-means For Large Datasets. In *NIPS*, pages 2375–2383, 2011.

[63] J. K. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[64] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*, pages 1009–1024, 2017.

[65] J. Vanschoren. Meta-Learning: A Survey. Technical report, 2018.

[66] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, and X. Qin. Fast large-scale trajectory clustering. *PVLDB*, 13(1):29–42, 2019.

[67] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[68] X. Wu, V. Kumar, Q. J. Ross, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z. H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[69] M. Zechner and M. Granitzer. Accelerating K-Means on the Graphics Processor via CUDA. In *INTENSIVE*, pages 7–15, 2009.

[70] W. Zhao, H. Ma, and Q. He. Parallel K -Means Clustering Based on MapReduce. In *CloudCom*, pages 674–679, 2009.

Table 7: A summary of common notations.

Symbol	Description
D, d	The dataset, and its dimensionality
x_i	A data point in D
c_j	The centroid of cluster S_j
$lb(i,j)$	The lower bound of x_i to cluster c_j
$lb(i)$	The Lower bound of x_i to its second nearest cluster
$ub(i)$	The Upper bound of x_i to its nearest cluster
$a(i)$, $a^{'}(i)$	The indices of x_i's new and previous cluster
p	The pivot point of node N
$\|c_j\|$	The L2-norm of centroid c_j
$\delta(j)$	The centroid drift of c_j

APPENDIX

A. SUPPLEMENTARY MATERIALS

Algorithm 2: Selective Running of Knob Configurations

```
   Output: two ground truth files: g1 and g2
1  for every test dataset D do
2      F ← MetaFeatureExtraction(D, k, T);
3      Run partial sequential methods;
4      write(F, identifier of the optimal knob m_o, g1);
5      Test the index method m_i with Ball-tree;
6      if m_i is slower than m_o then
7          write(F, not using index 1, g2);
8      else
9          Test index-single and index-multiple with m_o;
10         if index method is the fastest then
11             write(F, point traveseal 2, g2);
12         else
13             if index-single is the fastest then
14                 write(F, single traveseal 3, g2);
15             else
16                 write(F, multiple traveseal 4, g2);
17 return g1 and g2;
```

A.1 Common Notations

Table 7 lists the common notations used in this paper.

A.2 Space Analysis of Index Methods

Recall Definition 1, building an extra data structure will increase the space overhead, the size of a node N is comparably small – each node is composed of two vectors (p, sv), four floats (r, ψ, num, h), and two pointers to child nodes (left and right) or a set of points in leaf nodes (the number is limited as the capacity f). Thus, we estimate the space of leaf nodes as $2d+4+f$, and internal nodes as $2d+6$. Then the overall space cost (number of floats) of all the nodes will be $\frac{n}{f}\cdot(2d+4+f)+q\cdot(2d+6) = n+\frac{n}{f}\cdot(2d+4)+q\cdot(2d+6)$, where $\frac{n}{f}$ and $q=\frac{n}{f}\cdot(1-2^{1-log_2\frac{n}{f}})$ are the numbers of leaf nodes and internal nodes.

Such an estimation is based on an ideal case where each leaf node has f points and the index is a balanced Ball-tree with a height $log_2\frac{n}{f}$, and we skip the summation for geometric sequence here. In Figure 10 and 18, we can find that the footprint of Ball-tree is comparable with those space-saving sequential methods in the most datasets. For the datasets that have high footprint, most of them are high-dimension, which means that they will have more leaf nodes that cover less points than those datasets with lower dimensions. Note that one advantage of Ball-tree is that its footprint is fixed once it is built, and will not change with the increasing of k, while most sequential methods will need much more space.

A.3 More Experiment Results

Figure 16 to 19 show a detailed comparison between sequential methods when setting $k=10$, which mainly enhances Section 7.2.2. We can observe similar trends with $k=100$, and the main difference is that smaller k has fewer data accesses and lower pruning ratio. Table 9 and 10 present the time of assignment and refinement which partially decompose the results the overall speedup in Table 6.

A.4 Reproducibility

Our code is available at Github:

`https://github.com/tgbnhy/fast-kmeans`

We show how to run our experiments with simple commands and operations in the description of the repository. Readers can download the datasets in Table 2 online by clicking the links shown there, and put into local folders, then compile the code using Maven (`http://maven.apache.org/`). Clustering tasks can be conducted locally by executing the given exemplar commands by terminals. All the algorithms will be run, and the results will be shown in the terminal to monitor the running progress. After all the algorithms end, detailed results on each comparison metric, such as speedup and the number of data accesses, will be written into the log files for further plotting. We also show readers how to interpret the evaluation results based on the terminal and logs files.

A.5 Future Opportunities

More Meta-Features. In our evaluations, we used multiple measures without extra costs. The main reason that we do not use other features that can be extracted using data profiling [18] or meta-feature extraction [59] is that, they need much time to extract, such as the model-based method by training decision trees. Since a higher precision will recommend a better algorithm and may greatly reduce the clustering time, it is promising to apply these features to improve the precision with fast meta-feature extraction.

New ML Models to be Adopted. We have tested multiple classical models to predict the optimal algorithm, while the loss function generally computes exact matches and does not consider the importance of ranked lists of algorithms. Hence, designing a specific machine learning model with a loss function like MRR, which we used to evaluate the accuracy, will be crucial to further improve the prediction accuracy. Furthermore, deep neural networks (DNN) have also shown significant successes in various classification tasks, and it will be interesting to design a specific network structure to input a sampled dataset as features directly to predict the optimal algorithm. It is worth noting that this is out of the scope of this work and our evaluation framework is orthogonal to the choice of ML model adopted in the learning part.

Discovering New Configuration Knobs. In this evaluation, we have compared all existing fast k-means clustering

Table 8: Evaluation of knob configuration: training time and prediction time.

Efficiency	BDT	**Basic features**					**+ Tree-features**					**+ Leaf-features**				
		DT	RF	SVM	kNN	RC	DT	RF	SVM	kNN	RC	DT	RF	SVM	kNN	RC
Training (ms)	-	1.63	577	3.74	1.73	4.40	1.89	573	4.35	1.39	4.54	2.53	598	5.42	1.43	4.36
Prediction (μs)	-	3.98	251	6.28	36.4	5.33	6.09	259	7.14	39	6.72	7.15	271	10.2	37.7	8.57
S-Training (ms)	-	1.98	660	29	1.85	4.90	2.81	764	53.3	2.33	4.55	4.75	897	54.3	1.87	4.68
S-Prediction (μs)	-	1.53	97	10.7	27.9	1.39	1.24	105	13.3	32.00	1.82	**1.35**	97.2	14.6	34.5	1.90

algorithms and `UniK` with our new settings. It is worth mentioning that the predicted algorithm is only the fastest among the group that we tested, and k-means can be further accelerated in the future. As shown in Algorithm 1, we have listed multiple knobs, and the knob configuration space Θ is large, which means our tested algorithms are only a tiny proportion. The discovery of new configurations based on Algorithm 1 will enable us to combine various optimizations that have never been tried and tested. Such new configurations will form new algorithms that can be potentially fast for a certain group of clustering tasks.

Applying ML to Other Database Algorithm Selection Problems. In the database field, efficiency evaluations of existing algorithms have been an essential guide for users due to the diversity of tasks and datasets. Even though several clear insights are given for choosing the right decision in the end of evaluations, they essentially form an elementary and inaccurate decision tree for referring. Auto-tuning based on the evaluation logs and ML models can interpret the superiority of algorithms in a more accurate way than humans, and can directly predict the optimal algorithm. Hence, it will be interesting to apply the auto-tuning methodology presented in this paper to those evaluation studies that have made a good comparison but still manually selects algorithms.

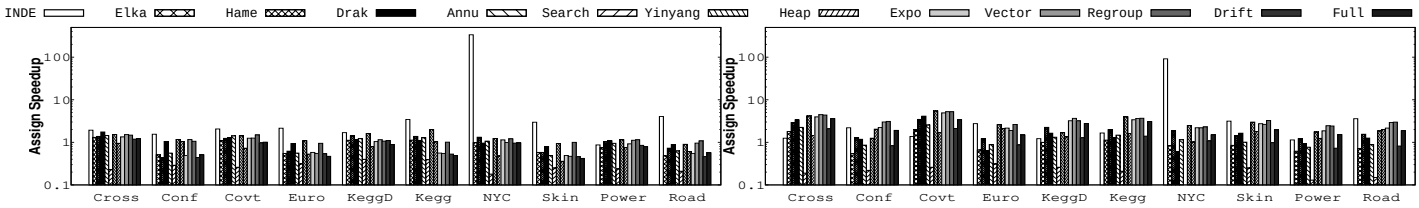


Figure 16: Overall assignment speedup in various datasets when setting k as 10 and 100, respectively.

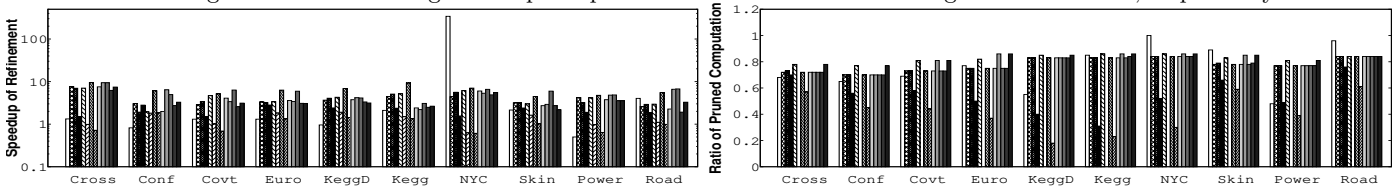


Figure 17: Refinement's speedup and distance calculation pruning ratio ($k = 10$).

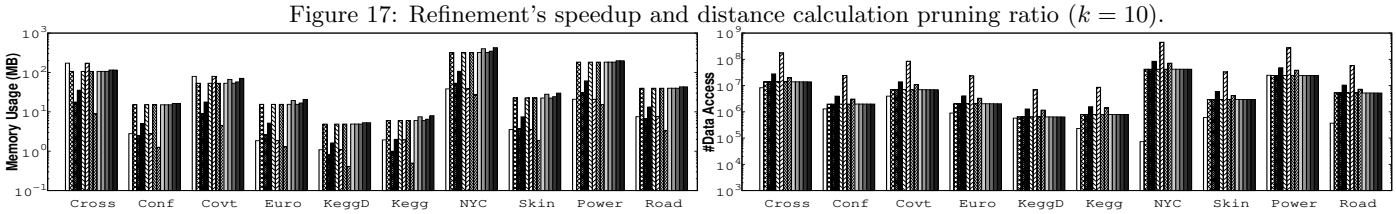


Figure 18: Statistics on the footprint of bound (index) and data accesses ($k = 10$).

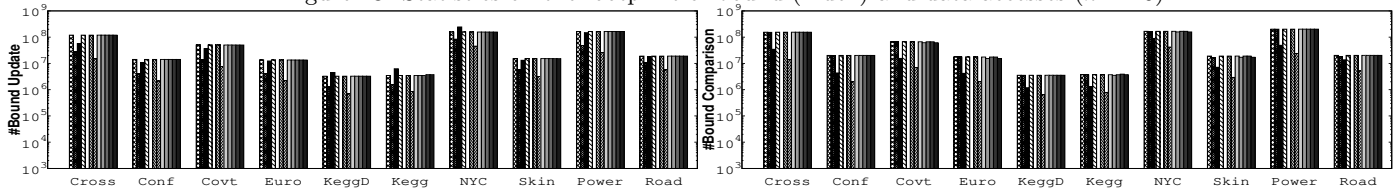


Figure 19: Statistics on the bound accesses and updates ($k = 10$).

Table 9: The assignment speedup over the running time (second) of Lloyd's algorithm (the gray column).

Data	$k = 10$					$k = 100$					$k = 1000$				
	Lloyd	×Speedup				Lloyd	×Speedup				Lloyd	×Speedup			
		SEQU	INDE	UniK	UTune		SEQU	INDE	UniK	UTune		SEQU	INDE	UniK	UTune
Cross	219.7	1.45	1.86	1.16	1.56	1381	2.73	2.20	3.10	4.54	13325	3.30	1.82	4.00	7.68
Conf	1.73	1.09	1.44	0.62	1.09	7.42	1.37	1.69	2	2.26	48.17	2.76	1.46	2.79	2.86
Covt	0.48	1.72	2.35	1.53	2.35	2.09	5.55	1.44	5.50	5.60	10.13	7.46	1.04	6.65	7.46
Euro	12.72	1.15	1.46	0.32	1.31	103.82	2.94	2.61	1.67	3.64	379.46	2.56	0.61	0.50	2.99
KeggD	0.35	2.57	4.11	2.69	3.74	1.04	2.47	1.21	2.00	5.52	8.29	6.56	1.23	2.06	7.51
Kegg	0.37	1.69	2.89	1.2	2.89	2.25	2.51	3.38	5.38	5.80	18.23	6.69	0.94	5.81	6.69
NYC	12.26	1.16	397.3	25.1	397.3	75.6	3.83	158.8	50.10	158.8	221.7	1.65	10.9	7.36	13.0
Skin	0.44	1.03	2.68	2.10	2.68	2.63	4.13	2.73	2.39	3.8	20.86	2.65	1.38	2.53	3.45
Power	5.31	1.26	0.90	0.77	1.26	29.79	2.26	1.06	2.38	2.46	220.19	2.15	0.97	2.24	2.4
Road	4.28	1.09	9.01	2.57	9.01	17.54	2.38	3.92	3.11	4.52	126.84	2.32	1.58	1.90	2.75

Table 10: The refinement speedup over the running time (second) of Lloyd's algorithm (the gray column).

Data	$k = 10$					$k = 100$					$k = 1000$				
	Lloyd	×Speedup				Lloyd	×Speedup				Lloyd	×Speedup			
		SEQU	INDE	UniK	UTune		SEQU	INDE	UniK	UTune		SEQU	INDE	UniK	UTune
Cross	42.6	8.03	1.34	5.56	8.74	83.0	7.34	1.68	12.8	16.6	205	4.77	1.26	11.3	18.8
Conf	0.72	6.60	6.60	3.54	6.60	1.58	1.37	1.27	11.5	10.1	2.58	7.59	1.55	11.1	8.40
Covt	0.17	7.46	1.39	7.54	1.39	0.44	1.17	1.17	12.76	8.90	0.25	8.13	0.97	9.85	8.13
Euro	2.62	7.90	1.25	2.69	3.19	6.79	11.41	2.08	3.49	16.9	2.44	6.87	1.65	2.48	15.3
KeggD	0.10	9.95	2.28	10.7	17.1	0.12	8.21	1.16	2.01	15.2	0.21	8.08	1.63	1.97	15.6
Kegg	0.12	10.2	2.57	6.76	2.57	0.25	15.8	1.78	34.6	22.0	0.41	5.15	0.83	15.4	5.15
NYC	3.06	6.87	361	348	361	8.76	14.2	122	162.7	122	8.05	4.61	15.3	21.1	24.2
Skin	0.12	6.79	2.19	8.90	2.19	0.29	8.81	1.88	9.70	15.2	0.55	8.10	1.34	8.71	13.4
Power	1.07	5.60	0.44	3.52	5.60	3.12	6.12	0.74	7.43	6.31	3.71	3.92	0.55	6.00	6.4
Road	1.74	5.88	7.65	13.0	7.65	3.63	6.58	2.50	8.76	16.2	5.96	7.48	1.61	15.2	15.3